Chalmers | Göteborgs Universitet
Andreas Abel, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Saturday, March 22, 2025, 8:30 - 12:30, J.

(including example solutions to programming problems)

- 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**: ⩾ 24 points, **4**: ⩾ 36 points, **5**: ⩾ 48 points.
  GU: Godkänd ⩾ 24 points, väl godkänd ⩾ 48 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 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.

A DSL for boxy text layout

Consider the following DSL for 2-dimensional text layout.
Types:

```
type Height  -- Non-negative integers.
type Width   -- Non-negative integers.
type HAlign  -- Horizontal alignment.
type VAlign  -- Vertical alignment.
type Box     -- Rectangular area.
```

Elementary constructors:

```
left, middle, right  :: HAlign
top, center, bottom :: VAlign
line     :: String → Box              -- Minimal box of height 1 holding the given text.
box      :: [String] → Box            -- Minimal box holding a text given by a list of lines.
empty    :: Box                       -- Empty box taking no space.
blank    :: Height → Width → Box      -- Empty box with the given dimension.
vphantom :: Height → Box              -- Zero-width box of the given height.
hphantom :: Width → Box               -- Zero-height box of the given width.
```

Combinators:

```
beside :: VAlign → Box → Box → Box
    -- Place two boxes beside each other with the given vertical alignment.
above  :: HAlign → Box → Box → Box
    -- Stack two boxes vertically with the given horizontal alignment.
```

Run function:

```
data Pic = Pic
  {height  :: Height
  , width   :: Width
  , content :: [String]   -- list of 'height' lines of length 'width'
  }
render :: Box → Pic
```

CLARIFICATION: *content* should be padded, so have exactly *height* many *String* entries of length exactly *width*.

Here is an example use of this DSL:

```
spaceToDot = mapλcase {' ' → '.'; c → c}
ex1 = spaceToDot ∘ unlines ∘ content ∘ render $
  foldr1 (beside top)
    [foldr1 (above center) $
      hphantom 12 :
      map line ["EU Country", replicate 10 '-', "France", "Germany", "Sweden"]
```

```
        , hphantom 2
        , foldr1 (above right) $
            map line ["Area/km²", replicate 8 '-', "551,695", "357,592", "450,295"]
        ]
```

ex1 generates the following picture:

```
.EU.Country...Area/km²
.----------...--------
..France.......551,695
..Germany......357,592
..Sweden.......450,295
```

CLARIFICATION: *above center* is a typo, it should be *above middle*.

Problem 1 (12p): (Shallow embedding of the DSL)

Implement DSL using a shallow embedding of *Box*.
You may assume the following padding functions to implement alignment:

```
  -- Insert padding according to 'VAlign' to give the 'Pic' a larger 'Height'.
vpad :: VAlign → Height → Pic → Pic
  -- Insert padding according to 'HAlign' to give the 'Pic' a larger 'Width'.
hpad :: HAlign → Width → Pic → Pic
```

CLARIFICATION: *vpad* and *hpad* take padded *Pic*s as input and produce such of possibly larger size. They may malfunction if the input is not correctly padded.

SOLUTION:

```
type Height = Int   -- non-negative
type Width  = Int   -- non-negative

type VAlign = Align
type HAlign = Align

data Align
   = Start     -- Align left (horizontal mode) or top (vertical mode).
   | Center    -- Center.
   | End       -- Align right (horizontal mode) or bottom (vertical mode).
  deriving (Eq, Show, Enum, Bounded)

left       = Start
middle     = Center
right      = End

top        = Start
center     = Center
```

```
bottom       = End

type Box     = Pic
render       = id

line s       = Pic 1 (length s) [s]

empty        = Pic 0 0 []
blank h w    = Pic h w $ replicate h $ replicate w ' '
hphantom w = blank 0 w
vphantom h  = blank h 0

beside a p1@(Pic h1 w1 c1) p2@(Pic h2 w2 c2)
   | h1 < h2  = beside a (vpad a h2 p1) p2
   | h1 > h2  = beside a p1 (vpad a h1 p2)
   | h1 ≡ h2  = Pic h1 (w1 + w2) $ zipWith (++) c1 c2

above a p1@(Pic h1 w1 c1) p2@(Pic h2 w2 c2)
   | w1 < w2 = above a (hpad a w2 p1) p2
   | w1 > w2 = above a p1 (hpad a w1 p2)
   | w1 ≡ w2 = Pic (h1 + h2) w1 $ c1 ++ c2

box [] = empty
box (s : ss) = above left (line s) (box ss)
```

This is the implementation of padding.

```
  -- Insert padding according to 'VAlign' to give the 'Pic' a larger 'Height'.
vpad :: VAlign → Height → Pic → Pic
vpad a h (Pic h' w ls) = Pic h w $ pad a h h' (replicate w ' ') ls
  -- Insert padding according to 'HAlign' to give the 'Pic' a larger 'Width'.
hpad :: HAlign → Width → Pic → Pic
hpad a w (Pic h w' ls) = Pic h w $ map (pad a w w' ' ') ls
  -- Pad a list of things with copies of a thing to stretch it to a new length.
  -- Precondition: n ⩾ n' and all elements of xs have length n'.
  -- Postcondition: all elements of the result have length n.
pad :: Align → Int → Int → a → [a] → [a]
pad a n n' x xs = case a of
  Start   → xs ++ replicate d x
  End     → replicate d x ++ xs
  Center → replicate d1 x ++ xs ++ replicate d2 x
  where
    d   = n − n'
    d1 = d `div` 2
    d2 = d − d1
```

GRADING: Common glitches (no points deducted): ±0p

- Suprisingly many did not simply define **type** *Box* = *Pic* (the standard choice for a shallow embedding) but a *Box* that was merely isomorphic to *Pic*, requiring back and forth conversions.

- Most ignored that *maximum* does not work for empty lists.
- Many opted to define *hphantom* and *vphantom* directly rather than via *blank*. (From a cognitive perspective, this is not suprising, as we tend to build understanding by looking first at the simple cases before attacking the general case. Mathematically, *hpantom* and *vphantom* are very much redundant; their purpose was to draw attention to the special cases of zero height and zero width *blank*s that could otherwise be overlooked.)

Common errors:

- Many missed that the *box* constructor needs to construct rectangular text (adding spaces at the right to achieve uniform width)—otherwise horizontal composition does not produce columns but faulty per-line concatenation: -2p
- Similar issues can arise with *blank* but suprisingly this was much less frequent.

Individual errors:

- **type** *Box* = [*String*] does not work since one cannot implement *hphantom*.
- Proving a deep instead of a shallow embedding: -2p
- Adding heights or widths where maximum is needed: -1p each.
- Missing *maximum* to compute *Width* of *Pic*: -1p
- Malformed call to *hpad* or *vpad*: -1p

Problem 2 (10p): (Laws of the DSL)

1. Formulate some laws involving *above* and *beside* you expect to hold. You can write down formulas or use precise mathematical language to describe them (e.g. "+ is a monoid with unit 0").
2. If I replace *foldr1* with *foldl1* in example *ex1*, I get a slightly different output:

   ```
   .EU.Country...Area/km²
   .----------...--------
   ...France......551,695
   ..Germany......357,592
   ...Sweden......450,295
   ```

   This indicates that a certain law fails to hold in my implementation.

   1 Which one?

   2 What could be the reason?

   3 Does it also fail in your implementation? (Justify your answer.)

SOLUTION: For each alignment mode a, we expect both *above* a and *beside* a to be monoids with unit *empty*. This is true except that:

1. Associativity fails for *Center* alignment, both for *above* and *beside*.
2. Reason are rounding errors when splitting uneven total padding into left and right padding. If we center a single letter in a box of width 3, we expect it in the middle with one space character on each side. However, if we first center it in a box of width 2, and the result in a box of width 3, then the character will be left aligned, because in each of the two steps we allocate the one-space padding at the right end. (Depending on implementation, it could also be that each time we allocate at the left end and get a right-aligned text as the final result.)
3. Yes, because in the centering case the left padding is computed by d ‘*div*‘ 2 from the total padding d and this rounds down each time.

For completeness, we spell out the properties as QuickCheck properties:

```
prop_empty_above  a p = above a empty p ≡ p
prop_above_empty  a p = above a p empty ≡ p
prop_empty_beside a p = beside a empty p ≡ p
prop_beside_empty a p = beside a p empty ≡ p
  -- Associativity fails in the Center case due to rounding errors
prop_above_assoc  a p1 p2 p3 = a ≡ Center
   ∨ above a (above a p1 p2) p3 ≡ above a p1 (above a p2 p3)
prop_beside_assoc  a p1 p2 p3 = a ≡ Center
   ∨ beside a (beside a p1 p2) p3 ≡ beside a p1 (beside a p2 p3)
prop_counterexample_assoc_center = (p1 <> p2) <> p3 ≢ p1 <> (p2 <> p3)
  where
    (<>) = above Center
    p1 = line "abc"
    p2 = line "x"
    p3 = line "yz"
```

GRADING:

- There are many laws about height and width that are not so interesting for the purpose of this problem, but gave partial credit if the monoid laws were not given.
- Many knew that the *foldl* vs. *foldr* discrepancy points to a failure of associativity.
- The best explanations of the failure of associativity would point to the asymmetry in padding when centering, caused by the need to round or truncate when diving an odd amount of padding by two. Explanations that did not get to the bottom of the problem, i.e., did not clearly point to the rounding error or lack of symmetry in centering, could get partial credit: -2p.

- Some suggested that *hpad* and *vpad* are just wrongly implemented. That is not the case. Their type does not permit an implementation that can compensate the rounding errors, because *Pic* cannot store that rounding has happened: by its specification, the *content* needs to be exactly *height* lines of exact length *width*.
- If one drops this specification, one could hack the information in there: for odd *width*, if all the *content* lines are short by one character, this could code that there is a virtual "half space" on each side. These half spaces can compensate a subsequent odd padding or they become a single space at the end during the final rendering. (One participant came up with this solution in Problem 3, but a clean implementation of this solution would add a *Bool* flag to *Box* whether such half space is present rather than hacking it into *content*.)

Problem 3 (8p): (Deeper embedding of the DSL)

Sketch a deeper embedding of the DSL that does not exhibit the problem with *foldr1* vs. *foldl1* in the previous question. You can refer to the code of your shallow embedding to explain the deeper embedding.

It is not necessary to write out a full implementation. It is sufficient to explain the idea of the solution and the data structure(s) needed to do so.

SOLUTION: To fix the problem with associativity of composition, we perform composition (and thus padding) lazily. Boxes are now modelled by a trees that in the leaf have *Pic*s and the nodes can either be horizontal or vertical compositions, each taking a list of subtrees. The nodes also store the alignment mode and cache the size of the picture resulting from the composition.

```
data Box'
   = P Pic                              -- basic text box
   | H Height Width Align [Box']        -- horizontal sequence of boxes
   | V Height Width Align [Box']        -- vertical sequence of boxes
size :: Box' → (Height, Width)
size = λcase
  P (Pic h w _) → (h, w)
  H h w _ _     → (h, w)
  V h w _ _     → (h, w)
```

The rendering of nodes takes care that all pictures in the composition are padded to the *final* height or width before concatenation.

```
render' :: Box' → Pic
render' = λcase
  P p          → p
  H h w a bs → foldr1 (beside a) $ map (vpad a h ∘ render') bs
  V h w a bs → foldr1 (above a) $ map (hpad a w ∘ render') bs
```

The composition operators *beside'* and *above'* need to make sure they merge compositions of the same alignment into one. This means, a *H* node with a given alignment may not directly contain *H* nodes with the same alignment again. (And analogously for *V* nodes.)

This sufficies as a sketch for the deeper embedding, we list the implementation for completeness sake.

The new horizontal composition *beside′* makes sure to merge horizontal compositions of the same alignment.

```
beside′ :: VAlign → Box′ → Box′ → Box′
beside′ a b1 b2 =
  case (hView a b1, hView a b2) of
    (Yes h1 w1 bs1, Yes h2 w2 bs2)
       → H (max h1 h2) (w1 + w2) a (bs1 ++ bs2)
    _  → H (max h1 h2) (w1 + w2) a [b1, b2]
      where
        (h1, w1) = size b1
        (h2, w2) = size b2
```

To this end, we introduce a view *hView* that tries to view a box as a horizontal composition of the given alignment. A single *Pic* can aways be seen as such.

```
data View
   = Yes Height Width [Box′]
   | No
hView :: VAlign → Box′ → View
hView a = λcase
  H h w a′ bs | a ≡ a′ → Yes h w bs
  b@(P (Pic h w _))    → Yes h w [b]
  _                    → No
```

Vertical composition is implemented analogously.

```
above′ :: VAlign → Box′ → Box′ → Box′
above′ a b1 b2 =
  case (vView a b1, vView a b2) of
    (Yes h1 w1 bs1, Yes h2 w2 bs2)
       → V (h1 + h2) (max w1 w2) a (bs1 ++ bs2)
    _  → V (h1 + h2) (max w1 w2) a [b1, b2]
  where
    (h1, w1) = size b1
    (h2, w2) = size b2
vView :: HAlign → Box′ → View
vView a = λcase
  V h w a′ bs | a ≡ a′ → Yes h w bs
  b@(P (Pic h w _))    → Yes h w [b]
  _                    → No
```

The basic constructors are straightforward:

```
box′      :: [String] → Box′
box′      = P ∘ pic
pic       :: [String] → Pic
pic []        = Pic 0 0 []
pic ss        = Pic (length ss) w $ zipWith (λw' s → pad Start w w' ' ' s) ws ss
  where
    ws = map length ss
    w  = maximum ws
line'       :: String → Box'
line' s     = box' [s]
blank'      :: Height → Width → Box'
blank' h w  = P $ Pic h w $ replicate h $ replicate w ' '
empty'      :: Box'
empty'      = blank' 0 0
```

GRADING:

- Without making the deep embedding concrete and explaining how it fixes the problems one could not get full points.
- The essential ingredient is a deep embedding where we keep stacks (vertical compositions) and sequences (horizontal compositions) of boxes around that can be extended by more elements of the same alignment.
- Only when it is clear that there will be no more extensions coming can these explicit compositions be collapsed into a *Pic*. Naturally, this is when we call *render*, but one could do it more eagerly, e.g. when a stack becomes an element of a sequence and thus cannot be extended by *above* anymore. (There is no obvious advantage of this eagerness, though.)
- This is the general lesson of deep embeddings: keep structural information (like the genesis of an element) around by turning operations into "syntax", and use this information to inspect, optimize, or (in our case) correct an element.

Monads for Cost Accounting

In the following problems, we will develop monads that allow to attach costs to computations and account for these costs.

```
class (Monad m, Monoid c) ⇒ MonadCost c m where
  pay :: c → m a → m a

type Cost c a

runCost :: Cost c a → (a, c)

instance Monoid c ⇒ Monad (Cost c)
instance Monoid c ⇒ MonadCost c (Cost c)
```

Using this API, we can implement cost-instrumented programs. For example, the following function computes the greatest common divisor and counts the number of subtractions needed to do so.

```
gcd :: MonadCost Integer m ⇒ Integer → Integer → m Integer
gcd m n
   | n ⩽ 0  = return m
   | m ⩽ 0 = return n
   | otherwise = case compare m n of
     LT → pay 1 $ gcd m (n − m)
     GT → pay 1 $ gcd (m − n) n
     EQ → return m

exGcd = runCost $ gcd 640 60
```

Definition *exGcd* has value $(20, 12)$ meaning that the *gcd* of 640 and 60 was computed to 20 using 12 subtractions.

Problem 4 (6p): (Implement *Cost*)

Implement the type *Cost*, function *runCost* and the instances for *Monad* and *MonadCost*. (You need not implement the *Functor* and *Applicative* instances.)

SOLUTION:

```
newtype Cost c a = Cost {runCost :: (a, c)}

instance Monoid c ⇒ Monad (Cost c) where
  return a          = Cost (a, mempty)
  Cost (a, c) >>= k = pay c (k a)

instance Monoid c ⇒ MonadCost c (Cost c) where
  pay c (Cost (a, c')) = Cost (a, c <> c')
```

GRADING:

- Another valid solution is to define *Cost c* as a state monad, with field $run :: c \to (a, c)$.

- Most defined $(\gg\!=)$ by hand, but first defining *pay* and then defining $(\gg\!=)$ in terms of it simplifies the proofs in Problem 5.

Common glitches (no points deducted):

- Defining *Cost* to be (a, c): without a **newtype** wrapper, it is not possible to make this type a *Monad* or *MonadCost* instance.
- Using 0 or $(+)$ instead of the monoid operators *mempty* and $(<>)/\mathit{mappend}$.

Common errors:

- Forgetting to combine the costs of the two computations in $(\gg\!=)$.
- Forgetting to define a *runCost* with the requested type, if defining *Cost* as a state monad.

Problem 5 (12p): (**Verify** *Cost*)

Prove the following properties (written as QuickCheck properties) using step-by-step equational reasoning. Each step must be explicitly justified, either ”by definition”, by appeal to some theorem or already proved property, or by some (induction) hypothesis.

$$\begin{array}{lcl}
\mathit{prop\_pay\_empty}\;m & = & \mathit{pay}\;\mathit{mempty}\;m \equiv m \\
\mathit{prop\_pay\_mappend}\;c\;c'\;m & = & \mathit{pay}\;c\;(\mathit{pay}\;c'\;m) \equiv \mathit{pay}\;(c <> c')\;m \\
\mathit{prop\_pay\_bind}\;c\;c'\;m\;k & = & (\mathit{pay}\;c\;m \gg\!= \lambda a \to \mathit{pay}\;c'\;(k\;a)) \equiv \mathit{pay}\;(c <> c')\;(m \gg\!= k)
\end{array}$$

The third property requires the *Monoid* c to be commutative.

SOLUTION: It suffices to show these properties for m of the form $\mathit{Cost}\;(a, c)$ for arbitrary a and c.

$$\begin{array}{l}
\mathit{prop\_eq\_pay\_empty}\;(\mathit{Cost}\;(a, c)) = \mathit{proof} \\
\quad \begin{array}{ll}
(\mathit{pay}\;\mathit{mempty}\;(\mathit{Cost}\;(a, c))) & \equiv\langle\; \mathit{Def}\;\texttt{"pay"} \;\rangle\equiv \\
(\mathit{Cost}\;(a, \mathit{mempty} <> c)) & \equiv\langle\; \mathit{Thm}\;\texttt{"monoid left unit"} \;\rangle\equiv \\
(\mathit{Cost}\;(a, c)) &
\end{array}
\end{array}$$

$$\begin{array}{l}
\mathit{prop\_eq\_pay\_mappend}\;c1\;c2\;(\mathit{Cost}\;(a, c3)) = \mathit{proof} \\
\quad \begin{array}{ll}
(\mathit{pay}\;c1\;(\mathit{pay}\;c2\;(\mathit{Cost}\;(a, c3)))) & \equiv\langle\; \mathit{Def}\;\texttt{"pay"} \;\rangle\equiv \\
(\mathit{pay}\;c1\;(\mathit{Cost}\;(a, c2 <> c3))) & \equiv\langle\; \mathit{Def}\;\texttt{"pay"} \;\rangle\equiv \\
(\mathit{Cost}\;(a, c1 <> (c2 <> c3))) & \equiv\langle\; \mathit{Thm}\;\texttt{"monoid associativity"} \;\rangle\equiv \\
(\mathit{Cost}\;(a, (c1 <> c2) <> c3)) & \equiv\langle\; \mathit{Def}\;\texttt{"pay"} \;\rangle\equiv \\
(\mathit{pay}\;(c1 <> c2)\;(\mathit{Cost}\;(a, c3))) &
\end{array}
\end{array}$$

$$\begin{array}{l}
\mathit{prop\_eq\_pay\_bind}\;c1\;c3\;(\mathit{Cost}\;(a, c2))\;k = \mathit{proof} \\
\quad \begin{array}{ll}
(\mathit{pay}\;c1\;(\mathit{Cost}\;(a, c2)) \gg\!= \lambda a \to \mathit{pay}\;c3\;(k\;a)) & \equiv\langle\; \mathit{Def}\;\texttt{"pay"} \;\rangle\equiv \\
(\mathit{Cost}\;(a, c1 <> c2) \gg\!= \lambda a \to \mathit{pay}\;c3\;(k\;a)) & \equiv\langle\; \mathit{Def}\;\texttt{"bind"} \;\rangle\equiv \\
(\mathit{pay}\;(c1 <> c2)\;(\mathit{pay}\;c3\;(k\;a))) & \equiv\langle\; \mathit{Thm}\;\texttt{"prop\_pay\_mappend"} \;\rangle\equiv \\
(\mathit{pay}\;((c1 <> c2) <> c3)\;(k\;a)) & \equiv\langle\; \mathit{Thm}\;\texttt{"commutative monoid"} \;\rangle\equiv \\
(\mathit{pay}\;((c1 <> c3) <> c2)\;(k\;a)) & \equiv\langle\; \mathit{Thm}\;\texttt{"prop\_pay\_mappend"} \;\rangle\equiv \\
(\mathit{pay}\;(c1 <> c3)\;(\mathit{pay}\;c2\;(k\;a))) & \equiv\langle\; \mathit{Def}\;\texttt{"bind"} \;\rangle\equiv \\
(\mathit{pay}\;(c1 <> c3)\;(\mathit{Cost}\;(a, c2) \gg\!= k)) &
\end{array}
\end{array}$$

GRADING:

- For some solutions to Problem 4, *prop_eq_pay_mappend* also required commutativity. Full points were given for either using commutativity or simply proving $\mathit{pay}\;c\;(\mathit{pay}\;c'\;m) \equiv \mathit{pay}\;(c' <> c)\;m$ instead.
- Proofs of *prop_eq_pay_bind* could be very long if $(\gg\!=)$ was not defined using *pay*. One participant who defined $(\gg\!=)$ by hand had the good idea to prove that $\mathit{Cost}\;(a, c) \gg\!= k = \mathit{pay}\;c\;(k\;a)$ as a lemma!

Common errors:

- Forgetting to use associativity, or only citing commutativity while also using associativity.

Problem 6 (12p): (Fueled computation)

In the following, rather than just being interested in the total cost of a computation, we want to limit the cost of a computation. We run computations with a budget from which the costs are paid. If the budget is spent before the computation has finished, we abort the computation. Function *try m handler* allows us to run *handler* if m was aborted, yet the *handler* only receives the budget that was left just before the first payment in m failed.

```
class (MonadCost c m, Num c, Ord c) ⇒ MonadFuel c m where
  try :: m a → m a → m a
type Fueled c a
instance (Monoid c, Num c, Ord c) ⇒ MonadCost c (Fueled c)
instance (Monoid c, Num c, Ord c) ⇒ MonadFuel c (Fueled c)
```

The runner *runFueled* for *Fueled c a* computations takes a fallback value in a and a initial budget in c. In any case, the remaining fuel is returned. If the computation ran out of fuel, the fallback value is returned, otherwise the result of the computation.

```
runFueled :: a → c → Fueled c a → (a, c)
```

As an example, let us first implement a helper *afford* that will run a priced computation if possible or return a fallback value.

```
afford :: MonadFuel c m ⇒ a → c → m a → m a
afford fallback cost computation =
  try (pay cost computation) (return fallback)
```

Using *afford*, the function *hare n* computes (m, n') such that m is largest with $n = n' + 1 + 2 + ... + m$ and n, n', m are natural numbers.

```
hare :: Integer → (Integer, Integer)
hare n = runFueled (−1) n $ loop 0
  where
    loop :: Integer → Fueled Integer Integer
    loop m = afford m (m + 1) $ loop (m + 1)
```

1. Implement type *Fueled*, function *runFueled* and the *Monad*, *MonadCost*, and *MonadFuel* instances. To that end, you may use the standard Haskell monad transformers.
2. What does your implementation return in the following example?

   ```
   exTry = runFueled "A" 4 do
     try (pay 3 $ pay 3 $ return "B") do
       try (pay 2 $ return "C") do
         pay 1 $ return "D"
   ```

   Explain the result.
3. Which of the laws of *pay* that we have required for *Cost* also hold for *Fueled*?

   Please justify your answer, but a formal proof is not required.

SOLUTION: 1. Implementation using the *ExceptT* monad transformer and the *State* monad:

```
data OutOfFuel = OutOfFuel
newtype Fueled c a = Fueled {unFueled :: ExceptT OutOfFuel (State c) a}
  deriving (Functor, Applicative, Monad)
runFueled :: a → c → Fueled c a → (a, c)
runFueled a c (Fueled m) = (fromRight a r, c′)
  where
    (r, c′) = runState (runExceptT m) c
instance (Monoid c, Num c, Ord c) ⇒ MonadCost c (Fueled c) where
  pay c m = Fueled do
    fuel ← lift get
    let fuel′ = fuel − c
    if fuel′ < 0 then throwError OutOfFuel else do
      lift $ put fuel′
      unFueled m
instance (Monoid c, Num c, Ord c) ⇒ MonadFuel c (Fueled c) where
  try m₁ m₂ = Fueled $ unFueled m₁ ‘catchError‘ λOutOfFuel → unFueled m₂
```

2. Definition *exTry* returns ("D", 0). From the initial budget of 4 units, first 3 units are spent. For the next payment of 3 the budget is short, so we abort the attempt of returning "B". The next option to pay 2 to return "C" also exceeds the budget. We can however pay 1 and return "D" but then our budget totally consumed. Since we managed to terminate the computation just in time, the fallback value of "A" does not become active.

3. We cannot aggregate payments in *Fueled* since they might then exceed our budget while individual payments could go through. Thus laws *prop_pay_mappend* and *prop_pay_bind* are not valid for *Fueled*. Only *prop_pay_empty* remains valid as paying 0 is possible with any budget and will not alter our budget.

GRADING: Common errors:

- Treating the c in a *Fueled* c a as the amount of fuel left in some places but as the amount of fuel used so far in other places.
- Defining *Fueled* c a as a wrapper around (a, c). If the c is the amount of fuel spent so far, it is impossible to tell how much is left; if it is the amount of fuel that is left, it is impossible to plug in an initial fuel value in *runFueled*.
- Missing that if *try m handler* goes to the handler, the fuel consumed by m should still be consumed. This is important for understanding *exTry* as well.