Domain Specific Languages of Mathematics
Course codes: DAT326 / DIT982

Patrik Jansson

2018-08-28

Contact Sólrún Einarsdóttir, Patrik Jansson (x5415)

Results Announced within 19 days

Exam check Fri. 2018-09-06 in EDIT 5468 at 12.30-12.55

Aids One textbook of your choice (e.g., Adams and Essex, or Rudin, or Beta - Mathematics Handbook). No printouts, no lecture notes, no notebooks, etc.

Grades 3: 40p, 4: 60p, 5: 80p, max: 100p

Remember to write legibly. Good luck!

For reference: the DSLsofMath 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. [30pts] Algebraic structure: Consider the Wikipedia definition of a *field*:

   > Formally, a field is a set F together with two operations called addition and multiplication. [...] These operations are required to satisfy the following properties, referred to as field axioms. In the following definitions, a, b and c are arbitrary elements of the field F.
   >
   > - Associativity of addition and multiplication: $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
   > - Commutativity of addition and multiplication: $a + b = b + a$ and $a \cdot b = b \cdot a$.
   > - Additive and multiplicative identity: there exist two different elements 0 and 1 in F such that $a + 0 = a$ and $a \cdot 1 = a$.
   > - Additive inverses: for every a in F, there exists an element in F, denoted $-a$, called additive inverse of a, such that $a + (-a) = 0$.
   > - Multiplicative inverses: for every $a \neq 0$ in F, there exists an element in F, denoted by a^{-1}, $1/a$, or $\frac{1}{a}$, called the multiplicative inverse of a, such that $a \cdot a^{-1} = 1$.
   > - Distributivity of multiplication over addition: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

   (a) Define a type class *Field* that corresponds to the field structure (with operations *add*, *mul*, *zero*, *one*, *negate*, *recip*).

   (b) Define a datatype $F\ v$ for the language of field expressions (with variables of type v) and define a *Field* instance for it. (These are expressions formed from applying the field 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 *Field* class.

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

   (e) Specialise the evaluator to the two *Field* instances defined in (1c). Take three field expressions of type F *String*, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

---

2. [20pts] Consider the following quote from Armitage & Griffiths, p. 158:

   > Let $f : G \to \mathbb{R}^n$ be differentiable at $\boldsymbol{a} \in G \subset \mathbb{R}^n$. If $d_{\boldsymbol{a}} f : \mathbb{R}^n \to \mathbb{R}^n$ is a bijection, then there exist neighbourhoods $U_h(\boldsymbol{a}) \subset G$, $V_k(f(\boldsymbol{a})) \subset \mathbb{R}^n$, and a function $g : V_k \to U_h$ such that g is differentiable at $f(\boldsymbol{a})$; and for all $\boldsymbol{v} \in V_k$, $\boldsymbol{u} \in U_h$,
   >
   > $$g(f(\boldsymbol{u})) = \boldsymbol{u}, \qquad f(g(\boldsymbol{v})) = \boldsymbol{v}$$

   Remark: In the context of the quote, $d_{\boldsymbol{a}} f$ denotes the differential of f in $\boldsymbol{a}$. The notation $U_r(x)$ is used to denote the set of points that are within a distance less than r to x ($r > 0$).

   (a) What is the type of d?

   (b) What are the types of h and k?

   (c) What are the types of U and V?

   (d) There seems to be an inconsistency between the types of U and V when they are introduced and their use in the following (in the types of $g, \boldsymbol{u}, \boldsymbol{v}$). What is the inconsistency? How can you fix it?

---

3. [20pts] Consider the following differential equation:

$$f(x) - 4f''(x) = e^x, \quad f(0) = 0, \quad f'(0) = -\frac{1}{2}$$

(a) 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', and fs''. What are the first four coefficients of fs?

(b) 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.

---

4. [30pts] Calculational proof of syntactic differentiation.

Consider the following DSL for functional expressions:

```
data FunExp = Const Double | Id | Exp FunExp
            | FunExp :+: FunExp | FunExp :*: FunExp
            deriving Show
```

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

```
type Func = ℝ → ℝ
eval :: FunExp       → Func
eval   (Const α)   = const α
eval   Id          = id
eval   (e₁ :+: e₂) = eval e₁ + eval e₂   -- note the use of "lifted +",
eval   (e₁ :*: e₂) = eval e₁ * eval e₂   -- "lifted *",
eval   (Exp e₁)    = exp (eval e₁)       -- and "lifted exp".
```

where the "lifted" operations are defined as follows:

```
instance Num a ⇒ Num (x → a) where
  f + g = λx → f x + g x   -- lifted +
  f * g  = λx → f x * g x   -- lifted *
instance Floating a ⇒ Floating (x → a) where
  exp f = exp ∘ f           -- lifted exp
```

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

(a) Starting with the specification, simplify $\mathit{eval}\ (\mathit{derive}\ (\mathit{Exp}\ e))$ step by step until you reach the form $\mathit{eval}\ \mathit{fe}$ for some expression $\mathit{fe} :: \mathit{FunExp}$ suitable for the definition of $\mathit{derive}\ (\mathit{Exp}\ e) = \mathit{fe}$. Briefly motivate each step of the calculation.

(b) Do the same for $\mathit{eval}\ (\mathit{derive}\ (f :\!*\!: g))$.

(c) Implement *derive*. You don't need to calculate the other cases.

Note: The subquestions do not need to be solved in order.