Chalmers | Göteborgs Universitet
Andreas Abel, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

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

- 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: 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 = map λcase { ' ' → '.'; 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/km^2", replicate 8 '-', "551,695", "357,592", "450,295"]
]
```

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

ex1 generates the following picture:

```
.EU.Country...Area/km^2
.----------...--------
..France.......551,695
..Germany......357,592
..Sweden.......450,295
```

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.

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/km^2
   .----------...--------
   ...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.)

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.

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.)

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.

```
prop_pay_empty m          = pay mempty m ≡ m
prop_pay_mappend c c' m = pay c (pay c' m) ≡ pay (c <> c') m
prop_pay_bind c c' m k    = (pay c m ≫= λa → pay c' (k a)) ≡ pay (c <> c') (m ≫= k)
```

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

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.

Good luck!