Chalmers | Göteborgs Universitet
Andreas Abel, Computer Science and Engineering

Advanced Functional Programming TDA342/DIT260

Thursday, 19 March 2026, 14:00 - 18:00, M, Johanneberg

(including example solutions to programming problems)

- Examiner: Andreas Abel (+46-31-772-1731) visits 15:00 and 17:00.
- 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.

Problem 1: (15p) (**Monads via** *join*)

Recall the *Monad* class from the course.

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

We consider an alternative definition of *Monad* using *join* instead of (>>=).

```
class Functor m ⇒ Monad2 m where
  return2 :: a → m a
  join2   :: m (m a) → m a
```

▶ **Task A (3p):** Without using **do**-notation, implement the bind function for any *Monad2 m*.

```
bind2 :: Monad2 m ⇒ m a → (a → m b) → m b
bind2 = ?
```

Hint: Every *Monad2* is also a *Functor*!

SOLUTION:

```
bind2       :: Monad2 m ⇒ m a → (a → m b) → m b
bind2 m k = join2 (fmap k m)
```

▶ **Task B (3p):** Implement *join* using the standard *Monad* methods. Again, do not use **do**-notation.

```
join :: Monad m ⇒ m (m a) → m a
join = ?
```

SOLUTION:

```
join     :: Monad m ⇒ m (m a) → m a
join m = m >>= id
```

If *Monad2* was how monads were defined in Haskell, the instances would also look a bit different. Let's explore how in the next tasks.

▶ **Task C (3p):** Finish this *Maybe* instance of *Monad2* by implementing *join2*:

```
instance Monad2 Maybe where
  return2 = Just
  join2   =  ?
```

Do it in this setting's most straightforward way (i.e., do not go via an implementation of *bind2*).

SOLUTION:

```
instance Monad2 Maybe where
  return2 a                = Just a
  join2 (Just (Just a)) = Just a
  join2 _                  = Nothing
```

Alternative is $join2 = maybe\ Nothing\ id$:

```
join2 (Just a) = a
join2 Nothing = Nothing
```

▶ **Task D (6p):** Finish this *State* instance of *Monad2* by implementing *fmap* and *join2*:

```
newtype State s a = State {runState :: s → (a, s)}


instance Functor (State s) where
  fmap = ?
instance Monad2 (State s) where
  return2 a = State λs → (a, s)
  join2     = ?
```

SOLUTION:

```
instance Functor (State s) where
  fmap f m = State λs → let
       (a,  s') = runState m s
    in (f a, s')
instance Monad2 (State s) where
  return2 a  = State λs → (a, s)
  join2   m = State λs → let
       (m', s') = runState m s
    in runState m' s'
```

Problem 2 (18p) (**Monad laws in terms of** *join*)
This problem continues the previous problem's adventure about monads in terms of *join*.

Note: Even if you have not solved problem 1 you can still try to solve this problem.

The monad laws can also be expressed in terms of *join*, *fmap* and *return*:

- Outer unit: $join \circ return = id$
- Inner unit: $join \circ fmap\ return = id$
- Associativity: $join \circ fmap\ join = join \circ join$

▶ **Task A (4p):** What is the type of *fmap return* at the use site in the inner unit law and what is the type of the rightmost *join* in the associativity law?

SOLUTION: Typing:

1. For the outer unit law, we have $return :: m\ a \to m\ (m\ a)$ so that $join \circ return$ has type $m\ a \to m\ a$.
2. For the inner unit law, we have $return :: a \to m\ a$ and $\underline{fmap\ return :: m\ a \to m\ (m\ a)}$ so that $join \circ fmap\ return$ has type $m\ a \to m\ a$.
3. For the associativity law, we have on the lhs $fmap\ join :: m\ (m\ (m\ a)) \to m\ (m\ a)$ so the equation has type $m\ (m\ (m\ a)) \to m\ a$. Thus, the rightmost occurence of *join* has type $\underline{m\ (m\ (m\ a)) \to m\ (m\ a)}$.

Now, consider this implementation of the writer monad:

```
data Writer s a = W s a
  deriving (Eq, Show)
instance Functor (Writer s) where
  fmap f (W s a) = W s (f a)
instance Monoid s ⇒ Monad2 (Writer s) where
  return2 a = W mempty a
  join2 (W s (W s' a)) = W (mappend s s') a
```

▶ **Task B (14p):** Show the above unit and associativity laws for this writer monad (*fmap*, *return2*, *join2*) by 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.
Hint: You may prove these laws pointwise, i.e., instead of proving an equation $f = g$ between functions $f, g :: a \to b$, you may prove the equation $f\ x = g\ x$ for an arbitrary $x :: a$.

SOLUTION: Proofs (4p + 4p + 6p):

$$
\begin{array}{l}
prop\_outer\_unit :: (Monoid\ s, Eq\ s, Eq\ a) \Rightarrow Writer\ s\ a \to Proof\ (Writer\ s\ a) \\
prop\_outer\_unit\ (W\ s\ a) = proof \\
\quad (join2\ (return2\ (W\ s\ a))) \quad \equiv\langle\ Def\ \texttt{"return2"} \quad \rangle\equiv \\
\quad (join2\ (W\ mempty\ (W\ s\ a))) \equiv\langle\ Def\ \texttt{"join2"} \quad \rangle\equiv \\
\quad (W\ (mappend\ mempty\ s)\ a) \quad \equiv\langle\ Thm\ \texttt{"monoid unit"} \rangle\equiv \\
\quad (W\ s\ a) \\
prop\_inner\_unit :: (Monoid\ s, Eq\ s, Eq\ a) \Rightarrow Writer\ s\ a \to Proof\ (Writer\ s\ a) \\
prop\_inner\_unit\ (W\ s\ a) = proof \\
\quad (join2\ (fmap\ return2\ (W\ s\ a))) \equiv\langle\ Def\ \texttt{"fmap"} \quad \rangle\equiv \\
\quad (join2\ (W\ s\ (return2\ a))) \quad \equiv\langle\ Def\ \texttt{"return2"} \quad \rangle\equiv \\
\quad (join2\ (W\ s\ (W\ mempty\ a))) \quad \equiv\langle\ Def\ \texttt{"join2"} \quad \rangle\equiv \\
\quad (W\ (mappend\ s\ mempty)\ a) \quad \equiv\langle\ Thm\ \texttt{"monoid unit"} \rangle\equiv \\
\quad (W\ s\ a) \\
prop\_assoc :: (Monoid\ s, Eq\ s, Eq\ a) \\
\quad \Rightarrow Writer\ s\ (Writer\ s\ (Writer\ s\ a)) \to Proof\ (Writer\ s\ a) \\
prop\_assoc\ (W\ s1\ (W\ s2\ (W\ s3\ a))) = proof \\
\quad (join2\ (fmap\ join2\ (W\ s1\ (W\ s2\ (W\ s3\ a))))) \equiv\langle\ Def\ \texttt{"fmap"} \quad \rangle\equiv \\
\quad (join2\ (W\ s1\ (join2\ (W\ s2\ (W\ s3\ a))))) \quad \equiv\langle\ Def\ \texttt{"join2"} \quad \rangle\equiv \\
\quad (join2\ (W\ s1\ (W\ (mappend\ s2\ s3)\ a))) \quad \equiv\langle\ Def\ \texttt{"join2"} \quad \rangle\equiv \\
\quad (W\ (mappend\ s1\ (mappend\ s2\ s3))\ a) \quad \equiv\langle\ Thm\ \texttt{"monoid assoc"} \rangle\equiv \\
\quad (W\ (mappend\ (mappend\ s1\ s2)\ s3)\ a) \quad \equiv\langle\ Def\ \texttt{"join2"} \quad \rangle\equiv \\
\quad (join2\ (W\ (mappend\ s1\ s2)\ (W\ s3\ a))) \quad \equiv\langle\ Def\ \texttt{"join2"} \quad \rangle\equiv \\
\quad (join2\ (join2\ (W\ s1\ (W\ s2\ (W\ s3\ a))))))
\end{array}
$$

Problem 3 (15p): (**Well-typed expressions via GADTs**)

Consider the program below for typing and evaluation of arithmetic and boolean expressions, adapted from lecture 13 in Spring 2026.

```
data Exp where                                  -- Untyped terms
  EInt  :: Int → Exp
  EBool :: Bool → Exp
  EPlus :: Exp → Exp → Exp
  EEq   :: Exp → Exp → Exp
  deriving Show

data Tm a where                                 -- Terms indexed by their Haskell type
  TLit  :: a → Tm a
  TPlus :: Tm Int → Tm Int → Tm Int
  TEq   :: (Eq a, Show a) ⇒ Tm a → Tm a → Tm Bool
deriving instance Show a ⇒ Show (Tm a)

eval :: Tm a → a                                -- Typed interpreter
eval (TLit a)       = a
eval (TPlus t1 t2) = eval t1 + eval t2
eval (TEq t1 t2)    = eval t1 ≡ eval t2

data Ty a where                                 -- Value representation of Haskell types
  TInt  :: Ty Int
  TBool :: Ty Bool
deriving instance Show (Ty a)

data TypedTm where                              -- Well-typed terms of some type
  (:::)   :: (Eq a, Show a) ⇒ Tm a → Ty a → TypedTm

data Equal a b where                            -- Equality type
  Refl  :: Equal a a

equal :: Ty a → Ty b → Maybe (Equal a b)        -- Checking equality of represented types
equal TInt  TInt  = Just Refl
equal TBool TBool = Just Refl
equal _     _     = Nothing

infer :: Exp → Maybe TypedTm                    -- Type inference
infer (EInt i)      = do return (TLit i ::: TInt)
infer (EBool b)     = do return (TLit b ::: TBool)
infer (EPlus e1 e2) = do t1 ← check e1 TInt
                         t2 ← check e2 TInt
                         return (TPlus t1 t2 ::: TInt)
infer (EEq e1 e2)   = do t1 ::: a ← infer e1
                         t2       ← check e2 a
                         return (TEq t1 t2 ::: TBool)

check :: Exp → Ty a → Maybe (Tm a)              -- Type checking
check e a           = do t ::: b  ← infer e
                         Refl     ← equal b a
                         return t
```

▶ **Task:** Extend the program by polymorphic if-then-else expressions. This amounts to adding new constructors to the data types (e.g., *EIf* to *Exp*) and updating the code where needed (e.g., a case for *EIf* in *infer*). Do not repeat existing lines of code unless they need to be modified.

SOLUTION: Add these constructors and function clauses (2p + 3p + 3p + 7p).

```
data Exp where
  EIf :: Exp → Exp → Exp → Exp
data Tm a where
  TIf :: Tm Bool → Tm a → Tm a → Tm a
eval (TIf t t1 t2) = if eval t then eval t1 else eval t2
infer (EIf e e1 e2) = do
  t       ← check e TBool
  t1 ::: a ← infer e1
  t2      ← check e2 a
  return (TIf t t1 t2 ::: a)
```

Problem 4 (12p): (**DSL for polynomials**)

Polynomials p with evaluation $p(x)$ and differentiation Dp can be succinctly defined via this table:

Form	Syntax p	Evaluation $p(x)$	Derivative Dp
Constant	Ka	a	$K0$
Diagonal	I	x	$K1$
Sum	$p_1 + p_2$	$p_1(x) + p_2(x)$	$Dp_1 + Dp_2$
Product	$p_1 * p_2$	$p_1(x) * p_2(x)$	$Dp_1 * p_2 + p_1 * Dp_2$

Herein $0, 1, a$ are drawn from the underlying ring, e.g., the integers or the reals, and $+$ and $*$ are overloaded as both ring operations and operations on polynomials.

We want to represent polynomials as elements of a type $P\ a$ where a is the underlying ring.

▶ **Task A (3p):** Write an API for polynomials that includes the constructions and operations given by the table. You can ignore type class constraints on a.

SOLUTION:

```
pVar          :: P a
pConst        :: a → P a
pAdd, pMul    :: P a → P a → P a
eval          :: P a → a → a
derive        :: P a → P a
```

▶ **Task B (9p):** Choose a representation of P and implement the API. (Again, you can ignore type class constraints on a.) Make sure your constructors implement simple optimizations when constants are involved, e.g. $K\ 0 * p$ should have the same representation as p.

Write a few sentences why you chose your particular representation of P.

SOLUTION: Alternative 1: Implementation as trees $P = Poly$. This implementation is basically already given in the table, so it is effortless. However, *derive* has a bad complexity for the case *derive* $(PMul\ p1\ p2)$ because it essentially duplicates $p1$ and $p2$.

```
data Poly a where
  PVar   :: Poly a
  PConst :: a → Poly a
  PAdd   :: Poly a → Poly a → Poly a
  PMul   :: Poly a → Poly a → Poly a
  deriving (Eq, Show)

pVar :: Poly a
pVar = PVar

pConst :: a → Poly a
pConst = PConst

  -- Smart addition, considering the zero cases.
pAdd :: (Eq a, Num a) ⇒ Poly a → Poly a → Poly a
pAdd (PConst a) (PConst b) = PConst (a + b)
pAdd (PConst 0) q = q
pAdd p (PConst 0) = p
pAdd p q = PAdd p q

  -- Smart multiplication, considering zero and unit cases.
pMul :: (Eq a, Num a) ⇒ Poly a → Poly a → Poly a
pMul (PConst a) (PConst b) = PConst (a * b)
pMul (PConst 0) _ = PConst 0
pMul (PConst 1) q = q
pMul _ (PConst 0) = PConst 0
pMul p (PConst 1) = p
pMul p q = PMul p q

peval :: Num a ⇒ Poly a → a → a
peval = λcase
  PVar        → λx → x
  PConst a    → λx → a
  PAdd p1 p2  → λx → peval p1 x + peval p2 x
  PMul p1 p2  → λx → peval p1 x * peval p2 x

pderive :: (Eq a, Num a) ⇒ Poly a → Poly a
pderive = λcase
  PVar        → PConst 1
  PConst _    → PConst 0
  PAdd p1 p2  → pderive p1 `pAdd` pderive p2
  PMul p1 p2  → (pderive p1 `pMul` p2) `pAdd` (p1 `pMul` pderive p2)
```

Alternative 2: Implementation $P = L$ as lists of coefficients $k_0, ..., k_{n-1}$ of coefficients with usual evaluation $\sum_{i=0..n-1} k_i * x^i$. Derivation can be done in linear time, but multiplication is $O(nm)$ for polynomials of degrees n and m, resp. We use a smart constructor *cons* to prevent trailing zero coefficients, which makes the representation canonical.

```
type L a = [a]

  -- Smart list constructor preventing trailing zeros.
cons :: (Eq a, Num a) => a -> L a -> L a
cons 0 [] = []
cons a as = a : as

lvar :: Num a => L a
lvar = [0, 1]

lconst :: (Eq a, Num a) => a -> L a
lconst a
  | a == 0    = []
  | True      = [a]

ladd :: (Eq a, Num a) => L a -> L a -> L a
ladd p       []       = p
ladd []      q        = q
ladd (a : p) (b : q)  = cons (a + b) (ladd p q)

lmul :: (Eq a, Num a) => L a -> L a -> L a
lmul [] _ = []
lmul p q = lmul' p q
  where
      -- p is non-empty here so we won't introduce trailing zeros
    lmul' p []      = []
    lmul' p (0 : q) = lmul' (0 : p) q
    lmul' p (k : q) = map (k*) p `ladd` lmul' (0 : p) q

leval :: Num a => L a -> a -> a
leval []      x = 0
leval (k : p) x = k + x * leval p x   -- Horner's method

lderive :: (Enum a, Num a) => L a -> L a
lderive []      = []
lderive (_ : p) = zipWith (*) p [1..]

  -- Remove trailing zeros (should never be necessary)
lnorm :: (Eq a, Num a) => L a -> L a
lnorm = foldr cons []
```

Alternative 3: Use a sparse list $(d_1, k_1), ..., (d_n, k_n)$ of non-zero coefficients with evaluation $\sum_{i=1..n} k_i * x^{d_i}$. This improves on the "list of coefficient" implementation in that multiplication is $O(nm)$ where n and m are the number of non-zero coefficients for the two polynomials. Depending on the application, this can perform vastly better. Implementation is easy using finite maps from degrees to coefficients $P = M$, but it can also be done with lists.

```
type M a = Map Integer a

mvar          :: Num a => M a
mvar          = Map.singleton 1 1

mconst        :: (Eq a, Num a) => a -> M a
mconst k
  | k == 0    = Map.empty
  | True      = Map.singleton 0 k

madd          :: (Eq a, Num a) => M a -> M a -> M a
madd m1 m2    = mnorm $ Map.unionWith (+) m1 m2

mmul          :: (Eq a, Num a) => M a -> M a -> M a
mmul m1 m2    = mnorm $ Map.fromListWith (+)
  [ (d1 + d2, k1 * k2)
  | (d1, k1) <- Map.toList m1
  , (d2, k2) <- Map.toList m2
  ]

meval         :: Num a => M a -> a -> a
meval m x     = Map.foldrWithKey (\d k acc -> k * (x ^ d) + acc) 0 m

mderive       :: Num a => M a -> M a
mderive       = Map.mapKeysMonotonic (\d -> d - 1)
              . Map.mapWithKey (\d k -> fromIntegral d * k)
              . Map.delete 0

  -- Auxiliary: Normalization: remove entries with coefficient 0
mnorm         :: (Eq a, Num a) => M a -> M a
mnorm         = Map.filter (/= 0)
```

Other solutions are possible...