Chalmers | Göteborgs Universitet
Andreas Abel, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Saturday, March 16, 2024, 8:30 - 12:30, HB3.

- Examiner: Andreas Abel (+46-31-772-1731), visits 9:30 and 11:30.
- The maximum amount of points you can score on the exam: 60 points. The grade for the exam is as follows:

  Chalmers: **3**: $\geqslant 24$ points, **4**: $\geqslant 36$ points, **5**: $\geqslant 48$ points.
  GU: Godkänd $\geqslant 24$ points, väl godkänd $\geqslant 48$ points.
  PhD student: $\geqslant 36$ 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 exam sheet 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.
  - Exam review: Monday, 25 March 2024, 11-12am, EDIT 5128.

Problem 1 (20p): (A Monad for non-determinism)

A non-deterministic program can exhibit different behaviors on different runs even for the same input. In this problem, we model *all possible results from a non-deterministic computation* via a monad *ND*.

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 $m \ggg\!= f$ consists of executing m, taking all its possible results (of type a), passing each of them to continuation $f :: a \to ND\ b$ and collecting all the possible results (of type b) of those 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 = choice [1, 42, 100]   -- possible numbers are 1,42, and 100.
number2 :: ND Int
number2 = choice [2000, 30000, 50000]   -- possible numbers are 2000,30000,50000
```

The primitive *choice 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
ND {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 simply considering that each computation returns a list of all the possible results.

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

The non-proper morphisms of *ND* are given through the *MonadPlus* interface.

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

Method *mzero* models a computation that has no results, and *mplus* m_1 m_2 a computation that has the results of computation m_1 plus the results of computation m_2.

This is an example of how *mplus* works:

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

For our purposes, an instance of *MonadPlus* should satisfy these properties:

1. *mplus* forms a monoid with *mzero* as left and right identity.
2. *mplus* and *mzero* distribute over ($\gg=$) from the left.

As QuickCheck properties, these laws read as follows:

```
prop_mplus_assoc m₁ m₂ m₃ = (m₁ `mplus` m₂) `mplus` m₃ ≡ m₁ `mplus` (m₂ `mplus` m₃)
prop_mplus_left_identity  m = mzero `mplus` m            ≡ m
prop_mplus_right_identity m = m `mplus` mzero            ≡ m
prop_mzero_bind           k  = (mzero >>= k)             ≡ mzero
prop_mplus_bind m₁ m₂ k      = ((m₁ `mplus` m₂) >>= k)   ≡ ((m₁ >>= k) `mplus` (m₂ >>= k))
```

a) Your first task is to give the *Monad* and *MonadPlus* instances for *ND*. It is sufficient to provide the definition for *return*, ($\gg=$), *mzero* and *mplus*.

b) Your second task is to provide an implementation of *choice* that works for all instances of *MonadPlus*.

```
choice :: MonadPlus m ⇒ [a] → m a
```

c) The non-deterministic monad enables to write simple and compact code. For instance, the following code produces all possible permutation of a list.

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

The line *ps* ← *perm xs* could be thought of 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 following invocation of *perms* shows how it works.

```
>>> perm [1,2,3] :: ND [Int]
ND {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 third task is to implement the function *insert*:

$$insert :: MonadPlus\ m \Rightarrow a \rightarrow [\,a\,] \rightarrow m\ [\,a\,]$$

The following example shows how *insert* works.

$$\ggg insert\ 10\ [1,2,3] :: ND\ [Int]$$
$$ND\ \{results = [[10,1,2,3],[1,10,2,3],[1,2,10,3],[1,2,3,10]]\}$$

Problem 2 (20p): (Proving the laws for *ND*)

In this problem, we prove the *Monad* and *MonadPlus* laws for *ND*.

You can assume the monoid laws for the append function $(+\!\!+)$ and those for *MonadPlus ND*, i.e., *prop_mplus_assoc*, *prop_mplus_left_identity* and *prop_mplus_right_identity*. Also, you may use the fact that $ND\ a$ is isomorphic to $[\,a\,]$ via constructor *ND* and destructor *results* and use the respective laws silently:

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

When you prove a property by induction, state what you induct on, what the base case is and what the step case is. Also make clear where you apply the inductive hypothesis.

Prove equality statements by equality chains where each step is just one of the following transformations:

- Unfolding or folding a definition. Say which definition you (un)fold.
- Applying a proven or assumed property (theorem, lemma). State which property you use in this step.
- Applying the induction hypothesis.

a) Prove the distributivity laws *prop_mzero_bind* and *prop_mplus_bind*.

b) For *Monad ND*, prove left identity $(return\ x \gg\!= f) \equiv f\ x$ and right identity $(m \gg\!= return) \equiv m$.

c) Prove the associative law:

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

Problem 3 (20p): (DSL for Tuple Functions)

Here is the API of a DSL for composing functions on tuples:

```
type Fun a b

idF    :: Fun a a
compF  :: Fun b c → Fun a b → Fun a c
unitF  :: Fun a ()
pairF  :: Fun a b → Fun a c → Fun a (b, c)
crossF :: Fun a c → Fun b d → Fun (a, b) (c, d)
fstF   :: Fun (a, b) a
sndF   :: Fun (a, b) b
swapF  :: Fun (a, b) (b, a)
assocRF :: Fun ((a, b), c) (a, (b, c))
assocLF :: Fun (a, (b, c)) ((a, b), c)
eval   :: Fun a b → a → b
```

Its shallow embedding simply makes *Fun* the Haskell function type.

```
type FunS a b = a → b
idS        = id
compS f g  = f ∘ g
unitS      = λa → ()
pairS f g  = λa → (f a, g a)
crossS f g = λ(a, b) → (f a, g b)
fstS       = fst
sndS       = snd
swapS      = λ(a, b) → (b, a)
assocRS    = λ((a, b), c) → (a, (b, c))
assocLS    = λ(a, (b, c)) → ((a, b), c)
evalS f    = f
```

However, we are looking for a deep embedding that allows us to inspect functions and optimize their composition.

```
data Fun a b
instance Show (Fun a b)
```

Your task is to develop an optimized deep embedding of the *Fun* DSL via the following methodology:

1. Identify laws that allow the simplification of combined elements of *Fun*. Such a law is *compF idF f* $\equiv$ *f* but there are many more.
2. Identify *Fun*s that can be defined in terms of others.
3. Make the others constructors of *Fun*.

4. Define the API functions as smart constructors, applying simplifications according to the laws whereever possible.

Subtasks:

a) Split the API functions into *primitive* and *defined* ones. For the latter, list their definitions. For example:

$$swapF = pairF\ sndF\ fstF$$

It is ok to use defined functions to express other defined functions, but make sure your definitions are not cyclic. E.g., do not both express *pairF* via *crossF* and vice versa.

b) For the primitive functions, find as many laws as possible and list them here. Make sure you have no redundancy: none of the laws should follow from the others. E.g. $compF\ idF\ idF \equiv idF$ would be trivially an instance of the law $compF\ idF\ f \equiv f$.

c) Define a data type *Fun* of primitive functions and implement the rest of the API in a optimizing way. E.g.

```
data Fun a b where
  Id :: Fun a a
  Comp :: Fun b c → Fun a b → Fun a c
  ...
idF = Id
compF Id f = f
compF ...
```

Define *Fun* such that it can be made an instance of the *Show* class:

```
deriving instance Show (Fun a b)
```

In particular, the following shallow embedding would not work because Haskell functions are in general not printable:

```
data Fun a b = Fun (a → b)
```

Note: you can of course define auxiliary data types that are not exported in the API.

d) Define $eval :: Fun\ a\ b \to a \to b$.

e) Revisit your laws and list those that do not hold literally. This means you have a law $t = u$ but t and u produce different elements of *Fun*. E.g. if you simply defined $compF = Comp$, then $compF\ idF\ f$ is just equal to $Comp\ Id\ f$ which is different from f, violating the law $compF\ idF\ f \equiv f$. For each of the laws in this list, justify why they cannot easily be handled by smart constructors.

Hint: Check for instance that *swapF* is its own inverse, or the *assocRF* and *assocLF* are inverses of each other.