Latvijas Universitāte
Datorikas fakultāte

Galuā lauku realizācija, izmantojot vispārējās programmēšanas paradigmu

Kvalifikācijas darbs

Autors:
Madars Virza
St. apl. nr.: mv07012

Vadītājs:
Sergejs Kozlovičs
Mg. dat.

Rīga, 2009

Anotācija

Darbs "Galuā lauku realizācija, izmantojot vispārējās programmēšanas paradigmu" apraksta bibliotēkas, kas ļauj darboties ar patvaļīgiem galīgajiem laukiem $\mathsf{GF}(\mathfrak{p}^{\mathfrak{n}})$, izstrādi. Tiek atbalstītas visas aritmētiskās operācijas ar lauka elementiem, kā arī kvadrātsaknes vilkšana un lauka parametru ģenerēšana. Bibliotēka dota C++ izejas tekstu veidā, ir strukturēta un savietojama ar vairākiem kompilatoriem vairākās arhitektūrās.
Atslēgvārdi: Galuā lauki, galīgie lauki, galīga lauka ģenerēšana, C++

Abstract

The paper "Generic programming implementation of Galois fields" describes development of software library which implements arbitrary finite fields $\mathsf{GF}(\mathtt{p}^{\mathtt{n}})$. Supported operations include all arithmetical operations on field elements, taking square root and generating finite fields. The library is implemented as C++ header files, it is well structured and supports multiple compilators and processor architectures.
Keywords: Galois fields, finite fields, finite field generation, C++

Saturs

Ievads **4**

1. Pamatdefinīcijas un teorēmas **6**

2. Programmatūras prasību specifikācija **9**

2.1. Ievads 9

2.1.1. Nolūks 9

2.1.2. Darbības sfēra 9

2.1.3. Saistība ar citiem dokumentiem 9

2.2. Vispārējais apraksts 10

2.2.1. Produkta perspektīva 10

2.2.2. Produkta funkcijas 10

2.2.3. Lietotāja raksturiezīmes 10

2.2.4. Pieņēmumi un atkarības 10

2.3. Konkrētās prasības 10

2.3.1. Funkcionālās prasības 10

2.3.2. Veiktspējas prasības 12

2.3.3. Aparatūras ierobežojumi 12

2.3.4. Nefunkcionālās prasības 13

2.3.5. Ārējās saskarnes prasības 13

3. Programmatūras projektējuma apraksts **14**

3.1. Ievads 14

3.1.1. Dokumenta nolūks 14

3.1.2. Darbības sfēra 14

3.1.3. Definīcijas 14

3.2. Saistība ar citiem dokumentiem 14

3.3. Dekompozīcijas apraksts 15

3.3.1. Ievads 15

3.3.2. Entītiju dekompozīcija 15

3.3.3. Funkciju dekompozīcijas projektējums 18

3.4. Atkarību projektējums 20
3.5. Detalizētais projektējums 21
3.5.1. `is_prime` 21
3.5.2. `ExtendedEuclideanAlgorithm` 22
3.5.3. `powmod` 22
3.5.4. `PrimeField` 23
3.5.5. `Polynomial` 24
3.5.6. `IsIrreducible` 29
3.5.7. `GenerateRandomPolynomial` 29
3.5.8. `GenerateIrreduciblePolynomial` 29
3.5.9. `GaloisField` 30
3.6. Projektējuma noteiktie ierobežojumi 33

4. Testēšanas dokumentācija **35**
4.1. Testu kopa `Polynomial` metodes `operator/` testēšanai 36
4.2. Testu kopa funkcijas `IsIrreducible` testēšanai 36
4.3. Testu kopa `GaloisField` metodes `sqrt` testēšanai 37
4.4. Testēšanas žurnāls 38
4.5. Problēmu ziņojumi 39
4.5.1. sqrt1 39
4.6. Bibliotēkas savietojamības testi 39

5. Programmatūras pirmkoda fragmenti **41**
5.1. Funkcija `ExtendedEuclideanAlgorithm` 41
5.2. `divmod` noteikšana, izmantojot template specialization 42
5.3. Funkcija `GenerateIrreduciblePolynomial` 42
5.4. Klase `GaloisField` 44
5.5. Bibliotēkas lietošanas piemērs kvadrātsaknes vilkšanai 48

6. Projekta organizācija **49**

7. Konfigurāciju pārvaldība **50**

8. Kvalitātes nodrošināšana **51**

9. Darbietilpības novērtējums **52**

10.Rezultāti **54**

11.Secinājumi **55**

Pateicības **56**

Izmantotā literatūra un avoti **57**

Ievads

Galīgie jeb Galuā lauki (nosaukti Evarista Galuā vārdā) ir ļoti svarīgs objekts skaitļu teorijā, kriptogrāfijā un kodēšanas teorijā. Ļoti daudzi mūsdienās lietotie kļūdas detektējošie vai labojošie kodi ir balstīti uz dažādu operāciju īpašībām Galuā laukos [7]. Tāpat dažādu operāciju sarežģītība Galuā laukos ir pamatā to plašam lietojumam kriptogrāfijas protokolu izstrādē, it īpaši eliptisko līkņu kriptogrāfijā [5].

Šī darba mērķis bija izstrādāt bibliotēku, kas realizē pamata aritmētiskās operācijas (saskaitīšanas, atņemšanas, reizināšanas un dalīšanas) patvaļīgā galīgajā laukā. Ne mazāk svarīgs mērķis bija bibliotēku noformēt saskaņā ar valsts standartiem un tās izstrādi veikt saskaņā ar labo programmēšanas praksi.

Lai sasniegtu darba mērķi tika izvēlēts veikt šādus uzdevumus:

- iepazīties ar teorētisko materiālu par Galuā lauku reprezentāciju un operācijām ar Galuā laukiem,
- aplūkot līdzšinējo pieredzi šādu problēmu risināšanā,
- iepazīties ar programmēšanas valodu C++ un tās vispārējās programmēšanas (generic programming) līdzekļiem,
- izveidot programmatūras prasību specifikāciju un atbilstoši tai veikt programmatūras projektēšanu,
- izstrādāt programamtūras produktu, iespējams, mainot projektējumu,
- veikt izveidotās produkta vienībtestēšanu un visas pavadošās dokumentācijas sagatavošanu nodošanai.

Izstrādātajā bibliotēkā tiek piedāvātas arī vairākas citas operācijas – lauka parametru ģenerēšana, kvadrātsaknes vilkšana. Tāpat ir realizēti arī polinomi pār galīgajiem laukiem un daži skaitļu teorijas algoritmi. Šāds operāciju komplekts varētu bibliotēku padarīt pievilcīgu cilvēkiem, kas vēlas pētīt, piemēram, uz galīgajiem laukiem balstītus kļūdas koriģējošos kodus.

Bibliotēka tiek rakstīta valodā C++. Realizācija vairākas reizes izmantoto dažādus līdzīgus algoritmus un datu struktūras, piem. Eiklīda algoritmus (gan skaitļiem, gan

polinomiem) vai polinomus pār kādu lauku. Lai novērstu koda dublēšanos tika izmantoti C++ pieejamie vispārējās programmēšanas (generic programming) līdzekļi, tādējādi ievērojami samazinot nepieciešamā koda daudzumu. Arī bibliotēkas saskarne izmanto šo paradigmu – daži lauka parametri tiek nodoti izmantojot vispārējo programmēšanu.

Darbā liela vērība tiks pievērsta, lai padarītu programmatūras kodu saprotamu un viegli maināmu tā lietotājiem.

Darbs satur izstrādāto programmatūras prasību specifikāciju, programmatūras projektējuma aprakstu, testēšanas dokumentācijas fragmentus. Darbā ir iekļauta arī projektu pavadošā dokumentācija par izstrādes projekta organizāciju, izmantotajiem kvalitātes nodrošināšanas un konfigurāciju pārvaldības paņēmieniem, kā arī veiktos projekta darbietilpības novērtējumus. Tāpat darbā iekļauts arī reprezentatīvs izstrādātā pirmkoda fragments.

Kvalifikācijas darba izstrādē tika izmantoti avoti [1-5,7]. Konsultācijām par valodu C++ tika izmantoti avoti [6,8]. Dokumentācijas tapšana notika atbilstoši valsts standartiem LVS 68:1996 “Programmatūras prasību specifikācijas ceļvedis” un LVS 72:1996 “Ieteicamā prakse programmatūras projektējuma aprakstīšanai”; kvalifikācijas darba rakstīšanā tika izmantota Latvijas Universitātes izstrādne “Noslēguma darbu izstrādāšanas un aizstāvēšanas kārtība”.

1. nodaļa

Pamatdefinīcijas un teorēmas

1. definīcija. *Par* ***lauku*** *sauc kopu* K *kopā ar divām visur definētām binārām operācijām* $\oplus$ *un* $\otimes$ *tādām, ka:*

1. *visiem* $\mathfrak{a}$, $\mathfrak{b}$, $\mathfrak{c}$ *no* K *izpildās* $(\mathfrak{a} \oplus \mathfrak{b}) \oplus \mathfrak{c} = \mathfrak{a} \oplus (\mathfrak{b} \oplus \mathfrak{c})$,
2. *visiem* $\mathfrak{a}$, $\mathfrak{b}$, $\mathfrak{c}$ *no* K *izpildās* $(\mathfrak{a} \otimes \mathfrak{b}) \otimes \mathfrak{c} = \mathfrak{a} \otimes (\mathfrak{b} \otimes \mathfrak{c})$,
3. *visiem* $\mathfrak{a}$ *un* $\mathfrak{b}$ *no* K *izpildās* $\mathfrak{a} \oplus \mathfrak{b} = \mathfrak{b} \oplus \mathfrak{a}$,
4. *visiem* $\mathfrak{a}$ *un* $\mathfrak{b}$ *no* K *izpildās* $\mathfrak{a} \otimes \mathfrak{b} = \mathfrak{b} \otimes \mathfrak{a}$,
5. *eksistē tāds* F *elements* 0, *ka:*
   (a) *visiem* $\mathfrak{a}$ *no* K *izpildās* $\mathfrak{a} \oplus 0 = \mathfrak{a}$,
   (b) *katram* $\mathfrak{a}$ *no* K *eksistē tāds* $\mathfrak{b}$ *no* F, *ka* $\mathfrak{a} \oplus \mathfrak{b} = 0$,
6. *eksistē tāds* K *elements* 1, *ka:*
   (a) *visiem* $\mathfrak{a}$ *no* K *izpildās* $\mathfrak{a} \otimes 1 = \mathfrak{a}$,
   (b) *katram* $\mathfrak{a}$, *kas nav* 0 *eksistē tāds* F *elements* $\mathfrak{b}$, *ka* $\mathfrak{a} \otimes \mathfrak{b} = 1$
7. *visiem* $\mathfrak{a}$, $\mathfrak{b}$, $\mathfrak{c}$ *no* K *izpildās* $\mathfrak{a} \otimes (\mathfrak{b} \oplus \mathfrak{c}) = \mathfrak{a} \otimes \mathfrak{b} + \mathfrak{a} \otimes \mathfrak{c}$ *(distributīvā īpašība).*

K *elementus sauc par atbilstošā lauka elementiem.*

2. definīcija. *Ja 1. definīcijā minētā kopa* F *ir galīga, tad šādu lauku sauc par* ***Galuā lauku*** *jeb* ***galīgu lauku****.*

Bieži no galīgo lauku definīcijas izslēdz t.s. triviālo lauku ar vienu elementu, pieprasot, lai laukā būtu vairāk par vienu elementu. Tad ir spēkā šāda teorēma:

1. teorēma. *Katrā galīgā laukā ir tieši* $\mathfrak{p}^{\mathfrak{n}}$ *elementu kaut kādam pirmskaitlim* $\mathfrak{p}$ *un naturālam* $\mathfrak{n}$. *Atbilstošo* $\mathfrak{p}$ *sauc par lauka* ***harasteriku****, bet lauka elementu skaitu* $\mathfrak{p}^{\mathfrak{n}}$ *par tā* ***kārtu****. Šādu galīgu lauku parasts apzīmēt ar* $\mathsf{GF}(\mathfrak{p}^{\mathfrak{n}})$ *[1, 3].*

2. teorēma. *Katram pirmskaitlim* p *eksistē galīgs lauks* $\mathbb{Z}/p\mathbb{Z}$*, ko veido skaitļi pēc moduļa* p *ar parastajām saskaitīšanas un reizināšanas darbībām, kas veiktas pēc moduļa* p *[1, 3].*

Lai noskaidrotu vai skaitlis ir pirmskaitlis ir iespējams izmantot Millera-Rabina testu [1, 4].

3. definīcija. *Par* ***polinomu pār lauku*** F *jeb vienkārši* ***polinomu*** *sauc tādu visur definētu funkciju* P*, kuru var izteikt šādā veidā:* $P(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$*, kaut kādai* a_i *izvēlei no* F*. Lielāko indeksu* i*, kam* $a_i \neq 0$*, ja tāds eksistē, sauc par polinoma pakāpi. Ja visi polinoma koeficienti ir* 0*, tad saka, ka polinoma pakāpe nav definēta.*

4. definīcija. *Polinomu sauc par* ***reducējamu****, ja to var izteikt kā divu citu polinomu reizinājumu, kuriem abiem pakāpes ir definētas un lielākas par 0. Pretējā gadījumā polinomu sauc par* ***nereducējamu****. [3]*

Lai darbotos ar polinomiem galīgo lauku kontekstā mums būs noderīga šāda teorēma:

3. teorēma. *Katram pirmskaitlim* p *un jebkuram* n *eksistē* n*-tās pakāpes nereducējams polinoms pār lauku* $\mathbb{Z}/p\mathbb{Z}$*. [1, 3, 5]*

5. definīcija. *Par divu nenulles polinomu* P *un* Q *lielāko kopīgo dalītāju LKD(*P,Q*) sauc tādu polinomu* D*, ka ir spēkā šādas īpašības:*

1. *gan* P*, gan* Q *bez atlikuma dalās ar* D*,*
2. *ja gan* P*, gan* Q *bez atlikuma dalās ar* D'*, tad* D *dalās ar* D' *bez atlikuma,*

Polinomu lielāko kopīgo dalītāju var izrēķināt ar Eiklīda algoritmu Paplašinātais Eiklīda algoritms diviem kāda lauka elementiem u un v dod arī tādu to lineāru kombināciju, ka $a \cdot u + b \cdot v = \mathrm{LKD}(u,v)$ [2, 3].

Noskaidrot vai polinoms ir nereducējams var izmantojot šādu teorēmu:

4. teorēma. *Polinoms* P *pār lauku* $\mathbb{Z}/p\mathbb{Z}$ *ir nereducējams tad un tikai tad, ja LKD(*$P(x)$,$x^{p^i} - x$*) ir nenulles konstante katram* i*,* $1 \leqslant i \leqslant \frac{n}{2}$*, kur* n *ir polinoma* P *pakāpe [3].*

Lai ģenerētu nejaušu nereducējamu polinomu ir noderīga šāda teorēma:

5. teorēma. *Vismaz* $\frac{1}{2n}$ *no visiem* n*-tās pakāpes polinomiem pār fiksētu lauku* $\mathbb{Z}/p\mathbb{Z}$ *ir nereducējami [5].*

Galīgos laukus var uzdot dažādi, piemēram, norādot to saskaitīšanas un reizināšanas operāciju tabulas. Tomēr praksē vienīgā izmantotā metode ir **Kronekera konstrukcija** [1, 3, 5, 7].

6. teorēma. *Kronekera konstrukcija Ja* P *ir* n*-tās pakāpes nereducējams polinoms pār* $\mathbb{Z}/p\mathbb{Z}$*, tad visu* $n-1$*-ās pakāpes polinomu pār* $\mathbb{Z}/p\mathbb{Z}$ *kopa veido lauku ar parastajām polinomu saskaitīšanas un reizināšanas darbībām, kas veiktas pēc polinoma* P *moduļa. Šādu lauku pierasts apzīmēt ar* $(\mathbb{Z}/p\mathbb{Z})[x]/P(x)$.

Laukā $(\mathbb{Z}/p\mathbb{Z})[x]/P(x)$ elementam apgriezto elementu iespējams izrēķināt ar paplašināto Eiklīda algoritmu [1, 7]. To var izdarīt šādi: ja elementam nenulles u atbilst polinoms $Q(x)$, tad ar paplašināto Eiklīda algoritmu varam izrēķināt divus tādus polinomus $A(x)$ un $B(x)$, ka $A(x)\cdot P(X)+B(x)\cdot Q(x)=\mathrm{LKD}(P(x),\,Q(x))=1$ jeb $B(x)\cdot Q(x)\equiv 1 \pmod{P(x)}$. Tātad u^{-1} atbilst polinoms $Q(x)$. Var pamanīt, ka šeit būtiski tiek izmantota polinoma P nereducējamība.

Laukā $\mathrm{GF}(p^n)$ elementa a kvadrātsakni, t.i. tādu x, ka $x^2=a$, ja tāds eksistē, ir iespējams izrēķināt ar Tonelli algoritmu [1, 4], ja lauka harasterika p ir nepāra. Ja lauka harasterika ir 2, tad kvadrātsakni iespējams izrēķināt tieši (eksistē tieša (explicit) formula, kā to izrēķināt) [4].

7. teorēma. *Galuā laukā* $\mathrm{GF}(2^n)$ *elementa* u *kvadrātsakne ir šī lauka elements* $u^{2^{p-1}}$. *[4]*

Lai noskaidrotu vai galīgā lauka elements ir kvadrāts var izmantot šādu teorēmu:

8. teorēma. *Pieņemsim, ka* u *ir Galuā lauka* F *elements, pie tam* F *kārta ir nepāra. Tādā gadījumā* u *ir kvadrāts tad un tikai tad, ja* $u^{\frac{|F|-1}{2}}=1$ *[1].*

Daudzām ar laukiem saistītām problēmām nav labu deterministisku algoritmu, bet tām eksistē ērti lietojami Monte Karlo algoritmi. Monte Karlo algoritmi no klasiskiem varbūtiskiem algoritmiem atšķiras ar to, ka tie drīkst retos gadījumos (ar varbūtību mazāku nekā viena puse) dot nepareizu atbildi. Monte Karlo algoritmu piemērs ir Miller-Rabin tests, kas par skaitli n saka, ka tas ir salikts vai nu ar varbūtību 0 (ja n patiesībā ir pirmskaitlis) vai arī ar varbūtību $\frac{1}{2}$ (ja n – salikts skaitlis). Vairākas reizes atkārtojot Miller-Rabin testu mēs ar lielu ticamību varam teikt vai n ir vai nav pirmskaitlis.

2. nodaļa

Programmatūras prasību specifikācija

2.1. Ievads

2.1.1. Nolūks

Programmatūras prasību specifikācija ir paredzēta izstrādājamās Galuā lauku bibliotēkas prasību aprakstīšanai. Šis dokuments ir paredzēts programmatūras moduļa projektētājam un pasūtītājam, lai saskaņotu prasības pret produkta funkcionalitāti.

2.1.2. Darbības sfēra

Šajā dokumentā ir definētas prasības Galuā lauku bibliotēkai "mgflib".

Programmatūras produkta mērķis ir realizēt aritmētiskās darbības ar lietotāja uzdota Galuā lauka elementiem. Produkta paredzētais pielietojums ir pirmkoda bibliotēka, ko lietotājs iekļaus citos savos produktos, iespējams, papildinot tās iespējas.

2.1.3. Saistība ar citiem dokumentiem

Programmatūras prasību specifikācija ir kvalifikācijas darba "Galuā lauku realizācija, izmantojot vispārējās programmēšanas paradigmu" sastāvdaļa.

Šis dokuments ir izstrādāts saskaņā ar standartu LVS 68:1996 "Programmatūras prasību specifikācijas ceļvedis". Šo specifikāciju ir ieteicams lietot kopā ar kādu paskaidrojošu dokumentu par Galuā laukiem, piemēram, pamatdefinīcijām (šī darba 1. nodaļa) vai kādu no grāmatām par šo tēmu [1, 3, 4, 5].

2.2. Vispārējais apraksts

2.2.1. Produkta perspektīva

Izstrādājamais produkts ir neatkarīga pirmkoda bibliotēka, kas tiks lietota kā lielāku programmatūras projektu sastāvdaļa un nodrošinās funkcionalitāti, kas saistīta ar operācijām Galuā laukos.

2.2.2. Produkta funkcijas

Programmatūras produktam ir jārealizē objekts – Galuā lauka elements, kuru konstruē lietotājs. Šī objekta iekšienē jārealizē četras aritmētiskās darbības (saskaitīšana, atņemšana, reizināšana, dalīšana), multiplikatīvā inversa atrašana, un vienādības pārbaude lauka elementiem. Papildus jārealizē kvadrātsaknes vilkšana Galuā laukā.

Programmatūrai nepieciešams atbalstīt laukus $\mathsf{GF}(\mathsf{p}^{\mathsf{n}})$ ar dažādām harasterikas p vērtībām.

2.2.3. Lietotāja raksturiezīmes

Bibliotēkai ir paredzētas tikai iekšējās programmēšanas saskarnes un tās lietotāji ir programmētāji, kuru zināšanām vajadzētu ietvert pamatzināšanas par darbībām laukos, objektorientēto programmēšanu un vispārējo programmēšanu. Saskarne paredzēta izmantojot atbilstošus C++ header failus.

2.2.4. Pieņēmumi un atkarības

Tiek pieņemts, ka bibliotēka tiks lietota galvenokārt pētniecības projektos, tāpēc prasības pret atmiņas un laika sarežģītību tiek specificētas tikai asimptotiski.

2.3. Konkrētās prasības

2.3.1. Funkcionālās prasības

2.3.1.1. Lauka elementa uzdošana

Bibliotēkai ir jāļauj uzdot lauka $\mathsf{Z/pZ[T]/P(T)}$ elementu, kuram atbilst polinoms $\mathsf{Q(T)}$, ievadā saņemot lauka harasterikas p vērtību un polinomus P, Q, kurus jāvar uzdot ar to koeficientu sarakstiem. Gan harasterika p, gan P pakāpe n būs zināmas kompilācijas laikā.

Ja p nav pirmskaitlis jebkuras operācijas ar šāda deģenerēta lauka elementiem rezultāts drīkst būt nedefinēts, bet bibliotēkai jāparedz funkcija, kas noskaidro vai dotais p ir pirmskaitlis.

2.3.1.2. Pārbaude vai skaitlis ir pirmskaitlis

Ievads Funkcijai jāpārbauda vai ievadā dotais skaitlis $\mathtt{n}$ ir pirmskaitlis. Zināms, ka $\mathtt{n}$ var ietilpināt tipā `unsigned short`, tomēr vēlāk var tikt uzstādīta prasība atbalstīt patvaļīga garuma skaitļus.

Ievade Funkcijas ievadā tiek padots viens naturāls skaitlis $\mathtt{n}$, par kuru jānoskaidro, vai tas ir pirmskaitlis.

Apstrāde Funkcijai izmantojot kādu no pirmskaitļu testiem jānoskaidro vai $\mathtt{n}$ ir pirmskaitlis; šai pārbaudei drīkst izmantot Monte Karlo algoritmu.

Izvade Funkcijai izvadā jādod `bool` vērtība `true`, ja $\mathtt{n}$ ir pirmskaitlis vai `false` pretējā gadījumā.

2.3.1.3. Nejauši izvēlēta lauka ģenerēšana

Bibliotēkai ir jānodrošina nejauši izvēlēta lauka $\mathsf{GF}(\mathtt{p}^{\mathtt{n}})$ ģenerēšana, ja gan $\mathtt{p}$, gan $\mathtt{n}$ var ietilpināt tipā `unsigned char`.

Nejauši izvēlēta lauka ģenerēšanai drīkst izmantot varbūtisku algoritmu.

2.3.1.4. Aritmētiskās operācijas laukā

Ievads Līdzīgas prasības attiecināmas visām aritmētiskajām operācijām, ko iespējams veikt ar Galuā lauka elementiem, t.i., funkcijām, kas nodrošina Galuā lauka elementu saskaitīšanu, atņemšanu, reizināšanu un dalīšanu.

Ievade Funkcijas ievadā tiek padoti divi Galuā lauka elementi $\mathtt{a}$ un $\mathtt{b}$, kas pieder vienam un tam pašam Galuā laukam.

Apstrāde Funkcijai jāpārliecinās vai elementi patiešām pieder vienam un tam pašam Galuā laukam, kā arī dalīšanas gadījumā – vai dalītājs $\mathtt{b}$ nav lauka aditīvā identitāte (nulle). Ja lauki atšķiras vai tiek pieprasīta dalīšana ar nulli funkcijai ir jāsignalizē izņēmuma situācija (*throw an exception*) un tās darbība tālāk netiek turpināta.

Pretējā gadījumā tiek pielietota pieprasītā aritmētiskā operācija un kā funkcijas rezultāts tiek atgriezts šī paša Galuā lauka elements, atbilstošās aritmētiskās operācijas rezultāts.

Izvade Funkcijas rezultāts ir viens Galuā lauka elements – atbilstošās aritmētiskās operācijas rezultāts.

2.3.1.5. Kvadrātsaknes vilkšana

Ievads Funkcija rēķina dotā elementa kvadrātsakni laukā, t.i. elementam a izrēķina tādu x, ka $x^2 = a$, ja tāds eksistē.

Ievade Funkcijas ievadā tiek padots viens Galuā lauka elements a, kuram nepieciešams izrēķināt kvadrātsakni.

Apstrāde Vispirms tiek pārbaudīts vai elements ir kvadrāts, t.i., vai eksistē izrēķināmā kvadrātsakne. Ja elements nav kvadrāts kā funkcijas rezultāts ir jāatdod šī paša lauka elements nulle.

Pretējā gadījumā tiek pielietots kāds no algoritmiem kvadrātsaknes rēķināšanai, piemēram, Tonelli algoritms [1, 4]. Atļauts izmantot Monte Karlo algoritmu kvadrātsaknes rēķināšanai.

Izvade Funkcijas rezultāts ir viens Galuā lauka elements – atbilstoši nulle (ja elements nav kvadrāts) vai tāds x, kam $x^2 = a$.

2.3.2. Veiktspējas prasības

Bibliotēkai jāspēj darboties ar laukiem ar kārtu, kas ietilpst vismaz `unsigned int`. Tālāk dotās veiktspējas prasības ir specificētas pieņemot, ka operācijas ar veseliem skaitļiem izpildās konstantā laikā.

Darbības laukā $GF(p^n)$ ir jārealizē saskaņā ar šādām veiktspējas prasībām:

- lauka elementa uzdošanai jānotiek laikā $O(n)$,
- elementa no nejauša lauka uzdošanai jānotiek laikā, kas vidēji ir $O(n^4)$.
- saskaitīšana un atņemšana jāveic laikā, kas nepārsniedz $O(n)$,
- reizināšanas asimptotiskais novērtējums nedrīkst pārsniegt $O(n^2)$, dalīšanas - $O(n^3)$,
- kvadrātsaknes vilkšanai jānotiek laikā, kas vidēji ir $O(n^4)$.

2.3.3. Aparatūras ierobežojumi

Bibliotēkai ir jākompilējas gan Windows, gan kādā Unix vidē, gan 32-bitu, gan (neobligāti) 64-bitu arhitektūrai. Lauka $GF(p^n)$ elementa uzturēšanai atmiņā drīkst izmantot ne vairāk kā $O(cn)$ atmiņas vienības, kur c – nepieciešamais atmiņas daudzums, lai noglabātu vienu skaitli, kas ir mazāks par p.

2.3.4. Nefunkcionālās prasības

Programmproduktam tiek izvirzītas šādas nefunkcionālās prasības:

- bibliotēkas izejas tekstiem ir jābūt atbilstošiem standartam ISO/IEC 14882:1998 jeb C++98,
- izejas tekstiem jākompilējas ar GNU C++ kompilatoru un (neobligāti) ar Microsoft Visual C++ kompilatoru,
- programmatūras pirmkodaam jābūt noformēti atbilstoši kādam vispāratzītam C++ kodēšanas standartam, piem. Google C++ Style Guide [8], pieļaujot saprātīgas atkāpes no koda vizuālā (ne semantiskā) noformējuma.

2.3.5. Ārējās saskarnes prasības

Izstrādājamais produkts ir C++ klases un funkcijas. Bibliotēkas ārējai saskarnei ir jāapmierina šādas prasības:

- visa bibliotēkas funkcionalitāte, kas pieejama no ārienes ir jāievieto vārdu telpā (namespace) `mgflib`,
- bibliotēkai jārealizē klase "GaloisField", kas reprezentē Galuā lauka elementu, aritmētiskās operācijas jārealizē kā šīs klases operatori `operator+`, `operator-`, `operator*`, `operator/`. Kvadrātsaknes vilkšana jārealizē kā šīs klases metode "sqrt",
- pārbaude vai skaitlis ir pirmskaitlis ir jārealizē kā `bool` funkcija `is_prime`.

3. nodaļa

Programmatūras projektējuma apraksts

3.1. Ievads

3.1.1. Dokumenta nolūks

Programmatūras projektējuma apraksts (PPA) ir paredzēts, lai aprakstītu kā izstrādājamās Galuā lauku bibliotēkas "mgflib" programmatūras prasību specifikācijā minētās prasības tiks realizētas projektā. Dokuments ir radīts, lai atvieglotu analīzi, plānošanu, projekta implementēšanu un lēmumu pieņemšanu projekta attīstīšanā; tā mērķauditorija ir bibliotēkas izstrādātāji, uzturētāji un testētāji.

3.1.2. Darbības sfēra

Izstrādājamais projekts ir neatkarīga bibliotēka, kas nodrošina darbību ar Galuā laukiem. Programmatūras produkta uzdevums ir atvieglot projektu, kas izmanto Galuā laukus, realizāciju, piedāvājot gatavu kodu, kas realizē lielu daļu no biežāk nepieciešamajām Galuā lauku operācijām.

3.1.3. Definīcijas

Šajā dokumentā lietotie projektam specifiskie jēdzieni ir definēti darba 1. nodaļā.

3.2. Saistība ar citiem dokumentiem

Programmatūras projektējuma apraksts ir kvalifikācijas darba "Galuā lauku realizācija, izmantojot vispārējās programmēšanas paradigmu" sastāvdaļa.

Šis dokuments ir lietojams kopā ar Galuā lauku bibliotēkas "mgflib" programmatūras prasību specifikāciju.

Dokuments ir izstrādāts vadoties pēc Latvijas Valsts standarta LVS 72:1996 "Ieteicamā prakse programmatūras projektējuma aprakstīšanai".

3.3. Dekompozīcijas apraksts

3.3.1. Ievads

Balstoties uz programmatūras prasību specifikāciju un izpētīto literatūru ir saprotami programmatūras prasību specifikācijā minēto objektu – Galuā lauka elementu – būvēt šādi:

- izveidot entītiju `PrimeField`, kas uzturēs lauka $\mathbb{Z}/\mathtt{p}\mathbb{Z}$ elementu,
- izveidot entītiju `Polynomial`, kas uzturēs polinomus pār patvaļīgu lauku `F`,
- izveidot entītiju `GaloisField`, kas, izmantojot polinomus pār lauku $\mathbb{Z}/\mathtt{p}\mathbb{Z}$, atbildīs Galuā lauka elementam.

Šādā veidā klases `GaloisField` lietošanas scenārijs, lai darbotos ar kādu lauku $\mathtt{GF}(\mathtt{p}^{\mathtt{n}})$ ir:

- izvēlēties un fiksēt `p` un `n`,
- ģenerēt vai kā citādi uzdot nereducējamu polinomu pār `PrimeField<p>`,
- izmantojot šo polinomu uzdot `GaloisField` elementa moduli (`modulus`), atbilstoši Kronekera konstrukcijai.

Ievērosim, ka gan `PrimeField::operator/`, gan `GaloisField::operator/` nodrošināšanai var izmantot vienu un to pašu paplašināto Eiklīda algoritmu tikai dažādiem tipiem. Gan kvadrātsaknes vilkšanai, gan skaitļa pirmskaitlības pārbaudei (*primality testing*)var izmantot operāciju `powmod` – kāpināšanu naturālā pakāpē pēc dota moduļa. Arī operācijas `powmod` implementācija skaitļiem un polinomiem ir ļoti līdzīga – atšķiras tikai tipi.

Tāpēc ieviesīsim divas vispārīgas (*generic*) funkcijas, attiecīgi `ExtendedEuclideanAlgorithm` un `powmod`.

3.3.2. Entītiju dekompozīcija

3.3.2.1. `is_prime`

Tips Funkcija

Nolūks Noskaidrot vai skaitlis `p` ir pirmskaitlis

Funkcija Izmantojot Miller-Rabin testu pārbauda vai $\mathtt{p}$ ir pirmskaitlis; pareizā rezultāta varbūtība ir vismaz $1 - 2^{-\mathtt{n}}$ jebkurai lietotāja uzdotai precizitātei $\mathtt{n}$.

3.3.2.2. `ExtendedEuclideanAlgorithm`

Tips Funkcija

Nolūks Realizēt paplašināto Eiklīda algoritmu.

Funkcija Izmantojot aritmētiskās operācijas no diviem lauka elementiem $\mathtt{a}$ un $\mathtt{b}$ izrēķina to LKD $\mathtt{d}$ un tādu $\mathtt{a}$ un $\mathtt{b}$ lineāru kombināciju, kuras vērtība ir $\mathtt{d}$

3.3.2.3. `powmod`

Tips Funkcija

Nolūks Realizēt kāpināšanu patvaļīgā laukā, rezultātu ņemot pēc dota moduļa.

Funkcija Izmantojot lauka operāciju `operator*` realizē kāpināšanu naturālā pakāpē, laikā $O(\log \mathtt{degree})$.

3.3.2.4. `PrimeField`

Tips Klase

Nolūks Implementēt lauku $\mathbb{Z}/\mathtt{p}\mathbb{Z}$.

Funkcija Uztur informāciju par elementa atlikumu klasi. Implementē aritmētiskās operācijas šī lauka elementiem.

Atkarības Funkcija `ExtendedEuclideanAlgorithm`

3.3.2.5. `Polynomial`

Tips Klase

Nolūks Realizēt polinomus pār patvaļīgu lauku $\mathtt{F}$.

Funkcija Uztur polinoma pakāpi un koeficientu sarakstu. Implementē aritmētiskās operācijas ar citiem polinomiem pār šo pašu lauku.

3.3.2.6. `IsIrreducible`

Tips Funkcija

Nolūks Noskaidrot vai polinoms pār doto lauku ir nereducējams.

Funkcija Izmantojot 4. teorēmas ideju un funkcijas `powmod`, `ExtendedEuclideanAlgorithm` noskaidro polinoma reducējamību.

Atkarības Funkcija `powmod`, funkcija `ExtendedEuclideanAlgorithm`.

3.3.2.7. `GenerateRandomPolynomial`

Tips Funkcija

Nolūks Ģenerēt pseidogadījumu polinomu pār doto lauku, ieskaitot moniska polinoma ģenerēšanu (monisks polinoms ir tāds viena mainīgā polinoms, kura vecākais koeficients ir 1).

Funkcija Izmantojot C++ standarta bibliotēkā iekļauto pseidogadījuma skaitļu ģeneratoru izveido pseidogadījumu koeficientu sarakstu un to padodot klases `Polynomial` konstruktoram iegūst pseidogadījuma polinomu.

Atkarības Klase `Polynomial`.

3.3.2.8. `GenerateIrreduciblePolynomial`

Tips Funkcija

Nolūks Ģenerēt pseidogadījumu nereducējamu polinomu pār doto lauku.

Funkcija Ģenerē pseidogadījumu polinomus-kandidātus, kamēr kāds no tiem ir nereducējams. Ievērojot 5. teorēmu, maziem n šis process ar lielu varbūtību beigsies pēc neliela mēģinājumu skaita.

Atkarības Funkcija `GenerateRandomPolynomial`, funkcija `IsIrreducible`.

3.3.2.9. `GaloisField`

Tips Klase

Nolūks Realizē galveno prasību specifikācijas objektu – Galuā lauka $\mathrm{GF}(p^n)$ elementu.

Funkcija Uztur informāciju par elementam atbilstošo polinomu un lauka moduli. Realizē aritmētiskās operācijas ar citiem šī lauka elementiem. Realizē metodi `sqrt` (kvadrātsaknes aprēķinu).

Atkarības Funkcija `ExtendedEuclideanAlgorithm`, funkcija `powmod`, klase `Polynomial`, klase `PrimeField`, funkcija `GenerateRandomPolynomial`.

3.3.3. Funkciju dekompozīcijas projektējums

Šis projektējums parāda iepriekš minēto klašu sīkāku dekompozīciju.

3.3.3.1. Modulis `PrimeField`

Identificējums	**Nolūks/funkcija**
`PrimeField` (konstruktors)	Izveidot objektu un inicializēt tā atlikumu klases atribūtu
`operator==`, `operator!=`	Pārbaudīt vai dotā instance un parametrā nodotā instance apraksta, resp., neapraksta vienu un to pašu atlikumu klasi
`inverse`	Atgriezt dotajam elementam apgriezto elementu, ja tāds eksistē, pretējā gadījumā paziņot par izņēmuma situāciju.
`operator+`, `operator-`, `operator*`, `operator/`, `operator%`	Veikt atbilstošās aritmētiskās darbības ar doto un parametrā nodoto instanci, atgriežot rezultātu kā jaunu klases `PrimeField` instanci. Kļūdas (piem. dalīšana ar nulles elementu) gadījumā paziņot par izņēmuma situāciju.

3.3.3.2. Modulis `Polynomial`

Identificējums	**Nolūks/funkcija**
`Polynomial` (konstruktors)	Izveidot objektu un inicializēt tā pakāpi un koeficientu sarakstu
`evaluate`	Aprēķināt polinoma vērtību dotajā punktā [1]
`minimize`	Panākt, ka polinoma vecākais koeficients ir nenulles vai arī uzstādīt tā pakāpi uz negatīvu
`make_zero`	Padarīt polinomu par nulles polinomu
`is_zero`	Noskaidrot vai dotais polinoms ir nulles polinoms
`shift`	Pareizināt polinomu `P(x)` ar x^n, kur `n` – vesels, atmetot `x` negatīvās pakāpes, ja tādas rodas
`operator==`, `operator!=`	Pārbaudīt vai dotais polinoms sakrīt un parametrā nodotais polinoms sakrīt, resp., nesakrīt pēc to koeficientu sarakstiem
`operator+`, `operator-`, `operator*`, `operator/`, `operator%`	Veikt atbilstošo aritmētisko darbību ar doto polinomu un parametrā nodoto objektu (vai nu konstanti vai polinomu) un atgriezt šīs operācijas rezultātu kā jaunu klases `Polynomial` instanci. Kļūdas (piem. dalīšana ar nulles polinomu) gadījumā paziņot par izņēmuma situāciju.
`divmod`	Atgriezt dotā polinoma dalīšanas ar parametrā nodotā polinoma rezultātus – dalījumu un atlikumu kā jaunu klases `Polynomial` instanču pāri

Visām klases `Polynomial` metodēm to nolūks ir skaidrs (tas izriet no entītiju projektējuma), izņemot dažas, kuru nolūki ir paskaidroti zemāk:

- Par metodi `minimize`. Vairākas citas klases `Polynomial` metodes izmanto invariantu, ka polinoma vecākais koeficients ir nenulles. Lietotāja tiešas darbības ar polinoma koeficientu sarakstu šo invariantu var izjaukt, tāpēc vajadzētu paredzēt vai nu *wrapper* piekļuvei polinoma koeficientu sarakstam vai arī ieviest metodi, kas šo invariantu atjauno. Projektējumā ir izvēlēts ieviest šādu metodi `minimize`.

- Metode `shift` loģiski seko no `operator/` implementācijas [2], kā arī ir bieža operācija darbā ar polinomiem, ko varētu vēlēties lietotājs.

- Metodes `divmod` ieviešanu pamato fakts, ka dalījums un atlikums tiek rēķināti kopā, bet katrs no operatoriem `operator/`, `operator%` atgriež tikai vienu rezultātu (dalījumu vai atlikumu). Tā kā reizēm ir vajadzīgs gan dalījums, gan atlikums

[1] Strikti runājot šo metodi neizmanto nevienas prasības realizācija pašreizējajā projektējumā, tomēr polinoma vērtības aprēķins varētu būt derīgs bibliotēkas potenciālajiem lietotājiem [5, 7]

(piem. Eiklīda algoritmā), tad operatoru `operator/` un `operator%` vietā var lietot šo metodi.

3.3.3.3. Modulis `GaloisField`

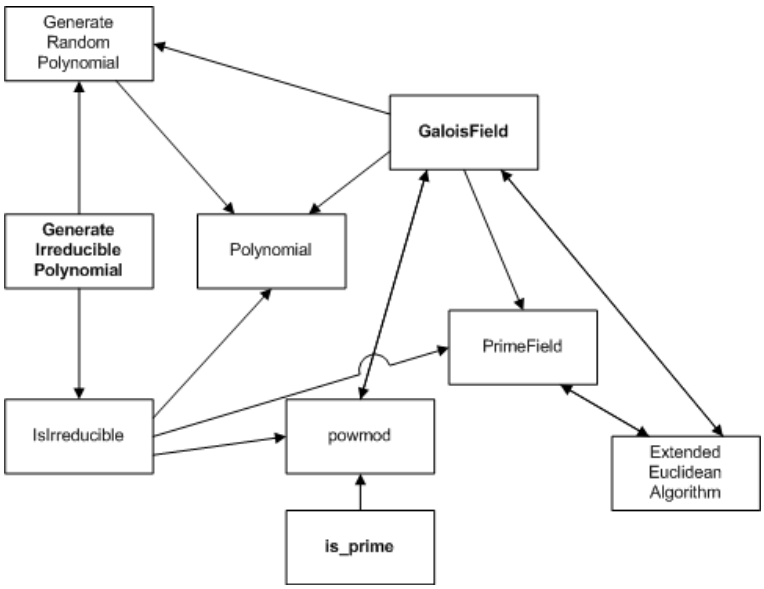
Lielākā daļa no "GaloisField" metodēm tieši izriet no programmatūras prasību specifikācijas, tomēr ir jāatzīmē kāda nianse. Tā kā projektējumā ir izvēlēts norādi uz attiecīgā lauka moduli glabāt katrā "GaloisField" instancē (skat. 3.5.9.), tad pirms katras aritmētiskās operācijas veikšanas ir nepieciešams pārliecināties vai šie moduļi sakrīt; to nevar izdarīt izmantojot tikai template parametru sakritības pārbaudi kompilācijas laikā. Tamdēļ tiks ieviesta metode "check_moduli", kas veiks moduļu pārbaudi.

Identificējums	**Nolūks/funkcija**
`GaloisField` (konstruktors)	Izveidot objektu un inicializēt tā moduļa polinomu un elementu raksturojošo polinomu (atbilstoši Kronekera konstrukcijai)
`check_moduli`	Pārliecināties vai dotā Galuā lauka elementa moduļa polinoms sakrīt ar parametros nodotā Galuā lauka elementa moduļapolinomu, nesakrišanas gadījumā signalizējot izņēmuma situāciju.
`operator==`, `operator!=`	Noskaidrot vai dotais Galuā lauka elements sakrīt, resp., nesakrīt ar parametros nodotā Galuā lauka elementu
`inverse`	Atgriezt dotajam elementam apgriezto elementu, ja tāds eksistē, pretējā gadījumā paziņot par izņēmuma situāciju.
`operator+`, `operator-`, `operator*`, `operator/`, `operator%`	Veikt atbilstošās aritmētiskās darbības ar doto un parametrā nodoto instanci, atgriežot rezultātu kā jaunu klases `GaloisField` instanci. Kļūdas (piem. dalīšana ar nulles elementu) gadījumā paziņot par izņēmuma situāciju.
`is_square`	Noskaidrot vai dotais Galuā lauka elements ir kvadrāts
`sqrt`	Atgriezt tādu lauka elementu, kurš reizināts pats ar sevi ir vienāds doto elementu (t.i. izvilkt kvadrātsakni). Šāda elementa neeksistences gadījumā atgriezt nulli.

3.4. Atkarību projektējums

Zemāk dotajā entītiju atkarību projektējumos no entītijas `a` uz entītiju `b` ir šķautne, ja `b` ir atkarīga no `a`.

Funkciju atkarību projektējums tā salīdzinoši lielās sarežģītības dēļ darbā grafiskā veidā nav parādīts.



3.1. zīm.: Entītiju atkarību grafs

3.5. Detalizētais projektējums

Skaidrības labad tika izvēlēts detalizēto projektējumu un funkciju saskarnes projektējumu veidot vienotu, jo detalizētājā projektējumā bieži nākas atsaukties uz funkciju saskarnes projektējumu.

3.5.1. `is_prime`

Nosaukums `is_prime<T>`

Ievade Šīs funkcijas ievadē tiek padots:

1. Pārbaudāmo pirmskaitli uzglabājošais tips T (padots ar vispārējās programmēšanas palīdzību),
2. Tipa T skaitlis p – pārbaudāmais pirmskaitlis,
3. Vesels skaitlis num_tries – cik Miller-Rabin testa iterācijas veikt

Apstrāde num_tries reizes tiek atkārtots Miller-Rabin tests, t.i.:

- tiek nejauši izvēlēts skaitlis a starp $[1, 2, .., p-1]$,
- $p-1$ tiek izteikts formā $2^s d$, kur d ir nepāra,
- tiek pārbaudīts vai kādam r $(0 \leqslant r \leqslant s-1)$ izpildās $a^{2^r} \equiv -1 (mod p)$,
- ja šāds r eksistē, tad p nav pirmskaitlis un funkcijas rezultāts ir "false", pretējā gadījumā tests atbildi nedod

Ja visās pārbaudēs tests atbildi nav devis, tad funkcijas rezultāts ir "true". Varbūtība, ka "true" tiek pateikts saliktam skaitlim jeb nepareizās atbildes varbūtība, pēc Miller-Rabin teorēmas ir ne lielāka par $1 - 2^{-\mathrm{num\_tries}}$.

Izvade `bool` vērtība, kas ir `true`, ja skaitlis ir pirmskaitlis, bet `false` pretējā gadījumā.

3.5.2. `ExtendedEuclideanAlgorithm`

Nosaukums `ExtendendEuclideanAlgorithm<T>`

Ievade Šīs funkcijas ievadā tiek padots:

1. Elementu, kuriem tiks izpildīts algoritms, tips T (padots ar vispārējās programmēšanas palīdzību), kas reprezentē Eiklīda apgabalu,
2. Divi tipa `T` elementi u un v,
3. Divi tipa `T` elementi `zero` un `one`, atbilstoši tādi, ka `zero` ir neitrālais elements saskaitīšanai apgabalā `T`, bet `one` ir neitrālais elements reizināšanai apgabalā `T`.

Apstrāde Rezultāta iegūšanai tiek izmantots paplašinātais Eiklīda algoritms [1, 2]. Tā soļi ir samērā tehniski, tāpēc šeit nav atkārtoti. Algoritma darbības laiks ir $O(\log \max(f(u), f(v)))$, kur f – atbilstošā Eiklīda funkcija Eiklīda apgabalam T. Polinomiem šāda funkcija ir funkcija deg, veseliem skaitļiem šāda funkcija ir identitātes funkcija.

Funkcija izmantojot template specialization noskaidro vai dotajam tipam `T` var uzzināt dalījumu un dalīšanas atlikumu vienlaicīgi. Ja tā, tad tiek izmantota šī metode, piemēram, klasei `Polynomial`.

Izvade Pāris `std::pair<T,T>`, kura elementi (apzīmēsim tos ar x un y) ir tādi, ka $u \cdot x + v \cdot y = \mathrm{LKD}(u, v)$.

3.5.3. `powmod`

Nosaukums `powmod<T>`

Ievade Šīs funkcijas ievadā tiek padots:

1. Elementa, kura pakāpe tiks izrēķināta, jeb bāzes tips `T` (padots ar vispārējās programmēšanas palīdzību),
2. Tipa `T` elements e – bāze,
3. `unsigned int` skaitlis power – pakāpe,
4. Tipa `T` elements `p`, pēc kura moduļa tiks veikti aprēķini

Apstrāde Funkcija izrēķina $e^{power} \pmod p$, savas darbības laikā uzturot invariantu, ka atbilde ir $r^{power} \cdot q \equiv e^{power} \pmod p$ un katrā solī power samazinot aptuveni par pusi, atbilstoši izmainot r un q. Algoritma darbības laiks ir $O(\log power)$ [1, 2, 4].

Izvade Tipa T elements, kas vienāds ar $e^{power} (mod p)$.

3.5.4. PrimeField

Visiem šīs klases objektiem ar vispārējās programmēšanas palīdzību ir norādīta lauka $\mathbb{Z}/p\mathbb{Z}$ harasterika P. Klasei ir viens iekšējais mainīgais - residue_class, kas norāda to, kurai atlikumu klasei pieder klases instances elements. Gan P, gan residue_class tips ir int.

Klases metodes uztur invariantu, ka $0 \leqslant$ residue_class $<$ P.

3.5.4.1. PrimeField

Nosaukums PrimeField<P>::PrimeField

Ievade Šim konstruktoram ir divas versijas: bez parametriem, un ar int tipa parametru elem.

Apstrāde Tiek izveidots atbilstošais lauka elements, ja izsaukts konstruktors ar parametru elem, tad par residue_class vērtību tiek uzstādīts tam atbilstošā atlikums, kas vienmēr būs no kopas $[0, 1, \ldots, P - 1]$, pretējā gadījumā residue_class vērtība tiek uzstādīta kā 0.

Ja tiek mēģināts izsaukt konstruktoru elementam ar nepozitīvu P, tiek signalizēta izņēmuma situācija.

3.5.4.2. operator==, operator!=

Nosaukums PrimeField<P>::operator==, PrimeField<P>::operator!=

Ievade Funkcija ievadē saņem elementu, kuram operators tika izsaukts, kā arī otru PrimeField<P> elementu other, ar kuru notiks salīdzināšana

Apstrāde Vienādības pārbaude notiek noskaidrojot vai dotā elementa un other mainīgie residue_class sakrīt pēc moduļa P. Nevienādības pārbaude tiek realizēta kā negācija no atbilstošās vienādības pārbaudes rezultāta.

Izvade bool vērtība true, ja elementi sakrīt, resp., nesakrīt, vai false pretējā gadījumā.

3.5.4.3. `inverse`

Nosaukums `PrimeField<P>::inverse`

Ievade Funkcija ievadē saņem elementu, kuram tā tiek izsaukta.

Apstrāde Ja `inverse` tiek mēģināts izsaukt elementam ar `residue_class` vienādu ar `0` (lauka nulles elementam), tad tiek signalizēta izņēmuma situācija.

Pretējā gadījumā izmantojot paplašināto Eiklīda algoritmu (funkcija `ExtendedEuclideanAlgorithm`) skaitļiem `residue_class` un `P` tiek noskaidroti tādi `x` un `y`, ka $\texttt{x}\cdot\texttt{residue\_class}+\texttt{y}\cdot\texttt{P}=1$ (jo LKD $= 1$). Redzams, ka tad arī $\texttt{x}\cdot\texttt{residue\_class}\equiv 1 \pmod{\texttt{p}}$ un elements atlikumu klasē `x` der par meklēto atbildi.

Izvade `PrimeField<P>` elements – dotā elementa multiplikatīvais inverss.

3.5.4.4. `operator+`, `operator-`, `operator*`, `operator%`

Nosaukums `PrimeField<P>::operator+`, `PrimeField<P>::operator-`, `PrimeField<P>::operator*`, `PrimeField<P>::operator%`

Ievade Funkcija ievadē saņem elementu, kuram operators tika izsaukts, kā arī otru `PrimeField<P>` elementu `other`, ar kuru notiks nepieciešamā aritmētiskā operācija

Apstrāde Saskaitīšana, atņemšana un reizināšana tiek realizētas tieši – izdarot minēto darbību ar abu ievadā saņemto elementu mainīgajiem `residue_class` un tās rezultātu padodot `PrimeField<P>` konstruktoram.

Dalīšana tiek realizēta kā reizināšana ar `other.inverse()`, pārbaudot vai `other` nav nulles elements. Dalīšanas ar nulli gadījumā tiek signalizēta izņēmuma situācija.

Dalīšanas atlikums vienmēr ir nulle, jo laukā jebkuru elementu var izdalīt ar jebkuru nenulles elementu. Izņēmums ir atlikums dalot ar nulli, par ko tiek signalizēta izņēmuma situācija.

Izvade `PrimeField<P>` elements – pieprasītās aritmētiskās operācijas rezultāts.

3.5.5. `Polynomial`

Visiem šīs klases objektiem ar vispārējās programmēšanas palīdzību ir norādīts pār kādu lauku `F` ir šie polinomi.

Klasei ir divi iekšējie mainīgie:

- `int` tipa mainīgais `degree`, kas ir vienāds ar polinoma pakāpi vai negatīvs, ja polinoms ir nulles polinoms,

- `std::vector<G>` tipa mainīgais `coeff`, kas satur polinoma koeficientu sarakstu, numurējot tos tā, ka `coeff.begin()` rāda uz koeficientu pie pakāpes x^0, utt.

Visas klases funkcijas/metodes uztur invariantu, ka ja `coeff` ir netukšs, tad tā pēdējais elements (polinoma vecākais koeficients) nav lauka `F` nulles elements. Vēl tiek uzturēts invariants, ka `degree` vienmēr ir vienāds ar `coeff.size()` -1.

Tāpat arī visas klases funkcijas/metodes katrā izsaukumā saņem arī lauku `F`, kas nodots ar vispārējās programmēšanas palīdzību.

3.5.5.1. `Polynomial`

Nosaukums `Polynomial<F>::Polynomial`

Ievade Šim konstruktoram ir trīs varianti:

1. bez parametriem,
2. ar tipa `G` parametru `zero_coeff`,
3. ar tipa `std::vector<G>` parametru `init_coeff`.

Apstrāde Ja konstruktors tika izsaukts bez parametriem, tad tiek izveidots nulles polinoms – polinoma iekšējais mainīgais `degree` tiek uzstādīts kā -1 un tā koeficientu saraksts ir tukšs.

Ja konstruktors tika izsaukts ar parametru `zero_coeff`, tad tiek izveidots polinoms ar pakāpi `0`, kuram brīvais loceklis kļūst vienāds ar `F(zero_coeff)`, bet ja `F(zero_zoeff)` ir `0`, tad tiek izveidots nulles polinoms (skat. augstāk).

Ja konstruktors tika izsaukts ar parametru `init_coeff`, tad polinoms tiek uzdots ar koeficientiem no `init_coeff` tādā veidā, ka init_coeff sākuma elements (*(init_coeff.begin())) kļūst par polinoma nulto koeficientu, nākošais – par otro, utt. Tālāk tiek izsaukta metode `minimize` (skat. 3.5.5.3.), atbrīvotos no tiem polinoma vecākajiem koeficientiem, kas varētu būt nulles.

3.5.5.2. `evaluate`

Nosaukums `Polynomial<F>::evaluate`

Ievade Funkcija ievadē saņem polinomu, kuram tā izsaukta, kā arī lauka `F` elementu `x` – punktu, kurā tiks rēķināta polinoma vērtība.

Apstrāde Tiek izrēķināta polinoma vērtība punktā `x` izmantojot Hornera shēmu [2]. Tas garantē, ka tiks izmantotas $O(\texttt{degree}^2)$ reizināšanas un $O(\texttt{degree})$ saskaitīšanas darbības.

Izvade Lauka `F` elements, kas vienāds ar `P(x)` – polinoma vērtību prasītajā punktā.

3.5.5.3. `minimize`

Nosaukums `Polynomial<F>::minimize`

Ievade Metode ievadē saņem polinomu, kuram tā izsaukta.

Apstrāde Reversi iterējot polinoma koeficientu sarakstu tiek noskaidrots tāds mazākais saraksta indekss `i`, ka `coeff_list(i)` nav lauka `F` nulles elements. Ja šāds indekss eksistē, tad tiek izdzēsti visi koeficienti ar lielākiem indeksiem, pretējā gadījumā tiek izdzēsti visi polinoma koeficienti. Apstrādes beigās `degree` vērtība tiek uzstādīta uz `coeff.size() - 1`, saglabājot invariantu.

3.5.5.4. `make_zero`

Nosaukums `Polynomial<F>::make_zero`

Ievade Metode ievadē saņem polinomu, kuram tā izsaukta.

Apstrāde Koeficientu saraksta `coeff` izmērs tiek uzstādīts uz `0`, bet `degree` uz -1.

3.5.5.5. `shift`

Nosaukums `Polynomial<F>::shift`

Ievade Metode ievadē saņem polinomu, kuram tā izsaukta, kā arī `int` tipa skaitli `n`, kas rāda ar kādu x^n polinoms tiks pareizināts.

Apstrāde Ja polinoma pakāpe `degree` ir negatīva vai arī `n` ir 0, tad turpmākā apstrāde tiek beigta, jo operācijas paredzamais rezultāts sakritīs ar esošo polinoma stāvokli. Ja `n` ir pozitīvs, tad polinoma koeficientu saraksta sākumā tiek ielikti `n` lauka `F` nulles elementi, atbilstoši palielinot `degree`. Ja `n` ir negatīvs, tad tiek izrēķināts cik polinoma koeficienti "pazudīs" reizinot ar x^n (šis skaitlis ne vienmēr ir `n`, t.i. gadījumos, kad `n` $>$ `degree`). Atbilstošais koeficientu skaits no koeficientu saraksta sākuma tiek izdzēsts un `degree` samazināts par izdzēsto elementu skaitu.

3.5.5.6. `operator==`, `operator!=`

Nosaukums `Polynomial<F>::operator==`, `Polynomial<F>::operator!=`

Ievade Funkcija ievadē saņem polinomu, kuram tā izsaukta, kā arī otru `Polynomial<F>` tipa polinomu `other`, ar kuru tiks veikta salīdzināšana.

Apstrāde Funkcija `operator==` tiek realizēta šādi: vispirms tiek salīdzinātas abu polinomu pakāpes, ja tās nesakrīt, tad funkcijas rezultāts ir `false`. Pretējā gadījumā funkcijas rezultāts ir koeficientu sarakstu salīdzināšanas rezultāts.

Funkcija `operator!=` tiek realizēta kā negācija no `operator==` rezultāta.

Izvade `bool` vērtība `true`, ja polinomi sakrīt, resp., nesakrīt, vai `false` pretējā gadījumā.

3.5.5.7. `operator+`, `operator-`

Nosaukums `Polynomial<F>::operator+`, `Polynomial<F>::operator-`

Ievade Funkcija ievadē saņem polinomu, kuram tā izsaukta, kā arī otru `Polynomial<F>` tipa polinomu `other`.

Apstrāde Tiek izveidots rezultāta polinoms `result` ar tādu pašu tipu kā ievadā saņemtie polinomi. Tālāk tiek iterēts pāri abu ievadā saņemto polinomu koeficientiem un rezultāta polinoma koeficientu sarakstam pievienots atbilstošo koeficientu summas, resp., starpības rezultāts. Ja kāda ievades polinoma koeficientu sarakstā ir beigušies koeficienti, tiek pieņemts, ka tālākie koeficienti ir lauka `F` nulles elementi. Funkcija savu darbu beidz, kad abu ievades polinomu koeficientu sarakstos ir beigušies elementi.

Izvade `Polynomial<F>` tipa polinoms, atbilstošās aritmētiskās operācijas rezultāts.

3.5.5.8. `operator*`

Nosaukums `Polynomial<F>::operator*`

Ievade Funkcija ievadē saņem pašu polinomu, kuram tā izsaukta, kā arī atkarībā, kurš no pārslogotajiem operatoriem tiek izsaukts:

- vai nu vienu lauka `F` elementu `elem`,
- vai arī `Polynomial<F>` tipa polinomu `other`

Apstrāde Ja tika pieprasīta reizināšana ar koeficientu no lauka `F`, tad rezultāta polinoms tiek rēķināts pareizinot visus dotā polinoma koeficientus ar `elem` un saglabājot dotā polinoma pakāpi.

Ja tika pieprasīta reizināšana ar polinomu, tad tiek veikta polinomu reizināšana atbilstoši parastajam polinomu reizināšanas algoritmam [2].

Rezultāts `Polynomial<F>` tipa polinoms, atbilstošās aritmētiskās operācijas rezultāts.

3.5.5.9. `operator/`

Nosaukums `Polynomial<F>::operator/`

Ievade Funkcija ievadē saņem pašu polinomu, kuram tā izsaukta, kā arī atkarībā, kurš no pārslogotajiem operatoriem tiek izsaukts:

- vai nu vienu lauka `F` elementu `elem`,
- vai arī `Polynomial<F>` tipa polinomu `other`

Apstrāde Ja tika pieprasīta dalīšana ar koeficientu no lauka `F`, tad vispirms funkcija pārliecinās vai `elem` nav lauka `F` nulles elements. Ja ir, tad tiek signalizēta izņēmuma situācija. Pretējā gadījumā rezultāta polinoms tiek izrēķināts, dalot visus dotā polinoma koeficientus ar `elem` un saglabājot dotā polinoma pakāpi.

Ja tika pieprasīta dalīšana ar polinomu, tad kā rezultāts tiek atgriezts atbilstošais rezultāts no `divmod` izsaukuma (skat. 3.5.5.10.).

Rezultāts `Polynomial<F>` tipa polinoms, atbilstošās aritmētiskās operācijas rezultāts.

3.5.5.10. `divmod`

Nosaukums `Polynomial<F>::divmod`

Ievade Funkcija ievadē saņem pašu polinomu, kuram tā izsaukta, kā arī `Polynomial<F>` tipa polinomu `other`.

Apstrāde Tiek realizēta polinomu dalīšana atbilstoši standarta polinomu dalīšanas algoritmam algoritmam, kas aprakstīts, piemēram, [2].

Rezultāts `std::pair<Polynomial<F>, Polynomial<F> >`, kura pirmais elements ir dalījums, bet otrais – dalīšanas atlikums.

3.5.5.11. `operator%`

Nosaukums `Polynomial<F>::operator%`

Ievade Funkcija ievadē saņem pašu polinomu, kuram tā izsaukta, kā arī `Polynomial<F>` tipa polinomu `other`.

Apstrāde Funkcija izsauc `divmod` abiem ievades polinomiem un kā rezultātu atgriež `divmod` izrēķināto dalīšanas atlikumu.

Rezultāts `Polynomial<F>` tipa polinoms – atbilstošais dalīšanas atlikums.

3.5.6. `IsIrreducible`

Nosaukums `IsIrreducible<P>`

Ievade Funkcija ievadē saņem `Polynomial<PrimeField<P> >` polinomu `f`, t.i. polinomu pār lauku $\mathbb{Z}/p\mathbb{Z}$.

Apstrāde Atbilstoši 4 teorēmai tiek izrēķināts LKD($f(x)$,$x^{p^i} - x$) visiem i, i, $1 \leqslant i \leqslant \frac{n}{2}$, kur n - f pakāpe `degree`. Ja kāds no šiem rezultātiem ir polinoms $P(x) = 1$, tad apstrāde tiek beigta un atdots rezultāts `false`. Ja neviens no iepriekš minētajiem rezultātiem nebija $P(x) = 1$, tad tiek atdots rezultāts `true`.

Izvade `bool` vērtība `true`, ja polinoms ir nereducējams, `false` pretējā gadījumā.

3.5.7. `GenerateRandomPolynomial`

Nosaukums `GenerateRandomPolynomial<int P>`

Ievade Funkcija ievadē saņem `unsigned int` tipa skaitli`degree` – ģenerējamā polinoma pakāpi, kā arī `bool` vērtību `generate_monic`, kas norāda vai tiks ģenerēts monisks polinoms.

Apstrāde Tiek izveidots `std::vector<PrimeField<P> >`, kurā tiek ielikti `degree` $- 1$ ar `std::rand()` palīdzību ģenerēti atlikumi mod$P$. Tālāk šis vektors tiek papildināts ar koeficientu 1 (ja `generate_monic` vērtība ir `true`) vai arī vēl vienu pseidogadījuma atlikumu.. Visbeidzot tiek izveidots `Polynomial<PrimeField<P> >` tipa polinoms, tam konstruktorā padodot tikko izveidoto vektoru.

Izvade `Polynomial<PrimeField<P> >` tipa polinoms ar pseidogadījumu koeficientiem vai arī šāda paša monisks koeficients ar pseidogadījumu koeficientiem (atkarībā no `generate_monic` vērtības).

3.5.8. `GenerateIrreduciblePolynomial`

Nosaukums `GenerateIrreduciblePolynomial<int P>`

Ievade Funkcija ievadē saņem `degere`, ģenerējamā nereducējamā polinoma pakāpi.

Apstrāde Izmantojot `GenerateRandomPolynomial<P>` tiek ģenerēti moniski polinomi-kandidāti ar pakāpi `degree`, līdz kamēr kādu no tiem funkcija `IsIrreducible<P>` atzīst par nereducējamu. Šādi atrasts polinoms arī ir funkcijas rezultāts.

Jāpiezīmē, ka lai gan funkcijai ir labs paredzamais izpildes laiks, tā ar mazu varbūtību var patērēt patvaļīgi ilgu darbības laiku. Tomēr varbūtība, ka tiks izpildītas vismaz `k` nereducējamības pārbaudes ir ne lielāka par $\frac{2}{\texttt{degree}}^{\texttt{k}}$, pie nosacījuma, ja $\texttt{degree} > 2$ [5]. Šī varbūtība eksponenciāli strauji sarūk pieaugot `k`, tāpēc paredzamais izpildes laiks ir $O(\texttt{degree}^4)$ [4]. Tāpat ir spēkā arī apsvērumi par `std::rand` izmantošanu, kas doti 3.6. apakšnodaļā.

Izvade Monisks `Polynomial<PrimeField<P> >` tipa polinoms, kas nav reducējams laukā $\mathbb{Z}/p\mathbb{Z}$.

3.5.9. `GaloisField`

Visiem `GaloisField<int P, int N>` elementiem ar vispārējās programmēšanas palīdzību tiek norādīti `P` un `N`, attiecīgi lauka $GF(p^n)$ parametri `p` un n.

Klasei `GaloisField` ir divi iekšējie mainīgie:

- `Polynomial<PrimeField<P> >` tipa polinoms `modulus`, kas ir lauku raksturojošais modulis no Kronekera konstrukcijas,
- `Polynomial<PrimeField<P> >` tipa polinoms `element_poly`, kas kopā ar `modulus` raksturo konkrēto lauka elementu.

Vēl klases iekšienē, koda skaidrības labad, tiek definēts tipa `Polynomial<PrimeField<P> >` aizstājvārds (*alias*) kā `PPoly`.

Visi `GaloisField` elementi `t`, ko izvadē atdod kāda `GaloisField` elementa `x` metodes, ir no šī paša lauka, t.i. `t` un `x` sakrīt gan `P`, gan `N`, gan arī lauku nosakošais polinoms `modulus`. Šī svarīgā detaļa īsuma labad netiks atrunāta katras metodes izvadē, bet pieņemta par skaidru.

Tāpat pirms katras lauka elementa operācijas ar kādu citu lauka elementu tiek izsaukta metode `check_modulus`, lai noskaidrotu vai šī operācija ir izdarāma, t.i. vai netiek mēģināts izdarīt operāciju ar elementiem no dažādiem laukiem $GF(p^n)$.

3.5.9.1. `GaloisField`

Nosaukums `GaloisField<P,N>::GaloisField`

Ievade Šim konstruktoram ir divas versijas ar atšķirīgu ievadi. Ievadē tiek saņemts:

- vai nu lauku raksturojošais polinoms `c_modulus`,

- vai arī lauku raksturojošais polinoms `c_modulus` un lauka elementu raksturojošais polinoms `c_element_poly`.

Apstrāde Tiek izveidots atbilstošais lauka elements, par tā `modulus` tiek uzstādīts `c_modulus`. Ja izsaukts konstruktors ar paremetru `c_element_poly`, tad par `element_poly` vērtību tiek uzstādīts `c_element_poly` pēc moduļa `c_modulus`. Pretējā gadījumā, ja `c_element_poly` nav norādīts, tad tiek konstruēts lauka $\mathrm{GF}(p^n)$ nulles elements, t.i., `element_poly` tiek uzstādīts kā nulles polinoms.

Ja tiek mēģināts izsaukt konstruktoru ar tādu `c_modulus`, kura pakāpe nesakrīt ar `N` tiek signalizēta izņēmuma situācija.

3.5.9.2. `check_moduli`

Nosaukums `GaloisField<P,N>::check_moduli`

Ievade Funkcija ievadē saņem elementu, kuram tā tika izsaukta, kā arī otru `GaloisField<P,N>` elementu `other`, ar kuru notiks moduļu pārbaude.

Apstrāde Tiek salīdzināti dotā elementa un `other` mainīgie `modulus`, izmantojot polinomu salīdzināšanu ar `operator==`. Ja tie sakrīt, tad turpmākā apstrāde tiek beigta, pretējā gadījumā tiek signalizēta izņēmuma situācija.

3.5.9.3. `operator==`, `operator!=`

Nosaukums `GaloisField<P,N>::operator==`, `GaloisField<P,N>::operator!=`

Ievade Funkcija ievadē saņem elementu, kuram operators tika izsaukts, kā arī otru `GaloisField<P,N>` elementu `other`, ar kuru notiks salīdzināšana.

Apstrāde Vienādības pārbaude notiek noskaidrojot vai dotā elementa un `other` mainīgie `element_poly` sakrīt pēc moduļa `modulus`. Nevienādības pārbaude tiek realizēta kā negācija no atbilstošās vienādības pārbaudes rezultāta.

Izvade `bool` vērtība `true`, ja elementi sakrīt, resp., nesakrīt, vai `false` pretējā gadījumā.

3.5.9.4. `inverse`

Nosaukums `GaloisField<P,N>::inverse`

Ievade Funkcija ievadē saņem elementu, kuram tā tiek izsaukta.

Apstrāde Ja `inverse` tiek mēģināts izsaukt elementam ar `element_poly` vienādu ar nulles polinomu (lauka nulles elementam), tad tiek signalizēta izņēmuma situācija.

Pretējā gadījumā izmantojot paplašināto Eiklīda algoritmu (funkcija `ExtendedEuclideanAlgorithm`) tiek izrēķināts dotajam elmentam apgrieztais elements dotajā laukā.

Izvade `GaloisField<P,N>` elements – dotā elementa multiplikatīvais inverss.

3.5.9.5. `operator+`, `operator-`, `operator*`, `operator%`

Nosaukums `GaloisField<P,N>::operator+`, `GaloisField<P,N>::operator-`, `GaloisField<P,N>::operator*`, `GaloisField<P,N>::operator%`

Ievade Funkcija ievadē saņem elementu, kuram operators tika izsaukts, kā arī otru `GaloisField<P,N>` elementu `other`, ar kuru notiks nepieciešamā aritmētiskā operācija

Apstrāde Saskaitīšana, atņemšana un reizināšana tiek realizēta izdarot minēto darbību atbilstoši Kronekera konstrukcijai, t.i., tiek veikta pieprasītā aritmētiskā darbība ar abu elementu `element_poly` un tās rezultāts ņemts pēc moduļa `modulus`. Rezultātā iegūtais elementu raksturojošais polinoms kopā ar `modulus` tiek padots `GaloisField<P,N>` konstruktoram, izveidojot rezultāta elementu.

Dalīšana tiek realizēta kā reizināšana ar `other.inverse()`, pārbaudot vai `other` nav lauka $\mathsf{GF}(p^n)$ nulles elements. Dalīšanas ar nulli gadījumā tiek signalizēta izņēmuma situācija.

Dalīšanas atlikums vienmēr ir nulle, jo jebkurā laukā jebkuru elementu var izdalīt ar jebkuru nenulles elementu. Izņēmums ir atlikums dalot ar nulli, par ko tiek signalizēta izņēmuma situācija.

Izvade `GaloisField<P,N>` elements – pieprasītās aritmētiskās operācijas rezultāts.

3.5.9.6. `is_square`

Nosaukums `GaloisField<P,N>::is_square`

Ievade Funkcija ievadē saņem elementu, kuram tā tika izsaukta.

Apstrāde Tiek šķiroti divi iespējami apstrādes scenāriji. Viens, kas lauka harasterikai esot 2 (t.i. `GaloisField<2,N>`) vienmēr atdod true, jo visiem lauka $\mathsf{GF}(2^n)$ elementiem eksistē kvadrātsakne. Otrs, kas laukiem ar nepāra harasteriku izmanto 8 teorēmu kvadrātiskuma noskaidrošanai.

Izvade `bool` vērtība `true`, ja elements dotajā laukā ir kvadrāts, un `false` pretējā gadījumā.

3.5.9.7. `sqrt`

Nosaukums `GaloisField<P,N>::sqrt`

Ievade Funkcija ievadē saņem elementu, kuram tā tika izsaukta.

Apstrāde Tiek šķiroti divi iespējami apstrādes scenāriji:

- viens, kas lauka harasterikai esot 2 (t.i. `GaloisField<2,N>`) izmantojot 7 teorēmu tieši izrēķina rezultātu,
- otrs, kas laukiem ar nepāra harasteriku izmanto Tonelli algoritmu [4] kvadrātsaknes izrēķināšanai. Tiek ģenerēts nejaušs dotā lauka elements, kamēr tiek atrasts tādu, kas nav kvadrāts. Izmantojot šo nekvadrātu tiek determinēti izrēķināta dotā elementa kvadrātsakne.

Izvade `GaloisField<P,N>` elements – dotā elementa kvadrātsakne dotajā laukā.

3.6. Projektējuma noteiktie ierobežojumi

Projektējums būtiski izmanto programmatūras prasību specifikācijā minēto faktu, ka lauka $\mathrm{GF}(\mathtt{p}^{\mathtt{n}})$ parametri $\mathtt{p}$ un $\mathtt{n}$ būs zināmi kompilācijas laikā.

Projektējumu bija iespējams izdarīt arī būtiski citādā veidā – ievērot, ka gan `GaloisField`, gan `PrimeField` varētu būt lauka `ModularField` apakšklases, kur `ModularField` realizētu kādu objektu lauku ar kuriem aritmētiskās darbības notiek pēc kāda cita objekta moduļa. `GaloisField` gadījumā šis modulis ir no klases `Polynomial<PrimeField<P> >`, bet `PrimeField` gadījumā šis modulis ir `PrimeField` parametrs `P`.

Šādai izvēlei par labu runā nelieli pirmkoda pārlietojamības (*source code reusability*) ieguvumi. Tomēr šādai izvēlei būtu bijuši arī dažādi trūkumi:

- `PrimeField` modulis `P` būtu jāglabā katras `PrimeField` instances iekšienē gluži kā `GaloisField` elementos tiek glabāta atsauce uz `modulus`. Šī ierobežojuma cēlonis ir C++ standarts, kas nepieļauj non-class template parametriem būt patvaļīgiem objektiem,
- augstāk minētais iemesls nākotnē varētu radīt ātrdarbības problēmas, jo `PrimeField` elementu moduļus vairs nevarētu salīdzināt kompilācijas laikā, bet tas būtu jādara programmatūras darbības laikā,

- tā kā `PrimeField` moduli nevar zināt kompilācijas laikā, tad nav iespējams arī rakstīt template specialization, lai iegūtu efektīvu `Polynomial<PrimeField<2> >` realizāciju. Ar šādu template specialization "mgflib" ātrdarbība plaši lietotajiem laukiem $\mathrm{GF}(2^n)$ varētu uzlaboties par vairākām kārtām, jo efektīvi tiktu izmantota darbību paralelizācija mašīnas vārdu līmenī (vairākas aritmētiskās operācijas viegli translējas par pabidu (*bitwise*) operācijām),

- šāds projektējums liktu domāt augstākā abstrakcijas līmenī, kas bibliotēkas lietotājiem bez labas galīgo lauku izpratnes varētu apgrūtināt tās modificēšanu.

Tāpēc arī projektējuma autors izvēlējās atbilstošos parametrus `P` un `N` nodot kompilācijas laikā un entītiju dekompozīciju veidot tieši šādu.

Projektējumā ir ierobežojums, ko uzdod `std::rand` lietojums. Šim pseidonejaušo skaitļu ģeneratoram ir mazs diapazons, kas neder nejaušu `PrimeField<P>` ģenerēšanai, ja `P` ir liels (neietilpst `unsigned short`). Tas nav pretrunā ar programmatūras prasību specifikāciju, tomēr var būt šķērslis nākotnē, lai gan apejams.

Tāpat arī `std::rand` dažās C++ standarta bibliotēkas implementācijās ir ļoti primitīvs un tāpēc tā ģenerētie pseidonejaušie skaitļi var likt funkcijai `GenerateIrreduciblePolynomial` savu darbu nebeigt, ja, piemēram, `std::rand` rezultāti ir cikliski ar mazu periodu.

Zema līmeņa algoritmu un datu struktūru projektējumā ir paredzēts `is_prime` realizēt ar Miller-Rabin testu, kas var būt neefektīvs (salīdzinot ar, piemēram, pilno pārlasi), ja pārbaudāmais skaitlis ir neliels. Tomēr šis tests ir izvēlēts ievērojot prasību specifikācijas aicinājumu nodrošināties lielāku skaitļu pārbaudēm nākotnē.

4. nodaļa

Testēšanas dokumentācija

Galuā lauku bibliotēkas "`mgflib`" testēšanas dokumentācija satur testēšanas specifikācijas, kā arī testu rezultātus, kādus deva testējamās bibliotēkas versija 1.0a (pašreizējā stabilā versija).

Vienībtestēšana notika no C++ programmas konstruējot minētos objektus un pārbaudot sagaidāmās atbildes ar aktuāli novērojamām. Ja kādā no testiem netiek saņemts norādītais rezultāts testēšana tika pārtraukta un noformēts īss kļūdu ziņojums, kas pievienots testēšanas dokumentācijai.

Jāievēro, ka testējamais objekts ir matemātiska bibliotēka un daudzus testpiemērus no cilvēka puses nav iespējams pārbaudīt grūti veicamo aprēķinu dēļ. Tāpēc par etalonrealizāciju tika izvēlēts lietot sistēmu Wolfram Mathematica 6.0 un par testu sagaidājiem rezultātiem tika uzskatīti tās dotie rezultāti. Tāpat daļa testu tika izstrādāta izmantojot publiski pieejamus testpiemērus, piem., nereducējamu polinomu sarakstus grāmatā [5]; arī šie testi nav viegli pārbaudāmi no cilvēka puses.

Par testēšanas prioritātēm. Tā kā autors vēlējās iegūt pilnīgu vienībtestēšanas aptvērumu, tad ierobežotā laika dēļ netika testēta piem. funkcija `GenerateRandomPolynomial`, jo tās rezultāts ir atkarīgs no funkcijas `std::rand()`. Autors uzskata, ka rakstīt `std::rand()` analogu ar paredzamu darbību un testēt to **un** `GenerateRandomPolynomial` neattaisno savu mērķi. Tā vietā tika izstrādāts īss funkcijas korektuma pierādījums (skat. pirmkodu) un nolemts uzskatīt šo funkciju par pareizi realizētu. Līdzīgi piem. `operator==` tika testēts mazāk nekā `sqrt`, saprotot šo divu funkciju atšķirīgās sarežģītības.

Tika veikta arī bibliotēkas savietojamības pārbaude ar dažādām arhitektūrām un dažādiem kompilatoriem atbilstoši programmatūras prasību specifikācijai.

Turpmākajās apakšnodaļās doti reprezentatīvākie vienībtestu piemēri. Jāpiebilst, ka visos šajos piemēros tiek testētas darbības ar matemātiskiem objektiem, tāpēc skaidrības labad to pieraksts ir dots augsta līmeņa simboliskā pierakstā nevis, piemēram, kā attiecīgo objektu aktuālās konstrukcijas no C++ koda. Testēšanas koda piemērs dots pielikumā.

4.1. Testu kopa `Polynomial` metodes `operator/` testēšanai

Šī testu kopa sastāv no vairākiem atsevišķiem `Polynomial<T>` metodes `operator/` testiem. Praktiski visus no tiem ir viegli pārbaudīt manuāli. Laukā "Sagaidāmais rezultāts" vērtība "Exception" nozīmē, ka šāda testa pareizais rezultāts ir izņēmuma situācijas signalizēšana. Visi testi ir autora sastādīti, neizmantojot palīglīdzekļus (piem. etalonrealizāciju).

T	**a**	**b**	**Sagaidāmais rezultāts**[1]
`int`	x^3	0	Exception
`PrimeField<3>`	0	0	Exception
`PrimeField<3>`	0	1	0
`int`	$x^3 + 3x^2 + 3x + 1$	$x + 1$	$x^2 + 2x + 1$
`int`	$x^3 + 3x^3 + 3x + 1$	$x^2 + 2x + 1$	$x + 1$
`PrimeField<11>`	x^3	x^4	0
`PrimeField<11>`	$x^3 + 1$	$x^3 + 10$	1
`PrimeField<2>`	$x^2 + 1$	$x + 1$	$x + 1$
`PrimeField<7>`	$x^9 + 3x^3 + 2$	1	$x^9 + 3x^3 + 2$
`int`	$2x^4 - 5x^3 + 10x + 11$	$x^2 - 3$	$2x^2 - 5x + 6$
`int`	$x^9 - 1$	$x^3 + 1$	$x^6 - x^3 + 1$

4.1. tabula: Testu kopa `Polynomial` metodes `operator/` testēšanai

4.2. Testu kopa funkcijas `IsIrreducible` testēšanai

Šī testu kopa sastāv no vairākiem atsevišķiem testiem, kas pārbauda funkcijas `IsIrreducible` korektu darbību. Testējot funkciju tiek sagatavots polinoms, konstruējot to no koeficientu saraksta, tālāk tas padots funkcijai `IsIrreducible` un salīdzināts rezultāts.

Vairums no piemēriem pār lauku $\mathbb{Z}/2\mathbb{Z}$ ir sagatavoti izmantojot nereducējamu polinomu sarakstus grāmatā [5]; visi pārējie ir paša autora atrasti, gan manuāli, gan izmantojot etalonrealizāciju.

Koeficientu lauks	**Polinoms**	**Sagaidāmais rezultāts**
$\mathbb{Z}/2\mathbb{Z}$	$x^{303} + x + 1$	`true`
$\mathbb{Z}/2\mathbb{Z}$	$x^{310} + x^{93} + 1$	`true`
$\mathbb{Z}/2\mathbb{Z}$	$x^{310} + x^{92} + 1$	`false`
$\mathbb{Z}/2\mathbb{Z}$	x	`true`
$\mathbb{Z}/2\mathbb{Z}$	$x^2 + x + 1$	`true`
$\mathbb{Z}/7\mathbb{Z}$	$x^9 + 3x^3 + 1$	`false`
$\mathbb{Z}/7\mathbb{Z}$	$x^9 + 4x^2 + 1$	`true`
$\mathbb{Z}/19\mathbb{Z}$	$x^{29} + 5x^9 + 16x^9 + 1$	`true`
$\mathbb{Z}/19\mathbb{Z}$	$x^{29} + 5x^9 + 16x^9 + 1$	`true`
$\mathbb{Z}/19\mathbb{Z}$	$x^{29} + 5x^9 + 16x^9 + 5$	`true`
$\mathbb{Z}/19\mathbb{Z}$	$x^{29} + 5x^9 + 16x^9 + 2$	`false`
$\mathbb{Z}/19\mathbb{Z}$	$x^{29} + 5x^9 + 16x^9 + 3$	`false`
$\mathbb{Z}/11\mathbb{Z}$	x^{13}	`false`

4.2. tabula: Testu kopa funkcijas `IsIrreducible` testēšanai

4.3. Testu kopa `GaloisField` metodes `sqrt` testēšanai

Šīs testu kopas testi pārbauda `GaloisField<P,N>::sqrt` darbību. Testēšana notiek konstruējot attiecīgo lauka elementu, tam izsaucot metodi `sqrt()` un salīdzinot rezultātā iegūto lauka elementu ar sagaidāmo.

Testa piemēri tika izvēlēti šādi: visi iespējamie testa piemēri pār lauku $(\mathbb{Z}/2\mathbb{Z})/(x^2 + x + 1)$, visi iespējamie testa piemēri pār lauku $(\mathbb{Z}/5\mathbb{Z})/(x)$. Tālāk vairāki testa piemēri pār lielākiem laukiem, kuri radīti ar etalonrealizāciju.

Šai testu kopai ir būtiska iezīme – pareizā atbilde var nebūt unikāla. Līdzīgi kā reālo skaitļu laukā skaitlim 1 ir divas kvadrātsaknes (1 un -1), tāpat arī laukos ar harasteriku lielāku par 2, ja x ir elementa u kvadrātsakne, tad tāda ir arī $-x$ un $x \neq -x$. Šī iezīme jāņem vērā pārbaudot `sqrt` atbildes.

Lauks	**Elementa polinoms**	**Kvadrātsaknes polinoms vai 0, ja elements nav kvadrāts**
$(\mathbb{Z}/2\mathbb{Z})/(x^2+x+1)$	0	0
$(\mathbb{Z}/2\mathbb{Z})/(x^2+x+1)$	1	1
$(\mathbb{Z}/2\mathbb{Z})/(x^2+x+1)$	x	$x+1$
$(\mathbb{Z}/2\mathbb{Z})/(x^2+x+1)$	$x+1$	x
$(\mathbb{Z}/5\mathbb{Z})/(x)$	0	0
$(\mathbb{Z}/5\mathbb{Z})/(x)$	1	1
$(\mathbb{Z}/5\mathbb{Z})/(x)$	2	0
$(\mathbb{Z}/5\mathbb{Z})/(x)$	3	0
$(\mathbb{Z}/5\mathbb{Z})/(x)$	4	2
$(\mathbb{Z}/13\mathbb{Z})/(x)$	10	6
$(\mathbb{Z}/3\mathbb{Z})/(x^3+2x^2+1)$	x^2+2x+2	x^2
$(\mathbb{Z}/3\mathbb{Z})/(x^3+2x^2+1)$	x^2+1	0
$(\mathbb{Z}/7\mathbb{Z})/(x^9+2)$	$5x^3+3$	$4x^6+6x^3+x^6$
$(\mathbb{Z}/7\mathbb{Z})/(x^9+2)$	$3x^3+1$	0

4.3. tabula: Testu kopa `GaloisField` metodes `sqrt` testēšanai

Visi minētie testpiemēri pēc vieglas modifikācijas ir derīgi arī metodes `is_square` testēšanai.

4.4. Testēšanas žurnāls

Bibliotēkas "mgflib" versijas 1.0pre-a testēšana tika veikta izmantojot operētājsistēmu Windows XP Service Pack 3 (32-bit), g++ kompilatoru (versija g++ (TDM-1 mingw32) 4.4.0).

Atbilstošie testēšanas žurnāla fragmenti parādīti zemāk.

Testējamā vienuma identificējums	**Izpildes rezultāts**	**Reģistrēto problēmu ziņojumu identificējumi**
...		
`Polynomial<F>::operator/`	Tests veiksmīgs	—
`IsIrreducible<F>`	Tests veiksmīgs	—
`GaloisField<N,P>::sqrt`	Tests neveiksmīgs	Problēma sqrt1
...		

4.4. tabula: Testēšanas žurnāla fragments

4.5. Problēmu ziņojumi

4.5.1. sqrt1

Problēmas ziņojuma identificējums sqrt1

Problēmas īss apraksts Laukā $(\mathbb{Z}/13\mathbb{Z})/(x)$ jeb $(\mathbb{Z}/13\mathbb{Z})$ elementam 10 kvadrātsaknes ir 6 un 7. Tomēr metodes `sqrt` rezultāts šim elementam ir 4, tātad nepareizs, jo kāpinot šo atbildi kvadrātā iegūstam nepatiesu vienādību: $4^2 \equiv 10 \pmod{13}$.

Problēmas cēlonis Problēmas cēloni izdevās identificēt kā nepareizu salīdzināšanu metodes `sqrt` iekšienē (salīdzināšanas operatora `!=` vietā kodā tiek izsaukts operators `==`), kas izraisīja cikla invarianta pārkāpšanu.

Labotāja ziņojums Problēma tika reproducēta. Pēc tās cēloņa noskaidrošanas tika izdarītas korekcijas `sqrt` realizācijā un problēma novērsta.

Atkārtotā testa rezultāts Tests izpildīts veiksmīgi. Problēmas statuss mainīts uz slēgtu.

4.6. Bibliotēkas savietojamības testi

Bibliotēkas savietojamības testēšana notika vienībtestēšanā izmantotos testpiemērus darbinot uz dažādām sistēmām. Tika pārbaudīts vai bibliotēka bez kļūdu paziņojumiem kompilējas izmantojot gan noklusētos, gan striktus kompilatora uzstādījumus (piem. `-Wall -Wextra` kompilatora `g++` gadījumā).

Kā iespējams secināt no zemāk redzamās tabulas, bibliotēka apmierina tai uzstādītās apartūras ierobežojumu un nefunkcionālās prasības:

Sistēma	**Kompilators**	**Rezultāts**
Windows XP Service Pack 3 (32-bit)	g++ (TDM-1 mingw32) 4.4.0	Tests iziets
Windows XP Service Pack 2 (64-bit)	Microsoft (R) C/C++ Optimizing Compiler Version 14.00.40310.41 for AMD64	Tests iziets
OpenBSD 4.4 (32-bit)	g++ (GCC) 4.2	Tests iziets

4.5. tabula: Savietojamības testu rezultāti

5. nodaļa

Programmatūras pirmkoda fragmenti

Viss programmatūras pirmkods ir par lielu, lai to iekļautu šajā darbā, tāpēc iekļauti tiek autoraprāt reprezentatīvi tā fragmenti. Zemāk iespējams apskatīt,

- funkciju `ExtendedEuclideanAlgorithm`, kopā ar to pavadošo `divmod` template specialization, kas optimizē darbību dažiem datu tipiem,
- klasi `GaloisField`,
- funkciju `GenerateIrreduciblePolynomial` nereducējama polinoma ģenerēšanai,
- bibliotēkas lietošanas piemērs kvadrātsaknes vilkšanai dotā laukā

Klasēm realizācija bija nošķirta no definīcijas, šeit tā tiek rādīta līdzās, neizdalot atsevišķu apakšnodaļu.

5.1. Funkcija `ExtendedEuclideanAlgorithm`

```
//
// mgflib by Madars Virza (c) 2009
//
// euclidean.hpp --- generic programming implementation of the
// Euclidean algorithm.
//
// Licensed under MIT license.
//

#ifndef MGFLIB_EUCLIDEAN_HPP_
#define MGFLIB_EUCLIDEAN_HPP_

#include "divmod.hpp"
#include <utility>

namespace mgflib {

    // Generic programming implementation of the Euclidean algorithm.
    template<typename T>
    std::pair<T,T> ExtendedEuclideanAlgorithm(const T &u, const T &v, const T& zero, const T& one)
    {
        // Invariant: u * t1 + v * t2 = t3,
        //            u * u1 + v * u2 = u3,
```

```
        //            u * v1 + v * v2 = v3
        T u1(one), u2(zero), u3(u);
        T v1(zero), v2(one), v3(v);
        T t1, t2, t3;
        while (v3 != zero)
        {
            const std::pair<T,T> dm_rez(divmod_wrap(u3, v3));
            t1 = u1 - v1 * dm_rez.first;
            t2 = u2 - v2 * dm_rez.first;
            t3 = dm_rez.second;

            u1 = v1;
            u2 = v2;
            u3 = v3;

            v1 = t1;
            v2 = t2;
            v3 = t3;
        }

        return std::make_pair(u1, u2);
    }
}
#endif
```

5.2. `divmod` noteikšana, izmantojot template specialization

```
#ifndef MGFLIB_DIVMOD_HPP_
#define MGFLIB_DIVMOD_HPP_
namespace mgflib {
    template<bool b>
    struct divmod_selector
    {
        template<typename T>
        static std::pair<T,T> divmod(T const& one, T const& other)
        {
            return std::pair<T,T>(one/other, one%other);
        }
    };

    template<>
    struct divmod_selector<true>
    {
        template<typename T>
        static std::pair<T,T> divmod(T const& one, T const& other)
        {
            return one.divmod(other);
        }
    };

    template<typename T>
    struct supports_optimised_divmod {
        static const bool value = false;
    };

    template<typename T>
    std::pair<T,T> divmod_wrap(T const& one, T const& other) {
        return divmod_selector<supports_optimised_divmod<T>::value>::divmod(one, other);
    }
}
#endif
```

5.3. Funkcija `GenerateIrreduciblePolynomial`

```
//
// mgflib by Madars Virza (c) 2009
//
// polyutils.hpp -- various generic programming routines, including
// one which tests irreducibility of polynomials with coefficients
// from PrimeField<p>, one that generates irreducible polynomial of
// given degree and one that generates a random polynomial of given
// degree.
//
// Licensed under MIT license.
```

```
//

#ifndef MGFLIB_POLYUTILS_HPP_
#define MGFLIB_POLYUTILS_HPP_

#include <utility>
#include <iostream>
#include <cstdlib>

#include "primefield.hpp"
#include "polynomial.hpp"
#include "pow.hpp"
#include "euclidean.hpp"

namespace mgflib
{
    template<int P>
    bool IsIrreducible(const Polynomial<PrimeField<P> > &f)
    {
        typedef Polynomial<PrimeField<P> > PPoly;
        // cache often used values (linear_term, zero, one) to avoid
        // recomputation
        const PPoly zero(0);
        const PPoly one(1);

        PPoly u(1);
        u.shift(1); // u(x) is now 1*x^1 + 0*x^0

        const PPoly linear_term(u);

        for (int i = 1; i <= f.degree/2 ; ++i)
        {
            u = powmod<PPoly>(u, P, f);
            const PPoly d(EuclideanAlgorithm<PPoly>(f, u - linear_term,
                                                    zero, one));

            if (d != one)
            {
                return false; // we can show a factor
            }
        }

        return true; // all posible factors must be exhausted by now,
                     // so the polynomial must be irreducible
    }

    // Generates a random monic polynomial with specified degree
    template<int P>
    Polynomial<PrimeField<P> > GenerateRandomPolynomial(const int degree,
                                                        bool generate_monic)
    {
        typedef Polynomial<PrimeField<P> > PPoly;
        PPoly p;

        // prepare a vector of random coefficients
        std::vector<PrimeField<P> > vec;
        for (int i = 0; i < degree; ++i) // invariant: after each step
                                         // vec.size() is i and each
                                         // element is random
        {
            vec.push_back(PrimeField<P>(std::rand() % P));
        }
        if (generate_monic)
        {
            vec.push_back(PrimeField<P>(1)); // leading monic coefficient
        }
        else
        {
            vec.push_back(PrimeField<P>(std::rand() % P));
        }

        // after these steps it is easy to show correctness of the
        // function. there are degree+1 elements and therefore the
        // degree of polynomial is the desired degree

        return PPoly(vec); // create a random polynomial from the random
                           // vector
    }


    // This function generates an irreducible polynomial modulo P. It
    // has been shown that irreducible polynomials of degree n are
```

```
    // almost 1/2n of all polynomials, so we generate a random
    // polynomial and hope for the best. Without loss of generality we
    // can generate monic polynomials and so we do.
    template<int P>
    Polynomial<PrimeField<P> > GenerateIrreduciblePolynomial(const int degree)
    {
        Polynomial<PrimeField<P> > p;
        do
        {
            p = GenerateRandomPolynomial<P>(degree, true);
        }
        while (!IsIrreducible<P>(p)); // while random polynomial is
                                      // not irreducible - repeat
        return p;
    }
}
#endif
```

5.4. Klase GaloisField

```
//
// mgflib by Madars Virza (c) 2009
//
// galoisfield.hpp/galoisfield.cc -- generic implementation of Galois
// fields given by the Kroenecker construction.
//
// Licensed under MIT license.
//

#ifndef MGFLIB_GALOISFIELD_HPP_
#define MGFLIB_GALOISFIELD_HPP_

#include "polynomial.hpp"
#include "primefield.hpp"

namespace mgflib
{
    // Generic programming implementation of finite field GF(p^n),
    // where p and n are specified by generic programming.
    template <int P, int N>
    class GaloisField
    {
      public:
        // We define a type alias for clarity, because
        // Polynomial<PrimeField is used quite often in this code
        typedef Polynomial<PrimeField<P> > PPoly;

        // Common modulus of all elements of the field
        PPoly modulus;

        // Unique identifier of this element
        PPoly element_poly;

      public:
        // Constructor for GaloisField element 0.
        GaloisField(const PPoly &c_modulus);

        // Constructor for GaloisField element which corresponds to
        // polynomial c_element_poly
        GaloisField(const PPoly &c_modulus, const PPoly &c_element_poly);

        // This auxilarily function checks if moduli of this and other
        // GaloisField element are equal. This is needed, because we
        // can't pass Modulus as template parameter; may be changed
        // with respect to further C++ standard development. If moduli
        // differ we throw an exception
        void check_moduli(const GaloisField<P,N>& other) const;

        // Returns if two GF(p^n) elements are equal.
        bool operator==(const GaloisField<P,N>& other) const;

        // Returns if two GF(p^n) elements differ.
        bool operator!=(const GaloisField<P,N>& other) const;

        // Returns sum of self and other GF(p^n) element.
        GaloisField<P,N> operator+(const GaloisField<P,N>& other) const;

        // Returns difference self minus other GF(p^n) element.
        GaloisField<P,N> operator-(const GaloisField<P,N>& other) const;
```

```
    // Returns GF(p^n) element which is multiple of self and
    // other GF(p^n) element.
    GaloisField<P,N> operator*(const GaloisField<P,N>& other) const;

    // Element count in this field
    unsigned int element_count() const;

    // Returns if current GF(p^n) element is a square, i.e. whether
    // there exists such x that x^2 = a.
    bool is_square() const;

    // Returns square root of current GF(p^n) element. If the
    // element is not a square returns zero element.
    GaloisField<P,N> sqrt() const;

    // Returns inverse element for current GF(p^n) element.
    GaloisField<P,N> inverse() const;

    // Returns quotient from division of self by other GF(p^n)
    // element. It is the same as calling: operator*(other.inverse())
    GaloisField<P,N> operator/(const GaloisField<P,N>& other) const;

    // Returns remainder from division of self by other prime
    // field element. It is obviously always zero. (Of course, an
    // exception may occur when getting, for example, 1%0).
    GaloisField<P,N> operator%(const GaloisField<P,N>& other) const;
};

// Constructor for GaloisField element 0.
template <int P, int N>
GaloisField<P,N>::GaloisField(const PPoly &c_modulus) : modulus(c_modulus), element_poly(0)
{
    if (modulus.degree != N)
    {
        throw std::runtime_error("GaloisField<P,N>::GaloisField/1 -- constructing "
                                 "field element with invalid degree modulus");
    }
}

// Constructor for GaloisField element which corresponds to
// polynomial c_element_poly
template <int P, int N>
GaloisField<P,N>::GaloisField(const PPoly &c_modulus, const PPoly &c_element_poly) : modulus(c_m
{
    if (modulus.degree != N)
    {
        throw std::runtime_error("GaloisField<P,N>::GaloisField/2 -- constructing "
                                 "field element with invalid degree modulus");
    }
    // normalise c_element_poly and store it in element_poly
    element_poly = c_element_poly % modulus;
}

// This auxilarily function checks if moduli of this and other
// GaloisField element are equal. This is needed, because we
// can't pass Modulus as template parameter; may be changed
// with respect to further C++ standard development. If moduli
// differ we throw an exception
template <int P, int N>
void GaloisField<P,N>::check_moduli(const GaloisField<P,N>& other) const
{
    if (modulus != other.modulus)
    {
        throw std::runtime_error("GaloisField<P,N> -- attemped "
                                 "operation with mixed moduli elements");
    }
}


// Returns if two GF(p^n) elements are equal.
template <int P, int N>
bool GaloisField<P,N>::operator==(const GaloisField<P,N>& other) const
{
    check_moduli(other);
    return ((element_poly - other.element_poly) % modulus).is_zero();
}

// Returns if two GF(p^n) elements differ.
template <int P, int N>
bool GaloisField<P,N>::operator!=(const GaloisField<P,N>& other) const
```

```
{
    check_moduli(other);
    return !(operator==(other));
}

// Returns sum of self and other GF(p^n) element.
template <int P, int N>
GaloisField<P,N> GaloisField<P,N>::operator+(const GaloisField<P,N>& other) const
{
    check_moduli(other);
    PPoly result_ep = (element_poly + other.element_poly);
    return GaloisField<P,N>(modulus, result_ep);
}

// Returns difference self minus other GF(p^n) element.
template <int P, int N>
GaloisField<P,N> GaloisField<P,N>::operator-(const GaloisField<P,N>& other) const
{
    check_moduli(other);
    PPoly result_ep = (element_poly - other.element_poly);
    return GaloisField<P,N>(modulus, result_ep);
}

// Returns GF(p^n) element which is multiple of self and other
// GF(p^n) element.
template <int P, int N>
GaloisField<P,N> GaloisField<P,N>::operator*(const GaloisField<P,N>& other) const
{
    check_moduli(other);
    PPoly result_ep = (element_poly * other.element_poly);
    return GaloisField<P,N>(modulus, result_ep);
}

// Returns inverse element for current GF(p^n) element.
template <int P, int N>
GaloisField<P,N> GaloisField<P,N>::inverse() const
{
    if (!element_poly.is_zero())
    {
        // To obtain the inverse we are solving a*x + b*modulus = 1
        // for a, b. Then a is our inverse
        const std::pair<PPoly,PPoly> eeres =
                ExtendedEuclideanAlgorithm<PPoly>(element_poly, modulus,
                                                  PPoly(0), PPoly(1));
        const PPoly eer = eeres.first * element_poly + eeres.second * modulus;
        return GaloisField<P,N>(modulus, eeres.first / *(eer.coeff.rbegin()));
    }
    else
    {
        throw std::runtime_error("GaloisField<P,N>::inverse — element zero has no inverse");
    }
}

// Element count in this field
template<int P, int N>
unsigned int GaloisField<P,N>::element_count() const
{
    return pow<unsigned int>(P, N);
}

// Returns if current GF(p^n) element is a square, i.e. whether
// there exists such x that x^2 = a.
template <int P, int N>
bool GaloisField<P,N>::is_square() const
{
    if (P == 2)
    {
        // All elements of GF(2^n) are squares (see below)
        return true;
    }
    else
    {
        // We use the following result: u is a square iff
        // u^((|F|-1)/2) = 1, where F = GF(p^n) with odd p.

        const int power((element_count() - 1) / 2);

        const PPoly res(powmod<PPoly>(element_poly, power, modulus));
        return (res == PPoly(1));
    }
```

```
}

// Returns square root of current GF(p^n) element. If the element
// is not a square returns zero element.

// We use template specialization to distinguish between fields
// with characteristic 2 and fields with odd charasteristic.
template <int P, int N>
GaloisField<P,N> GaloisField<P,N>::sqrt() const
{
    if (P == 2)
    {
        // If field characteristic is 2 we already know the answer,
        // because by Fermat's Little Theorem a^(2^n) == a. Therefore
        // if we set x to a^(2^(n-1)), then x^2 = a as needed.
        PPoly new_ep(element_poly);

        // Loop invariant: after each loop iteration new_ep is
        // a^(2^i), so when i has reached N-1 we have our answer.
        for (int i = 1; i < N; ++i)
        {
            new_ep = new_ep * new_ep;
        }

        return GaloisField<P,N>(modulus, new_ep);
    }
    else
    {
        // We know that P is odd (see case for P=2 below), so we may
        // use Shank-Tonelli's algorithm for computing the square
        // root.
        typedef GaloisField<P,N> GF;
        // const GF one(modulus, PPoly(1));
        const PPoly one(1);
        const int q_minus(element_count() - 1);

        int t = q_minus;
        int s = 0;

        while (t % 2 == 0)
        {
            ++s;
            t /= 2;
        }
        // Now we have q_minus = 2**s * t (simple invariant)

        // Find a non-square element of the same field.
        GF g(modulus);

        do
        {
            g = GF(modulus, GenerateRandomPolynomial<P>(N-1));
        }
        while (g.is_square());

        // const GF ginv(g.inverse());
        PPoly ginv(g.inverse().element_poly);

        int e = 0;
        int two_i = 2;
        for (int i = 2; i <= s; ++i)
        {
            two_i *= 2; // two_i <- 2**i
            const PPoly aginve((element_poly * powmod<PPoly>(ginv, e, modulus)) % modulus);
            const PPoly order_val(powmod<PPoly>(aginve, q_minus/two_i, modulus));

            if (order_val != one)
            {
                e += (two_i / 2);
            }
        }

        const PPoly h((element_poly * powmod<PPoly>(ginv, e, modulus)) % modulus);
        const PPoly rez(powmod<PPoly>(g.element_poly, e/2, modulus) * powmod<PPoly>(h, (t+1)/2, r
        return GF(modulus, rez);
    }
}

// Returns quotient from division of self by other GF(p^n)
// element. It is the same as calling: operator*(other.inverse())
template <int P, int N>
```

```
    GaloisField<P,N> GaloisField<P,N>::operator/(const GaloisField<P,N>& other) const
    {
        check_moduli(other);
        return (operator*(other.inverse()));
    }

    // Returns remainder from division of self by other GF(p^n)
    // element. It is obviously always zero. (Of course, an exception
    // may occur when getting, for example, 1*x^0 % 0 * x^0).
    template <int P, int N>
    GaloisField<P,N> GaloisField<P,N>::operator%(const GaloisField<P,N>& other) const
    {
        check_moduli(other);
        if (!other.element_poly.is_zero())
        {
            return GaloisField<P,N>(modulus, PPoly(0));
        }
        else
        {
            throw std::runtime_error("GaloisField<P,N>::operator% — division by zero");
        }
    }
}

#endif
```

5.5. Bibliotēkas lietošanas piemērs kvadrātsaknes vilkšanai

```
#include <cstdio>
#include <utility>
#include <vector>
#include <cstdlib>
#include <ctime>

#include "primefield.hpp"
#include "polynomial.hpp"
#include "polyutils.hpp"
#include "galoisfield.hpp"
using namespace mgflib;
using namespace std;

typedef Polynomial<PrimeField<2> > Poly2;

typedef vector<int> VI;
#define PB push_back

int main(void) {
    srand(time(NULL)); // randomize the timer

    VI v; v.PB(2);
    v.PB(0); v.PB(0); v.PB(0);
    v.PB(0); v.PB(0); v.PB(0);
    v.PB(0); v.PB(0); v.PB(1);
    // prepare modulus — Z/7Z(x^9+2)

    VI vv; vv.PB(3); vv.PB(0); vv.PB(0); vv.PB(5);
    // prepare element 5x^3 + 3

    GaloisField<7,9> x(v, vv); // actually generate the element
    printf("isskv = %d\n", x.is_square());
    GaloisField<7,9> root(x.sqrt()); // take square root

    printf("%d\n", root.degree); // list coefficients
    for (unsigned int i=0; i < root.coeff.size(); i++)
        printf("%d ", root.coeff[i].residue_class);
    printf("\n");
}
```

6. nodaļa

Projekta organizācija

Galuā lauku bibliotēka "mgflib" tikai izstrādāta par pamatu ņemot spirālveida modeli.

Vispirms tika veikta teorētiskā materiāla izpēte un galveno prasību izvirzīšana. Tad tika veikta projektēšana tās rezultātā izveidojot entītiju dekompozīciju. Pēc tam notika entītiju secīga detalizēta projektēšana un implementēšana, atbilstoši izdarot precizējumus programmatūras prasību specifikācijā. Visbeidzot tika veikta programmatūras testēšana un produkta dokumentācijas galīgās versijas izstrāde. Vienībtestēšanas laikā tika atklātas implementācijas kļūdas, kuras uzreiz tika labotas.

Nepieciešamo teorētisko materiālu iegūšanai un pamatprasību definēšanai autoram palīdzēja darba vadītājs. Turpmākā projekta izstrāde tika veikta patstāvīgi, tajā piedaloties vienam cilvēkam (darba autoram), kas veica projektētāja, programmētāja, dokumentācijas izstrādātāja un testētāja funkcijas.

Projekta izstrāde notika izmantojot g++ kompilatoru, GNU emacs teksta redaktoru. Koda un dokumentācijas sekmīga konfigurācija pārvaldība tika nodrošināta izmantojot Mercurial. Kvalifikācijas darbs un projekta pavadošā dokumentācija tika veidota izmantojot LaTeX. Darba noformējumam tika izmantots stila fails `NoslegumaDarbs.cls`[1] ar nelielām darba autora modifikācijām.

[1] Pieejams: http://susurs.mii.lu.lv/sergejs/sagataves.exe

7. nodaļa

Konfigurāciju pārvaldība

Gan programmatūras produkta, gan dokumentācijas konfigurāciju pārvaldībai tika izmantota Mercurial versiju pārvaldības sistēma. Šīs sistēmas izvēli noteica autora iepriekšējā pieredze ar to; tikpat labi uzdevumam būtu derējusi praktiski jebkura cita sistēma.

Projekta kods kopā ar konfigurāciju repozitoriju tika glabāts uz izstrādes datora cietā diska, kā arī katras izstrādes dienas beigās tā saturs tika sinhronizēts ar kopiju uz attālināta servera, izmantojot Mercurial iekļauto funkcionalitāti.

Veicot revīziju uzskaiti par katru tika saglabāts neliels komentārs par veiktajām izmaiņām; iekšējās atsauces uz revīzijām tika noformētas, izmantojot attiecīgās Mercurial revīzijas numuru. Tāpat tika ievērots, ka katrai versijai, kas tiek saglabāta, ir jābūt strādājošai. Šo pasākumu ieviešana nodrošināja ērtāku kļūdu atrašanas un labošanas procesu.

8. nodaļa

Kvalitātes nodrošināšana

Lai nodrošinātu kvalitatīva produkta izstrādi tika veiktas šādas darbības:

- projektu pavadošā dokumentācija tika izstrādāta vadoties pēc atbilstošajiem Latvijas Valsts standartiem;
- lai nodrošinātu koda noformējuma konsistenci tika izmantots Google C++ Style Guide [8], kas nosaka vienotus nosacījumus koda noformējumam (mainīgo nosaukumu izvēle, atkāpes, maksimālie rindiņu garumi, utml.) un semantikai (piem. tādas valodas C++ iespējas kā reference mainīgo marķēšana ar const, lai novērstu to nejaušu izmainīšanu);
- bibliotēkas kods tika pilnīgā komentēts angļu valodā, ieskaitot metožu nolūkus un saskarnes aprakstus. Sarežģītākajām metodēm tika veidoti komentāri par to iekšējo darbību;
- sarežģītākajām metodēm tika izstrādāti un komentēti arī cikla invarianti. Šīs darbības, kas ir daļa no programmatūras korektuma pierādīšanas, ievērojami atvieglo testēšanu – liela daļa kļūdu tika atklātas jau izstrādes fāzē;
- gandrīz katrai produkta daļai tikai veikta arī vienībtestēšana, lai pārliecinātos par tās pareizu darbību.

9. nodaļa

Darbietilpības novērtējums

Viens no populārākajiem modeļiem darbietilpības novērtēšanai ir COCOMO modelis. Lai ar to iegūtu darbietilpības novērtējumu ir nepieciešams prognozēt aptuveno koda rindiņu skaitu, ko savukārt var prognozēt izmantojot funkcijpunktu modeli.

Moduļa `GaloisField` projektējumā var apvienot, piemēram, metožu `operator+`, `operator-` un `operator*` saskarnes, jo tās pēc loģikas un uzbūves ir ļoti līdzīgas, tāpēc nopietns darbs notiek uzrakstot pirmo no tām, bet pārējās ir no tās atvasinātas. Līdzīgi spriežot iegūst, ka `GaloisField` satur, pēc COCOMO metrikas, 2 vienkāršus ievadus, 5 vienkāršus izvadus 2 vienkāršus iekšējo datu failus. Vēl sistēmā no ārpuses tiek pieprasīts atklāt 3 funkcijas ar vienkāršu saskarni un `Polynomial` konstruktoru. Tas kopā dod 10 vienkāršus ievadus/izvadus saskarnes un 2 vienkāršus datu failus, kas atbilst $13 \cdot 3 + 2 \cdot 7 = 53$ nepielāgotiem (unadjusted) funkcijpunktiem. Zinot, ka viena funkcijpunkta realizācijai valodā C++ ir nepieciešamas aptuveni 53 rindiņas iegūstam novērtējamo pirmkoda rindiņu skaitu kā 2809.

Šāds novērtējums ir nedabisks – efektīvu bibliotēku var realizēt daudz mazākā rindu skaitā; to varam skaidrot ar COCOMO metodes piemērotību informācijas sistēmu projektiem. Toties, ja par ievadi uzskatam ne tikai iespēju kāda operatora parametros padot lietotāja konstruētu lauka elementu, bet gan tikai paša elementa konstruēšanu, tad iegūstam ievērojami mazāku funkcijpunktu skaitu, 20, kas atbilstu 1060 koda rindiņām, kas jau ir tuvāk patiesajam. Tomēr šāds spriedums būtu kļūdains – tad katras jaunas GaloisField metodes pievienošana (piem. tāda, kas rēķinātu diskrēto logaritmu, kas ir ļoti tehniski sarežģīts algoritms) neko nedotu rindiņu skaitam. Šajā izstrādes laika novērtējumā netiek ņemts vērā fakts, ka, lai gan izstrādātais koda apjoms nav liels, tomēr izstrādājamais projekts ir algoritmiski sarežģīts.

Jārīkojas citādi. Ņemot vērā prognozēto funkciju sarežģītības (no vienkārša `operator==` līdz kvadrātsaknes vilkšanai Galuā laukā), varam prognozēt, ka vidēji viena funkcija aizņems 20 izpildāmās rindiņas. Kopējais produkta izmērs tātad tiktu prognozēts kā 780 rindiņas liels. Izmantojot COCOMO darbietilpības aprēķinus "organic" projektā

ar sarežģītību, kas lielāka par vidējo iegūstam, ka projekta darbietilpību varētu prognozēt kā 3.2 personmēnešus liela.

Autors veica nelielu aptauju savu LU Matemātikas un informātikas institūta kolēģu vidū, lai noskaidrotu faktisko darbietilpību līdzīgas specifikas projektos (zinātnisks uzdevums; izstrādes valoda C++; neliels projekta izmērs; izstrādāts gan kods, gan visa pavadošā dokumentācija, kods iztestēts). Tika noskaidrots, ka starp koda izmēru l (mērīts tūkstošos izpildāma koda rindiņu) un izstrādes laiku personmēnešos t pastāv vidēji spēcīga korelācija: $t = 1.746 \cdot 2.764^l$ ($R^2 = 0.537$).

Izmantojot šādu metriku izstrādātā projekta darbietilpība tiktu novērtēta kā aptuveni 3.9 mēnešus liela. Jāteic gan, ka šis novērtējums nav atzīstams par zinātnisku aptaujas mazā izmēra dēļ (6 projekti).

Reāli projekta realizācija un dokumentācijas izveide notika aptuveni 4 personmēnešu laikā; vislielākais laiks tika pavadīts projekta projektēšanas fāzē. Projekts sastāv no ~ 850 izpildāmā koda rindiņām (netiek skaitītas tukšās rindas un komentāri).

10. nodaļa

Rezultāti

Darba izstrādes gaitā ir izveidota pilnīgi funkcionējoša galīgo lauku bibliotēka, kas atbilst izstrādātajai prasību specifikācijai. Tika veikta visas izstrādes dokumentēšana un izstrādājamā produkta vienībtestēšana. Visa izstrāde notika atbilstoši valsts un starptautiskajiem standartiem un labajai programmēšanas praksei.

Darba izpildes gaitā autors iepazinās ar projekta izstrādes dzīves ciklu posmiem un apguva jaunas iemaņas programmēšanas valodā C++.

Autoru īpaši priecē fakts, ka izveidotā bibliotēka piedāvā darboties ar praksē reti realizētu, tomēr pētniecībā nepieciešamu, galīgo lauku $\mathsf{GF}(\mathfrak{p}^{\mathfrak{n}})$ veidu, kurā $\mathfrak{p}$ un $\mathfrak{n}$ abi var būt patvaļīgi skaitļi; vairumā autora atrasto bibliotēku tika uzstādīts kāds no ierobežojumiem $\mathfrak{p} = 2$ vai $\mathfrak{n} = 1$. Tāpat daļa šādu bibliotēku nepietiekami pārlietoja (reused) kodu, kas, cerams, nav gadījums ar "mgflib".

Šī dokumenta rakstīšanas brīdī projekts ir pabeigts un tiek sagatavots izplatīšanai kā atklātā pirmkoda produkts.

11. nodaļa

Secinājumi

Pēc darba paveikšanas autors secināja, ka veiksmīgāka projekta izstrāde būtu notikusi izmantojot ūdenskrituma modeli; tas būtu palīdzējis atklāt vairāk projektējuma nepilnību jau dzīves cikla sākumā. Tāpat autors uzskata, ka šis darbs būtu ieguvis no neatkarīgas testēšanas un citiem paņēmieniem, kas strādājot vienatnē nav pieejami, piemēram, formālām tehniskajām apskatēm. Tomēr izstrādātā bibliotēka pierādā, ka rezultātus var sasniegt arī viena cilvēka komandā ar spirālveida modeli.

Kā interesanta problēma darba projektēšanas un realizācijas gaitā atklājās jautājums, kurus Galuā lauka parametrus labāk nodot kā template parametrus un kurus – caur klases konstruktoru; plašāka diskusija par to ir šī darba 3.6. apakšnodaļā. Cerams, ka nākamajās bibliotēkas versijās gan $\mathfrak{p}$, gan $\mathfrak{n}$ tips varēs būt gari skaitļi, tādējādi bibliotēku padarot praktiskāku lietojumiem kriptogrāfijā.

Nākotnē būtu ērti vienībtestu veikšanai izmantot kādu testēšanas ietvaru (framework), kā arī izstrādāt publicējamu funkcionālās testēšanas rīku.

Pateicības

Autors izsaka pateicību darba vadītājam Sergejam Kozlovičam par interesantas tēmas ieteikšanu, nozīmīgiem ieteikumiem darba rakstīšanas procesā un viņa ieguldījumu projekta attīstības sekmēšanā. Pateicība pienākas arī Andrejam Vihrovam un Dāvim Brēdikam par padomiem darbā ar C++ un iepazīstināšanu ar template specialisation.

Paldies arī maniem draugiem un ģimenei, kas saprotoši izturējās pret manu aizņemtību darba rakstīšanas gaitā.

Izmantotā literatūra un avoti

[1] **Ноден, П., Китте, К.** *Алгебраическия алгоритмика (с упражнениями и решениями).* М.: Мир, 1999. 720 с.

[2] **Knuth D.** *The Art of Computer Programming, vol. 2. Seminumerical Algorithms (3rd ed.).* Reading, Massachusetts: Addison-Wesley, 1998. 784 pp.

[3] **Ireland, M., Rosen, K.** *A Classical Introduction to Modern Number Theory (2nd ed.).* Springer, 1998. 389 pp.

[4] **Bach, E., Shallit, J.** *Algorithmic Number Theory, Vol. 1: Efficient Algorithms.* Cambridge, Massachussetts: The MIT Press, 1996. 496 pp.

[5] **Hankerson, D., Menezes, A. J., Vanstone, S.** *Guide to Elliptic Curve Cryptography.* Springer, 2004. 311 pp.

[6] **Vandevoorde, D., Josuttis, N. M.** *C++ Templates: The Complete Guide.* Addison Wesley, 2002. 552 pp.

[7] **Pretzel, O.** *Error-Correcting Codes and Finite Fields.* Oxford: Clarendon Press, 1992. 398 pp.

[8] *Google C++ Style Guide* [tiešsaiste]. [atsauce 15.02.2009]. Pieejams: `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`.

Kvalifikācijas darbs

"Galuā lauku realizācija, izmantojot vispārējās programmēšanas paradigmu"

Ar savu parakstu apliecinu, ka darbs veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti, un iesniegtā darba elektroniskā kopija atbilst izdrukai. Piekrītu sava darba publicēšanai internetā.

Autors: Madars Virza .....

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto kvalifikācijas darbu un atzīstu to par p i e m ē r o t u / n e p i e m ē r o t u (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes studiju programmas "Programmeēšana un datortīklu administrēšana" kvalifikācijas pārbaudījuma komisijas sēdē.

Darba vadītājs: Mg. dat. Sergejs Kozlovičs .....

Darbs iesniegts Datorikas fakultātē .....

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Metodiķe: .....

Recenzents: Filips Jeļisejevs

Darbs aizstāvēts kvalifikācijas pārbaudījuma komisijas sēdē ..... prot. Nr. ....., vērtējums ..... (.....).

(.....)

Komisijas sekretārs: ....