Chalmers | Göteborgs Universitet
Alejandro Russo, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Saturday, March 19, 2022, 8:30.

(including example solutions to programming problems)

Robert Krook, tel. 0707 772 409
(Examiner 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): (A Monad for non-determinism)

A non-determinism program can exhibit different behaviors on different runs even for the same input. Monads can be used to model *all possible results from a non-deterministic computation*, and in this exam, we will see how.

When we have a computation $m :: ND\ a$, we should think about it as a computation that might return many different results of type a due to some source of non-determinism. The source of non-determinism or how non-determinism gets introduced in programs is not important here.

The computation $m \gg\!= f$ consists of executing m, taking all its possible results (of type a), and applying each of them to f and collecting all the possible results (of type b) of the yield computations.

For instance, the following program models all the possible outputs of adding two non-deterministic computations producing integers.

```
ndSum :: ND Int → ND Int → ND Int
ndSum m₁ m₂ = do n₁ ← m₁
                 n₂ ← m₂
                 return (n₁ + n₂)
```

Variables n_1 and n_2 can be seen as representing one of the many possible values that m_1 and m_2 might respectively produce due to the presence of non-determinism. Overall, *ndSum* performs the sums for *all* the possible combination of numbers being provided by m_1 and m_2.

To be more concrete, the following code models two programs that can produce different integers in a non-deterministic manner.

```
number₁ :: ND Int
number₁ = nonDeterminism [1, 42, 100]   -- possible numbers are 1,42, and 100.
number₂ :: ND Int
number₂ = nonDeterminism [2000, 30000, 50000]   -- possible numbers are 2000,30000,50000
```

The primitive *nonDeterminism xs* models a computation that produces values from the given list. If we apply *ndSum* to the programs above, we get the following output:

```
> ndSum number1 number2
LM {results = [2001,30001,50001,2042,30042,50042,2100,30100,50100]}
```

Observe that *ndSum* $number_1$ $number_2$ captures all the possible results of adding two numbers comming from $number_1$ and $number_2$, respectivelly.

One way to implement the monad *ND* is by simple considering that each computation returns a list of all the possible results.

```
newtype ND a = ND {results :: [a]}
```

a) Your task is to provide the definition for *return* and ($\gg\!=$) for the monad *ND*. *(10p)*

Solution:

```
instance Monad ND where
return x = ND [x]
```

$$\begin{aligned}&\mathit{ND}\ [\,] &&\gg\!= k = \mathit{ND}\ [\,]\\ &\mathit{ND}\ (x : xs) &&\gg\!= k = \mathit{ND}\ (\mathit{results}\ (k\ x) +\!\!+\ \mathit{results}\ (\mathit{ND}\ xs \gg\!= k))\end{aligned}$$

b) So far, we have been focusing on the *proper morphisms* of the ND monad (i.e., return and $(\gg\!=)$). In this exercise, we introduce the following non-proper morphism:

$$\mathit{option} :: \mathit{ND}\ a \to \mathit{ND}\ a \to \mathit{ND}\ a$$

which takes two non-deterministic computations and returns the computation which models both! This is an example of how it works:

```
> option number1 number2
LM {results = [1,42,100,2000,30000,50000]}
```

Your tasks is to provide an implementation of option, and based on that, give an implementation of $\mathit{nonDeterminism} :: [a] \to \mathit{ND}\ a$. *(5p)*

Solution:

$$\begin{aligned}&\mathit{option} :: \mathit{ND}\ a \to \mathit{ND}\ a \to \mathit{ND}\ a\\ &\mathit{option}\ (\mathit{ND}\ xs)\ (\mathit{ND}\ ys) = \mathit{ND}\ (xs +\!\!+\ ys)\\ &\mathit{nonDeterminism} :: [a] \to \mathit{ND}\ a\\ &\mathit{nonDeterminism} = \mathit{foldr1}\ (\lambda e\ lm \to \mathit{option}\ e\ lm) \circ (\mathit{map}\ \mathit{return})\end{aligned}$$

c) Give the instance definition for $\mathit{Functor}\ \mathit{ND}$ and $\mathit{Applicative}\ \mathit{ND}$. *Important*: your definitions must not use the monadic interface. *(5p)*

Solution:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ \mathit{ND}\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{ND}\ xs) = \mathit{ND}\ (\mathit{map}\ f\ xs)\\ &\textbf{instance}\ \mathit{Applicative}\ \mathit{ND}\ \textbf{where}\\ &\quad \mathit{pure}\ x = \mathit{ND}\ [x]\\ &\quad \mathit{ND}\ fs < * > \mathit{ND}\ xs = \mathit{ND}\ (fs < * > xs)\end{aligned}$$

Problem 2 (5pt): (Examples)

The non-deterministic monad enables to write simple and compact code.

a) For instance, the following code produces all possible permutation of a list.

```
perm :: [a] → ND [a]
perm []       = return []
perm (x : xs) = do ps ← perm xs
                   insert x ps
```

The line *ps* ← *perm xs* could be think as "*ps* is one of the possible permutations of *xs* (*perm xs*) selected in a non-deterministic manner", and *insert x ps* models that *x* is inserted into *ps* in a non-deterministic manner, i.e., in some position of the list *ps*.

The next invocation of *perms* shows how it works.

```
> perm [1,2,3]
LM {results = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]}
```

Observe that computing all the possible outputs of *perm* $[1, 2, 3]$ gives the actual permutations of the list. Your task is to implement the function *insert*:

```
insert :: a → [a] → ND [a]
```

The next example shows how *insert* works.

```
> insert 10 [1,2,3]
LM {results = [[10,1,2,3],[1,10,2,3],[1,2,10,3],[1,2,3,10]]}
```

Hint: do not forget the function *option* from the first exercise.

Solution:

```
insert x []       = return [x]
insert x (y : ys) = option (return (x : y : ys))
                           (do ls ← insert x ys
                               return (y : ls))
```

(5p)

Problem 3 (15pt): (Proof)

Prove that your definitions of *return* and ($\gg\!=$)) from the first exercise fullfils the monadic laws. Since *ND* is defined using a record, you might find useful to assume the following equations when doing the proofs.

$m = ND\ (results\ m)$
$results\ (ND\ xs) = xs$

a) Prove the left and right identity monadic laws:

-- Left identity
$return\ x \gg\!= f \equiv f\ x$
-- Right identity
$m \gg\!= return \equiv m$

(5p)

Solution:

-- Left identify
$return\ x \gg\!= k \equiv$
-- by def. return
$ND\ [x] \gg\!= k\ \ \equiv$
-- by def. bind
$ND\ (results\ (k\ x) +\!\!+\ results\ (ND\ [\,] \gg\!= k)) \equiv$
-- by def. LM []
$ND\ (resutls\ (k\ x) +\!\!+\ (results\ (ND\ [\,]))) \equiv$
-- by def. results
$ND\ (results\ (k\ x) +\!\!+\ [\,]) \equiv$
-- by def. concat
$ND\ (results\ (k\ x)) \equiv$
-- by def. accessor, i.e., LM (results m) = m
$(k\ x)$

-- Right identify
-- Proof: by induction on the number of results.
-- Base case: m = LM []
$ND\ [\,] \gg\!= return \equiv$
-- by def. ¿¿= on empty lists
$ND\ [\,]$

-- Inductive case: m = LM (x:xs)
$ND\ (x : xs) \gg\!= return \equiv$
-- by def. bind
$ND\ (results\ (return\ x) +\!\!+\ results\ (ND\ xs \gg\!= return)) \equiv$
-- by IH with LM xs ¿¿= return
$ND\ (results\ (return\ x) +\!\!+\ results\ (ND\ xs)) \equiv$
-- by def. results

$ND\ (results\ (return\ x) +\!\!+\ xs) \equiv$
-- by def. return
$ND\ (results\ (ND\ [x]) +\!\!+\ xs) \equiv$
-- by def. results
$ND\ ([x] +\!\!+\ xs)$
-- by def. concat and lists
$ND\ (x : xs)$

b) Prove the associative law:

$$(m \gg\!\!= f) \gg\!\!= g \equiv m \gg\!\!= (\lambda x \to f\ x \gg\!\!= g)$$

Solution:

-- Associativity
-- Proof: by induction on the number of results on m

-- Base case: m = LM []
$(ND\ [\,] \gg\!\!= f) \gg\!\!= g \equiv$
-- by def. bind on empty lists
$ND\ [\,] \gg\!\!= f \equiv$
-- by def. bind on empty lists
$ND\ [\,] \equiv$
-- by def. bind on empty lists
$ND\ [\,] \gg\!\!= (\lambda x \to f\ x \gg\!\!= g)$
-- by def. m
$m \gg\!\!= (\lambda x \to f\ x \gg\!\!= g)$

-- Inductive case: m = LM (x:xs)
$(ND\ (x : xs) \gg\!\!= f) \gg\!\!= g \equiv$
-- by def. bind
$ND\ (results\ (f\ x) +\!\!+\ results\ (ND\ xs \gg\!\!= f)) \gg\!\!= g \equiv$
-- by Aux. lemma (see below)
$ND\ (results\ (f\ x \gg\!\!= g) +\!\!+\ results\ ((ND\ xs \gg\!\!= f) \gg\!\!= g)) \equiv$
-- by IH
$ND\ (results\ (f\ x \gg\!\!= g) +\!\!+\ results\ (ND\ xs \gg\!\!= (\lambda y \to f\ y \gg\!\!= g))) \equiv$
-- by eta-expansion
$ND\ (results\ ((\lambda y \to f\ y)\ x \gg\!\!= g) +\!\!+\ results\ (ND\ xs \gg\!\!= (\lambda y \to f\ y \gg\!\!= g))) \equiv$
-- since y is a fresh variable not appearing into g, we can extend the scope of the lambda
$ND\ (results\ ((\lambda y \to f\ y \gg\!\!= g)\ x) +\!\!+\ results\ (ND\ xs \gg\!\!= (\lambda y \to f\ y \gg\!\!= g))) \equiv$
-- by def. of bind
$ND\ (x : xs) \gg\!\!= (\lambda y \to f\ y \gg\!\!= g)$

-- Aux. lemma:
$ND\ (results\ xs +\!\!+\ results\ ys) \gg\!\!= g \equiv ND\ (results\ (xs \gg\!\!= g) +\!\!+\ results\ (ys \gg\!\!= g))$

-- Prof: by induction on the number of results in xs.
-- Base case: xs = LM []

$ND\ (results\ (ND\ [\,])\ +\!\!+\ results\ ys) \gg\!= g \equiv$
  -- by def. results
$ND\ ([\,]\ +\!\!+\ results\ ys) \gg\!= g \equiv$
  -- by def. concat
$ND\ (results\ ys) \gg\!= g \equiv$
  -- by property of results
$ys \gg\!= g \equiv$
  -- by property of results
$ND\ (results\ (ys \gg\!= g))$
  -- by empty lists
$ND\ ([\,]\ +\!\!+\ results\ (ys \gg\!= g))$
  -- by def. results
$ND\ (results\ (ND\ [\,])\ +\!\!+\ results\ (ys \gg\!= g))$
  -- by def. bind on empty lists
$ND\ (results\ (ND\ [\,] \gg\!= g)\ +\!\!+\ results\ (ys \gg\!= g))$
  -- Inductive case: xs = LM (z:zs)
$ND\ (results\ (ND\ (z : zs))\ +\!\!+\ results\ ys) \gg\!= g \equiv$
  -- by def. results
$ND\ ((z : zs)\ +\!\!+\ results\ ys) \gg\!= g \equiv$
  -- by def. concat
$ND\ (z : (zs\ +\!\!+\ results\ ys)) \gg\!= g \equiv$
  -- by def. bind
$ND\ (results\ (g\ z)\ +\!\!+\ results\ (ND\ (zs\ +\!\!+\ results\ ys) \gg\!= g))$
  -- by def. results
$ND\ (results\ (g\ z)\ +\!\!+\ results\ (ND\ (results\ (ND\ zs)\ +\!\!+\ results\ ys) \gg\!= g))$
  -- by IH
$ND\ (results\ (g\ z)\ +\!\!+\ results\ (ND\ (results\ (ND\ zs \gg\!= g)\ +\!\!+\ results\ (ys \gg\!= g))))$
  -- by property results
$ND\ (results\ (g\ z)\ +\!\!+\ (results\ (ND\ zs \gg\!= g)\ +\!\!+\ results\ (ys \gg\!= g))) \equiv$
  -- by assoc. (++)
$ND\ ((results\ (g\ z)\ +\!\!+\ results\ (ND\ zs \gg\!= g))\ +\!\!+\ results\ (ys \gg\!= g)) \equiv$
  -- by def. results
$ND\ (results\ (ND\ (results\ (g\ z)\ +\!\!+\ results\ (ND\ zs \gg\!= g)))\ +\!\!+\ results\ (ys \gg\!= g)) \equiv$
  -- by def. bind
$ND\ (results\ (ND\ (z : zs) \gg\!= g)\ +\!\!+\ results\ (ys \gg\!= g))$

(10p)

Problem 4 (20pt): (Monad transformer)

We want to obtain a monad transformer for the *ND* monad. We will call it *NDT m*, where *m* is the underlying monad. *NDT m* has the following interface:

```
newtype NDT m a
return :: a → NDT m a
(≫=)   :: NDT m a → (a → NDT m b) → NDT m b
option :: NDT m a → NDT m a → NDT m a
```

a) Your tasks is to provide an implementation for *NDT m a*. *(5p)*

Solution:

```
newtype NDT m a = MkNDT {unmk :: m [a]}
```

b) Given the implementation in **a)**, provide a definition for *return* and (≫=). *(5p)*

Solution:

```
instance Monad m ⇒ Monad (NDT m) where
return a             = MkNDT (return [a])
(MkNDT m) ≫= k = MkNDT $ do results ← m
                                     let lmts = map (unmk ∘ k) results
                                     fmap concat $ sequence lmts
```

c) Give an implementation in **a)**, provide a definition for *option*. *(5p)*

Solution:

```
option (MkNDT m1) (MkNDT m2) = MkNDT (do ls1 ← m1
                                         ls2 ← m2
                                         return (ls1 ++ ls2))
```

d) Every monad transformer has a *run function*, here we will call it *runNDT*. Your task is to give its type and implementation. *(5p)*

```
runNDT :: NDT m a → ?
runNDT m = ?
```

Solution:

```
runMLT :: NDT m a → m [a]
runNDT (MkNDT m) = m
```