Domain Specific Languages of Mathematics
Course codes: DAT326 / DIT982

Patrik Jansson

2019-03-19

Contact	Maximilian Algehed, Abhiroop Sarkar, Patrik Jansson (x5415)
Results	Announced within 19 days
Exam check	2019-04-09 in EDIT 5468 at 12.15-12.45
Aids	One textbook of your choice (e.g., Beta - Mathematics Handbook, or Rudin, or Adams and Essex). No printouts, no lecture notes, no notebooks, etc.
Grades	To pass you need **a minimum of 5p on each question (1 to 4)** and also reach these grade limits: 3: >=48p, 4: >=65p, 5: >=83p, max: 100p

Remember to write legibly. Good luck!

For reference: the learning outcomes. Some are tested by the hand-ins, some by the written exam.

- Knowledge and understanding
  - design and implement a DSL (Domain Specific Language) for a new domain
  - organize areas of mathematics in DSL terms
  - explain main concepts of elementary real and complex analysis, algebra, and linear algebra
- Skills and abilities
  - develop adequate notation for mathematical concepts
  - perform calculational proofs
  - use power series for solving differential equations
  - use Laplace transforms for solving differential equations
- Judgement and approach
  - discuss and compare different software implementations of mathematical concepts

1. [15p] Consider the following quote (THEOREM 6 from Adams and Essex 2010):

   The Chain Rule
   If $f(u)$ is differentiable at $u = g(x)$, and $g(x)$ is differentiable at x, then the composite function $f \circ g(x) = f(g(x))$ is differentiable at x, and

   $$(f \circ g)'(x) = f'(g(x))g'(x).$$

   (a) [10p] Give the types of the symbols involved: $f, u, g, x, (\circ)$ and D, where $(D\ f = f')$.

   (b) [5p] A *point-free* definition of a function does not mention the actual arguments it will be applied to. Give a point-free definition of the Chain Rule: $D\ (f \circ g) = ...$ You can use the "lifted" numeric operations (*FunNumInst*):

   ```
   instance Num a ⇒ Num (x → a) where (+) = addF; (∗) = mulF;    -- ...
   addF, mulF :: Num a ⇒ (x → a) → (x → a) → (x → a)
   addF = liftOp (+)   -- a point-free definition of "lifted +"
   mulF = liftOp (∗)   -- ... and lifted ∗
   liftOp :: (a → b → c) → (x → a) → (x → b) → (x → c)
   liftOp op f g = λx → op (f x) (g x)
   ```

---

2. [30p] Calculational proof of syntactic differentiation.

   Consider the following DSL for functional expressions:

   ```
   data FunExp where
     (:+:) :: FunExp → FunExp → FunExp
     (:∗:) :: FunExp → FunExp → FunExp
     (:∘:) :: FunExp → FunExp → FunExp
     Exp   :: FunExp
     deriving Show
   ```

   The intended meaning (the semantics) of elements of the *FunExp* type is functions:

   ```
   type Func = ℝ → ℝ
   eval :: FunExp    → Func
   eval  (e1 :+: e2) = eval e1 + eval e2   -- note the use of "lifted +",
   eval  (e1 :∗: e2) = eval e1 ∗ eval e2   -- note the use of "lifted ∗",
   eval  (e1 :∘: e2) = eval e1 ∘ eval e2
   eval  Exp         = exp
   ```

   On the semantics side we write $D : Func \to Func$ to denote computing the derivative. Your task is to calculate and implement the function $derive :: FunExp \to FunExp$ on the syntactic side. It is specified by $eval\ (derive\ e) == D\ (eval\ e)$ for all $e :: FunExp$.

   (a) [10p] Starting with the specification, simplify $eval\ (derive\ (f :∗: g))$ step by step until you reach the form $eval\ dfg$ for some expression $dfg :: FunExp$ suitable for the definition of $derive\ (f :∗: g) = dfg$. Briefly motivate each step of the calculation.

   (b) [15p] Do the same for $eval\ (derive\ (f :∘: g))$.

   (c) [5p] Use the results from the calculations to implement *derive*.

3. [30p] Algebraic structure: Group (lightly edited from the Wikipedia entry)

   A group is a set, G, together with an operation $\cdot$ (called the group law of G) that combines any two elements a and b to form another element, denoted $a \cdot b$ or ab. To qualify as a group, the set and operation, $(G, \cdot)$, must satisfy four requirements known as the group axioms:

   - Closure: For all a, b in G, the result of the operation, $a \cdot b$, is also in G.
   - Associativity: For all a, b and c in G, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
   - Identity element: There exists an element e in G such that, for every element a in G, the equation $e \cdot a = a \cdot e = a$ holds.
   - Inverse element: For each a in G, there exists an element b in G, commonly denoted a^{-1}, such that $a \cdot b = b \cdot a = e$, where e is the identity element.

   (a) Define a type class *Group* that corresponds to the group structure.

   (b) Define a datatype $G\ v$ for the language of group expressions (with variables of type v) and define a *Group* instance for it. (These are expressions formed from applying the group operations to the appropriate number of arguments, e.g., all the left hand sides and right hand sides of the above equations.)

   (c) Find and implement two other instances of the *Group* class.

   (d) Give a type signature for, and define, a general evaluator for $G\ v$ expressions on the basis of an assignment function.

   (e) Specialise the evaluator to the two *Group* instances defined in (3c). Take three group expressions of type G *String*, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

---

4. [25p] Consider the following differential equation:

$$2f(x) + 3f'(x) + f''(x) = 2e^{-3x}, \quad f(0) = -2, \quad f'(0) = 0$$

   (a) [10p] Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *integ* and the differential equation to express the relation between *fs*, *fs*$'$, *fs*$''$, and the power series *rhs* for the right hand side. What are the first four coefficients of *fs*?

   (b) [15p] Solve the equation using the Laplace transform. You should need this formula (and the rules for linearity + derivative):

$$\mathscr{L}\,(\lambda t.\, e^{\alpha * t})\, s = 1/(s - \alpha)$$

   Show that your solution does indeed satisfy the three requirements.