Chalmers | Göteborgs Universitet
Andreas Abel, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Wednesday, June 11, 2025, 8:30 - 12:30, Lindholmen

(including example solutions to programming problems)

- Examiner: Andreas Abel (+46-31-772-1731)
- Teacher Evan Cavallo 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: please contact the examiner.

Background: Probability Distributions

A *probability distribution* for a *sample space* Ω assigns a probability $p \in [0;1]$ to each event from the sample space. In general, an event is a subset of the sample space, but we will confine ourselves to *elementary events* $\omega \in \Omega$. Further, the probability may be different from 0 only for a *finite number* such events. Finally, we consider only rational probabilities $p \in \mathbb{Q}$. These restrictions allows us to represent distributions simply as *weighted lists*:

```
type P = Rational
type D a = [Weighted a]
data Weighted a = W {weight :: P, event :: a}
  deriving (Eq)
```

Such a weighted list is a proper probability distribution if the weights sum to 1.

For instance, the probability distribution of a fair (aka Laplace) coin is given by:

```
data Coin = Heads | Tails
  deriving (Eq, Bounded, Enum)

coinFlip :: D Coin
coinFlip = [W (1 / 2) Heads, W (1 / 2) Tails]
```

More generally, for a finite type (like *Coin*) we can define the *uniform* distribution that assigns each event the same probability.

```
uniformD :: (Enum a, Bounded a) => D a
uniformD = map (W p) events
  where
    events = [minBound .. maxBound]
    p = 1 / fromIntegral (length events)
```

Thus, *coinFlip* = *uniformD* :: *D Coin*.

The *Bernoulli distribution* assigns an event a given probability p and its opposite the counterprobability $1 - p$.

```
bernoulliD :: P -> D Bool
bernoulliD p = [W p True, W (1 - p) False]
```

Given two independent events, their joint probability can be computed as the product of the individual events:

```
crossD :: D a -> D b -> D (a, b)
crossD da db = [W (pa * pb) (a, b) | W pa a <- da, W pb b <- db]
```

For example, rolling two 6s with a standard die gives the following distribution, interpreting *True* as one 6:

```
double6 :: D (Bool, Bool)
double6 = crossD sixD sixD

sixD :: D Bool
sixD = bernoulliD (1 / 6)
```

double6 evaluates to the following distribution:

```
[ W (1 % 36) (True, True)
, W (5 % 36) (True, False)
, W (5 % 36) (False, True)
, W (25 % 36) (False, False)]
```

If we have a valuation $f : \Omega \to \mathbb{Q}$ of events and a probability distribution $d : \Omega \to [0; 1]$ we can compute the *expected value* as the sum:

$$\sum_{\omega \in \Omega} d(\omega) \cdot f(\omega)$$

Generalizing the valuation to $f : \Omega \to M$ where M is any *module*, meaning it is an additive monoid with scaling $scale\ p = p \cdot \_$, we obtain the function

```
runD :: Module m ⇒ (a → m) → D a → m
runD f = foldMap λ(W p x) → scale p (f x)
```

($foldMap :: Monoid\ m \Rightarrow (a \to m) \to [a] \to m$ does the summation using the monoidal <>.)

The definition of modules shall be given by the *Module* class:

```
class (Monoid m, Scale m) ⇒ Module m where
class Scale m where
  scale :: P → m → m
```

The simplest module are the rationals themselves:

```
instance Semigroup Rational where
  (<>) = (+)
instance Monoid Rational where
  mempty = 0
instance Scale Rational where
  scale = (∗)
instance Module Rational
```

As an example of an expected value, let us consider a game with 2 dice with an initial payment 1 EUR where you get your 1 EUR back if you roll one 6 but get 10 EUR if you roll two 6s:

```
expectedWin :: Rational
expectedWin = runD value double6 − 1
  where
    value = λcase
      (True, True) → 10
      (False, False) → 0
      _ → 1
```

(Aside question: Is this game fair?)

Problem 1 (20p): (Probability Monad)

Probability distributions can be seen as monad similar to the non-determinism monad. A monadic value like *coinFlip*::*D Coin* can be seen as choosing *Heads* or *Tails* non-deterministically with their associated probabilities (which are both $\frac{1}{2}$ in this case). Using the *Monad D* instance, example *double6* can be written as:

```
double6 :: D (Bool, Bool)
double6 = do
  x ← sixD
  y ← sixD
  return (x, y)
```

► **Task:** Define *Functor*, *Applicative* and *Monad* instances for *D*.
(Note: GHC would require a **newtype** definition like **newtype** *D a* = *D* [*Weighted a*] but you are welcome to use the plain type synonym **type** *D a* = [*Weighted a*] to save you some writing.)

SOLUTION:

```
mapEvent :: (a → b) → Weighted a → Weighted b
mapEvent f (W p a) = W p (f a)

mapD :: (a → b) → D a → D b
mapD f = map (mapEvent f)

returnD :: a → D a
returnD a = [W 1 a]

zipWithD :: (a → b → c) → D a → D b → D c
zipWithD f da db = [W (pa * pb) (f a b) | W pa a ← da, W pb b ← db]

apD :: D (a → b) → D a → D b
apD = zipWithD ($)

bindD :: D a → (a → D b) → D b
bindD da k = [W (pa * pb) b | W pa a ← da, W pb b ← k a]


instance Functor D where
  fmap = mapD

instance Applicative D where
  pure = returnD
  (<*>) = apD

instance Monad D where
  (>>=) = bindD
```

Problem 2 (15p): (Application: Risk)

In the popular boardgame *Risk* battles are fought between the armies of the attacker and the armies of the defender by rolling standard dice.

```
data Die = D6 | D5 | D4 | D3 | D2 | D1
  deriving (Eq, Ord, Show, Bounded, Enum)
```

In reach round, attacker and defender both role a number of dice limited by the number of their armies. (In the game, the maximum number of dice is capped to 3 for the attacker, and for the defender to 2.) The dice of attacker and defender are sorted descendingly and then compared positionwise. A lower number causes an army loss, in case of a draw the attacker loses.

For instance, if the attacker rolls 641 and the defender 54, each loses one army. If the attacker rolls 66 and the defender 6, just the attacker loses an army. If the attacker rolls 331 and the defender 21, the defender loses two armies.

To implement a Risk simulation, we represent the outcome of one round by the following record:

```
data Outcome = Outcome
  {defenderLosses :: Rational
  , attackerLosses :: Rational}
```

► **Task:** Write functions

```
riskD :: Int → Int → D Outcome
riskE :: Int → Int → Outcome
```

that given the number of attacker and defender dice return a probability distribution *riskD* and expected value *riskE* for the outcome of a round.

You may use standard Haskell functions freely (from packages like *base* shipped with GHC).

SOLUTION:

```
dieD :: D Die
dieD = uniformD
instance Semigroup Outcome where
  Outcome d1 a1 <> Outcome d2 a2 = Outcome (d1 + d2) (a1 + a2)
instance Monoid Outcome where
  mempty = Outcome 0 0
instance Scale Outcome where
  scale s (Outcome x y) = Outcome (s ∗ x) (s ∗ y)
instance Module Outcome
  -- Outcome for one army if 'True' is interpreted as attacker winning.
outcome :: Bool → Outcome
outcome True = Outcome 1 0
outcome False = Outcome 0 1


riskD nA nD = do
  as ← sort ⟨$⟩ replicateM nA dieD
```

```
    ds ← sort ⟨$⟩ replicateM nD dieD
    let attackerWins = zipWith (>) as ds
    return $ foldMap outcome attackerWins
riskE nA nD = runD id $ riskD nA nD
```

Problem 3 (20p): (**Monad laws**)

► **Task:** Prove the 3 monad laws for D using step-by-step equational reasoning.
Each step must be explicitly justified, either by "definition" or "computation", by appeal to some theorem or already proven property, or by some (induction) hypothesis.

SOLUTION:

```
prop_return_bind :: Eq b ⇒ a → (a → D b) → Proof (D b)
prop_return_bind a k = proof
  (bindD (returnD a) k)                               ≡⟨ Def bindD   ⟩≡
  [W (pa ∗ pb) b | W pa a ← returnD a, W pb b ← k a]  ≡⟨ Def returnD ⟩≡
  [W (pa ∗ pb) b | W pa a ← [W 1 a], W pb b ← k a]    ≡⟨ Conv        ⟩≡
  [W (1 ∗ pb) b | W pb b ← k a]                       ≡⟨ Conv        ⟩≡
  [W pb b | W pb b ← k a]                             ≡⟨ Conv        ⟩≡
  (k a)

prop_bind_return :: Eq a ⇒ D a → Proof (D a)
prop_bind_return d = proof
  (bindD d returnD)                                   ≡⟨ Def bindD   ⟩≡
  [W (pa ∗ pb) b | W pa a ← d, W pb b ← returnD a]    ≡⟨ Def returnD ⟩≡
  [W (pa ∗ pb) b | W pa a ← d, W pb b ← [W 1 a]]      ≡⟨ Conv        ⟩≡
  [W (pa ∗ 1) a | W pa a ← d]                         ≡⟨ Conv        ⟩≡
  [W pa a | W pa a ← d]                               ≡⟨ Conv        ⟩≡
  d

prop_bind_assoc :: Eq c ⇒ D a → (a → D b) → (b → D c) → Proof (D c)
prop_bind_assoc d f g = proof
  (bindD (bindD d f) g)                                            ≡⟨ Def bindD    ⟩≡
  [W (p ∗ pc) c | W p b ← bindD d f, W pc c ← g b]                 ≡⟨ Def bindD    ⟩≡
  [W (p ∗ pc) c
    | W p b ← [W (pa ∗ pb) b | W pa a ← d, W pb b ← f a]
    , W pc c ← g b]                                                ≡⟨ Conv         ⟩≡
  [W ((pa ∗ pb) ∗ pc) c | W pa a ← d, W pb b ← f a, W pc c ← g b]  ≡⟨ Thm "assoc*" ⟩≡
  [W (pa ∗ (pb ∗ pc)) c | W pa a ← d, W pb b ← f a, W pc c ← g b]  ≡⟨ Conv         ⟩≡
  [W (pa ∗ p) c
    | W pa a ← d
    , W p c ← [W (pb ∗ pb) c | W pb b ← f a, W pc c ← g b]]        ≡⟨ Def bindD    ⟩≡
  [W (pa ∗ p) c | W pa a ← d, W p c ← bindD (f a) g]               ≡⟨ Def bindD    ⟩≡
  (bindD d (λa → bindD (f a) g))
```

Problem 4 (5p): (More efficient representation of distributions)

Suppose we define **type** $D\ a = Map\ a\ P$ to use *tree maps* instead of lists for the representation of the distribution.

► **Task:** Answer the following questions:

1. Which constraints are placed on a to enable such a representation?

2. What formal problem would we run into when trying to define the *Monad* instance for this D?

3. How can we (at least partially) address this problem?

SOLUTION:

1. Type a needs to be a decidable linear order, i.e., it needs to implement $Ord\ a$.

2. The type of bind is $D\ a \to (a \to D\ b) \to D\ b$, but to construct a distribution $D\ b$ over b we would need b to be ordered. Yet the type does not accommodate an $Ord\ b$ constraint.

   Worse, the type of idiomatic application is $D\ (a \to b) \to D\ a \to D\ b$ and generally there is no decidable order on functions.

3. We could add the constraint to the type of bind in our own definition of "monads of ordered types". However, the **do** notation would likely not be available.

   ```
   class OMonad m where
     returnO :: a → m a
     bindO :: Ord b ⇒ m a → (a → m b) → m b
   ```

   Idiomatic application cannot be salvaged, but we can implement applicative functors using *liftA2*.

   ```
   class OFunctor m where
     mapO :: Ord b ⇒ (a → b) → m a → m b
   class OApplicative m where
     pureO :: a → m a
     liftA2O :: Ord c ⇒ (a → b → c) → m a → m b → m c
   ```