Lean

ashvni.n20

July 2020

Contents

1 Introduction

I am going to describe how I defined discrete valuation rings in Lean. The real challenge lies in making Lean understand what may seem mathematically not too difficult to write on paper.

2 Brief introduction to Lean for mathematicians

Lean is an interactive theorem prover. This means that, based on a set of axioms(Peano's axioms), one can verify a proof of almost all theorems, using logic. To some extent, Lean is also an automated theorem prover, that is, it can construct proofs. The Lean project was launched by Leonardo de Moura at Microsoft Research Redmond in 2012. It is an ongoing, long-term effort, and much of the potential for automation will be realized only gradually over time. Lean is released under the Apache 2.0 license, a permissive open source license that permits others to use and extend the code and mathematical libraries freely.

2.1 Type

Let us first understand what types are. Types are essentially sets, with the exception that a given element can belong to only one type. For example, a natural number, say n, is of type $\mathbb{N}$. It cannot be of type $\mathbb{Z}$. One can build types from existing types, such as $\mathbb{N} \times \mathbb{N}$, or $\mathbb{N} \to \mathbb{Z}$.

Unless specified, Lean automatically infers the type of anything that is defined. For example,

```
keywordcolorlet blackz symbolcolor:= 0,
 commentcolor--commentcolor commentcolorzcommentcolor
    commentcolor:commentcolor commentcolorℕcommentcolor
    commentcolorsymbolcolor:=commentcolor commentcolor0
```

We shall discuss the let tactic in a while. Over here, I am using it to (locally) define z to be 0. The – can be used to put a comment line in the code. Alternatively, one may use /- and -/ at the beginning and end of the comment. I have commented the output that Lean shows. Lean infers that z has type $\mathbb{N}$.

2.2 Definition

The command to make a definition is anti climactically definition. A definition looks like :

```
keywordcolordefinition blackname [blackimplicit blackargument] {
    blackexplicit blackinput} : sortcolorType symbolcolor:=
    blackstatement
```

The square brackets are put around implicit arguments, while round brackets are used for arguments that are to be explicitly provided. One may choose to not give the type explicitly, in which case Lean infers it.

Assume that a field has been defined. As an example, we have :

```
keywordcolordefinition blackval symbolcolor:= 0
#keywordcolorcheck blackval commentcolor--commentcolor
    commentcolorvalcommentcolor commentcolor:commentcolor
    commentcolorℕ

keywordcolordefinition (blackx : ℕ) symbolcolor:= 0
#keywordcolorcheck blackval commentcolor--commentcolor
    commentcolorvalcommentcolor commentcolor:commentcolor
    commentcolorℕcommentcolor commentcolor→commentcolor
    commentcolorℕ

keywordcolordefinition blackval (blackK: sortcolorType
    symbolcolor*) [blackfield blackK] : blackK symbolcolor:= 0
#keywordcolorcheck blackval commentcolor--commentcolor
    commentcolorvalcommentcolor commentcolor:commentcolor
    commentcolorK

keywordcolordefinition blackval (blackx :ℕ) symbolcolor:= 0
#keywordcolorcheck blackval commentcolor--commentcolor
    commentcolorvalcommentcolor commentcolor:commentcolor
    commentcolorℕcommentcolor commentcolor→commentcolor
    commentcolorℕ

keywordcolordefinition blackval (blackK: sortcolorType
    symbolcolor*) [blackfield blackK] (blackx :ℕ) : blackK
    symbolcolor:= 0
#keywordcolorcheck blackval commentcolor--commentcolor
    commentcolorvalcommentcolor commentcolor:commentcolor
    commentcolorℕcommentcolor commentcolor→commentcolor
    commentcolorK
```

Note that Lean infers 0 to be of type $\mathbb{N}$, unless specified. $0 : K$ is just $0 : \mathbb{N}$ along with $0 \in K$. The latter fact is proved by a lemma named has_zero. Also, $K : Type*$ allows Lean to infer the type of K. $check$ is used to print val.

For a definition with multiple arguments, one may think of it as a function, with the domain being a product of the types of the arguments. For example, if there are n arguments of type A_i respectively, and the definition is of type B, then one may think of the definition as a function $\prod_i A_i \to B$, which is equivalent to $A_1 \to \ldots A_{n-1} \to A_n \to B$. This shall play a crucial role when we define lemmas, and apply tactics to it.

A way to have a definition in the local context is by using *let*. We have seen an example above.

2.3 Propositions and theorems

One particular type that is often used is Prop. This is one of the most critical aspects of understanding proof writing in Lean. All propositions are taken to be of type Prop. Say we have a lemma or proposition named h. Then h is of type Prop, and a proof of h is of type h. Thus, the type h is empty if the proposition h is false, and nonempty otherwise. All proofs of h are equivalent, and the type h can have at most a single element. Hence, to prove h, we need only construct a term of type h.

We can now define the syntax of a theorem. Note that lemmas, corollaries and theorems have the same syntax, hence we shall use them interchangeably. In fact, it is the same syntax as definition. This makes sense, because, as explained above, the proof is an element of type being the statement of the theorem. As an example,

```
keywordcolorlemma blackrefl' (blacka : ℕ) : blacka symbolcolor=
    blacka symbolcolor:=
keywordcolorbegin
blacksorry,
keywordcolorend
```

sorry changes the goal to solved. Since this is precisely the reflexive relation, the tactic refl solves the goal.

Note that one may think of theorems as functions from the product of types of arguments to Prop.

[Start from Pg 51] Subdivide into understanding the structures of Lean and doing the math proofs - tactics.

2.4 Variables

The *variables* command allows us to construct variables which need not be defined repeatedly. The scope of variables is within the file. For example, I would like to fix throughout the file that K is a field :

```
keywordcolorvariables {blackK : sortcolorTypesymbolcolor*} [
    blackfield blackK]
```

The curly braces tell Lean to keep the argument implicit, that is, Lean infers the argument. Square and curly brackets differ in the mechanism that Lean uses to infer the argument.

2.5 Import and namespaces

The first thing we need to prove a theorem is to assemble all the data that is already in mathlib, which might be needed in the proof. This is done via the command import. For example, I need the theory of ideals, which is located in mathlib in the branch $ring_theory$, and the file name is ideal.lean. Hence, I start my code with importing all the files I need :

```
keywordcolorimport blackring_theory.blackideals
```

A namespace is used to group code. For example, if a lemma named $lemma$ is proved under the namespace $algebra$, then, the identity of the lemma in other Lean files is $algebra.lemma$. Note that $lemma$ can be accessed only once $algebra$ has been imported. The command $open < namespace >$ makes the namespace accessible (including the variables in it), until $end < namespace >$ is applied. That is, if $openalgebra$ is used, then we may access $lemma$ without the $algebra$ prefix.

2.6 Tactics

Now that we know how to state a theorem, let us look at some common tactics that might help prove it. The foundation of all tactics are logic, or set theory. We shall see examples of all these tactics in the next section.

2.6.1 Simp

The most marvelous tactic to use is simp. Numerous "basic" lemmas in mathlib, such as addition by zero is trivial, are given an attribute of simp. This means that when one uses simp, each of the lemmas given the attribute simp is applied to the goal, and used to simplify the goal. There are several variations of simp, such as $dsimp$, which only uses definitional equalities, or simp only $[t_1, \ldots, t_n]$, which uses only $t_1, \ldots, t_n$ to simplify the goal. Since the functionality of simp changes every time mathlib is updated, our code can come up with errors if simp is used in the early stages of a proof. A way to tackle this is by using $squeeze_simp$, which gives us the particular lemmas simp is using, and suggests applying simp only with those specific lemmas. This is very useful to learn more about the preexisting theorems in mathlib.

Other tactics that work like simp include $norm_cast$, $norm_num$, and $linarith$. In general, it is a good idea to apply simp or any of these tactics and hope that it simplifies, if not solves the goal. If the goal is already present as a local assumption, then one may use the tactic assumption to close the goal.

2.6.2 Intros

This tactic is used to introduce an assumption. For example, if the goal is p $\rightarrow$ q, then intro f introduces an assumption h : p and changes the goal to q.

This tactic cannot be used on local assumptions. It has several variations, such as intros and rintros, which is slightly stronger, and introduces as many assumptions as required.

2.6.3 Rewrite

Suppose there exists a theorem t that the given goal A is equivalent to (via an if and only if statement, or an equality) to B, that is, $t : A \iff B$. We may then use the rewrite tactic. rw t then changes the goal to B. Instead, if the goal is B, and you wish to change it to A, you can use rw $\leftarrow$ t . If we have a local proposition named h, then one may apply the rewrite tactic at h by rw t at h .

Rewrite has several variations, such as simprw or rw_assoc, which are pairing with simp and associativity respectively.

2.6.4 Apply

Recall that rewrite works only for equalities or if and only if statements. To apply an if statement to our goal, the tactic apply must be used. apply works just like rewrite does. Unlike rewrite though, apply cannot be used on local propositions.

Suppose we have a lemma t : p $\rightarrow$ q,and our goal is of the form q. That is, the goal is to produce a proof of q, or an element of type q. The lemma t says that having a proof of p implies the existence of a proof of q. Hence apply t changes the goal to constructing a proof of p. If t has multiple assumptions, or hypotheses, then applying t creates goals to prove that each hypothesis is satisfied.

Conversely, if we have a proof of p, say x, then apply t x solves the goal.

2.6.5 Cases

This tactic is used to split up the target. Some types have building blocks called constructors, and cases split up the target into these building blocks. For example, the type nat, which is the type of natural numbers, is made from 2 constructors, 0 and succ(n) for some $n \in \mathbb{N}$. Here, succ(n) stands for the successor of n, $n + 1$. This means that every $n : nat$ has the form 0 or succ(m) for some $m : nat$. Hence, cases n will split the goal into two, adding assumptions that $n = 0$ and $n = succ(m)$ respectively.

Similarly, given a local assumption f, cases f breaks into various cases of f. For example, if f is an expression of the form A $\wedge$ B, then cases f with f1 f2 breaks f into f1 : A and f2 : B.

2.6.6 Exact

This tactic is used to close the goal, not to simplify it. Suppose we have a lemma or assumption f : q, and a goal q, then exact f closes the goal. Moreover, if f is a generalised statement, and the goal is f taking a specific value, say 1, then exact f 1 solves the goal.

2.6.7 Have

The tactic have is used to create a local assumption. have f : p $\to$ q creates a goal p $\to$ q, and the main goal becomes secondary. Once this goalis solved, f is added to the list of local data. This is a useful tactic when we wish to create a sublemma from the specific given information. However, one must use it with caution, because overutilizing it may end up clogging the workspace.

2.6.8 Contradiction

The tactic $by_contradiction$ is used to solve the goal by contradiction. If the goal is p, it creates a local assumption h : $\neg$p (the negation of p), and changes the goal to false.

If all the ingredients required to show that the given goal is false are present in the local workspace, one may use the tactic exfalso, which changes the goal to false. exfalso applies the lemma false -> p, which is true for all expressions p.

2.6.9 Contrapose

We know that the contrapose of p $\implies$ q is $\neg$q $\implies$ $\neg$p. This is what the tactic contrapose strives to achieve. If we have a local assumption f : p and our goal is q, then contrapose f turns f into f : $\neg$q, and changes the goal to $\neg$p.

2.6.10 Library search

Mathlib is Lean's library for all the math that has been formalised till date. It is important to keep it updated. Moreover, Mathlib is vast, and it can be non-trivial to look for what you need. The command $library_search$ finds lemmas that might help in solving the goal.

While working in VSCode, when we right click on a word, there is an option, Peek Definition. This is very useful, because there are often several lemmas near the definition, which might come in handy.

3 The DVRs

The code I have written for discrete valuation rings is divided into 2 sections. The first section contains some lemmas regarding with top $\mathbb{Z}$, which shall be required in the second section, which has results regarding discrete valuation rings.

There are several equivalent definitions of discrete valuation rings to choose from. The definition we wanted to work with is that of being a local principal ideal domain, which is not a field. However, Lean was having trouble with combining the overlap in local rings and principal ideal domains. Thus, we chose to work with the definition of a DVR being a principal ideal domain with a unique non-zero prime ideal.

A discrete valuation field is then defined to be a field with a non-trivial additive discrete valuation on it. A valuation ring is defined to be a subset of the valuation field, such that every element has non-negative valuation. The aim is to prove that a valuation ring obtained from a discrete valuation field is a discrete valuation ring, as defined above. I shall provide snippets of the proof that the valuation ring is a principal ideal domain. The proof relies heavily on the tactics discussed in the previous section.

3.1 with top Z

Let us first understand the definition of a discrete valuation. An additive valuation on a field K is defined to be a surjective map $v : K \to \mathbb{Z} \bigcup \infty$, such that :
(1) $\forall x, y \in K$, $v(xy) = v(x) + v(y)$
(2) $\forall x, y \in K$, $v(x + y) \geq min(v(x), v(y))$
(3) $\forall x \in K$, $x = 0$ if and only if $v(x) = \infty$
Lean has a lattice structure for $\mathbb{Z} \bigcup \infty$, which is denoted $with_top\mathbb{Z}$. In particular, its elements are ∞, which is denoted T, and a lift of each integer n, denoted $\uparrow n$. It has the usual order that integers have, and $\uparrow n \leq T$ for all $n : \mathbb{Z}$. with top $\mathbb{Z}$ also has addition and multiplication, but no subtraction.

Note that every natural number a, which is of type $\mathbb{N}$(or equivalently of type nat), also has a lift to with top $\mathbb{Z}$, which is denoted $\uparrow a$. This can be confusing at times, as one may confuse it with $\uparrow a : \mathbb{Z}$. Thus, $a : \mathbb{N}$ can be realised in with top $\mathbb{Z}$ in two ways, one is by directly lifting it to with top $\mathbb{Z}$, denoted $\uparrow a$, and the other is by first lifting it to $\mathbb{Z}$, and then to with top $\mathbb{Z}$, denoted $\uparrow\uparrow a$. We shall use the following lemma frequently (which is also given the attributes simp and $norm_cast$) :

```
keywordcolorlemma blackcoe_nat: symbolcolor∀ (blackn : ℕ), ↑↑
    blackn symbolcolor= ↑blackn
```

I have proved the following lemmas regarding with top $\mathbb{Z}$ (the first two are due to Prof Kevin Buzzard) :

```
keywordcolorlemma blackwith_top.blackcases (blacka :
    blackwith_top ℤ) : blacka symbolcolor= ⊤ ∨ symbolcolor∃
    blackn : ℤ, blacka symbolcolor= blackn symbolcolor:=
keywordcolorlemma blacksum_zero_iff_zero (blacka : blackwith_top
    ℤ) : blacka symbolcolor+ blacka symbolcolor= 0 ↔ blacka
    symbolcolor= 0 symbolcolor:=
keywordcolorlemma blackwith_top.blacktransitivity (blacka blackb
    blackc : blackwith_top ℤ) : blacka ≤ blackb -> blackb ≤
    blackc -> blacka ≤ blackc symbolcolor:=
keywordcolorlemma blackwith_top.blackadd_happens (blacka blackb
    blackc : blackwith_top ℤ) (blackne_top : blacka ≠ ⊤) :
    blackbsymbolcolor=blackc ↔ blackasymbolcolor+blackb
    symbolcolor= blackasymbolcolor+blackc symbolcolor:=
keywordcolorlemma blackwith_top.blackadd_le_happens (blacka
    blackb blackc : blackwith_top ℤ) (blackne_top : blacka ≠ ⊤)
    : blackb ≤ blackc ↔ blacka symbolcolor+ blackb ≤ blacka
    symbolcolor+blackc symbolcolor:=
```

```
keywordcolorlemma blackwith_top.blackdistrib (blacka blackb
    blackc : blackwith_top ℤ) (blackna : blacka ≠ ⊤) (blacknb :
    blackb ≠ ⊤) (blacknc : blackc ≠ ⊤) : (blacka symbolcolor+
    blackb)symbolcolor*blackc symbolcolor= blackasymbolcolor*
    blackc symbolcolor+ blackbsymbolcolor*blackc symbolcolor:=
keywordcolorlemma blackone_mul (blacka : blackwith_top ℤ) : 1
    symbolcolor* blacka symbolcolor= blacka symbolcolor:=
keywordcolorlemma blackwith_top.blacksub_add_eq_zero (blackn : ℕ)
     : ((-blackn : ℤ) : blackwith_top ℤ) symbolcolor+ (blackn :
    blackwith_top ℤ) symbolcolor= 0 symbolcolor:=
    commentcolor--commentcolor commentcolor⊢commentcolor
    commentcolor↑commentcolor-commentcolor↑commentcolorn
    commentcolor commentcolorsymbolcolor+commentcolor
    commentcolor↑commentcolorncommentcolor
    commentcolorsymbolcolor=commentcolor commentcolor0
keywordcolorlemma blackwith_top.blackadd_sub_eq_zero (blackn : ℕ)
     : (blackn : blackwith_top ℤ) symbolcolor+ ((-blackn : ℤ) :
    blackwith_top ℤ) symbolcolor= 0 symbolcolor:=
    commentcolor--commentcolor commentcolor⊢commentcolor
    commentcolor↑commentcolorncommentcolor
    commentcolorsymbolcolor+commentcolor commentcolor↑
    commentcolor-commentcolor↑commentcolorncommentcolor
    commentcolorsymbolcolor=commentcolor commentcolor0
```

Notice that the last couple of lemmas take $n : withtop\mathbb{Z}$ and $-n : \mathbb{Z}$. We know that n does not have type $withtop\mathbb{Z}$, Lean infers the former statement as $\uparrow n : withtop\mathbb{Z}$, and the latter as $- \uparrow n$.

3.2 Discrete valuation ring

Let us now have a look at the definition of discrete valuation ring in Lean.

```
keywordcolorclass blackdiscrete_valuation_ring (blackR :
    sortcolorType blacku) [blackintegral_domain blackR] [
    blackis_principal_ideal_ring blackR] symbolcolor:=
(blackprime_ideal' : blackideal blackR)
(blackprimality : blackprime_ideal'.blackis_prime)
(blackis_nonzero : blackprime_ideal' ≠ ⊥)
(blackunique_nonzero_prime_ideal : symbolcolor∀ blackP :
    blackideal blackR, blackP.blackis_prime → blackP symbolcolor=
     ⊥ ∨ blackP symbolcolor= blackprime_ideal')
```

The structure class is used to facilitate a group of definitions. Here, we define a discrete valuation ring R to be an integral domain and a principal ideal ring which contains an ideal $prime_ideal'$. The expressions primality and $is_nonzero$ state that $prime_ideal'$ is prime and nonzero respectively. The final condition states that any ideal in R that is prime must be trivial or $prime_ideal'$.

For the purposes of this article, we shall not be needing any of the other code pertaining to discrete valuation rings, hence I shall skip it.

3.3 Discrete valuation field

Let us now define a discrete valuation field, which is a field K with an additive valuation on it, as defined in a previous section.

```
keywordcolorclass blackdiscrete_valuation_field (blackK :
    sortcolorTypesymbolcolor*) [blackfield blackK] symbolcolor:=
(blackv : blackK -> blackwith_top ℤ )
(blackhv : blackfunction.blacksurjective blackv)
(blackmul : symbolcolor∀ (blackx blacky : blackK), blackv(blackx
    symbolcolor*blacky) symbolcolor= blackv(blackx) symbolcolor+
    blackv(blacky) )
(blackadd : symbolcolor∀ (blackx blacky : blackK), blackmin (
    blackv(blackx)) (blackv(blacky)) ≤ blackv(blackx symbolcolor+
     blacky)  )
(blacknon_zero : symbolcolor∀ (blackx : blackK), blackv(blackx)
    symbolcolor= ⊤ ↔ blackx symbolcolor= 0 )

keywordcolornamespace blackdiscrete_valuation_field

keywordcolordefinition blackvaluation (blackK : sortcolorType
    symbolcolor*) [blackfield blackK] [
    blackdiscrete_valuation_field blackK ] : blackK ->
    blackwith_top ℤ symbolcolor:= blackv

keywordcolorvariables {blackK : sortcolorTypesymbolcolor*} [
    blackfield blackK] [blackdiscrete_valuation_field blackK]
```

Notice that in the definition of a discrete valuation field, K is of $Type*$, which means that Lean may infer the type of K. After the definition, we open a namespace called discrete valuation field. It would be convenient to have a definition of valuation, since it is used so often. Finally, we make K a variable representing a field and a discrete valuation field. Notice that we need to put an argument of K being a field that is used by the definition of discrete valuation field.

The following lemmas are proved :

```
keywordcolorlemma blackval_one_eq_zero : blackv(1 : blackK)
    symbolcolor= 0 symbolcolor:=
keywordcolorlemma blackval_minus_one_is_zero : blackv((-1) :
    blackK) symbolcolor= 0 symbolcolor:=
black@[blacksimp] keywordcolorlemma blackval_zero : blackv(0:
    blackK) symbolcolor= ⊤ symbolcolor:=
```

The first lemma has been proved by Prof Kevin Buzzard. The last lemma has been given an attribute simp, which means that applying simp to a goal containing $v(0)$ will change it to T. Also, notice how the type of 1 and 0 must be specified to be K. Failure to do this results in an error :

```
blackfailed blackto blacksynthesize blacktype keywordcolorclass
    keywordcolorinstance blackfor
⊢ blackfield ℕ
```

This is because v takes values in a field, and since 1 has type $\mathbb{N}$, Lean tries to infer that $\mathbb{N}$ is a field and fails.

We now define the valuation ring obtained from a discrete valuation field :

```
keywordcolordef blackval_ring (blackK : sortcolorTypesymbolcolor*
    ) [blackfield blackK] [blackdiscrete_valuation_field blackK]
    symbolcolor:= { blackx : blackK | 0 ≤ blackv blackx }

keywordcolorinstance (blackK : sortcolorTypesymbolcolor*) [
    blackfield blackK] [blackdiscrete_valuation_field blackK] :
    blackis_add_subgroup (blackval_ring blackK) symbolcolor:=
{
  blackzero_mem symbolcolor:= keywordcolorbegin
              blackunfold blackval_ring,
              blacksimp,
              keywordcolorend,
  blackadd_mem symbolcolor:= keywordcolorbegin
            blackunfold blackval_ring,
            blacksimp blackonly [blackset.blackmem_set_of_eq],
            blackrintros,
            keywordcolorhave blackg : blackmin (blackv(blacka)) (
    blackv(blackb)) ≤ blackv(blacka symbolcolor+ blackb),
            {
              blackapply blackadd,
            },
            blackrw blackmin_le_iff keywordcolorat blackg,
            blackcases blackg,
            {
              blackexact blackwith_top.blacktransitivity black_
    black_ black_ blacka_1 blackg,
            },
            {
              blackexact blackwith_top.blacktransitivity black_
    black_ black_ blacka_2 blackg,
            },
            keywordcolorend,
  blackneg_mem symbolcolor:= keywordcolorbegin
            blackunfold blackval_ring,
            blackrintros,
            blacksimp blackonly [blackset.blackmem_set_of_eq],
            blacksimp blackonly [blackset.blackmem_set_of_eq]
    keywordcolorat blacka_1,
            keywordcolorhave blackf : -blacka symbolcolor= blacka
     symbolcolor* (-1 : blackK) symbolcolor:= keywordcolorby
    blacksimp,
            blackrw [blackf, blackmul, blackval_minus_one_is_zero
    ],
            blacksimp [blacka_1],
            keywordcolorend,
```

```
}

keywordcolorinstance (blackK:sortcolorTypesymbolcolor*) [
    blackfield blackK] [blackdiscrete_valuation_field blackK] :
    blackis_submonoid (blackval_ring blackK) symbolcolor:=
{ blackone_mem symbolcolor:= keywordcolorbegin
            blackunfold blackval_ring,
            blacksimp,
            blackrw blackval_one_eq_zero,
            blacknorm_num,
            keywordcolorend,
  blackmul_mem symbolcolor:= keywordcolorbegin
            blackunfold blackval_ring,
            blackrintros,
            blacksimp,
            blacksimp keywordcolorat blacka_1,
            blacksimp keywordcolorat blacka_2,
            blackrw blackmul,
            blackapply blackadd_nonneg' blacka_1 blacka_2,
            keywordcolorend, }

keywordcolorinstance blackvaluation_ring (blackK:sortcolorType
    symbolcolor*) [blackfield blackK] [
    blackdiscrete_valuation_field blackK] : blackis_subring (
    blackval_ring blackK) symbolcolor:=
{}

keywordcolorinstance blackis_domain (blackK:sortcolorType
    symbolcolor*) [blackfield blackK] [
    blackdiscrete_valuation_field blackK] : blackintegral_domain
    (blackval_ring blackK) symbolcolor:=
blacksubring.blackdomain (blackval_ring blackK)

keywordcolordef blackunif (blackK:sortcolorTypesymbolcolor*) [
    blackfield blackK] [blackdiscrete_valuation_field blackK] :
    blackset blackK symbolcolor:= { π | blackv π symbolcolor= 1 }

keywordcolorvariables (π : blackK) (blackhπ : π ∈ blackunif
    blackK)
```

The definition of val_ring does not accept the variable K. Moreover, Lean automatically infers val_ring K : set K, that is, it is a subset of K.

An instance is essentially a property with a proof. At the end of this code, Lean understands that val_ring K is a subring of K. In order to prove the instance that val_ring K is an additive subgroup of K, we must prove that it satisfies all the conditions specified in the definition of $is_add_subgroup$. An additive subgroup is defined to be a set containing zero ($zero_mem$), and being closed under addition (add_mem) and negation (neg_mem). Providing a proof for each of these properties suffices. Similarly,a submonoid is defined to be a set containing 1

($one_m em$) and is closed under multiplication ($mul_m em$).

We first prove that the valuation ring is an additive subgroup and a submonoid of K, which then implies that $val_r ing$ K is a subring of K. We then use the fact that a subring of a domain is an integral domain, to show that the valuation ring is an integral domain. Finally, we define the set of uniformisers of K, that is, the elements of K having valuation 1. We then take π to be a variable denoting a uniformiser of K.

We now prove a bunch of lemmas before proving that $val_r ing$ K is a principal ideal ring (we have already shown it is a domain) :

```
keywordcolorlemma blackval_unif_eq_one (blackhπ : π ∈ blackunif
    blackK) : blackv(π) symbolcolor= 1 symbolcolor:=
keywordcolorlemma blackunif_ne_zero (blackhπ : π ∈ blackunif
    blackK) : π ≠ 0 symbolcolor:=
keywordcolorlemma blackval_inv (blackx : blackK) (blacknz :
    blackx ≠ 0) : blackv(blackx) symbolcolor+ blackv(blackx)⁻¹
    symbolcolor= 0 symbolcolor:=
keywordcolorlemma blackcontra_non_zero (blackx : blackK) (blackn
    : ℕ) (blacknz : blackn ≠ 0) : blackv(blackx^blackn) ≠ ⊤ ↔
    blackx ≠ 0 symbolcolor:=
keywordcolorlemma blackcontra_non_zero_one (blackx : blackK) :
    blackv(blackx) ≠ ⊤ ↔ blackx ≠ 0 symbolcolor:=
keywordcolorlemma blackval_nat_power (blacka : blackK) (blacknz :
     blacka ≠ 0) : symbolcolor∀ blackn : ℕ, blackv(blacka^blackn)
     symbolcolor= (blackn : blackwith_top ℤ)symbolcolor*blackv(
    blacka) symbolcolor:=
keywordcolorlemma blackval_int_power (blacka : blackK) (blacknz :
     blacka ≠ 0) : symbolcolor∀ blackn : ℤ, blackv(blacka^blackn)
     symbolcolor= (blackn : blackwith_top ℤ)symbolcolor*blackv(
    blacka) symbolcolor:=
keywordcolorlemma blackunit_iff_val_zero (α : blackK) (blackhα :
    α ∈ blackval_ring blackK) (blacknzα : α ≠ 0) : blackv (α)
    symbolcolor= 0 ↔ symbolcolor∃ β ∈ blackval_ring blackK, α
    symbolcolor* β symbolcolor= 1 symbolcolor:=
keywordcolorlemma blackval_eq_iff_asso (blackx blacky : blackK) (
    blackhx : blackx ∈ blackval_ring blackK) (blackhy : blacky ∈
    blackval_ring blackK) (blacknzx : blackx ≠ 0) (blacknzy :
    blacky ≠ 0) : blackv(blackx) symbolcolor= blackv(blacky) ↔
    symbolcolor∃ β ∈ blackval_ring blackK, blackv(β) symbolcolor=
     0 ∧ blackx symbolcolor* β symbolcolor= blacky symbolcolor:=
keywordcolorlemma blackunif_assoc (blackx : blackK) (blackhx :
    blackx ∈ blackval_ring blackK) (blacknz : blackx ≠ 0) (blackh
    π : π ∈ blackunif blackK) : symbolcolor∃ β ∈ blackval_ring
    blackK, (blackv(β) symbolcolor= 0 ∧ symbolcolor∃! blackn : ℤ,
     blackx symbolcolor* β symbolcolor= π^blackn) symbolcolor:=
keywordcolorlemma blackval_is_nat (blackhπ : π ∈ blackunif blackK
    ) (blackx : blackval_ring blackK) (blacknzx : blackx ≠ 0) :
    symbolcolor∃ blackm : ℕ, blackv(blackx:blackK) symbolcolor= ↑
```

```
    blackm symbolcolor:=
keywordcolorlemma blackexists_unif : symbolcolor∃ π : blackK,
    blackv(π) symbolcolor= 1 symbolcolor:=
```

Notice that the hypothesis $h\pi$ uses that $\pi \in unif$ K, not $\pi : unif$ K. This is because π has already been defined to be of type K. Also, $a \neq b$ is the same as $\neg a = b$, which is the same as $a = b \implies false$. In the lemma val_inv, it would have been more convenient if one could write $v(x^{-1}) = -v(x)$, however subtraction is not defined in with top $\mathbb{Z}$, because one would then have to define T - T. A lot of the lemmas are proved for both $n : \mathbb{N}$, and $n : \mathbb{Z}$. Dealing with the case $n : \mathbb{N}$ is slightly harder than $n : \mathbb{Z}$. An alternative would have been to work in enat, which is analogous to with top $\mathbb{N}$. In order to do this, one would have to restrict the valuation to the valuation ring, and apply it there. This could turn out to be problematic, in case an element from K is needed.

If K is a field, for $\alpha :$ K, α^{-1} is defined to be the inverse of α if α is non-zero, and 0 if $\alpha = 0$. Division a/b is defined to be $a * b^{-1}$, so this means $1/0 = 0$ as well. The nonzero part is added as a hypothesis in lemmas that need it, such as, when $a * a^{-1} = 1$.

3.4 Proof of being a PID

```
keywordcolorinstance blackis_pir (blackK:sortcolorType
    symbolcolor*) [blackfield blackK] [
    blackdiscrete_valuation_field blackK] :
    blackis_principal_ideal_ring (blackval_ring blackK)
    symbolcolor:=
```

The proof is split into several lemmas, which are then patched up together. I shall reproduce a proof of some of them. The definition of $is_principal_ideal_ring$ is :

```
keywordcolorclass blackis_principal_ideal_ring (blackR :
    sortcolorType blacku) [blackcomm_ring blackR] : sortcolorProp
     symbolcolor:=
(blackprincipal : symbolcolor∀ (blackS : blackideal blackR),
    blackS.blackis_principal)
```

Thus, we (only) need to show that every ideal in val_ringK is principal. At first, we need to choose a uniformiser π. Note that this is the only place where I have encountered requiring the valuation to be non-zero, or surjective. I shall skip the proof of this lemma, named :

```
blackhπ: π ∈ blackunif blackK
```

where unif K is the set of uniformisers of K. The next tactic, rintros, changes the goal from

```
⊢ symbolcolor∀ (blackS : blackideal [U+21A5](blackval_ring blackK
    )), blacksubmodule.blackis_principal blackS
```

to

```
blackS: blackideal [U+21A5](blackval_ring blackK)
⊢ blacksubmodule.blackis_principal blackS
```

Thus rintros, which is a stronger version of intros, introduces the variable S, an ideal in $val_r ingK$. We now deal with the trivial case of S being empty.

```
blackby_cases blackS symbolcolor= ⊥,
{ blackrw blackh,
  blackuse 0,
  blackapply blackeq.blacksymm,
  blackrw blacksubmodule.blackspan_singleton_eq_bot, },
```

```
blackh: blackS symbolcolor= ⊥ ⊢ bla
blackis_principal blackS
⊢ blacksubmodule.blackis_principal
⊢ ⊥ symbolcolor= blacksubmodule.bla
blackval_ring blackK) {0}
⊢ blacksubmodule.blackspan [U+21A5](
 symbolcolor= ⊥
blackgoals blackaccomplished
```

The $by_c ases$ tactic separates the goal into cases of S being empty and nonempty. The rewrite tactic replaces an equality (or if and only if statement) in the goal with the equality (or if and only if statement) given in the input. $submodule.is_p rincipal$ is looking for a generator, which is provided by the tactic use. The next statement applies the lemma $eq.symm$, which says that $a = b$ implies $b = a$. The final lemmma that solves the goal says that the span of a submodule is 0 if and only if it is generated by 0.

We now have S to be nonempty, which is stored in the proposition h. Next, we define the set Q of all naturals that correspond to the valuation of some x in S. The infimum of Q shall be denoted by $InfQ$. We know that S is then generated by π^{InfQ} We also state a lemma g (whose proof shall be omitted), which says that the valuation of π^{InfQ} is the image of $InfQ$ in $with_t op(\mathbb{Z})$.

```
keywordcolorlet blackQ symbolcolor:= {blackn : ℕ |
symbolcolor∃ blackx ∈ blackS, (blackn : blackwith_top ℤ)
symbolcolor= blackv(blackx:blackK) },

keywordcolorhave blackg : blackv(π ^(blackInf blackQ))
symbolcolor= ↑(blackInf blackQ),
```

Note that Lean automatically sets Q to be of type set ℕ. Also, since we put the condition $x \in S$, x is automatically taken to be of type $val_r ing$ K(arrow?).

```
keywordcolorhave blacknz : π^(blackInf blackQ) ≠ 0,
{ blackby_contradiction,
  blacksimp keywordcolorat blacka,
  blackapply_fun blackv keywordcolorat blacka,
  blackrw [blackg, blackval_zero] keywordcolorat blacka,
  blackapply blackwith_top.blacknat_ne_top (blackInf blackQ),
  blackexact blacka, },
```

```
⊢ π ^ blackInf blackQ ≠ 0
blacka: ¬π ^ blackInf blackQ ≠ 0 ⊢ blac
blacka: π ^ blackInf blackQ symbolcolor=
blacka: blackv (π ^ blackInf blackQ) sym
blackfalse
blacka: ↑(blackInf blackQ) symbolcolor=
blacka: ↑(blackInf blackQ) symbolcolor=
) symbolcolor= ⊤
blackgoals blackaccomplished
```

The above lemma nz, shows that $\pi^{InfQ} \neq 0$. The proof is by contradiction. The first tactic, $by_c ontradiction$ makes the negation of the goal the proposition a, and turns the goal to false. Simp at a turns a into a simpler form. The $apply_f un$ tactic applies the function v to a. The next line is the rewrite tactic on a, using g and the lemma val_zero (v(0) = T). The next lemma to be used is imported from the file $with_top$, and it states that for all $n \in \mathbb{N}$, $(n : with_top\mathbb{Z} \neq T)$, or equivalently, $(n : with_top\mathbb{Z} = T) \implies false$. Since our goal is false, the apply tactic, taking the specific value $n = InfQ$ changes it to the precise statement of a. exact a then solves the goal.

```
blackuse π^(blackInf blackQ),
blackapply blacksubmodule.blackext,

blackrintros,

blacksplit,
```

```
⊢ π ^ blackInf blackQ ∈ blackval_ring blackK
⊢ symbolcolor∀ (blackx : [U+21A5](blackval_ring blackK)),
blackx ∈ blackS ↔ blackx ∈ blacksubmodule.blackspan
[U+21A5](blackval_ring blackK) {⟨π ^ blackInf blackQ, bl
blackx: [U+21A5](blackval_ring blackK)
⊢ blackx ∈ blackS ↔ blackx ∈ blacksubmodule.blackspan [U+2
blackval_ring blackK) {⟨π ^ blackInf blackQ, black_⟩}
⊢ blackx ∈ blackS → blackx ∈ blacksubmodule.blackspan [U+2
blackval_ring blackK) {⟨π ^ blackInf blackQ, black_⟩}
```

We now get into the proof that π^{InfQ} is the generator of S. After applying use π^{InfQ}, we must first show that π^{InfQ} is in fact an element of val_ring K. We shall skip this proof. The theorem ext (imported from submodule.lean) says that if, for every x in a module, and for submodules p and q, $x \in p \iff x \in q$, then $p = q$. In order to apply the theorem, Lean must make sure the hypothesis is true, hence the goal changes to the hypothesis. rintros introduces a variable x in val_ringK, and the tactic split splits the goal into 2 goals, which say that each ideal is contained in the other. Note that ideals are defined as submodules of the ring.

We now solve the second goal(we omit the proof of the first goal), which is to show that $submodule.span\pi^{InfQ} \subset S$. This is done by showing that $InfQ \in Q$, thus there exists $z \in S$ such that $v(z) = InfQ$, and z is associated to π^{InfQ}, hence $\pi^{InfQ} \in S$.

```
keywordcolorhave blackf' : symbolcolor∃ blackx ∈ blackS,
blackx ≠ (0 : blackval_ring blackK),
  { blackcontrapose blackh,

    blacksimp keywordcolorat blackh,
    blacksimp,
    blackapply blackideal.blackext, blackrintros, blacksimp
blackonly [blacksubmodule.blackmem_bot],
    blacksplit,
    { blackrintros,
      blackspecialize blackh blackx_1, blacksimp
keywordcolorat blackh,
      blackapply blackh blacka_1, },
    },
```

```
⊢ symbolcolor∃ (
blackH : blackx
)
blackh: ¬symbolc
)) (blackH : bla
blackK) ⊢ ¬¬bla
blackh: symbolco
blackval_ring bl
blackx, blackx_1
blackx_1: [U+21A
⊢ blackx_1 ∈ bla
⊢ blackx_1 ∈ bla
blacka_1: blackx
blackh: blackx_1
⊢ blackx_1 symbo
blackgoals black
```

We start by proving that since S is nonempty, it must have a nonzero element. Note that we must specify that 0 is in val_ring K, else Lean assumes it to be of type $\mathbb{N}$, and gives a type mismatch error. We prove this using the contrapose tactic, which takes a known expression, and transforms the goal into the contrapositive with respect to the given argument. In this case, we choose h, which says that S is nonempty. The simp statements apply the negation on h and the goal respectively. There is a way to check which lemmas or theorems simp is applying. One may type $squeeze_simp$, and Lean will give suggestions for the applicable lemmas. In the next line, Lean suggests applying the lemma mem_bot, which says that every element of the trivial submodule is 0. We then have an if and only if statement, that splits into the two implications. We omit the proof of $x_1 = 0 \implies x_1 \in S$. In the proof of the other implication, rintros introduces a variable $x_1 \in S$. The tactic specialize applies the specific case of x_1 to h. simp does the nontrivial job of converting h to the given more pleasant form.

```
keywordcolorhave blackp : blackInf blackQ ∈ blackQ,
  { blackapply blacknat.blackInf_mem,
    blackcontrapose blackh,
    blacksimp,
    blackby_contradiction,
    blackcases blackf' keywordcolorwith blackx' blackf',
    keywordcolorhave blackf_1 : symbolcolor∃ blackm : ℕ,
blackv(blackx':blackK) symbolcolor= ↑(blackm),
    { blackapply blackval_is_nat,
      blackexact blackhπ,
      blackcases blackf',
      blackcontrapose blackf'black_h,
      blacksimp,
      blacksimp keywordcolorat blackf'black_h,
      blackrw blackf'black_h, },
    blackcases blackf_1 keywordcolorwith blackm' blackf_1,
    keywordcolorhave blackg' : blackm' ∈ blackQ,
    { blacksimp,
      blackuse blackx',
      blacksimp,
      blacksplit,
      blackcases blackf',
      blackassumption,
      blackexact blackeq.blacksymm blackf_1, },
    blackapply blackh,
    blackuse blackm',
    blackapply blackg', },
```

Next, we show that $InfQ \in Q$. We apply a lemma called Inf_mem, which says that any nonempty subset of the naturals contains its infimum. Hence, the goal then changes to showing that Q is nonempty. We use the contrapose h tactic again, which changes the goal to Q being empty implies S is empty. This is done by using the $by_contradiction$ tactic. We now use f to construct $x' \neq 0$ in S. The cases tactic introduces the case of f' for the variable x'. We want to then show that the valuation of x' is the image of a natural number in $with_topℤ$, say m'. Note that this is not trivial to Lean, since x' being nonzero implies that there exists some integer whose image is the valuation of x'. The central issue is that $\mathbb{Z}$ and $\mathbb{N}$ are different types, and every natural number is not immediately identified as an integer. This is caused because Lean identified Q as a set in $\mathbb{N}$, and thus it will not accept integers as elements. Once we have m' of type $\mathbb{N}$, we want to show that $m' \in Q$. This follows from the definition of x' and Q. The contradiction now lies in the fact that h says Q is empty, while we have shown $m' \in Q$.

```
keywordcolorhave blackf : symbolcolor∃ blackz ∈ blackS,
blackv(blackz : blackK) symbolcolor= ↑(blackInf blackQ),
```

Since we have shown that $InfQ \in Q$, by definition, this implies that $\exists z \in S$ such that $v(z) = InfQ$. This is the proof of the above lemma f. We now have all the ingredients for our proof.

```
    blackcases blackf keywordcolorwith blackz blackf,
    blackrw symbolcolor<-blackg keywordcolorat blackf,
    blacksimp keywordcolorat blackf,
    blackcases blackf,
    blackrw blackval_eq_iff_asso keywordcolorat blackf_right,
    { blackcases blackf_right keywordcolorwith blackw blackf_1,
      blackcases blackf_1 keywordcolorwith blackf_1 blackf_2,
      blackcases blackf_2 keywordcolorwith blackf_2 blackf_3,
      blackrw blackset.blacksingleton_subset_iff,
      blacksimp blackonly [blacksubmodule.blackmem_coe],
      blacksimp_rw ←[ blackf_3],
      blackchange blackz symbolcolor* ⟨blackw,blackf_1⟩ ∈
blackS,
      blackapply blackideal.blackmul_mem_right blackS
blackf_left, },
```

First, using the cases tactic, we fix a $z \in S$ with valuation $InfQ$. The lemma $val_eq_iff_asso$ is a lemma proved by me, which states that,

```
keywordcolorlemma blackval_eq_iff_asso (blackx blacky : blackK) (
    blackhx : blackx ∈ blackval_ring blackK) (blackhy : blacky ∈
    blackval_ring blackK) (blacknzx : blackx ≠ 0) (blacknzy :
    blacky ≠ 0) : blackv(blackx) symbolcolor= blackv(blacky) ↔
    symbolcolor∃ β ∈ blackval_ring blackK, blackv(β) symbolcolor=
     0 ∧ blackx symbolcolor* β symbolcolor= blacky symbolcolor:=
```

This is equivalent to saying that β is a unit in val_ringK. After applying cases, we get the required expression and variables, $z, w \in val_ringK$ and $z \in S$ such that f_3 holds. Note that z with the arrow in f_3 denotes the image, or lift, of z in K. Thus f_3 is an expression of multiplication in K. The following part was one of the most challenging parts of the proof, because it seemed that all that needed to be done was rewriting f_3 into the goal, after which one may use that S is an ideal, and since $z \in S$, $\pi^{InfQ} \in S$. However, I kept getting errors, because π^{InfQ} is of type K, while, in order to be able to apply any properties of ideals on it, it must be of type val_ringK. Note that the goal shows S with an arrow, which means that we are looking at the lift of S in K. The lemma mem_coe brings the goal down to S. The <,> signifies that the condition $pi^{InfQ} \in val_ringK$ is also encoded in the goal. The tactic change replaces the goal with the provided expression, as long as it is definitionally equal to the goal. We finally manage to successfully change it into a goal which has multiplication in S. We can then apply the lemma mul_mem_right, which states that the product (right multiplication) of an element of an ideal with an element of the ring remains in the ideal. This solves the goal.

4 Future work

Iwasawa theory

5 KB enlightenment

If R is a subring of K, then a term of type R is a pair, consisting of an element of K and a proof that it satisfies the property defining R.

Kevin Buzzard: If you want to change any goal you like to false, you can do exfalso. Kevin Buzzard: If P is any Proposition (either true or false), then false -> P Kevin Buzzard: so you can apply false -> P to your goal, if your goal is P, and it changes the goal to false Kevin Buzzard: Inductive types (I was talking about them earlier) have principles of induction attached to them. Kevin Buzzard: The principles of induction (or more generally principles of recursion, depending on whether you are proving things or defining things) are generated automatically from the constructors of the inductive type. Kevin Buzzard: For example nat has two constructors, zero and succ n, so the principle of induction says "if you've proved it for zero, and if you can prove it for succ n given that you have already proved it for n, then you've proved it for all natural numbers" Kevin Buzzard: false has no constructors! Kevin Buzzard: So the principle of induction for false is that you don't have to do anything at all with P, and the conclusion is that false -> P

Kevin Buzzard: In general the answer is that if you want to use a previous lemma then you can just write have h : <statement of lemma> := <name of lemma> Kevin Buzzard: When you write lemma unique$_max_ideal : \exists! I : idealR, I.is_maximal := ...KevinBuzzard : youaresayingthis.ThereisaProposition, whichis\exists! I : idealR, I.is_maximal.Thisisatrue/falsestatement.IthastypeProp.KevinBuzzard : Andthereisalsoitsproof, whichyouaregoingtocallunique_max_idealKevinBuzzard : Whenwesaythingslike"theBolzanoWeierstrasstheoremsays(blah)"wearetalkingaboutthestatementofthet$ $Ittookmeawhiletorealisethatmathematiciansusethephrase"theorem"tomeanbothofthosethings.InLeanwe$ $2 = 5isaproposition), andproofs.PropositionshavetypePropandifPisapropositionitsproofhastypeP.Thest$

Kevin Buzzard: If you have h : P and f : P -> Q and you really just need a proof of Q and don't care about P any more, another possibility is replace h := f h. Kevin Buzzard: Or if you want to change f : P -> Q to f : Q then you can specialize f h

Kevin Buzzard: If f : P -> false and h : P then h is an element of set of proofs of P, and f is a function mapping proofs of P to proofs of false. Kevin Buzzard: So f h : false Kevin Buzzard: i.e. applying the function f to the term h gives you a term of type false. Kevin Buzzard: If you want to make a new term of type false you can do have h2 : false := f h or just have h2 := f h Kevin Buzzard: If you want to change h from a proof of P to a proof of false you can do apply$_f unfathKevinBuzzard : Ifyoujustwanttocloseyourgoalthough, youcanjustdocases(fh)(maybeyoudo$
$becauseifx : AandAisaninductivetypewithnnnconstructors(e.g.likenatisaninductivetypewith2constructors$
$natthencasesnwillproducetwogoals, onewithn = 0n = 0n = 0andonewithn =$
$succd(actuallyyouhadbetterputcasesnwithdtomakesureit'scalledd, otherwiseitwillgetsomejunknameliken_1$

Kevin Buzzard: That's what a subset of K is in Lean. It is simply a map from K to Prop, sending an element a to the true/false statement saying that a is in the subset Kevin Buzzard: Every time you build a new thing, a new subset, a new definition – every time you make one, you are making your structure bigger.

6 Bibliography

Source code - https://github.com/laughinggas/DVR/blob/master/src/Test.lean

Tactics in Mathlib - https://leanprover-community.github.io/mathlib$_d ocs/tactics.htmltop$

Theorem Proving in Lean - https://leanprover.github.io/tutorial/tutorial.pdf

Natural numbers game - http://wwwf.imperial.ac.uk/ buzzard/xena/natural$_n umber_g ame/$

Mathematics in Lean - https://leanprover-community.github.io/mathematics$_i n_l ean/index.html$