Online Exam: Advanced Functional Programming TDA342/DIT260

Published Edit ⋮

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

Preliminaries

Below you find the information that it 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 checking 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)

```
Topic: Exam questions - AFP - round I
Time: Mar 21, 2020 09:30 AM Stockholm
```

```
Join from PC, Mac, Linux, iOS or Android: https://chalmers.zoom.us/j/203762828?pwd=cTl0YVp4b2NpSW42WXBBUjF4K2UwZz09
    Password: 005054

Or Skype for Business (Lync):
    https://chalmers.zoom.us/skype/203762828

Or an H.323/SIP room system:
    H.323: 109.105.112.236 or 109.105.112.235
    Meeting ID: 203 762 828
    Password: 005054

    SIP: 203762828@109.105.112.236 or 203762828@109.105.112.235
    Password: 005054

Or Skype for Business (Lync):
    https://chalmers.zoom.us/skype/203762828
```

Round 2 (11:30)

```
Topic: Exam question - AFP - round 2
Time: Mar 21, 2020 11:30 AM Stockholm

Join from PC, Mac, Linux, iOS or Android: https://chalmers.zoom.us/j/150806314?pwd=U2JvQmpMODlFNXJaQzBMNktYcmhRQT09
    Password: 074327

Or Skype for Business (Lync):
    https://chalmers.zoom.us/skype/150806314

Or an H.323/SIP room system:
    H.323: 109.105.112.236 or 109.105.112.235
    Meeting ID: 150 806 314
    Password: 074327

    SIP: 150806314@109.105.112.236 or 150806314@109.105.112.235
    Password: 074327

Or Skype for Business (Lync):
    https://chalmers.zoom.us/skype/150806314
```

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 (22 points)

File `src/Exam/Ex1.hs`

This exercise is based on the module *Type-based modeling* of the course. If you do not remember it, do not worry, we will briefly recap it here. In any case, the content of the lecture is available.

We consider a simple (non-monadic) DSL for integers and booleans expressions. The DSL gets implemented in a deep-embedded manner.

```
data Expr where
  -- Constructors
  LitI   :: Int  -> Expr
  LitB   :: Bool -> Expr
  -- Combinators
  (:+:)  :: Expr -> Expr -> Expr
  (:==:) :: Expr -> Expr -> Expr
  If     :: Expr -> Expr -> Expr -> Expr
```

and the run function, which is the most interesting piece of the code:

```
eval :: Expr -> Value
```

Since expressions can be reduced to integers or booleans, an element of type `Value` is either an integer or a boolean.

```
data Value = VInt Int | VBool Bool
```

There are some aspects of this implementation that we need to remark.

1. This DSL allows to evaluate ill-typed expressions, so calling `eval` can fail miserably but we do not worry about that here.

   ```
   > eval (LitB True :==: LitI 42)
   *** Exception: Crash!
   ```

2. You can inspect the *type of values* as follows.

   ```
   > showTypeOfVal (eval (LitI 42))
   "Int"
   > showTypeOfVal (eval (LitB True :==: LitB False))
   "Bool"
   ```

3. You can also nicely print objects of the EDSL as follows.

   ```
   > show  $ If (LitB False) (LitI 2) (LitI 2 :+: LitI 1736)
   "if False then 2 else 2 + 1736"
   ```

4. You can *randomly* generate terms of type `Expr` as follows.

   ```
   > import Test.QuickCheck
   > generate (arbitrary :: Gen Expr)
   if (if True + True then if -22 then False else False else -8 + -20) == True
   then 16 else -13 == (False == -6) + 4 + (if False then -19 else True)
   ```

The output that you get when running `generate (arbitrary :: Gen Expr)` surely varies due to the randomness used by QuickCheck.

In this exercise, your goal is to extend the EDSL to work with lists. So, we extend the EDSL with two new constructors (`Nil` and `Cons`) as follows.

```
data Expr where
  LitI   :: Int  -> Expr
  LitB   :: Bool -> Expr
  (:+:)  :: Expr -> Expr -> Expr
  (:==:) :: Expr -> Expr -> Expr
  If     :: Expr -> Expr -> Expr -> Expr
  Nil    :: Expr                         -- new
  Cons   :: Expr -> Expr -> Expr         -- new
```

To help you out along the way, we have include many test cases (definitions `test1`, `test2`, .., `test11`).

Task 1.1 (5 pts)

Extend the pretty printing of the language, i.e., definition of `showsPrec`, to consider lists. For instance, your solution should produce the following outputs.

```
> let test1 = Nil in show test1  -- def. test1
"[]"
> let test2 = Cons (LitI 1) (Cons (LitB True) Nil) in show test2
"[1,True]"
> let test3 = Cons (LitI 1) (Cons (LitB True) (Cons (LitI 42) Nil)) in show test3
"[1,True,42]"
> show test6 -- see def. test6
"[[1,True],[1,True,42]]"
```

Make sure that your function crashes when it is not a list.

```
> show $ Cons (LitI 42) (LitB True)
"[*** Exception: The list got broken
```

Task 1.2 (5 pts)

Extend `arbExpr :: Int -> Gen Expr` to generate also lists.

- Make sure that you do not generate empty lists.
- Keep in mind that your extension to `arbExpr` very often will generate ill-typed terms like `Cons (LitI 42) (LitB True)` and it is Ok!

```
> generate (arbitrary :: Gen Expr)
[-29]
> generate (arbitrary :: Gen Expr)
(if if True then -18 else [(if if [*** Exception: The list got broken
```

Do not worry about these two points. We will clarify them later on.

Task 1.3 (10 pts)

Extend the definition of `Value` and `eval` to consider lists.

- Comparison between lists is `True` **only** when both lists have the same length and the elements are equal point-wise.

```
> show test5
"[1,True] == [1,True,42]"
> eval test5
VBool False
> show test7
"[[1,True],[1,True,42]] == [[1,True],[1,True,42]]"
> eval test7
VBool True
```

- The addition of lists is performed point-wise and *only* when both list contain numbers and have the same size.

```
> show test8
"[1,2]"
```

```
> show test9
"[100,1]"
> show test10
"[1,2] + [100,1]"
> eval test10 -- it should result in the list [101,3]
...
> show test11
"[[1,2]] + [[100,1]]"
> eval test11 -- it fails since it is adding a list of lists
*** Exception: Problems adding lists!
```

Task 1.4 (2 pts)

Extend function `showTypeOfEval` to display `[Value]` when dealing with expressions which evaluate to lists.

```
> show test2
"[1,True]"
> showTypeOfVal (eval test2)
"[Value]"
> show test6
"[[1,True],[1,True,42]]"
> showTypeOfVal (eval test6)
"[Value]"
```

Exercise 2 (38 points)

File `src/Exam/Ex2.hs`

To avoid evaluating many of the ill-form expressions in the DSL from Exercise 2, e.g., `Ex1.eval (Ex1.LitB True :==: Ex1.LitI 42)`, we saw during the **lectures** how to use `GADTs` and implement a typed DSL. We briefly recapitulate what we did in that lecture here. From now on, all definitions related to the previous exercises are referred in a qualified form as `Ex1`.

1. We introduced a GADT of the form `Expr t`, where t is a Haksell type which indicates the value that the expression denotes.

```
data Expr t where
  LitI   :: Int -> Expr Int
  LitB   :: Bool -> Expr Bool
  (:+:)  :: Expr Int -> Expr Int -> Expr Int
  (:==:) :: Eq t => Expr t -> Expr t -> Expr Bool
  If     :: Expr Bool -> Expr t -> Expr t -> Expr t
```

For instance, now the expression `LitB True :==: LitI 42` is not well-typed for Haskell.

```
>:t LitB True :==: LitI 42
error:
    • Couldn't match type ‘Int’ with ‘Bool’
      Expected type: Expr Bool
        Actual type: Expr Int
```

2. The DSL's run function is

```
eval :: Expr t -> t
```

which knows the resulting type of the evaluation, i.e., `t`, so it is not necessary to keep *typing tags* around during evaluation (like `Ex1.VInt` and `Ex1.VBool` in the previous exercise).

3. We show that this DSL (based on `Expr t`) is more restrictive than the one in the previous exercise (based on `Ex1.Expr`) by removing the typing information. The function `forget` takes a term in `Expr t` and builds one in `Ex1.Expr`.

```
forget :: Expr t -> Ex1.Expr
```

4. We use some advanced Haskell's type-system to check if an expression of type `Ex1.Expr` has a valid type by rewriting it into one of type `Expr t` for some `t`, which is done by function `infer`.

```
infer :: Ex1.Expr -> Maybe TypedExpr
```

5. We implement a function `evalT` as a safe wrapper on `Ex1.eval`:

    ```
    evalT :: Ex1.Expr -> Maybe Ex1.Value
    ```

    This function only calls `Ex1.eval` if the given argument is well-typed.

6. There is a property written in QuickCheck that says that the result produced by `evalT` should match that of `Ex1.eval`. In that way, we have not changed the semantics of our DSL when we moved to a more *typed discipline*.

    ```
    prop_eval :: Ex1.Expr -> Property
    ```

    You can check the property as follows:

    ```
    > main
    +++ OK, passed 100 tests:
    51% Bool
    49% Int
    ```

    which indicates that the test has passed, where QuickCheck generated 51% and 49% of well-typed Boolean and integer expressions `Ex1.Expr`, respectively.

Task 2.1 (5 pts)

The good aspects of using `GADTs` in this way is that it allows to check for the well-form of lists, lists of lists, lists of lists of lists, etc. Extend the definition of `Expr t` to consider constructor `Nil` and `Cons`

```
data Expr t where
  LitI   :: Int -> Expr Int
  LitB   :: Bool -> Expr Bool
  (:+:)  :: Expr Int -> Expr Int -> Expr Int
  (:==:) :: Eq t => Expr t -> Expr t -> Expr Bool
```

```
  If     :: Expr Bool -> Expr t -> Expr t -> Expr t
  Nil    :: Expr t  -- change this
  Cons   :: Expr t  -- change this
```

and make sure that your extension allows to build the following examples:

```
ok1 = Cons (LitI 1) (Cons (LitI 2) Nil)                        -- [1,2]
ok2 = Cons (Cons (LitI 1) Nil) (Cons (Cons (LitI 2) Nil) Nil) -- [[1],[2]]
```

and rejects these ones:

```
bad1 = Cons (LitI 1) (Cons (LitB True) Nil)         -- [1,True]
bad2 = Cons (Cons (LitI 1) Nil) (Cons (LitI 2) Nil) -- [[1],2]
```

Task 2.2 (2 pts)

Extend the definition of `forget` to consider the cases for `Nil` and `Cons`.

Task 2.3 (3 pts)

Extend `eval` with the cases for `Nil` and `Cons`.

At this point, we ignore how to extend `(:+:)` and `(:==:)` to consider lists.

Task 2.4 (5 pts)

Extend `eval` to consider the addition and comparison of lists. The addition and comparison of lists should have the same semantics as the previous exercise. For instance, we only consider addition of lists of numbers when both lists have the same length.

Task 2.5 (8 pt)

Function `infer` relies on the GADT `Type t`, which indicates the type of expressions:

```
data Type t where
  TInt  :: Type Int
  TBool :: Type Bool
```

Importantly, function `(=?=)` implements equality of types by returning `Just Refl` for those cases where `s` and `t` are the same time:

```
(=?=) :: Type s -> Type t -> Maybe (Equal s t)
```

Extend the definition of `Type t`, `Show (Type t)`, and `(=?=)` to consider lists.

Task 2.6 (5 pts)

In this task, you should extend the definition of `infer` to work with lists. Before doing that, we need to clarify a problem inherent to the approach we have followed in our DSL.

What should be the inferred type of the empty list ?

```
infer Ex1.Nil
```

Well, in principle, `Ex1.Nil` could be a list of integer, booleans, list of list of integer, list of list of boolean, and so on. *So, we do not know the type of the empty list by just looking at the empty list*. We need either:

1. more context, i.e., to see how the empty list gets used,
2. support for type variables, or
3. support for polymorphism.

We are **not** going to pursue option 2 and 3. Instead, we stick to option 1. An easy fix is to disallow typing empty lists and only type lists of size, *at least* one. So, `infer` will raise an error when trying to infer the type of just `Nil`:

```
infer e =
  ...
  Ex1.Nil -> Nothing
```

However, `infer` should be defined for lists of size, at least, one. That is why in Exercise 1, Task 1.2, you were asked to *not* generate empty lists.

> In this point, do not modify the code for the case `r1 Ex1.:+: r2`. This means that you will not be able to type-check terms where you are adding lists.

Task 2.7 (5 pt)

Extend `evalT` to handle lists.

After you have done that, you can test that you have not changed the semantics of evaluating lists w.r.t to Exercise 1 by using QuickCheck:

```
+++ OK, passed 100 tests:
48% Bool
42% Int
10% [Value]
```

As you see above, you should not expect a high percentage of test cases (often less than 10%) related to lists. The reason being that it is hard for QuickCheck to generate well-typed lists with such a simple generator as the one in Exercise 1.

Task 2.8 (5 pt)

In this task, you should extend `infer` to work for addition of lists, e.g., it should work as follow in the next example:

```
> infer $ (Ex1.Cons (Ex1.LitI 2) Ex1.Nil) Ex1.:+: (Ex1.Cons (Ex1.LitI 10) Ex1.Nil)
Just [2] + [10] :: [Int]
```

Points 60

Submitting a file upload

Due	For	Available from	Until
-	Everyone	-	-

+ Rubric