Lessons learnt from formalizing theoretical computer science

Martin Dvorak

March 12, 2024

What is Lean

Lean 4 is a powerful programming language
Emphasis on formal verification
Type system based on the Calculus of Inductive Constructions
Large library of formally-verified mathematics (over 10^5 lemmas)

Showcase in VS Code

Showcase in VS Code

The rest is all about my mistakes

Showcase in VS Code

The rest is all about my mistakes (and what I learnt from them

Showcase in VS Code

The rest is all about my mistakes (and what I learnt from them (sometimes))

Trying to prove a statement that doesn't hold

Trying to prove a statement that doesn't hold

Yeah, duh!

Grammars are closed under concatenation

Grammars are closed under concatenation

Construction:

$$
\begin{aligned}
G_1 &= (N_1, T, P_1, S_1) \\
G_2 &= (N_2, T, P_2, S_2) \\
G &= (N_1 \cup N_2 \cup \{S\},\ T,\ P_1 \cup P_2 \cup \{S \to S_1 S_2\},\ S)
\end{aligned}
$$

Grammars are closed under concatenation

Construction:

$$G_1 = (N_1, T, P_1, S_1)$$
$$G_2 = (N_2, T, P_2, S_2)$$
$$G = (N_1 \cup N_2 \cup \{S\},\ T,\ P_1 \cup P_2 \cup \{S \to S_1 S_2\},\ S)$$

Counterexample:

$$P_1 = \{S_1 \to S_1 a,\ S_1 \to \epsilon\}$$
$$P_2 = \{S_2 \to S_2 a,\ S_2 \to \epsilon,\ aS_2 \to b\}$$

Grammars are closed under concatenation

Construction:

$$
\begin{aligned}
G_1 &= (N_1, T, P_1, S_1) \\
G_2 &= (N_2, T, P_2, S_2) \\
G &= (N_1 \cup N_2 \cup \{S\},\ T,\ P_1 \cup P_2 \cup \{S \to S_1 S_2\},\ S)
\end{aligned}
$$

Counterexample:

$$
\begin{aligned}
P_1 &= \{S_1 \to S_1 a,\ S_1 \to \epsilon\} \\
P_2 &= \{S_2 \to S_2 a,\ S_2 \to \epsilon,\ aS_2 \to b\}
\end{aligned}
$$

We get:

$$L_1 = L_2 = \{a^n \mid n \in \mathbb{N}_0\} = L_1 L_2$$

Grammars are closed under concatenation

Construction:

$$G_1 = (N_1, T, P_1, S_1)$$
$$G_2 = (N_2, T, P_2, S_2)$$
$$G = (N_1 \cup N_2 \cup \{S\},\ T,\ P_1 \cup P_2 \cup \{S \to S_1 S_2\},\ S)$$

Counterexample:

$$P_1 = \{S_1 \to S_1 a,\ S_1 \to \epsilon\}$$
$$P_2 = \{S_2 \to S_2 a,\ S_2 \to \epsilon,\ aS_2 \to b\}$$

We get:

$$L_1 = L_2 = \{a^n \mid n \in \mathbb{N}_0\} = L_1 L_2$$

However:

$$S \Rightarrow S_1 S_2 \Rightarrow S_1 a S_2 \Rightarrow S_1 b \Rightarrow b$$

Carelessly assuming union of assumptions

```
[OrderedCancelAddCommMonoid C]
lemma Function.HasMaxCutProperty.
    forbids_commutativeFractionalPolymorphism

[OrderedAddCommMonoidWithInfima C]
lemma FractionalOperation.
    IsFractionalPolymorphismFor.
    expressivePowerVCSP

[OrderedCancelAddCommMonoidWithInfima C]
theorem ValuedCSP.CanExpressMaxCut.
    forbids_commutativeFractionalPolymorphism
```

The two latter classes extend `[CompleteSemilatticeInf C]`.

Carelessly assuming union of assumptions

```
[OrderedCancelAddCommMonoid C]
lemma Function.HasMaxCutProperty.
    forbids_commutativeFractionalPolymorphism

[OrderedAddCommMonoidWithInfima C]
lemma FractionalOperation.
    IsFractionalPolymorphismFor.
    expressivePowerVCSP

[OrderedCancelAddCommMonoidWithInfima C]
theorem ValuedCSP.CanExpressMaxCut.
    forbids_commutativeFractionalPolymorphism
```

The two latter classes extend `[CompleteSemilatticeInf C]`.
The assumption is too strong!

```
(∃ x y : C, x < y) → False
```

Not taking time to develop good notation

We write $A\,x$ using a function `Matrix.mulVec`

```
A.mulVec x
```

We write $x^\top A$ using a function `Matrix.vecMul`

```
Matrix.vecMul x A
```

Not taking time to develop good notation

We write $A\,x$ using a function `Matrix.mulVec`

```
A.mulVec x
```

We write $x^\top A$ using a function `Matrix.vecMul`

```
Matrix.vecMul x A
```

Now we have infix operators $*_v$ and ${}_v*$

```
A *ᵥ x
x ᵥ* A
```

Not factoring out useful lemmas

```
lemma {x1 x2 z1 z2 : List T} {a1 a2 : T}
    (notin_x : a2 ∉ x1) (notin_z : a2 ∉ z1) :
  (x1 ++ [a1] ++ z1) = (x2 ++ [a2] ++ z2) ⟺
    (x1 = x2) ∧ (a1 = a2) ∧ (z1 = z2)
```

Reïnventing the wheel

Writing an existing definition from scratch

`List.count`

Reïnventing the wheel

Writing an existing definition from scratch

`List.count`

Developing lemmas about it

`List.count_eq_zero`

Reïnventing the wheel

Writing an existing definition from scratch

`List.count`

Developing lemmas about it

`List.count_eq_zero`

Not knowing existing theorems

`Classical.choose_spec`
`Finset.prod_erase_eq_div`

Using too many “collection types”

```
Fin n → T
Array
List
Multiset
Finset
Fintype
```