Chalmers | Göteborgs Universitet
Andreas Abel, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Monday, August 26, 2024, 14:00 - 18:00, J.

(including example solutions to programming problems)

- Examiner: Andreas Abel (+46-31-772-1731), visits 14:45 and 16:15.
- 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.

An interface for errors and warnings

This is a possible interface for a monad that allows to throw fatal errors and non-fatal warnings:

```
class (Monad m, Monoid w) ⇒ MonadErrorWarning e w m
  where
    raise  :: e → m a
    handle :: (e → w → m a) → m a → m a
    warn   :: w → m ()
    clear  :: m a → m (a, w)
```

Herein, *e* is the type of errors and *w* the type of warnings. Only one error can be thrown in a computation, and it will abort the computation. However, many warnings can be raised; those do not abort the computation. Warnings are merely collected; thus, the type *w* is required to be a *Monoid*. Recall the interface to monoids (slightly simplified, not requiring a *Semigroup* superclass):

```
  class Monoid a where
    mappend :: a → a → a   -- Associative binary operation.
    mempty  :: a           -- Left and right unit for mappend.
```

In *MonadErrorWarning*, *mempty* is used for "no warnings" and *mappend* for joining two warning collections.

The individual methods of *MonadErrorWarning* shall implement the following informal specification. (In the specification we refer to some laws *prop_XYZ* given later which can be ignored in the first reading.)

1. *raise e* throws fatal error *e*. Subsequent statements are not executed (*prop_bind_raise*).
2. *handle h comp* runs computation *comp*. If *comp* raises no error, its result is returned (*prop_handle_warn_return*). Otherwise, its result is discarded and the handler *h w e* is run, where *w* is the collection of warnings raised in *comp* and *e* the error thrown in *comp* (*prop_handle_warn_raise*).
3. *warn w* appends *w* to the so far collected warnings (*prop_warn_mempty* and *prop_warn_mappend*). It has no result otherwise.
4. *clear comp* runs *comp* and returns its result and collected warnings. The collection of warnings is reset, thus, the computation *clear comp* itself is warning-free (*prop_clear_warn_return*).

We do not attempt to make the specification completely formal, but we identify the following equational laws, written as *QuickCheck* properties, that should hold for each instance of *MonadErrorWarning*:

```
  prop_bind_raise e k             = (raise e ≫= k)             ≡ raise e

  prop_handle_warn_raise h w e    = handle h (warn w ≫ raise e)  ≡ h e w
  prop_handle_warn_return h w a   = handle h (warn w ≫ return a) ≡ (warn w ≫ return a)

  prop_warn_mempty                = warn mempty                ≡ return ()
  prop_warn_mappend w1 w2         = warn (w1 `mappend` w2)     ≡ (warn w1 ≫ warn w2)

  prop_clear_warn_return w a      = clear (warn w ≫ return a)  ≡ return (a, w)
  prop_clear_raise e              = clear (raise e)            ≡ raise e
```

(Remember that $m_1 \gg m_2$ is just $m_1 \ggg= \lambda\_ \rightarrow m_2$.)

Problem 1 (8p): (Derived laws for *MonadErrorWarning***)**

By equational reasoning, derive the following additional laws for *MonadErrorWarning*.

$$\begin{aligned}
&\mathit{prop\_handle\_return}\ h\ a &&= \mathit{handle}\ h\ (\mathit{return}\ a) &&\equiv \mathit{return}\ a\\
&\mathit{prop\_handle\_raise\_warn}\ h\ w\ e &&= \mathit{handle}\ h\ (\mathit{raise}\ e \gg \mathit{warn}\ w) &&\equiv h\ e\ \mathit{mempty}
\end{aligned}$$

Reason step by step, only using one law at a time. For each step, state which law you used. Use the laws for *Monoid*, *Monad*, and the original laws for *MonadErrorWarning*. You may (but need not) use the *EquationalReasoning* format from the lecture.

SOLUTION:

$$\begin{aligned}
&\mathit{prop\_handle\_return}\ h\ a = \mathit{proof}\\
&\quad (\mathit{handle}\ h\ (\mathit{return}\ a)) &&\equiv\langle\ \mathit{Thm}\ \texttt{"monad\_left\_unit"} &&\rangle\equiv\\
&\quad (\mathit{handle}\ h\ (\mathit{return}\ () \gg\!= \lambda\_ \to \mathit{return}\ a)) &&\equiv\langle\ \mathit{Def}\ \texttt{">>"} &&\rangle\equiv\\
&\quad (\mathit{handle}\ h\ (\mathit{return}\ () \gg \mathit{return}\ a)) &&\equiv\langle\ \mathit{Thm}\ \texttt{"prop\_warn\_mempty"} &&\rangle\equiv\\
&\quad (\mathit{handle}\ h\ (\mathit{warn}\ \mathit{mempty} \gg \mathit{return}\ a)) &&\equiv\langle\ \mathit{Thm}\ \texttt{"prop\_handle\_warn\_return"} &&\rangle\equiv\\
&\quad (\mathit{warn}\ \mathit{mempty} \gg \mathit{return}\ a) &&\equiv\langle\ \mathit{Thm}\ \texttt{"prop\_warn\_mempty"} &&\rangle\equiv\\
&\quad (\mathit{return}\ () \gg \mathit{return}\ a) &&\equiv\langle\ \mathit{Thm}\ \texttt{"monad\_left\_unit"} &&\rangle\equiv\\
&\quad (\mathit{return}\ a)
\end{aligned}$$

$$\begin{aligned}
&\mathit{prop\_handle\_raise\_warn}\ h\ w\ e = \mathit{proof}\\
&\quad (\mathit{handle}\ h\ (\mathit{raise}\ e \gg \mathit{warn}\ w)) &&\equiv\langle\ \mathit{Thm}\ \texttt{"prop\_bind\_raise"} &&\rangle\equiv\\
&\quad (\mathit{handle}\ h\ (\mathit{raise}\ e)) &&\equiv\langle\ \mathit{Thm}\ \texttt{"monad\_left\_unit"} &&\rangle\equiv\\
&\quad (\mathit{handle}\ h\ (\mathit{return}\ () \gg \mathit{raise}\ e)) &&\equiv\langle\ \mathit{Thm}\ \texttt{"prop\_warn\_mempty"} &&\rangle\equiv\\
&\quad (\mathit{handle}\ h\ (\mathit{warn}\ \mathit{mempty} \gg \mathit{raise}\ e)) &&\equiv\langle\ \mathit{Thm}\ \texttt{"prop\_handle\_warn\_raise"} &&\rangle\equiv\\
&\quad (h\ e\ \mathit{mempty})
\end{aligned}$$

Problem 2 (16p): (A monad implementing *MonadErrorWarning***)**

Implement *MonadErrorWarning* by a suitable type $EW\ e\ w\ a$. Concretely, provide a definition of EW and the following type class instances.

$$\textbf{instance}\ \mathit{Monoid}\ w \Rightarrow \mathit{Monad}\ (EW\ e\ w)$$
$$\textbf{instance}\ \mathit{Monoid}\ w \Rightarrow \mathit{MonadErrorWarning}\ e\ w\ (EW\ e\ w)$$

Also provide a suitable function

$$\mathit{runEW} :: EW\ e\ w\ a \rightarrow \_$$

that allows to run EW-computations, returning the collected warnings and a value of $\mathit{Either}\ e\ a$, containing either the thrown error e or the regular computation result a.

You need not give instances of *Functor* and *Applicative*.

CLARIFICATION: But the code for *return* must be presented.

SOLUTION:

```
newtype EW e w a = EW {runEW :: (Either e a, w)}
instance Monoid w ⇒ Monad (EW e w) where
  return a                  = EW (Right a, mempty)
  EW (Left e, w)   >>= k = EW (Left e, w)
  EW (Right a, w) >>= k = case k a of
    EW (r, w′) → EW (r, w ‘mappend‘ w′)
instance Monoid w ⇒ MonadErrorWarning e w (EW e w) where
  raise e                          = EW (Left e, mempty)
  handle h (EW (Left e, w))  = h e w
  handle h (EW (Right a, w)) = EW (Right a, w)
  warn w                           = EW (Right (), w)
  clear (EW (Left e, w))        = EW (Left e, w)
  clear (EW (Right a, w))      = EW (Right (a, w), mempty)
```

Problem 3 (16p): (Proving some laws for *EW*)
By equational reasoning, derive the following *MonadErrorWarning* laws for *EW*.

a) *prop_bind_raise*

b) *prop_handle_warn_raise*

c) *prop_warn_mappend*

d) *prop_clear_warn_return*

Prove the 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.

You may use the *Monoid* laws and standard Haskell computation laws.

SOLUTION:

```
prop_bind_raise e k = proof
  (raise e >>= k)                                    ≡⟨ Def "raise"               ⟩≡
  (EW (Left e, mempty) >>= k)                        ≡⟨ Def ">>="                 ⟩≡
  (EW (Left e, mempty))                              ≡⟨ Def "raise"               ⟩≡
  (raise e)
prop_handle_warn_raise h w e = proof
  (handle h (warn w >> raise e))                     ≡⟨ Def "warn"                ⟩≡
  (handle h (EW (Right (), w) >> raise e))           ≡⟨ Def "raise"               ⟩≡
  (handle h (EW (Right (), w) >> EW (Left e, empty))) ≡⟨ Def ">>="                ⟩≡
  (handle h (EW (Left e, w `mappend` empty)))        ≡⟨ Thm "monoid_right_unit"   ⟩≡
  (handle h (EW (Left e, w)))                        ≡⟨ Def "handle"              ⟩≡
  (h e w)
prop_warn_mappend w1 w2 = proof
  (warn (w1 `mappend` w2))                           ≡⟨ Def "warn"                ⟩≡
  (EW (Right (), w1 `mappend` w2))                   ≡⟨ Def ">>="                 ⟩≡
  (EW (Right (), w1) >> EW (Right (), w2))           ≡⟨ Def "warn"                ⟩≡
  (warn w1 >> warn w2)
prop_clear_warn_return w a = proof
  (clear (warn w >> return a))                       ≡⟨ Def "warn"                ⟩≡
  (clear (EW (Right (), w) >> return a))             ≡⟨ Def "return"              ⟩≡
  (clear (EW (Right (), w) >> EW (Right a, mempty))) ≡⟨ Def ">>="                 ⟩≡
  (clear (EW (Right a, w `mappend` mempty)))         ≡⟨ Thm "monoid_right_unit"   ⟩≡
  (clear (EW (Right a, w)))                          ≡⟨ Def "clear"               ⟩≡
  (EW (Right (a, w), mempty))                        ≡⟨ Def "return"              ⟩≡
  (return (a, w))
```

Problem 4 (10p): (Interpreter for multiplicative expressions)
Make a little domain-specific language (DSL) for expressions consisting of integer literals, integer multiplication, and integer division.

```
type Exp
eLit  :: Integer → Exp
eMul :: Exp → Exp → Exp
eDiv  :: Exp → Exp → Exp
```

Further, write an interpreter for such expressions that returns an *Integer* and reports fatal and non-fatal issues (errors and warnings).

- Error *DivisionByZero* when division by 0 is attempted during evaluation.
- Warnings *MultiplicationWithOne* and *DivisionByOne* when the *original expression* contains a multiplication with 1 or a division by 1. For example, interpreting either *eMul* (*eLit* 1) *e* or *eMul* *e* (*eLit* 1) or *eDiv* *e* (*eLit* 1) should trigger such a warning. However, e.g. *eMul* (*eLit* 4) (*eDiv* (*eLit* 2) (*eLit* 2)) should *not* trigger the warning, because the multiplication by 1 is not contained in the original expression but only happens during evaluation.

The interpreter should have type

```
eval :: Exp → EW E W Integer
```

for suitably defined types E and W.

CLARIFICATION: It might be better to require a more polymorphic type:

```
eval :: MonadErrorWarning E W m ⇒ Exp → m Integer
```

SOLUTION:

```
data Exp
   = ELit Integer
   | EMul Exp Exp
   | EDiv Exp Exp
  deriving Eq
data Warn
   = MultiplicationWithOne
   | DivisionByOne
data Err
   = DivisionByZero
eLit :: Integer → Exp
eLit = ELit

eMul :: Exp → Exp → Exp
eMul = EMul

eDiv :: Exp → Exp → Exp
eDiv = EDiv

  -- eval :: Exp -¿ EW Err [Warn] Integer
eval :: MonadErrorWarning Err [Warn] m ⇒ Exp → m Integer
eval = λcase
  ELit i → return i
  EMul e1 e2 → do
    when (e1 ≡ ELit 1) $ warn [MultiplicationWithOne]
    when (e2 ≡ ELit 1) $ warn [MultiplicationWithOne]
    i1 ← eval e1
    i2 ← eval e2
    return $ i1 ∗ i2
  EDiv e1 e2 → do
    when (e2 ≡ ELit 1) $ warn [DivisionByOne]
    i1 ← eval e1
    i2 ← eval e2
    when (i2 ≡ 0) $ raise DivisionByZero
    return $ i1 ‘div‘ i2
```

Problem 5 (10p): (Monad transformer)
Recall the concept of a monad transformer:

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a
```

Define a type *EWT e w m a* that adds the *MonadErrorWarning* functionality on top of any *Monad m*. Implement the following instances:

```
instance (Monoid w, Monad m) ⇒ Monad (EWT e w m)
instance (Monoid w, Monad m) ⇒ MonadErrorWarning e w (EWT e w m)
instance (Monoid w)          ⇒ MonadTrans (EWT e w)
```

EWT should satisfy the *Monad* and *MonadErrorWarning* laws albeit you do not need to prove them. Also, you need not give *Functor* or *Applicative* instances.

CLARIFICATION: But the code for *return* must be presented.

SOLUTION:

```
newtype EWT e w m a = EWT {runEWT :: m (Either e a, w)}
instance (Monad m, Monoid w) ⇒ Monad (EWT e w m) where
  return a = EWT $ return (Right a, mempty)
  m ≫= k  = EWT do
    runEWT m ≫= λcase
      (Left e, w)  → return (Left e, w)
      (Right a, w) → do
        (r, w′) ← runEWT (k a)
        return (r, w ‘mappend‘ w′)
instance (Monad m, Monoid w) ⇒ MonadErrorWarning e w (EWT e w m) where
  raise e      = EWT $ return (Left e, mempty)
  handle h m = EWT do
    runEWT m ≫= λcase
      (Left e, w)  → runEWT $ h e w
      (Right a, w) → return (Right a, w)
  warn w       = EWT $ return (Right (), w)
  clear m      = EWT do
    runEWT m ≫= λcase
      (Left e, w)  → return (Left e, w)
      (Right a, w) → return (Right (a, w), mempty)
instance (Monoid w) ⇒ MonadTrans (EWT e w) where
  lift m = EWT do
```

```
    $a \leftarrow m$
    $return\ (Right\ a, mempty)$
```