CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

HasLin - ett DSL för linjär algebra

Utvecklandet av ett matematiskt domänspecifikt språk för linjär algebra i Haskell

Kandidatarbete inom Data- och informationsteknik

Adam Eliasson
Daniel Nikoalev
Filip Nordmark
Sebastian Sjögren
Linus Sundkvist

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2022

Kandidatarbete 2022

HasLin - ett DSL för linjär algebra

Utvecklandet av ett matematiskt domänspecifikt språk för linjär algebra i Haskell

Adam Eliasson
Daniel Nikoalev
Filip Nordmark
Sebastian Sjögren
Linus Sundkvist

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
Göteborgs universitet
Göteborg, Sverige 2022

HasLin - ett DSL för linjär algebra
Utvecklandet av ett matematiskt domänspecifikt språk för
linjär algebra i Haskell
Adam Eliasson Daniel Nikoalev Filip Nordmark Sebastian Sjögren Linus Sundkvist

Handledare: Patrik Jansson, Institutionen för data- och informationsteknik
Examinator: Andreas Abel, Institutionen för data- och informationsteknik

Kandidatprojekt 2022
Institutionen för data- och informationsteknik
Chalmers tekniska högskola och Göteborgs universitet
SE-412 96 Göteborg
Telefon: +46 31 772 1000

Göteborg, Sverige 2022

Abstract

Mathematics is an important part of computer science and linear algebra is a common subject at university-level engineering programmes. Interpreting mathematics within the context of a domain-specific language can bridge the gap between mathematics and computer science. The goal of the project was thus to create a domain-specific language for linear algebra, named *HasLin*. The purpose was to investigate how the correctness of HasLin can be proven and how the domain can be pedagogically communicated to strengthen future users' knowledge of the domain and computer science. HasLin is embedded in the functional programming language Haskell. To demonstrate the correctness of HasLin, tests were constructed in Haskell as well as proofs in the proof assistant Agda. The results show that HasLin supports a large set of basic operations for linear algebra, however correctness is not shown to the extent that initially was intended. HasLin is adapted to be easy-to-use and is published via a browser-based user interface, which allows use of HasLin without the need for installed software. Some further development is needed, mainly in testing and verification, to prove the correctness of HasLin to better serve the purpose of the project.

Keywords: Haskell, DSL, Linear algebra, Agda, pedagogy, proof.

Sammandrag

Matematik är en viktig del av datavetenskap och ett vanligt förekommande ämne på teknikinriktade program på universitet är linjär algebra. Om matematik tolkas inom kontexten av ett domänspecifikt språk kan klyftan mellan matematik och datavetenskap överbryggas. Målet med projektet var därmed att skapa ett domänspecifikt språk för linjär algebra, benämnt *HasLin*. Arbetet var menat till att undersöka hur HasLins korrekthet kan bevisas samt hur domänen kan pedagogiskt förmedlas för att stärka framtida användares kunskaper inom domänen och datavetenskap. HasLin är inbäddat i det funktionella programmeringsspråket Haskell. För att visa HasLins korrekthet konstruerades tester i Haskell samt bevis i bevisassistenten Agda. Resultatet visar att HasLin stödjer en stor mängd grundläggande operationer för linjär algebra, däremot visas inte korrekthet i den mån syftet ämnade. HasLin är anpassat för att vara lätt-använt och är publicerat via ett webbbaserat användargränssnitt vilket tillåter användning av programmet utan ett behov av installerad mjukvara. Viss vidareutveckling behövs, främst inom test och verifikation, för att bevisa HasLins korrekthet för att bättre tjäna projektets syfte.

Nyckelord: Haskell, DSL, Linjär algebra, Agda, pedagogik, bevis.

Förord

Vi vill säga ett stort tack till vår handledare Patrik Jansson, som har stöttat och väglett oss i rätt riktning igenom hela projektets gång. Vi vill även tacka de frivilliga testare som deltog i andvändartestet.

Innehåll

Figurer

Tabeller

1

Inledning

Matematik är en viktig del av datavetenskap samtidigt som matematikens beroende av mjukvara ökar. Detta tyder på att en gemensam förståelse för datavetenskap och matematik skulle gynna båda parterna. Trots detta existerar en klyfta mellan dem.

Tentamenstatistiken i kursen Linjär algebra på Chalmers tekniska högskola visar att genomsnittliga antalet godkända studenter som går det datatekniska programmet är runt 70% [Fys22]. Det kan anses vara en låg andel godkända i jämförelse med programmet industriell ekonomi, ett annat program på Chalmers, där i genomsnitt 90% av studenterna blir godkända. Detta visar på en avsaknad av matematisk förståelse hos datatekniker.

Det väcker frågan om det är möjligt att skapa ett domänspecifikt språk som kan överbrygga klyftan mellan datavetenskap och andra matematiska discipliner. Det skulle potentiellt innebära att matematiken blir mer påtaglig för datavetare och programmerare i allmänhet, samtidigt som det ger matematiker bättre möjligheter att använda datavetenskap.

1.1 Bakgrund

Inom utvecklingen av ett *DSL*, *domänspecifikt språk*, eller *Domain-specific language* på engelska , fokuseras det på att modellera en enda domän medan generella programmeringsspråk kan tillämpas i flera olika domäner. Fördelen med att specialisera inom en domän är att syntax och semantik kan göras mer lämpad för att lösa specifika problem. Generellt sätt är DSL mindre komplexa och de kan variera i sin omfattning beroende på vilken domän de modellerar. Ett välkänt domänspecifikt språk är HTML (*HyperText Markup Language* [htm22]) som används för att ordna strukturen av element i webbapplikationer.

Om matematik tolkas inom kontexten av DSL kan klyftan mellan datavetenskap och andra matematiska discipliner potentiellt överbryggas. Det datavetenskapliga perspektivet ger en informativ synvinkel på matematik och i detta avseende kan utbildningen av matematik generellt ta nytta av tillvägagångsättet från datavetenskap [JIB22a]. Genom att implementera en mjukvarubaserad abstraktion av den matematiska syntaxen så kan en bättre insikt ges kring den tvetydiga semantiken inom den klassiska matematiken.

Det finns många ämnen inom matematiken som har användningsområden inom datavetenskap [Sed16, Wes21]. Linjär algebra spelar bland annat en fundamental roll i datorgrafik,

där matris-operationer används vid rendering av 3D- och 2D-miljöer. Dessutom används linjär algebra inom maskinlärning i form av, bland annat, linjär regression [Met20]. På grund av bredden av användningsområdena i linjär algebra för datavetenskap, ska detta projekt fokusera på att kommunicera det matematiska språket kring linjär algebra i form av ett DSL. Detta med syftet att demonstrera hur matematik från datavetenskapens perspektiv kan underlätta förståelsen av matematiska koncept för datavetare.

Projektet baseras framför allt på den valbara kursen *Domain-Specific Languages of Mathematics*, förkortas *DSLsofMath*, [JIB22b] som undervisas för datavetare och matematiker på Chalmers tekniska högskola och Göteborgs universitet. Kursens syfte är att presentera matematik utifrån ett datavetenskapligt perspektiv genom funktionell programmering. Mycket av metodiken som finns i den kursen har inspirerat detta projekt. Det innebär att ett DSL implementeras och utvecklas utifrån en matematisk domän baserat på ett funktionellt programmeringsspråk. Mer specifikt den matematiska grenen linjär algebra i funktionella programmeringsspråken Haskell och Agda.

1.2 Relaterade Arbeten

Program som hanterar linjär algebra är ingen ny uppfinning. Det finns många existerande program i flera olika programmeringsspråk som tillåter detta. Skillnaden mellan dessa program och detta projekt är inte effektivitet eller användbarhet, utan strukturen. Ett funktionellt programmeringsspråk som Haskell ger en matematisk korrekthet väl lämpad för bevisföring och forskning, medan andra tillämpningar av linjär algebra är skapade för att ge högsta effektiviteten möjlig för datorspel och liknande, där precision inte vägs lika tungt.

Att skapa en domänspecifik implementering av linjär algebra inom Haskell och Agda är inget unikt nyskapande arbete heller. Liknande implementeringar existerar redan, de mest välkända av dem finns som importerbara bibliotek inom språken. I Haskell är exempelvis biblioteket *Linear* [Kme21] ett bibliotek för typer och kombinationer för linjär algebra på fria vektorrum. Biblioteket har över 100 000 nedladdningar och har funnits i över 80 olika versioner. I biblioteket omfattas mycket liknande logik för matris- och vektoroperationer och egenskaper hos vektorrum som görs i detta arbete. Ett annat välanvänt Haskell-bibliotek är *hmatrix : Numeric.LinearAlgebra* [Rui21]. Baserat på BLAS (*Basic Linear Algebra Subprograms*) [Net21] och LAPACK (*Linear Algebra Package*) [Net22] innehåller biblioteket rutiner för matris- och vektoroperationer samt även för annan numerisk linjär algebra som lösningar av linjära ekvationssystem och egenvärdesproblem.

I Agdas standard bibliotek, *Agda-stdlib*, finns det underbibliotek för datatyper och operationer för vektorer och bevisande av vissa egenskaper [Dev22]. Agda-arbetet i detta projekt efterliknar mycket av vad som hittas i *agda-stdlib/src/Data/Vec/* med huvudfokus på bevisandet av vektorers egenskaper.

De ovan nämnda domänspecifika implementeringarna av linjär algebra omfattar majoriteten av den matematik som modelleras i detta arbete, det visar att det redan finns ett intresse och användningsområde för att modellera domänen ur ett datavetenskapligt perspektiv. Detta arbete ämnar inte till att vara nyskapande eller hävdar till att det är en unik implementation. Arbetet är menat till att undersöka hur domänens korrekthet

kan bevisas och pedagogiskt förmedlas för att stärka både utvecklarnas samt användarnas kunskaper inom domänen och datavetenskap.

1.3 Syfte

Projektets syfte är att skapa ett domänspecifikt språk för linjär algebra. Språket ska utvecklas med fokus på domänens lagar så att korrekthet i största mån kan garanteras. För att uppnå detta kommer projektet därför att specificera och bevisa egenskaper hos uttryck i språket. Dessa specifikationer och bevis kan ses som ett komplement till den huvudsakliga produkten: det domänspecifika språket, benämnt HasLin. Förhoppningen är att språket ska kunna användas till att lösa konkreta problem, öka användarens förståelse av linjär algebra och som ett hjälpmedel för bevis av matematiska satser.

1.4 Avgränsningar

Projektet avgränsas till grundläggande principer inom linjär algebra, vilket inkluderar: vektorer, matriser, egenvärden, linjära rum, linjära avbildningar och linjära ekvationssystem. Böckerna *Linear Algebra Done Right* av Sheldon Axler [Axl95] samt *Linjär algebra* av Gunnar Sparr [Spa97] studeras och därmed avgränsas projektet till matematik som de tar upp. Det betyder inte att allt material i böckerna representeras, utan böckerna används som riktlinjer för vad som kan inkluderas och i vilken ordning det domänspecifika språket utvecklas.

HasLin hanterar godtyckligt stora matriser, men inte oändliga matriser och vektorer. Det här görs eftersom hantering av oändliga matriser och vektorer är för komplext och tidskrävande att implementera. Linjär algebra är ett brett ämne där mycket kan implementeras och därför borde det undvikas att överkomplicera vissa delar av det för att möjliggöra en bredare implementation.

1.5 Projektmål

Projektmålet är att utveckla ett domänspecifikt språk som kan ge en pedagogisk insyn i den matematiska grenen linjär algebra från ett datavetenskapligt perspektiv. Utvecklingen ska ske i de funktionella programspråken Haskell [Has22b] och Agda [The22]. Uppbyggnaden av det domänspecifika språket syftar till att öka förståelsen inom linjär algebra för framtida studenter eller andra intresserade genom att ge mer material att arbeta med. Det genomförs genom specifikation av introducerade begrepp, lagar och uppmärksammande av syntax och typer.

Processen att utveckla domänspecifika språk för en domän är en lång process bestående av flertal olika moment. Uppbyggnaden av projektet delas därav upp i följande olika projektmål. Genomförande av samtliga projektmål är menat att resultera i en slutprodukt vilket är ett komplement till linjär algebra ur ett datavetenskapligt perspektiv. Följande moment har identifierats:

Analys av linjära algebrans grund: För en holistisk bild över vad som behöver inkluderas inom DSL-implementeringen görs en litteraturgenomgång av linjär algebra. Det inkluderar studerande av vektorer, matriser, linjära rum, linjära avbildningar, linjära ekvationssystem, egenvärden och egenvektorer.

Matematiken i funktionell programmering: Baserat på linjära algebra analysen kodas matematiken i funktionell programmering. Utveckling av en miljö för att möjliggöra exekvering av generella matematiska operationer samt specialiserade operationer för grenen linjär algebra. Det inkluderar framför allt uppbyggnad av nödvändiga matematiska typer, specifikationer och lagar.

Det finns väsentliga skillnader mellan matematik och dess tolkning i mjukvara. Datorer har en exekverings- och platskostnad som begränsar användning av vissa matematiska objekt. Till exempel kan inte reella tal implementeras exakt utan måste approximeras – vilket motsäger deras matematiska definition. Projektet kommer därför att behöva göra vissa begränsningar så att språket ska vara användbart. Dessa avvikelser behöver göras med avvägning att den: semantiskt sätt passar domänen; inte kompromissar med korrekthet; och har en rimlig komplexitet.

Testning och bevisande av korrekthet: Testande och bevisande av den matematiska korrektheten i det utvecklade domänspecifika språket. Detta innebär att den utvecklade miljön behöver inkludera moduler med testfall som visar att språket följer de lagar som specificeras inom linjär algebra.

Skapandet av en interaktiv upplevelse: Utveckling av ett kompletterande material, där framtida studenter eller andra intresserade själva kan utföra tester och bevis med de framtagna modulerna för att fördjupa sin egen förståelse.

2

Teori

Detta kapitel klargör den teoretiska bakgrunden för projektet. Teoridelen är avgränsad till att framföra en generell överblick kring programmerings- och matematik-koncept som faller inom ramen för det domänspecifika språket.

2.1 Externa och inbäddade DSL

Det finns två olika typer av DSL, externa och inbäddade [Fow19]. Externa DSLs är implementerade via en oberoende tolk eller kompilator, exempelvis är LaTeX [LaT22] som används för att skriva denna rapport ett externt DSL för typsättning. Med LaTeX domänspecifika syntax och text som indata struktureras olika typer av dokumentstilar. Inbäddade DSL [Fow19], förkortat eDSLs (*embedded domain specific languages*), är DSL som typiskt sett är implementerade som ett bibliotek inom ett generellt programmeringsspråk. På motsvarande nivå är det samma typ av DSL som utvecklas i detta projekt. Det domänspecifika språket tenderar därav att bli begränsad till syntaxen av värdspråket.

Inom eDSLs kan det ytterligare specificeras vad för sort av inbäddning som implementerats, nämligen djup eller ytlig inbäddning. I en djup inbäddning implementeras ett abstrakt syntax-träd för att beskriva termer i DSL [GW14]. Till detta behövs det så kallade *programtolk* vars syfte är att tolka abstrakta syntax-träd för att beräkna värdet av termer. Jämfört med ytlig inbäddning implementeras termerna direkt som värdet av den termen. Detta innebär att det inte behövs någon tolkningsfunktion, då denna skulle vara ekvivalent till identitetsfunktionen.

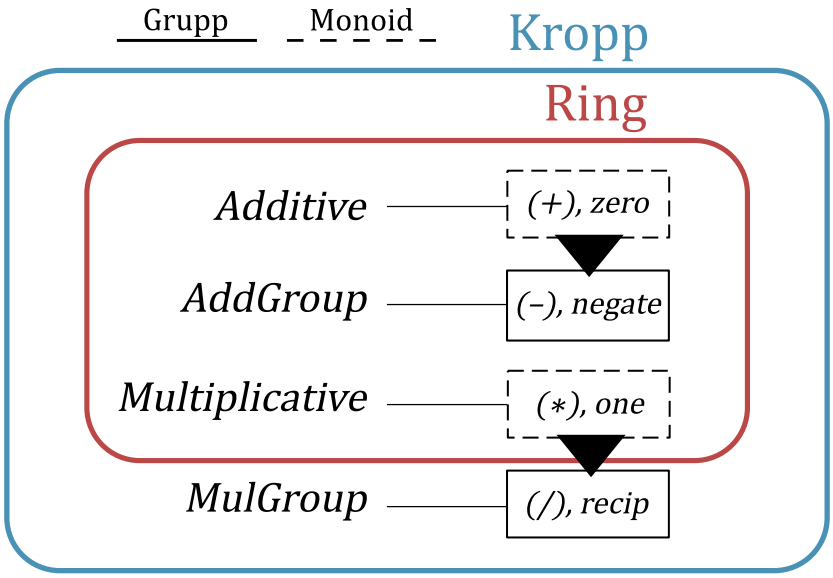
2.2 Abstrakt algebra

Inom matematiken finns det ett område som behandlar så kallade algebraiska strukturer: detta område kallas abstrakt algebra [Pra21]. I HasLin introduceras ett par algebraiska strukturer som är relevanta för att beskriva domänen för linjär algebra. Bland annat tillämpas ringar, grupper och kroppar vilket beskrivs i sektionerna nedan. I verkligheten finns det många fler strukturer men för syftet av arbetet framhävs endast det som är implementerat i detta DSL.

Ett intuitivt sätt att se på algebraiska strukturer är att de härleds i en hierarkisk ordning där olika strukturer uppfyller diverse axiom. Ringar uppfyller alla axiom som grupper har och utöver det följer de lagarna för kommutativitet. Exempelfiguren 2.1, inspirerad

från kursboken DSLsofMath [JIB22a], illustrerar hierarkin mellan strukturerna. Samma modell för algebraiska strukturer som på figuren är implementerat i det domänspecifika språket för linjär algebra.

Anledningen till att dessa matematiska begrepp introduceras i HasLin är för att de formaliserar matematiska lagar kring den domän som specificerar objekten. Ytterligare är det intressant att studera homomorfier [Pra21], där två algebraiska strukturer av samma typ har strukturen bevarad vid avbildningen av en funktion som kallas en homomorfi. Det finns olika sorter av homomorfier beroende på typen av strukturen exempelvis: gruppomomorfier, ringhomomorfier eller homomorfier mellan vektorrum, som kallas linjära avbildningar.



Figur 2.1: Denna bild visar sambanden mellan strukturerna och deras hierarki. Pilarna indikerar att grupperna ärver operationer från en monoid såsom Additive eller Multiplicative.

2.2.1 Monoid

En monoid presenteras matematiskt som en trippel som består av en underliggande mängd av värden, en associativ binär operator mellan par av element i mängden, samt ett enhetselement [Pra21].

Låt trippeln $(A, e, \bullet)$ vara en godtycklig monoid.

A : Den underliggande mängden, exempelvis de naturliga $\mathbb{N}$ talen eller heltalen $\mathbb{Z}$

e : enhetselementet $e \in A$

$\bullet$: Binär operation. $A \times A \to A$

Följande tre axiom måste upprätthållas av en monoid.

Slutenhet : $\forall a, b \in A.\ a \bullet b \in A$

Associativitet : $\forall a, b, c \in A.\ (a \bullet b) \bullet c = a \bullet (b \bullet c)$

Identitet : $\forall a \in A.\ e \bullet a = a \bullet e = a$

En homomorfi mellan två monoider är en avbildning mellan dess underliggande mängder där strukturen bevaras. Låt h vara en funktion för en avbildning mellan monoiderna (M, e_M, $\bullet$) och (N, e_N, $\times$). $h : M \to N$ är en homomorfi om:

$$\text{Avbildning av identitet} : h(e_M) = e_N$$
$$\text{Avbildning av operationer} : \forall x, y \in M.\ h(x \bullet y) = h(x) \times h(y)$$

2.2.2 Grupp

Grupper bygger på strukturen av monoider. En grupp är en monoid där varje element har en invers [Pra21]. Grupper följer samma axiom som monoider: slutenhet, associativitet och identitet (se 2.2.1), därutöver ska den även uppfylla inversen.

Låt trippeln $(G, e, \bullet)$, vara en godtycklig grupp.

Invers : $\forall a \in G.\ \exists b \in G.\ a \bullet b = b \bullet a = e$

Uttrycket syftar på att varje element a har en invers a^{-1} och att en binär operation mellan ett element och dess invers i en grupp, resulterar i gruppens identitetselement. Inversen går att uttrycka som en unär operation.

Som ett exempel tar vi avbildningen i AddGroup 2.1.

$$negate : G \to G$$
$$\forall a \in G.\ negate(a) = -a$$

Homomorfi definieras på samma sätt mellan grupper som för monoider fast med gruppers strukturer där inversen också bevaras vid avbildning.

$$\forall a \in G.\ h(a^{-1}) = h(a)^{-1}$$

En grupphomomorfi som är relevant i vårt DSL är determinanten, se teori 2.3.7. Determinanten är en avbildning mellan två multiplikativa strukturer. Den kan definieras som en avbildning $det : R^{n \times n} \to R$. Eftersom determinanten är en homomorfi, bevaras strukturen mellan grupperna $(R^{n \times n}, I, \times)$ och $(R, e, *)$ vid avbildningen.

$$\forall A, B \in R^{n \times n}.\ det(A \times B) = det(A) * det(B)$$

Notera ovanför att i vänster led utförs en binär operation för multiplikation mellan matriser och i höger led utförs multiplikation mellan reella tal.

2.2.3 Ring

Ring är en algebraisk struktur som beskrivs i figur 2.1. Det vill säga en mängd med två binära operationer med egenskaper av addition och multiplikation [Pra21]. Båda operationerna är slutna operationer i mängden vilket innebär att addera eller multiplicera element från mängden ger ett element från samma mängd.

$$\textbf{Slutenhet: } \forall a, b \in R. a + b \in R \ \ \& \ \ a * b \in R$$

Utöver den slutna egenskapen krävs följande för operationerna i en ring:

Addition	**Multiplikation**
$a + 0 = 0 + a = a$ (Identitet)	$a * 1 = 1 * a = a$ (Identitet)[1]
$a + (b + c) = (a + b) + c$ (Associativitet)	$a * (b * c) = (a * b) * c$ (Associativitet)
$a + (-a) = (-a) + a = 0$ (Invers)	
$a + b = b + a$ (Kommutativ/Abelsk)	

En ring är alltså en kommutativ grupp över addition som också har multiplikation. Additionen och multiplikationen länkas sedan genom den distributiva lagen:

$$\textbf{Distributiva lagen: } \begin{aligned} a * (b + c) &= a * b + a * c \\ (a + b) * c &= (a * c) + (b * c) \end{aligned}$$

2.2.4 Kropp

Det finns olika typer namn på ringar baserat på hur nära det är en grupp under multiplikation. En ring som är kommutativ under multiplikation samt har en multiplikativ invers för alla element skilda från noll, vilket är division, är vad som kallas en kropp [Pra21].

$$\begin{aligned} \text{Kommutativ/Abelsk multiplikation: } & a * b = b * a \\ \text{Multiplikativ invers: } & a \neq 0 \Rightarrow a * (a^{-1}) = (a^{-1}) * a = 1 \end{aligned}$$

En kropp är med andra ord också en fundamental algebraisk struktur likt en ring fast med något fler krav av multiplikativa egenskaper [Ada07].

2.3 Linjär Algebra

Denna sektion bygger vidare på koncepten kring algebraiska strukturer och redogör grundläggande teori för linjär algebra. Innan den praktiska delen behandlas kommer en kortfattad sammanställning kring teorin i linjär algebra som omfattas av detta DSL.

2.3.1 Vektorer och vektorrum

Vektorer beskriver storlek (skalär) och riktning [Spa97]. Förutom hastigheten av ett föremål som beskrivs av en skalär, så kan en vektor också presentera riktningen av det

[1] Inte ett krav men oftast inkluderat

föremålet. Vanligtvis för ändliga vektorrum, se teori 2.3.3, är representationen av vektorer i form av en lista med koordinater vars värden är ändpunkterna som tillsammans bildar riktningen och storleken av vektorn.

$$v = [a_1, a_2, ..., a_n]$$

Låt V vara en mängd av vektorer. V sägs vara ett vektorrum över en kropp F om det är slutet under operationerna för addition och skalärmultiplikation [Nee07]. Med andra ord kan vektorer adderas och skalas med en koefficient till att bilda en vektor i samma vektorrum.

2.3.2 Delrum

Om en delmängd U till ett vektorrum V också är ett vektorrum, med samma addition och skalär multiplikation, kallas U för ett delrum av V [Axl95]. Detta uppfylls om följande gäller för U:

$$\begin{aligned} \text{Additiv identitet :} &\quad 0 \in U \\ \text{Sluten under addition :} &\quad v, w \in U \implies v + w \in U \\ \text{Sluten under skalär multiplikation :} &\quad s \in F, v \in U \implies sv \in U \end{aligned}$$

där V är ett vektorrum över F.

Nära relaterat till delrum är *kvotrum* [Axl95]. Ett kvotrum Q är ett förflyttat delrum och kan skrivas på formen $Q = v + U$ där U är ett delrum till V och $v \in V$.

2.3.3 Ändliga vektorrum och baser

Ett vektorrum är av ändlig dimension om alla vektorer kan skrivas som en linjär kombination från en ändlig delmängd av rummet [Axl95].

Konkret är V ett ändligt vektorrum om det finns en lista $l = v_1, v_2, ..., v_n \in V$ så att $V = span(l)$. Där spannet av en lista vektorer definieras av

$$span(v_1, v_2, ...v_n) = \{a_1v_1 + a_2v_2 + ... + a_nv_n \mid a_i \in F\}$$

där V är ett vektorrum över F.

Om en lista $v_1, v_2, ..., v_n$ spänner upp rummet V och är linjärt oberoende, så att ingen vektor kan skrivas som en linjär kombination av de andra vektorerna, utgör listan en *bas* för V. Alltså är en bas en lista med vektorer så att hela rummet kan representeras unikt som en linjärkombination av listans vektorer. Längden på en sådan lista är lika med dimensionen av vektorrumet.

2.3.4 Linjär avbildning

En linjär avbildning, även kallad linjär transform, är en funktion mellan två vektorrum som bevarar den linjära strukturen [Axl95]. Den linjära strukturen bevaras om $f : V \to W$, där både V och W är vektorrum över F, uppfyller följande egenskaper:

$$\begin{aligned} \text{Additiv}: &\quad f(u+v) = fu + fv \\ \text{Homogen}: &\quad f(sv) = s(fv) \end{aligned}$$

där $u, v \in V$ och $s \in F$.

Mängden av linjära funktioner mellan två givna vektorrum utgör också ett vektorrum. Addition och skalär multiplikation är definierat enligt:

$$\begin{aligned} \text{Addition}: &\quad (f+g)v = fv + gv \\ \text{Skalär multiplikation}: &\quad (sf)(v) = s(fv) \end{aligned}$$

där $f, g : V \to W, v \in V$ och $s \in F$.

Varje linjär avbildning har två direkt relaterade delrum: nollrum och kolumnrum [Axl95]. Nollrummet hos en linjär avbildning $T : V \to W$ definieras av

$$\{v : Tv = 0\},$$

alltså mängden av vektorer som avbildas på nollvektorn. Därav innehåller nollrummet alltid nollvektorn. Kolumnrummet, även kallad värdemängden, delfinernas av

$$\{Tv : v \in V\}.$$

alltså mängden av alla möjliga avbildningar. Vidare är nollrummet ett delrum av V och kolumnrummet ett delrum av W. Dimensionen av kolumnrummet ges ett eget namn: rank.

2.3.5 Matris

Matriser är ett effektivt sätt att representera linjära transformer mellan ändliga vektorrum [Axl95]. Som en följd av egenskaperna hos linjära avbildningar och ändliga vektorrum kan transformer beskrivas entydigt genom deras effekt på basvektorerna. För en linjär transform $T : V \to W$ där $v_1, v_2, ..., v_n$ är en bas för V och $w_1, w_2, ..., w_m$ är en bas för W kan en matris

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

skapas så att $Tv = Av$ där $v \in V$. Konkret har vi att för en basvektor v_j är $Tv_j = Av_j = \sum_{i=1}^{m} a_{i,j} w_i$, alltså linjär kombinationen av skalärerna i kolumn j med basvektorerna i W. Eftersom alla vektorer i ett ändligt vektorrum kan skrivas som en linjär kombination av basvektorerna är tolkningen av en matris entydig.

2.3.6 Linjära ekvationssystem

Linjära ekvationssystem består av två eller flera linjära ekvationer där varje ekvation innehåller en eller flera okända variabler. Värdena på variablerna som ger lösningar på

alla ekvationer samtidigt ger lösningen eller lösningarna för det linjära ekvationssystemet [Axl95, Ryb22].

Ett ekvationssystem kan antingen ha inga möjliga lösningar, en unik lösning eller oändligt många lösningar. Lösningen till ett ekvationssystem kan tolkas som den punkt eller spann i ett koordinatsystem där lösningarna för samtliga ekvationerna skär eller överlappar varandra. Vid inga lösningar finns det ingen punkt eller spann i koordinatsystemet som uppfyller ekvationerna samtidigt. Vid en unik lösning finns det en specifik punkt i koordinatsystem som uppfyller ekvationerna samtidigt, här är variablerna oberoende av varandra och alla har ett konstant värde i lösningen. Vid oändligt många lösningar är variablerna beroende av varandra. Det finns därav ett oändligt antal lösningar som sträcker sig över något spann i koordinatsystemet.

Linjära ekvationssystem kan visualiseras som matriser där ekvationerna är radvektorer i en matris. Genom Gausselimination, se teori 2.5.1, kan lösningen fås ut från systemet.

2.3.7 Determinant

Determinant är en funktion som kan beräknas på kvadratiska matriser vilket resulterar i en skalär. Determinanten kan användas för att karakterisera matematiska egenskaper samt dess matris linjära avbildning. Mer specifikt inkluderar det bevisande av ifall kolumnvektorerna i matrisen är linjärt beroende och matrisens inverterbarhet [Axl95].

Ifall determinanten av en matris är skild från noll innebär det att kolumnvektorerna i matrisen är linjärt oberoende, att matrisen har en invers samt att det endast existerar en unik lösningen till matrisens linjära ekvationssystem. Ifall determinanten är noll för ett homogent ekvationssystem innebär det att dess ekvationer är linjärt beroende och att systemet har oändligt många lösningar.

Determinanter används även för att beräkna en matris karakteristiska polynom med dess egenvärden som rötter. $det(A - \lambda I) = 0$ där I är identitetsmatrisen och rötterna till ekvationen är egenvärdena λ för $(n \times n)$ matrisen A [Axl95].

Den vanligaste metoden som används för att beräkna determinanten är Laplace-expansion. Till skillnad från Sarrus regel för 3x3-matriser fungerar denna metod för kvadratiska matriser av valfri storlek. Determinanten av en $(n \times n)$ matris A ges av följande uttryck:

$$det(A) = \sum_{j=1}^{n} a_{1j}(-1)^{1+j} M_{1j}$$

Expansionen sker med så kallade underdeterminanter som är determinanter på delmatriser av grundmatrisen [Wei22]. Underdeterminanten M_{ij} bildas genom att eleminera rad i och kolumn j från A. Determinanten av A expanderas på följande sätt med hjälp av

underdeterminanterna.

$$det(A) =$$

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n3} & \cdots & a_{nn} \end{vmatrix} +$$

$$\cdots \pm a_{1n} \begin{vmatrix} a_{21} & a_{22} & \cdots & a_{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n(n-1)} \end{vmatrix}$$

2.3.8 Egenvektor och egenvärde

Egenvektorer, eller karakteristiska vektorer som de ibland kallas, är de vektorer vars spann inte ändras efter en linjär transformation utan skalas med ett konstant värde under transformationen [Axl95, 3Bl16]. Det konstanta värdet är vad som kallas vektorns egenvärde, eller karakteristiska värde. Om en linjär avbildning representeras av en matris, kan egenvärden och egenvektorer beräknas från matrisen och kallas ofta matrisens egenvektorer med dess associerade egenvärden.

Ett oförändrat spann för en vektor betyder att riktningen av vektorn förblir oförändrad. Det vill säga att en icke-noll egenvektor av en transformation är en vektor som behåller samma riktning och skalas med sitt associerade egenvärde efter genomförd transformation. Denna likhet ger upphov till följande ekvation:

$$Av = \lambda v \iff Av - \lambda Iv = 0 \iff (A - \lambda I)v = 0 \iff det(A - \lambda I) = 0$$

Där A = matrisen för den linjära transformationen, v = egenvektorn, λ = egenvärdet, I = identitetsmatrisen, det = determinant.

Ekvationen $det(A - \lambda I) = 0$ är känd som den karakteristiska ekvationen där det karakteristiska polynomet $det(A - \lambda I)$ sätts till noll och egenvärdena λ för matrisen A kan beräknas som rötterna av ekvationen [Axl95]. Utifrån samma ekvation, $Av - \lambda Iv = 0$ kan sedan egenvektorn för respektive funnet egenvärde beräknas fram genom att finna den homogena lösningen för det linjära ekvationsystemet beskrivet av matrisen $A(v) - \lambda Iv$.

2.4 Gles matris

En gles matris [Mat22] är en matris med mestadels nollor som element. Detta tillåter speciella lagringsmetoder där endast de nollskilda elementen lagras. Två representationer av glesa matriser är CSR eller Compressed sparse row och quadtree format.

2.4.1 Compressed Sparse Row

CSR formatet [EGSS82] representerar en $m \times n$ matris M genom tre listor, *elems*, *col* och *row*. Listan *elems* innehåller alla nollskilda element i matrisen M, sorterade i ordningen de förekommer från vänster till höger, rad för rad. Listan *col* innehåller kolumn-indexen för elementen i *elems* och är identiskt sorterade. Både V och *col* har storleken *NNZ*, där *NNZ* är antalet nollskilda element i matrisen M. Ett värde på index r i *row* representerar hur många nollskillda element det finns i matrisen i alla rader innan rad r.

Det första värdet i *row* är därmed 0, det sista är alltid *NNZ* och storleken blir $m + 1$. Exempelvis matrisen nedan är en 4×4 matris med bara 4 nollskilda element. CSR representationen visas till höger.

$$\begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix} \quad \begin{aligned} \mathit{elems} &= [5, 8, 3, 6] \\ \mathit{col} &= [0, 1, 2, 1] \\ \mathit{row} &= [0, 1, 2, 3, 4] \end{aligned}$$

Notera att skillnaded mellan värderna i *row* på index r och $r + 1$, dvs *row*$[r + 1]$ - *row*$[r]$, representerar hur många element det finns på raden r. Detta betyder att ifall ett element a skilt från noll tilläggs på första raden i exempel matrisen, blir matris och CSR representationerna följande:

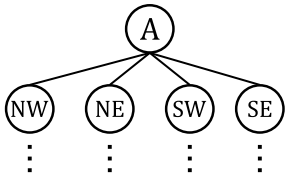
$$\begin{pmatrix} 5 & 0 & 0 & a \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix} \quad \begin{aligned} \mathit{elems} &= [5, \mathbf{a}, 8, 3, 6] \\ \mathit{col} &= [0, \mathbf{3}, 1, 2, 1] \\ \mathit{row} &= [0, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}] \end{aligned}$$

Exempelvis fås antal element på rad 0 genom att ta *row*$[1]$ - *row*$[0] = 2$. Det följer att för att extrahera en rad r ska elementen från index *row*$[r]$ till index *row*$[r + 1]$ tas ut ur *elems*.

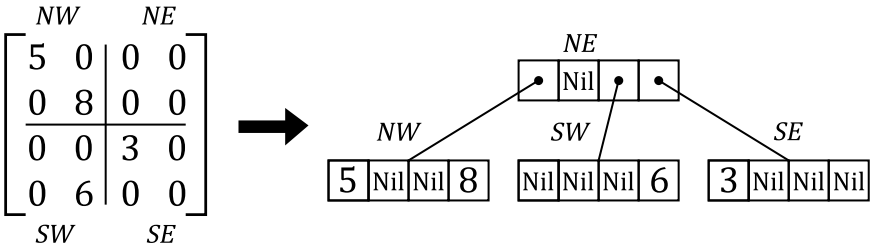
2.4.2 Quadtree

Quadtree är en trädbaserad datastruktur som baseras på principen av rekursiv nedbrytning [Sam84]. Som namnet föreslår är det ett träd där varje föräldernod har fyra barnnoder. Anledningen till deras användbarhet är för att denna struktur möjliggör att fokusera på datan som är mest viktig genom att representera den i en hierarki där noder av större höjd innehåller mer information om objektet som presenteras, medan noder av lägre höjd innehåller mindre information. Nyttan av att använda sig av denna struktur är att den effektivt hanterar minnesanvändning genom komprimering.

Ett användningsområde som är bra för denna presentation är glesa matriser där många delar utav matrisen inte har någon ny information eftersom den består mestadels av nollor. När en matris omvandlas till en quadtree-matris görs en rekursiv nedbrytning av matrisen i fyra delmatriser (noder).



Delmatriserna brukar vara märkta nordväst, nordost, sydväst och sydost som motsvarar kvadranterna i ett koordinatsystem. För varje delmatris går det att utföra ytterligare nedbrytningar fram tills den minsta beståndsdelen, som är ett enskilt element, nås. Detta är vad som menas med rekursiv nedbrytning. Löven i trädet erhålls till följd av den rekursiva nedbrytningen som antingen resulterar i en skalär skiljt från noll, eller en homogen delmatris som inte kräver mer nedbrytning.



Figur 2.2: Här illustreras den rekursiva nedbrytningen för en 4×4 matris och dess motsvarande quadtree presentation. Den nordöstra delmatrisen är en nollmatris och representeras av en Nil i quadtree presentationen.

Ett villkor är att matrisen behöver vara kvadratisk och i dimensioner av $2^n \times 2^n$ för att nedbrytningen ska fungera. Däremot går att lägga till rad och nollkolumner på en godtycklig $n \times n$ matris innan komprimering för att omvandla matrisen till rätt dimensioner.

2.5 Algoritmer

Inom detta projekt är två matematiska algoritmer av särskild betydelse, Gausselimination och Newtons metod. Gausselimination är en metod för att lösa ekvationssystem och Newtons metod är en metod för att hitta rötter till ett polynom. Dessa algoritmer kan användas enskilt eller i samband med andra metoder för att exempelvis hitta egenvärdet för matriser, se teori 2.3.8.

2.5.1 Gausselimination

Gausselimination eller radreduktion, är en algoritm för att lösa linjära ekvationssystem, se 2.3.6, och är benämnd efter tyske matematikern Carl Friedrich Gauss (1777–1855) [Axl95]. Gausseleminering kan även användas för andra matrisberäkningar som rang, determinant, invers och nollrum [Spa97]. Algoritmen består av en sekvens av elementära radoperationer för att nollställa elementen under matrisens diagonal. Det finns tre typer av elementära radoperationer.

Radbyte Radbyte görs genom att flytta en rad upp eller ned i matrisen. Ekvationerna som utgör ett linjärt ekvationssystem blir oförändrade, kolumnvektorernas element byter plats.

Radmultiplikation Multiplikation av varje element i en rad med en konstant skild från noll.

Radaddition Addition av en rad till en annan. Ofta kombinerad operation med radmultiplikation för att åstadkomma den vanligt använda eliminationsmetoden för att lösa ut ekvationer i linjära ekvationssystem. Genom radmultiplikation med en negativ konstant kombinerat med radaddition kan radsubtraktion åstadkommas.

Genom användning av dessa radoperationer transformeras matriser till trappstegsform. De elementära operationerna bevarar kolumnrummet, se teori 2.3.4, vilket innebär att matrisen i trappstegsform har samma lösningar som den ursprungliga matrisen innan Gausselimnationen. Fortsatt användning av elementära radoperationer kan matrisen konverteras till en unik reducerad trappstegsform. Gausselimination till reducerad trappstegsform kallas ofta för Gauss-Jordan elimination. Gauss och Gauss-Jordan elimination kan även utföras på kolumner istället för rader genom att transponera matrisen innan eliminationen utförs.

2.5.2 Newtons metod

Newtons metod är en numerisk metod som används för att hitta approximerade värden för nollställen av polynom [GYK22]. Det finns olika implementationer av algoritmen men för detta arbete används varianten som räknar ut roten för funktioner av en enda variabel.

Metoden baseras på att funktioner som är differentierbara och kontinuerliga kan approximeras av en rät linje som tangerar dem. Algoritmen av metoden är iterativ på så sätt att rötterna som erhålls vid en iteration kan användas för nästa iteration för att hitta en närmare approximation.

Låt $f : R \to R$ vara en kontinuerlig, differentierbar funktion. En första gissning utförs genom att välja en punkt x_0 som ligger godtyckligt nära roten vid första iterationen. Enligt Newtons metod finns en bättre approximation av roten som ges av:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Detta kan repeteras flera gånger om vilket är tanken bakom den iterativa processen av algoritmen. För varje rot x_n som ges av metoden, skall det finnas en bättre approximation x_{n+1}.

2.6 Haskell

Haskell är ett statiskt typat rent funktionellt programmeringsspråk [Has22b]. Att språket är rent funktionellt innebär att funktioner och uttryck kan betraktas utifrån deras matematiska definition. Alltså garanteras att deras värde inte ändras vilket lämpar sig väl för algebraiskt manipulation och för att skapa domänspecifika språk.

2.6.1 Language extensions

Utöver Haskell2010 standarden kommer GHC, The Glasgow Haskell Compiler, med ett flertal tillägg till språket [Has22a]. De viktigaste tilläggen för detta projekt beskrivs enligt följande:

DataKinds gör det möjligt att lyfta en datatyp till en sort [GHC22b]. På samma sätt som ett värde har en typ har varje typ en sort. Som ett exempel kan de naturliga talen lyftas till typnivå genom:

```
data Nat = Zero | Succ Nat
```

TypeFamilies kan ses som funktioner på typnivån [GHC22b]. Med `Nat` definierat ovan kan vi till exempel definiera addition på typnivå:

```
type family Add a b where
        Add Zero b = b
        Add (Succ a) = Add a (Succ b)
```

Tillägget kommer även med *associerade* typer. Dessa definieras som en del av en typklass, och är då endast definierade för typer som tillhör klassen.

```
class Container c where
    type Elem c :: *
    ...
```

GADTs, Generalised Algebraic Data Types, ger data konstruerare möjlighet att påverka typen [GHC22b]. Som ett exempel på detta se nedan:

```
data Container a where
    CInt  :: Int  -> Container Int
    CBool :: Bool -> Container Bool
    CSum  :: Container Int -> Container Int -> Container Int
```

Dessa språktillägg efterliknar funktionaliteten hos beroende typer, se teori 2.7.2.

2.6.2 QuickCheck

QuickCheck är ett bibliotek för testning av kod [Cla20]. För att använda QuickCheck skrivs funktioner som motsvarar de egenskaper som testas, t.ex:

```
prop_comutative :: Int -> Int -> Bool
prop_comutative a b = a + b == b + a
```

Dessa egenskaper kan sedan testas genom att låta QuickCheck generera ett stort antal slumpade testvärden. Biblioteket innehåller även funktionalitet för att skapa generatorer, som genererar slumpade testvärden, för egendefinierade datatyper.

2.6.3 TypeLits

Haskell modulen GHC.TypeLits lyfter värdena för naturliga tal, strängar och karaktärer till typnivån [GHC22a]. Modulen bygger på GHCs språktillägg DataKinds och TypeFamilies, som beskrivs i sektion 2.6.1. Detta innebär att, exempelvis, `1`, `5`, `12`, `"abc"`, `"typ"`, `'a'` och `'b'` blir konkreta typer samt att primitiva funktioner som `(+)`, `(-)`, `(*)`, `(^)` definieras för dem.

En viktigt del av TypeLits är att den möjliggör konvertering från typ till värde. För naturliga tal sker detta via klassen `KnownNat` och funktion

```
natVal :: KnownNat n => proxy n -> Integer
```

Notera att proxy är en typvariabel och behövs endast då Haskell funktioner kräver ett värde som indata.

2.7 Agda

Agda är ett beroende typat funktionellt programmeringspråk implementerat i Haskell [The22]. Bland Agdas kännetecken finns *induktiva typer, beroende typer, mönstermatchning, metavariabler och terminationskontroll.*

Utöver att vara ett programmeringspråk är Agda även ett interaktivt system för att skriva bevis. Detta då Agda är baserat på intuitionistisk typteori av Per-Martin Löf [ML75]. På följd av detta samt *terminerings kontroll*, konstrueras och testas bevis vid kompilering. Egenskaper av program kan beskrivas som satser av propositioner och predikat vars bevis härleds genom ekvationsresonemang. Logiska satser kan översättas till program med hjälp av Curry-Howard-korrespondensen som visar korrespondensen mellan propositioner och typer [Abe16].

2.7.1 Induktiva typer

Inductive types[BD09] är typer som kan defineras utifrån konstanter och funktioner som skapar värden av den typen. Ett exempel på en induktiv typ är denna definition av naturliga talen enligt Peanos axiom.

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

Detta betyder att det finns två sätt att skapa ett naturligt tal; `zero` som är ett naturligt tal (basfallet) eller funktionen `succ` som tar ett naturligt tal och ger tillbaka nästa naturligt tal (det induktiva steget).

2.7.2 Beroende typer

En beroende typ[BD09] är en typ vars definition beror på ett värde och används inom logik och datavetenskap. Inom logik används beroende typer för att koda logiska kvantifierare

så som “för alla” och “det existerar”. Inom datavetenskap används beroende typer för att möjligöra fortsatt tilldelning av typer för att minska mängden möjliga implementationer.

$\prod$-typer innehåller funktioner där de liknar vanliga funktioner med att de har en inmatning- och en utmatningstyp. $\prod$-typer är dock mer kraftfulla än vanliga funktioner då utmatningstypen kan bero på inmatningsvärdet. Ett exempel på hur detta kan användas är $\prod_{n:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, n)$. Detta är typen på en funktion som givet ett naturligt tal n returnerar en vektor som innehåller n reella tal.

$\sum$-typer innehåller ordnade par och kan, precis som vanliga ordnade par, beskriva kartesisk produkten mellan två typer. Utöver det är $\sum$-typer mer kraftfulla än vanliga par då typen av det andra värdet kan bero på det första värdet. Exempelvis kan en lista definieras enligt $\sum_{n:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, n)$ vilket säger att det första värdet är ett naturligt tal n och det andra värdet är en vektor av längd n.

3

Metod

Projektet delades upp i tre huvudmoment som kommer att beskrivas i detta kapitel. Huvudmomenten omfattade genomförandet av arbetet mot de uppsatta delmålen från sektion 1.5, testning och verifikation samt utvärderingen kring hur framgångsrikt projektet var.

3.1 Genomförande

Genomförandet bestod av HasLins implementation och därefter utvecklandet av en interaktiv miljö vilket tillät användarinteraktion med det domänspecifika språket.

3.1.1 Litteraturstudie

Ett fokus för projektet var korrekthet och därför påbörjades projektet med studerande av ett flertal läroböcker inom linjär algebra och Haskell. Detta gjordes för att säkerställa att gruppen hade tillräckliga grundkunskaper för att kunna realisera matematiken ur ett datavetenskapligt perspektiv.

I första hand studerades Sheldon Axlers bok *Linear Algebra Done Right* [Axl95], både innan och kontinuerligt under utvecklandet. Detta gjordes att säkerställa en bra förståelse för matematiken när det implementeras som kod. I andra hand studerades även boken *Linjär Algebra* av Gunnar Sparr [Spa97]. Boken behövdes när oklarheter uppstod från läsandet av den förstnämnda boken och när fler källor behövdes. Dessutom studerades Boken *Domain-Specific Languages of Mathematics* [JIB22a] från kursen *DSLofMath* [JIB22b] för att få en konkret bild av hur matematiken kunde överföras till Haskell. Utöver dessa studerades även en samling av datavetenskapliga källor som referens till algoritmer och datastrukturer.

3.1.2 Implementationen av DSL

En viktig del av DSL implementeringen var att skapa väl anpassade datatypsrepresentationer för grundtyperna som vektorer och matriser. Ett fokusområde var därför hur Haskells typ-system kunde nyttjas så att vektorers dimensioner är väl integrerade i det domänspecifika språket.

Ett flertal olika implementationer testades och användes för typerna och operationerna

mellan dem. Detta krävdes för att de olika implementationerna hade olika styrkor, vilket skapade ett val mellan korrekthet, prestanda och enkelhet.

3.1.3 Skapandet av en interaktiv upplevelse

I det slutliga steget sammanställdes en fil från det utvecklade domänspecifika språket för att möjliggöra bättre interaktion för användare. Filen inkluderade nödvändiga moduler som kombinerat gav möjligheten för användare att exekvera funktionalitet från matematiken beskrivet i det domänspecifika språket. Filen har laddats upp som öppen källkod på projektets GitHub [EJN+22a] och finns tillgänglig utan behov av nedladdning i webbläsaren med hjälp av Replit [SS22].

För att göra användandet mer pedagogiskt och materialet lättanvänt för samtliga användare inkluderades även viss handledning i filen. Handledningen var i form av kommentarer i källkoden som förklarade vad som kan utföras i respektive filer kombinerat med exempel.

3.1.4 Test och verifikation

Genom projektets gång användes tre olika slags test, enhetstestning, användartest och prestandatest. Programmeringspråket Agda användes även för att bevisa vissa delar. De matematiska lagar som bevisades valdes från den utvalda linjär algebra litteraturen, *Linear Algebra Done Right* [Axl95] och *Linjär Algebra* [Spa97].

Enhetstest användes för att verifiera att de matematiska egenskaperna för vissa skapade funktioner och operationer var korrekta. Enhetstestning skedde med QuickCheck [Cla20] på en del egenskaper av programmet, vilket inkluderar ett par egenskaper för matriser och vektorer samt några operationer mellan dem. För att möjligöra testning med QuickCheck skapades det vissa egenskaper och generatorer, där egenskaperna var de lagar som testades och generatorerna användes för att testa slumpmässigt genererade argument.

Bevisande av lagar har delvis även skett i Agda för ett urval av de lagar som definierats i Haskell. Det här gjordes eftersom Agda använder sig av propositioner som typer vilket resulterar i att de bevis som testades konstruerades av språket under typ kontrollen. Det här gav ett starkare bevis än de generade funktions argument som produceras av QuickCheck. Agda är särskilt lämplig när reella tal hanteras eftersom dess lagar kan representeras direkt, medans reella tal i Haskell måste representeras som icke exakta flyttal. I dessa situationer krävdes en annan metod att bevisa korrekthet av vårt DSL och då implementerades ett bevis i Agda.

Användartest utfördes på ett par frivilliga personer som fick testa att följa en ihopsatt handledning inom filen. Testarna fick ge respons över hur enkelt dem tyckte det var att orientera sig genom filen, hur praktiskt användbart dem ansåg HasLin vara samt vilka möjliga förändringar som skulle kunna förbättra upplevelsen.

Prestandatest användes för att se skillnaden i prestanda mellan de olika implementationerna i programmet. Detta krävdes för att kunna bevisa och föra ett argument kring implementationernas styrkor och svagheter.

3.2 Utvärdering

Under projektets gång skedde en kontinuerlig dialog med handledare Patrik Jansson, examinator för kursen DAT326/ DIT982 *DSLofMath* på Chalmers och GU [JIB22b]. All källkod var tillgänglig för handledaren under arbetets gång att fritt analysera och kommentera. Genom denna kontinuerliga respons på arbetet gjordes ett flertal mindre utvärderingar där de hittills uppnådda framstegen jämfördes mot delmålen. Vid projektets avslut utvärderades arbetet genom att svara på de följande frågorna:

Inkluderar HasLin de matematiska områden som avsågs enligt avgränsningarna? Identifieras de grundtyper och koncept nödvändiga för ett DSL inom linjär algebra? Uppnår projektet delmålen?

Projektet utvärderades även utifrån användartester och resultaten av prestandatester.

4

Resultat

Detta kapitel redovisar resultaten som uppnåtts i projektet. Resultaten presenteras utifrån de projektmål som beskrevs i sektion 1.5. För enklare läsbarhet tas endast en del exempel upp för hur saker blev gjorda och de generella resultaten av algoritmer och tester. För den fullständiga informationen och koden, se projektets GitHub [EJN+22a].

4.1 Analys av linjära algebrans grund

Litteraturstudien vilket projektet byggts på skedde huvudsakligen från *Linear Algebra Done Right* [Axl95], *Linjär Algebra* [Spa97] och *Domain-Specific Languages of Mathematics* [JIB22a] vilket var bestämt vid projektets start. Samtliga av de grenar som arbetet avgränsades till, vektorer, matriser, linjära rum, linjära avbildningar, linjära ekvationssystem, egenvärden och egenvektorer ingick i den studien.

4.2 Matematiken i funktionell programmering

I denna sektion kommer de mest centrala delarna av den programmerade matematiken att uppvisas. All matematik som beskrivs i teorin, se kapitel 2, kommer här visas i kod samt även medföljande exempel på hur koden kan användas.

4.2.1 Algebra

I teoridelen, se avsnitt 2.2, introduceras algebraiska strukturer. Ett naturligt sätt att representera strukturer i Haskell är att använda typklasser. Följande typklasser definieras som en motsvarighet till den algebraiska hierarki som beskrevs i avsnitt 2.2.

```haskell
class AddGroup a where
    (+)  :: a -> a -> a
    zero :: a
    (-)  :: a -> a -> a
    a - b = a + neg b

    neg :: a -> a
    neg a = zero - a
```

```haskell
class Mul a where
    (*) :: a -> a -> a
    one :: a
```

```haskell
type Ring a = (AddGroup a, Mul a)

class Ring a => Field a where
    (/) :: a -> a -> a
    a / b = a * recip b

    recip :: a -> a
    recip a = one / a
```

Här är den abstrakta algebrans grund definierad i Haskell. En multiplikativ monoid vilket innehåller en associativ binär operator multiplikation `(*)`, samt identitetselementet för multiplikation `one`. En multiplikativ monoid kombinerat med en additiv abelsk grupp, med operatorerna `(+)`, `(-)` och identitet `zero`, bildar en ring, se avsnitt 2.2.3. Ringen kompletteras sedan med multiplikativ invers, division, och bildar en kropp (`Field`), se avsnitt 2.2.4.

Från denna grund av abstrakt algebra kan instanser skapas som följer de matematiska lagarna. Detta inkluderar exempelvis instanser för simplare datatyper så som heltal (`Int`) och flyttal (`Double`) vilket gör det möjligt att utföra operationer på heltal och flyttal. Heltal och flyttal är datatyper som redan finns i Haskells standardmodul `Prelude`, förkortat nedan med `P`.

```haskell
instance AddGroup Int     where (+) = (P.+); (-) = (P.-); zero = 0
instance AddGroup Double  where (+) = (P.+); (-) = (P.-); zero = 0

instance Mul Int          where (*) = (P.*); one = 1
instance Mul Double       where (*) = (P.*); one = 1

instance Field Double     where (/) = (P./);
```

Instanser av den abstrakta algebran kan vidare skapas för mer komplexa datatyper samt även egendefinierade sådana. Detta möjliggör att skapa egna datatyper vilka kan vara användbara för att specificera en viss domän och sedan konstruera instanser som följer matematiska lagar för datatyperna utifrån abstrakta algebran. Nedanför visas en sådan implementerad datatyp `Exp` för uttryck av en variabel. `Exp` har konstruerare för de aritmetiska uttryck som inkluderas i en kropp, se teoriavsnitt 2.2.4, en konstruerare för variabler och en för konstanter. Med denna datatyp kan uttryck av en variabel uttryckas i HasLin.

```haskell
type R = Double
data Exp  =  Const R
          |  X
          |  Exp :+: Exp
          |  Negate Exp
          |  Exp :*: Exp
          |  Recip Exp
```

För denna skapade datatyp kan sedan instanser av den abstrakta algebran implementeras. Utöver det kan även instanser för funktioner skapas. Nedanför visas hur instanser av `AddGroup` har implementerats för datatypen `Exp` samt för alla funktioner `a->b` om `b` är en instans av `AddGroup`.

```
instance AddGroup Exp   where
    (+) = (:+:)
    neg = Negate
    zero = Const 0

expressionE :: Exp
expressionE = neg (Const 2) + X + zero

ghci> expressionE
-(2.0) + X

instance AddGroup b => AddGroup (a -> b)  where
    f + g = \x -> f x + g x
    neg f = \x -> neg (f x)
    zero = const zero

expressionF :: Double -> Double
expressionF = neg (const 2) + id + zero

ghci> expressionF 3
1
```

`Exp` är en djup inbäddning av ett DSL, se avsnitt 2.1. Datatypen representerar ett abstrakt syntaxträd för uttryck av en variabel. Syntaxen av uttrycken kan sedan genom användning av en evaluerare översättas till semantiska värden. Evalueraren som implementerats tar emot ett syntaktiskt uttryck, ett värde för variabeln och evaluerar uttrycket till ett semantiskt värde. Nedanför visas hur uttryck evalueras genom mönstermatchning och rekursion på inmatat uttryck.

```
-- Eval for expressions, apply a value for X
evalExp :: Exp -> R -> R
evalExp (Const alpha)  =  const alpha
evalExp X              =  id
evalExp (e1 :+: e2)    =  evalExp e1 + evalExp e2
evalExp (e1 :*: e2)    =  evalExp e1 * evalExp e2
evalExp (Negate e)     =  neg (evalExp e)
evalExp (Recip e)      =  recip (evalExp e)
```

```
exp1 :: Exp                 ghci> evalExp exp1 2
exp1 = X                    2.0

exp2 :: Exp                 ghci> evalExp exp2 2
exp2 = Const 2 * X          4.0

exp3 :: Exp                 ghci> evalExp exp3 2
exp3 = recip (X * X)        0.25
```

4.2.2 Vektorer och linjära rum

Med den abstrakta algebrans grund implementerad kunde implementation av vektorer och vektorrum, även kallat linjärtrum, påbörjas. I klassdefinitionen av ett vektorrum medföljer att vektortypen `v` behöver vara del av klassen `AddGroup` samt att den underliggande typen, det vill säga skalären till `v`, har en instans av `Ring`. Klassen `VectorSpace` representerar ett vektorrum av en mängd vektorer som är slutet under addition och skalärmultiplikation.

```
class (AddGroup v, Ring (Under v)) => VectorSpace v where
    type Under v
    (£) :: Under v -> v -> v
```

I klass deklarationen definieras den associerade typen `Under v` för att ange skalärens typ. Namnet är en tolkning av det typiska uttrycket "V är ett vektorrum *över* F" men ger större vikt på Fs beroende av V. Givet detta kan (£) utläsas som skalär multiplikation.

En noterbar skillnad från definitionen av vektorrum, se avsnitt 2.3.1, är att denna klass endast kräver att skalären är en *ring* snarare än en *kropp*, vilket definierar den generaliserade strukturen av vektorrum nämligen modul över en ring. Detta då ring är tillräckligt för att definiera vektoregenskaperna. Samtidigt kan kravet av en kropp specificeras för de funktioner som kräver det. Som ett exempel på en vektorrepresentation ges följande instans för funktioner, notera att funktioner redan har en instans för `AddGroup`:

```
instance Ring b => VectorSpace (a -> b) where
    type Under (a -> b) = b
    s £ f = \x -> s * f x
```

En mer typisk representation av vektorer är listor med värden av en given längd. Denna representation implementeras i HasLin enligt följande:

```
newtype Vector f (n :: Nat) = V [f]
```

Sort-signaturen `n :: Nat`, vilket görs möjligt av språktillägget DataKinds, se teori 2.6.1, markerar att `n` är ett naturligt tal. Detta används i HasLin för att typkontrollera operationer så som vektoraddition och matris-vektor-multiplikation. När vektorer skapas måste

därför listans längd vara lika med vektorns dimension, vilket kontrolleras med den smarta konstruktorn `vec`. Som exempel på datatypen kan följande instanser göras.

```
vec :: KnownNat n => [f] -> Vector f n
vec ss = if (vecLen v == length ss) then v
                                    else error errorMsg
    where v = V ss
          errorMsg = "Vector is of dimension " ++ show (vecLen v) ++
                " but was given a list of length " ++ show (length ss)

v1 :: Vector Int 4
v1 = vec [1,2,3,4]

v2 :: Vector Char 3
v2 = vec ['v','e','c']
```

Givet har `Vector` en `VectorSpace` instans. Först definieras `AddGroup` enligt:

```
instance (KnownNat n, AddGroup f) => AddGroup (Vector f n) where
    (+) = zipWithV (+)
    (-) = zipWithV (-)
    zero = zeroVec
```

Instansen introducerar två nya funktioner, `zipWithV` och `zeroVec`. `zipWithV` motsvarar `zipWith` på listor men är endast typad för vektorer av samma längd. `zeroVec` skapar en `Vector` som endast innehåller `zero`.

```
zeroVec :: (KnownNat n, AddGroup f) => Vector f n
zeroVec = let v = V $ replicate (vecLen v) zero in v
```

`zeroVec` är intressant då den tydligt visar hur vektorns värde är beroende av dess typ. Notera särskilt hur `vecLen v` måste returnera ett värde innan `v` kan skapas samtidigt som den beror på `v`. Detta är möjligt då `vecLen` endast behöver typinformation.

Med en `AddGroup` instans ges följande `VectorSpace` instans:

```
instance (KnownNat n, Ring f) => VectorSpace (Vector f n) where
    type Under (Vector f n) = f
    s £ v = mapV (s*) v
```

Vidare definieras skalär- och kryssprodukt enligt nedan.

```
dot :: Ring f => Vector f n -> Vector f n -> f
V v1 `dot` V v2 = sum $ zipWith (*) v1 v2

cross :: Ring f => Vector f 3 -> Vector f 3 -> Vector f 3
V [a1,a2,a3] `cross` V [b1,b2,b3] = V [a2*b3 - a3*b2,
                                       a3*b1 - a1*b3,
                                       a1*b2 - a2*b1]
```

4.2.3 Matriser och linjära avbildningar

Matriser har tre olika representationer i HasLin: Listor i listor, CSR och Quad. Dessa tre representationer har sina egna styrkor och svagheter beroende på användarens behov och matrisens struktur. Gemensamt för representationerna är att de tillhör följande matrisklass:

```
class Matrix (mat :: * -> Nat -> Nat -> *) where

    values :: mat f m n -> [((Fin m, Fin n), f)]

    tabulate :: AddGroup (mat f m n) =>
                            [((Fin m, Fin n), f)] -> mat f m n

    ...
```

Konceptuellt bygger klassen på att varje värde i en matris är associerat till ett index. Klassen syfte är därför att generalisera ett indexbaserat gränssnitt för HasLins olika matrisrepresentationer. Med detta i åtanke behöver två saker specificeras: sort-signaturen av en matristyp och en gemensam index-typ.

Sort signaturen `mat :: * -> Nat -> Nat -> *` uttrycker att en matristyp tar tre argument: en typ och två naturliga tal. Dessa argument motsvarar följande i matrisen: skalärens typ, antalet rader och antalet kolumner.

Indextypen som används i klassen och dess smarta konstruktor är följande:

```
newtype Fin (n :: Nat) = Fin Int

fin :: KnownNat n => Int -> Fin n
fin i = finite
    where finite | 1 <= i && i <= fromInteger (natVal finite) = Fin i
                 | otherwise = error $ "Index is out of bounds, got: "
                   ++ show i ++ " in constraint 0<" ++ show i ++ "<="
                   ++ show (natVal finite)
```

Likt `Vector` från sektion 4.2.2, använder sig `Fin n` av `Nat` för att markera dess storlek.

I detta fall syftar storleken till det största indexet. Med andra ord kan typen `Fin 3` ses som en mängd innehållande värdena { `Fin 1`, `Fin 2`, `Fin 3` }.

Med detta som bakgrund kan klassens främsta funktioner `values` och `tabulate` beskrivas. `values` konverterar en matris till en lista med index och värde-par. `tabulate` konverterar en lista med index och värde-par till en matris. För att bättre stödja glesmatriser tolkas frånvaro av ett index i listorna som att indexet är associerat med värdet 0. Med dessa funktioner är det möjligt att skapa och omvandla matriser av olika representationer. Några exempel på implementerade funktioner är:

```
transpose :: (Matrix mat, AddGroup (mat f n m)) =>
                                        mat f m n -> mat f n m
transpose = tabulate . map (\((i,j),a) -> ((j,i),a)) . values


identity :: (KnownNat n, Matrix mat, AddGroup (mat f n n), Mul f) =>
                                        mat f n n
identity = tabulate [ ((i,i), one) | i <- [minBound .. maxBound]]


changeRep :: (Matrix mat1, Matrix mat2, AddGroup (mat2 f m n)) =>
                                        mat1 f m n -> mat2 f m n
changeRep = tabulate . values
```

Utifrån klassdeklarationen behövs det även skapas instanser av denna klass. Den första och enklaste representationen av matriser är matriser som listor i listor vilket implementerades enligt följande:

```
newtype Matrix f (m :: Nat) (n :: Nat) = M (Vector (Vector f m) n)
```

I denna implementation definieras matriser som en vektor av längd `n` som innehåller vektorer av längd `m` som innehåller `f`. I denna kontext är `m` och `n` naturliga tal som säger hur långa vektorerna är och `f` är typen på de värden som fyller upp matrisen. Typen `f` är inte bunden till att vara någon speciell typ i matrisen, däremot kan funktioner och instansdeklarationer, likt `AddGroup (Matrix f m n)`, ha begränsningar gällande `f`'s typ.

```
instance (KnownNats m n, AddGroup f) => AddGroup (Matrix f m n)
    where
    M as + M bs = M $ zipWithV (+) as bs
    M as - M bs = M $ zipWithV (-) as bs
    zero = let v = V $ replicate (vecLen v) zero in M v
```

Då dessa matriser är uppbyggda av vektorer kan matrisfunktioner ta hjälp av de funktioner som skrivits för vektorer. Detta har förenklat skapandet av funktioner och instansdeklarationer för matriser i och med att när matriser packas upp är de bara vektorer och

det går därmed att behandla dem på samma sätt som vektorer. Ett exempel på detta är `AddGroup (Matrix f m n)` instansen ovan. Några ytterligare exempel på detta finns nedan, där (££) är matris-vektor multiplikation och (£££) är matris-matris multiplikation.

```
instance (KnownNats m n, Ring f) =>
        VectorSpace (Matrix f m n) where
    type Under (Matrix f m n) = f
    s £ m = (s £) `onCols` m

(££) :: (KnownNat m, Ring f) =>
        Matrix f m n -> Vector f n -> Vector f m
M vs ££ v = linComb vs v

linComb :: VectorSpace v => Vector v n -> Vector (Under v) n -> v
linComb (V vs) (V fs) = sum $ zipWith (£) fs vs

(£££) :: (KnownNat a, Ring f) =>
        Matrix f a b -> Matrix f b c -> Matrix f a c
a £££ b = (a ££) `onCols` b

onCols :: (Vector f b -> Vector f a) ->
        Matrix f b c -> Matrix f a c
onCols f (M v) = M $ mapV f v
```

`CSR` representationen av glesa matriser är implementerad på följande sätt:

```
data CSR f (m :: Nat) (n :: Nat) =
     CSR { elems :: [f], col :: [Int], row :: [Int] }
```

Typen `f` är även här typen för elementen i matrisen. Listorna `elems`, `col` och `row` representerar elementen och indexen för matrisen. Se mer detaljerad beskrivning i teoriavsnittet 2.4.1.

En exempelfunktion som lämpar sig väl att applicera på matriser i CSR-format är funktionen `getSparseRow`, som returnerar en given gles rad som lagrar nollskilda elementen och vilken kolumn de ligger på i en lista av index-värde-par:

```
-- | returns a given sparse row in the sparse matrix
getSparseRow :: CSR f m n -> Int -> [(Int, f)]
getSparseRow (CSR elems col row) i = case td i 2 row of
        [a,b] -> td a (b-a) (zip col elems)
        _     -> []
        where td i j = take j . drop i
```

Detta görs genom samma metod som nämnd i 2.4.1, nämligen att plocka ur alla element från *elems* och dess korresponderande kolumn värden från *col* som ligger mellan index *row[i]* och index *row[i+1]*

Den tredje matrisrepresentationen i HasLin är `Quad`. Quad baseras på quadtree, se teoriavsnitt 2.4.2, och representerar glesa matriser genom att låta dess kvadranter antingen vara en mindre matris, en skalär eller en noll-matris. Genom att använda GADTs och Datakinds, se avsnitt 2.6.1, kan en Quad garantera att alla dess skalärer hamnar på samma djup, vilket nyttjas i denna implementation.

```
data Quad (n :: Nat4) a where
    Zero :: Quad n a
    Scalar :: a -> Quad One a
    Mtx :: Sized n => {
        nw :: Quad n a,
        ne :: Quad n a,
        sw :: Quad n a,
        se :: Quad n a
           } -> Quad (Suc n) a
```

Datatypen introduceras en ny sort `Nat4` som representerar djupen, och som följd även storleken hos en Quad-matris.

```
data Nat4 = One | Suc Nat4
```

Som exemplen är `Quad One a` en 1×1 matris, `Quad (Suc One) a` en 2×2 matris, `Quad (Suc (Suc One)) a` en 4×4 matris och så vidare. Utifrån datatypen noteras att storleken på Quads konstruerare är begränsade: `Zero` är en $2^n \times 2^n$ matris, `Scalar a` är en 1×1 matris, `Mtx nw ne sw se` är en $2^{n+1} \times 2^{n+1}$ matris givet att storlekarna på `nw, ne, sw, se` är $2^n \times 2^n$.

En fördel med att representera glesmatriser som quadtrees är att de lämpar sig väl för funktionell programmering. Huvudsakligen för möjligheten att använda mönstermatchning och rekursion vid skapandet av funktioner. Som exempel definieras addition och transponat enligt följande:

```
addQ :: AddGroup a => Quad n a -> Quad n a -> Quad n a
Zero     `addQ` x        = x
x        `addQ` Zero     = x
Scalar a `addQ` Scalar b = Scalar (a+b)
Mtx nw1 ne1 sw1 se1 `addQ` Mtx nw2 ne2 sw2 se2 =
                             Mtx (nw1 `addQ` nw2) (ne1 `addQ` ne2)
                                 (sw1 `addQ` sw2) (se1 `addQ` se2)

transposeQ :: Quad n a -> Quad n a
transposeQ Zero = Zero
transposeQ (Scalar s) = Scalar s
transposeQ (Mtx nw ne sw se) = Mtx (transposeQ nw) (transposeQ sw)
                                   (transposeQ ne) (transposeQ se)
```

Det bör noteras att `addQ` inte har något fall vid addition av `Scalar` och `Mtx`. Detta är säkert då användandet av GADTs ger dessa konstruerare olika typer, alltså nyttjas Haskells typsystem för att garantera att addition aldrig kan ske mellan dem.

En nackdel med att `Quad` endast representerar matriser av storlek $2^n \times 2^n$ är att den inte kan ges en `Matrix` instans. Därav packas `Quad` in i en ny datatyp `QuadM`.

```
data QuadM f (m :: Nat) (n :: Nat) = Sized (ToNat4 m n) =>
                                     QuadM (Quad (ToNat4 m n) f)
```

`QuadM` kan ses som ett $m \times n$ fönster till den annars kvadratiska `Quad`. För att garantera att `Quad` är minst lika stor som detta fönster används typfunktionen `ToNat4` som returnerar en `Nat4` av korrekt storlek.

4.2.4 Linjära ekvationsystem

Linjära ekvationssystem representeras med matriser där varje kolumn representerar en variabel och varje rad representerar en ekvation, se avsnitt 2.3.6.

$$\begin{cases} 2x + y - z = 8 \\ -3x - y + 2z = -11 \\ -2x + y + 2z = -3 \end{cases} \implies \left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right]$$

I HasLin representeras ovanstående linjära ekvationssystem enligt följande.

```
-- Augmented matrix
m :: Matrix R 3 4
m = toMatT [[ 2,  1, -1,  8],
            [-3, -1,  2, -11],
            [-2,  1,  2, -3]]
```

Matrisen `m` representerar det linjära ekvationssystemet. Systemet är redo att lösas med Gausseliminering och därmed användande av elementära radoperationer, se avsnitt 2.5.1. Följande datatyp har implementerats för att representera de tre elementära radoperationerna radbyte, radmultiplikation och radaddition.

```
data ElimOp n a = Swap   (Fin n) (Fin n)
                | Mul    (Fin n) a
                | MulAdd (Fin n) (Fin n) a
```

Utifrån datatypen `ElimOp` kan användaren utföra elementära radoperationer för hand. Nedanför visas exempel på hur Gausselemination kan utföras på matrisen m. Funktionen `foldElemOpsFunc` tar en lista av `ElimOp`s och utför motsvarande radoperationer på en matris.

```
ghci> m
|  2.0   1.0  -1.0    8.0 |
| -3.0  -1.0   2.0  -11.0 |
| -2.0   1.0   2.0   -3.0 |

ghci> foldElemOpsFunc [Mul 1 (1/2), MulAdd 1 2 3, MulAdd 1 3 2] m
|  1.0   0.5  -0.5    4.0 | -- Matrix and trace after row 1
|  0.0   0.5   0.5    1.0 |
|  0.0   2.0   1.0    5.0 |

ghci> foldElemOpsFunc [Mul 2 2, MulAdd 2 3 (-2)] it
|  1.0   0.5  -0.5    4.0 | -- Matrix and trace after row 2
|  0.0   1.0   1.0    2.0 |
|  0.0   0.0  -1.0    1.0 |

ghci> foldElemOpsFunc [Mul 3 (-1)] it
|  1.0   0.5  -0.5    4.0 | -- Matrix and trace after row 3
|  0.0   1.0   1.0    2.0 |
|  0.0   0.0   1.0   -1.0 |
```

Gausselimination kan också utföras med funktionen `gauss` som direkt returnerar matrisen i trappstegsform. Funktionen `gaussTrace` kan returnera motsvarande elementära radoperationer som utförts.

```
ghci> gauss m
| 1.0  0.5  -0.5   4.0 |
| 0.0  1.0   1.0   2.0 |
| 0.0  0.0   1.0  -1.0 |

ghci> gaussTrace m
[Mul 1 0.5, MulAdd 1 2 3.0, MulAdd 1 3 2.0,
 Mul 2 2.0, MulAdd 2 3 (-2.0), Mul 3 (-1.0)]
```

För att få ut lösningsmängden till det linjära ekvationssystemet introduceras begreppen nollrum, se avsnitt 2.3.4, samt delrum och kvotrum, se avsnitt 2.3.2. Ett delrum, `Subspace v`, är definierat som en lista av vektorerna `v` vilka spänner upp rummet. Nollrummet, `nullSpace`, kan därmed definieras som ett delrum av de vektorer som avbildas på nollvektorn. I termer av linjära ekvationssystem innebär det att nollrummet som beräknas nedan ger lösningsmängden till det homogena ekvationssystemet $Ax = 0$. Kvotrummet, `QuotientSpace v`, definieras som ett delrum förflyttat av en vektor.

```
newtype Subspace v = Sub [v]

nullSpace :: (KnownNat n, Field f, Eq f) =>
                            Matrix f m n -> Subspace (Vector f n)
nullSpace m =
  Sub $
    [ V b | (a,b)<- splitAt height <$> reduceCol m, all (== zero) a]
        where height = length (head $ unpack m)
              reduceCol m =
                unpack $ transpose $ gauss (transpose m `append` idm)

data QuotientSpace v = Quot v (Subspace v)
```

Lösningen till ett ekvationssystem representeras genom ett kvotrum bestående av systemets homogena lösningsmängd förflyttat av en partikulärlösning till systemet. Hur en partikulärlösning beräknas ut genom bakåtsubstitution visas nedan.

```
solve :: (Eq f, Field f) => [[f]] -> [f]
solve m = foldr next [last (last m)] (init m)
    where next row found = let
            subpart = init $ drop (length m - length found) row
            solved = last row - sum (zipWith (*) found subpart)
            in solved : found

particularSol :: (Eq f, Field f) => Matrix f m n -> Vector f (n -1)
particularSol = V . solve . unpack . transpose . gauss

triM = gauss m
ghci> triM                          ghci> particularSol triM
| 1.0  0.5  -0.5   4.0 |            |  2.0 |   -- x = 2
| 0.0  1.0   1.0   2.0 |            |  3.0 |   -- y = 3
| 0.0  0.0   1.0  -1.0 |            | -1.0 |   -- z = -1
```

Ett kvotrum för lösningsmängden beräknas enligt någon av följande funktioner, `solveQ` eller `solveQ'`.

```
-- | Returns the set of solutions to Ax=v
solveQ :: (KnownNat n, Field f , Eq f, (n ~ (n+1-1)) )
          => Matrix f m n -> Vector f m -> QuotientSpace(Vector f n)
solveQ m v = Quot (particularSol $ m `appendV` v) (nullSpace m)

-- | Equivalent to solveQ but takes a matrix A `append` v of Ax=v
solveQ' :: (KnownNat n, Field f, Eq f, (n ~ (n+1-1)) )
              =>  Matrix f m (n+1)  -> QuotientSpace (Vector f n)
solveQ' m = let (v', m') = last $ separateCols m in solveQ m' v'
```

Nollrummet för en matris där vektorerna är linjärt oberoende är alltid noll, det vill säga att matrisen har endast en lösning vilket är den partikulära lösningen. För matriser vars kolumnvektorer är linjärt beroende har systemet oändligt med lösningar inom kvotrummet. Nedan visas exempel på hur två olika linjära ekvationssystem löses ut, ena linjärt oberoende, och den andra linjärt beroende.

```
ghci> aM1                          ghci> aM2
|  2.0   1.0  -1.0 |               | 1.0  0.0  1.0 |
| -3.0  -1.0   2.0 |               | 2.0  1.0  2.0 |
| -2.0   1.0   2.0 |               | 1.0  1.0  1.0 |

ghci> bM1                          ghci> bM2
|   8.0 |                         | 1.0 |
| -11.0 |                         | 3.0 |
|  -3.0 |                         | 2.0 |

-- | Checks if the vectors in a matrix are linearly independent
linIndep :: (Eq f, Field f) => Matrix f m n -> Bool
linIndep = not . any (\(v, m) -> m `span'` v ) . separateCols

ghci> linIndep aM1                 ghci> linIndep aM2
True                               False

ghci> nullSpace aM1                ghci> nullSpace aM2
Sub []                             Sub [
                                   |  1.0 |
                                   | -0.0 |
                                   | -1.0 |
                                   ]
-- Augmented matrices
m1 = aM1 `append` bM1              m2 = aM2 `append` bM2

ghci> particularSol m1             ghci> particularSol m2
|  2.0 |                          |  1.0 |
|  3.0 |                          |  1.0 |
| -1.0 |                          |  0.0 |
```

Lösningen till vardera av det två ekvationssystemen ges av kvotrummen *particularSol* + *nullSpace*. Hela beräkningen av kvotrummet kan göras genom användandet av `solveQ` alternativt `solveQ'`. För att visualisera svaret i form av strängar, i syftet att förmedla svaret annorlunda för att gagna pedagogiken, implementerades även en funktion *showSol*.

```
ghci> solveQ' m1
Quot
|  2.0 |
|  3.0 |
| -1.0 |
 (Sub [])

-- solution m1 =
|  2.0 |
|  3.0 |
| -1.0 |
--

ghci> showSol $ gauss m1
x1 = 4.0 - 0.5*x2 + 0.5*x3
x2 = 2.0 - 1.0*x3
x3 = -1.0
```

```
ghci> solveQ' m2
Quot
|  1.0 |
|  1.0 |
|  0.0 |
(Sub [
|  1.0 |
| -0.0 |
| -1.0 |
])

 solution m2 =
| 1.0 |         |  1.0 |
| 1.0 |  +  s * |  0.0 |
| 0.0 |         | -1.0 |
 where (s) is an arbitrary scalar

ghci> showSol $ gauss m2
x1 = 1.0 - 1.0*x3
x2 = 1.0
```

4.2.5 Egenvärden och egenvektorer

I HasLin har en implementation gjorts för att beräkna egenvärden och vektorer från den karakteristiska ekvationen, se teoriavsnitt 2.3.8. För att få fram det karakteristiska polynomet behövdes en implementation för att beräkna matrisers determinant, se avsnitt 2.3.7, två olika implementationer skapades för determinanter.

```
detNN :: Field f => Matrix f n n -> f
detNN (M (V [V [x]])) = x
detNN m = sum $ zipWith (*) (cycle [one, neg one]) $ do
    (V (s:_), subM) <- separateCols m
    let (_, m') = head $ separateRows subM
    return $ s * detNN m'

detGauss :: (Field f, Eq f) => Matrix f n n -> f
detGauss m =
    let V diag = getDiagonal utfM in product diag / traceProduct
        where trace = utfTrace m
              utfM = foldElemOpsFunc trace m
              traceProduct = product [ s | Mul _ s <- trace ]
```

`detNN` baseras på Laplace expansion, vilket är den algoritm som traditionellt sätt lärs ut, och `detGass` på Gausselimination, se teori 2.5.1.

Det karakteristiska polynomet representeras genom den skapade datatypen **Exp**, notera dess implementation i resultatdel 4.2.1. Med **Exp** kan matriser av envariabelsuttryck skapas och ett polynom kan beräknas fram från matrisens determinant på följande vis.

```
m :: Matrix Exp 2 2
m = toConst (toMat[[3/4, 1/4],[1/4, 3/4]]) - X & idm

ghci> m                    ghci> detNN m
| 0.75 - X       0.25 |    (0.75 - X) * (0.75 - X) - (0.25 * 0.25)
|     0.25   0.75 - X |
```

Med det karakteristiska polynomet funnet kvarstår att finna rötterna till ekvationen lika med 0. En implementation av Newtons metod, se avsnitt 2.5.2, skapades för att approximera rötterna till karakteristiska ekvationen och därmed få ut egenvärdena hos matrisen. För genomförande av metoden krävs beräkning av den karakteristiska ekvationens derivata. En funktion *derive* har implementerats som med mönstermatchning och rekursion beräknar derivatan av uttryck enligt derivatans lagar för aritmetiska uttryck. Implementationen visas nedan där egenvärdena för 2x2 matrisen approximeras till 0.5 och 1.

```
derive :: Exp -> Exp
derive (Const alpha) = Const 0
derive X             = Const 1
derive (e1 :+: e2)   = derive e1 :+: derive e2
derive (e1 :*: e2)   = (derive e1 :*: e2) :+: (e1 :*: derive e2)
derive (Negate e)    = neg (derive e)
derive (Recip e)     = neg (derive e :*: (recip (e^2)))

evalExp' :: Exp -> R -> R
evalExp' =  evalExp . derive

newton :: Exp -> R -> R -> R
newton f eps x =
     if abs fx < eps then x
                     else if abs fx' > eps then newton f eps next
                                           else newton f eps (x+eps)
      where fx  = evalExp f x
            fx' = evalExp' f x
            next = x - (fx/fx')

roots :: Exp -> [R] -> [R]
roots f as = map (newton f 0.001) as

ghci> roots (detNN m) [0, 0.5 .. 2]
[0.4980392156862745,0.5,1.0,1.0019607843137255,1.000762380097245]
```

Utifrån de beräknade egenvärdena kan egenvektorer för respektive egenvärde beräknas fram, se avsnitt 2.3.8. En implementation för att applicera givet egenvärde på matrisen $(A - \lambda I)$ skapades och egenvektorerna hittas genom att lösa det homogena ekvationssystemet för varje applicerat egenvärde.§§ Det vill säga matrisernas nollrum.

```
eigenvalues = [0.5, 1]

eigen05 = evalMat m 0.5
eigen1  = evalMat m 1

ghci> eigen05              ghci> eigen1
| 0.25  0.25 |             | -0.25   0.25 |
| 0.25  0.25 |             |  0.25  -0.25 |

ghci> nullSpace eigen05    ghci> nullSpace eigen1
Sub [                      Sub [
|  1.0 |                   | 1.0 |
| -1.0 |                   | 1.0 |
]                          ]
```

4.3 Test och verifikation

En del av detta projekt var att bevisa korrektheten av de implementerade egenskaperna inom linjär algebra. De definierade typerna är beroende typer. Detta ger att typsystemet fungerar som en första nivån av kontroll som eliminerar risken för inkorrekta dimensioner på operationer på vektorer och matriser.

I denna del redovisas resultaten av dessa tester och vilka steg som togs för att säkerställa korrektheten och testa prestanda.

4.3.1 Enhetstester

Enhetstesterna användes för att testa de matematiska egenskaperna hos implementerade operationer och funktioner. Detta gjordes genom QuickCheck, vilket behöver generatorer för att fungera, se avsnitt 2.6.2. Generatorer för att konstruera vektorer och matriser av given storlek med slumpmässiga element implementerades på följande sätt:

```
instance forall n f. (KnownNat n, Arbitrary f) => Arbitrary (Vector f n) where
    arbitrary = V <$> vector ( vecLen (undefined :: Vector () n) )

instance forall m n f. (KnownNat m, KnownNat n, Arbitrary f) =>
                        Arbitrary (Matrix f m n) where
    arbitrary = do
        let v = arbitrary @(Vector f m)
        M . V <$> vectorOf (vecLen (undefined :: Vector () n)) v
```

Därefter skapades tester som använder dessa generatorer för att testa vissa lagar. Exempelvis skapades tester för att bevisa identitetslagen, den associativa lagen samt den kommutativa lagen för vektoraddition:

```
-- | Tests vector addition properties for a given vector lenght

prop_vectorAddZero :: (KnownNat n, AddGroup f, Eq f) => Vector f n -> Bool
prop_vectorAddZero v = v + zero == v

prop_vectorAddComm :: (KnownNat n, AddGroup f, Eq f) =>
                      Vector f n -> Vector f n -> Bool
prop_vectorAddComm v1 v2 = v1 + v2 == v2 + v1

prop_vectorAddAssoc :: (KnownNat n, AddGroup f, Eq f) => Vector f n ->
                       Vector f n -> Vector f n -> Bool
prop_vectorAddAssoc v1 v2 v3 = (v1 + v2) + v3 == v1 + (v2 + v3)
```

Testningen gav ett flertal intressanta resultat. Till exempel misslyckas testet för associativitet ifall typen på värdena i vektorn var `Double`, där `f` avser den underliggande typen. Detta sker eftersom Haskell använder sig av icke exakta flyttal som avrundar talen. Därför är additionerna inte lika och testet ger falskt som resultat. Det här är ett bra argument för att använda Agda, då Agda kan bevisa dessa lagar utan behovet att representera denna icke exakta datatyp.

Ytterligare testades homomorfi mellan två multiplikativa strukturer i QuickCheck. Determinanten är en homomorfi mellan två sådana strukturer, se teoriavsnitt 2.2.2. Determinanten bevarar strukturen vid avbildning mellan multiplikation av matriser och multiplikation mellan element i den underliggande typen `f`.

```
prop_detHomomorphism :: (KnownNat n, Field f, Eq f) =>
                        Matrix f n n -> Matrix f n n -> Bool
prop_detHomomorphism m1 m2 =
  detNN (m1 £££ m2) == detNN(m1) * detNN(m2)
```

Alla tester som uppvisats i denna sektion passerar för `Rational` och för en fullständig lista på enhetstester se GitHub [EJN+22b].

4.3.2 Agda

Som komplement till de konstruerade testerna, implementerades även en del av HasLin i Agda för att bevisa korrektheten av implementationen. Som beskrivet i avsnittet 2.7, vid kompilering av kod utformas ett bevis för den implementerade koden.

Till en början implementerades vektorer och matriser i Agda som ungefärligen motsvarar implementationen i Haskell tillsammans med dess språktillägg, se avsnitt 2.6.1.

```
data Vector (A : Set a) : (n : ℕ) → Set a where
    [] : Vector A zero
    _::_ :  A → Vector A n → Vector A (suc n)

Matrix : (A : Set a) → (m n : ℕ) → Set a
Matrix A m n = Vector (Vector A m) n
```

Vektorn är ett typiskt exempel på en beroende datatyp; parametriserad av en typ A och indexerad av de naturliga talen. Vektorn har i sin tur en fast längd som är bunden till värdet av detta index. Likaväl går det att definiera matriser som vektorer av kolumnvektorer.

Nästa steg var att implementera operationer över datatypen för vektorer. I detta syfte användes en ring från Agdas standardbibliotek för att definiera algebraiska egenskaper hos vektorer med dess binära operationer.

Här ett kort utdrag över några implementerade operationer:

```
-- Vector addition
_+v_ : (v1 v2 : Vector Carrier n) → Vector Carrier n
_+v_ = zipV _+_

-- Scale vector
_◁_ : Carrier → Vector Carrier n → Vector Carrier n
c ◁ v = mapV (c *_) v

-- Dot product
_•_ : (v1 v2 : Vector Carrier n) → Carrier
v1 • v2 = sumV (zipV _*_ v1 v2)

-- Cross product
_×_ : (u v : Vector Carrier 3) → Vector Carrier 3
(v1 :: v2 :: v3 :: []) × (u1 :: u2 :: u3 :: []) =
                                    (v2 * u3 + -(v3 * u2) ::
                                     v3 * u1 + -(v1 * u3) ::
                                     v1 * u2 + -(v2 * u1) :: [])
```

Implementationerna efterliknar vad som kodats i Haskell, se resultat 4.2.2. Det behövdes däremot implementeras typspecifika hjälpfunktioner `zipV`, `sumV` och `mapV`. Dessa funktioner efterliknar funktioner i Haskell som används på listor, men behövdes omdefinieras för att fungera med den implementerade vektortypen. Då dessa vektorer är beroende datatyper.

Fortsättningsvis kan egenskaper av dessa operationer bevisas med hjälp av Curry-Howard korrespondens i Agdas typsystem. Till detta syfte implementerades en typ för ekvivalens mellan vektorer.

```
_=v_ : Vector Carrier n → Vector Carrier m → Set (c ⊔ ℓ)
_=v_ = EqV _≈_
```

Likhet mellan vektorer '=v' är en upphöjd version av den underliggande likheten i ringens komponenter från standardbiblioteket. Likhetstypen är i princip en vektor av likhetsbevis mellan par av motsvarande element.

På liknande sätt som egenskaperna för addition mellan vektorer testades i Haskell, se avsnitt 4.3.1, så bevisades kommutativitet och associativitet lagen med ring biblioteket i Agda.

```
-- Right proof of identity with vector addition

vectorAddIdentityʳ : ∀ {n} (v1 : Vector Carrier n) →
                              (v1 +v 0v) =v v1
vectorAddIdentityʳ [] = eq-[]
vectorAddIdentityʳ (x1 :: v1) =
    eq-:: (+-identityʳ x1) (vectorAddIdentityʳ v1)

-- Vector addition is commutative (statement, and inductive proof)
vectorAddComm : ∀ {n} (v1 v2 : Vector Carrier n) →
                        (v1 +v v2) =v (v2 +v v1)
vectorAddComm [] [] = eq-[]
vectorAddComm (x1 :: v1) (x2 :: v2) =
    eq-:: (+-comm x1 x2) (vectorAddComm v1 v2)

-- Vector addition is associative (statement, and inductive proof)
vectorAddAssoc : ∀ {n} (v1 v2 v3 : Vector Carrier n) →
                         ((v1 +v v2) +v v3) =v (v1 +v (v2 +v v3))
vectorAddAssoc [] [] [] = eq-[]
vectorAddAssoc (x1 :: v1) (x2 :: v2) (x3 :: v3) =
    eq-:: (+-assoc x1 x2 x3) (vectorAddAssoc v1 v2 v3)
```

Även Quad implementerades i Agda. Då Haskell implementationen indexerades med den induktiva typen `Nat4` kunde Agda implementationen, för såväl datatypen som dess relaterade funktioner, göras snarlik.

```
data Nat4 : Set where
  One : Nat4
  Suc : Nat4 -> Nat4

data Quad ( A : Set a ) : Nat4 → Set a where
  Zero : Quad A n
  Scalar : A → Quad A One
  Mtx : ( nw ne sw se : Quad A n ) → Quad A (Suc n)
```

```
_+q_ : ( x y : Quad Carrier n) → Quad Carrier n
Zero           +q y              = y
Scalar x       +q Zero           = Scalar x
Scalar x       +q Scalar y       = Scalar (x + y)
Mtx x x1 x2 x3 +q Zero           = Mtx x x1 x2 x3
Mtx x x1 x2 x3 +q Mtx y y1 y2 y3 = Mtx (x  +q y)  (x1 +q y1)
                                       (x2 +q y2) (x3 +q y3)
```

På liknandesätt som för vektorerna definierades en ekvivalenstyp för Quad, _=q_, som baseras på en underliggande ring. Med denna bevisades identitet och kommutativitet av _+q_:

```
-- Zero is the left identity for addition on quad
quad-zerol : ∀ { n } ( x : Quad Carrier n ) → ( Zero +q x ) =q x
quad-zerol x = quad-refl

-- Zero is the right identity for addition on quad
quad-zeror : ∀ { n } ( x : Quad Carrier n ) → ( x +q Zero ) =q x
quad-zeror Zero = quad-refl
quad-zeror (Scalar x) = quad-refl
quad-zeror (Mtx x x1 x2 x3) = quad-refl

-- proof that addition is commutative for quad matrices
quad-comm : ( a b : Quad Carrier n ) → ( a +q b) =q ( b +q a )
quad-comm Zero b = quad-sym (quad-zeror b)
quad-comm (Scalar x) Zero = quad-refl
quad-comm (Scalar x) (Scalar y) = eq-Scalar (+-comm x y)
quad-comm (Mtx a a1 a2 a3) Zero = quad-refl
quad-comm (Mtx a a1 a2 a3) (Mtx b b1 b2 b3) =
                eq-Mtx (quad-comm a  b ) (quad-comm a1 b1)
                       (quad-comm a2 b2) (quad-comm a3 b3)
```

I bevisen syftar quad-refl och quad-sym till två elementära egenskaperna av ekvivalens, att $\forall x.x = x$ respektive att $\forall x, y.x = y \Rightarrow y = x$.

4.3.3 Prestandatest

Ett par enkla prestandatest utfördes för de tre olika matrisrepresentationerna, LIL (listor i listor), CSR och QuadM. Två olika exempelmatriser av storlek 500×500 användes för testning, en gles matris och en tät matris. Den valda glesa matrisen för testerna var identitetsmatrisen, medan den täta matrisen hade ett nollskilt element i varje position.

Testet skedde med hjälp av det inbyggda GHCi kommandot :set +s vilket räknar tiden till avslut för alla körda kommandon och funktioner [GHC22b]. Därför skapades också två test funktioner för addition och multiplikation som kunde köras med testmatriserna:

```
performAddTest :: (AddGroup a, Eq a) => a -> Bool
performAddTest m = a == a
    where a = m + m

performMulTest :: (Mul a, Eq a) => a -> Bool
performMulTest m = a == a
    where a = m * m
```

Testerna innefattade både addition och multiplikation, där matriserna adderades eller multiplicerades med sig själv. Testet kör även ett ekvivalenstest för att tvinga GHCi att evaluera operationerna samtidigt som det undviker utskrivning av matriserna. Det krävs då tiden det tar för GHCi att skriva ut matriserna räknas med i den inbyggda tidtagaren.

Matristyp	Operation	Representation	Tid (s)
Gles Matris	Addition	LIL	0,02
		CSR	0,02
		QuadM	0,03
	Multiplikation	LIL	5,9
		CSR	9,7
		QuadM	0,03
Tät Matris	Addition	LIL	16
		CSR	5,2
		QuadM	13
	Multiplikation	LIL	27
		CSR	600
		QuadM	140

Tabell 4.1: Prestandatest

Testet, se tabell 4.1, visar att för multiplikation med glesa matriser är QuadM nästan 200 gånger snabbare än LIL. För addition med glesa matriser är alla representationer likvärdiga. För täta matriser dock ger CSR den bästa prestandan för addition medan LIL är den bästa för multiplikation.

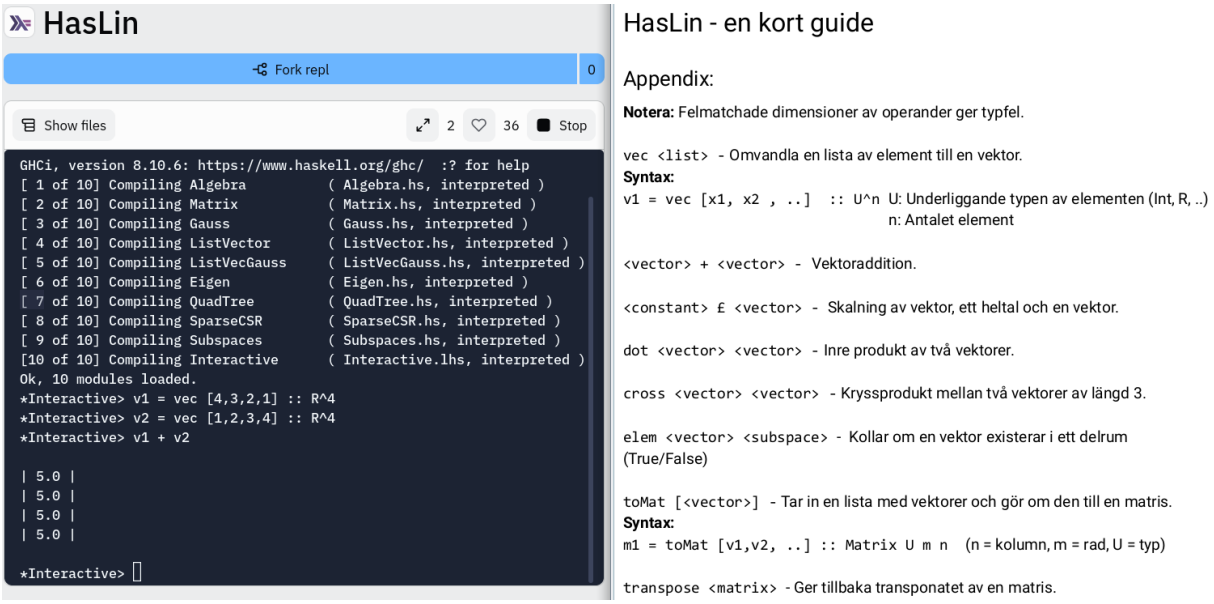
Notera att konstruktionen av täta matriser är betydligt mer krävande än glesa och att konstruktionstiden inkluderas i den totala tiden för operationerna. Detta resulterar i att en tät LIL matris tar längre tid att utföra operationerna jämfört med en gles LIL matris, trots att operationerna bör ta lika lång tid. På grund av detta borde resultaten inom en täthet endast jämföras med varandra.

4.4 Interaktiv upplevelse

Ett kompletterande material framtogs för att gagna användarupplevelsen. Materialet är i form av en källkodsfil, kallad `Interactive.lhs`, där det importerats ett urval av essentiell funktionalitet från HasLin. För att förenkla åtkomsten och förhoppningsvis bredda målgruppen av användare laddades GitHub-repot [EJN+22a] upp till en webbaserad ut-

vecklingsmiljö kallad *Replit*, finns här `https://replit.com/@Sebastiansjogrn/HasLin`. Replit möjliggör att använda HasLin utan att ha vare sig GHCi eller någon utvecklingsmiljö installerat lokalt. På Replit körs den sammanställda källkodsfilen och all funktionalitet kan utföras från webbterminalen. Valet finns även för den som vill att kompilera och köra kod från någon av HasLins moduler.

Dokumentation för HasLin befinner sig i den sammanställda filen `Interactive.lhs`. Utöver det finns även en sammandragen lathund med exempeluppgifter och dokumentation i PDF-format bifogat i Replit. Lathunden innehåller exempel på hur olika typer av vektorer och matriser kan skapas, samt en lista över tillgängliga funktioner som går att utföra. I figur 4.1 syns hur användandet kan se ut av att köra `Interactive.lhs` via Replit och ta hjälp av lathunden.



Figur 4.1: Skärmdump av interaktiv upplevelse med lathund

4.4.1 Användartest och utvärdering

Vid slutskedet av arbetet testade några frivilliga försökspersoner den interaktiva upplevelsen. Personerna fick tillgång till HasLins Replit-sida och den bifogade lathunden. Försökspersonerna fick anonymt lämna kommentarer om deras upplevelse. Syftet med detta var för att försöka få en generell uppfattning av vad utomstående tyckte om användbarheten av HasLin.

Responsen var övervägande bra och syntaxen tycktes vara begriplig. En av deltagarna jämförde HasLin med *WolframAlpha* när det gäller funktionalitet och tyckte det var smidigt med att ha ett terminalbaserat verktyg för matris- och vektorberäkningar. Kommentarerna innehöll även konstruktiv återkoppling om vad som skulle kunna förbättras i HasLin som exempelvis ett hjälpkommando som listar ut alla tillgängliga funktioner. Därutöver ansågs det vara krångligt att behöva specificera storleken av vektorer och matriser.

5

Diskussion

Nedan följer en diskussion kring resultaten som uppnåtts i projektet. Resultatdiskussionen omfattar resultatens relevans till projektets syfte, ifall resultaten omfattar allt som var utsatt att göras eller ifall några oväntade resultat uppnåtts. Konsekvenser av resultaten diskuteras samt hur teoretiska och metodtekniska val kunde gjorts annorlunda. Vidare uppmärksammas framtida utvecklingsområden för arbetet och hur det kan vara en byggsten i framtida studier. Därefter diskuteras etiska aspekter kopplade till arbetet och sammanfattningsvis presenteras en slutsats för hela projektet.

5.1 Resultatdiskussion

Projektet tog delvis en annan riktning än vad som från början var planerat. Noterbart har fler resurser lagts på projektmål 2, matematiken i funktionell programmering, vilket främst har gått ut över projektmål 3, testning och bevis av korrekthet. Resultatets koppling till arbetets syfte blev därav inte lika starkt som det var önskat. En detaljerad kritisk analys av arbetsgången och resultatens validitet presenteras i denna sektion.

5.1.1 Resultatanalys

Som nämnt i metoden studerades datavetenskapliga källor som komplement till den matematiska litteraturen. Källorna bestod i huvudsak av vetenskapliga publikationer men i somliga fall har alternativa källor använts, främst i form av webbsidor. Detta kan kritiseras då projektet i viss mån inte bygger på referentgranskade texter och validiteten av faktainsamlingen kan därav ifrågasättas. Med detta sagt bygger dock majoriteten av litteraturanalysen på publicerade böcker och vetenskapliga artiklar.

Stöd för alla matematiska områden som ingick i litteraturstudien har implementerats i HasLin, vilket visas i delsektionerna av matematiken i funktionell programmering, se resultat 4.2. Detta har resulterat i ett språk med mycket funktionalitet i flera moduler, där mycket arbete även lagts på att göra språket enhetligt mellan modulerna. Enhetligheten gynnar användaren då olika typer av beräkningar kan kombineras i samma interaktiva miljö och tar bort behovet av att behöva orientera sig bland HasLins moduler. Detta visas i resultatet för den interaktiva upplevelsen, se resultat 4.4.

Utvecklingen av de matematiska modulerna i kombination med interaktionen mellan dem har resulterat i en större kodbas som krävt mer resurser än vad som ursprungligen var

förväntat. Detta har lett till negativa konsekvenser för somliga av projektets andra delar, huvudsakligen för test och verifikation, se resultat 4.3. Testning har enbart gjorts på en mindre mängd av basala matematiska lagar för vektorer och matriser. Verifikation i Agda har också skett i mindre utsträckning än planerat. HasLins korrekthet kan därför inte garanteras i den mån det var ämnat i syftet. Med detta sagt har HasLin utvecklats i samspel med Haskells typsystem för att matcha språkets typer med domänens definitioner, främst genom användandet av `Nats`, se resultat 4.2. Säkerheten hos språket kan därför i viss mån försäkras via typkontroll.

Det mest oväntade resultatet av projektet, och något som bidrog till det ökade resurskravet, var implementationen av olika matrisrepresentationer, se resultat 4.2.3. Både CSR och Quad skapades för att effektivt hantera glesa matriser, vilket öppnar upp för enklare användning av stora matriser, se tabell 4.1. För att binda samman de nya matriserna till HasLin skapades en gemensam matrisklass. Användare kan använda sig av matrisklassen för att skapa egna matrisrepresentationer och använda dem med färdigdefinierade funktioner såsom `gauss` och `transpose`. Denna funktionalitet, trots oväntad, ses som ett av resultatets höjdpunkter för användarvänligheten hos HasLin.

5.1.2 Resultatets konsekvenser

Med HasLins brist av test och verifikation, får detta konsekvenser hur språket kan och bör användas. HasLin bör i huvudsak inte användas för bevisande av matematiska satser. Utan istället användas i vetskap om att korrektheten av matematiken inte är fullt garanterad. I detta stadie av utvecklingen, rekommenderas därför inte HasLin att användas i någon officiell kapacitet på lärosäten eller företag.

Med tanke på att majoriteten av arbetets fokus gick åt att implementera ny funktionalitet, så har HasLin fått en förhållandevis bred verktygslåda för att utföra matematiska beräkningar av olika slag. I grund och botten är HasLin ett verktyg för linjär algebra men överlappar även med andra matematiska discipliner så som matematisk analys. Under fördjupandet kring egenvärden och egenvektor uppkom ett behov att utöka språket för att kunna representera matematiska uttryck. Till de matematiska uttrycken utvecklades även relaterade operationer för grundläggande abstrakt algebra och derivering, se i resultaten 4.2.1 och 4.2.5. Detta i kombination med HasLins resterande funktionaliteten för linjär algebra resulterar i en ganska kapabel matematiktolk.

5.1.3 Teoretiska och metodtekniska insikter

Vad som tydligt framkommer i sektionerna ovan var att för mycket tid spenderats på att utveckla HasLins funktionalitet snarare än att testa och verifiera det, vilket sannolikt är en konsekvens av den metodik som använts under projekts gång. Med den metodik som använts har utveckling av funktionalitet skett frånskilt testning och verifiering. Denna metod har visat sig ineffektiv då alldeles för stort fokus hamnade på funktionalitetsutveckling vilket resulterade i att alldeles för lite tid lämnades kvar till att bevisa korrektheten.

Ett alternativt val av metod, som sannolikt skulle jämnat ut tidfördelningen, vore att arbeta utefter konceptet av testdriven utveckling. Testdriven utveckling syftar till att tester konstrueras innan själva funktionaliteten är utvecklad. Därmed testas funktionali-

teten kontinuerligt under hela utvecklingsprocessen. En alternativ version av testdriven utveckling hade varit att för varje ny implementerad funktionalitet, ses den till att testas noggrant innan nytt utvecklas. Denna typ av metodik hade garanterat en jämnare tidfördelning och därmed tjänat syftet med projektet bättre.

Ett annat metodteknist val, som även det sannolikt hade hjälpt förbättra resultats relevans till syftet, hade varit att minska arbetsresurserna lagda för att få sammanhängande moduler. Vad som även här kan inkluderas vore att ignorera implementation av flera matrisrepresentationer. Trots att samhörigheten bland modulerna samt de olika matrisrepresentationerna är några av resultatets höjdpunkter, har de krävt mycket resurser att genomföra. Resurser som annars kunde spenderats på mer test och verifikation.

Insikter kring teoretiska val har även uppkommit. Som nämnts tidigare i diskussionen har storleken av litteraturstudien identifierats som en av orsakerna till den ojämna resultats fördelningen. En snävare avgränsad litteraturstudie skulle minskat det teoretiska underlaget för arbetet. Därmed göra processen att implementera matematiken kortare och möjliggöra för mer tid åt test och verifikation av HasLins korrekthet.

5.2 Framtida utvecklingsområden

Då projektet avslutats och HasLin granskas i sitt nuvarande stadie noteras att språket fortfarande har intressanta utvecklingsmöjligheter. Denna sektion syftar därför till att lyfta några möjliga utvecklingsområden. För den intresserade läsaren kan denna sektion även ses som en uppmaning till att bidra till språket, som är öppet publicerat på GitHub [EJN+22a].

Övergripande i diskussionen har bristen av test och verifikation noterats. Ett givet utvecklingsområde är därför att specificera och visa att funktionaliteten i HasLin följer de domänspecifika lagar som innefattats av litteraturstudien. Detta syftar både till utformandet av flera enhetstest och till att överföra flera delar av den implementerade koden till Agda, varpå den kan verifieras.

Ett annat framtida utvecklingsområde är vidare utveckling av matrisklassen och stöd för generella operationer. Det för att bredda användningsområdet, speciellt med hänsyn till glesa matriser. Matrisklassens utveckling skulle även främja den interaktiva upplevelsen genom en ännu mer enhetlig syntax.

Utvecklingsmöjligheterna diskuterade i denna sektion skulle bredda mängden av möjliga tillämpningsområden för HasLin. Något som förhoppningsvis skulle leda till en större målgrupp av användare och möjliggöra för HasLin att användas i en mer officiell kapacitet. Med bredare målgrupp ses god möjlighet till en tillväxt av medverkande utvecklare av HasLin, tack vare naturen av öppen källkod. Vilket fortsätter att möjliggöra för framtida utveckling långt efter projektets avslut.

5.3 Etiska konsekvenser

Övergripande kan det anses svårt att identifiera några betydelsefulla sociala eller etiska konsekvenser som kan orsakas av arbetet i detta projekt. Däremot har ett par sådana identifierats, huvudsakligen kan de orsakas av ett felanvändande av språket.

HasLin är tänkt att kunna användas som ett komplement för personer som vill lära sig om linjär algebra eller datavetenskap. Användare ska kunna se kopplingen mellan matematiken och datastrukturer samt använda programmet för att utföra beräkningar, vilket förhoppningsvis hjälper dem att få en djupare förståelse av linjär algebra och dess användningsområden. Däremot skulle användandet kunna ha negativ inverkan på lärandet, då programmet kan användas för att enbart få lösningar till problem utan att lära sig metodiken. I en situation där språket används som komplement till en kurs i linjär algebra av studenter på ett lärosäte, exempelvis Chalmers, kan det därför ha både positiva och negativa konsekvenser på studenternas lärande. I så fall kan språket användas kontraproduktivt mot vad dess syfte är menat att vara, vilket speciellt i utbildningssammanhang kan få samhällsetiska konsekvenser på grund av bristande kunskap hos elever.

Vidare diskussion kring HasLins användning på lärosäten. Personer med förkunskaper av datavetenskap, främst programmering, kan ha enklare att komma i gång och förstå sig på hur HasLin kan användas. Att använda språket på lärosäten kan därför anses vara orättvist då det kan ge en betydande fördel till elever med datavetenskaplig bakgrund. Speciellt om HasLin används i en mer officiell kapacitet som ett hjälpmedel integrerat i läroplanen bör det ses över för vilka program och klasser detta görs. På somliga program, som datateknik och informationsteknik, bör samtliga elever ha tillräckliga förkunskaper. Medan för andra program eller klasser är detta inte nödvändigtvis lika självklart, därför bör HasLin användas eftertänksamt för att inte orsaka orättvisa.

5.4 Slutsats

Projektets syfte var att skapa ett domänspecifikt språk för linjär algebra och undersöka hur domänen kan pedagogiskt förmedlas samt hur språkets korrekthet kan bevisas. Undersökningen ämnade att stärka framtida användares kunskaper för linjär algebra och datavetenskap.

Resultatet från litteraturstudien och den matematik som programmerats i språket tyder på att en god mängd funktionalitet lyckats modelleras. Noterbar styrka är samhörigheten hos språkets funktionalitet, vilket möjliggör för den interaktiva upplevelsen att bli enhetlig till skillnad från en splittrad samling av individuella moduler.

Testning och verifikation gjordes på utvalda basala matematiska lagar men domänens korrekthet kan inte garanteras i den mån det var önskat. Det blev en effekt av att litteraturstudien hade behövt en snävare avgränsning, gällande vilka delar av domänen som skulle modelleras, samt det val av utvecklingsmetod, som visade sig vara ineffektiv.

Ett domänspecifikt språk som pedagogiskt och enhetligt kan förmedla mycket av den linjära algebrans essens lyckades skapas. Arbetet har publicerats som öppen källkod i

förhoppning av att det kan gagna framtida användares förståelse för domänen och datavetenskapen. Då språkets korrekthet däremot inte kan garanteras i större mån, bör arbetet inte användas för bevisande av matematiska satser. Följaktligen behövs ytterligare utveckling, huvudsakligen i form av test och verifikation för att bevisa HasLins korrekthet och kunna stärka framtida användares kunskaper ännu mer.

Litteraturförteckning

[3Bl16] 3Blue1Brown. Eigenvectors and eigenvalues | chapter 14, essence of linear algebra, 2016. Hämtad: 2022/04/07. URL: https://www.youtube.com/watch?v=PFDu9oVAE-g&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&index=14&ab_channel=3Blue1Brown.

[Abe16] Andreas Abel. Natural deduction and the curry-howard-isomorphism, 2016. URL: https://www.cse.chalmers.se/~abela/esslli2016/lecture1-nd.pdf.

[Ada07] Iain T. Adamson. *Introduction to Field Theory*. Dover Publications, 2007.

[Axl95] Sheldon Axler. *Linear Algebra Done Right*. Springer, 1995.

[BD09] Ana Bove and Peter Dybjer. Dependent types at work. *Language Engineering and Rigorous Software Development*, page 57–99, 2009. doi:10.1007/978-3-642-03153-3_2.

[Cla20] Koen Claessen. Quickcheck: Automatic testing of haskell programs, 2020. Hämtad: 2022/02/09. URL: https://hackage.haskell.org/package/QuickCheck.

[Dev22] Agda Developers. The agda standard library, 2022. Hämtad: 2022/04/20. URL: https://github.com/agda/agda-stdlib.

[EGSS82] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package, 1982. Hämtad: 2022/04/26. URL: https://cpsc.yale.edu/sites/default/files/files/tr112.pdf.

[EJN+22a] Adam Eliasson, Patrik Jansson, Daniel Nikoalev, Filip Nordmark, Sebastian Sjögren, and Linus Sundkvist. Bscproj2022, 2022. Hämtad: 2022/04/16. URL: https://github.com/DSLsofMath/BScProj2022.

[EJN+22b] Adam Eliasson, Patrik Jansson, Daniel Nikoalev, Filip Nordmark, Sebastian Sjögren, and Linus Sundkvist. Bscproj2022, 2022. Hämtad: 2022/04/28. URL: https://github.com/DSLsofMath/BScProj2022/blob/master/src/tests/Test.hs.

[Fow19] Martin Fowler. Domain-specific languages guide, 2019. Hämtad: 2022/04/26. URL: https://martinfowler.com/dsl.html.

[Fys22] Fyskikteknologsektionen. Course statistics at chalmers, 2022. Hämtad: 2022/02/08. URL: https://stats.ftek.se/.

[GHC22a] GHC. GHC.TypeLits, 2022. Hämtad: 2022/04/13. URL: https://hackage.haskell.org/package/base-4.16.1.0/docs/GHC-TypeLits.html.

[GHC22b] GHC Team. *GHC User's Guide Documentation – Release 9.2.2*, 2022. URL: https://downloads.haskell.org/~ghc/9.2.2/docs/users_guide.pdf.

[GW14] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, United States, August 2014. Association for Computing Machinery (ACM). doi:10.1145/2628136.2628138.

[GYK22] Ariel Gershon, Edwin Yung, and Jimin Khim. Newton raphson method, 2022. Hämtad: 2022/04/26. URL: https://brilliant.org/wiki/newton-raphson-method/.

[Has22a] Haskell.org. The glasgow haskell compiler, 2022. Hämtad: 2022/04/11. URL: https://www.haskell.org/ghc/.

[Has22b] Haskell.org. Haskell, an advanced, purely functional programming language, 2022. Hämtad: 2022/02/09. URL: https://www.haskell.org/.

[htm22] html.com. Html for beginners the easy way: Start learning html & css today, 2022. Hämtad: 2022/04/11. URL: https://html.com/.

[JIB22a] Patrik Jansson, Cezar Ionescu, and Jean-Philippe Bernardy. *Domain-Specific Languages of Mathematics*, volume 24 of *Texts in Computing*. College Publications, January 2022.

[JIB22b] Patrik Jansson, Cezar Ionescu, and Jean-Philippe Bernardy. Domain-specific languages of mathematics, 2022. Hämtad: 2022/02/11. URL: https://github.com/DSLsofMath/DSLsofMath.

[Kme21] Edward A. Kmett. linear : Linear algebra, 2021. Hämtad: 2022/04/18. URL: https://hackage.haskell.org/package/linear.

[LaT22] LaTeX Team. Latex – a document preparation system, 2022. Hämtad: 2022/04/11. URL: https://www.latex-project.org/.

[Mat22] MathWorks. Sparse matrices, 2022. Hämtad: 2022/05/10. URL: https://www.mathworks.com/help/matlab/sparse-matrices.html.

[Met20] Sara A. Metwalli. 5 applications of linear algebra in data science, 2020. Hämtad: 2022/04/11. URL: https://towardsdatascience.com/5-applications-of-linear-algebra-in-data-science-81dfc5eb9d4.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. *Logic Colloquium '73, Proceedings of the Logic Colloquium*, page 73–118, 1975. doi:10.1016/s0049-237x(08)71945-1.

[Nee07] Michael J. Neely. General vector spaces over a field, 2007. Hämtad: 2022/04/10. URL: https://viterbi-web.usc.edu/~mjneely/pdf_papers/vector-space.pdf.

[Net21] Netlib. Blas (basic linear algebra subprograms), 2021. Hämtad: 2022/04/20. URL: http://www.netlib.org/blas/.

[Net22] Netlib. Lapack (linear algebra package), 2022. Hämtad: 2022/04/20. URL: http://www.netlib.org/lapack/.

[Pra21] Vaughan Pratt. Algebra. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2021. Hämtad: 2022/04/02. URL: https://plato.stanford.edu/archives/spr2021/entries/algebra/.

[Rui21] Alberto Ruiz. hmatrix: Numeric linear algebra, 2021. Hämtad: 2022/04/18. URL: https://hackage.haskell.org/package/hmatrix.

[Ryb22] Simon Rybrand. Linjära ekvationssystem, vad är det?, 2022. Hämtad: 2022/04/04. URL: https://eddler.se/lektioner/linjara-ekvationssystem/.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. Technical report, Computer Sdence Department, University of Maryland, 1984. Hämtad: 2022/04/11. URL: http://www.cs.umd.edu/~hjs/pubs/SameCSUR84-ocr.pdf.

[Sed16] Lincoln Sedlacek. Math education: The roots of computer science, 2016. Hämtad: 2022/04/11. URL: https://www.edutopia.org/blog/math-education-roots-computer-science-lincoln-sedlacek.

[Spa97] Gunnar Sparr. *Linjär Algebra*. Studentlitteratur, 1997.

[SS22] Sebastian Sjögren and Linus Sundkvist. Haslin, 2022. Hämtad: 2022/05/11. URL: https://replit.com/@Sebastiansjogrn/HasLin.

[The22] The Agda Wiki. Agda, 2022. Hämtad: 2022/02/09. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[Wei22] Eric W Weisstein. Determinant, 2022. Hämtad: 2022/05/08. URL: https:

//mathworld.wolfram.com/Determinant.html.

[Wes21] Western Governors University. How much math does computer science require?, 2021. Hämtad: 2022/04/11. URL: https://www.wgu.edu/blog/how-much-math-computer-science-require2110.html#openSubscriberModal.