UNIVERSITY OF
CAMBRIDGE

Formalizing Clifford algebras and related constructions in the Lean theorem prover

Eric Francis Wieser

Gonville & Caius

Supervisor: Professor Joan Lasenby

February 2024

This dissertation is submitted for the degree of
Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text.

It is not substantially the same as any work that has already been submitted before for any degree or other qualification except as declared in the preface and specified in the text.

It does not exceed the prescribed word limit for the Engineering Degree Committee.

Eric Francis Wieser
February 2024

Abstract

Geometric Algebra (GA) is a mathematical framework used primarily by engineers and physicists, while theorem proving software has its background originally in computer science departments. This thesis brings the former into the language of the latter, in the more abstract setting of Clifford algebras. It does so via the theorem proving language "Lean", which is seeing increasing adoption in mathematics departments.

The focus is much broader than simply formalizing Clifford algebras; Lean has an expansive and monolithic library of formalized mathematics, mathlib, which the author made Clifford algebras a part of. Over the course of this work, mathlib was reshaped and extended in various ways.

Part I of this thesis provides an introduction to geometric algebra, Clifford algebra, and an exploration of the author's path from numerical software through symbolic tools to the Lean theorem prover.

Part II describes the various ways in which mathlib's algebra libraries were shaped by the author, covering: the use of typeclasses to manage automatic inference of scalar actions, the use of the extensionality "tactic" to provide leverage when proving results about complicated algebraic objects, a novel formalization of graded monoids, rings and algebras, and a deeper dive into intricate traps around typeclasses that originate in the language itself.

Part III begins by illuminating informally how to build a library of results around Clifford algebras using the universal property alone, then uses this approach to constructively build an expansive set of well-known isomorphisms in a basis-free manner; contributing many formalizations relating to other algebraic topics to mathlib along the way. These topics include dual quaternions, tensor products, quadratic forms, alternating maps, and exponential operators.

As well as summarizing the author's key contributions and possible avenues for future development, the conclusions outline how other users of Lean are already building upon parts of the author's work.

Acknowledgements

While this thesis represents the culmination of my journey as a PhD student, the path that led here is not one I could have taken alone.

I am very grateful to my supervisor Joan Lasenby for her assistance throughout my time as a PhD student; Joan always made time to meet to discuss ideas, suggest direction, and reply to emails at unexpected hours of the night!

For introducing me to geometric algebra, I thank: my fellow PhD candidate Hugo Hadfield, who planted the idea both in my mind and Joan's of me joining the two of them in Cambridge to pursue research in that area, and introduced me to many relevant people in the field; Leo Dorst, whose *Geometric Algebra for Computer Science* was essential reading on the commute in the months up to beginning the PhD, and by coincidence whose website of hand-drawn Lego ideas provided me engineering inspiration more than a decade earlier; Steven De Keninck, whose `ganja.js` tool proved invaluable for building early intuition; and Petr Vašík, who provided repeated opportunities to share ideas in person with students and faculty alike studying geometric algebra in Brno.

For the inspiration to pivot towards formalization in Lean, I thank: Utensil Song, who broadening my horizons for geometric-algebra related software; Kevin Buzzard, whose "Natural Number Game" captured my imagination; the LftCM 2020 organizers Johan Commelin and Patrick Massot, for organizing an excellent virtual event that really cemented my path; the "CLUG" regulars in Cambridge, Ed Ayers, Yaël Dillies, and Bhavik Mehta, for keeping me on it; and the Lean community as a whole, for providing amazingly fast answers to any questions I had on Zulip. I thank the mathlib maintainers, for inviting me to join the team despite my engineering background. Thanks to Alexander Bentkamp and Marcus Zibrowius, who organized and invited me to LftCM 2023, I was fortunate enough to finally meet many of these community members in person.

I thank those who coauthored papers with me, Utensil Song and Jujian Zhang, the anonymous reviewers who helped improve them, and those who otherwise assisted in preparing those papers:

- for "Graded Rings in Lean's Dependent Type Theory": Kevin Buzzard for his justified insistence on needing an interface to talk about internal gradings, and Anne Baanen for picking up the mantle I dropped on mathlib's `set_like` refactor;
- for "Multiple-Inheritance Hazards in Dependently-Typed Algebraic Hierarchies": Gabriel Ebner, for campaigning for η-reduction support in Lean 4; Kazuhiko Sakaguchi, for providing insight into analogous situations in Coq; and Yaël Dillies and Filippo A. E. Nuccio, for

valuable feedback on the manuscript.

- for "Computing with the Universal Properties of the Clifford Algebra and the Even Subalgebra": David Cohoe, for illuminating some ideas that inspired the paper.

Much of the work in this thesis resulted in software contributions to Lean's mathlib; I am greatly thankful to those who volunteered their time to review them, a process that was both educational for me, and beneficial to mathlib as a whole. Ordered by the somewhat-arbitrary metric of the number of my contributions they reviewed, I thank: Johan Commelin, Anne Baanen, Scott Morrison, Oliver Nash, Yaël Dillies, Bryan Gin-ge Chen, Yury G. Kudryashov, Sébastien Gouëzel, Riccardo Brasca, Floris van Doorn, Rob Lewis, Jireh Loreaux, Rémy Degenne, Chris Hughes, Frédéric Dupuis, Kyle Miller, Gabriel Ebner, Bhavik Mehta, Junyan Xu, Alex J Best, Kevin Buzzard, Mario Carneiro, Patrick Massot, Eric Rodriguez, Damiano Testa, Anatole Dedecker, Heather Macbeth, Jeremy Tan Jie Rui, Yakov Pechersky, Joël Riou, Ruben Van de Velde, Moritz Doll, Joseph Myers; and 49 others who reviewed a small handful of contributions.

I thank those who made me feel a part of the Signal Processing lab at Cambridge, despite the increasingly diverging nature of my research: Hugo Hadfield, for reminding me that janky software can still be excellent; Alex Grafton, for his challenging pop-culture references and willingness to overlook shoe-based confusion; Shirley Liu, for arranging socials to bring the lab together; and everyone who made the period of virtual-only coffee breaks feel slightly less isolating.

I thank my friends: those pursing academic careers of their own who inspired me; those who helped me uproot my life in the US to return to the UK for this PhD; those who welcomed me back to the UK as if I had never left; and everyone who took the initiative to arrange getting together in person when I lacked the activation energy to do so myself.

I thank the Cambridge Trust for funding my research, and my new employer, Google DeepMind, for a generous part-time schedule that allowed time to complete it. I thank the people behind Zulip and GitHub, whose products opened effortless doors for collaboration that this work could not have happened without; those at Gitpod, who generously granted free quota on the cloud on which nearly all my development took place; and everyone who helped LaTeX suck a little less than the alternatives, through their packages or their advice on tex.stackexchange.com.

I thank my family, for every part they had to play on my path to this point. I thank my sister Penny, who I couldn't let be the only family member with a doctorate. I thank my wife Flo, for her love for and patience with me even in the periods when the direction of my PhD was unclear, her consideration for our wildly different working hours, and her ability to ply me with hot drinks and baked goods when I needed them most. Finally, I thank my parents, for their encouragement to pursue interest in math and computers at a young age, for every home-cooked dinner they offered during my time as a PhD student, and for everything else they have done for me through the years.

Contents

List of Tables, Figures, and Listings

1

Introduction

At its core, "geometric algebra" is an exercise in unification; its proponents remark that it combines and unifies the scalar (dot) and vector (cross) products in euclidean space, that it generalizes over the complex numbers and the quaternions, that it elegantly merges geometry with algebra, and even that it merges Maxwell's four equations into one [5, §7.1]. Pedagogically, this unification is a great asset; learning the mathematical framework of geometric algebra provides transferrable intuition to other areas of mathematics, while the connection between geometric and algebraic ideas enable this intuition to partially be gained visually.

From the author's perspective, the most compelling areas for applying geometric algebra are in computer graphics, computer vision, and robotics; all areas where software, which is inevitably algebraic in nature, is primarily manipulating geometric objects. Placing geometric algebra in the toolbox of software developers, engineers, and researchers in these areas, especially in toolboxes unequipped with quaternions, provides a valuable alternative in places where the typical fallback, matrices, are an ill-fitting tool; and indeed, there are a plethora of software libraries for geometric algebra, across various frameworks and languages.

To extend the metaphor, however, a single standard toolbox is not appropriate for everyone. A swiss army knife is rarely of use in a fully-equipped workshop; and in the same way, geometric algebra loses some appeal in the context of more abstract mathematics, where there are already more specialized tools (Lie groups, orthogonal groups, affine subspaces, …) within reach. To its credit, geometric algebra is not absent from this "workshop of abstract mathematics"; it can be regarded as a repackaged presentation of Clifford algebras, which are of independent (albeit somewhat niche) mathematical interest.

This thesis builds bridges between these two outlooks on the same ideas: linking the presentation of geometric algebra seen in software applications with the abstract mathematical concepts from which geometric algebra is distilled. The locations in which these bridges are placed is largely devoid of novelty, but they are constructed with an unusual set of materials: not the usual language of proofs in abstract mathematics, nor the numerical outputs of computational computer code, but a hybrid of the two; a computer language capable of providing formals proofs, the Lean theorem prover [6; 7].

Computer-formalized mathematics, while arriving at the bleeding edge of some mathematical fields [8; 9], is still a long way from catching up to conventional mathematics as a whole, and so a significant fraction of this thesis will be spent building the formal groundwork needed to bring Clifford algebras, and results about them, within reach. This effort is far from wasted; the groundwork described here now forms a small part of a much larger community-maintained formalization library (mathlib) used by almost all mathematical users of Lean.

1.1. Structure of this thesis

This thesis begins with some background material in part I, chapter 2 with a brief and fairly typical introduction to geometric algebra, followed by an even briefer introduction to the mathematical objects usually faced when working more abstractly with Clifford algebras. Chapter 3 explores a sampling of the challenges faced by the author when writing software for geometric algebra, and ultimately provides motivation for moving from symbolic software to proof assistants. It is here that the reader is provided an explanation of the Lean theorem prover and its mathematics library mathlib, and taught some basic syntax.

Part II explores a variety of design challenges that appear when formalizing the algebraic objects that will turn out to be prerequisites to formalizations in later chapters. The topics covered are scalar actions (chapter 4), extensionality (chapter 5), and graded rings (chapter 6); as well as a somewhat more foundational algebraic issue that posed a risk to mathlib as a whole (chapter 7).

In part III, the focus of the thesis narrows in on Clifford algebras, applying the tools developed in part II.

Chapter 8 presents a somewhat atypical way to think about geometric algebra, and the link between this outlook and some ideas from functional programming; it introduces the reader to the universal property of the Clifford algebra, and uses it to outline constructive generalizations of some known results. This chapter is mostly free of Lean code.

Having set up an excess of mathematical and formal preliminaries, chapter 9 summarizes some prior work on formalizing Clifford algebras in other proof systems, and presents the author's formalization of Clifford algebras in Lean, largely as it now appears in mathlib.

Chapter 10 follows on by demonstrating how universal properties can be used to link Clifford algebras with other algebraic constructions in mathlib; introducing universal properties for the dual numbers, complex numbers, and quaternions along the way. In addition to these concrete constructions, this chapter explores isomorphisms with more abstract constructions, heavily leaning on part II along the way.

It would have been unreasonable to expect to fully formalize every result about Clifford algebras; chapter 11 contains some formalizations that are in some way relevant to Clifford algebra, but for which further work would be needed to make the connection formally.

Chapter 12 summarizes the key contributions of this thesis, makes note of work by others that

builds upon the work here, and outlines avenues for further formalization.

1.2. Connections with published work

Chapter 9 and sections 3.4 and 10.1.2 are from the "Formalizing Geometric Algebra in Lean" paper [10] that set the direction of this thesis. Chapter 4 is a much longer version of "Scalar Actions in Lean's Mathlib" [11]. After writing this thesis, chapter 5 was extracted into "Chaining extensionality lemmas in Lean's Mathlib" [12]. Chapter 6 is largely the same as "Graded Rings in Lean's Dependent Type Theory" [2]. Chapter 7 is "Multiple-Inheritance Hazards in Dependently-Typed Algebraic Hierarchies" [3], with some errata corrected. Chapter 8 is "Computing with the Universal Properties of the Clifford Algebra and the Even Subalgebra" [13], which is itself the extended version of [4].

The vast majority of the code in this thesis, if not from the work listed here, is adapted from the author's contributions to mathlib.

Part I.

Motivation

2

Mathematical background

> One geometry cannot be more true than another; it can only be more convenient.
>
> *(Henri Poincaré)*

2.1. Geometric algebra

It is at this point that the reader, if unacquainted with geometric algebra, probably feels that they are overdue an explanation of what exactly geometric algebra *is*.

A typical introduction to linear algebra might start by talking about a vector in 3D euclidean space ($\mathbb{R}^3$) via its coordinates $v = [v_x, v_y, v_z]^\mathsf{T}$, or alternatively by coefficients of an explicit basis (which we shall prefer in this section), as $v = v_x e_x + v_y e_y + v_z e_z$. Not long after, the vector dot and cross products are revealed, as

$$v \cdot w = v_x w_x + v_y w_y + v_z w_z \tag{2.1}$$

$$v \times w = (v_y w_z - v_z w_y)e_x + (v_z w_x - v_x w_z)e_y + (v_y w_z - v_z w_y)e_z, \tag{2.2}$$

for which some algebraic intuition arises from the multiplication tables

$\cdot$	e_x	e_y	e_z
e_x	1	0	0
e_y	0	1	0
e_z	0	0	1

and

$\times$	e_x	e_y	e_z
e_x	0	e_z	$-e_y$
e_y	$-e_z$	0	e_x
e_z	e_y	$-e_x$	0

, (2.3)

and some geometric intuition arises from $v \cdot w = \|v\|\|w\| \cos\theta$ and $v \times w = \|v\|\|w\| \sin\theta\, n$, where θ is the angle between the vectors and n is a unit vector orthogonal to v and w.

2.1.1. The wedge product

A natural question that arises is how $\times$ should be generalized when the number of dimensions is not three; for which there is no satisfactory answer while retaining a bilinear function that takes two vectors and produces a third. The answer provided by geometric algebra is to use an entirely different product, the wedge product $v \wedge w$, characterized (on vectors) by the multiplication table

$\wedge$	e_x	e_y	e_z
e_x	0	e_{xy}	$-e_{zx}$
e_y	$-e_{xy}$	0	e_{yz}
e_z	e_{zx}	$-e_{yz}$	0

.

This table has the obvious benefit that it generalizes, noting that every entry is just a concatenation of its operands, with a minus sign if the order is swapped. The catch is that new e_{xy}, e_{yz}, and e_{zx} basis symbols have been conjured from thin air! These are neither scalars nor vectors, but a new type of object: bivectors or 2-vectors. Intuitively[1], these correspond to scaled and oriented planes through the origin; as $2e_x$ corresponds to a line in the positive x direction with magnitude 2, $3e_{xy}$ corresponds to the xy plane with positive orientation and magnitude 3. In general, $v \wedge w$ describes the plane spanned by v and w, and is 0 if there is no such plane due to the vectors being parallel. As a consequence, we have $v \wedge v = 0$ and $v \wedge w = -w \wedge v$.

The wedge product has one more trick up its sleeve; it is in fact not just a binary function of *vectors*, but of r-vectors, for which vectors are the special case with $r = 1$ (and scalars are the case with $r = 0$). Notably, we can evaluate $2 \wedge e_x \wedge e_{yz} = 2e_{xyz}$, and obtain a trivector (or 3-vector); the full definition of the wedge product in 3D is shown, shaded, in table 2.1. Collectively, $\{1, e_x, e_y, e_z, e_{yz}, e_{zx}, e_{xy}, e_{xyz}\}$ are known as "basis blades". The geometric intuition remains; the wedge product of n vectors is the subspace spanned by them, scaled by the volume of the parallelepiped between them; and so zero if the vectors are linearly dependent.

This provides a powerful link between algebra and geometry; if we have a plane (through the origin) represented by the bivector B, and a vector v, the expression $v \wedge B = 0$ is true if and only if v lies on B. This generalizes to hyperplanes in higher dimensions, where we refer to n-ary wedge products of vectors as n-"blades". We often choose to ignore the magnitude of our blades, instead only considering their direction; in our interpretation, e_x and $2e_x$ both represent the same line along the x-axis in the positive direction.

2.1.2. The geometric product

Geometric algebra brings one more crucial product, which when restricted to the vectors combines the $\cdot$ and $\wedge$ products such that $vw = v \cdot w + v \wedge w$. This raises a new surprise; we are adding the scalar (or 0-vector) $v \cdot w$ to the bivector (or 2-vector) $v \wedge w$, giving us an object of "mixed grade" which we call a multivector. We use the notation $\mathcal{G}(\mathbb{R}^3)$ to refer to multivectors constructed from

[1] At least, in 3D.

vectors in $\mathbb{R}^3$, similarly to [14]. We can easily extend the wedge product in section 2.1.1 from r-vectors to multivectors by linearity.

From $vw = v \cdot w + v \wedge w$ and $v \wedge w = -w \wedge v$ we can conclude

$$wv = 2v \cdot w - vw \qquad \text{and} \qquad v^2 = v \cdot v, \tag{2.4}$$

which (along with bilinearity, and the fact that our basis is orthogonal) is sufficient to evaluate the geometric product of any two basis blades by swapping pairs of adjacent vectors until all repeated basis vectors are eliminated; for instance, $e_{yz}e_{xy} = (e_y e_z)(e_x e_y) = (-e_z e_y)(-e_y e_x) = e_z(e_y \cdot e_y)e_x = e_{zx}$. The resulting Cayley table (a multiplication table for the generators) is table 2.1.

1	e_x	e_y	e_z	e_{xy}	e_{zx}	e_{yz}	e_{xyz}
e_x	1	e_{xy}	$-e_{zx}$	e_y	$-e_z$	e_{xyz}	e_{yz}
e_y	$-e_{xy}$	1	e_{yz}	$-e_x$	e_{xyz}	e_z	e_{zx}
e_z	e_{zx}	$-e_{yz}$	1	e_{xyz}	e_x	$-e_y$	e_{xy}
e_{xy}	$-e_y$	e_x	e_{xyz}	-1	e_{yz}	$-e_{zx}$	$-e_z$
e_{zx}	e_z	e_{xyz}	$-e_x$	$-e_{yz}$	-1	e_{xy}	$-e_y$
e_{yz}	e_{xyz}	$-e_z$	e_y	e_{zx}	$-e_{xy}$	-1	$-e_x$
e_{xyz}	e_{yz}	e_{zx}	e_{xy}	$-e_z$	$-e_y$	$-e_x$	-1

Table 2.1.: The Cayley table for the geometric product in 3D

The first row and column double as the headings of the table, and its entries show the product of the row heading by the column heading. The shaded cell shows $e_{yz}e_{xy} = e_{zx}$. The $\wedge$ product can be read off by only looking at the cells of this color and considering all others zero; so for instance, $e_{yz} \wedge e_{xy} = 0$.

The fact that the wedge product can be read off from the same table as the geometric product in table 2.1 indicates that there is extra structure to capture; that of "grade selection", which restricts a multivector to a piece of a certain degree. As an example, we would write that the piece of degree 2 of $e_{xy} + 2e_{yz} + 1$ is $\langle e_{xy} + 2e_{yz} + 1\rangle_2 = e_{xy} + 2e_{yz}$. The property that we see in table 2.1 is that for homogeneous multivectors a and b of degrees i and j, we have the following connection between the wedge and geometric product

$$a \wedge b = \langle ab \rangle_{i+j}. \tag{2.5}$$

Extracting geometric intuition from the geometric product is ironically rather harder than it was from the wedge product; though we can use it as a building block to build a more geometric operation.

2.1.3. Transformations

The geometric operation which can be constructed from the geometric product in section 2.1.2 is that of applying orthogonal transformations; given two vectors, we can reflect x in a with

$$x' \coloneqq axa = (2x \cdot a - xa)a = (2x \cdot a)a - \|a\|^2 x, \tag{2.6}$$

where juxtaposition is the geometric product, and the expansion of axa follows from eq. (2.4). Conveniently, this results in x' being a pure vector, despite the geometric product usually being of mixed grade.

It would be preferable for these transformations to be composable, such that we can express the composition of two reflections, in a and b, as a single element. We do this by introducing a reversion operator, defined such that for vectors $a_1, \ldots, a_n$ we have $\widetilde{(a_1 \cdots a_n)} = \tilde{a_n} \cdots \tilde{a_1}$, and for sums of such terms it extends linearly[2]. Adjusting eq. (2.6) to

$$x' \coloneqq Ax\tilde{A}, \tag{2.7}$$

where A is equal to an arbitrary product of vectors (which we call a "versor"), we can now view a sequence of transformations as a single transformation, as we can rewrite $a(bx\tilde{b})\tilde{a}$ as $(ab)x\widetilde{(ab)}$. Hence, eq. (2.7) for $A \coloneqq abc$ describes reflecting x in each of a, b, and c in sequence.

It should be clear that this is an orthogonal transformation, as reflections themselves are orthogonal. In fact, by the Cartan–Dieudonné theorem, *every* orthogonal transformation can be built in this way. When A is a product of an *even* number of vectors, we call it a "rotor", and it represents the ortho*normal* transformations.

There is one more valuable property to extract from this definition; that it generalizes to multivectors X, where it distributes over products. On versors $A = \prod_i a_i$, we have $\tilde{A}A = \prod_i \|a_i\|^2$ which is a scalar, and so $(Ax\tilde{A})(Ay\tilde{A}) = A(x\tilde{A}Ay)\tilde{A} = (\prod_i \|a_i\|^2)A(xy)\tilde{A}$. Indeed, we often choose $\tilde{A}$ to be a "unit" versor, such that $\prod_i \|a_i\|^2 = 1$. It follows that this also distributes over wedge products.

Respectively, this provides two pieces of geometric intuition: for the geometric product, that transformations can themselves be transformed, as $(UB\tilde{U})(Ux\tilde{U})\widetilde{(UB\tilde{U})} = U(Bx\tilde{B})\tilde{U}$; and for the wedge product, that vectors can equivalently be transformed either before or after being combined into blades.

Rotors behave much the same way as matrices would in linear algebra—the rotors of $\mathcal{G}(\mathbb{R}^3)$ are equivalent to the 3×3 rotation matrices, but are represented with 4 coefficients not 9. This is valuable for two reasons: it allows transformations to be computationally more efficient, and it prevents numerical error manipulating unwanted degrees of freedom, such as a skew component being introduced into a matrix. Rotors are already used under a different name in many engineering applications—the rotors of $\mathcal{G}(\mathbb{R}^3)$ are exactly the quaternions, and the rotors of

[2] It is clear this is a well-defined operation if restricted to basis vectors; we shall see it is so in section 8.2.3 without mention of this restriction.

$\mathcal{G}(\mathbb{R}^{3,0,1})$ (a notation which will be explained in section 2.1.4) are the dual quaternions with real magnitude. The value geometric algebra (GA) provides here is twofold: it unifies these similar objects and extends them to arbitrary dimension and metric, and it allows them to be applied to not just vectors but all of the geometric objects we saw in table 2.3.

2.1.4. Further geometric expressiveness

So far we have been working with just the basis vectors e_x, e_y, and e_z, in the algebra $\mathcal{G}(\mathbb{R}^3)$, which we can more precisely call $\mathcal{G}(\mathbb{R}^{3,0,0})$. $\mathbb{R}^{p,q,r}$ here refers to a vector space spanned by $p+q+r$ linearly-independent orthogonal vectors, of which p square to (i.e., have $v \cdot v$ equal to) 1, q square to -1, and r square to 0; while $\mathcal{G}(V)$ refers to the GA constructed over that vector space. We tend to omit trailing zeros in the "signature" p, q, r for brevity.

The $\mathcal{G}(\mathbb{R}^3)$ algebra is quite limiting: its blades represent only points, lines, planes, and volumes at the origin, and its versors represent reflections and rotations around the origin. This section outlines how introducing additional basis vectors makes $\mathcal{G}(V)$ more geometrically expressive.

Projective Geometric Algebra (PGA)

For the sake of visualization, let us consider an even simpler algebra with just two basis vectors, $\mathcal{G}(\mathbb{R}^2)$, which we would like to extend to include objects away from the origin. We achieve this by adding a third basis vector n_o, and interpreting objects via their intersection with the plane $n_o + xe_x + ye_y$ as shown in fig. 2.2a—a line in the new 3D space is interpreted as a point in 2D, and a plane in the new 3D space is interpreted as a line in 2D.

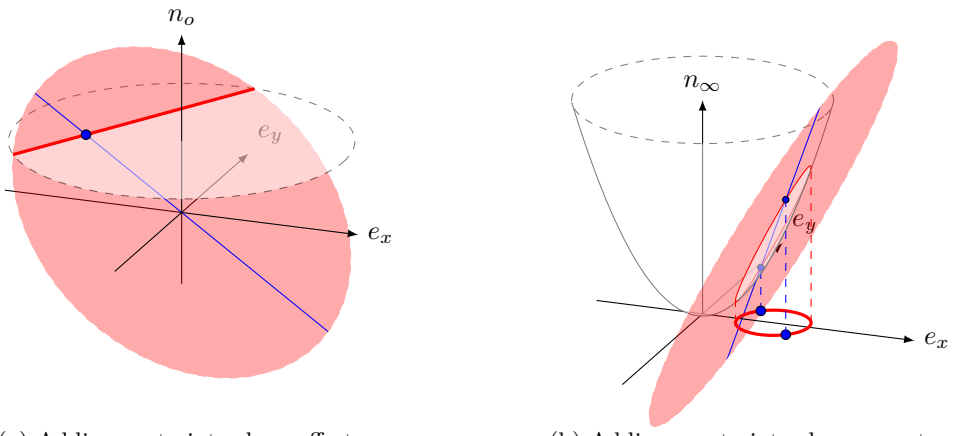
If we stop here, and declare that $n_o^2 = 0$, we have arrived at the algebra $\mathcal{G}(\mathbb{R}^{2,0,1})$, which is a variant[3] of an algebra known as 2D PGA. In the harder-to-visualize 3D analogue, 1-, 2-, and 3-blades now represents points, lines, and planes away from the origin. Our rotors now additionally represent translations and rotations away from the origin, using 8 coefficients to represent 6 degrees of freedom.

Conformal Geometric Algebra (CGA)

We can make our algebra even more powerful by adding one more basis vector n_∞. In fig. 2.2b, we do not show the n_o axis, and assume that all of our blades can now represent elements which need not pass through the origin. We now add another layer of interpretation over this 3D space, and interpret these blades in 2D via their intersection with the paraboloid $xe_x + ye_y + \frac{1}{2}(x^2+y^2)n_\infty$. A point in 2D is therefore represented by a point that lies on this parabola, for which we introduce the up function

$$X = \text{up}(x) = n_o + x + \tfrac{1}{2}|x|^2 n_\infty. \tag{2.8}$$

[3] PGA typically refers to the dual algebra, $\mathcal{G}^*(\mathbb{R}^{n,0,1})$ know as "Plane-based Geometric Algebra", where points are represented as n-vectors.



(a) Adding n_o to introduce offsets

The red plane and blue line through the origin are interpreted as an offset line and point via their intersection with the white plane with an n_0 coordinate of 1.

(b) Adding n_∞ to introduce curvature

The offset red plane and blue line are interpreted as a circle and point-pair via their intersection with the white paraboloid.

Figure 2.2.: The path to conformal geometry

We can see in fig. 2.2b that our algebra can now represent circles (of which lines are a special case), along with the other objects in table 2.3.

Point	$X = \text{up}(x) = n_o + x + \frac{1}{2}\lvert x\rvert^2 n_\infty$
Point-pair	$P = X_1 \wedge X_2$
Circle	$C = X_1 \wedge X_2 \wedge X_3$
Line	$L = X_1 \wedge X_2 \wedge n_\infty$
Sphere	$S = X_1 \wedge X_2 \wedge X_3 \wedge X_4$
Plane	$\Pi = X_1 \wedge X_2 \wedge X_3 \wedge n_\infty$

Table 2.3.: Constructions in 3D CGA

X_i are conformal points coincident with the object in question.

Our final step is to define a useful metric[4] for our new basis vectors. The choice of $n_o \cdot n_\infty \coloneqq -1$, $n_o^2 \coloneqq 0$, $n_\infty^2 \coloneqq 0$, $n_o \cdot e_i \coloneqq 0$, $n_\infty \cdot e_i \coloneqq 0$ (shown as a table in fig. 2.5b) is particularly convenient, as among other reasons it means for two 1-vectors $X \coloneqq \text{up}(x)$ and $Y \coloneqq \text{up}(y)$,

$$2(X \cdot Y) = 2(n_o + x + \tfrac{1}{2}\lvert x\rvert^2 n_\infty) \cdot (n_o + y + \tfrac{1}{2}\lvert y\rvert^2 n_\infty) \tag{2.9}$$

$$= 2(x \cdot y) - \lvert x\rvert^2 - \lvert y\rvert^2 \tag{2.10}$$

$$= -\lvert x - y\rvert^2, \tag{2.11}$$

which relates $X \cdot Y$ to the Euclidean distance between x and y. With this metric, the geometric product provides the transformations in table 2.4.

[4] A term used in geometric algebra to refer to a matrix representing an arbitrary symmetric bilinear form.

Translation by a	$T(a) = 1 + \frac{1}{2}an_\infty$	$= \exp\left(\frac{1}{2}an_\infty\right)$
Rotation by θ around $\hat{B}$	$R(\theta, \hat{B}) = \cos\left(\frac{1}{2}\theta\right) + \hat{B}\sin\left(\frac{1}{2}\theta\right)$	$= \exp\left(\frac{1}{2}\hat{B}\theta\right)$
Dilation by $\exp(-\alpha)$	$D(\alpha) = \cosh\left(\frac{1}{2}\alpha\right) + n_\infty \wedge n_o \sinh\left(\frac{1}{2}\alpha\right)$	$= \exp\left(\frac{1}{2}\alpha n_\infty \wedge n_o\right)$

Table 2.4.: Transformation rotors in 3D CGA

Here, a is a spatial vector, α and θ are scalars, and $\hat{B}$ is a unit spatial bivector. We say a multivector is spatial if it has no n_o or n_∞ components. The expression in terms of exp follows from a standard series expansion.

It turns out that from two new unit vectors $e_+^2 = 1$ and $e_-^2 = -1$ we can construct $n_o = e_- + e_+$ and $n_\infty = \frac{1}{2}(e_- - e_+)$ while still satisfying fig. 2.5b. For this reason, CGA can be described as $\mathcal{G}(\mathbb{R}^{n+1,1})$. In general, thanks to Sylvester's law of inertia, any non-diagonal metric can be converted via a change of basis into a diagonal metric with only 1, −1, and 0—the choice of basis does not affect the properties of the space, only its signature does.

Other algebras

While CGA and PGA are some of the more common geometric algebras, they are certainly not the only ones in use. To name some others: Space-Time Algebra (fig. 2.5c) gives an alternative representation of Pauli matrices in physics, and "GA for conics" extends 2D CGA to include all conic sections.

·	e_1	e_2	e_3	n_o
e_1	1	0	0	0
e_2	0	1	0	0
e_3	0	0	1	0
n_o	0	0	0	0

(a) $\mathcal{G}(\mathbb{R}^{3,0,1})$, PGA

·	e_1	e_2	e_3	n_o	n_∞
e_1	1	0	0	0	0
e_2	0	1	0	0	0
e_3	0	0	1	0	0
n_o	0	0	0	0	1
n_∞	0	0	0	1	0

(b) $\mathcal{G}(\mathbb{R}^{4,1})$, CGA

·	e_1	e_2	e_3	e_t
e_1	-1	0	0	0
e_2	0	-1	0	0
e_3	0	0	-1	0
e_t	0	0	0	1

(c) $\mathcal{G}(\mathbb{R}^{1,3})$, STA

Figure 2.5.: Metrics of common geometric algebras

2.2. Clifford algebra

In section 2.1, we presented geometric algebra in a way that was centered around a choice of basis e_x, e_y, and e_z, and assumed that our coefficients were all real numbers. In this section, we will look at a much more general definition, that of Clifford algebras[5].

[5]Some liberty is being taken here with this naming; indeed, some authors use the two names interchangeably.

2.2.1. Abstract algebra

Before summarizing a more abstract outlook on geometric algebras, it helps to be reminded what abstract outlooks on the real numbers look like. In section 2.1 we really only used a few such perspectives; some examples include:

- that they have a commutative $+$, a 0, and a $-$, all of which combine in the expected ways[6]; we say they form a "commutative additive group".
- that they have a commutative $\times$ and a 1, that combine with each other in the expected ways; we say they form a "commutative monoid".
- that in addition to forming a commutative additive group and a commutative monoid, the $+$ and $\times$ combine in the expected ways; we say they form a "commutative ring".
- that in addition to forming a commutative ring, they have at least two elements, and a $/$ defined on non-zero elements that combines with $\times$ and 1 in the expected ways; we say they form a "field".

These categorizations belong to abstract algebra, and permit us to define or prove things once, then apply the same results to multiple concrete types; integers (a commutative ring), rationals (a field), angles modulo 2π (a commutative additive group), etc.

The $v = v_x e_x + v_y e_y + v_z e_z$ approach to vectors is pedagogically appealing, and close to the representation of vectors in software; but it is not a particularly general mathematical abstraction, as it usually limits us to $\mathbb{R}^n$. The abstract approach is to talk about "vector spaces"; commutative additive groups which can be multiplied by scalars from a field. We say that $\mathbb{R}^n$ is an $\mathbb{R}$-vector space, as it can be scaled by the real numbers; but this definition permits more surprising objects like $\mathbb{R} \to \mathbb{R}$, the space of real-valued functions, to be vector spaces.

A further generalization of vectors spaces is that of modules; these drop the requirement that the scalars belong to a field, allowing them to belong to a (possibly non-commutative) ring instead. This allows us to talk about integer coordinates $\mathbb{Z}^3$ as a $\mathbb{Z}$-module, or $\mathbb{H}^2$ (pairs of quaternions) as an $\mathbb{H}$-module. We are still perfectly within our rights to say that $\mathbb{R}^3$ is an $\mathbb{R}$-module; indeed, when K is a field, then K-module and K-vector space are synonyms by definition.

We need one last abstraction before we can move onto Clifford algebras; a meaning for the word "algebra". An (associative) R-algebra over a commutative ring R is an R-module such that scalar multiplication commutes across multiplication; for instances, complex matrices are an $\mathbb{R}$-algebra because for a real number c and complex matrices M and N, we have $cMN = McN = MNc$, even though matrices do not commute in general.

[6] Here is as good a place as any to note that IEEE `floats` used in software do not behave as expected!

2.2.2. Notation

In this thesis, we will use the colon notation $x : R$ to say "x is in the ring R", and $v : V$ to say "v is in the vector space V". Similarly, we will use $F : V \to W$ to say "F is a function from the space V to the space W". When referring to the value of a function $F : V \to W$, we will use $F = (v \mapsto w)$. Whether $\to$ refers to a function, a linear map, or some other type of morphism is usually left to the prose; as is the distinction as to whether R and V are rings, modules, vector spaces, or some other abstract object. In some cases we will write $F : V \to_R W$ to indicate that a linear map or algebra morphism is linear with respect to the ring R.

For functions of two variables, we have two choices of notation; $F : U \times V \to W$ or $F : U \to (V \to W)$, where we will omit the parentheses. This second "curried"[7] interpretation may seem unusual, but it is convenient for us for reasons that eventually become apparent in eq. (8.2). Similarly, we shall use $F = (u \mapsto v \mapsto w)$ for writing the values of such functions, and use $f(u, v)$ and $f(u)(v)$ interchangeably.

Note in particular that for commutative R, an R-bilinear map $F : U \times V \to W$ can be considered as an R-linear map from U to the space of R-linear maps from V to W, which is the $F : U \to V \to W$ spelling.

2.2.3. Quadratic Forms

In defining $\mathcal{G}(V)$ for an R-module V, we have omitted an important property of the V; R-modules do not provide their elements $v : V$ with a dot product $v \cdot w$, which we need to write eq. (2.4). The natural generalization would be to write $\mathcal{G}(V, B)$ which provides a (symmetric) bilinear map $B : V \to V \to R$; one that is linear in both arguments. Indeed, [15] uses this definition, but labels the resulting construction a "'quantum' Clifford Algebra".

Instead, most definitions choose to provide a "quadratic form" $Q : V \to R$, which according to[8] [16, §3.4], satisfies $Q(rm) = r^2 Q(m)$, and additionally is such that the polar form $\text{polar}[Q](x, y) \coloneqq Q(x + y) - Q(x) - Q(y)$ is bilinear. We recover the $v \cdot w$ product from this definition as $\frac{1}{2}\,\text{polar}[Q](v, w)$; we call this the "associated bilinear form", and it is useful because it satisfies $v \cdot v = Q(v)$.

When we write $\mathcal{G}(\mathbb{R}^{p,q,r})$ above[9], we are interpreting the tables in fig. 2.5 as bilinear forms $B^{p,q,r}$, then defining $Q^{p,q,r}(v) \coloneqq B^{p,q,r}(v, v)$; so for instance, $Q^{3,0,0}(v_x e_x + v_y e_y + v_z e_z) \coloneqq v_x^2 + v_y^2 + v_z^2$. For the majority of the rest of this thesis we shall concern ourselves with general quadratic forms

[7] So-named in reference to the mathematician Haskell Curry.

[8] In fact this definition generalizes further to *semi*-modules (those without negation), according to [17] and [*mathlib#14303*], instead requiring $Q(x+y) = Q(x)+Q(y)+B_C(x, y)$, where B_C is some (possibly non-unique) "companion" bilinear form; but the author is not aware of this generalization being considered in the context of Clifford algebras.

[9] which some other authors write $\mathcal{C}\ell(p, q, r)$.

instead of this special case, and so shall write[10] $\mathcal{G}(V, Q)$.

2.2.4. The tensor algebra, $\mathcal{T}(V)$

Our last stepping stone we will need is that of the tensor algebra, which makes explicit the notion of "inventing" the new e_{xy} symbols as we did in section 2.1.1. The tensor algebra starts with an R-module V, and augments it with an R-bilinear multiplication (normally written $u \otimes v$, but we shall simply write uv) to obtain the R-algebra $\mathcal{T}(V)$. Effectively, this algebra lets us write down any products of vectors such as $1 + u + v^2$, and tells us that we can only consider two elements equal if they are equal under the axioms of *non-commutative* rings. It is the largest algebra generated by V with this property. Contrasting with the product in section 2.1.1, the product in the tensor algebra has no rule about simplifying v^2.

2.2.5. A definition of $\mathcal{G}(V, Q)$

We now have all the pieces we need to describe a Clifford algebra. Starting with an R-module V and a quadratic form $Q : R \to V$, we define $\mathcal{G}(V, Q)$ by starting with $\mathcal{T}(V)$, and introducing the extra rule that when determining if two elements are equal, we can apply the rule $v^2 = Q(v)$. The multiplication we obtain is none other than the geometric product from section 2.1.2.

As an example of applying this rule, we can find that

$$\begin{aligned} wv &= (w + v)^2 - w^2 - v^2 - vw && (2.12)\\ &= Q(w + v) - Q(w) - Q(v) - vw && (2.13)\\ &= \text{polar}[Q](v, w) - vw, && (2.14) \end{aligned}$$

which is the more abstract version of eq. (2.4), replacing $2v \cdot w$ with $\text{polar}[Q](v, w)$. This is helpful for generality, as by skipping the associated bilinear form from section 2.2.3 entirely, we avoid requiring that our scalars are divisible by 2.

More formally, we say we are taking a quotient by the (closure of the) relation $v^2 = Q(v)$. The reader will be spared the full formality of this definition for now, as we will have plenty of time for it later. As $\mathcal{T}(V)$ is an R-algebra, this quotient, $\mathcal{G}(V, Q)$, is also an R-algebra.

2.2.6. The exterior algebra, $\bigwedge(V)$

While section 2.2.5 provides us the geometric product, it does not provide a wedge product. A partial solution to this is to work with the exterior algebra, $\bigwedge(V)$. This is in fact just a special case of the Clifford algebra, where instead of requiring $v^2 = Q(v)$ we take $v^2 = 0$ (and so $Q := 0$); that is, we can consider $\bigwedge(V)$ and $\mathcal{G}(V, 0)$ identical.

[10] This notation could be argued both to be unnecessarily verbose (as the module V is implied in the choice of Q, so we could write $\mathcal{G}(Q)$) and insufficiently precise (as the base ring R is left implicit, so we should write $\mathcal{G}_R(V, Q)$). In the rare cases in this thesis where the base ring is not R, it can be inferred from adjacent $\otimes_R$ or $\cong_R$ notation.

This $\bigwedge(V)$ gives us the wedge product as the regular product, but in the formulation in section 2.1, we would like to have access to both the geometric and wedge products of $\mathcal{G}(V, Q)$ at the same time. Defining the wedge product on $\mathcal{G}(V, Q)$ amounts to choosing an isomorphism $\text{to\_ext} : \mathcal{G}(V, Q) \cong \bigwedge(V)$ that sends vectors to vectors, with which we can write

$$x \wedge y = \text{to\_ext}^{-1}(\text{to\_ext}(x)\,\text{to\_ext}(y)). \tag{2.15}$$

We shall explore how this isomorphism is constructed in section 8.4.

The exterior algebra also provides us with the notion of grade selection used in eq. (2.5); more formally, we say that $\bigwedge(V)$ is an $\mathbb{N}$-graded ring, a concept that will be explained further in chapter 6. To recover the "grade selection" operator on $\mathcal{G}(V, Q)$ from geometric algebra[11], we must once again transport along to_ext.

[11] Which unless $Q = 0$ does not qualify as an $\mathbb{N}$-graded ring.

3

Software

> Programmers are always surrounded by complexity; we cannot avoid it. [...] If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.
>
> *(C. A. R. Hoare)*

It appears to be a rite of passage for theses about geometric algebra to introduce a new software library; [18, §4] introduces LibCGA, [19, §A] introduces Versor, and [20] is devoted in its entirety to Gaigen. Section 3.1 outlines one of the reasons that developing such software is appealing; there are many ways to encode the structure of geometric algebra into the type system of different programming languages. The author's original intent was to avoid perpetuating this trend, instead adopting and contributing new features to existing geometric algebra packages; notably the numeric package `clifford` (section 3.2) and the symbolic package `galgebra` (section 3.3), both for the Python programming language.

Ultimately, it was the disconnect between the structure of these libraries and the mathematics they encapsulate that motivated the main direction of this thesis. This section outlines some of these challenges, and introduces the reader to a rather unusual kind of "programming" language; that of formal mathematics (section 3.4). In particular, it provides an introduction to one specific language of this kind, the Lean theorem prover; from which many code samples will be presented throughout the rest of this thesis.

3.1. Typing considerations

Writing software libraries for Geometric Algebra presents an unusual challenge—such libraries should not only provide algorithms, but data types which look, feel, and perform like the numeric types native to the programming language they target. Programming languages today already have native support for a limited set of algebras; namely the real numbers $\mathcal{G}(\mathbb{R}^0)$ and often the

complex numbers $\mathcal{G}(\mathbb{R}^{0,1})$; but support typically stops there.

It is worth expanding a little on the concept of "types". In the language of Type Theory, every object or "term" has an associated "type", which defines its meaning and behavior: for instance, in Python the term `42` has type `int`. Exactly how granular the types are is up to the language, a common example of which is whether the natural numbers $\mathbb{N}$ are considered to be the same type as the integers $\mathbb{Z}$. Ascription of types to terms is therefore mathematically subjective, but it is an important part of library design—types are used in the world of software engineering to associate operators and memory layouts to terms.

Just as complex numbers and real numbers can be considered to have different types, multivectors from distinct algebras can be too. But algebras themselves can be considered terms, perhaps of type `Algebra`. This means that the type of our multivector is a function of the algebra it belongs to—in the language of type theory, it is a "dependent type". As an example working in $\mathcal{G}(\mathbb{R}^2)$, we could say that the term $1 + e_{12}$ has type `Multivector` $\mathcal{G}(\mathbb{R}^2)$.

Dependent types live on the border between software engineering and computer science, in that they are hard to work with in languages known for performance, and easy to work with in languages which are not. Existing GA implementations in the former group deal with this in multiple ways:

- Type erasure. The type of multivectors is unified across all algebra, and each multivector term carries around data about its associated algebra. In type theory, we say this is a "sigma type". In implementations, this comes with memory and speed costs.
- Code generation. The generator works with the algebra terms, and emits code which declares specific multivector types, via either
  - Source code generation, which often supports multiple target languages from a single generator (Garamon [21], ganja.js [22], Gaalop [23])
  - Compile-time generation, using mechanisms like C++ templates where a limited meta-language allows restricted manipulation of the algebra objects. (GATL [24], GAL [25], versor [19])
- Runtime types, often via just-in-time compilation so as not to sacrifice efficiency (ganja.js [22])

3.2. Numeric

As a short example of how these typing considerations manifest in practice, let us look at `clifford` [26]; a GA package for the Python programming language for numeric computation in arbitrary algebras. While the package has been around since 2006, sizeable contributions have been made to it as part of the early work in this thesis. The source code is publicly available on GitHub, and the package is periodically released to the Python Package Index (PyPI) from which it can be installed with `pip install clifford`.

`clifford` uses one of the simplest representations of a multivector—a dense `numpy` [27] array of the 2^n coefficients of the basis blades, where n is the dimension of the algebra. Each multivector carries around a reference to the algebra it belongs to, in what was described as the "type erasure" approach in section 3.1. The library encodes the geometric product as a product with a constant tensor G, $(ab)_j = \sum_{i,k} a_i G_{ijk} b_k$, where i, j, k are the indices of each basis blade. This tensor is stored sparsely, as for an n-dimensional algebra only 2^{2n} of its 2^{3n} elements are non-zero. The tensors for the outer ($G^\wedge$) product is simple to express in terms of G via the expression in eq. (2.5) as

$$G^\wedge_{ijk} = \begin{cases} G_{ijk} & \text{if } i + k = j \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

This approach is neither novel nor efficient compared to other more sophisticated approaches like [21], but it is simple, and is performant enough for small algebras and pedagogical use cases.

At any rate, performance considerations would be dominated by the overhead of the Python programming language, which is interpreted rather than compiled to native machine code. In general, the way this problem is solved in Python is to delegate performance-critical loops to C code. An example of this is multivector addition, which calls into the `numpy` package to perform elementwise addition. To make the geometric product efficient, a more sophisticated approach is used. For this, `clifford` leverages the `numba` [28] Python package, which uses LLVM to perform just-in-time compilation (or JIT-ing) of Python code into machine code the first time it is run. This is used to compile a custom product function for each algebra.

3.2.1. Accelerator compatibility

It is this `numba` integration within `clifford` where the design of the type system becomes complex; at least, after the substantial redesign performed by the author described in this section.

As a simple example, let us imagine we are writing the `rotor_between_planes` function from the `clifford` library for the first time. Our pure-Python implementation might start as shown in fig. 3.1a, where we use `*` as the geometric product, and `.normal()` to normalize the magnitude of the resulting rotor to 1.

Before the overhaul, enabling JIT-compilation would require a total rewrite of the function into the form shown in fig. 3.1b. Essentially, there were two entirely different interfaces to the library—a high level interface that worked on `MultiVector` objects and convenient operator, and a low-level interface for only $\mathcal{G}(\mathbb{R}^{4,1})$ that worked on raw `numpy` arrays of blade coefficients. Converting a regular function to a JIT-ed function required swapping the implementation to use the low-level interface, and then writing a wrapper function to re-expose the high-level interface over the top—resulting in twice as many functions to keep track of. For an algorithm built up of multiple functions, as shown on the left of fig. 3.2, JIT-ing the outermost function resulting in this duplication of every function in the call graph, as shown on the right of fig. 3.2.

This is at odds with one of the key selling points of `numba`, which according to the project website is that "[the user can] Just apply one of the Numba decorators to [their] Python function, and Numba does the rest". The difficulty here is that `numba` does not have any knowledge of the type of a multivector, which `clifford` works around by dropping down to the level of coefficient arrays, objects which `numba` does know the type of.

Thankfully, `numba` includes an extension mechanism to teach it about new types, which after substantial effort, enabled writing code in the style of fig. 3.1c. The core of this extension process is to define a mapping that determines the appropriate numba type for a pure-Python object. Often, these types are parametric; for instance, an array of integers would have type `ndarray` in Python, but would have type `Array(int64)` in `numba` (fig. 3.3c)—when mapping into `numba`, parts of the value are lifted into the type. For `clifford.Layout` objects (that is, the algebra itself, $\mathcal{G}(\cdots)$), we take the rather unorthodox approach of pushing the entirety of the pure-Python value into the `numba` type, leaving the `numba` value completely void of data (fig. 3.3b)! This makes it impossible to construct new algebras within JIT-ed code, but this is rarely a relevant limitation. With `LayoutType` defined, the rest of the design falls out quite simply—a `MultiVectorType` is

```python
def rotor_between_planes(P1, P2):
    return (1 - (P2 * P1)).normal()
```

(a) Pure-Python implementation

```python
@numba.jit
def _rotor_between_planes_val(P1_val, P2_val):
    P1_P2_val = -gp_val(P1_val, P2_val)
    P1_P2_val[0] += 1  # scalar part
    return norm_val(P1_P2_val)

def rotor_between_planes(P1, P2):
    rotor_val = _rotor_between_planes_val(
        P1.value, P2.value)
    return MultiVector(rotor_val)
```

(b) Old method of JIT-ing

```python
@numba.jit
def rotor_between_planes(P1, P2):
    return (1 - (P2 * P1)).normal()
```

(c) New method of JIT-ing

Figure 3.1.: Comparison of implementations showing improvements to the JIT-ing interface

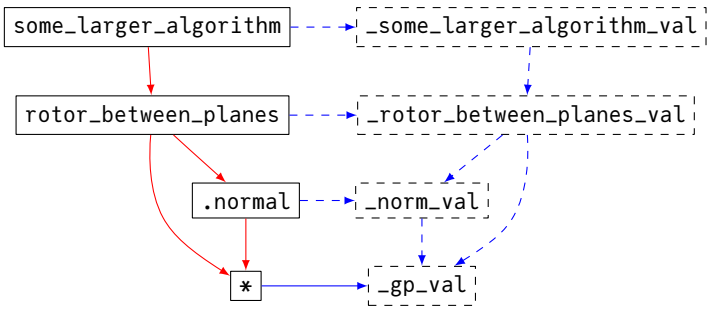


Figure 3.2.: The call-graph of a pure-Python algorithm in the style of fig. 3.1a, shown in red, vs the same algorithm JIT-ed in the old style of fig. 3.1b, shown in blue.

With the changes to `clifford` that enable fig. 3.1c, all of the dashed elements are handled automatically by `numba`, rather than having to be written by hand.

parametrized by its layout and the scalar type of its coefficients (fig. 3.3a).

With the types in place, the remaining work was to: provide "boxing" and "unboxing" functions written in LLVM assembly to convert between Python objects and the new types; to hook up the operators like `^` and `|` to perform the appropriate GA operations; and to clean up the tens of functions written in the style of fig. 3.1b to look like fig. 3.1c, a step which removed hundreds of lines of code.

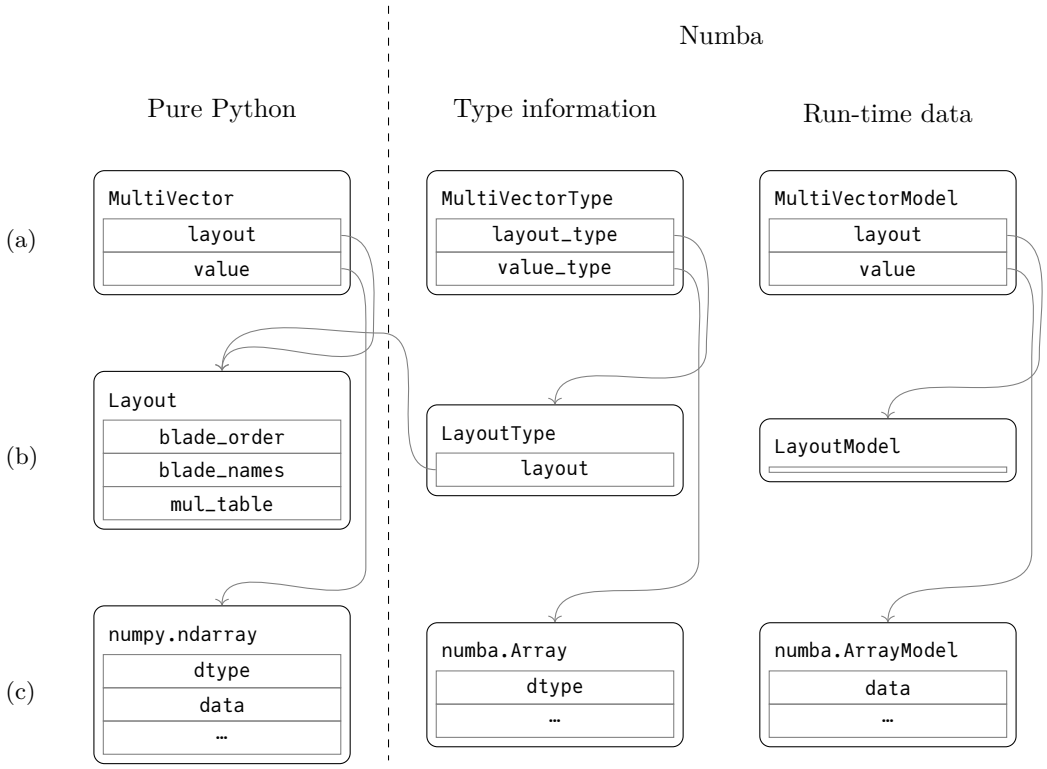


Figure 3.3.: The architecture of the type system interfacing `clifford` and `numba`.

Each rounded box shows an object, with the rectangles within it listing its fields. Arrows from a field point to the type of value it holds. Adding a new type to numba requires two pieces—compile-time information about the type itself, and run-time information about its layout in memory. The array types shown in the last row, fig. 3.3c, are not part of `clifford`, but are included to illustrate how `numba` handles other types—in the case, the fields are split between the compilation and runtime.

3.3. Symbolic

While for engineering applications, numerical computation is often sufficient, symbolic computation using computer algebra packages is invaluable in a research setting, as it provides a mechanism for checking results. Notable example of such systems are Axiom, Maple, Magma, Mathematica,

and SageMath.

One part of SageMath is `sympy` [29], the canonical Python package for performing symbolic computing, which covers a sprawling range of fields from combinatorics to quantum physics. Notably absent is GA—which in fact was present until its removal in 2015 in part due to lack of communication between its developer and the other `sympy` maintainers. Thankfully, work continued in the `galgebra` [30] Python package. As with section 3.2, significant development of this package has taken place as part of the early work in this thesis.

Under the hood, `galgebra` works by creating a unique `sympy.Symbol` for each basis blade, and then multivectors are stored as algebraic expressions of their coefficients. Multiplication is essentially performed in the same way as `clifford`, except instead of a numeric tensor, a lookup table mapping two basis blades to their product is used. What sets `galgebra` apart from `clifford` is its ability to work in arbitrary, symbolic, and even curvilinear metrics.

As well as an explorative tool, symbolic computations have one more benefit—an algorithm can be symbolically optimized, and then the resulting expression to generate code generation for a variety of languages. This is the entire premise behind Gaalop [23], but it suffers from inventing its own GA-exclusive frontend rather than integrating with existing tools.

3.3.1. Example: multivector derivatives

Prior to the author's work, `galgebra` supported only two differentiation modes: differentiation by a scalar, and the usual vector calculus operators obtained by differentiating by a vector. After a complete overhaul of its formerly-untested differential operator framework, support for the multivector derivative characterized by [14, §2, eq. (2.13)] was implemented. Listing 3.4 shows it in action.

Note that the expressions produced by `galgebra` are in terms of basis blade coefficients, and do not generalize across different algebra dimensions (the 3 would be n for a generic algebra). This tool is therefore primarily useful for *checking* derivatives when applied to a particular problem, rather than for computing derivatives from start to finish.

```
>>> # create the algebra
>>> from galgebra.ga import Ga
>>> g = Ga('e_x e_y e_z', g=[1, 1, 1])

>>> # create symbolic blades
>>> a = g.mv('a', 'vector'); a
a = a^x e_x + a^y e_y + a^z e_z
>>> B = g.mv('B', 'bivector'); B
B = B^xy e_x ^ e_y + B^xz e_x ^ e_z + B^yz e_y ^ e_z
```

(a) Creating symbolic blades in $\mathcal{G}(\mathbb{R}^3)$

```
>>> # create a differential operator
>>> dB = g.make_grad(B); dB
-e_x ^ e_y d/dB^xy - e_x ^ e_z d/dB^xz - e_y ^ e_z d/dB^yz

>>> # apply it to an expression, and check the result
>>> dB * (B*a)
3a^x e_x + 3a^y e_y + 3a^z e_z
>>> dB * (B*a) == 3 * a
True
```

(b) Using a derivative operator ∇_B

Listing 3.4.: Example usage of the `galgebra` package

Note that unlike `clifford`, output is shown in LaTeX in compatible environments like Jupyter Notebooks.

3.3.2. Flexibility concerns

The coordinate-based representation in section 3.3.1 is a symptom of a lack of flexibility in the design of algebra within `sympy`. There are various other ways in which the experience of working with `galgebra` multivectors is much worse than the usual experience when working with real numbers. To understand why, it helps to understand how various other number systems are represented in `sympy`.

In listing 3.5, some `sympy` code is shown for each of four common "number" types that constructs an expression in two symbols and inspects its representation. Under the hood, `sympy` is building an expression tree, where each node is an operator, and the terminal nodes are either variables or constants. `show_repr` is displaying that tree in the form of nested function calls. For real (listing 3.5a) and complex (listing 3.5b) numbers, the behavior is exactly the same; in both cases, the operator nodes are generic `Mul` and `Add` nodes[1], For matrices (listing 3.5c), the story is different; not only do we need to use an entirely new `MatrixSymbol` type, but the expression we end up with is built out of a new set of operator types that are not the same as those in listings 3.5a and 3.5b. For quaternions (listing 3.5d), the outlook is worrying; there is no means to create a quaternion-valued symbol at all, and instead we must assemble a quaternion from symbolic coefficients. As a result, what we get back is an expression for each coefficient (which we did not see for real and imaginary parts in listing 3.5b).

This coordinate-based approach to symbolic algebra used by `sympy.Quaternion` is particularly inconvenient when it comes to checking algebraic results, as can be seen in listing 3.6; the results produced by `sympy` are always in the fully-expanded form, which in the cases in listing 3.6 is far more verbose than the output shown in the captions.

It is unfortunate then that this is effectively the same as the approach used by `galgebra`, and that like `sympy.Quaternion`, `galgebra.Mv` (the multivector type) is in many ways a "second class citizen" in `sympy` as a result: functions like `sympy.exp` or `sympy.diff` cannot be used on `galgebra`s

[1] `sympy` does not have a `Sub` node, so $x - y$ is converted to $x + (-1)y$.

```
>>> x, y = symbols('x y', real=True)
>>> expr = (x + y)*(x - y); expr
(x − y)(x + y)
>>> show_repr(expr)
Mul(Add(x, Mul(Integer(-1), y)),
    Add(x, y))
```

(a) Real numbers

```
>>> x, y = symbols('x y', complex=True)
>>> expr = (x + y)*(x - y); expr
(x − y)(x + y)
>>> show_repr(expr)
Mul(Add(x, Mul(Integer(-1), y)),
    Add(x, y))
```

(b) Complex numbers

```
>>> m, n = symbols('m n')
>>> X = MatrixSymbol('X', m, n)
>>> Y = MatrixSymbol('Y', m, n)
>>> expr = (X - Y)*(X + Y).T; expr
(X − Y)(X^T + Y^T)
>>> show_repr(expr)
MatMul(MatAdd(X, MatMul(Integer(-1), Y)),
       MatAdd(Transpose(X), Transpose(Y)))
```

(c) Matrices

```
>>> x, xi, xj, xk = symbols('x x_i x_j x_k');  xq = Quaternion(x, xi, xj, xk)
>>> y, yi, yj, yk = symbols('y y_i y_j y_k');  yq = Quaternion(y, yi, yj, yk)
>>> expr = xq * yq; expr
(xy − x_i y_i − x_j y_j − x_k y_k) + (xy_i + x_i y + x_j y_k − x_k y_j) i
  + (xy_j − x_i y_k + x_j y + x_k y_i) j + (xy_k + x_i y_j − x_j y_i + x_k y) k
>>> show_repr(expr)
Quaternion(Add(Mul(x, y), Mul(Integer(-1), x_i, y_i), Mul(Integer(-1), x_j, y_j), Mul(Integer(-1), x_k, y_k)),
           Add(Mul(x, y_i), Mul(x_i, y), Mul(x_j, y_k), Mul(Integer(-1), x_k, y_j)),
           Add(Mul(x, y_j), Mul(Integer(-1), x_i, y_k), Mul(x_j, y), Mul(x_k, y_i)),
           Add(Mul(x, y_k), Mul(x_i, y_j), Mul(Integer(-1), x_j, y_i), Mul(x_k, y)))
```

(d) Quaternions

Listing 3.5.: Comparison of expression tree representations for various number systems in `sympy`.

While real (a) and complex (b) numbers use the same objects for their representation, matrices (c) have to reinvent addition, multiplication, and `Symbol`. Quaternions (d) do not support symbolic operators at all, only coefficient-wise operations.

```
>>> (xq * xq * xq**-1 * yq).simplify()
(xy − x_i y_i − x_j y_j − x_k y_k)
      + (xy_i + x_i y + x_j y_k − x_k y_j) i
      + (xy_j − x_i y_k + x_j y + x_k y_i) j
      + (xy_k + x_i y_j − x_j y_i + x_k y) k
```

(a) Showing that $x_q x_q x_q^{-1} y_q = x_q y_q$

```
>>> def q_diff(Φ, dq):  # similar to ∇Φ
...     ℍ = Quaternion
...     I, J, K = ℍ(0,1,0,0), ℍ(0,0,1,0), ℍ(0,0,0,1)
...     return (Φ.diff(dq.a) + I*Φ.diff(dq.b)
...         + J*Φ.diff(dq.c) + K*Φ.diff(dq.d))
>>> q_diff(xq*yq, xq)
−2y + −2y_i i + −2y_j j + −2y_k k
```

(b) Showing that $\nabla_{x_q}(x_q y_q) = -2y_q$

Listing 3.6.: The cost of coefficient-wise representations

The variables `xq` and `yq` are as in listing 3.5. While the results are still correct, they are not in the simplified form we wanted them, because that form cannot even be *represented* in `sympy`.

multivector, as these functions expect an expression, not a container holding multiple expressions. Even integrating `galgebra.Mv` objects with the `sympy` LaTeXprinter required a substantial amount of work by the author to both `sympy` and `galgebra`.

While it may seem that the approach used by matrices (listing 3.5c) in `sympy` would be more desirable for `galgebra.Mv`, this is substantially more difficult than it appears. Not only would we have to create `MvAdd`, `MvMul`, and `MvSymbol` classes, but due to the ad-hoc nature of `sympy`'s dynamic dispatch (the process of choosing which behavior a function should have based on the types of its arguments) means that we would have to patch around 100 places across `sympy`[2] to add special handling for these new types. Our prospects look even worse if we want to support matrices of multivectors, as we would have to do the same thing all over again for a `MatMvAdd`; the design just doesn't compose.

3.3.3. Correctness issues

Even if we did take on the enormous task of building matrix-style support for multivectors into `sympy`, we would likely find ourselves faced with matrix-style rough edges; which is to say, if you add 100 special cases across the codebase, there is a very large surface for subtle bugs.

To demonstrate this, we can look at an example of working with determinants of matrices in `sympy`, as in listing 3.7. We can ask `sympy` to determine whether $x = y$ is true by entering `Eq(x, y)`. In some cases, `sympy` automatically "evaluates" our expressions, performing hard-coded and somewhat arbitrary transformations. For instance, listings 3.7b and 3.7c both end up dispatching to `MatMul._eval_determinant`, which knows how to evaluate a variety of determinant expressions. The ability of this mechanism stops there though, and we have to invoke the `.simplify()` method to fully demonstrate the expected result in listing 3.7c. Even `.simplify()` can go so far; when faced with the "Weinstein–Aronszajn identity" in listings 3.7d and 3.7e on rectangular or even square matrices, we hit an exception somewhere within `sympy`'s internals.

Bugs of course happen in software, and we are fortunate here that the resulting behavior was a crash and not a wrong answer. In this particular case, the bug was fixed in [sympy#20691] in the `sympy` 1.9 release. However, the *kind* of bugs that arise are in some sense a function of the foundations and language that we choose to build upon. There is usually a trade-off to be made here: using languages with manual memory management can bring better performance, but risks memory access bugs; using languages with weak typing can make development faster, but risks type-related bugs; using untyped numerals for physical calculations is more compatible with other libraries, but risks unit- and dimension- related bugs[3]. In the case of sympy, we are, in the author's opinion, using a framework with a fragile expression model built atop of a language that is ill-equipped to verify that our expressions and simplifications are well-formed.

`sympy` is of course just one of many symbolic software packages, and other systems may well have less error-prone and extensible architectures. Fundamentally though, all such tools are still

[2] Estimated by searching for `MatAdd` and `is_Matrix` in the `sympy` source code.
[3] For which the classic example is the loss of the Mars Climate Orbiter, [31].

```
>>> m, n = symbols('m, n')
```

```
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, n)
```

```
>>> P = MatrixSymbol('P', m, n)
>>> Q = MatrixSymbol('Q', n, m)
```

(a) Initialization of symbols for square ($A, B : \mathbb{R}^{n\times n}$) and rectangular ($P : \mathbb{R}^{m\times n}$, $Q : \mathbb{R}^{n\times m}$) matrices

```
>>> Eq(det(A*B), det(A)*det(B))
True
```

(b) $\det AB = \det A \det B$

```
>>> Eq(det(A*B*A.inv()), det(B))
|A^-1| |A| |B| = |B|
>>> Eq(det(A*B*A.inv()), det(B)).simplify()
True
```

(c) $\det ABA^{-1} = \det B$

```
>>> I = Identity
>>> Eq(det(I(m) + P*Q), det(I(n) + Q*P))
|𝕀 + PQ| = |𝕀 + QP|
>>> Eq(det(I(m) + P*Q),
...    det(I(n) + Q*P)).simplify()
NonSquareMatrixError:
  Det of a non-square matrix
```

(d) $\det(I_m + PQ) = \det(I_n + QP)$

```
>>> I = Identity
>>> Eq(det(I(n) + A*B), det(I(n) + B*A))
|𝕀 + AB| = |𝕀 + BA|
>>> Eq(det(I(n) + A*B),
...    det(I(n) + B*A)).simplify()
AttributeError:
  'Mul' object has no attribute 'shape'
```

(e) restriction of (d) to square matrices

Listing 3.7.: Manipulating matrix determinants in `sympy` 1.8

After declaring some matrices in (a), `sympy` automatically reduces simple statements like (b) to their truthiness. For more complex statements like (c), we must invoke `.simplify()`. On the "Weinstein–Aronszajn identity" in (d), `sympy` internally creates an illegal expression and crashes. Acquiescing to the error message produces an even more opaque error in (e).

vulnerable in a crucial way: even if we find a system that can be flexibly extended to support GA, designed in a way that makes constructing nonsensical expressions impossible, at some point we will have to teach it GA-specific simplification rules, and it will be entirely on us to promise that they are *mathematically* correct. If we want our system to help us with "mathematical" correctness, it needs to understand proofs; but once our system does that, it's no longer a symbolic algebra system, but a full-fledged theorem prover.

3.4. Formal

> Parts of this section are extracted from §3-5 of "Formalizing Geometric Algebra in Lean" [10], instead of appearing with the main part of that work in chapter 9.

Theorem proving software must do more than simply encode and verify mathematical truth; it must make doing so ergonomic for the user, and minimize the inevitable friction when converting a pen and paper argument into a machine-readable one. The typical workflow for this conversion is split into two parts—translating the theorem statement, and then translating the proof.

The use of dependent type theory (as opposed to the set theory typically used by mathematicians) helps with the theorem statements, as it provides a mechanism to reject nonsensical statements like $1 = (2 < 3)$ (with a type error like "$2 < 3$ is a Prop, expected an $\mathbb{N}$"). Flexible notation is

also valuable in theorem statements, as mathematics is symbol-heavy; for instance, being able to write $\sum$ instead of "sum" makes it easier to align paper with screen.

Where the software can really shine though is in the proofs, via interactive and automated theorem proving. The former is about showing unresolved "goals" as the proof progresses, to help the user as they write and to guide them what to do next. The latter is invaluable for discharging goals the user views as trivial, such as $(a + b) + c = (b + c) + a$—while the user could manually apply associativity and commutativity lemmas, automation prevents them having to waste time thinking about this. The scale of automation can vary from filling in the blanks (converting a proof of $x < x + 1$ into a proof of $2 < 3$), to performing a CAS-like simplification, to applying a sophisticated machine-learning model [32; 33]. Since automation is typically implemented as proof-generation, it doesn't impact the trustworthiness of the prover—the generated proof is subject to the same mechanical scrutiny as a hand-written one. The line between interactive and automated is frequently blurred.

The Lean language used in this thesis is just one of many theorem proving languages, and while it has all of the properties described above, it is not unique in that it does so. Its use of dependent types is very similar to Coq, and its heavy use of notation similar to Agda. Its automation tools are currently less powerful than those of Isabelle/HOL, without an equivalent to Isabelle/HOL's powerful `sledgehammer` tactic [34]; however, it is increasingly catching up with tactics like `aesop`[35].

Lean underwent a substantial upgrade during the writing of this thesis, from Lean 3 to Lean 4. This came with many improvements, though arguably in the long run the most important change is that Lean 4 is now actively developed[4] and funded, and so will continue to improve. For consistency with already-published papers, this thesis uses a mixture of both languages. The languages are thankfully very similar, but for clarity, they will be presented in different-styled boxes, as

```
-- this is Lean 3 code
```

```
-- this is Lean 4 code
```

or inline, as `lean_three_code` and `leanFourCode`.

3.4.1. An introduction to the Lean theorem prover

In this section, we will give a very brief introduction to the Lean language, in order to aid with reading the fragments of Lean code[5] in the rest of this thesis. The Lean community website provides a much more in-depth set of reference materials [36], as well as an in-browser Lean 4 environment at `http://live.lean-lang.org`. Most of the code samples will be presented as Lean 3, but when the Lean 4 code is not identical, both will be shown side by side.

We'll start with a simple definition, showing how to define a value with a given name and type.

[4] Even at the beginning of the work in this thesis, Lean 3 as a language was effectively in "maintenance mode", receiving no new language features.

[5] Note that Lean source code makes heavy use of non-ASCII unicode characters, which some PDF readers are unable to copy to the clipboard faithfully.

Here, we define the name `two` to refer to the natural number (`ℕ`) 2:

```
-- name : type := value (or "term")
def two : ℕ := 2
```

Function *types* are declared with a `→` separating the types of their inputs and outputs. Function *values* introduce their variable binders using a `λ` (or `fun` in Lean 4) followed by variable names. Here, we define a function that takes a natural number and doubles it:

```
def double : ℕ → ℕ := λ a, two * a
```

```
def double : ℕ → ℕ := fun a => two * a
```

For convenience, the language lets us introduce the variable and declare its type in one place; so we could also have written this:

```
def double (a : ℕ) : ℕ := two * a
```

Function application is notated with juxtaposition, which has higher precedence than most operators. Note that sometimes a `$` (or `<|` in Lean 4) is used in place of parentheses.

```
/- examples of function application -/
def four := double 2
def five := double 2 + 1
def six := double (2 + 1)
def six' := double $ 2 + 1
```

What makes Lean a language for theorem proving, and not just a regular functional programming language, is its `Prop` type, which holds mathematical statements.

```
def likely : Prop := ∃ x : ℕ, x + x = 2
def unlikely : Prop := ∃ x : ℕ, x*x = 2
```

Mathematical statements are themselves types (are permitted to appear to the right of a colon), whose values constitute their proofs. We use the keyword `lemma` or `theorem` instead of `def` when providing proofs, but the syntax is otherwise identical:

```
lemma likely_proof : likely := ⟨1, rfl⟩
lemma unlikely_proof : unlikely := sorry
```

We can prove `likely` by claiming that `1` is a suitable `x`, and proving that `1 + 1 = 2` by definition (`rfl` is a proof that `a = a`). However, while we are free to make the false statement `unlikely`, Lean does not allow us to prove it. Instead, it grants us an escape with the `sorry` keyword, which is used to mark proofs as omitted. Using `sorry` sets a contagious flag that marks a proof and all its dependencies as incomplete, ensuring that the user eventually goes back to fill in the missing proof. In this thesis, we will often use `sorry` in code examples to indicate proofs not interesting to the reader, but present in the formalization.

Usually we do not separate our statements and proof as we did for `likely`. The example below shows a slightly longer proof, using our earlier definition of `double`:

```
lemma double_is_add_self :
  ∀ a, double a = a + a :=
begin
  intro a, apply two_mul
end
```

```
lemma double_is_add_self :
    ∀ a, double a = a + a := by
  intro a
  apply two_mul
```

This can be read as a function, that takes a number `a` and emits a proof of a statement about that specific `a`. This is what it means for Lean to be dependently-typed; the type of the result of a function can depend on its input. When the inputs of a function are themselves proofs, this concept of proofs as values leads naturally to the Howard-Curry correspondence; an implication of the form $P \implies Q$ is represented as a function that converts proofs of P into proofs of Q.

The second line of this proof enters *tactic mode* using the `begin` (respectively, `by`) keyword. Once within this mode, tactics like `intro` and `apply` can be used. These tactics are often themselves Lean programs, and this mode is what makes Lean an "interactive" theorem prover. When in this mode, compatible text editors will show what is left to be proven after each tactic has been applied.

The logical foundation of Lean is the Calculus of Inductive Constructions. The following example demonstrates a typical inductive construction; an inductive type used to represent the logical or of two propositions:

```
/-- a pedagogical copy of the built-in `or` -/
inductive my_or (P Q : Prop) : Prop
| left (p : P) : my_or
| right (q : Q) : my_or
```

This reads as "there are two ways to construct a proof of $P \vee Q$; either by providing a proof of P (`my_or.left p`), or by providing a proof of Q (`my_or.right q`)". From this definition, Lean provides us with an induction principle `my_or.rec` which allows this procedure to be reversed, as "To prove[6] $P \vee Q \to R$, it is sufficient to prove $P \to R$ and $Q \to R$".

These "inductive" types are used to build almost every object needed in formalized mathematics and computer science; examples include the integers, lists, logic operators, and existential quantifiers. Sometimes we need one more fundamental building block, "quotient" types. These allow associating an equivalence relation with a type, for instance to ensure that $\frac{1}{2} = \frac{2}{4}$. Lean supports these too, and we will see more of them in section 9.3.1.

3.4.2. Revisiting the matrix examples

Armed with this new tool, let us revisit the matrix examples which caused `sympy` trouble in listing 3.7.

In listing 3.8a we use the `variables` command, which declares up-front that we are going to use the same function/theorem arguments in the rest of our examples, and we don't want to repeat them each time. Otherwise, the setup is pretty similar to listing 3.7a, except we had to

[6] Here and in Lean, $\vee$ is the logical disjunction, not the regressive product of GA.

be precise that the coefficients in our matrices are real. The use of `fin n` (the type of natural numbers less than `n`) also indicates a typical practice when doing mathematics in Lean: be as general as reasonably possible! While listing 3.8a works with `m n : ℕ` to keep the analogy with listing 3.7a, the `matrix` type is defined over arbitrary index types[7].

In listing 3.8b, we demonstrate the `example` command, which is like `lemma` but for a throwaway result. Unlike in listing 3.7b the system does not check things for us for free, but the `simp` tactic can prove it.

In listing 3.8c we see our first significant difference with listing 3.7c; we were forced to add `h : is_unit A` (i.e. assume that the matrix `A` is invertible), otherwise the statement would not have been true[8], and our attempt to find the result in the library using `library_search` would have failed.

For our final example in listing 3.8d, we find that Lean runs into the same issue as `sympy`; it also doesn't know about the "Weinstein–Aronszajn identity". It is here that the difference with `sympy` is most stark: we can teach Lean this result ourselves by providing a rigorous proof. The outline of the proof we provide is written via the `calc` tactic, and represents the chain of equalities

$$\det(I + PQ) = \det \begin{bmatrix} I & -P \\ Q & I \end{bmatrix} = \det(I + QP). \tag{3.2}$$

The `{}` braces then surround the proofs of each equality in this chain, where we invoke standard results about determinants of block matrices.

While not the main topic of this thesis, working with matrices can be a great way for readers with an engineering or software background to become familiar with Lean. To this end, the author set out to formalize the often-cited "matrix cookbook" [38], both to use as a rosetta stone to translate between informal mathematics and Lean, and as a way to evaluate the completeness of Lean's mathematics library (which we shall introduce very shortly in section 3.4.3). The result can be found at `https://github.com/eric-wieser/lean-matrix-cookbook`, and contains a mapping between the 550 numbered equations in [38], and mixture of formal statements (28% of [38]) and proofs (20% of [38]), as shown in fig. 3.9. This isn't particularly impressive coverage, but it did motivate a number of contribution to the mathematics library; notably `!![a, b; c, d]` notation for matrices [*mathlib#14991*], tactic machinery for expanding multiplications of this form [*mathlib#18711*], and results about matrix norms and exponentials (which we shall revisit in section 11.2.1).

[7] This turns out to be handy for things like the Kronecker product [*mathlib#8560*], where the matrix $M \otimes N$ can be indexed by product types, which still preserves the structure of the original indexing.

[8] Lean (or rather, mathlib) defines the inverse of a non-invertible matrix to be zero, to match the fact that in general `0⁻¹ = 0` (for which some explanation can be found in [37]).

```
variables (m n : ℕ)
variables (A B : matrix (fin n) (fin n) ℝ)
variables (P : matrix (fin m) (fin n) ℝ) (Q : matrix (fin n) (fin m) ℝ)

open matrix         -- for the `det` function
open_locale matrix  -- for `·` notation
```

(a) Initialization of variables for square ($A, B : \mathbb{R}^{n\times n}$) and rectangular ($P : \mathbb{R}^{m\times n}$, $Q : \mathbb{R}^{n\times m}$) matrices

```
example : det (A·B) = det A * det B :=
by simp
```

(b) $\det AB = \det A \det B$

```
example (hA : is_unit A) : det (A·B·A⁻¹) = det B :=
by library_search  Try this: matrix.det_conj h
```

(c) $\det ABA^{-1} = \det B$ (when A is invertible)

```
lemma matrix.det_one_plus_comm :
  det (1 + P·Q) = det (1 + Q·P) :=
begin
  calc det (1 + P·Q) = det (from_blocks 1 (-P) Q 1) : _
                 ... = det (1 + Q·P)                : _,
  { rw [det_from_blocks_one₂₂, neg_mul, sub_neg_eq_add] },
  { rw [det_from_blocks_one₁₁, mul_neg, sub_neg_eq_add] },
end
```

```
example :
  det (1 + P·Q) = det (1 + Q·P) :=
by library_search
  Try this: matrix.det_one_plus_comm
```

(d) $\det(I_m + PQ) = \det(I_n + QP)$

Listing 3.8.: Manipulating matrix determinants in Lean

After declaring some matrices in (a), the `simp` "tactic" (b) can prove some simple statements. For more complex statements like (c), we can search the library for the result with `library_search`, and interactively discover the name of the result via "Try this" messages from the editor. For the "Weinstein–Aronszajn identity" in (d), the result was not present (prior to [*mathlib#12767*]), so we have to prove it first!

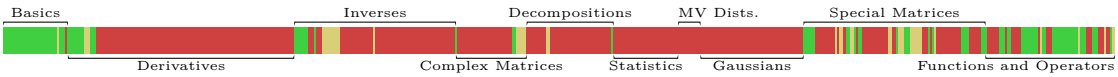


Figure 3.9.: Progress in formalizing the Matrix cookbook

Labels correspond to section titles in [38], and each vertical line is a theorem. Green boxes represent proven, yellow stated, and red unstated theorems. Much of [38] is inaccessible due to gaps in the calculus library.

3.4.3. Lean's mathematical library

By itself, Lean is very much "batteries not included". Its standard library is not opinionated on whether you use it for mathematics or software verification, and as a result comes with little beyond basic data types (e.g. `int`, `list`) and constructive logic. Notably, it does not contain proofs of statements like "the integers form a ring", nor even the definition of "ring" required to make such a statement! Indeed, we told a slight lie in listing 3.8; the example does not work in Lean by itself, as even `matrix` is not present in the standard library. This is not a short-coming in Lean—it's just a choice of division of responsibility.

The mathematically interesting statements come from mathlib, which is "a community-driven effort to build a unified library of mathematics formalized in the Lean proof assistant" [39], spanning topics like linear algebra, topology, and analysis. It is from here that our `matrix` is defined, and that the proofs for listing 3.8c was automatically found. The library is constantly growing [40], with over 300 contributors whose backgrounds are widely spread across both fields and seniority, with valuable contributions coming from undergraduates and professors alike. Perhaps a particularly notable property of mathlib is that one of its goals is to formalize the entirety of a particular undergraduate mathematics curriculum [41].

The coherency and breadth of this library makes formalization work in Lean particularly attractive, as new structures can often be built as a thin layer on top of existing structures. Furthermore, it reduces the effort needed to bring formalizations in different branches of mathematics together—for instance, in the process of formalizing statements about the wedge product in this work (section 9.3.5), the mathlib definition of the matrix determinant was shown to be an alternating map [*mathlib#5124*] (which is discussed more in section 11.1).

Lean is not unique in having a standard library of mathematics—two obvious contenders are the math-comp of Coq, and the stdlib of Agda. A thorough comparison of the three libraries is beyond the scope of this work, but a very rough estimate of breadth and accessibility to new users can be obtained by counting the lines of code, comments, and contributors, as shown in table 3.10. This estimate is a poor one, as differences between the languages themselves could easily result in the same idea taking a different number of lines to express—but it's sufficient to demonstrate that Lean is at least on-par with its competitors. For reference we also include Hol-light and Isabelle's "Archive of Formal Proofs" (AFP) in table 3.10, but as noted in the table caption these are not suited for direct comparison.

The intent of this thesis is for its formalized results about Clifford algebras to not only interface well with mathlib, but also for the majority of them to ultimately end up a part of it. Contributing code into mathlib ensures its ongoing maintenance [42], and has numerous advantages: community review by Lean experts, automated detection of bad practices by software tools, and generated documentation published to the web. Perhaps most important though is that the community as a whole takes on the responsibility of keeping our contributions compatible with the rest of

Language	Library	Source lines[1]	Comment lines[1]	Contributors[2]
Lean	mathlib	388k	100k	159
Agda	agda-stdlib	66.9k	20.4k	98
Coq	math-comp	95.9k	7.6k	35
Hol-light[3]		707k	40.4k	8
Isabelle	AFP[4]	4.26M	108k	386

[1] Counted using the `scc` program.
[2] According to `git`. Some projects do not record all authors here.
[3] The standard library and language are hard to distinguish, so the metrics include both.
[4] The "Archive of Formal Proofs" is not a standard library but a collection of libraries, and as such is larger but less interlinked.

Table 3.10.: Comparison of metrics between standard libraries of popular provers using data from their GitHub repositories as of July 2021.

mathlib as it evolves[9].

The quality bar for inclusion into mathlib is high, and as is common in software development, the review process favors lots of small contributions over one giant contribution. In practice this means it works best to develop larger formalizations outside mathlib, while in parallel continually shifting parts of their foundations back into mathlib; this allows rapid iteration unimpeded by reviewer delay, while still insuring against the formalization diverging irrecoverably from mathlib.

Throughout this thesis, the reader will find references of the form [*mathlib#4430*] or [mathlib4#3840]. These refer to contributions to mathlib, which come in the form of GitHub "pull requests". The difference between "mathlib" and "mathlib4" is largely unimportant, and simply reflects that the numbering was restarted for the move to Lean 4. The important difference is that the italicized [*mathlib#4430*] indicates that contribution was initiated[10] by the author, while the upright [mathlib4#3840] indicates a contribution initiated by another contributor.

In the rest of this section, we will introduce the elementary parts of mathlib that will be essential to formalizing Clifford algebras.

Algebraic structures

A central language feature used by mathlib in expressing algebraic structure is that of typeclasses [39, section 4], which are used to equip types with canonical operators and properties of those operators. Working with typeclasses breaks down into three parts; declaring them with the `class` keyword, consuming them with `[ ]` around a typeclass name, and providing them with the `instance` keyword.

A simple example is the typeclass `semigroup M` which equips the type `M` with the operator `*` and an associativity axiom `mul_assoc`. The first part appears as

[9] This turned out to be invaluable when the entirety of mathlib was ported to a new version of Lean in 2023; taking the contributions from this thesis with it.
[10] Most contributions are started by one author, but cleaned up in review by multiple other contributors.

```
class has_mul (G : Type*) :=
(mul : G → G → G)

infix * := has_mul.mul
```

```
class semigroup (G : Type*) extends has_mul G :=
(mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
```

With this typeclass in place, the second part allows us to write a theorem that applies to any semigroup by writing `[semigroup G]` in our argument list, as we do in `mul_assoc₂` below. This tells Lean that whenever the `mul_assoc₂` lemma is used on a type `G`, it should perform a typeclass search for a term of type `semigroup G`. In turn, it means that inside `mul_assoc₂` we have access to the `mul_assoc` axiom of semigroups on `G`.

```
lemma mul_assoc₂ {G : Type*} [semigroup G] (a b c d : G) : a * (b * (c * d)) = ((a * b) * c) * d :=
by rw [mul_assoc, mul_assoc]
```

The final piece of the puzzle is how to inform the typeclass search that `semigroup G` is available for a particular type `G`. To demonstrate this, we define a structure `mul_opposite α` that wraps a single element of an arbitrary type `α`. Using the `instance` keyword, we then equip it with a reversed multiplication structure, and express "For any type `α` such that `α` is itself a semigroup, `mul_opposite α` is also a semigroup". This method of "chaining" instances is central to the power of typeclasses, and is used extensively by mathlib in situations like equipping a product of groups with a group structure, or polynomials over a ring with a ring structure.

```
structure mul_opposite (α : Type*) := (x : α)

instance (α : Type*) [semigroup α] : semigroup (mul_opposite α) :=
{ mul := λ a b, ⟨b.x * a.x⟩,
  mul_assoc := λ a b c, congr_arg mul_opposite.mk (mul_assoc c.x b.x a.x).symm }
```

Of particular interest to formalization of geometric algebra are the `module R V` and `algebra R A` type-classes[11], which respectively describe an R-module structure on V (roughly a vector space) and an R-algebra structure on A. The former is of interest as it characterizes an arbitrary vector space over which we wish to define the geometric algebra, while the latter is a minimum requirement which our definition must meet in order to be considered useful.

Morphisms

In mathematics, it is frequently useful to talk about structure-preserving morphisms, such as an R-linear map; a function $f : A \to B$ that $f(x + y) = f(x) + f(y)$ and $\forall c : R, f(cx) = cf(x)$. In mathlib this is represented by the type `linear_map R A B` that "bundles" the map itself with proofs of the aforementioned properties. Since such bundled maps are quite common in mathlib, shorthand notation is provided. Table 3.11 summarizes mathlib's bundled maps [39, §4.1.2] and their notations used in this thesis.

[11] Which themselves require *at least* `[semiring R] [add_comm_monoid V]` and `[comm_semiring R] [semiring A]`, respectively.

Notation	Name	Properties of the map $f : A \to B$
`A →ₗ[R] B`	`linear_map R A B`	Preserves $+$, 0, and scaling by `R`
`A →ₐ[R] B`	`alg_hom R A B`	Preserves $+$, $\times$, 1, 0, and scaling by `R`
`A ≃ B`	`equiv A B`	Has an associated map $f^{-1} : B \to A$ which is a left- and right- inverse to f
`A ≃ₗ[R] B`	`linear_equiv R A B`	The properties of both `A →ₗ[R] B` and `A ≃ B`
`A ≃ₐ[R] B`	`alg_equiv R A B`	The properties of both `A →ₐ[R] B` and `A ≃ B`

Table 3.11.: Notation for homomorphisms and isomorphisms in mathlib

Part II.

Algebraic infrastructure

4

Scalar actions

> The art of doing mathematics is finding that special case that contains all the germs of generality.
>
> *(David Hilbert)*

This chapter is a significantly extended version[1] of "Scalar Actions in Lean's Mathlib" [11], notably including more recent work on right actions in section 4.7.1.

Scalar actions (a generalization of group actions) are ubiquitous in mathematics, usually appearing under the guise of multiplication; we write $x + y\mathrm{i}$ when $x, y : \mathbb{R}$ but $i : \mathbb{C}$, or $xi + yj + zk$ to scale unit vectors $i, j, k : \mathbb{R}^3$ by coefficients $x, y, z : \mathbb{R}$, or qvq^{-1} to apply a transformation represented by a quaternion $q : \mathbb{H}$ to a vector $v : \mathbb{R}^3$. Very few programming languages support implicit multiplication-by-juxtaposition like this, but many allow this kind of expression to be written using the regular multiplication operator, `*`. In programming languages for scientific computing like Python or Julia, scalar actions fall out as a special case of "broadcasting" [27, fig. 1e], and can be written with the regular multiplication operators. Sadly, Lean does not even provide the luxury of using the regular multiplication with the `*` operator, as this requires the two inputs and the output to all be of the same type[2].

mathlib's solution to these difficulties is to define a new scalar multiplication operator `•`. In this chapter we explore through examples how Lean's typeclasses are used to implement a flexible range of scalar actions, illustrate some of the problems which come up when using them such as compatibility of actions and non-definitionally-equal diamonds, and note how these problems can be solved.

[1] The original had a very short page limit.

[2] While this restriction was lifted in Lean 4, it was preserved in mathlib, and as described in section 4.8, simply opens a door to more problems.

4.1. Basic typeclasses

The typeclass[3] we are most interested in this section is `has_smul M α`, which equips a type `α` with an action by elements of `M` denoted `m • a`. Here, `smul` stands for *s*calar *mul*tiplication. In practice, this is almost always used for monoid or group actions, which are actions that satisfies the additional fields in `mul_action M α`:

```
class has_smul (M : Type*) (α : Type*) := (smul : M → α → α)

infixr ` • `:73 := has_smul.smul

class mul_action (M : Type*) (α : Type*) [monoid M] extends has_smul M α :=
(one_smul : ∀ a : α, (1 : M) • a = a)
(mul_smul : ∀ (x y : M) (a : α), (x * y) • a = x • y • a)
```

Note here that because we use `[monoid M]` instead of `extends monoid M`, we are stating that `mul_action M α` requires `M` to already be equipped with a monoid structure, rather than allowing `mul_action M α` to itself provide that structure.

mathlib extends these two typeclasses with a variety of additional axioms (i.e., fields holding proofs) for when `M` and `α` are themselves equipped with extra structures, such as distributivity over addition and actions by zero. Figure 4.1a shows the majority of these typeclasses, while details of their fields can be found either in [39, section 5.1] or in the mathlib docs.

4.2. Elementary actions

Scalar actions can be roughly divided into two types: elementary actions which are intrinsic to a particular family of types, and derived actions which operate elementwise on "bigger" types built out of smaller types. We will start by giving some examples of the former.

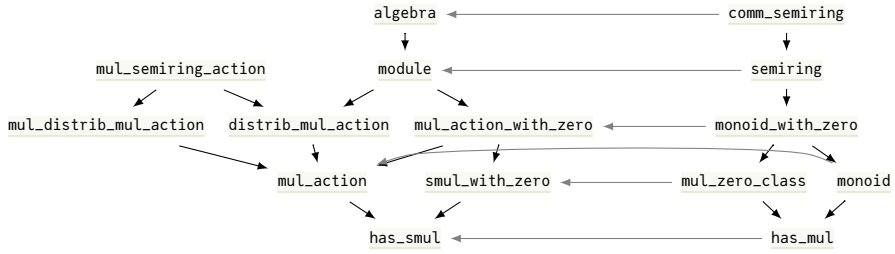
4.2.1. Left multiplication

One of the simplest actions we can construct is that of left-multiplication, with `a • b = a * b`, which mathlib provides as follows.

```
instance has_mul.to_has_smul (α : Type*) [has_mul α] : has_smul α α := { smul := (*) }
```

As the properties of the multiplication on `α` becomes stronger, so do those of this scalar action on `α`; for instance when we have `monoid α` we can deduce `mul_action α α`, and when we have `semiring α` we can deduce `module α α`. Figure 4.1b shows these available left multiplication structures, and the corresponding links with fig. 4.1a are shown with gray arrows.

[3] The mechanism introduced in section 3.4.3 for algebraic structures.



(a) Scalar action typeclasses (b) Multiplicative typeclasses

Figure 4.1.: Hierarchy of scalar action and multiplicative typeclasses

Arrows indicate implications. Grey arrows indicate implied left-multiplication actions.

4.2.2. Repeated addition and subtraction

Another simple action we can construct is that of repeated addition (an instance of `module ℕ α`) when `α` is a commutative additive monoid, which can be defined recursively for a natural number as `(0 : ℕ) • x = 0` and `∀ n : ℕ, (n + 1) • x = n • x + x`. A similar approach can be used to define a `module ℤ α` instance when `α` additionally forms an additive group. These are respectively promoted to `algebra ℕ α` and `algebra ℤ α` structures when `α` forms a `semiring` or `ring`.

4.2.3. Application of endomorphisms and automorphisms

If we have an endomorphism or automorphism `f : E` (a structure-preserving map or equivalence, respectively, from `α` to itself), then we can obtain a `mul_action E α` instance characterized by `f • x = f x`; that is, the action is just function application. Depending on the endomorphism/automorphism in question, this `mul_action` instance can be promoted to a stronger type: for instance, if it is an endomorphism of additive monoids, then this action can be promoted to `module (add_monoid.End α) α` [*mathlib4#8395*]; if it is an automorphism of R-algebras, then this action can be promoted to `mul_semiring_action (A ≃ₐ[R] A) A` [*mathlib#8724*]. In Lean, this latter instance is defined as follows, where `($)` is the function application operator.

```
instance alg_equiv.apply_mul_semiring_action : mul_semiring_action (A₁ ≃ₐ[R] A₁) A₁ :=
{ smul := ($),
  smul_one  := alg_equiv.map_one,   smul_mul := alg_equiv.map_mul,
  smul_zero := alg_equiv.map_zero,  smul_add := alg_equiv.map_add,
  one_smul  := λ _, rfl,            mul_smul := λ _ _ _, rfl }
```

Further instances were added to mathlib in [*mathlib4#8396*].

These actions act as important glue between the typeclass approach of formalizing actions that we focus on in this section, and an alternate morphism-based approach. These two approaches are shown in the columns of table 4.2. The typeclass approach is great for when a canonical action is available; but when multiple possible actions are available, such as in representation theory, the

Typeclass		Morphism
`mul_action M α`	`mul_action.to_end_hom` ⇉ ⇇ `mul_action.comp_hom`	`M →* function.End α`
`distrib_mul_action M A`	`distrib_mul_action.to_add_monoid_end` ⇉ ⇇ `distrib_mul_action.comp_hom`	`M →* add_monoid.End A`
`mul_distrib_mul_action M A`	`mul_distrib_mul_action.to_monoid_end` ⇉ ⇇ `mul_distrib_mul_action.comp_hom`	`M →* monoid.End A`
`module R M`	`module.to_monoid_end` ⇉ ⇇ `module.comp_hom`	`R →+* add_monoid.End M`
`mul_semiring_action R S`	`mul_semiring_action.to_ring_hom` ⇉ ⇇ `mul_semiring_action.comp_hom`	`R →* (S →+* S)`

Table 4.2.: Morphism- vs typeclass-based representations of actions in mathlib, and translations between them.

Note the composition implied by the `comp_hom` name is referring to composition with the endomorphism action in section 4.2.3. For the rightwards arrows, a similar family of maps is available for the *auto*morphisms.

morphism approach is more predictable. To go to an entry in the right column from a term `f` whose type is in the left column, we can build the action characterized by `m • x = f m • x`, where the second `•` is the action induced from morphism application we just described. These "actions after composing a function" are generated from the family of `comp_hom` definitions in the center column. The rightwards arrows in table 4.2 show the more direct definitions [*mathlib#8968*] that exist to go in the reverse direction.

4.3. Derived actions

A typical example of a module action might be that of a scalar $\mathbb{R}$ on the vector space $\mathbb{R}^3$ (`fin 3 → ℝ`), which multiplies each component separately. After making the obvious generalization to an arbitrary type `α` and index set `ι`, the easy way to write this down would be as follows, where again we can provide a stronger `module α (ι → α)` if we know `α` forms a `semiring`.

```
instance function.has_smul (ι α : Type*) [has_mul α] : has_smul α (ι → α) :=
{ smul := λ r v, (λ i, r * v i) }
```

This definition is perfectly fine for the action we wanted, but we can still generalize it much more. Consider now the action on matrices[4] `ι₁ → ι₂ → R` by their coefficients `R`. We would like to show `has_smul R (ι₁ → ι₂ → R)`, but that doesn't match the `function.has_smul` instance we just defined. While we could obviously define this operation trivially just as we did there, we would have to do so again if working with a vector of matrices or similar; this approach doesn't scale. Instead, we should be *deriving* an action of an arbitrary type `M` on `ι → α` from its action on `α`, such that this exploits the chaining that occurs during typeclass search.

[4] though not quite mathlib's spelling of them.

4.3.1. Function types, through their codomain

mathlib defines such a derived action on function types as follows:

```
instance function.has_smul' (ι M α : Type*) [has_smul M α] : has_smul M (ι → α) :=
{ smul := λ r v, (λ i, r • v i) }
```

This instance is strictly more general than the previous one—typeclass search will recover our original `has_smul α (ι → α)` instance by setting `M := α` and finding `has_smul α α` from `has_mul.to_has_smul`, but can also find the `has_smul R (ι₁ → ι₂ → R)` we wanted by setting `M := R` and `α := (ι₂ → R)`, and finding `has_smul R (ι₂ → α)` by recursive application of this instance.

This action propagates the axioms of the original action of `M` on `α`; we can show that if we additionally have `[module M α]`, then our action above satisfies `module M (ι → α)`, and similarly for all the other typeclasses in fig. 4.1a.

4.3.2. Sets, through their elements

On `set`s, mathlib defines a derived action via the action on the elements [mathlib#997], as

```
instance has_smul_set [has_smul α β] : has_smul α (set β) :=
{ smul := λ a s, (λ b, a • b) '' s }
```

which satisfies `a • {x, y} = {a • x, a • y}`. Once again, the axioms of the original action are propagated; though to a much lesser extent, as `0 • s = 0` (where `0` on the RHS is the set `{0}`) does not hold if `s` is the empty set. An analogous construction exists for `finset`s [mathlib#12865].

For historical reasons, these instances are not globally available by default; they must be requested locally using `open_locale pointwise`.

4.3.3. Morphisms of additive groups, through their codomain

For functions in section 4.3.1 and sets in section 4.3.1, the action we describe contains no proof obligations—we did not need to know any properties of `[has_smul M α]` to define `has_smul M (ι → α)`. This is not always the case; we cannot conclude `has_smul R (M →+ N)` from `[has_smul R N]` as we don't know enough about the action of `R` on `N` to know if additive maps remain additive. Moving away from the root node in fig. 4.1a towards stronger typeclasses is usually enough to resolve this—in this particular case, we can conclude `distrib_mul_action R (M →+ N)` from `[distrib_mul_action R N]` [*mathlib#6891*].

4.3.4. Polynomials, through their coefficients

Another simple example of a derived action is that polynomials $R[X]$ (`polynomial R` or `R[X]`) inherit an action by a type S when S acts upon their coefficients. This is a stronger statement than `has_smul R R[X]` (that polynomials are acted upon *by* their coefficients), as it generalizes in the same way as the instance we saw in section 4.3.1 to allow `R` to act on `R[X][X]` (a polynomial in two variables). Until [mathlib#4784], only this weaker statement was available.

As with the action on additive morphisms in section 4.3.3, we cannot directly conclude `has_smul S R[X]` from `has_smul S R`, this time because if we had our action on the coefficients satisfy `1 • (0 : R) = 1`, then we would end up with `1 • (0 : R[X]) = 1 + X + X^2 + ⋯` which has infinite support and thus is not a polynomial at all! Once again, we can instead start at a stronger typeclass in fig. 4.1a and provide the typeclass instance showing that `distrib_mul_action S R` implies `distrib_mul_action S R[X]` [*mathlib#7664*], as `distrib_mul_action` provides the crucial proof that `1 • (0 : R) = 0` and ensures that the scaled polynomial is well formed. This instance solves the `has_smul R R[X][X]` case by having us search for `distrib_mul_action R R[X]` and then `distrib_mul_action R R` before finally finishing the search at the instance in section 4.2.1.

Polynomials in mathlib are at the end of a chain of simpler constructions; they are defined as the special case of a "monoid algebra" whose generators are the natural numbers corresponding to the powers of X. A monoid algebra is in turn defined as a "finitely supported function" representing its coefficients; functions which are zero at all but finitely-many points. Before the upgraded `has_smul` instance could be put on polynomials [mathlib#4784], the author had to first upgrade a corresponding instance on monoid algebras [*mathlib#4365*], which in turn relied on an earlier upgrade to the instance on finitely supported functions [mathlib#284]; a sequence spanning over two years!

Multivariate polynomials are treated separately in mathlib, but the handling of scalar actions largely mirrors the univariate case; this time, it was the author's turn to perform the generalization from `has_smul R (mv_polynomial σ R)` to `has_smul S (mv_polynomial σ R)`, in [*mathlib#6533*].

4.3.5. Interactions with other actions

The strategy used for additive maps in section 4.3.3 of choosing stronger typeclasses from fig. 4.1a can only take us so far. Once we start working with types that themselves ingrain a preferred action, we need some additional tools. For instance, the closely-related types for `R`-linear maps `M →ₗ[R] N` and `R`-submodules `submodule R N` ingrain a preferred `R`-action. For the first of these cases, we can start by attempting to build a general action by an arbitrary type `α`. If we do this we find ourselves left with two proof obligations, indicated by the `show ..., from` syntax.

```
instance {α R M N : Type*}
  [semiring R] [add_comm_monoid M] [add_comm_monoid N] [has_smul α N] [module R M] [module R N] :
  has_smul α (M →ₗ[R] N) :=
{ smul := λ a f, { to_fun := λ m, a • f m,
                   map_add' := λ m₁ m₂, (congr_arg _ $ f.map_add _ _).trans $
                     show a • (f m₁ + f m₂) = a • f m₁ + a • f m₂, from sorry,
                   map_smul' := λ r m, (congr_arg _ $ f.map_smul _ _).trans $
                     show a • r • f m = r • a • f m, from sorry } }
```

The goal in `map_add'` tells us we need to strengthen `[has_smul α N]` to `[monoid α] [distrib_mul_action α N]`, just as we already would have done when building the instance in section 4.3.3.

The goal in `map_smul'` is more troublesome. The easy way out is to replace `α` with a commutative `R` so that our statement becomes

```
instance {α M N : Type*}
  [comm_semiring R] [add_comm_monoid M] [add_comm_monoid N] [module R M] [module R N] :
  has_smul R (M →ₗ[R] N) :=
```

and the `sorry` can be closed with `a • r • f m = (a * r) • f m = (r * a) • f m = r • a • f m` which follows from the axioms of `mul_action` and commutativity of `R`. Another approach would be to require `R` to be an `α`-algebra with `[algebra α R]`, and that the `α`-action on `R` and `N` is compatible with the `R`-action on `N`.

To best solve this problem, mathlib provides two additional typeclasses about scalar actions. The first expresses the compatibility condition we would need to use `[algebra α R]` as mentioned above, as

```
class is_scalar_tower (M N α : Type*) [has_smul M N] [has_smul N α] [has_smul M α] : Prop :=
(smul_assoc : ∀ (x : M) (y : N) (z : α), (x • y) • z = x • (y • z))
```

The name alludes to towers of algebras, which are described in more detail in [43, Section 4.2]. Our particular problem can be solved more directly with the second typeclass, `[smul_comm_class α R N]`, which expresses exactly the condition we require:

```
class smul_comm_class (M N α : Type*) [has_smul M α] [has_smul N α] : Prop :=
(smul_comm : ∀ (m : M) (n : N) (a : α), m • n • a = n • m • a)
```

After this typeclass was introduced in [mathlib#4770], the author contributed and drove the review of a large number of instances of it [*mathlib#6534*; *mathlib#6614*; *mathlib#8965*; *mathlib#15876*; *mathlib#10262*], most notably those for polynomials [*mathlib#6542*; *mathlib#6592*], product types [*mathlib#6139*], and the repeated addition actions in section 4.2.2 [*mathlib#5205*; *mathlib#5369*; *mathlib#5509*; *mathlib#13174*].

4.4. Algebras and not-quite algebras

The mathlib `algebra R A` describes an associative unital `R`-algebra over `A` given a `comm_semiring R` and `semiring A`. The definition is roughly

```
class algebra (R A : Type*) [comm_semiring R] [semiring A] extends has_smul R A :=
(algebra_map : R →+* A)
(commutes : ∀ r x, algebra_map r * x = x * algebra_map r)
(smul_def : ∀ r x, r • x = algebra_map r * x)
```

which states that there is a canonical ring homomorphism from `R` to `A` which agrees with `•` and sends `R` to the center of `A`. This parameterization of the axioms is difficult to generalize to A being a non-unital and non-associative ring. However, mathlib also provides a definition to construct an algebra from an alternate set of axioms:

```
def algebra.of_module (R A : Type*) [comm_semiring R] [semiring A] [module R A]
  (h₁ : ∀ (r : R) (x y : A), (r • x) * y = r • (x * y))
  (h₂ : ∀ (r : R) (x y : A), x * (r • y) = r • (x * y)) : algebra R A := sorry
```

If we look carefully, we note that `h₁` and `h₂` closely resemble `smul_assoc` and `smul_comm` from

section 4.3.5, but with some `*`s substituted for `•`s. But if we look back to section 4.2.1, we remember that when `x` and `y` are the same type, `x * y = x • y` by definition! This means that `h₁` and `h₂` correspond directly with `is_scalar_tower R A A` and `smul_comm_class R A A`, respectively.

This is a valuable insight, because it allows us to use the follow sequences of typeclass arguments interchangeably:

```
variables [comm_semiring R] [semiring A] [algebra R A]
```

```
variables [comm_semiring R] [semiring A] [module R A] [is_scalar_tower R A A] [smul_comm_class R A A]
```

Knowing this, it becomes immediately obvious how to generalize various statements to non-unital algebras (which were needed in [mathlib#7932]); we switch from from the first form to the second form, and then replace `[semiring A]` with `[non_unital_semiring A]`, something which was not permitted on the unexpanded version. Another generalization this permits is one that allows putting "most of" an `R'`-algebra structure on `A` when `R'` is only a monoid, which comes up for instance when `R' := units R`. In this case, we replace `[comm_semiring R] [semiring A] [module R A]` with `[monoid R] [semiring A] [distrib_mul_action R A]`. This generalization was used when proving intermediate results needed for Sylvester's law of inertia [mathlib#7416].

4.5. Typeclass diamonds

Frequently, there are multiple ways for Lean to construct a typeclass. One example arises when considering how the ring of module endomorphisms, `M →ₗ[R] M` with `*` as composition, acts on itself; `module (M →ₗ[R] M) (M →ₗ[R] M)`. There are two routes to find this instance, as shown in fig. 4.3; we call this situation a "typeclass diamond", in light of the shape of the figure[5]. Once route is to use the action from section 4.2.1, where `•` is just defined as `*`; this gives an action characterized by `(f • g) x = (f * g) x = f (g x)`. The other route is to combine the codomain-wise action from section 4.3 with the endomorphism action from section 4.2.3; this gives an action characterized by `(f • g) x = f • g x = f (g x)`. Depending on the order of the search, Lean could take either of these two paths; though as the result is the same (the paths "commute"), we do not care here.

4.5.1. Non-commuting diamonds

While Lean does not care about the existence of multiple paths and will happily just pick one, for the typeclass to be useful it needs to be predictable to the user—all they see is a `•` in the goal state. This means that whenever we have a diamond, we want all the paths to produce the same `•` such that the actual path taken does not matter. In this section, we shall give an example of an instance that violates this rule.

mathlib contains the following variant of the `function.has_smul'` that we saw earlier, which

[5] Although in fact not all such situations actually resemble diamonds!

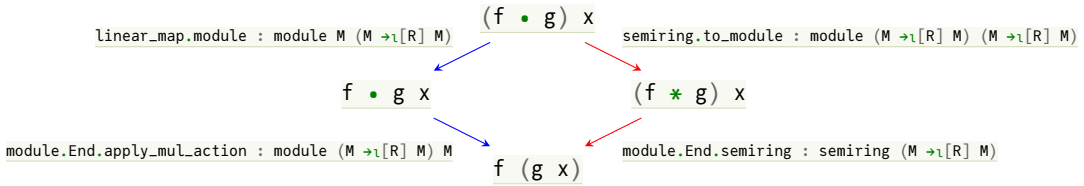


Figure 4.3.: A commuting diamond in typeclass search

Here the nodes show the expression, while the edges represent using a certain instance to populate the `•` or `*`, and point towards the definition implied by that instance.

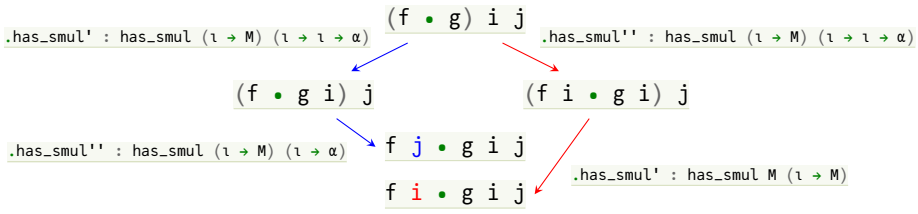


Figure 4.4.: A non-commuting diamond in typeclass search

applies scalar multiplication pointwise between two families (or vectors) of elements[6].

```
instance function.has_smul'' (ι M α : Type*) [has_smul M α] : has_smul (ι → M) (ι → α) :=
{ smul := λ r v, (λ i, r i • v i) }
```

Mathematically this is a very reasonable operation, but in the context of typeclass diamonds we shall see it is not. In particular, consider a typeclass search for `has_smul (ι → M) (ι → ι → α)` in the presence of `has_smul M α`, as shown in fig. 4.4. Here, we find that one path gives `(f • g) i j = f i • g i j`, while the other gives `(f • g) i j = f j • g i j`, where the argument to `f` is different. We say these paths are not "propositionally equal" (as it can be proved in most cases[7] that they do not agree), and call this instance diamond "non-commuting".

4.5.2. Definitional equality

There is a reason that section 4.5.1 specifically refers to issues with *propositional* equality in typeclass resolution; dependent type theory leaves us with another important kind of equality, *definitional* equality, which holds only if things are true by construction.

The following example, which shows that a family of additive maps are a module under scaling by natural numbers, is vulnerable to issues around definitional equality.

```
example {ι A B} [add_comm_monoid A] [add_comm_monoid B] : module ℕ (ι → A →+ B) := by apply_instance
```

The possible typeclass-resolution paths available are shown in fig. 4.5.

[6] In some sense, providing a one-sided "broadcasting" multiplication like that found in [27]; though only for when the right array has more dimensions than the left.

[7] The exception being when `subsingleton ι` (i.e. when there is only a single inhabitant of `ι`, and so `i = j`) or similar!

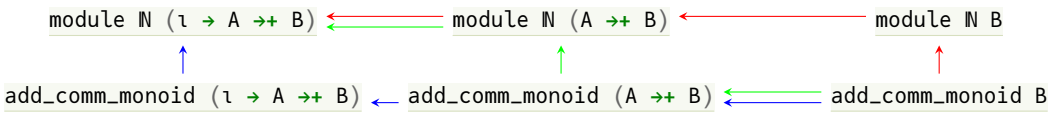


Figure 4.5.: Compounding diamonds in typeclass search

Three possible paths to resolve `module ℕ (ι → A ×+ B)`, with arrows showing implications. Diamonds are created by the choice between inheriting a module structure (horizontal edges), or deriving it from the additive structure (vertical edges).

From the viewpoint of *propositional* equality, we are safe from non-commuting paths; it can be proven that `∀ (M : Type*) [add_comm_monoid M], subsingleton (module ℕ M)` (that is that all ℕ-module structures are equal), and thus we can conclude the paths must be equal from the type of their endpoint alone. To a user of Lean, this means that they can be confident that the `•` carries the right mathematical meaning.

When it comes to applying lemmas about `•`, it is not sufficient that `•` carry the right mathematical meaning; the `•` in the lemma statement needs to unify with the `•` in the target. In practice, this means that the instances found for each need to be *definitionally* equal, otherwise users are left with baffling error messages about how `n • x` does not match `n • x`.

In older versions of mathlib, the paths taken around the diamonds in fig. 4.5 resulted in instances that were propositionally equal, but not definitionally equal, as shown in fig. 4.6. The underlying reason was that the recursor `@nat.rec` for natural numbers (which underpins the `^[n]` notation in fig. 4.6) does not commute definitionally with lambda introduction `λ a,`; that is, the following example fails:

```
example {α β} (f₀ : α → β) (f : α → ℕ → β → β) (n : ℕ) :
  @nat.rec (λ n, α → β) (λ a, f₀ a) (λ n ih a, f a n (ih a)) n =
  λ a, @nat.rec (λ n, β) (f₀ a) (λ n ih, f a n ih) n :=
rfl
```

This meant that lemmas about the natural ℕ-action (blue path, fig. 4.5) such as `∑ x in s, c = s.card • c` would fail to match goals containing a derived ℕ-action (green and red paths, fig. 4.5). This was fixed in [mathlib#7084] by requiring the definition of `add_comm_monoid M` to include an implementation of the ℕ-module structure; namely, a new `nsmul` field, and a proof that it coincides propositionally with the naïve recursive implementation.

While mathematically it is bizarre to say "a commutative additive monoid has a zero, addition, *and a scalar-multiplication by naturals*, such that …", in Lean this is crucial to allow manual control of definitional equality such that the green and blue paths in fig. 4.5 can be made definitionally equal to the red path. This is analogous to the situation described in [39, section 4.1] for topologies associated with metric spaces, and follows the "forgetful inheritance" pattern described in [44]. Further discussion of this `nsmul` field can be found in [45, §7].

A similar place in which this comes up is when constructing `algebra ℕ S`, `algebra ℤ R`, and `algebra ℚ K` instances for a semiring, ring, or characteristic-zero division ring, respectively.

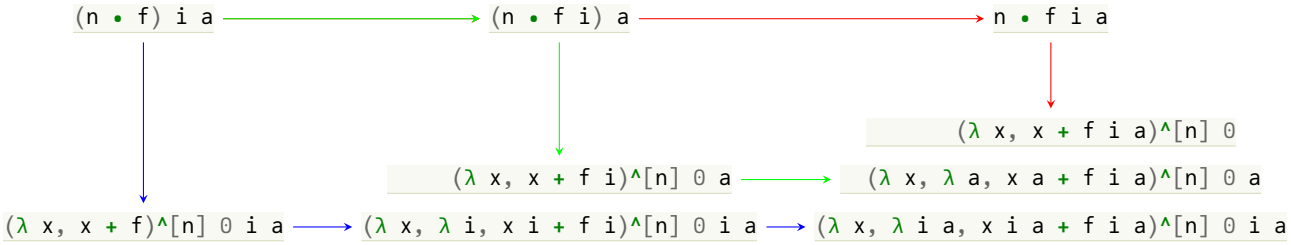


Figure 4.6.: Non-commuting diamonds in repeated addition actions

> Prior to [mathlib#7084], the expressions shown here are the ones found through the paths in fig. 4.5, where the `g^[n] x` operation is iterated function application, `g^[2] x = g (g x)`. The arrows mean "unfolds to" as they did in figs. 4.3 and 4.4.

In each case, the type of the instance is a subsingleton and so instance paths can be seen trivially to commute propositionally. The danger arises when constructing the `algebra_map` fields; the "obvious" way is to do so recursively, by recursing structurally on `n : ℕ`/`z : ℤ`/`q : ℚ` and setting `algebra_map _ _ (n + 1) = algebra_map _ _ n + 1`, etc. This approach not only fails on a very similar example to fig. 4.5, but also fails in the case when `S = ℕ`/`R = ℤ`/`K = ℚ`, as the identity function is certainly not equal by definition to such a recursive scheme. The solution in [mathlib#12182; mathlib#14894] was also similar; adding `of_nat`/`of_int`/`of_rat` field to the `semiring`/`ring`/`division_ring` typeclasses[8] and proof fields demonstrating that they are well-behaved.

This is not the last time we shall have to concern ourselves with definitional equality; we shall see the relevance of it again in sections 6.3.1 and 7.3.2.

4.6. Conjugation, via type synonyms

While section 4.5 gives some more involved examples of how derived actions can induce conflicting definitions, we can obtain conflicting actions much more simply. Consider for instance the ways in which a group can act on itself: both $g \cdot h = gh$ and $g \cdot h = ghg^{-1}$ are reasonable actions, but we clearly can't use the same notation for both without causing confusion[9], and so only the former (section 4.2.1) gets the privilege of the `has_smul G G` typeclass.

This is of no help to us if we really want to work with the latter conjugation action[10]. In some cases we can avoid this by working with `mul_distrib_mul_action G H` for arbitrary groups `G` and `H` instead, which is a more abstract representation of actions (like conjugation) which distribute over multiplication. Even then, this abstraction only postpones the inevitable; it cannot be specialized to the concrete case of $g \cdot h = ghg^{-1}$ until we solve the original issue.

[8] Or rather, suitable ancestors of these classes, for reasons related to non-associative algebras.
[9] Lean may be happy for us to overload the notation in this way, but the humans writing proofs who see this notation in their goal are unlikely to be!
[10] Which for instance we may want to do when formalizing the transformation by rotors in section 2.1.3.

The solution used by mathlib is that of "type synonyms", which can be written as either of the following forms

```
-- nominal types
@[deriving group]
def conj_act (G : Type*) : Type* :=
  G

def of_conj_act : conj_act G ≃* G :=
  mul_equiv.refl _

def to_conj_act : G ≃* conj_act G :=
  of_conj_act.symm
```

```
-- one-field structures
structure conj_act (G : Type*) : Type* :=
  to_conj_act :: (of_conj_act : G)

instance [has_mul G] : has_mul (conj_act G) :=
{ mul := λ g h, to_conj_act (g.of_conj_act * h.of_conj_act) }

instance [Group G] : group (conj_act G) :=
function.injective.group to_conj_act
  sorry sorry sorry sorry sorry sorry sorry
```

where in both cases, we define a new type `conj_act G` which is a copy of `G` with the same group structure, and a pair of functions `to_conj_act` and `of_conj_act` to translate from `G` to `conj_act G` and back. The trade-offs between the two approaches are not particularly relevant to this chapter[11], and in this particular case mathlib opts for the left approach.

So far, the type synonym has achieved nothing; it behaves in exactly the same way as `G`! To give it purpose, we give it a special action on `G` that is the conjugation action

```
instance : has_smul (conj_act G) G :=
{ smul := λ cg h, of_conj_act cg * h * (of_conj_act cg)⁻¹ }
```

where we convert from `conj_act G` back to `G` before implementing the expected expression. This allows us to write the conjugation of `g` on `h` as `to_conj_act g • h`, and prove the various axioms of the stronger typeclasses in fig. 4.1a. The synonym is placed on the type doing the acting (`has_smul (conj_act G) G`) rather than the type being acted upon (`has_smul G (conj_act G)`) as this permits us to use both the multiplication and conjugation action simultaneously[12] (via different spellings).

The verbosity of this spelling might make it seem unappealing; if a spelling this long is acceptable, it would be tempting to conclude we could have just used the spelling `conj g h` where `conj : G →* monoid.End G` is the appropriate morphism taken from the right column of table 4.2. The reason we avoid making this choice is that it would exclude the conjugation action from incorporation into the derived instances in section 4.3. One action in particular is of interest here; the pointwise action [*mathlib#8945*] (matching section 4.3.2) induced by elements `g : G` on a subgroup `S : subgroup H` when `mul_distrib_mul_action G H`. Combined with the conjugation action we just saw, this gives us the usual conjugation action of an element on a subgroup! The type synonym in this section was added in [mathlib#8627] by Chris Hughes, in response to the author's review of Chris' previous approach in an early version of [mathlib#8592], which aimed to define only the conjugation action.

[11] They come down to the left version being easier to misuse (since Lean is willing to confuse the types `G` and `conj_act G` in some situations), and the right version being much more work to set up (since Lean is *not* ever willing to confuse the types and we must therefore rebuild the group structure from scratch, hence all the `sorry`s). If used correctly, the two approaches are in practice equivalent to the end user.

[12] Though in fact there is a third option, explored in section 4.9.

4.7. Right actions

The scalar action typeclass `has_smul` in mathlib is intended for left actions (those where $a(bX) = (ab)X$), which is apparent both in the definition of the `mul_smul` axiom of `mul_action`, and in the order in which the arguments appear in the notation. However, this does not mean that right actions (those where $(Xb)a = X(ba)$) are impossible.

The trick is to use another type synonym (section 4.6) from mathlib, `mul_opposite α`, which is built similarly to the pedagogical example in section 3.4.3 and which reverses the multiplication order. With this in mind, we can require a right action by writing `[mul_action (mul_opposite M) α]`, which permits us to write `op a • X` as a messy spelling for a right action on `X` by `a`.

The author introduced mathlib's first right action in [*mathlib#7630*], via the instance

```
instance monoid.to_opposite_mul_action [monoid α] : mul_action (mul_opposite α) α :=
{ smul := λ c x, x * c.unop,
  one_smul := mul_one,
  mul_smul := λ x y r, (mul_assoc _ _ _).symm }

lemma op_smul_eq_mul [monoid α] {a b : α} : op a • b = b * a := rfl
```

which mirrors the left-multiplication action in section 4.2.1. Similar instances were introduced for the other stronger typeclasses in fig. 4.1a.

One big advantage of this design over introducing a new `right_mul_action` type is that the vast majority of the derived actions from section 4.3 come for free: as an example, `mul_action (mul_opposite α) (ι → α)` is found automatically and corresponds to the action `(op a • f) i = op a • f i = f i * a`. This is precisely the action we want, if for example we want to multiply a family of quaternions (or indeed, any object with non-commutative multiplication) by a constant on the right.

4.7.1. Bimodules

Often, we wish to consider simultaneous right and left actions, such as an `R`-`S`-bimodule `M`, where `R` acts on the left of `M`, and `S` on the right. Crucially, these actions must be compatible; $(rm)s = r(ms)$. Mathematically, this looks like associativity, and so we might hope to capture it using the `smul_assoc` from `is_scalar_tower`. In actuality, to Lean the statement is `r • op s • m = op s • r • m`, and so this is commutativity! We can thus capture the structure of the bimodule `M` as follows:

```
variables [module R M] [module (mul_opposite S) M] [smul_comm_class R (mul_opposite S) M]
```

When `R` is commutative, we typically don't bother with talking about left and right actions, as they are usually equivalent[13]. However, it would be a bad idea to make `module R M` imply `module (mul_opposite R) M`, as even though the commutativity of `R` means we need not worry about

[13] and if they weren't, mathlib would typically force the use of a type synonym (section 4.6) to declare the less-canonical action.

propositionally equal typeclass diamonds arising, we are almost guaranteed to run into definitional ones. The solution was to introduce an `is_central_scalar R m` typeclass [*mathlib#10543*], which instead of producing a new instance and risking diamond issues, simply asserts that two existing instances interact in the desired way; namely that `op_smul_eq_smul : op r • m = r • m`.

Adding this typeclass is just the tip of the iceberg; the real work is to saturate mathlib with instances of this typeclass. [*mathlib#10543*] claimed the low-hanging fruit, providing instances for commutative monoids, product types (`M × N`, `Π i, M i`), finitely supported functions (`ι →₀ M`, `Π₀ i : ι, M i`), `ULift M`, polynomials and their generalizations (`monoid_algebra R M`, `add_monoid_algebra R M`, `R[X]`, `mv_polynomial σ R`), `matrix m n M`, morphisms (`M →+ N`, `M →ₗ[R] N`), complex numbers, and pointwise instances[14] (`set M`, `submonoid M`, `add_submonoid M`, `subgroup M`, `add_subgroup M`, `subsemiring M`, `subring M`, `submodule M`). Further contributions [*mathlib#10720*; *mathlib#11291*; *mathlib#11297*; *mathlib#12248*; *mathlib#12272*; *mathlib#12434*; *mathlib#13710*; *mathlib#15359*; mathlib#18682] brought the total up to more than 60 `is_central_scalar` instances across mathlib.

4.7.2. Interaction with `algebra`

The addition of all these `is_central_scalar` instances was not simply motivated by completeness: it was a prerequisite for solving a larger issue, the fact that we want every `algebra R A` to automatically be a left- and right- `R`-module. Without this inference happening automatically, we find that the act of "generalizing" results in mathlib to bimodules (an example of which we will see much later in section 10.1.5) results in them no longer applying to algebras!

We already saw in section 4.4 that mathlib knows that an `algebra R A` implies a left `module R A` structure, and forces it to agree with left-multiplication. To make it also imply a right-module (`module (mul_opposite R) A`) structure, we need it to carry an additional `to_has_op_smul : has_smul (mul_opposite R) A` field (the right action), and a proof that it coincides with *right*-multiplication, `op_smul_def' x r : op r • x = x * to_fun r`. Such a refactor is a rather herculean task; there are at least 130 instances of `algebra` in mathlib[15], and adding fields to `algebra` requires every instance to provide values for these fields. To make matters worse, mathlib is a moving (and growing) target, and once you think you're almost done, you merge in the latest changes from other contributors and have even more `algebra` instances to fix!

The exercise of adding a large number of `is_central_scalar` instances was a means to slowly close the gap; the act of adding these results often involved adding associated `has_smul (mul_opposite R) A` instances, getting half the work out of the way. Additionally, the `op_smul_eq_smul` lemma provided by these instances paves a quick path to proving `op_smul_def'`. Getting these intermediate results merged into mathlib before the full project was a way of offloading the work of keeping up: while results are only in your local modifications, it falls primarily on you to deal with conflicts and proof breakages arising from others' changes; once results are in

[14] elementwise actions on the elements of a set or sub-object.
[15] According to the generated documentation.

mathlib, the responsibility transfers to the community at large. As it turned out, this approach of contributing early and often had further benefits; the author's prototype of a complete refactor in [*mathlib#10716*] was "caught in the porting tide", which is to say that the translation ("port") from Lean 3 to Lean 4 (which starting from the simplest files and rose up through the import hierarchy) caught up with the files it modified, forcing it to be abandoned. The same fate did not befall the `is_central_scalar` instances, as these were already merged before the port began.

As discussed by the author in [*mathlib#7152*], this refactor introduces further complications almost exactly analogous to the ones we saw in fig. 4.5. If every algebra implies a right action, then we need $\mathbb{N}$-algebras to imply right-$\mathbb{N}$-actions. We saw in section 4.5.2 that to make this work without definitional typeclass diamonds for left-actions, we needed to add an `nsmul` field to additive monoids (and `zsmul` to additive groups); for right-actions, we need to do the same for `op_nsmul` and `op_zsmul`, making the mathlib `add_comm_group` even further from the expected mathematical definition!

4.7.3. Other compatibility concerns

In section 4.3.5, we saw how when working even just with left-actions, needs arise for compatibility typeclasses like `is_scalar_tower` and `smul_comm_class`. In section 4.7.1, we saw that the latter can be repurposed to provide a compatibility between left and right actions. However, there are other common interactions of left and right actions for which mathlib has no compatibility typeclass.

Introducing briefly for clarity the notation[16] `a •> b` for `a • b` and `b <• a` for `op a • b`, there is no typeclass in mathlib capable of expressing `(a <• b) •> c = a •> (b •> c)`. Some examples of when this situation arises are `[monoid M] (a b c : M)` (all three variables belong to the same non-commutative monoid), `[monoid M] (a c : M) (S : submonoid M) (b : S)` (the second belongs to a submonoid of the monoid containing the other two), and `[monoid M] (a b : M) (c : ι → M)` (the third variable is a coordinate vector).

4.7.4. On functions, through their domains

In section 4.6, we remarked that sometime the left action we want is not the one by left-multiplication, and that to resolve this we must introduce a type synonym. The same issue arises for right actions. A particularly typical example is the right action on function types (or in general, morphisms) that acts through their domain; the action where for $f : G \to G$, $g, h : G$, we define the right action fg such that $(fg)(h) = f(gh)$. This is problematic, because the derived action in section 4.3.1 gives us the action where $(fg)(h) = f(h)g$, which when combined with the instance we wanted produces a non-commuting instance diamond.

Type synonyms again provide a solution here; in [mathlib4#5368], after some discussion with the author, Yury Kudryashov introduces a `DomMulAct M` type synonym which induces precisely

[16] Which is proposed for inclusion in mathlib in [*mathlib4#8909*].

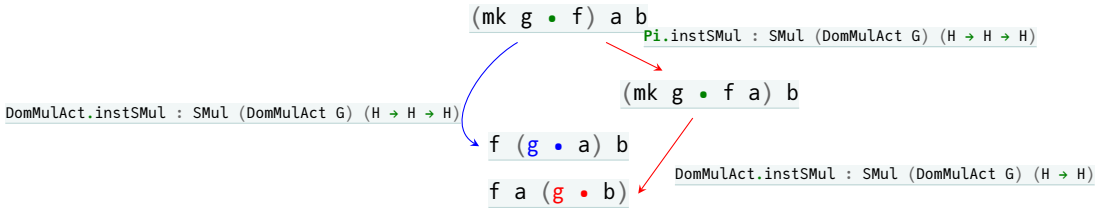


Figure 4.7.: A non-commuting diamond caused by `DomMulAct`

> Here we have an action `SMul G H`, variables `g : G`, `a b : H` and `f : H → H → H`. The ambiguity arises through choosing which argument of `f` to act upon.

this $(fg)(h) = f(gh)$ action, which Lean characterizes as:

```
example [SMul M α] (c : M) (f : α → β) (a : α) : (mk c • f) a = f (c • a) := rfl
```

which for our example specializes to

```
example [Group G] (g : G) (f : H → H) (h : H) : (mk g • f) a = f (g * h) := rfl
```

With this synonym, Lean cannot take a wrong turn towards using the instance in section 4.3.1 with $(fg)(h) = f(h)g$, as there is no action of `SMul (DomMulAct G) G` available.

Unfortunately, this solution only postpones the diamonds to one step further down the road. When faced with an action on a function taking two arguments, Lean now faces an ambiguity about which of the two domains the action should act upon (thanks to the action in section 4.3.1), as shown in fig. 4.7. This isn't quite as bad as the diamond in fig. 4.4, as it only impacts users of `DomMulAct`; but it is indicative that type synonyms are not a silver bullet.

We could resolve this by removing the `Pi.instSMul` instance in section 4.3.1 that formed one of the offending edges, but this would prevent writing `r • ![x, y, z]` to scale a vector of coefficients. A compromise is available through the use of even more type synonyms; we could demote the instance for actions through the codomain to a `CodMulAct` synonym, and then the two actions in fig. 4.7 would be spelled as `(DomMulAct.mk g • f) a b = f (g * a) b` and `(CodMulAct .mk (DomMulAct.mk g) • f) a b = f a (g * b)`, and `r • ![x, y, z]` would need to be written as `CodMulAct.mk r • ![x, y, z]`. The verbosity could be somewhat reduced by introducing some shorthand notation, which for these three examples could resemble `g •D f`, `g •CD f`, and `r •C ![x, y, z]`.

Such a compromise would involve a substantial refactor to mathlib, and for now the costs of the instance diamond in fig. 4.7 do not in the author's opinion outweigh the effort involved in *performing* the refactor, let alone the negative impact of the increased verbosity.

4.8. Lean 4's new `HMul` typeclass

Lean 4 generalizes the meaning of the `*` syntax, allowing it to be used to multiply elements of different types. This is done by means of a new `HMul α β γ` typeclass that provides the operator

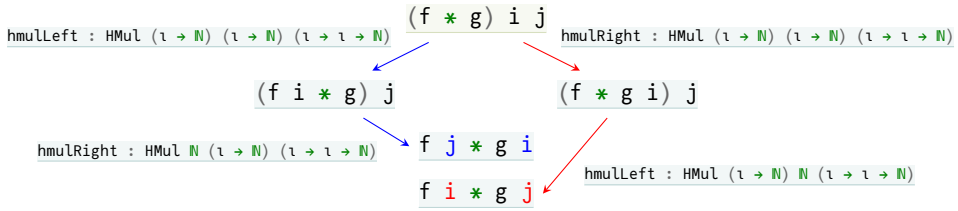


Figure 4.8.: A non-commuting diamond caused by `HMul` for `f g : ι → ℕ`

`*` through the function `HMul.hMul : α → β → γ`, where the `H` stands for "heterogeneous". This generalization is very useful for multiplication of matrices (which mathlib switched to using in [*mathlib4#6487*]), as here multiplication is naturally heterogeneous in the dimensions of the matrices; we have `HMul.hMul : Matrix l m R → Matrix m n R → Matrix l n R`.

At first sight, the introduction of `HMul` would appear to make `has_scalar.smul : α → β → β` redundant, as it falls out as a special case. Similarly, we could even see `HMul` as the answer to the right actions in section 4.7, since we can also recover `β → α → β` as a special case. Even `smul_comm_class` and `is_scalar_tower` could likely be subsumed by a typeclass for generalized heterogeneous commutativity or associativity, respectively.

Unfortunately, for our use-case the flexibility of `HMul` is also its downfall; attempting to build basic left- and right- actions for "vectors" (as in section 4.3) using it almost immediately leads to instance diamonds in the style of section 4.5.1. The two basic instances in question are:

```
-- HMul generalizes to families on the right
instance hmulRight [inst : HMul α β γ] : HMul α (ι → β) (ι → γ) where
  hMul a b := fun i => a * b i

-- HMul generalizes to families on the left
instance hmulLeft [inst : HMul α β γ] : HMul (ι → α) β (ι → γ) where
  hMul a b := fun i => a i * b
```

and the diamond is formed when querying for `HMul (ι → ℕ) (ι → ℕ) (ι → ι → ℕ)`, as shown in fig. 4.8. The following code can be used to verify the diagram is correct.

```
variable (ι)
abbrev blue : HMul (ι → ℕ) (ι → ℕ) (ι → ι → ℕ) := hmulLeft (inst := hmulRight)
abbrev red  : HMul (ι → ℕ) (ι → ℕ) (ι → ι → ℕ) := hmulRight (inst := hmulLeft)

-- the two paths give conflicting results, swapping the placements of `i` and `j`
example (f g : ι → ℕ) (i j : ι) : letI := blue ι; (f * g) i j = f i * g j := rfl
example (f g : ι → ℕ) (i j : ι) : letI := red ι;  (f * g) i j = f j * g i := rfl
```

Strictly speaking, our problem is not that `HMul` is too general, but that `hmulLeft` and `hmulRight` are. We can resolve this by eliminating `γ`, and using repeated type variables to distinguish right and left actions:

```
-- right-HMul generalizes to families on the right
instance hmulRight [inst : HMul α β β] : HMul α (ι → β) (ι → β) where
  hMul a b := fun i => a * b i

-- left-HMul generalizes to families on the left
instance hmulLeft [inst : HMul α β α] : HMul (ι → α) β (ι → α) where
  hMul a b := fun i => a i * b
```

At this point though, we may as well have defined `LeftSMul α β := HMul α β β` and `RightSMul α β := HMul α β α`, an approach which is not all that different from the current mathlib approach which is effectively using `LeftSMul α β := SMul α β` and `RightSMul α β := SMul (MulOpposite α) β`. The only real difference is the choice of symbol, `*` vs `•`.

4.9. Alternatives to type synonyms

In sections 4.6 and 4.7.4 we explored how mathlib uses `DomMulAct` and `ConjAct` synonyms that wraps the type doing the acting, in order to distinguish these action from other "more canonical" actions. However, this comes at the cost of introducing some annoying boilerplate in the form of `DomMulAct.mk` and `ConjAct.toConjAct`, which we use to juggle between `DomMulAct G`, `ConjAct G`, and `G`. While section 4.7.4 suggests that the pain of this boilerplate can be reduced with clever notation, it still ends up being something that has to be manipulated within proofs. This section briefly outlines an alternative design that has no prior use in mathlib.

A possible redesign of the `SMul` typeclass could be

```
def SMul.Discr := Type
class SMul (M : Type*) (α : Type*) (discr : SMul.Discr) where
  smul : M → α → α
notation3:72 a:72 " •[" discr "] " b:72 => @SMul.smul _ _ discr _ a b
```

where the `discr` parameter acts as a discriminator to implement "tag dispatching" (to borrow the term from C++) and contains information relevant only at typeclass-search-time about *which* action to use; information that was previously tracked by attaching it to type synonyms wrapping `M` and `α`. The `discr` parameter would be copied to all the other typeclasses in fig. 4.1a.

The actions in sections 4.2.1, 4.3.1 and 4.7.4 can then respectively be written

```
inductive SMul.Discr.leftMul : SMul.Discr
instance (M : Type*) [Mul M] : SMul M M .leftMul where
  smul m n := m * n

inductive SMul.Discr.domAct (_ : SMul.Discr) : SMul.Discr
instance (ι α M : Type*) (discr) [SMul α ι discr] : SMul α (ι → M) discr.domAct where
  smul a f := fun i => f (a •[discr] i)

inductive SMul.Discr.codAct (_ : SMul.Discr) : SMul.Discr
instance (ι α M : Type*) (discr) [SMul α M discr] : SMul α (ι → M) discr.codAct where
  smul a f := fun i => a •[discr] (f i)
```

where the `domAct` and `codAct` discriminators are parameterized to record the discriminator they inherit from.

With this infrastructure in place, the diamonds in fig. 4.7 are avoided by forcing the user to spell which action they want:

```
variable {G H} (discr) [SMul G H discr] (g : G) (f : H → H → H) (a b : H)
example : (g •[discr.domAct] f) a b = f (g •[discr] a) b := rfl
example : (g •[discr.domAct.codAct] f) a b = f a (g •[discr] b) := rfl
```

This strategy is not without its downsides. It makes cases where the typeclass diamonds do commute (such as fig. 4.3) more awkward to work with, requiring an additional compatibility typeclass that states that two discriminators describe the same action; Additionally, bundled linear maps would now need to take two extra arguments to specify the discriminator for their source and domain. Determining whether these trade-offs are acceptable would require an attempt at refactoring large pieces of mathlib, which the author will leave to a particularly interested reader.

4.10. Summary

This chapter described in great detail the design of the `•` ("smul") operator in mathlib, and the typeclass infrastructure around it; which the author was heavily involved in the design of. Of particular interest is the discussion on right actions (section 4.7), which are relatively new to mathlib.

Many of the results in later chapters will require this infrastructure to even *state* their formalizations: especially section 10.1.5, where right actions turn out to be instrumental for working with the dual quaternions, and section 10.2, where a tangled web of algebraic towers (section 4.3.5) must be traversed.

The author is fortunate that for their use-cases, almost all actions are "canonical". As a result, the concerns of diamonds in typeclass search will not be of much interest in part III, though we shall still find we have to think about them occasionally. This sadly is not a universal experience for mathlib users, and further exploration of the ideas in section 4.9 may alleviate these diamond-based difficulties.

5

Extensionality

> One of the endearing things about mathematicians is the extent to which they will go to avoid doing any real work.
>
> *(Matthew Pordage)*

This chapter was, after writing this thesis, adapted into [12].

A vital tactic to many of the formalizations in this thesis is the `ext` tactic [mathlib#104]. In its most basic form, it reduces equalities of functions (`f = g`) into equality at every evaluation (`∀ x, f x = g x`), and equalities of sets (`s = t`) into equivalence of membership in each set (`∀ x, x ∈ s ↔ x ∈ t`). The tactic is extensible; new scenarios can be enabled by adding an `@[ext]` attribute to a theorem, for instance to add support for finite sets analogous to the support for sets:

```
@[ext] theorem finset.ext {α} {s t : finset α} (h : ∀ x, x ∈ s ↔ x ∈ t) : s = t :=
  sorry
```

We will call such theorems "extensionality lemmas". Configuration for morphisms (such as linear maps) and sub-objects (such as subspaces) can be added in a similar way without much difficulty, such as:

Theorem 5.1. *For a commutative ring R and a pair of R-modules M, N, to show two R-linear maps $f, g : M \to_R N$ are equal, it suffices to show that they agree everywhere; $\forall m, f(m) = g(m)$.*

In some cases, we can write a more specialized extensionality lemma. One particularly useful example is

Theorem 5.2. *For a commutative ring R and an R-module M, to show two R-linear maps $f, g : R \to_R M$ are equal, it suffices to show that they agree on 1; $f(1) = g(1)$.*

For a more interesting example, let us consider linear maps from the tensor product of two modules, for which the natural statement of extensionality is

```
variables {R M N : Type*}
variables [comm_semiring R] [add_comm_monoid M] [add_comm_monoid N] [module R M] [module R N]
```

```
theorem tensor_product.comm_symm :
  (tensor_product.comm R M N).symm
    = tensor_product.comm R N M :=
begin
  ext mn,
  show (comm R M N).symm nm = comm R N M nm,
  induction nm using tensor_product.induction_on
    with n m x y hx hy,
  { -- the case with `nm = 0`
    simp only [map_zero] },
  { refl }, -- the case with `nm = n ⊗ₜ m`
  { -- the case when `nm = x + y` and we have
    -- `hx : (comm R M N).symm x = comm R N M x`
    -- `hy : (comm R M N).symm y = comm R N M y`
    simp only [hx, hy, map_add] }
end
```

(a) Using only `linear_map.ext`, theorem 5.1

```
theorem tensor_product.comm_symm :
  (tensor_product.comm R M N).symm
    = tensor_product.comm R N M :=
begin
  ext n m,
  show
    (comm R M N).symm (n ⊗ₜ m) = comm R N M (n ⊗ₜ m),
  refl -- true by definition!
end
```

(b) Using `tensor_product.ext`, theorem 5.3

Listing 5.1.: Two proofs showing that the natural braiding of the tensor product is symmetric
Using theorem 5.3 in (b) results in a much simpler argument than (a).

Theorem 5.3. *For a commutative ring R and a trio of R-modules M, N, P, to show two R-linear maps $f, g : (M \otimes_R N) \to_R P$ are equal, it suffices to show that they agree on the pure tensors; $\forall m, \forall n, f(m \otimes n) = g(m \otimes n)$.*

Due to its weaker assumption, this is a stronger statement than the extensionality lemma for linear maps in theorem 5.1. We can write theorem 5.3 in Lean as follows:

```
theorem tensor_product.ext {R M N P : Type*}
  [comm_semiring R] [add_comm_monoid M] [add_comm_monoid N] [add_comm_monoid P]
  [module R M] [module R N] [module R P]
  {f g : (M ⊗[R] N) →ₗ[R] P}
  (H : ∀ (m : M) (n : N), f (m ⊗ₜ n) = g (m ⊗ₜ n)) : f = g :=
sorry
```

This a much more useful lemma than the one that requires `H : ∀ (mn : M ⊗[R] N), f mn = g mn`, as it saves us from having to split `mn` into pure tensors ourselves. To see this benefit, we can work through a proof that `(tensor_product.comm R M N).symm = tensor_product.comm R N M`; that is, the natural braiding of the tensor product that on the pure tensors sends $m \otimes n \mapsto n \otimes m$ is symmetric. Listing 5.1 compares the formalization of such a proof with and without theorem 5.3. Without the assistance of theorem 5.3, we are forced to induct on the structure of the tensor product, and end up with two additional subgoals that we'd prefer not to think about.

5.1. Chaining extensionality lemmas

However, even theorem 5.3 still only scratches the surface of the power behind the `ext` tactic. Where it really excels is in its ability to *chain* extensionality lemmas. A simple example of this is reducing equalities of two-argument functions into showing they agree when fully-applied (`∀ x y, f x y = g x y`), but there are far more interesting cases. In particular, we shall explore how the `ext` tactic can be chained on equalities of morphisms, specifically linear maps and algebra morphisms. The key insight that massively boosts the power of `ext` is the fact that turning an equality of morphisms into an equality of its evaluations should be a last resort! This may seem surprising, since in the simple examples it seemed like the raison d'être of `ext` was to introduce these `∀ x` quantifiers; but there are frequently far better approaches.

To better understand this insight, we start with a warning of what happens if we overlook it, by examining the following extension of theorem 5.3:

Theorem 5.4. *For a commutative ring R and a quadruplet of R-modules M, N, P, Q, to show two R-linear maps $f_3, g_3 : ((M \otimes_R N) \otimes_R P) \to_R Q$ are equal, it suffices to show that they agree on the pure tensors; $\forall m, \forall n, \forall p, f((m \otimes n) \otimes p) = g((m \otimes n) \otimes p)$.*

The statement of this extensionality lemma raises an immediate red flag; it suggests that we are doomed to state a new theorem for every possible arity and associativity of tensor products[1] (and of course to prove each of them!). We can try to lessen this blow by proving theorem 5.4 in terms of theorem 5.3, but it only gets us halfway; we are left to prove that $\forall x_{mn} : M \otimes N, \forall p, f(x_{mn} \otimes p) = g(x_{mn} \otimes p)$, where we have successfully taken apart only one of the two tensor products, and are once again forced to induct upon the structure of x_{mn}.

Our trouble here is that theorem 5.3 is too weak; it cannot be chained with itself, because it consumes an equality of elements not an equality of morphisms. To correct this, we state it as in theorem 5.5:

Theorem 5.5. *For a commutative ring R and a trio of R-modules (M, N, P), to show two R-linear maps $f, g : (M \otimes_R N) \to_R P$ are equal, it suffices to show that they agree* when composed with *the canonical bilinear map $(\otimes) : M \to_R N \to_R (M \otimes_R N)$; $f \circ_2 (\otimes) = g \circ_2 (\otimes)$. This equality is an equality of bilinear maps (linear maps with a codomain that is itself a linear map) of type $M \to_R N \to_R P$.*

The proof of theorem 5.5 follows immediately from that of theorem 5.3; indeed, applying `ext` turns the former into the latter! In Lean, this condition is written `(tensor_product.mk R M N) .compr₂ f = (tensor_product.mk R M N).compr₂ g`, where `b.compr₂ f` (or $f \circ_2 b$) is the bilinear map such that `b.compr₂ f m n = f (b m n)`.

We now arrive at our key conclusion; theorem 5.4 can be proven using iterated applications of theorem 5.5. The approach is shown in fig. 5.2, where T is theorem 5.5 and L is theorem 5.1. It is here where our earlier remark that "turning an equality of morphisms into an equality of its

[1] Indeed, proving that vector spaces form a monoidal category requires two different associativities of the 4-ary version.

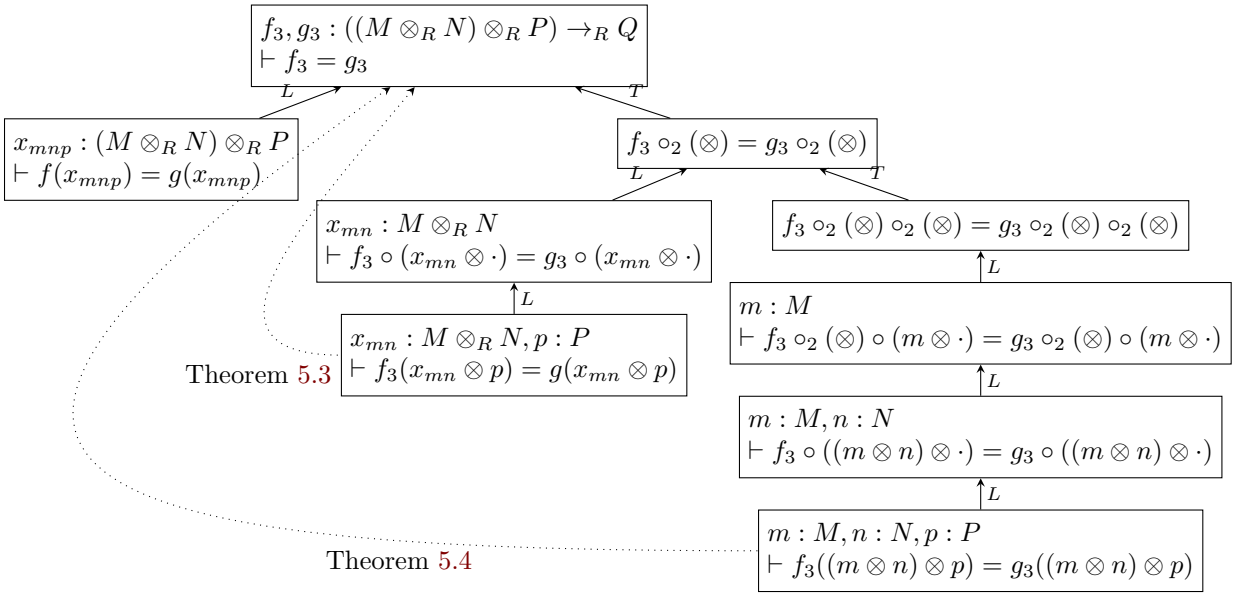


Figure 5.2.: A factorization of theorems 5.3 and 5.4 into theorem 5.5 (T) and theorem 5.1 (L)

Arrows show implications. Branching indicates that *either* child is sufficient, not that both are necessary. Indeed, extensionality between tensor products of any arity of associativity factors through T and L.

evaluations should be a last resort" comes into play; the most structured (and thus easiest to prove) statement is reached by preferring T edges over L edges, as taking an L edge prematurely leads to a dead end. The `ext` tactic can handle this graph traversal automatically; by setting the T edges to have higher priority, they will be attempted first. It can be seen that in fact, variants of theorem 5.4 for *any* arity of associativity of linear maps from tensor products can be tackled in the same way; T edges will always split the left-most tensor product, and L edges will consume non-tensor products from the left.

5.2. Wider applications

The benefits of this strategy (which we call "partially-applied `ext` lemmas") extends far beyond tensor products; there are numerous other situations where we can apply them:

- To show two linear maps $(M \oplus N) \to_R P$ from binary direct sums of modules agree, it suffices to show that they agree when composed with $\mathrm{inl} : M \to_R M \oplus N := m \mapsto (m, 0)$ and $\mathrm{inr} : N \to_R M \oplus N := n \mapsto (0, n)$; $f \circ \mathrm{inl} = g \circ \mathrm{inl}$ and $f \circ \mathrm{inr} = g \circ \mathrm{inr}$.
- To show two linear maps $f, g : (\bigoplus_i M_i) \to_R N$ from n-ary direct sums of modules agree, it suffices to show that they agree when composed with every canonical injection into the ith component $\iota_i : M_i \to \bigoplus_i M_i$; $\forall i, f \circ \iota_i = g \circ \iota_i$.

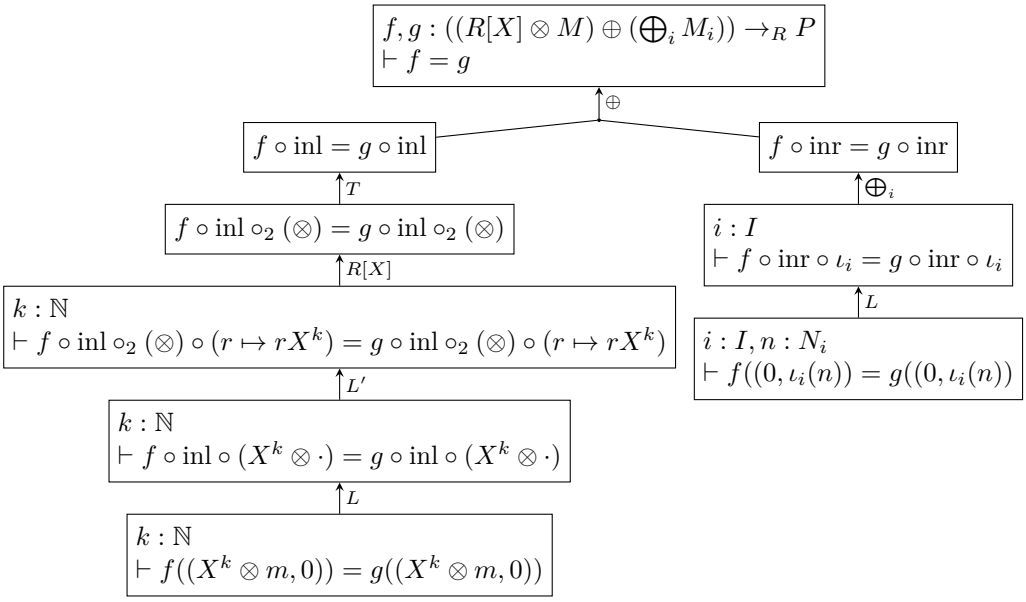


Figure 5.3.: Extensionality for a linear map from an arbitrarily-chosen compound type.

Here, the edges labelled $R[X]$, $\oplus$, and $\bigoplus_i$ are the extensionality lemmas listed in section 5.2, L' is theorem 5.2, and L and T are the same as in fig. 5.2.

- To show two linear maps from polynomials $f, g : R[X] \to_R M$ agree, it suffices to show they agree when composed with each of the maps that scale the kth power of X; $\forall n, f \circ (r \mapsto rX^k) = g \circ (r \mapsto rX^k)$.

Arguably the cases that these examples apply to are all just different special cases of free modules, but to mathlib they are genuinely different objects, and so we must teach `ext` about each of them separately. Similar families of extensionality lemmas exist for special cases (or quotients) of free monoids (and monoid morphisms), free rings (and ring morphisms), and notably free algebras (and algebra morphisms). We shall see many examples of the algebra case throughout this thesis. Crucially, within these families, the "partially-applied" lemmas are mutually compatible; as each makes the minimal amount of progress and avoids applying L (theorem 5.1). For instance, if faced with a pair of maps $f, g : ((R[X] \otimes M) \oplus (\bigoplus_i N_i)) \to_R P$, then `ext` will leave us to prove $\forall k, \forall m, f((X^k \otimes m, 0)) = g((X^k \otimes m, 0))$ and $\forall i, \forall n, f((0, \iota_i(n))) = g((0, \iota_i(n)))$, as shown in fig. 5.3.

We will find this compatibility across types to be very useful in chapter 10, for which a particularly compelling example is shown in fig. 10.2.

5.3. As a motivation for point-free statements

So far, we have seen how extensionality lemmas can be designed to greatly aid the task of proving equalities between morphisms from complex types. Unfortunately, most equalities we face are between the objects, and we rarely face these equalities of morphisms unless we deliberately frame our problems in a certain way.

A simple example of adjusting our statements to make such an equality appear arises when building an isomorphism of modules $M \cong_R N$ from the forward and inverse maps $f : M \to_R N$ and $f^{-1} : N \to_R M$. The conventional way to construct the isomorphism would be to show that these morphisms are left and right inverses through $\forall m, g(f(m)) = m$ and $\forall n, f(g(n)) = n$; which are equalities of objects, not morphisms. If we instead re-frame our statements to $g \circ f = \mathrm{id}$ and $f \circ g = \mathrm{id}$, we are faced with equalities of morphisms that we can apply `ext` to. mathlib already contains many definitions that assemble isomorphisms in this way. Once again turning to tensor products as an illustrative example, if we have $M \coloneqq M_1 \otimes_R M_2$ and $N \coloneqq N_1 \otimes_R N_2$, then this approach of showing that the composition is the identity, combined with the `ext` tactic, allows us to prove f and g are inverses by considering only $\forall m_1, \forall m_2, g(f(m_1 \otimes m_2)) = m_1 \otimes m_2$ and $\forall n_1, \forall n_2, f(g(n_1 \otimes n_2)) = n_1 \otimes n_2$.

More generally, when faced with an equality of objects in terms of two functions of a free variable, we can reduce our problem to an equality of morphisms by pushing that variable all the way to an application on the right. For instance, if we want to show that $\forall x, x \times y = y \times x$ (where $\times$ is an arbitrary bilinear operation), we can:

- rewrite as $\forall x, (\cdot \times y)(x) = (y \times \cdot)(x)$, where the free variable x is now on the right on both sides;
- note that $(\cdot \times y)$ and $(y \times \cdot)$ describe linear maps
- conclude that it would be sufficient to prove $(\cdot \times y) = (y \times \cdot)$, which is an equality of morphisms

This allows us to apply any extensionality lemmas that replace x with the more restricted structured values (such as pure tensors). We can then repeat to put y on the right, and apply more relevant extensionality lemmas. A particularly common situation where this trick is useful is for putting multiplicative structures upon new algebras, such as upon tensor products, tensor powers, and direct sums. We shall see more concrete examples of this in sections 6.3.3 and 10.3.2.

As it turns out, there is a much more concise way of presenting this reasoning for this example; we can show outright[2] that $\times$ is a bilinear map, and write our goal with all the free variables on the right as $(\times)(x, y) = \mathrm{flip}(\times)(x, y)$, where flip turns one bilinear map into another. We conclude that it is sufficient to prove that $(\times) = \mathrm{flip}(\times)$, an equality of bilinear maps on which we can then apply all our extensionality lemmas in one go.

[2] A fact that was pointed out by Greg Price, who tidied the author's proofs in [mathlib#7029].

In general, this style of writing functions with no free variables is called "point-free", and is a fairly common trick in function programming languages[3]. To give a more complex example, the function $x \mapsto y \mapsto f(x) \times g(y)$ can be written in a point-free style as either $\text{flip}(\text{flip}(\times) \circ g) \circ f$ or as $\text{flip}(\circ)(g) \circ (\times) \circ f$. What sets our situation apart from the typical point-free approach is that we are not simply constructing functions, but morphisms that preserve algebraic operators. As a result, we need an extensive library of composition operators that are themselves morphisms; which for linear maps, ties heavily back to the infrastructure described in section 4.3.5.

5.4. Summary

This short chapter summarized the state of the `ext` tactic in mathlib, and explained how "partially-applied ext-lemmas" can provide significant value; especially when combined with careful construction of point-free statements,

The author cannot claim credit for the `ext` tactic, nor for the idea of using "partially-applied ext lemmas", but is responsible for extending this pattern through many relevant types in mathlib. The first sign of such a lemma in mathlib that the author is aware of was added by Chris Hughes in [mathlib#3408] for the semidirect product, though it was not tagged with `@[ext]`. Chris Hughes was also responsible for suggesting a similar lemma for the tensor algebra in review of [mathlib#3531], though once again it was never tagged `@[ext]`. Scott Morrison appears to have realized the value of the `@[ext]` attribute when generalizing the construction of the tensor algebra to a quotient of the free algebra in [mathlib#4078]. The first explicit mention of chaining that the author can find was by Yury Kudryashov, who noted in [mathlib#4741] that it was useful for working with free modules and algebras.

The author is responsible for documenting the pattern in [*mathlib#5484*], and for contributing extensionality lemmas for:

- algebra morphisms from the exterior algebra in [*mathlib#4297*]
- algebra morphisms from the Clifford algebra in [*mathlib#4430*]
- linear maps from n-ary direct sums in [*mathlib#4821*]
- ring morphisms from $\mathbb{Z}[\sqrt{d}]$ in [*mathlib#5640*]
- linear maps from tensor products in [*mathlib#6105*] (theorem 5.5)
- linear maps from binary direct sums in [*mathlib#6124*]
- algebra morphisms from the complex numbers in [*mathlib#8105*]
- morphisms from quotient constructions in [*mathlib#8641*]: quotients by subgroups, submodules, lie submodules, and ideals
- algebra morphisms from graded algebras in [*mathlib#8783*]
- algebra morphisms from the dual numbers in [*mathlib#10730*]
- algebra morphisms from the trivial square-zero extension in [*mathlib#10754*]
- linear maps from the exterior algebra in [*mathlib#14803*]

[3]For instance, the Haskell Wiki has a page on it at https://wiki.haskell.org/Pointfree.

- algebra morphisms from the tensor product of algebras in [*mathlib4#6417*]
- algebra morphisms from polynomials over an algebra in [*mathlib4#8116*]

The `ext` tactic will be used extensively in many of the constructions in this thesis. Many of the results in the list above will appear in more detail in part III.

6

Graded rings

> Equality may perhaps be a right, but no power on earth can ever turn it into a fact.
>
> *(Honoré de Balzac)*

This chapter adapts "Graded Rings in Lean's Dependent Type Theory" [2], which was joint work with Jujian Zhang.

The vast majority of the text was written by the author, though any remarks about the Proj construction in algebraic geometry are entirely thanks to Jujian. The author was largely responsible for the design of the typeclasses described in this section [*mathlib#6053*; *mathlib#8783*; *mathlib#9586*], though Jujian contributed a typeclass from section 6.5 [mathlib#10115]. The results about tensor and Clifford algebras were contributed by the author, while Jujian takes credit for the results about polynomials and algebraic geometry[1].

In principle, dependent type theory should provide an ideal foundation for formalizing graded rings, where each grade can be of a different type. However, the power of these foundations leaves a plethora of choices for how to proceed with such a formalization. This chapter explores various different approaches to how formalization could proceed, and then demonstrates precisely how the author and Jujian Zhang formalized graded algebras in Lean's mathlib. Notably, we show how this formalization was used as an API; allowing us to formalize various graded structures such as those on tuples, free monoids, tensor algebras, and Clifford algebras.

6.1. Introduction

One way to introduce graded rings and algebras is by noting that they generalize an early staple in mathematics education; that of single-variate polynomials in X. Any polynomial can be written

[1] not to be confused with geometric algebra!

as a (finite) weighted sum of powers of X, and multiplication only requires the knowledge that $X^m X^n = X^{m+n}$.

If we define the $\mathbb{N}$-indexed family of homogeneous polynomials $A = (i \mapsto \{aX^i \mid a : R\})$, then we can say "the polynomials in a ring R over X, $R[X]$, are an algebra graded by A"[2]; by which we mean:

1. Each of the elements of the family A_i are closed under addition and scalar multiplication by elements of R.
2. There is a $1 \in A_0$.
3. For any $p \in A_i$ and $q \in A_j$, we have $pq \in A_{i+j}$. Equivalently, as sets $A_i A_j \subseteq A_{i+j}$.
4. Every element p can be expressed uniquely as $p = \sum_i p_i$ where $p_i \in A_i$.

The above acts as a general definition of an algebra graded by some arbitrary family of submodules A, which can in general be indexed by any additive monoid ι, not just the natural numbers.

To build some intuition for this generalization, it is worth enumerating some other examples:

Multivariate polynomials, $R[X, Y, \dots]$. Over two variables we can grade either by the $\mathbb{N}$-indexed family of elements of homogeneous degree $A = (i \mapsto \{aX^j Y^{i-j} \mid a : R, j \leq i\})$ where X^3 and XY^2 e.t.c. have the same grade, or by an $\mathbb{N} \oplus \mathbb{N}$-indexed family on the individual variables, $A = ((i, j) \mapsto \{aX^i Y^j \mid a : R\})$.

The tensor algebra, $\mathcal{T}(V)$. Conventionally we grade this by the $\mathbb{N}$-indexed family where A_i spans the i^{th} tensor powers $V^{\otimes i}$.

The exterior algebra, $\bigwedge(V)$. The exterior algebra is graded in exactly the same way, but when V is of dimension n we find that A_i for $n < i$ is the trivial submodule.

The Clifford algebra, $\mathcal{C}\ell(V, Q)$[3]. We cannot[4] use exactly the same approach for the Clifford algebra, as for a vector v, we have $v^2 = Q(v)$, where the LHS would be of grade 2 and the RHS would be of grade 0. This can be resolved by having just two grades; one corresponding to sums of "even" monomials (those which are a product of an even number of elements of V), and one corresponding to sums of odd monomials. Phrased another way, the family is indexed by[5] $\mathbb{Z}/2\mathbb{Z}$, the integers modulo two.

Any ring α. Any ring can be equipped with the trivial grading structure, where the index type contains only one element 0 corresponding to the entire ring.

As this is a thesis about formalization, we will predictably proceed by finding all the different ways to "pull legs off"[6] this definition. By relaxing items 2 and 3, we can talk about gradings of

[2] Or "a graded algebra of type $\mathbb{N}$ over the ring R with graduation A" in the language of [46, III, §3, 1.].
[3] Where $\mathcal{C}\ell(V, Q)$ is notation to specify the quadratic form Q and vector space V.
[4] At least, when $Q \neq 0$. If $Q = 0$ then $\mathcal{C}\ell(V, Q) = \bigwedge(V)$ and we can proceed as above.
[5] Note that literature referring to an $\mathbb{N}$-grading is referring to the grading on $\bigwedge(V)$ via the canonical module equivalence.
[6] https://en.wikipedia.org/wiki/Centipede_mathematics

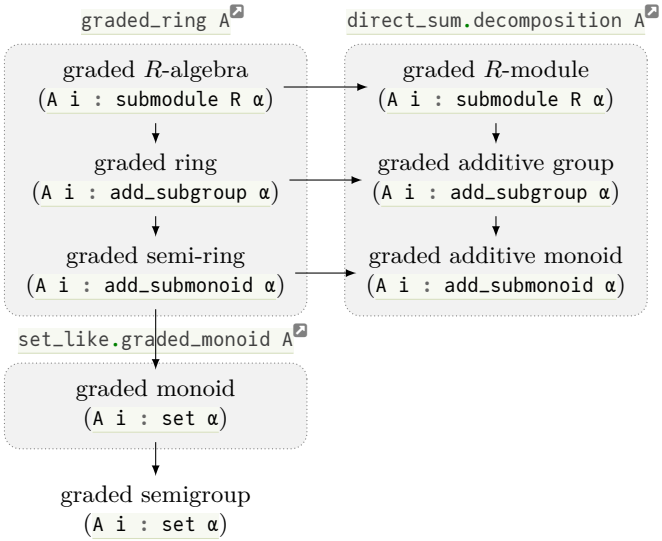


Figure 6.1.: The algebraic hierarchy of graded objects discussed in this chapter.

The meanings of the typeclasses introduced in section 6.4 for grading `α` internally by `A` are shown as labelled gray regions, with the objects representing each internal grade (that is, the type of A_i) shown in parentheses.

additive monoids, additive groups ([46, II, §11, 1.]), and R-modules[7]. By relaxing item 1, we can additionally talk about gradings by families of additive subgroups, additive submonoids, or even just sets; which we refer to as graded rings, graded semirings, and graded monoids respectively. For graded monoids (such as the n-tuples α^n or tensor powers $M^{\otimes n}$) there is no summation, so item 4 is interpreted as the statement that p must belong to exactly one A_i. Figure 6.1 outlines the connection between these various generalizations.

While the existence of many examples following the same pattern is already a good reason to formalize that pattern, it is only half the picture; just as important is to have situations where the generalization itself is necessary. For instance, without a formalization of commutative additive monoids, we can't even define what it means to take the sum of a finite set, and would instead be forced to repeat this definition for $\mathbb{N}$, $\mathbb{Z}$, etc. A particularly motivating example for need a formalization of graded rings is that of the $\operatorname{Proj} S$ construction in algebraic geometry [47, Tag 01M3], a definition which requires a notion of homogeneous ideals, which in turn requires precisely the notion of graded rings this chapter is about. Another recent motivation appeared in the "Liquid Tensor Experiment" [48], which required a proof of Gordan's lemma; one proof of which goes via graded rings[8].

[7] Some sources use a more general definition of graded R-modules and R-algebras, where R is itself a graded ring such that $R_i M_j \subseteq M_{i+j}$. For brevity, we will not discuss these here (in essence considering only the special case when R has the trivial graduation), but our approach would extend to this straightforwardly.

[8] In the absence of our work a proof via convexity was used instead.

6.2. Prior formalizations

In Coq, [49, p. 3.] refers to graded modules as `nat -> FreeModule R` in the context of homology. No mention of graded multiplicative structures appears. A similar formalization of graded modules exists for Lean 2 in [50, `algebra/graded.hlean`], although the language has changed substantially since then, in particular dropping the experimental HoTT mode which [50] builds upon.

In Agda, [51] shows that the axioms of a commutative graded ring are satisfied by a particular construction, the graded cohomology groups. No attempt seems to be made to provide a general definition of what it means for an object to satisfy those axioms.

Extensive discussion about formalizing "Commutative Differential Graded algebras", algebraic objects with some additional axioms on top of the graded algebras discussed in this chapter, has taken place on the Lean Zulip Chat [52]. While these discussions refer to a current version of Lean 3 [6], the ideas explored in [52] never resolved into a contribution to Lean's monolithic mathematics library mathlib [39]. It is by this metric that [49; 50] are substantially different in scope to this work; in those formalizations, a definition was chosen for the particular use case of interest to the authors, with little regard for interoperability with large amounts of existing code. Conversely, one of the reasons the work in [52] never made it into mathlib is likely the lack of applications to verify that the design is the right one. Since the closure of that thread, mathlib has grown by a factor of five in terms of total lines, and has gained formalizations of many new objects of interest to us: tensor algebras, Clifford algebras, and tensors powers.

6.3. External gradings

There are two ways to think about a grading in dependent type theory; either as a family of sets of a single type (internal), or as an indexed family of distinct types. There are merits to both approaches; which is most useful depends on whether it is more natural to define the single type then break it into pieces (as with the monoid under concatenation of `list`s graded by their `length`), or to define the family of types then glue them together (as with the monoid under concatenation of the tuples `fin n → α` graded by `n`). A crucial factor in the coherence of mathlib as a unified library is its ability to translate between multiple ways of stating the same thing, so we do not want to have to choose between these approaches in an exclusive manner. Thankfully, the former approach can be represented via the latter; an internal grading can be written as an external grading over the family of subtypes corresponding to each grade, shown in parentheses in fig. 6.1. We will revisit this equivalence in section 6.4.

It is worth remembering that when building externally-graded objects in this way, that the grades are disjoint by definition. If for example our indexed family of types is `λ i : ℕ, R` (a family indexed by the naturals, all equal to the same ring), then this is viewed as a countable sequence of *copies* of `R`, which makes this construction exactly analogous to the single-variate polynomials.

6.3.1. Graded semigroups

Let us now try to develop the framework for talking about externally-graded semigroups[9] over a family of types `A` indexed by an additive semigroup `ι`. We would like to be able to express these via Lean's "typeclasses", as this matches how the usual non-graded algebraic structure is expressed. This means that to talk about a graded monoid, a user might write:

```
def sq {ι : Type*} {A : ι → Type*} [add_semigroup ι] [g_semigroup A]
  (i : ι) (x : A i) : A (i + i) :=
g_semigroup.mul x x
```

To explain this syntax briefly; `sq` names the definition, `{name : type}` and `(name : type)` introduce implicit and explicit variables, `[type]` introduces a typeclass variable, the trailing `:` prefixes the result type, and `:=` prefixes the value of the definition. Typeclass variables are special; the `[add_semigroup ι]` variable is used to define the meaning of `i + i` via mathlib's algebra framework, while the `g_semigroup.mul` would be defined by the `[g_semigroup A]` variable. A user calling this `sq` function might write `sq _ x`, where `_` acts as a wildcard which Lean works out automatically by looking at `x`. The same mechanism is used to infer the implicit `ι` and `A` arguments, but typeclass search is used to populate the two arguments in square brackets; for instance, if `ι := ℕ` then Lean finds `nat.add_semigroup : add_semigroup ℕ`. More thorough introductions to typeclasses in Lean and mathlib can be found in [45, §2] and chapter 4.

Attempting to define a new `g_semigroup A` typeclass by directly writing down item 3 and a suitable associativity axiom as

```
variables {ι : Type*} (A : ι → Type*)

class g_semigroup [add_semigroup ι] :=
(mul {i j} : A i → A j → A (i + j))
(mul_assoc {i j k} (x : A i) (y : A j) (z : A k) :
    mul (mul x y) z = mul x (mul y z))
```

leads to the error "term `mul x (mul y z)` has type `A (i + (j + k))` but is expected to have type `A ((i + j) + k)`". While these are "obviously" equal, that's not enough for Lean; for the statement to type-check, we need the types to be *definitionally* equal. We would have similar problems with `A (i + 0)` and `A i` if we were trying to prove `mul x one = x` for a graded monoid. To escape this problem, we could:

1. Use heterogeneous equality (denoted `==`), which allows us to express equality between distinct types:

   ```
   class g_semigroup [add_semigroup ι] :=
   (mul {i j : ι} : A i → A j → A (i + j))
   (mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
     mul (mul x y) z == mul x (mul y z))
   ```

2. Express the equality in terms of sigma types or dependent pairs, denoted `Σ i, A i`:

[9] Chosen for brevity due to having the fewest axioms, not because they are interesting.

```
class g_semigroup [add_semigroup ι] :=
(mul {i j : ι} : A i → A j → A (i + j))
(mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
  (⟨_, mul (mul x y) z⟩ : Σ i, A i) = ⟨_, mul x (mul y z)⟩)
```

3. Express the grading constraint as an equality on sigma types:

```
class g_semigroup [add_semigroup ι] extends semigroup (Σ i, A i) :=
(fst_mul {i j : ι} (x : A i) (y : A j) :
  (⟨_, x⟩ * ⟨_, y⟩ : Σ i, A i).fst = i + j)
```

4. Provide an explicit proof that the equality is type correct using the recursor for equality, `eq.rec`:

```
class g_semigroup [add_semigroup ι] :=
(mul {i j : ι} : A i → A j → A (i + j))
(mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
  (add_assoc i j k).rec (mul (mul x y) z) = mul x (mul y z))
```

5. Store a canonical map between objects of the "same" grade to use instead of using `eq.rec`, to allow better definitional control:

```
class g_semigroup [add_semigroup ι] :=
(cast {i j : ι} (h : i = j) : A i → A j)
(cast_rfl {i} (x : A i) : cast rfl x = x)
(mul {i j : ι} : A i → A j → A (i + j))
(mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
  cast (add_assoc i j k) (mul (mul x y) z) = mul x (mul y z))
```

6. Take an additional index into `mul` and a proof that it is equal to `i + j`:

```
class g_semigroup [add_semigroup ι] :=
(mul {i j k : ι} (h : i + j = k) : A i → A j → A k)
(mul_assoc {i j k ij jk ijk : ι}
  (hij : i + j = ij) (hjk : j + k = jk)
  (hi_jk : i + jk = ijk) (hij_k : ij + k = ijk)
  (x : A i) (y : A j) (z : A k) :
    (mul hij_k (mul hij x y) z) = mul hi_jk x (mul hjk y z))
```

Many of these options are derived from the discussions in [52]. When deciding between these options, we need to consider both the ease of providing an instance with `instance : g_semigroup A`, and the ease of consumer working with one using `[g_semigroup A]`. Note that it is straightforward to expose different interfaces to the consumer and producer, and provide a layer of translation in between. This is especially the case when the interfaces differ only in their statement of the propositional fields. Table 6.2 outlines a rough comparison between these approaches.

Taking a step back from this problem, we also need to decide on a spelling for the consumer, as writing `mul` instead of a multiplication symbol is hardly pleasant. There are essentially two options

Approach	item 1 `==`	item 2 `Σ i, A i`	item 3 `extends`	item 4 `eq.rec`	item 5 `cast`	item 6 `h : i+j=k`
`h : (i+j)+k=i+(j+k)` needed by	producer	producer	—	—	—	consumer
Estimated difficulty for the producer	medium	harder	easier	harder	easier	medium
Directions of consumer `rw` tactic use	0	2	2	1	1	2

Table 6.2.: Merits of the various approaches to defining `g_semigroup`

"Producer" refers to the code providing the `instance : g_semigroup A`, while "consumer" refers to the code with a `[g_semigroup A]` argument.

here: either introduce new notation for our graded `mul`, or hook into the existing `*` notation. The latter is a far more appealing option, as it means we can reuse all the lemmas we have about `*` by providing the appropriate algebraic typeclasses. The only catch is that the existing `*` notation requires the operation to be homogeneous; acting on a single type, rather than three elements of a family[10].

To achieve this homogenization, we can use the built-in `sigma` type of dependent pairs, storing the grade of the monoid alongside the value at that grade such that `x : A i` is represented by `sigma.mk i x : Σ i, A i`.

```
instance g_semigroup.to_semigroup [add_semigroup ι] [g_semigroup A] :
  semigroup (Σ i, A i) :=
{ mul := λ (x y : Σ i, A i), ⟨x.fst + y.fst, g_semigroup.mul x.snd y.snd,
  mul_assoc := λ (x y : Σ i, A i), sorry }
```

If we choose item 3 from table 6.2, then this code is generated for us automatically! However, the fact that the grade of the multiplication is known only propositionally and not definitionally can make things harder in section 6.3.3, so we avoid this choice. As the next most appealing option, choosing item 2 makes the `sorry` above[11] fall out immediately, and so this is what mathlib does. This decision is far from final, but the best way to compare the option of table 6.2 is to thoroughly implement one of them, and then come back and see whether changing the definition to something different makes the existing proofs better or worse.

6.3.2. Graded monoids

In reality, we do not define `gsemigroup` at all in mathlib, and jump straight to graded monoids due to lack of need for the former. We also don't actually put the instance on the `sigma` type, as this would not be a sufficiently canonical choice to be worthy of a global instance. Instead, we define `graded_monoid A` as an alias for `sigma A`, and place the instances on that. By splitting apart the typeclasses a little and interleaving the instances for `graded_monoid A`:

[10] Lean 4 lifts this notation restriction, but the algebraic typeclasses provided by mathlib would need reworking.
[11] The syntax in Lean for an incomplete proof.

```
class ghas_one [has_zero ι] := (one : A 0)
class ghas_mul [has_add ι] := (mul {i j} : A i → A j → A (i + j))

instance ghas_one.to_has_one [has_zero ι] [ghas_one A] :
  has_one (graded_monoid A) := { one := ⟨_, ghas_one.one⟩ }
instance ghas_mul.to_has_mul [has_add ι] [ghas_mul A] :
  has_mul (graded_monoid A) := { mul := λ x y, ⟨_, ghas_mul.mul x.snd y.snd⟩ }
```

we make the notation for the instance much more pleasant

```
class gmonoid [add_semigroup ι] extends ghas_one A, ghas_mul A :=
(one_mul (a : graded_monoid A) : 1 * a = a)
(mul_one (a : graded_monoid A) : a * 1 = a)
(mul_assoc (a b c : graded_monoid A) : a * b * c = a * (b * c))
```

It is worth remembering that while this may look identical to the definition of a regular monoid, it is constraining the grade-preserving behavior of the multiplication by construction. As is always the case with formalization, it is never quite as simple as you would hope it would be. In fact, the definition of `gmonoid` in `mathlib` contains three additional fields!

```
(gnpow : Π (n : ℕ) {i}, A i → A (n • i))
(gnpow_zero' : Π (a : graded_monoid A), graded_monoid.mk _ (gnpow 0 a.snd) = 1)
(gnpow_succ' : Π (n : ℕ) (a : graded_monoid A),
  (graded_monoid.mk _ $ gnpow n.succ a.snd) = a * ⟨_, gnpow n a.snd⟩)
```

These describe the power operator by the natural numbers, and ensure that its grade too is known definitionally following the "forgetful inheritance" [44] pattern, the relevance of which is explored in section 4.5.2.

Example 6.1 (the n-tuples)**.** *This typeclass allows us to express the graded monoid structure of the n-tuples under concatenation, as*

```
instance : gmonoid (λ n : ℕ, fin n → α) :=
{ one := ![], -- the empty tuple
  mul := λ i j a b, fin.add_cases a b,
  ..sorry /- boring proofs -/ }
```

6.3.3. Graded (semi)rings

For graded rings, we do not run into any new equality problems, as addition remains within a grade. To state the requirement for a family of types to represent a graded ring, we can simply extend the monoid structure from earlier:

```
class gsemiring [add_monoid ι] [Π i, add_comm_monoid (A i)]
  extends gmonoid A :=
(mul_zero : ∀ {i j} (a : A i), mul a (0 : A j) = 0)
(zero_mul : ∀ {i j} (b : A j), mul (0 : A i) b = 0)
(mul_add : ∀ {i j} (a : A i) (b c : A j), mul a (b + c) = mul a b + mul a c)
(add_mul : ∀ {i j} (a b : A i) (c : A j), mul (a + b) c = mul a c + mul b c)
-- For "forgetful inheritance" like the previous `gnpow` field
(nat_cast : ℕ → A 0) (nat_cast_zero : sorry) (nat_cast_succ : sorry)
```

We *almost* do not need any axioms about negation; to work with a graded ring as opposed to a graded semiring, the user could write `[Π i, add_comm_group (A i)] [gsemiring A]`. Unfortunately, our hand is forced by "forgetful inheritance" to define `gring` anyway in order to add an `int_cast` operation and associated axioms.

Just as we used `graded_monoid A` in section 6.3.1 to bundle our graded monoid with its grade to enable reuse of the `monoid` API, we'd like to be able to enable reuse of the `semiring` API on graded semirings. We cannot use `graded_monoid A` here, as in a graded ring an element can consist of distinct grades added together; instead we use `direct_sum ι A`, with notation `⊕ i, A i`. Here, the element `x : A i` is represented by `direct_sum.of A i x`. This comes with all the additive structure we need already; all we have to do is extend our multiplicative structure onto it linearly, with our end goal being to produce a `semiring (⊕ i, A i)` instance.

To do this, we first promote our `mul` to a bundled homomorphism (section 3.4.3) that is additive in each argument

```
def gmul_hom [gsemiring R] {i j} : A i →+ A j →+ A (i + j) :=
{ to_fun := λ a,
  { to_fun := λ b, gsemiring.mul a b,
    map_zero' := gsemiring.mul_zero _,
    map_add' := gsemiring.mul_add _ },
  map_zero' := add_monoid_hom.ext $ λ a, gsemiring.zero_mul a,
  map_add' := λ a₁ a₂, add_monoid_hom.ext $ λ b, gsemiring.add_mul _ _ _}
```

as this allows us to lift this map to consume and produce elements of the direct sum as:

```
def mul_hom : (⊕ i, A i) →+ (⊕ i, A i) →+ ⊕ i, A i :=
direct_sum.to_add_monoid $ λ i,
  add_monoid_hom.flip $ direct_sum.to_add_monoid $ λ j, add_monoid_hom.flip $
    (direct_sum.of A _).comp_hom.comp $ gmul_hom A
```

Unfortunately working with bundled maps in mathlib forces you to write things in the rather unreadable point-free style as above. The benefit of working in a theorem prover is that we can at least verify that something unreadable still behaves as we want:

```
lemma mul_hom_of_of {i j} (a : A i) (b : A j) :
  mul_hom A (of _ i a) (of _ j b) = of _ (i + j) (gsemiring.mul a b) := sorry
```

It might feel like we're on the home stretch to providing the semiring instance; indeed, the proofs relating multiplication and the additive structure follow trivially from `mul_hom`. Unfortunately, we're now faced with actually using the API we built in section 6.3.2 to prove the multiplicative

properties! The key result we need to do this is that our two notions of equality are equivalent; that is,

```
lemma of_eq_of_graded_monoid_eq {i j : ι} {a : A i} {b : A j}
  (h : graded_monoid.mk i a = graded_monoid.mk j b) :
  direct_sum.of A i a = direct_sum.of A j b := sorry
```

After once again fighting against point-free nonsense to turn associativity into equality of two tri-additive maps, we can use the `ext` tactic to reduce our problem to associativity of three terms of the form `of A i x`:

```
private lemma mul_assoc (a b c : ⊕ i, A i) : a * b * c = a * (b * c) :=
-- `λ a b c, a * b * c = λ a b c, a * (b * c)` as bundled homomorphisms
suffices (mul_hom A).comp_hom.comp (mul_hom A)
       = (add_monoid_hom.comp_hom flip_hom $
           (mul_hom A).flip.comp_hom.comp (mul_hom A)).flip,
  from fun_like.congr_fun (fun_like.congr_fun (fun_like.congr_fun this a) b) c,
begin
  ext ai ax bi bx ci cx : 6,
  show mul_hom A (mul_hom A (of A ai ax) (of A bi bx)) (of A ci cx) =
       mul_hom A (of A ai ax) (mul_hom A (of A bi bx) (of A ci cx)),
```

from which the rest follows using our previous results:

```
  rw [mul_hom_of_of, mul_hom_of_of, mul_hom_of_of, mul_hom_of_of],
  exact of_eq_of_graded_monoid_eq
    (mul_assoc (graded_monoid.mk ai ax) ⟨bi, bx⟩ ⟨ci, cx⟩),
end
```

A similar approach can be used to prove the `mul_one` and `one_mul` fields in order to finish the construction of `semiring (⊕ i, A i)`. The "point-free nonsense" approach is not the only path available to us; but the alternative of using `induction` creates annoying side-goals to prove that the map is additive.

There is another important construction we will want for working with this direct sum representation of a graded ring; a way to construct a ring homomorphism out of the graded ring given a suitable family of homomorphisms from each piece. This can be written as:

```
def direct_sum.to_semiring
  (f : Π {i}, A i →+ R)
  (hone : f gsemiring.one = 1)
  (hmul : ∀ {i j} (ai : A i) (aj : A j), f (gsemiring.mul ai aj) = f ai * f aj) :
  (⊕ i, A i) →+* R :=
{ map_one' := sorry, map_mul' := sorry, .. direct_sum.to_add_monoid f }
```

We will find this instrumental for relating external and internal direct sums in section 6.4.4.

Example 6.2 (the n^{th} tensor powers)**.** *This typeclass allows us to express the graded ring structure of the n^{th} tensor power over an R-module V in [mathlib#10255] as*

```
instance : gsemiring (λ n : ℕ, ⊗[R]^n V) := sorry
```

We can then show with the aid of `direct_sum.to_semiring` *that $\mathcal{T}(V)$ is isomorphic as a ring (and algebra) to $\bigoplus_n V^{\otimes n}$; that is* `tensor_algebra R V ≃ₐ[R] (⊕ n, ⊗[R]^n V)`*.*

6.4. Internal gradings

6.4.1. Decompositions of sets

For an external decomposition with no algebraic structure, the task is simple; a unique decomposition of a type `α` into its pieces `A : ι → Type*` can be spelled as the equivalence to a sigma type, `decompose : α ≃ Σ i : ι, A i`. For an internal decomposition where `A : ι → set α`, we have some other options. If we don't care about a constructive decomposition and are happy with a classical one, we can just state that the components span the entire type and are disjoint, as:

```
(⋃ i, A i) = set.univ ∧ pairwise (disjoint on A)
```

If we do care about constructiveness, we can instead have a function `grade : α → ι` that respects `∀ (a : α) (i : ι), a ∈ A i ↔ grade a = i`. Whichever approach we pick, it is straightforward to recover the externally-graded viewpoint via the map `decompose : α ≃ Σ i : ι, ↥(A i)`. Here, `↥` is the operator that lets us view a set as a subtype of the type of its elements.

6.4.2. Graded monoids

A preliminary attempt at formalizing an internal multiplicative grading structure might look like

```
class set.graded_monoid [monoid M] [add_monoid ι] (A : ι → set M) : Prop :=
(one_mem : 1 ∈ A 0)
(mul_mem : ∀ ⦃i j : ι⦄ {gi gj : M}, gi ∈ A i → gj ∈ A j → gi * gj ∈ A (i + j))
```

This works fine for graded monoids; but for graded semirings, rings, and algebras we need to apply the additional constraints that each `A i` is closed under the appropriate operations.

To avoid having to write separate typeclasses for each case and ending up with our API in triplicate, we instead generalize over `A : ι → set M`; with the goal being able to talk about `A : ι → add_submonoid R`, `A : ι → add_subgroup R`, and `A : ι → submodule S R` in a unified way. As well as avoiding the need for three different typeclasses, this also means we can reuse all the theory we already have about `add_submonoid`, `add_subgroup`, and `submodule`. To do that, we introduce a `set_like` class, which was one of the inspirations for the `fun_like` generalization described in [45, §6.3]. This class lets us express that elements `s` of a type `S` has a canonical interpretation as a set `↑s : set α`, and is defined as:

```
class set_like (S : Type*) (α : out_param Type*) :=
(coe : S → set α)  -- the function that is used for '↑' coercion notation
(coe_injective' : function.injective coe)
```

This equips `s : S` with membership notation `a ∈ s : Prop` and a coercion to type `↥s : Type*`, and permits us to write

```
class set_like.graded_monoid {ι M S : Type*}
  [set_like S M] [monoid M] [add_monoid ι] (A : ι → S) : Prop :=
(one_mem : 1 ∈ A 0)
(mul_mem : ∀ ⦃i j : ι⦄ {gi gj : M}, gi ∈ A i → gj ∈ A j → gi * gj ∈ A (i + j))
```

At this point, we can deliver on the earlier claim that the internal viewpoint can be expressed via the external viewpoint. To do this, we show that the family of subtypes `λ i, ↥(A i)` has a graded multiplicative structure, as:

```
-- this implies 'monoid (graded_monoid (λ i, ↥(A i)))' via 'gmonoid.to_monoid'
instance set_like.gmonoid [set_like S M] [monoid M] [add_monoid ι]
  (A : ι → S) [set_like.graded_monoid A] :
  gmonoid (λ i, ↥(A i)) :=
{ one := ⟨1, set_like.graded_monoid.one_mem,
  mul := λ i j a b, ⟨(a * b : R), set_like.graded_monoid.mul_mem a.prop b.prop,
  mul_assoc := λ ⟨i, a, ha⟩ ⟨j, b, hb⟩ ⟨k, c, hc⟩,
    sigma.subtype_ext (add_assoc _ _ _) (mul_assoc _ _ _),
  ..sorry  /- etc -/ }
```

Note here that we are paying the cost outlined in the first row of table 6.2 of having to reprove associativity of addition of the grades, which is a mark against our choice of item 2.

Example 6.3 (the free monoid over α)**.** *This typeclass allows us to express*[a] *the internal graded monoid structure of the free monoid, with elements graded by the number of generators*

```
instance :
  set_like.graded_monoid
    (λ i : ℕ, (set.range (free_monoid.of : α → free_monoid α)) ^ i) :=
{ one_mem := by rw [pow_zero, set.mem_one],
  mul_mem := λ i j x y hx hy, by { rw pow_add, exact set.mul_mem_mul hx hy} }
```

[a] After enabling the appropriate by-default-disabled instances.

We are not quite done yet; we have shown that the subtypes can be glued together to form an object with graded multiplication, but the `set_like.graded_monoid` typeclass above does nothing to ensure that the glued-together type `graded_monoid (λ i, ↥(A i))` is in bijection with `M`. We can reuse either the classical or constructive approach from section 6.4.1, but with our new `monoid (graded_monoid (λ i, ↥(A i)))` instance we can promote the equivalence `decompose : α ≃ Σ i : ι, ↥(A i)` to a multiplicative isomorphism, stated as `decompose : α ≃* graded_monoid (λ i, ↥(A i))`.

6.4.3. Decompositions of additive monoids and R-modules

For a unique decomposition with an additive structure, we cannot use the same approach as section 6.4.1 but instead need to decompose into a direct sum, as `decompose : α ≃+ ⊕ i : ι, ↥(A i)`. Let us consider the internal decomposition of an additive group `α` into the family `A : ι → add_subgroup α`. We need to be a little more careful when describing the disjointness condition, as what we actually require is that every component is disjoint (in the sense of having trivial intersection $\{0\}$) from the span of all the others. We can spell that as

```
(⨆ i, A i) = ⊤ ∧ complete_lattice.independent A
```

but for additive submonoids this condition, while necessary, is still not sufficient; consider when $A_+ = \{z : \mathbb{Z} \mid 0 \leq z\}$ and $A_- = \{z : \mathbb{Z} \mid 0 \geq z\}$, which are disjoint and span all of $\mathbb{Z}$, but clearly do not permit a decomposition [*mathlib#9214*]. As such, we cannot use this as our definition. Instead, we require that the canonical map from `⊕ i : ι, A i` to `α` (defined as roughly `λ x, ∑ i, ↑(x i)`) is bijective.

Once again we're on the precipice of stating things in triplicate, as we want to state this condition (and the consequences of it) for `add_monoid α` and `submodule R α` as well to cover the cases on the right of fig. 6.1. Until very recently, stating this condition in triplicate was exactly what mathlib did; but thanks to [mathlib#11750] which transfers the success in [45] from `fun_like` to `set_like`, we can now generalize over `add_subgroup α` as `S` where `(S : Type*) [set_like S α] [add_submonoid_class S α]`. This lets us define a single canonical map `coe_add_monoid_hom` that works for all three cases as

```
protected def coe_add_monoid_hom [add_comm_monoid α]
  [set_like S α] [add_submonoid_class S α] (A : ι → S) :
  (⊕ i, A i) →+ M :=
direct_sum.to_add_monoid (λ i, add_submonoid_class.subtype (A i))
```

which in turn allows us to state our condition just once to cover all three cases. We can state it either constructively, by carrying around an explicit inverse:

```
class decomposition (A : ι → S) :=
(decompose' : α → ⊕ i, A i)  -- split elements into their grades
(left_inv : function.left_inverse (coe_add_monoid_hom A) decompose')
(right_inv : function.right_inverse (coe_add_monoid_hom A) decompose')
```

or classically by simply proving bijectivity. We provide a proof in mathlib that on submodules over additive groups the `complete_lattice.independent` formulation is equivalent to these definitions.

6.4.4. Graded (semi)rings

To talk about a semiring with an internally grade-compatible multiplication, we thankfully need to define no further typeclasses; we can write

```
variables {ι R S : Type*} [add_monoid ι] [semiring R] [set_like S R]
variables [add_submonoid_class S R] (A : ι → S) [set_like.graded_monoid A]
```

where `set_like.graded_monoid` handles our conditions on the multiplicative structure, `add_submonoid_class` handles our conditions on the additive structure, and `semiring R` ensures the compatibility between the two. We'd like to end up with a `gsemiring (λ i, ↥(A i))` instance as a result of these hypotheses so as to also have a `semiring (⨁ i, ↥(A i))` instance, which we achieve as

```
instance set_like.gsemiring : direct_sum.gsemiring (λ i, ↥(A i)) :=
{ mul_zero := λ i j _, subtype.ext (mul_zero _),
  zero_mul := λ i j _, subtype.ext (zero_mul _),
  mul_add := λ i j _ _ _, subtype.ext (mul_add _ _ _),
  add_mul := λ i j _ _ _, subtype.ext (add_mul _ _ _),
  ..set_like.gmonoid A }
```

Once again, the introduction of `add_submonoid_class` excused us from needing three copies of this definition.

Now that we have this instance, we can build the canonical ring morphism from `⨁ i, ↥(A i)` to `R` that amounts to summing the elements from each grade (after passing them through the canonical additive morphism from the subtype), building upon the `direct_sum.to_semiring` definition at the end of section 6.3.3:

```
def direct_sum.coe_ring_hom [add_monoid ι] [semiring R] [set_like S R]
  [add_submonoid_class S R] (A : ι → S) [set_like.graded_monoid A] :
  (⨁ i, ↥(A i)) →+* R :=
direct_sum.to_semiring
  (λ i, add_submonoid_class.subtype (A i)) rfl (λ _ _ _ _, rfl)
```

The `direct_sum.coe_add_monoid_hom` we mention in section 6.4.3 has a definitionally equal underlying function to this, meaning we can reuse the `decomposition A` from that section defined in terms of the former to obtain the canonical ring isomorphism `decompose_ring_equiv : R ≃+* ⨁ i, ↥(A i)` between an internally-graded ring `R` and the direct sum of its grades `⨁ i, ↥(A i)`. For convenience, we provide a single typeclass that provides access to this operation:

```
class graded_ring (A : ι → σ) extends graded_monoid A, decomposition A.
```

Among other functions and lemmas which build on this convenience, we provide projection maps as additive maps `proj A i : R →+ R` so that any $x : R$ can be written as $x = \sum_i x_i$, with x_i being the i-th projection of x with respect to grade `A`, without having to explicitly go via the direct sum of subtypes `⨁ i, ↥(A i)`.

6.5. Graded R-algebras

This chapter would not be complete without building external and internal graded R-algebras on top of the graded rings; indeed, we have added definitions of these to mathlib, as the typeclass `galgebra`, the instance `submodule.galgebra`, and the shorthand `graded_algebra`; but the process of defining these presented no new challenges over those already faced when defining graded rings. Just as we were able to recover a ring isomorphism in section 6.4.4, these definitions

let us recover the analogous R-algebra isomorphism `decompose_alg_equiv : X ≃ₐ[R] ⨁ i, ι(A i)`.

Example 6.4 (the multivariate polynomials)**.** *Returning to the examples in section 6.1, we can define the homogeneous graduation:*

```
instance : graded_algebra (λ i : ℕ, homogeneous_submodule σ R i) := sorry
```

where `σ` *represents the variables in the multivariate polynomial ring and* `homogeneous_submodule σ R i` *the homogeneous polynomials of degree* i*.*

Example 6.5 (the tensor algebra $\mathcal{T}(V)$)**.** *Given* `ι R` *as the canonical map* $V \to \mathcal{T}(V)$*, we write the typical internal grading as:*

```
instance : graded_algebra
  ((^) (ι R : M →ₗ[R] tensor_algebra R M).range : ℕ → submodule R _) := sorry
```

Example 6.6 (the Clifford algebra $\mathcal{C}\ell(V, Q)$)**.** *Similarly, we can write*[a]

```
def even_odd (i : zmod 2) : submodule R (clifford_algebra Q) :=
⨆ (j : {n : ℕ // ↑n = i}), (ι Q).range ^ (j : ℕ)
instance : graded_algebra (clifford_algebra.even_odd Q) := sorry
```

where `ι Q` *is a similar canonical map, and* `even_odd 0` *and* `even_odd 1` *are the even and odd submodules respectively.*

[a] Thus resolving the further work in [10, §8.1].

6.6. Summary

This chapter outlined the substantial amount of busy-work required to define the various types of graded structure in mathlib, and the various design choices made along the way. Instead of choosing between internal and external grading, we opted to use the latter to implement the former. When it comes to classical vs constructive decompositions, we opted to have both, just like mathlib has classical vs constructive finite sets. Finally, for stating equality between non-definitionally-equal grades, we opted to use sigma types.

To verify that our design decisions are sensible, we demonstrated a variety of results stated using our new typeclasses. While not discussed in detail in this chapter, our supplemental repository includes an extended example by Jujian Zhang of a preliminary development of the Proj construction in algebraic geometry, which using the result from example 6.4 culminates in defining the projective n-space. The success of this formalization indicates that the design of graded objects is coherent with other theories in mathlib.

A snapshot of the unabridged code from this work, permalinked throughout via "", is available at https://github.com/eric-wieser/lean-graded-rings. It comprises around 1150 sloc[12] of API development, 1250 sloc of applications, and 5300 sloc of the extended Proj example. Permalinks

[12] source lines of code

resembling "↷" refer to declarations in mathlib itself that were infeasible to extract into our isolated repository. Much of this code has already been integrated into mathlib over the course of over 30 pull requests[13].

The monolithic nature of mathlib development means that our design decisions are easy to revisit at a later date, as the assumption is already that code written against one version of mathlib is not guaranteed to work without modification on a later version. Meanwhile, the act of having made the decisions enables downstream work to progress; as well as unblocking diverging formalization projects by the two authors[14], our foundations have allowed mathlib to gain formalizations by other contributors about various internal decompositions in inner product spaces and torsion modules.

We shall revisit graded rings later in this thesis, in sections 9.5 and 10.3.

[13] Tracked in https://github.com/leanprover-community/mathlib/projects/12.
[14] In geometric algebra and algebraic geometry, respectively!

7

Multiple-inheritance hazards in dependently-typed algebraic hierarchies

> The computing scientist's main challenge is not to get confused by the complexities of his own making.
>
> *(Edsger W. Dijkstra)*

This chapter is reproduced from "Multiple-Inheritance Hazards in Dependently-Typed Algebraic Hierarchies" [3]. Section 7.4.3 corrects the incorrect claim made by [3, §4.3]

Abstract algebra provides a large hierarchy of properties that a collection of objects can satisfy, such as forming an abelian group or a semiring. These classifications can be arranged into a broad and typically acyclic directed graph. This graph perspective encodes naturally in the typeclass system of theorem provers such as Lean, where nodes can be represented as structures (or records) containing the requisite axioms. This design inevitably needs some form of multiple inheritance; a ring is both a semiring and an abelian group.

In the presence of dependently-typed typeclasses that themselves consume typeclasses as type-parameters, such as a vector space typeclass which assumes the presence of an existing additive structure, the implementation details of structure multiple inheritance matter. The type of the outer typeclass is influenced by the path taken to resolve the typeclasses it consumes. Unless all possible paths are considered judgmentally equal, this is a recipe for disaster.

This chapter provides a concrete explanation of how these situations arise (reduced from real examples in mathlib), compares implementation approaches for multiple inheritance by whether judgmental equality is preserved, and outlines solutions (notably: kernel support for η-reduction of structures) to the problems discovered.

7.1. Introduction

It becomes clear very early in the development of mathematical libraries that a generalization over algebraic properties is essential; as soon as we are able to speak about $\mathbb{N}$ and $\mathbb{Z}$, we will want to have available that $a + b = b + a$ whether $a, b : \mathbb{N}$ or $a, b : \mathbb{Z}$, and it would be strongly preferable that we can refer to this property by a single name.

The generalization we seek is of course well-studied as the field of abstract algebra, and the commutativity property above can be phrased as "$\mathbb{N}$ and $\mathbb{Z}$ are both semirings", or using language more precise to the specific property we care about "$\mathbb{N}$ and $\mathbb{Z}$ are both abelian monoids". At least when considering only those which operate on a single carrier type, algebraic structures can be connected into a directed graph; all rings are semirings and abelian groups, so we can draw a pair of edges from "ring" to "semiring" and "abelian group". An illustration of the depth and breadth of such a graph can be seen in [39, fig. 1], while a reduced example that we will use in this chapter can be seen in fig. 7.1.

Encoding this directed graph into the machinery of a particular theorem prover can be done in multiple ways, which are outlined in [45, §1] and presented with example code across a variety of languages in [53, fig. 1]. This chapter focuses on the typeclass approach used by mathlib [39] in the Lean 3 theorem prover [6]; though the observations generalize to other implementations in dependent type theory built upon "structure" types.

In this approach, the graph is pruned to be acyclic, and then a typeclass is created for each node carrying its operators (data fields) and the properties they satisfy (proof fields). The edges correspond to functions converting from stronger structures to weaker structures, each registered as a typeclass instance. This encodes naturally in "record" or "structure" types with multiple inheritance, where we can write down the desired edges declaratively in the form of a list of base structures, and have the language generate the necessary "forgetful" instances automatically. A simple example of this can be found in [45, §4].

Unfortunately, the devil is in the details; in Lean, Coq, Agda, and Isabelle, support for multiple inheritance is not part of the underlying type theory, so types that use multiple inheritance have to be translated by the elaborator into inductive types that do not. There are multiple ways to perform such a translation, and the choice is not inconsequential.

In section 7.2 we outline two such approaches, and show how they can each be used to construct a much-reduced version of mathlib's abstract algebra library. Section 7.3 introduces a more complex use of a typeclass from mathlib, and demonstrates how in the absence of special kernel support for η-reduction on structure types, its design is incompatible with "nested" approach to structures. Section 7.4 outlines some workarounds that permit the "nested" approach to be used even in the absence of this support. Section 7.5 explains how the problem is not unique to typeclass-based approaches.

The problems explored here are far from hypothetical; the migration of mathlib from Lean 3 to Lean 4 [7] forces a switch from the approach in section 7.2.1 to that in section 7.2.2, which has presented a significant stumbling block [lean4#2074].

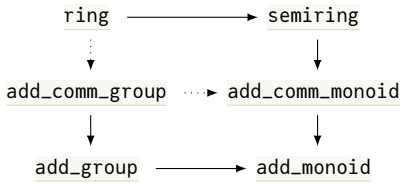


Figure 7.1.: A hierarchy of algebraic typeclasses, where arrows indicate a stronger typeclass implying a weaker typeclass.

Dotted arrows correspond to the "non-preferred" typeclass paths which are relevant to section 7.2.2.

```
class add_monoid (α : Type) :=
(zero : α) (add : α → α → α)

class add_comm_monoid (α : Type) extends add_monoid α

class semiring (α : Type) extends add_comm_monoid α :=
(one : α) (mul : α → α → α)

class add_group (α : Type) extends add_monoid α :=
(neg : α → α)

class add_comm_group (α : Type) extends add_group α, add_comm_monoid α

class ring (α : Type) extends semiring α, add_comm_group α
```

Listing 7.2.: The hierarchy in fig. 7.1 described using `extends` clauses.

7.2. Types of structure inheritance

Lean 3 supports two types of structure inheritance: the default "new style", which we will refer to as "nested", and does not support multiple inheritance; and the legacy "old style" (enabled with `set_option old_structure_cmd true`) which we will refer to as "flat", and *does* support multiple inheritance. Lean 4 (as a language) does away with the "flat" mode, but extends the "nested" mode to support multiple inheritance.

To compare these approaches, this section demonstrates how to build the miniature algebraic hierarchy shown in fig. 7.1. If we permit ourselves to use the built-in language support for multiple inheritance, we could write this as in listing 7.2. As they are not going to be relevant to the discussion in this chapter, the proof fields such as `one_mul : ∀ a : α, mul one a = a` have all been omitted.

To avoid this chapter being about a specific implementation of inheritance in a specific version of Lean, we will avoid the `extends` keyword, instead emulating it via different possible encodings of inheritance into regular structures. For simplicity this chapter is largely presented as about Lean, but the supplemental repository referenced in section 7.7 demonstrates how the Lean 3 samples presented here can be replicated in Coq[1] and in Lean 4[2].

[1] Albeit somewhat non-idiomatically.
[2] At least, in old versions without pertinent fixes!

7.2.1. Flat structures

The "flat" approach to structure inheritance is to copy all of the fields from the base classes into the derived class. If multiple base classes share a field of the same name, then these fields are merged[3]. The forgetful instances are then implemented by unpacking all the relevant fields of the derived class and passing them to each base class constructor (which in Lean 3 can be written as `{ ..derived }`).

This can be seen for the toy example from listing 7.2 in listing 7.3a; `ring` extends both `semiring` and `add_comm_group`, so inherits the union of the four fields of `semiring` (`zero`, `add`, `one`, `mul`) and the three fields of `add_comm_group` (`zero`, `add`, `neg`). The `ring.to_semiring` and `ring.to_add_comm_group` instances generate constructor applications that reassemble the corresponding fields.

This approach is straightforward to implement in a theorem prover, and is the one used (via `set_option old_structure_cmd true`) in the majority of mathlib's algebraic hierarchy in Lean 3. A downside to this approach is that it can produce more work for unification (leading to poor performance) in long inheritance chains [45, §10].

7.2.2. Nested structures

A naïve approach to multiple inheritance for `ring` would be simply to create a structure containing a `to_semiring` field and a `to_add_comm_group` field. The problem with this approach is that the resulting structure contains two separate `add` fields. Compatibility of these fields could in principle be enforced with a proof field along the lines of `add_ok : to_semiring.add = to_add_comm_group.add`, but this makes the API very unpleasant to use as the user now has to rewrite between all the different copies of `add`.

The way to modify this approach to avoid this pitfall is to add a field for each base class that doesn't overlap with any previous base classes, otherwise fall back to the "flat" approach and add the non-overlapping fields directly. We call these non-overlapping base-classes "preferred" instances, as the projections for these fields can be registered directly with the typeclass system using `attribute [instance] derived.to_base`. What remains are the "non-preferred" instances, which can be constructed in a similar way to what was done in section 7.2.1, though with somewhat messier expressions. Note that unlike section 7.2.1, this approach is influenced by the order of the base classes.

This can be seen in listing 7.3b; `ring` contains a `to_semiring` field for its first base class, but `add_comm_group` would overlap so its remaining non-overlapping field (`neg`) is added separately. The "preferred" `ring.to_semiring` projection is then registered with the typeclass system, while the "non-preferred" `ring.to_add_comm_group` is painstakingly assembled piece-by-piece. To encourage Lean to avoid the "non-preferred" instance, we give it a low priority of 100 (the default is 1000).

This approach is more complicated to implement (and indeed, was not implemented in Lean

[3] Unless they are of different types, which raises an error.

until Lean 4), but can have performance advantages for unification as the "preferred" instance paths do not introduce a constructor application.

The result of listing 7.3b is that the graph in fig. 7.1 is imbued with an asymmetry; the dotted edges are provided by "non-preferred" instances. These edges can be chosen on any spanning tree[4] of the overall graph, and indeed can be optimized to fall on the paths most used by the library [mathlib4#3840].

For the purpose of this chapter, the opposite is true; their placement has been pessimized to deliberately cause a failure, which we shall see in section 7.3.2!

[4] In general this is a spanning diamond-free directed acyclic graph, but for this chapter it suffices to consider a tree.

```
class add_monoid (α : Type) :=
(zero : α) (add : α → α → α)

class add_comm_monoid (α : Type) :=
(zero : α) (add : α → α → α)

instance add_comm_monoid.to_add_monoid (α : Type)
  [i : add_comm_monoid α] : add_monoid α := { ..i }

class semiring (α : Type) :=
(zero : α) (add : α → α → α)
(one : α) (mul : α → α → α)

instance semiring.to_add_comm_monoid (α : Type)
  [i : semiring α] : add_comm_monoid α := { ..i }

class add_group (α : Type) :=
(zero : α) (add : α → α → α)
(neg : α → α)

instance add_group.to_add_monoid (α : Type)
  [i : add_group α] : add_monoid α := { ..i }

class add_comm_group (α : Type) :=
(zero : α) (add : α → α → α) (neg : α → α)

instance add_comm_group.to_add_group (α : Type)
  [i : add_comm_group α] : add_group α := { ..i }

instance add_comm_group.to_add_comm_monoid (α : Type)
  [i : add_comm_group α] : add_comm_monoid α := { ..i }

class ring (α : Type) :=
(zero one : α) (add mul : α → α → α) (neg : α → α)

instance ring.to_semiring (α : Type)
  [i : ring α] : semiring α := { ..i }

instance ring.to_add_comm_group (α : Type)
  [i : ring α] : add_comm_group α := { ..i }
```

(a) The flat approach (section 7.2.1), copying base fields to derived classes.

```
class add_monoid (α : Type) :=
(zero : α) (add : α → α → α)

class add_comm_monoid (α : Type) :=
(to_add_monoid : add_monoid α)

attribute [instance] add_comm_monoid.to_add_monoid

class semiring (α : Type) :=
(to_add_comm_monoid : add_comm_monoid α)
(one : α) (mul : α → α → α)

attribute [instance] semiring.to_add_comm_monoid

class add_group (α : Type) :=
(to_add_monoid : add_monoid α)
(neg : α → α)

attribute [instance] add_group.to_add_monoid

class add_comm_group (α : Type) :=
(to_add_group : add_group α)

attribute [instance] add_comm_group.to_add_group

@[priority 100] instance add_comm_group.to_add_comm_monoid
  {α : Type} [i : add_comm_group α] : add_comm_monoid α :=
{ to_add_monoid := i.to_add_group.to_add_monoid, ..i }

class ring (α : Type) :=
(to_semiring : semiring α)
(neg : α → α)

attribute [instance] ring.to_semiring

@[priority 100] instance ring.to_add_comm_group
  (α : Type) [i : ring α] : add_comm_group α :=
{ to_add_group :=
  { to_add_monoid :=
      i.to_semiring.to_add_comm_monoid.to_add_monoid, ..i },
  .. i }
```

(b) The nested approach (section 7.2.2), inserting the first parent as a field and copying the remaining fields.

Listing 7.3.: Two approaches to implementing inheritance, by elaborating the `extends` clauses in listing 7.2 as the highlighted lines.

7.3. Typeclasses depending on typeclasses

In section 7.2, we concerned ourselves with the typical examples of typeclasses which depend on a single type. In Lean, it is possible for typeclasses to depend not only on multiple types, but on typeclasses that constrain those types. A simple typeclass of this form is `module R M`, which is used to declare that given a semiring R and an abelian monoid M, there is an R-module structure on M. A more complete explanation of this typeclass can be found in chapter 4 and [45, §5]. For the purpose of this chapter, we can imagine the simpler definition as follows:

```
class module (R M : Type) [semiring R] [add_comm_monoid M] :=
(smul : R → M → M)
-- (one_smul : ∀ (x : M), smul 1 x = x)
-- (mul_smul : ∀ (r s : R) (x : M), smul (r * s) x = smul r (smul s x))
-- (add_smul : ∀ (r s : R) (x : M), smul (r + s) x = smul r x + smul s x)
-- (zero_smul : ∀ (x : M), smul 0 x = 0)
```

Here, the proof fields within the typeclass depend on the operators imbued upon the types R and M. Just as in section 7.2, we shall ignore these proof fields as they are not relevant to the discussion other than providing motivation for the `[semiring R]` `[add_comm_monoid M]` parameters.

7.3.1. Equality of typeclass arguments

A natural use of this typeclass is to record the fact that any semiring is a module over itself, where the scalar action `smul` is just multiplication (as in section 4.2.1). This can be written in Lean as

```
instance semiring.to_module (R) [iS : semiring R] : module R R :=
{ smul := semiring.mul }
```

The type of this instance is misleading; while a human reader could be forgiven for assuming that the type is just `module R R`, to Lean the type is

```
@module R R iS (@semiring.to_add_comm_monoid R iS)
```

where `@` is syntax to tell Lean that even the automatically-populated typeclass arguments should be spelled out explicitly[5]. The expressions for these implicit arguments are visualized graphically in fig. 7.4a

Lean can now tell us that a *ring* is a module over itself, as after all every ring is also a semiring. We can ask this question with:

```
example (R) [iR : ring R] : module R R := by apply_instance
```

Once again, the type is misleading; the true type can be seen in fig. 7.4b. Comparing the types for fig. 7.4a and fig. 7.4b, we see that the former unifies with the latter by setting `iS = @ring.to_semiring R iR`; for this reason, Lean finds our instance as `@semiring.to_module`

[5] This style of display can be enabled with `set_option pp.implicit true` in Lean 3 and `set_option pp.explicit true` in Lean 4.

`R (@ring.to_semiring R iR)`.

7.3.2. Inequality of typeclass arguments

Let's imagine now that we want to write a lemma that applies to a module over a ring (as opposed to a semi-module over a semiring), and states that $(-r)m = -(rm)$. We write this as[6]

```
lemma neg_smul {R M} [ring R] [add_comm_group M] [module R M] (r : R) (m : M) :
  module.smul (add_group.neg r) m = add_group.neg (module.smul r m) := sorry
```

To complete our setup, let's check that this lemma applies to the R-module structure on R:

```
example {R} [iR : ring R] (r : R) (r' : R) :
  module.smul (add_group.neg r) r' = add_group.neg (module.smul r r') :=
neg_smul r r'
```

If we use the "flat" design in listing 7.3a, then this continues to work as expected. The same is not true of the "nested" design in listing 7.3b, which fails to synthesize type class instance for

```
@module R R (@ring.to_semiring R iR)
  (@add_comm_group.to_add_comm_monoid R (@ring.to_add_comm_group R iR))
```

which is shown graphically in fig. 7.4c. The `neg_smul` lemma is an example of how typeclass resolution can be steered through a specific node of the graph in fig. 7.1.

In Lean 3, the reason this fails is nothing to do with typeclass search; the problem is that the type in fig. 7.4c is not equal to type in fig. 7.4b, due to the implicit `add_comm_monoid M` arguments (shown in red) not being considered equal. Considerations of equality between the red paths in figs. 7.4b and 7.4c are often referred to as a "typeclass diamonds" due to the shape they form when overlaid; though this is a rather more subtle diamond problem than the ones described in section 4.5 and [44, §3.1] as it is caused by code that would normally be invisible to the user.

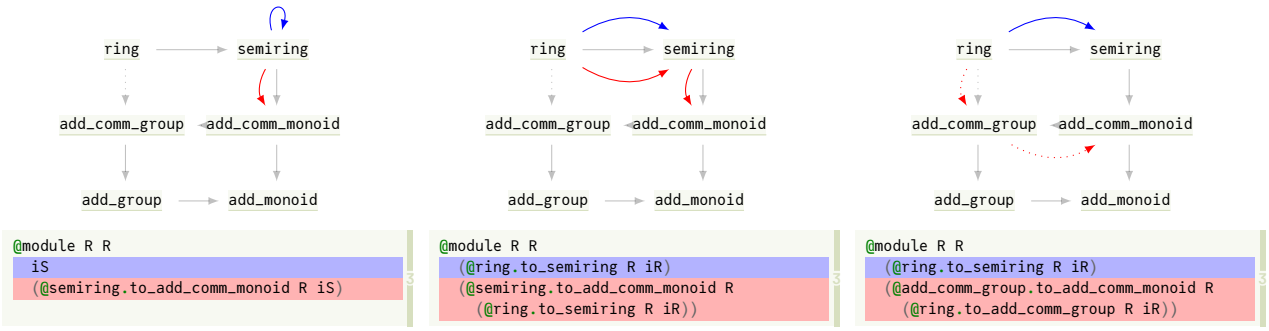
To mathematicians, this diagram obviously commutes; weakening a ring to an abelian monoid via a semiring is the same as doing so via an abelian group. But Lean doesn't care about "obviously": when determining equality of types, it's not enough for them to just be *provably* the same; they need to be *definitionally* (sometimes called judgmentally) so. A proof of `rfl` can be used to determine if two terms are judgmentally equal; under listing 7.3b, we get an error confirming they are not:

```
example (R) [iR : ring R] :
  (@semiring.to_add_comm_monoid R (@ring.to_semiring R iR)) =
  (@add_comm_group.to_add_comm_monoid R (@ring.to_add_comm_group R iR)) :=
rfl -- fails in Lean 3 with listing 7.3b
```

7.3.3. Impact of the inheritance strategy

The `rfl` in section 7.3.2 that fails under listing 7.3b but not listing 7.3a tells us that the nested inheritance is certainly to blame here. The underlying cause is the difference between the

[6] Omitting the usual `-` and `•` notation to keep listing 7.3 short.



(a) Instance to match (b) Matching paths (c) Mismatching paths

Figure 7.4.: Paths taken through the graph in fig. 7.1 when filling the two implicit arguments of the type of `module R R`.

Dotted lines again refer to "non-preferred" edges.

"preferred" and "non-preferred" paths.

The "non-preferred" edges in listing 7.3b are implemented directly as a constructor application via the `{ }` syntax; so by virtue of following "non-preferred" edges, the red path in fig. 7.4c unfolds to an application of the `add_comm_monoid` constructor. The "preferred" edges correspond to a projection; unless applied to something that unifies against a constructor, these operations themselves do not unify against a constructor. As the red path in fig. 7.4b consists of only "preferred" edges, it only unifies with this `add_comm_monoid` constructor if `iR` unifies with a `ring` constructor.

If `iR` is a concrete instance such as `instance int.ring : ring ℤ`, then it will almost certainly unify with a `ring` constructor, and the overall unification problem is solvable. However, if `iR` is a free variable, it will only unify with a constructor in systems which support "η-reduction for structures". Lean 3 is not such a system, which makes unification impossible.

7.3.4. Other examples in mathlib

The `module` typeclass is far from the only typeclass in mathlib that follows the pattern introduced in section 7.3; some others typeclasses (all of which fall afoul of the issue in section 7.3.2) include

- `algebra (R A : Type) [comm_semiring R] [semiring A]`, indicating that A is an R-algebra.
- `star_ring (R : Type) [non_unital_semiring R]`, indicating that there is a $\star$ operator compatible with the existing ring structure on R.
- `cstar_ring (R : Type) [non_unital_normed_ring R] [star_ring R]`, indicating that the existing norm, $\star$, and ring structure are suitable to declare R a $C^\star$-ring.

Like the `module` example, the design of the first of these is brought on by a need to work with two separate carrier types, and the need to avoid "dangerous instances" [45, §5.1].

The other two can be described as "mixin" typeclasses, and are motivated by a desire to avoid a combinatorial explosion of typeclass variations: an attempt at `star_ring` without mixins could easily end up needing all 16 variations of unital/non-unital commutative/non-commutative normed? star rings/fields. This motivation is largely a pragmatic one; the introduction of a tool like Coq's Hierarchy Builder [54] to mathlib would eliminate the cost of manually authoring such an explosion of typeclasses.

7.4. Mitigation strategies

7.4.1. Perform η-reduction of structures in the kernel

A key difference between the type theory of Lean 3 and Lean 4 is that Lean 4 adds a kernel reduction rule that η-reduces structures[7], which is precisely what we concluded we needed in section 7.3.3. The following example demonstrates what this means:

```
structure point := (x y : ℤ)

-- fails in Lean 3, succeeds in Lean 4
example (p : point) : p = { x := p.x, y := p.y } := rfl
```

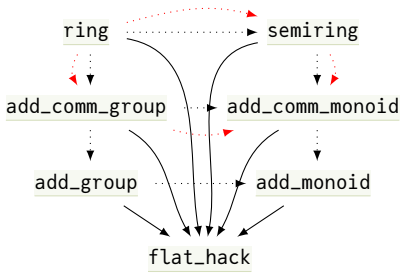
In essence, any value from a `structure` type is considered judgmentally equal to its constructor applied to its projections.

This feature was motivated by various "convenience" definitional equalities (as requested by [lean4#777]), such as wanting `e.symm.symm = e` for an equivalence `e : α ≃ β`; but in a thankful coincidence happens to be precisely the tool needed to resolve the trap in section 7.3.2 that Lean 4 dropping support for "flat" structures would otherwise have ensnared us in. In particular, the Lean 4 version of the failing `example ... := rfl` above succeeds.

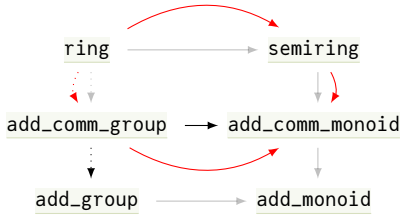
Until 2023-02-22, the structure η-reduction rule was disabled in Lean 4 during typeclass search; both due to performance concerns, and an absence of any evidence that it was necessary in the first place. As evidence mounted [lean4#2074], a compromise was reached to unblock the Lean 4 version of mathlib that allowed it to be temporary enabled[8] in places where there was no other choice but taking the performance hit. After some unification performance improvements which are out of scope for this chapter, this behavior was turned on globally on 2023-05-16 [lean4#2210].

Lean 4 is not the only language to have taken an experimental approach to structural η; Coq supports it too, under the disabled-by-default `Primitive Projections` option. In contrast, Agda enables it by default for `inductive` types[9], but allows it to be disabled via `no-eta-equality`.

[7] Strictly speaking, it η-reduces `inductive` types with one constructor; `structure`s are not native to the type theory of Lean, and instead just syntax for generating a suitable inductive type.
[8] Via `set_option synthInstance.etaExperiment true`.
[9] Some motivating discussion can be found in [55].



(a) A hack to force the behavior of flat inheritance when only nested inheritance is available.



(b) A variant of fig. 7.1 formed by swapping the two black arrows, that prevents the problem in fig. 7.4c.

Figure 7.5.: Alternate placements of the "preferred" spanning tree, with the diamond discussed in fig. 7.4 overlaid.

7.4.2. Use "flat" inheritance

The obvious approach to avoiding problems with "nested" inheritance is to simply not use it. Unfortunately, in the absence of elaborator support for translating a variation of listing 7.2 into listing 7.3a (such as in Lean 4) this would have to be done by hand, which can be rather tedious and error-prone.

There is however a trick; since the elaborator can translate listing 7.2 into listing 7.3b, we can construct a pathological graph such that all the edges we care about are forced to be "non-preferred". We do this by adding an empty `flat_hack` structure as the first base class of every structure, which ensures that the base classes always overlap (due to the `to_flat_hack` field), and so the only "preferred" base class is the unused `to_flat_hack` projection. The spanning tree of "preferred" base classes across all such typeclasses is a star with `flat_hack` at its center, as shown in fig. 7.5a.

This forces all the typeclass resolution to go through the "non-preferred" paths, which behave identically to their "flat" counterparts by unfolding to a constructor application.

7.4.3. Carefully select "preferred" paths

In section 7.2.2, we mention that the choice of where to place the spanning tree of "preferred" paths could be optimized for performance. In light of section 7.3.2, we could instead attempt to optimize to ensure that the problematic diamonds never arise. Indeed, there are many arrangements of the "preferred" paths in fig. 7.1 that do not run into the *specific example* in fig. 7.4c, such as fig. 7.5b.

For our purposes, an adequate rule for why the red arrows of fig. 7.5 commute but the ones of fig. 7.4 do not is that the paths commute only if their least common ancestor (with respect to "preferred" edges) are equal[10]; fig. 7.5a commutes due to the common `flat_hack` ancestor, fig. 7.5b commutes due to the common `add_comm_monoid` ancestor, but the red paths in fig. 7.4 do

[10]With thanks to Mario Carneiro for this characterization.

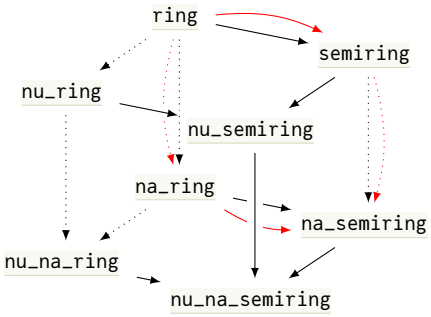


Figure 7.6.: An algebraic hierarchy where a suitable spanning tree placement can ensure all squares commute

The red paths highlight a square that was falsely claimed to not commute in [3]. `na` and `nu` are abbreviated from mathlib's `non_unital` and `non_assoc`(iative).

not because the least common ancestor in fig. 7.4b is `add_comm_monoid` while the least common ancestor in fig. 7.4c is `add_monoid`.

The version of this chapter published as [3] prematurely concluded from discussion in [56] that it is not in general possible to choose a spanning tree such that all pairs of paths commute, using as a counterexample the set of 8 typeclasses arranged in a cube in fig. 7.6, and claiming the red path did not commute. In fact, the red path does commute, and it can be checked that the other 5 faces also contain commuting paths. It remains an open question whether other counterexamples exist; though a brute-force search by Mario Carneiro on small graphs (after a graph-theoretical framing of the problem later in [56]) suggests they do not.

7.4.4. Ban non-root structures in dependent arguments

The problem in section 7.3.2 is caused by a typeclass argument to a typeclass being inferable both via "preferred" and "non-preferred" routes. In section 7.4.2, this can be worked around by ensuring every path is maximally "non-preferred". An alternative is to ensure that every path is "preferred", by only accepting typeclass arguments that appear as roots of the spanning subgraph. This could look like

```
class module (R M : Type)
  [has_zero R] [has_add R] [has_one R] [has_mul R]
  [has_zero M] [has_add M] :=
(smul : R → M → M)
-- (one_smul : ∀ (x : M), smul 1 x = x)
-- (mul_smul : ∀ (r s : R) (x : M), smul (r * s) x = smul r (smul s x))
-- (add_smul : ∀ (r s : R) (x : M), smul (r + s) x = smul r x + smul s x)
-- (zero_smul : ∀ (x : M), smul 0 x = 0)
```

where each of the operators for R and M is taken as a separate typeclass argument.

This approach has two main downsides: it results in larger proof terms, because now it has 6

typeclass arguments instead of four, which have to be resolved all the way down to the smallest typeclass instead of stopping part-way along the graph; and it doesn't extend to cases where not just the data fields carrying the operators on the type arguments, but also the proof fields carrying their properties, are needed to define the fields of the dependent typeclass.

7.5. Implications for packed structures

Up until this point we have focused only on typeclasses, as these are (at the time of writing) the idiomatic way to represent algebraic structure in Lean. While Coq also supports typeclasses, and the previous examples can be faithfully reproduced in it, this is not the idiomatic way to do things in MathComp.

Instead, Coq's "Hierarchy builder" [54, §4] generates "packed" structures [57] with a field for the type itself, rather than consuming the type as a parameter. These structures are then ineligible for typeclass search, but can be located automatically via "canonical structures" (or as they are known in Lean, "unification hints") instead. These can in fact be built on top of the typeclasses from section 7.2.1 or section 7.2.2:

```
structure packed_semiring := (carrier : Type) [semiring carrier]
structure packed_add_comm_monoid := (carrier : Type) [add_comm_monoid carrier]
```

A naïve encoding of a module in this packed view would be:

```
structure packed_module :=
(R : packed_semiring) (M : packed_add_comm_monoid) [module R.carrier M.carrier]
```

As `packed_module` has no parameters and is therefore not dependently-typed, it cannot fall afoul of the problem in section 7.3.2.

Unfortunately, this encoding is effectively useless mathematically [58, §3]; we have no way to talk about two modules over the same ring without something involving equality of types and operators[11] like `(V W : packed_module) (hVW : V.R = W.R)`; a much worse version of the duplicate `add` fields described at the start of section 7.2.2.

A more reasonable representation that avoids this problem is to only partially pack the structure, as

```
structure packed_module (R : packed_semiring) :=
(M : packed_add_comm_monoid M) [module R.carrier M.carrier]
```

which allows `(V W : packed_module R)`. This is roughly analogous to the approach taken in Coq's MathComp [59] and in mathlib's category theory library.

While this representation avoids the *specific* problem in section 7.3.2 due to its type not depending on the `add_comm_monoid` path (the red arrows in fig. 7.4), it is nonetheless dependently-typed. This make it vulnerable to an analogous problem where the diamond is instead formed by

[11] Or alternatively, by packing the ring and both modules into a single structure, as `(VW : packed_module₂) (v : VW.1) (w : VW.2)`. This is a viable approach for a module over two rings (as rarely are many rings needed), but doesn't scale for n modules over the same ring.

the `semiring` path (the blue arrows in fig. 7.4) after adding two new `comm_semiring` and `comm_ring` nodes.

Fortunately for MathComp, the "Hierarchy builder" uses *flat* packed structures[12], and so avoids these issues for the same reason that flat typeclasses do in section 7.3.1.

7.6. Related work

While this work is of course directly related to the work of porting Lean 3's mathlib to Lean 4, the lessons here are transferable to Coq (where [54] seemingly correctly chose to use flat structures by coincidence) and Agda (which has adopted structure η-reduction globally due to other motivations [55]); even if only to provide further understanding of why the respective choices that have already been made in those systems are the correct ones. To the author's awareness, no previously demonstrated *algebraic* motivations have been given for η-reduction in the kernel. Some in-depth analysis of "coherence" in algebraic typeclass paths is provided by [60, definition 3.3] (another name for our comparison in fig. 7.4), but it does not provide an example to show *why* η-reduction specifically should be assumed.

The analysis in sections 7.3 and 7.4 is only relevant to systems that use dependent type theory, as concerns of equalities between the values of type parameters cannot arise in a language that does not permit those parameters in the first place. The Isabelle proof assistant which uses simple type theory is therefore immune to this class of problem; and at any rate [61, §5.4] advocates avoiding its record types entirely for algebraic structure, in favor of using locales.

Algebraic hierarchies certainly do not only exist in proof assistants; they are an essential part of computer algebra systems too. However, most computer algebra systems do not make use of dependent types [62, §1], with a notable exception being the Axiom Library Compiler, Aldor. Despite supporting dependent types, the type system of Aldor is too restrictive for sections 7.3 and 7.4 to be relevant. Aldor does not implement definitional equality of types (referred to as "value-equality" by [62, §2.4]), and so falls at a much earlier hurdle than the one in section 7.3; it does not consider `Vector(2+3)` and `Vector(5)` to be the same type [62, §2.3], meaning that even fig. 7.4b would be considered a mismatch, and *every* square in fig. 7.6 would not commute.

This work focuses on how a seemingly innocuous implementation detail can be crucial to ensuring the success of *existing* approaches to algebraic hierarchies in dependently-typed proof assistants. The broader analysis of these hierarchies, and possible alternative designs (for which computer algebra systems can provide inspiration), is left to [45; 54; 58; 53].

[12] Presumably due to simplicity of implementation; there is no mention in [54] that using nested inheritance instead would have run into the issues described here.

7.7. Summary

In this chapter we have shown that for the "nested" approach to multiple inheritance to be viable in the context of dependently-typed typeclasses or packed structures, either we have to severely restrict how such inheritance is used (sections 7.4.2 to 7.4.4), or the kernel of the theorem prover must implement η-reduction for structures (section 7.4.1).

This scenario was a major stumbling block for mathlib's transition from Lean 3 to Lean 4, as typeclasses of this form are used extensively in linear algebra. This chapter provides a clear explanation of exactly what was going wrong, and a selection of various solutions that were considered before ultimately settling on the kernel change.

The code examples throughout this chapter, along with translations into Lean 4 and Coq, and the version information needed to run them, can be found at https://github.com/eric-wieser/lean-multiple-inheritance.

Part III.

Formalizations

8

Universal properties as a computational tool

> The purpose of computing is insight, not numbers.
>
> *(Richard Hamming)*

This chapter is adapted from "Computing with the Universal Properties of the Clifford Algebra and the Even Subalgebra" [13], which in turn is an extended version of [4].

One of the core claims of geometric algebra is that it lets you perform geometric manipulation in a "coordinate-free" way. Exactly what this means typically depends on the author, but a common interpretation is "the choice of basis vectors does not affect the result of the manipulation"[63]. For clarity, algorithms with this property will be referred to as "basis-agnostic".

Let us quickly summarize some examples of operations which are and are not "basis-agnostic". Basic algebraic operations like multiplication and the wedge product have a precise geometric meaning with no mention of coordinates, so are "basis-agnostic". However, the pseudoscalar $I = \prod_i e_i$ is not basis-agnostic; choosing the basis vectors in a different order results in a change of sign. In 3D this basis-dependence is transferred to the cross product, which in GA can be expressed as $a \times b = -I(a \wedge b)$; the handedness of the cross-product depends on the handedness of the vector space, which is determined by the basis.

There are some operations which despite being "basis-agnostic", are still typically defined by first making a choice of basis. This works well computationally, but can obscure insight mathematically. In particular, operations defined in terms of coordinates on a multivector basis can be difficult to rigorously show to be "coordinate-free"—that is, invariant with respect to the choice of basis—especially in large algebras.

This chapter explores the use of the "universal property" to ensure that operations on the Clifford algebra[1] are "coordinate-free" by construction. To build some insight for applying the universal property, this chapter will draw parallels to the process of writing recursive programs.

[1] In this chapter, we will use "Clifford algebra" to refer to the algebraic object, and "geometric algebra" to refer to the field of study.

After introducing some intuition for recursion in section 8.1, this chapter shows how the "universal property" can be used as a computational tool in the place of choosing a basis, and in section 8.2.1 demonstrates how to view this tool as a variant of recursion. Section 8.3 derives a new universal property for the even subalgebra from our first universal property, and uses this in sections 8.3.1 and 8.3.2 to construct two well-known isomorphism in an unusual way. Section 8.4 shows how this recursive technique can be applied to construct the left-contraction from one of its properties alone. While this chapter contains no Lean code, Lean versions of the results are provided throughout this chapter via "↗" links, which lead to the snapshot repository submitted with [13].

8.1. Recursors

In functional programming, a list is usually defined inductively; either it is empty, "[ ]", or it is an element a followed by another list l, "$a :: l$". This inductive definition provides a recursion principle, or recursor: "To define a function from a list of elements, it suffices to define its value on [ ], and define its value on $a :: l$ given its value on l". Consider computing in this way a sum of a list of elements of a ring R. In a functional programming language, we would usually do so as:↗

$$\mathrm{sum} : \mathrm{list}\, R \to R \tag{8.1}$$

$$\mathrm{sum}([\,]) \coloneqq 0 \tag{8.1a}$$

$$\mathrm{sum}(a :: l) \coloneqq a + \mathrm{sum}(l) \tag{8.1b}$$

One way to describe the list recursor is as a "fold"; if we have a function $f : \alpha \to \beta \to \beta$, then $\mathrm{fold}[f] : \mathrm{list}\,\alpha \to \beta \to \beta$. This satisfies $\mathrm{fold}[f]([\,], b_0) = b_0$ and $\mathrm{fold}[f](a :: l, b_0) = f(a, \mathrm{fold}[f](l, b_0))$. The "pattern matching" in eq. (8.1) can be trivially transformed by the compiler into an application of $\mathrm{fold}[f]$, as $\mathrm{sum}(l) = \mathrm{fold}[a \mapsto v \mapsto a + v](l, 0)$, where $\mathrm{sum}(l)$ in eq. (8.1b) has been replaced with v.

Sometimes implementing a recursion scheme requires keeping track of intermediate state. As an example, consider producing an accumulated sum of the elements of a list, starting from zero, such that $\mathrm{accum}([a, b]) = [0, a, a + b]$. To implement this, we use the recursor to define an auxiliary helper function:↗

$$\mathrm{accum\_from} : \mathrm{list}\, R \to R \to \mathrm{list}\, R \tag{8.2}$$

$$\mathrm{accum\_from}([\,]) \coloneqq a \mapsto [a] \tag{8.2a}$$

$$\mathrm{accum\_from}(b :: l) \coloneqq a \mapsto a :: \mathrm{accum\_from}(l)(a + b) \tag{8.2b}$$

Note the unusual type signature in eq. (8.2); for each list, it produces not a value but another function. It is this function that consumes our intermediate state $a : R$, which is the value to resume the accumulation from; allowing us to thread this value through the recursion while still

sticking to the rules of our list recursor. We can recover our desired function by simply initializing this state:

$$\text{accum} : \ \text{list}\, R \to \text{list}\, R \tag{8.3}$$
$$\text{accum} \coloneqq \quad l \mapsto \text{accum\_from}(l)(0) \tag{8.3a}$$

Finally, let us consider the example where the running sum should be in reverse. As we recurse, we will keep track of what the first element of our list is. Instead of introducing this intermediate state into the input as we did in eq. (8.2) by producing a function, we introduce it into the output by producing a pair ($\times$):

$$\text{rev\_accum}_{\text{aux}} : \text{list}\, R \to \quad (R \times \text{list}\, R) \tag{8.4}$$
$$\text{rev\_accum}_{\text{aux}}([\,]) \coloneqq \quad (0, [\,]) \tag{8.4a}$$
$$\text{rev\_accum}_{\text{aux}}(a :: l) \coloneqq (a + b, b :: l') \text{ where } (b, l') \coloneqq \text{rev\_accum}_{\text{aux}}(l) \tag{8.4b}$$

Instead of initializing the state as in eq. (8.3a), we post-process it:

$$\text{rev\_accum} : \ \text{list}\, R \to \text{list}\, R \tag{8.5}$$
$$\text{rev\_accum} \coloneqq \quad l \mapsto a :: l' \text{ where } (a, l') \coloneqq \text{rev\_accum}_{\text{aux}}(l) \tag{8.5a}$$

In this section, we have seen two important tricks for taking a simple recursor and implementing more complex recursion schemes. In the rest of this chapter, we will show how these principles translate to the language of universal properties.

8.2. The universal property of the Clifford Algebra

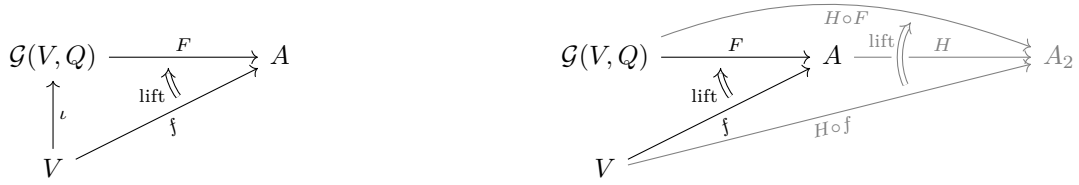
To state the universal property of the Clifford algebra [64, §14.4; 65, p. II.1.1], we will need the terminology of R-modules and R-algebras from abstract algebra, as introduced in section 2.2.1. We say an "R-algebra morphism" is an R-linear map that additionally preserves multiplication and 1, the multiplicative identity. Armed with these definitions, we can state the universal property,

Theorem 8.1 (Universal property of the Clifford algebra)**.** *For every R-algebra A and R-module V, and a quadratic form $Q : V \to R$, we have a one-to-one correspondence between:*

linear maps $f : V \to A$		*algebra morphisms*
satisfying $f(v)^2 = Q(v)$	*and*	$F : \mathcal{G}(V, Q) \to A$.

This correspondence is compositional; given another R-algebra A_2 and an algebra morphism $H : A \to A_2$, if f corresponds with F then $v \mapsto H(f(v))$ corresponds with $x \mapsto H(F(x))$. We write this correspondence as $\text{lift}[f] = F$.

It can be helpful to present this graphically, as is done in fig. 8.1.



(a) via the map ι

(b) via the property $\text{lift}[H \circ f] = H \circ \text{lift}[f]$

Figure 8.1.: Graphical representation of two equivalent ways to define the universal property, where (b) corresponds to theorem 8.1.

Setting $F = (x \mapsto x)$ in (b) recovers (a). In this chapter we will mostly elide ι, leaving it implicit wherever we turn an element of V into an element of $\mathcal{G}(V, Q)$.

Proof. The details depend on the construction of $\mathcal{G}(V, Q)$; if defined as a quotient of the tensor algebra $\mathcal{T}(V)$ such as in [65], then it follows from the universal property of $\mathcal{T}(V)$; if constructed from a basis on V, then $\text{lift}[f]$ can be defined trivially (as we briefly elaborate on in eq. (8.10)); if "defined"[2] via category theory as in [64, §14.4], then theorem 8.1 is true by definition. □

To give a concrete example of the behavior of $F = \text{lift}[f]$, for $u, v, w : V$ we have $F(1 + 2uv + 3w) = 1 + 2F(u)F(v) + 3F(w) = 1 + 2f(u)f(v) + 3f(w)$, where the first equality follows from properties of algebra morphisms, and the second follows from reading off $F(\iota(v)) = f(v)$ from fig. 8.1a.

As this chapter is about computation, we will not just use the fact that a suitable $\text{lift}[f]$ is known to exist, but will also assume that the construction of $\mathcal{G}(V, Q)$ provides a $\text{lift}[f]$ with computational content (i.e. an algorithm for converting between f and F), just as we assumed that the recursor for lists did in section 8.1. This will permit us to define further computations in terms of $\text{lift}[f]$.

8.2.1. Universal properties as recursors

In eq. (8.2), we saw a trick to thread extra state through our recursor by choosing our output to itself be a function. We can play a similar trick with the universal property, although we are forced to work within the functions that form an algebra. These include the endomorphism algebra $End_R(W)$ (the R-linear maps of the form $W \to W$); where $1 : End_R(W)$ is the identity map and $\times : End_R(W) \to End_R(W) \to End_R(W)$ is composition. The scalars of this algebra happen to also be the "scaler"s; that is, the canonical map $R \to End_R(W)$ is chosen such that the image of $r : R$ is the endomorphism corresponding to a uniform scaling by r.

This specialization to $A = End_R(W)$ allows us to apply the universal property to produce a "fold" operation (so named due to its analogy to the list version described just below eq. (8.1)) by an R-bilinear map $f : V \to W \to W$ to obtain an algebra morphism into the endomorphism

[2] Strictly speaking, the categorical approach only "describes" Clifford algebras, stopping short of demonstrating that objects satifying the description exist.

algebra ([mathlib#14619]):

$$\text{fold[f]} : \mathcal{G}(V, Q) \to \overbrace{W \to W}^{End_R(W)} \tag{8.6}$$

$$\text{fold[f]}(v : V) \coloneqq \ w \mapsto f(v, w) \tag{8.6a}$$

where the $f(v)^2 = (w \mapsto f(v, f(v, w))) = Q(v)$ condition can be rewritten as

$$f(v, f(v, w)) = Q(v)w \tag{8.7}$$

Similarly, the fact this is an algebra morphism tells us that $\text{fold}[f](r, v) = rv$ for $r : R$ and $\text{fold}[f](xy, v) = \text{fold}[f](x, \text{fold}[f](y, v))$. As an example of what "fold"ing means in the context of a Clifford algebra, if $c + u + vw : \mathcal{G}(V, Q)$ and $x : W$ then

$$\text{fold}[f](c + u + vw, x) = cx + f(u, x) + f(v, f(w, x)). \tag{8.8}$$

8.2.2. Universal properties as a universal interface

If two different representations of a Clifford algebra are available, $\mathcal{G}_1$ and $\mathcal{G}_2$, then the universal property of $\mathcal{G}_1$ provides a map between the two:

$$\text{convert} : \mathcal{G}_1(V, Q) \to \mathcal{G}_2(V, Q) \tag{8.9}$$

$$\text{convert}(v : V) \coloneqq v \quad (= \iota_2(v)) \tag{8.9a}$$

In the language of software; if two libraries implement the universal property "API", then they can interoperate without direct knowledge of each other. For instance, a library L_C that implements its multivectors as a series of coefficients a^i in a chosen basis e_i, such as storing $A_C = a + a^1 e_1 + a^2 e_2 + a^{12} e_{12}$ as $[a, a^1, a^2, a^{12}]$, can implement the forward direction of the universal property as

$$\text{lift}_{L_C}[f](A_C) = a + a^1 f(e_1) + a^2 f(e_2) + a^{12} f(e_1) f(e_2) \tag{8.10}$$

Similarly, if another library L_X with some other representation also provides an implementation of the universal property, then we can use it to convert a multivector A_X from L_X into a multivector A_C in L_C as

$$A_C = \text{lift}_{L_X}[(a^1 e_1 + a^2 e_2) \mapsto [0, a^1, a^2, 0]](A_X) \tag{8.11}$$

Note this still requires the two libraries to agree upon a representation of the vector space (here, $a^1 e_1 + a^2 e_2$).

This construction is a computational interpretation of a result from category theory, that

$\mathcal{G}(V, Q)$ is functorial over V. As a functor, this takes a linear map (or isomorphism) $f : V \to W$ that preserves the quadratic form as $Q_W(f(v)) = Q_V(v)$ and extends it to an algebra map (or isomorphism) $F : \mathcal{G}(V, Q_V) \to \mathcal{G}(W, Q_W)$ by taking $F(\iota(v)) = \iota(f(v))$. This operation is very similar to the "outermorphism" [1, §4.2] which it itself a functor which produces instead a morphism in the exterior algebra, $F : \bigwedge(V) \to \bigwedge(W)$.

8.2.3. Elementary GA operations via the universal property

For one of the simplest examples of applying the universal property, consider using $f : V \to \mathcal{G}(V, Q) = (v \mapsto -v)$, which trivially satisfies $f(v)^2 = (-v)^2 = v^2 = Q(v)$; the resulting lift$[f]$ is the familiar "grade involution" operator $x \mapsto \hat{x}$. A key insight is that we can write this in the style of eq. (8.1) as:

$$\text{grade\_invol} : \mathcal{G}(V, Q) \to \mathcal{G}(V, Q) \tag{8.12}$$

$$\text{grade\_invol}(v : V) \coloneqq -v \tag{8.12a}$$

Note that unlike in eq. (8.1), we are additionally obliged to show that eq. (8.12a) is linear, and that $(-v)^2 = v^2 = Q(v)$.

A more complex example involves constructing the grade reversal operation $x \mapsto \tilde{x}$, which for example sends $e_1e_2e_3$ to $e_3e_2e_1$. For this case, we need a different choice of A than $\mathcal{G}(V, Q)$. What we choose is $\mathcal{G}(V, Q)^{\text{op}}$, where A^{op} is the algebra A but with multiplication reversed. This comes with two obvious R-linear maps, $\text{op} : A \to A^{\text{op}}$ and $\text{op}^{-1} : A^{\text{op}} \to A$, which convert between the two spaces. Note that $\text{op}(ab) = (\text{op}\, b)(\text{op}\, a)$, so these are not algebra morphisms; but we do still have $\text{op}\, 1 = 1$. Using the notation of eq. (8.12), we implement this in the style of eq. (8.4) as:

$$\text{grade\_rev}_{\text{aux}} : \mathcal{G}(V, Q) \to \mathcal{G}(V, Q)^{\text{op}} \tag{8.13}$$

$$\text{grade\_rev}_{\text{aux}}(v : V) \coloneqq \text{op}\, v \tag{8.13a}$$

Again, we must show that eq. (8.13a) is linear, and that $(\text{op}\, v)^2 = \text{op}(v^2) = \text{op}(Q(v)) = Q(v)$. To recover the reversion operator (a linear map that reverses multiplication), we simply compose this with op^{-1} to eliminate the $^{\text{op}}$.

$$\text{grade\_rev} : \quad \mathcal{G}(V, Q) \to \mathcal{G}(V, Q) \tag{8.14}$$

$$\text{grade\_rev} \coloneqq \qquad x \mapsto \text{op}^{-1}(\text{grade\_rev}_{\text{aux}}(x)) \tag{8.14a}$$

While these applications of the universal property let us implement computations, the universal property can also be used to assemble proofs. One of the most direct ways to do so is to use the fact that universal properties provide an equivalence, and to pull an equality across this equivalence. In the context of theorem 8.1, this allows us to prove that two algebra morphisms

are equal by passing both through the injective lift$[\cdot]$ operation, which amounts to theorem 8.2.

Theorem 8.2. *To show that two algebra morphisms $f, g : \mathcal{G}(V, Q) \to A$ are equal, it suffices to show they agree on the generators $v : V$: that $f \circ \iota = g \circ \iota$.*

In general, this strategy can be used to construct extensionality theorems from most universal properties.

A more complex example is the induction principle in theorem 8.3.

Theorem 8.3. *To show a property $P(x)$ for all elements of the Clifford algebra $\mathcal{G}(V, Q)$, it suffices to show:*

- *That $P(r)$ holds on all scalars $r : R$*
- *That $P(u)$ holds on all vectors $u : V$*
- *That $P(x + y)$ holds on all multivectors $x, y : \mathcal{G}(V, Q)$ if $P(x)$ and $P(y)$ hold*
- *That $P(xy)$ holds on all multivectors $x, y : \mathcal{G}(V, Q)$ if $P(x)$ and $P(y)$ hold*

The proof requires some careful bookkeeping, which is best left to the Lean version in section 9.3.4.

8.3. The universal property of the even subalgebra

The even subalgebra $\mathcal{G}^+(V, Q)$ of a Clifford algebra $\mathcal{G}(V, Q)$ is the subalgebra consisting of the closure under addition and multiplication of all elements of the form vw where $v, w : V$; its members are known [14, (1.29)] as the "even" multivectors[3]. We will now show that this subalgebra has its own universal property, theorem 8.4:

Theorem 8.4 (Universal property of the even subalgebra)**.** *For every R-algebra A and R-module V, we have a one-to-one correspondence between:*

R-bilinear maps $f : V \to V \to A$ satisfying:			*algebra morphisms*
$f(v, v) = Q(v)$	(8.15)	*and*	*out of the even subalgebra*
$f(u, v)f(v, w) = Q(v)f(u, w)$	(8.16)		$F : \mathcal{G}^+(V) \to A$.

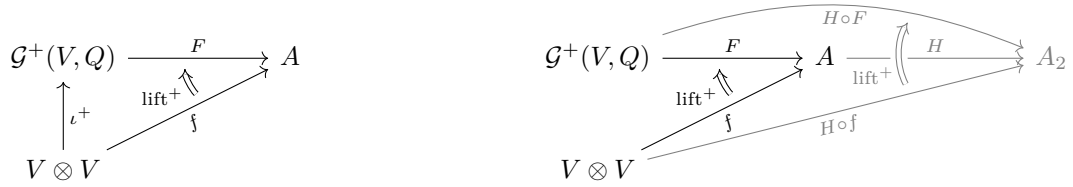
This correspondence is compositional; if f corresponds with F then $v \mapsto w \mapsto H(f(v, w))$ corresponds with $x \mapsto H(F(x))$. We write this correspondence as $\text{lift}^+[f] = F$.

Again, it can be helpful to present this graphically, as is done in fig. 8.2.

We are not just going to prove that there *is* a correspondence (as [66, Theorem 3.3] does for the less general case of a non-degenerate quadratic form over a field), but will provide an explicit basis-agnostic computation of that correspondence in terms of lift$[f]$ from theorem 8.1 (for we assume an algorithm is provided).

We will start by showing the reverse direction, which given the algebra morphism F :

[3] Although the definition in [14] needs the construction in section 8.4 and therefore doesn't work in characteristic 2.



(a) via the inclusion ι^+

(b) via the property $\text{lift}^+[H \circ f] = H \circ \text{lift}[f]$

Figure 8.2.: The universal property of the even subalgebra

So as to resemble fig. 8.1, we show the bilinear map $f : V \to V \to A$ as the equivalent linear map from the tensor product, $f : V \otimes V \to A$. Here, $\iota^+(v \otimes w) = \iota(v)\iota(w) = vw$.

$\mathcal{G}^+(V,Q) \to A$ we choose as

$$\text{lift}^{+\,-1}[F] = f = (v \mapsto w \mapsto F(vw)) \tag{8.17}$$

which trivially satisfies theorem 8.4:

$$f(v,v) = F(vv) = F(Q(v)) = Q(v) \tag{8.18}$$

$$f(u,v)f(v,w) = F(uv)F(vw) = F(uvvw) = F(uQ(v)w) = Q(v)F(uw) \tag{8.19}$$

$$= Q(v)f(u,w) \tag{8.20}$$

To construct the forwards direction, we are going to use the same trick as we did in eq. (8.6), setting $W = A \oplus S$ to produce an auxiliary function $\text{lift}^+_{\text{aux}}[f] : \mathcal{G}(V,Q) \to (A \oplus S) \to (A \oplus S)$. Here, A is our target algebra, while S is some additional state which mirrors the extra recursor state we saw in eq. (8.4). $A \oplus S$ is their direct sum, which is to say it consists of pairs (a,s) with $(a_1,s_1) + (a_2,s_2) = (a_1 + a_2, s_1 + s_2)$ and $r(a,s) = (ra, rs)$. Note that for this to be an R-module as required by theorem 8.4, we need S to also be an R-module. We will deduce precisely what to choose for S shortly.

We want our fold to apply f on pairs of vectors $v, w : V$ at a time; that is,

$$\text{lift}^+_{\text{aux}}[f](vwx, (a,s)) = (f(v,w)\,\text{lift}^+_{\text{aux}}[f](x,(a,s)), s'). \tag{8.21}$$

Using the fact that $\text{lift}^+_{\text{aux}}[f]$ will be an algebra morphism, this simplifies to

$$\text{lift}^+_{\text{aux}}[f](v, \text{lift}^+_{\text{aux}}[f](w,(a,s))) = (f(v,w)a, s'); \tag{8.22}$$

that is, each application of $\text{lift}^+_{\text{aux}}[f]$ needs to apply "half" of f. An obvious choice would be to pick $S = V \to A$, the space of R-linear maps which includes the "half"-applied maps like $v \mapsto f(v,w)a$. Note that this is essentially using the trick in eq. (8.2) for a second time, but instead of producing an unconstrained function we are required to produce a linear map. We can

then define

$$\text{lift}^+_{\text{aux}}[f] : \mathcal{G}(V,Q) \to (A \oplus S) \to (A \oplus S) \tag{8.23}$$
$$\text{lift}^+_{\text{aux}}[f](v : V) := \quad (a,s) \mapsto (s(v), w \mapsto f(w,v)a), \tag{8.23a}$$

where the second component of the pair contains a partially-applied version of f, while the first component finishes off the invocation from the previous iteration. Note that we cannot take the product of $s(\cdot)$ and a in a single step as then this operation would cease to be linear; which is why we instead weave these terms back and forth between the left and right halves of the pair.

We now verify that our $\text{lift}^+_{\text{aux}}[f]$ satisfies the required property in eq. (8.7) as

$$\begin{aligned}
\text{lift}^+_{\text{aux}}[f](v, \text{lift}^+_{\text{aux}}[f](v,(a,s))) &= \text{lift}^+_{\text{aux}}[f](v,(s(v), w \mapsto f(w,v)a)) && (8.24)\\
&= (f(v,v)a, w \mapsto f(w,v)s(v))) && (8.25)\\
&\overset{?}{=} (Q(v)a, w \mapsto Q(v)s(w)) && (8.26)\\
&= Q(v)(a,s), && (8.27)
\end{aligned}$$

where $\overset{?}{=}$ is the equality to be checked. $f(v,v) = Q(v)$ was theorem 8.4, so we can easily match up the first half of the pair. However, matching up the second half of the pair requires $f(w,v)s(v) = Q(v)s(w)$, which is a stronger requirement than theorem 8.4 and not true for all linear maps $s : S$.

To solve this problem, we need to pick a smaller space S, the module spanned by linear maps $s : V \to A$ of the form $s = (v \mapsto f(v,w)a)$ for all $w : V$ and $a : A$. Our definition of $\text{lift}^+_{\text{aux}}$ in eq. (8.23a) trivially adapts to this definition, as $w \mapsto f(w,v)a$ lies in the new S by definition. We are now in a position to solve $f(w,v)s(v) = Q(v)s(w)$, as we can write $s = (w \mapsto \sum_i f(w,u_i)a_i)$ (for some arbitrary finite set of $u_i : V$ and $a_i : A$) to get:

$$\begin{aligned}
f(w,v)\left(\textstyle\sum_i f(v,u_i)a_i\right) &= \textstyle\sum_i f(w,v)f(v,u_i)a_i && (8.28)\\
&= \textstyle\sum_i Q(v)f(w,u_i)a_i && (8.29)\\
&= Q(v)\textstyle\sum_i f(w,u_i)a_i && (8.30)\\
&= Q(v)s(w) && (8.31)
\end{aligned}$$

where we go from eq. (8.28) to eq. (8.29) using theorem 8.4.

We can now extract $\text{lift}^+[f]$ as

$$\text{lift}^+[f] : \ \mathcal{G}^+(V,Q) \to A \tag{8.32}$$
$$\text{lift}^+[f] := \quad x^+ \mapsto a \text{ where } (a,s') = \text{lift}^+_{\text{aux}}[f](x,(1, v \mapsto 0)) \tag{8.32a}$$

This is obviously linear, as it is the composition of operations each of which is linear. We can show that $\text{lift}^+[f](r) = r$ by using properties of eq. (8.6). To complete the proof that this is an

algebra morphism, we must show that within the even subalgebra it preserves multiplication. We will do this by induction, for which we need theorem 8.5.

Theorem 8.5. *To show a property $P(x)$ for all elements of the even subalgebra $\mathcal{G}^+(V, Q)$, it suffices to show:*

- *That $P(r)$ holds on all scalars $r : R$*
- *That $P(x + y)$ holds on all elements $x, y : \mathcal{G}^+(V, Q)$ if $P(x)$ and $P(y)$ hold*
- *That $P(uvx)$ holds on all elements $u, v : V$ and $x : \mathcal{G}^+(V, Q)$ if $P(x)$ holds*

Proof. Follows by noting that $\mathcal{G}^+(V, Q)$ is the vector space spanned by all products of even numbers of vectors in V, that such products can be decomposed into pairs, and through an appropriate induction principle for spans of vectors. □

We apply this principle with $P(x)$ as $\forall y, \text{lift}^+[f](xy) = \text{lift}^+[f](x)\,\text{lift}^+[f](y)$. The first two conditions follow trivially by $\text{lift}^+[f](r) = r$ and linearity. The third condition can be shown as

$$\begin{aligned}
\text{lift}^+[f](vwy) &= a && \text{where } (a, s') = \text{lift}^+_{\text{aux}}[f](vwy, (1, v \mapsto 0)) && (8.33)\\
&= f(v, w)a && \text{where } (a, s') = \text{lift}^+_{\text{aux}}[f](y, (1, v \mapsto 0)) && (8.34)\\
&= f(v, w)\,\text{lift}^+[f](vwy) && && (8.35)\\
&= \text{lift}^+[f](vw)\,\text{lift}^+[f](y) && && (8.36)
\end{aligned}$$

We now have $\text{lift}^+[f] : \mathcal{G}^+(V, Q) \to A$, the forward direction of the universal property. Combined with our result in eq. (8.17), all that remains is to show that these two directions are inverses. Showing that this operation is a left-inverse ($\text{lift}^{+\,-1}[\text{lift}^+[f]] = f$) is straightforward. Showing that it is a right-inverse requires the extensionality principle in theorem 8.6, which we use in eq. (8.38).

$$\begin{aligned}
&\text{lift}^+[\text{lift}^{+\,-1}[F]](vw) = \text{lift}^+[F](v, w) = F(vw) && (8.37)\\
\implies\ &\text{lift}^+[\text{lift}^{+\,-1}[F]] = F && (8.38)
\end{aligned}$$

Theorem 8.6. *To show that two algebra morphisms from the even subalgebra $f, g : \mathcal{G}^+(V, Q) \to A$ are equal, it suffices to show they agree on the products of two generators $v, w : V$.*

Proof. Rephrase as $\forall x, f(x) = g(x)$, apply theorem 8.5, and use the properties of algebra homomorphisms. □

8.3.1. The isomorphism with the even subalgebra

A well known $\mathbb{R}$-algebra isomorphism in geometric algebra [67, corollary 15.35] is that between $\mathcal{G}^+(\mathbb{R}^{p,q+1,r})$ and $\mathcal{G}(\mathbb{R}^{p,q,r})$, often introduced in connection with the complex numbers as $\mathcal{G}(\mathbb{R}^{0,1}) \cong$

$\mathbb{C} \cong \mathcal{G}^+(\mathbb{R}^{0,2})$ or in connection with the quaternions as $\mathcal{G}(\mathbb{R}^{0,2}) \cong \mathbb{H} \cong \mathcal{G}^+(\mathbb{R}^{0,3})$. We will now construct the general basis-free version of this result.

Theorem 8.7. *$\mathcal{G}(V, Q)$ is isomorphic as an R-algebra to $\mathcal{G}^+(V \oplus R, Q')$, where $Q'((v, r)) \coloneqq Q(v) - r^2$.*

Here $V \oplus R$ combines V (as elements of the form $(v, 0)$) with an extra basis vector $e = (0, 1)$ that squares to -1, and so we will write (v, r) as $v + re$. In [68, Chapter 1, Theorem 3.7], this isomorphism is evaluated by choosing a basis for V, and then copying coefficients by inspection: in the forward direction, each basis vector e_i is replaced with ee_i; and in the backwards direction[4], where all basis vectors appear in pairs, $e_i e_j$ is left alone and $e_i e$ is mapped back to e_i. We will proceed without choosing a basis for V, and use our pair of universal properties instead.

Proof. To construct the forward map, we can directly write down the coefficient copying approach by applying theorem 8.1 with $f = (v \mapsto ev)$, which satisfies $f(v)^2 = (ev)(ev) = (-ve)(ev) = -v(-1)v = Q(v)$. This gives us $F : \mathcal{G}(V, Q) \to \mathcal{G}^+(V, Q') = \text{lift}[f]$, and is exactly the approach used in [68, Chapter 1, Theorem 3.7]. This reference does not give an explicit construction for the reverse mapping, noting that to verify one exists we must "check $[F]$ on a linear basis".

The reverse map f^{-1} needs to satisfy the pair of rules above:

$$f^{-1}(0 + e,\ \ 0 + e) = -1 \qquad (\text{as } e^2 = -1 \text{ in } \mathcal{G}^+(V, Q')) \tag{8.39}$$

$$f^{-1}(0 + e, v + 0e) = v \qquad (\text{remove } e \text{ from pairs of the form } ev) \tag{8.40}$$

$$f^{-1}(u + 0e,\ \ 0 + e) = -u \qquad (\text{rewrite } ue = -eu \text{ and do the above}) \tag{8.41}$$

$$f^{-1}(u + 0e, v + 0e) = uv \qquad (\text{leave blades without } e \text{ untouched}) \tag{8.42}$$

$$\implies f^{-1}(u + re, v + se) = (u + r)(v - s) \tag{8.43}$$

where eq. (8.43) follows by linearity. Theorem 8.4 holds as $f^{-1}(v + se, v + se) = v^2 - s^2 = Q'(v + se)$ and theorem 8.4 follows similarly. We can thus apply theorem 8.4 with this f to obtain $F = \text{lift}^+[f^{-1}]$; or in our functional notation:

$$F^{-1} : \mathcal{G}^+(V, Q') \to \mathcal{G}(V, Q) \tag{8.44}$$

$$F^{-1}((u + re)(v + se)) \coloneqq f^{-1}(u + re, v + se) = (u + r)(v - s) \tag{8.44a}$$

All that remains to conclude our construction of this isomorphism is to show that these operations are inverses, that is for all $x : \mathcal{G}(V, Q)$ and $x^+ : \mathcal{G}^+(V, Q)$,

$$F^{-1}(F(x)) = x, \qquad F(F^{-1}(x^+)) = x^+. \tag{8.45}$$

[4] For which [68] proves only existence.

Rewriting eq. (8.45) as equalities of functions gives

$$(x \mapsto F^{-1}(F(x))) = (x \mapsto x), \qquad (x^+ \mapsto F(F^{-1}(x^+)) = (x^+ \mapsto x^+), \tag{8.46}$$

which allows us to apply theorems 8.2 and 8.6 to solve these equations:

$$\begin{aligned} & F^{-1}(F(v)) \\ &= F^{-1}(f(v)) \\ &= F^{-1}(ev) \\ &= f^{-1}(e, v) \\ &= (0+1)(v+0) \\ &= v \end{aligned} \qquad \begin{aligned} & F(F^{-1}((u+re)(v+se))) \\ &= F(f^{-1}(u+re, v+se)) \\ &= F((u+r)(v-s)) \\ &= (f(u)+r)(f(v)-s) \\ &= euev + rev - seu - rs \\ &= uv + rev - seu + rse^2 \\ &= (u+re)(v+se) \end{aligned} \tag{8.47}$$

□

8.3.2. The isomorphism between even subalgebras of negated quadratic forms

Sometimes, the isomorphism in section 8.3.1 is instead stated with p and q exchanged [67, corollary 15.35] as $\mathcal{G}^+(\mathbb{R}^{p+1,q,r}) \cong \mathcal{G}(\mathbb{R}^{q,p,r})$, as this gives $\mathcal{G}(\mathbb{R}^{0,2}) \cong \mathbb{H} \cong \mathcal{G}^+(\mathbb{R}^3)$ where the right side is more obviously $SO(3)$. Rather than repeat the construction in section 8.3.2, we can instead show that in general $\mathcal{G}^+(\mathbb{R}^{p,q,r}) \cong \mathcal{G}^+(\mathbb{R}^{q,p,r})$, from which the desired result follows by composition.

To construct this, we shall once again proceed in a basis-free manner to demonstrate another application of theorem 8.4.

Theorem 8.8. *$\mathcal{G}^+(V, Q)$ is isomorphic as an R-algebra to $\mathcal{G}^+(V, -Q)$, where $(-Q)(v) \coloneqq -Q(v)$.*

Proof. We begin by defining the bilinear map $f_Q : V \to V \to G^+(V, -Q)$ as $f(u, v) = -uv$, which trivially satisfies theorem 8.4:

$$f_Q(v, v) = -Q(v) = (-Q)(v) \tag{8.48}$$

$$f_Q(u, v) f_Q(v, w) = (-uv)(-vw) = uQ(v)w \tag{8.49}$$

$$= (-Q(v))(-uw) = (-Q)(v) f_Q(u, w). \tag{8.50}$$

This therefore gives us the map $\text{lift}^+[f_Q] : G^+(V, Q) \to G^+(V, -Q)$ as the forward direction of our isomorphism; and $\text{lift}^+[f_{-Q}] : G^+(V, -Q) \to G^+(V, Q)$ as the reverse direction.

By symmetry we only need to check that $\text{lift}^+[f_Q] \circ \text{lift}^+[f_{-Q}] = (x \mapsto x)$ to conclude that this

is an isomorphism. Noting that

$$\text{lift}^+[f_Q](\text{lift}^+[f_{-Q}](uv)) = \text{lift}^+[f_Q](-uv) = -(-uv) = uv, \tag{8.51}$$

we conclude the desired result via theorem 8.6. □

8.4. The isomorphism to the exterior algebra

A key result in geometric algebra is that the Clifford algebra is isomorphic as an R-module to the exterior algebra over the same vector space, as this provides the non-metric wedge product; or stated another way, $\mathcal{G}(V, Q) \cong \mathcal{G}(V, 0)$. In this section, we shall show how to construct this isomorphism using the universal property (as is done implicitly in [69, eq. 22]).

In [70, Theorem 34], this is shown in the more general case $\mathcal{G}(V, Q_1) \cong \mathcal{G}(V, Q_2)$, by defining[5] $v\rfloor^f : \mathcal{T}(V) \to \mathcal{T}(V)$ and $\alpha^f : \mathcal{T}(V) \to \mathcal{T}(V)$ for some bilinear form $f : V \to V \to R$, characterized by [70, Theorems 6 and 21] on $u : V$ and $U : \mathcal{T}(V)$ as

$$v \rfloor^f (u \otimes U) = f(v, u)U - u \otimes (v \rfloor^f U), \tag{8.52}$$

$$\alpha^f(u \otimes U) = u \otimes \alpha^f(U) - u\rfloor^f(\alpha^f(U)). \tag{8.53}$$

Note that in [70] these are defined on the tensor algebra; only later is it proved that under suitable assumptions on f, these mappings can be transferred to a Clifford algebra where the $\otimes$ is simply multiplication[6]. Overloading the notation from eqs. (8.52) and (8.53), this transferral provides the mappings $v \rfloor^f : \mathcal{G}(V, Q_1) \to \mathcal{G}(V, Q_1)$ and $\alpha^f : \mathcal{G}(V, Q_1) \to \mathcal{G}(V, Q_2)$, characterized on $u : V$ and $U : \mathcal{G}(V, Q_1)$ by the very similar:

$$v \rfloor^f (uU) = f(v, u)U - u(v \rfloor^f U) \tag{8.54}$$

$$\alpha^f(uU) = u\alpha^f(U) - u\rfloor^f(\alpha^f(U)) \tag{8.55}$$

Without repeating the entire proof here, we will proceed by showing how to directly construct eqs. (8.54) and (8.55) rather than going via eqs. (8.52) and (8.53).

To warm up, we shall first construct eq. (8.55) via an auxiliary function

$$\alpha^f_{\text{aux}} : \mathcal{G}(V, Q_1) \to \mathcal{G}(V, Q_2) \to \mathcal{G}(V, Q_2) \tag{8.56}$$

$$\alpha^f_{\text{aux}}(u : V) := \qquad x \mapsto ux - v\rfloor^f x \tag{8.56a}$$

from which we recover

$$\alpha^f(x) = \alpha^f_{\text{aux}}(1). \tag{8.57}$$

[5] Confusingly, [70] uses $u \llcorner U$ as notation for $u\rfloor U$, with the symbol flipped.
[6] Equation (8.54) also appears as a special case of [14, (1.41a)] with $r = 1$.

This could alternatively be written as a simple application of "fold" in eq. (8.6) as $\alpha^f(U) = \text{fold}[u \mapsto x \mapsto ux - u \rfloor^f x](U, 1)$. We must again show that eq. (8.56a) is linear in u and x, which is straightforward, and eq. (8.7), which is less so:

$$\begin{aligned}
\alpha^f_{\text{aux}}(u, \alpha^f_{\text{aux}}(u, x)) &= u(ux - u \rfloor^f x)) - u \rfloor^f (ux - u \rfloor^f x) && (8.58)\\
&= uux - u(u \rfloor^f x) - u \rfloor^f (ux) + u \rfloor^f (u \rfloor^f x) && (8.59)\\
&= Q_2(u)x - u(u \rfloor^f x) - (f(u, u)x - u(u \rfloor^f x)) + 0 && (8.60)\\
&= (Q_2(u) - f(u, u))x && (8.61)\\
&\stackrel{?}{=} Q_1(u)x. && (8.62)
\end{aligned}$$

For the $\stackrel{?}{=}$ line to hold, we need to define f such that $f(u, u) = (Q_2 - Q_1)(u) = \delta Q(u)$. A typical choice of f is the bilinear form associated with δQ, that is $f(x, y) = \frac{1}{2}(\delta Q(x+y) - \delta Q(x) - \delta Q(y))$; thus imposing the restriction that the commutative ring R is not of characteristic 2.

Defining $v \rfloor^f$ is more troublesome; we cannot directly use this same trick from "fold" in eq. (8.6) here, as we not only need the current vector "u" and the result so far "$v \rfloor^f U$", but we also need the accumulation of the input so far, "U". The solution is to first apply the trick in eq. (8.4), where we compute the value of U as we go along:

$$\begin{aligned}
(v \rfloor^f)_{\text{aux}} &: \mathcal{G}(V, Q) \to \mathcal{G}(V, Q) \oplus \mathcal{G}(V, Q) \to \mathcal{G}(V, Q) \oplus \mathcal{G}(V, Q) && (8.63)\\
(v \rfloor^f)_{\text{aux}}(u : V) &\coloneqq \quad (U, x) \quad \mapsto \quad (uU, f(v, u)U - ux) && (8.63a)
\end{aligned}$$

This is a fold over the pairs $\mathcal{G}(V, Q) \oplus \mathcal{G}(V, Q)$, with the first entry U holding the input so far, and the second entry holding our result x. Equation (8.63a) is obviously linear, and we are obliged to show

$$\begin{aligned}
(v \rfloor^f)_{\text{aux}}(u)^2 &= (U, x) \mapsto (uuU, f(v, u)(uU) - u(f(v, u)U - ux)) && (8.64)\\
&= (U, x) \mapsto (Q(u)U, f(v, u)uU - f(v, u)uU + Q(u)x) && (8.65)\\
&= (U, x) \mapsto (Q(u)U, Q(u)x) && (8.66)\\
&= Q(u). && (8.67)
\end{aligned}$$

All that remains is to initialize $(U, x) = (1, 0)$ in $(v \rfloor^f)_{\text{aux}}$, then discard x' (which holds a copy of x anyway):

$$\begin{aligned}
(v \rfloor^f) &: \ \mathcal{G}(V, Q) \to \mathcal{G}(V, Q) && (8.68)\\
(v \rfloor^f) &\coloneqq \quad x \mapsto c \text{ where } (x', c) = (v \rfloor^f)_{\text{aux}}(x, (1, 0)) && (8.68a)
\end{aligned}$$

With the aid of [70, Theorem 32] we can the show that α^{-f} is a two-sided inverse to α^f, recovering the promised isomorphism.

8.5. Formalization

The approaches in this chapter are particularly amenable to formalization, as avoiding a basis allows them to hold in greater generality. Notably, avoiding a basis ensures our constructions continue to be valid in cases where V is not a free module, and does not have a basis at all. The results in sections 8.3 and 8.3.1 link via "↗" to a formalization in Lean, building on top of the work in [10], which can be found online at `https://github.com/pygae/lean-ga`. They were contributed to mathlib in [*mathlib#14790*].

As an example of what that formalization looks like, and a preview of what is to come later in this thesis, theorem 8.4 is stated there as the `even.lift` in

```
variables {R V A : Type*}
variables [comm_ring R] [add_comm_group V] [module R V]
variables [semiring A] [algebra R A]
variables (Q : quadratic_form R V)

structure even_hom :=
(f : V →ₗ[R] V →ₗ[R] A)
(contract (v : V) : f v v = algebra_map R A (Q v))
(contract_mid (v₁ v₂ v₃ : V) : f v₁ v₂ * f v₂ v₃ = Q v₂ • f v₁ v₃)

def even.lift : even_hom Q A ≃ (clifford_algebra.even Q →ₐ[R] A) :=
sorry
```

where `contract` is theorem 8.4 and `contract_mid` is theorem 8.4. Of note here is that Lean forces us to be very precise about our assumptions on the ring R and algebra A. Indeed, the formalization of the construction in section 8.4 (contributed to mathlib in [*mathlib#11468*]) requires us to additionally write down that our ring is not of characteristic two, as `[invertible (2 : R)]`. This choice is further discussed in section 9.7.

8.6. Summary

Using the universal property for anything beyond trivial constructions like eq. (8.12) can appear anywhere between demanding and impossible. This chapter demonstrates some essential building blocks to bring more demanding constructions within reach. While we only concerned ourselves with the universal properties related to the Clifford algebra, the strategies used apply to many other algebraic constructions with analogous properties.

The approach of building algebraic isomorphisms using universal properties is one that we shall revisit in chapter 10, where the Lean versions are emphasized more prominently. The trick of using endomorphisms to construct complex recursion schemes will appear once more in section 11.1.2.

9

Formalizing Clifford algebras

> It soon became clear that the only real long-term solution to the problems that I encountered is to start using computers in the verification of mathematical reasoning.
>
> *(Vladimir Voevodsky)*

This chapter is adapted from "Formalizing Geometric Algebra in Lean" [10], which was joint work with Utensil Song. The text and formalizations shown here are the author's, though Utensil takes credit for introducing the author to Lean, and for exploring many early dead ends not discussed here. Section 9.7 is entirely new, and was not part of [10].

9.1. Remarks on type theory

It is typical on paper to avoid distinguishing "the multivectors of grade zero" from "the scalars", or "the multivectors of grade 1" from "the vectors"; but the strict dependent typing of Lean forces us to treat these separately. Instead of saying that the multivectors $\mathcal{G}(V, Q)$ "contain" the scalars R and vectors V, we need to provide explicit mappings between these distinct types. Respectively, these come in the form of a ring homomorphism `algebra_map` : $R \to \mathcal{G}(V, Q)$ and a linear map $\iota : V \to \mathcal{G}(V, Q)$[1].

Typically we would assume these mappings are injective, but we can obtain many results without needing to do so. As it turns out, the definitions outlined below permit these maps to be non-injective, for particularly pernicious choices of R and V [71], which we shall explore further in section 9.7.2.

[1] The names of which are taken from the mathlib conventions.

9.2. Existing formalizations of geometric algebra

While there has been no published use of Lean to formalize geometric, Clifford, or Grassman-Cayley algebras, there are already formalizations in other theorem proving languages. To aid the reader, when describing these other formalizations this section presents code snippets translated into their (roughly) equivalent Lean code.

Of particular interest in these existing formalizations is the underlying definition used to define a multivector, and how this definition can be used to define a product of some kind. While proofs are obviously important to a formalization, every proof has to start with a theorem statement, the expressivity of which is limited by the definitions at hand.

They will be presented roughly in order of generality.

9.2.1. Fixed-dimension representations

In [72], a formalization in Isabelle/HOL specific to $\mathcal{G}(\mathbb{R}^3)$ is presented that enumerates all the r-vectors explicitly by assigning a name to each grade:

```
@[ext]
structure multivector :=
(scalar : ℝ) (vector : ℝ × ℝ × ℝ) (bivec : ℝ × ℝ × ℝ) (trivec : ℝ)
```

From here, the operations of an abelian group can be defined componentwise:

```
instance : has_zero multivector := ⟨
{ scalar := 0, vector := 0, bivec := 0, trivec := 0}⟩
instance : has_add multivector := ⟨λ x y,
{ scalar := x.scalar + y.scalar, vector := x.vector + y.vector,
  bivec := x.bivec + y.bivec, trivec := x.trivec + y.trivec}⟩
-- and for `has_sub`, `has_neg`
```

and shown to satisfy its axioms using the `ext` lemma (chapter 5) generate by `@[ext]`, which states "`multivector`s are equal if their four components are equal":

```
instance : add_comm_group multivector :=
{ zero := 0, add := (+), sub := has_sub.sub, neg := has_neg.neg,
  add_zero := λ _, ext _ _ (add_zero _) (add_zero _) (add_zero _) (add_zero _),
  ..sorry /- the 4 other axioms are proved the same way -/ }
```

Already we can see that this approach inevitably leads to a lot of repetition, as while it would be easy to generalize over the integer grades $0, 1, 2, 3$, it's challenging to generalize over the names `scalar`, `vector`, `bivec`, and `trivec`.

The problem only becomes worse when defining the geometric product, as now we have 4 terms in each of the components. As in [72], the full definition is omitted below.

```
instance : has_one multivector := ⟨
{ scalar := 1, vector := 0, bivec := 0, trivec := 0}⟩
instance : has_mul multivector := ⟨λ x y,
{ scalar := x.scalar * y.scalar + dot_product x.vector y.vector
           - dot_product x.bivec y.bivec - x.trivec * y.trivec,
  vector := sorry, bivec := sorry, trivec := sorry}
```

From here, [72] goes on to prove that multivectors form a ring, by showing that multiplication associates and distributes, and operates as expected with one. Again, this would have to be done component-wise, and the trivial but presumably verbose proofs are omitted from [72].

The tedium of the component-wise definitions and proofs can be reduced by generation from CAS implementations and automation features within the theorem provers, but this scales poorly to more complex statements, which may need to be tackled by hand. Needless to say, this formalization does not scale at all to other dimensions and signatures of algebra, as no part of it is generalized.

9.2.2. Recursive tree representations

A convenient way to escape this death-by-cases is to use a recursive definition of a multivector. This is the approach taken by the Coq formalization of Grassmann-Cayley algebra in [73] (and resembles the approach used by the computational Garamon library, [21]). There, the definition is built as a balanced binary tree, where each branch indicates the presence or absence of a basis blade, and the leafs contain the corresponding coefficient.

```
def Gₙ : ℕ → Type
| 0       := ℝ            -- a scalar coefficient
| (n + 1) := Gₙ n × Gₙ n  -- the parts without and with $eₙ$
```

In this Lean translation, the `|` syntax is used to pattern match against an integer representing the remaining depth of the tree, while `×` captures the branching. For instance, $a + a_1e_1 + a_2e_2 + a_{12}e_{12}$ is represented as a term of type `Gₙ 2` as $((a, a_2), (a_1, a_{12}))$.

The recursive data definition leads naturally to a recursive operator definition, which resembles the following[2]:

```
reserve infix ` ∧ `:70
def wedge : Π {n}, Gₙ n → Gₙ n → Gₙ n
| 0       x       y         := (x * y : ℝ)
| (n + 1) ⟨x₁, x₂⟩ ⟨y₁, y₂⟩ := let infix ` ∧ ` := wedge in
      (x₂ ∧ y₂, x₁ ∧ y₂ + x₂ ∧ ¯y₁ᵈ)
infix ` ∧ ` := wedge
```

Here, the pattern-matching is against the depth of the tree, and for non-root elements, the two branches. This elegantly avoids having to deal with coefficient-wise proofs, and results in a definition that works on algebras of arbitrary dimension `n`.

While [73] does not extend to defining a metric or the geometric product it infers, their

[2] The `¯y₁ᵈ` notation is introduced in [73], but not essential to get a feel for how the recursive definitions look.

follow-up work [74] does so. This design still shares a shortcoming of the previous design—it imbues the definition with a canonical and orthogonal basis, which is at odds with our goal of being coordinate-free.

9.2.3. Indexed coordinate representations

The HOL light formalization in [75] offers the same generality as the one in section 9.2.2, but slightly more abstractly describing a multivector as a set of coefficients indexed by the IDs of its basis vectors.

```
variables (n : ℕ)

-- mapping from subsets of 1:n to coefficients
abbreviation idx := set (fin n)
def multivector : Type := idx n → ℝ
```

[75] goes on to define the geometric product and various derived products for arbitrary metrics, which means their formalizations can be used for both CGA and PGA. A rough translation of their generalized product formalization is as follows.

```
def generic_prod (a b : multivector n) (sgn : idx n → idx n → ℝ) : multivector n :=
∑ ai bi : idx n,
  pi.single (ai Δ bi) ((a ai * b bi) * sgn ai bi)
```

Here `ai Δ bi` is the symmetric difference, while `pi.single i x` is the function with $f(i) = x$ and elsewhere $f(\cdot) = 0$. This is then used to derive the wedge and other products as

```
def wedge (a b : multivector n) : multivector n :=
generic_prod a b $ λ ai bi,
  if ai ∩ bi ≠ ∅ then
    0  -- matching blades
  else
    -- compute sign from counting swaps
    (-1) ^ (finset.card $ (ai.product bi).filter $ λ abj, abj.1 > abj.2)
```

Overall, the formalization that follows in [75] is expansive, covering topics ranging from the existence of inverses to outermorphisms. However, the initial definition of `multivector` ingrains a preferred choice of orthogonal (finite) basis, which while in line with many numerical packages for Clifford algebras, is at odds with how vector spaces are formalized in mathlib. mathlib's approach is typically axiomatic, introducing explicit sets of basis vectors only when needed; often, only the proof of the existence of a set of basis vectors is used.

To mesh well with mathlib, our formalization will need to support algebras over a variety of vector spaces, not just those with coefficients in $\mathbb{R}^n$. We are of course free to take on extra assumptions should we need them (such as the scalars forming a field, or the dimension of the space being finite), but by making the initial definitions more general we leave the door open to future researchers interested in other algebras which do not satisfy these assumptions (such as the cases in section 9.7).

9.3. The basics

9.3.1. Construction via quotients

Section 2.2.5 outlines a way to define $\mathcal{G}(V, Q)$ for an R-vector space V and a quadratic form $Q : V \to R$ via constructing a quotient of the tensor algebra $\mathcal{T}(V)$ by the "closure" of the relation $v^2 = Q(v)$.

This is not quite the typical definition; usually [16, §9.1] we take the quotient by the two-sided ideal I_Q generated from the set $\{v \otimes v - Q(v) \,|\, v \in V\}$. Generating an ideal from this set amounts to taking the smallest superset of it that is closed under addition, and under left- and right-multiplication by elements of $\mathcal{T}(V)$. When we take a quotient by an ideal, we are saying that two elements are considered equivalent if their subtraction is in that ideal. It follows then that we have $v \otimes v \approx Q(v)$, and therefore this construction ensures that vectors square to scalars, and is analogous to the approach in section 2.2.5.

As of writing, mathlib does not have direct support for two-sided ideals (though the work in section 4.7.1 treads a path towards them); but it does support the equivalent operation of taking the quotient by a "ring congruence" relation, `ring_con`[3] [*mathlib#17833*], which is a relation that is reflexive, transitive, and respects ring operations. It is this concept of a ring congruence that makes precise the meaning of "closure" in section 2.2.5. In fact, mathlib provides a shorthand for taking the closure of a relation `r` as a ring congruence relation, and then using it to form a quotient `ring_quot r`[4]. As such, the quotient definition still translates fairly naturally into Lean despite the lack of two-sided ideals:

```
variables {R : Type*} [comm_ring R]
variables {V : Type*} [add_comm_group V] [module R V]
variables (Q : quadratic_form R V)

inductive rel : tensor_algebra R V → tensor_algebra R V → Prop
| of (v : V) : rel
    (tensor_algebra.ι R v * tensor_algebra.ι R v)
    (algebra_map R (tensor_algebra R V) (Q v))

/-- `clifford_algebra Q` is the algebra over `V` with metric `Q` -/
@[derive [ring, algebra R]]
def clifford_algebra := ring_quot (clifford_algebra.rel Q)

/-- `ι Q v` is the embedding of the vector `v : V` into the Clifford
algebra with quadratic form `Q`. -/
def clifford_algebra.ι : V →ₗ[R] clifford_algebra Q :=
(ring_quot.mk_alg_hom R _).to_linear_map.comp (tensor_algebra.ι R)
```

The author was involved in discussion about, but did not directly author, the `tensor_algebra`

[3] While not relevant to us here, this `ring_con.quotient` construction is actually more general than quotients by two-sided ideals, as it permits constructing quotients of semirings where we cannot talk about subtraction.

[4] Which in fact predates `ring_con`, originating in [mathlib#4078].

definition from [mathlib#4079] that this builds upon.

What makes this definition particularly attractive is that thanks to the `@[derive ...]` attribute (a Lean meta-program built into mathlib), the `ring` and `algebra` structure can be automatically proved by Lean using its knowledge of `ring_quot`. Note that it is this `ring (clifford_algebra Q)` structure which provides the geometric product `*`!

The operations described in section 9.1 also fall out with minimal effort: the derived `algebra R (clifford_algebra Q)` structure provides the map from the scalars `algebra_map R _ r`; while the map from the vectors `clifford_algebra.ι Q v` is obtained by first mapping the vectors into the tensor algebra (using `tensor_algebra.ι R`, the analogous $\iota : V \to \mathcal{T}(V)$ for the tensor algebra), and then embedding them within the quotient using the API around `ring_quot`. Putting these together, we can verify that our construction does indeed square vectors to scalars,

```
theorem ι_sq_scalar (v : V) : ι Q v * ι Q v = algebra_map R _ (Q v) :=
begin
  erw [←alg_hom.map_mul, ring_quot.mk_alg_hom_rel R (rel.of v), alg_hom.commutes],
  refl,
end
```

and a few other basic algebraic results about vectors,

```
lemma ι_mul_ι_add_swap (a b : V) :
  ι Q a * ι Q b + ι Q b * ι Q a = algebra_map R _ (quadratic_form.polar Q a b) := sorry

lemma ι_mul_comm (a b : V) :
  ι Q a * ι Q b = algebra_map R _ (quadratic_form.polar Q a b) - ι Q b * ι Q a := sorry

lemma ι_mul_ι_mul_ι (a b : V) :
  ι Q a * ι Q b * ι Q a = ι Q (quadratic_form.polar Q a b • a - Q a • b) := sorry
```

where `ι_mul_comm` is eq. (2.14), and `ι_mul_ι_mul_ι` resembles eq. (2.6).

The remaining code to explain is our `rel` definition, which demonstrates how `inductive` types can be used for propositions, as was introduced briefly in section 3.4.1. Here, we declare `rel T₁ T₂` as a proposition over pairs of elements in $T(V)$, but provide only one way to construct it, `rel.of v`. In essence, this means that a proof of `rel T₁ T₂` is a proof that there exists some v such that $T_1 = \iota(v) \otimes \iota(v)$ and $T_2 = Q(v)$. Note that mathlib uses `*` not `⊗` for the product in the tensor algebra.

While not written by the author[5], a better understanding of the power of inductive types can be obtained by looking inside mathlib's definition of `ring_quot`. The inductive type used by this definition to extend our `rel` definition to its closure under ring operations[6] is roughly as follows:

[5] Though the author was involved in its review in [mathlib#4078].

```
inductive crel (rel : R → R → Prop) : R → R → Prop
| of ⦃x y : R⦄ (h : rel x y) : crel x y
| add_left ⦃a b c⦄ : crel a b → crel (a + c) (b + c)
| mul_left ⦃a b c⦄ : crel a b → crel (a * c) (b * c)
| mul_right ⦃a b c⦄ : crel b c → crel (a * b) (a * c)
```

This reads as "A pair of elements are satisfied by the closure of a relation if:"

`of` "They satisfy the relation `rel`"

`add_left` "They can each be split into an addition with the same right operand `c`, and with left operands satisfying the closure of the relation."

`mul_left`, `mul_right` "They can each be split into a multiplication with the same left/right operand, and with a right/left operand satisfying the closure of the relation."

One might remark that something still seems odd about this relation, as neither this `crel` nor our `rel` imply reflexivity, symmetry, or transitivity — however, these properties follow from the axiomatization of Lean's `quot` type, and are provided as part of the Lean prelude as `quot.exact : quot.mk r a = quot.mk r b → eqv_gen r a b`[7] where `eqv_gen r` is the closure of the relation `r` under the three missing properties.

9.3.2. Recovering the universal property

While convenient to define, the quotient can be hard to state further definitions and prove theorems about. When defining operations over a quotient, the approach is almost always to operate on the data within the quotient, and then prove that for any operands that are considered equal under the quotient, the output of the operation is unchanged.

Such proofs can be very challenging, especially given some short-comings in Lean when it comes to recursing over nested inductive types (such as `crel` which wraps our `rel` above). As chapter 8 should hopefully have made clear, we would prefer to work with the universal property from theorem 8.1.

The universal property (theorem 8.1) can be stated in Lean as

```
def lift :
  {f : V →ₗ[R] A // ∀ v, f v * f v = algebra_map _ _ (Q v)}
    ≃ (clifford_algebra Q →ₐ[R] A) := sorry
```

which with the help of table 3.11, reads piecewise as:

`lift : _ ≃ _` "lift" is an equivalence between...

`V →ₗ[R] A` ... R-linear maps from the vector space V to the algebra A...

`{f : _ // ∀ v, f v * f v = algebra_map _ _ (Q v)}` ... whose output squares to the metric ...

[7] The corresponding construction for `ring_con` *does* ensure these three properties.

`clifford_algebra Q →ₐ[R] A` ... and maps between $\mathcal{G}(V, Q)$ and A which are R-linear and preserve multiplication.

The implementation of `lift` (as opposed to just its type) is not included in this thesis, but can be found in [*mathlib#4430*], and consists largely of invoking uninteresting machinery around `ring_quot` already in `mathlib`. This machinery within `mathlib` encapsulates the task of working under the quotient, so that we don't have to.

Note that this is a `def` not a `lemma`; the universal property is a transformation of data, and so its construction is important. While the construction is not shown above, to be convinced that it is truly theorem 8.1, we need only show that $\text{lift}[f](\iota(v)) = f(v)$; or in Lean:

```
theorem lift_ι_apply (f : V →ₗ[R] A) (cond : ∀ v, f v * f v = algebra_map _ _ (Q v)) (v : V) :
  lift Q ⟨f, cond⟩ (ι Q v) = f v :=
```

If we apply this equivalence in reverse to the identity algebra automorphism $\mathcal{G}(V, Q) \to \mathcal{G}(V, Q)$ (that is, evaluate `(lift Q).symm (alg_hom.id R _)`), then we recover a linear map from V to $\mathcal{G}(V, Q)$ whose results square to scalars. Intuitively[8], this is $\iota(v)$.

The type of `lift` alone is enough for us to prove statements like $(\text{lift}\, f)(a+b*c) = f(a)+f(b)f(c)$, while its composition with ι (`lift_ι_apply` above) gives us the remaining interesting properties.

For the rest of our formalization, our proofs and definitions depend only on the properties of `lift` and `ι`, and never on the quotient construction from section 9.3.1. By leveraging the approach in section 8.2.2, this would enable future work to transfer proofs of theorems about our construction to any other construction, such as those in section 9.2, provided that those definitions are equipped with their own `lift` and `ι`.

9.3.3. Conjugations

Grade involution

Beyond the identity mapping `(lift Q) (ι Q) = alg_hom.id R _`, the next simplest operation we can use `lift` to define is the grade involution $\hat{X}$, which flips the sign of every component vector. We already described how to do this using the universal property in eq. (8.12); the Lean translation is

```
def involute : clifford_algebra Q →ₐ[R] clifford_algebra Q :=
clifford_algebra.lift Q ⟨-(ι Q), λ v, by simp⟩
```

where `by simp` is producing the trivial proof that `-(ι Q v)*-(ι Q v) = (ι Q v)*(ι Q v)`. Just as we saw earlier with `lift`, the type of `involute` alone is enough to prove statements about involutions of addition and multiplication. This time, we also get from the type (i.e. that is an `→ₐ[R]`) the fact that `involute` acts as the identity on scalars. All that remains is to prove that involution negates vectors, which get almost for free using some `@[simp]` lemmas about algebra morphisms and `lift_ι_apply` from above:

[8] But thankfully, also proven by Lean!

```
@[simp] lemma involute_ι (v : V) : involute (ι Q v) = -ι Q v :=
by simp [involute]
```

To check that the Lean simplifier is suitably trained for `involute`, we show that taking the involution of a product of n vectors (`(l.map $ ι Q).prod`) is equivalent to scaling by $(-1)^n$:

```
lemma involute_prod_map_ι : ∀ l : list M,
  involute (l.map $ ι Q).prod = ((-1 : R)^l.length) • (l.map $ ι Q).prod
| [] := by simp
| (x :: xs) := by simp [pow_add, involute_prod_map_ι xs]
```

As `involute` is an `→ₐ[R]`, we could also have proved this via the fact it distributes over products, `map_list_prod`; but this turns out to be more annoying.

Grade reversal

A similar approach can be performed to define grade reversion $\tilde{X}$, although this time instead of inserting a minus sign for each vector, we need to flip the multiplication order, following the approach described by eqs. (8.13) and (8.14). As we saw in section 4.7, mathlib provides us a `MulOpposite.op : X → Xᵐᵒᵖ` mapping for exactly that, which by definition satisfies `op x * op y = op (y * x)`. Applying this to each of our vectors will then give `op (reverse x))`, which we can map back into `X` with `MulOpposite.unop : Xᵐᵒᵖ → X`. It can be trivially shown that this mapping is linear and invertible (`opLinearEquiv _ : X ≃ₗ[R] Xᵐᵒᵖ`), which combined with some rather ugly boilerplate gives us a complete definition for grade reversal.

```
def reverseOp : CliffordAlgebra Q →ₐ[R] (CliffordAlgebra Q)ᵐᵒᵖ :=
CliffordAlgebra.lift Q
  ⟨(opLinearEquiv R).toLinearMap ∘ₗ ι Q, fun m => unop_injective <| by simp⟩

@[simp]
theorem reverseOp_ι (m : M) : reverseOp (ι Q m) = op (ι Q m) := lift_ι_apply _ _ _

def reverse : CliffordAlgebra Q →ₗ[R] CliffordAlgebra Q :=
  (opLinearEquiv R).symm.toLinearMap.comp reverseOp.toLinearMap

@[simp]
theorem reverse_ι (m : M) : reverse (ι Q m) = ι Q m := by simp [reverse]

@[simp] theorem op_reverse (x : CliffordAlgebra Q) : op (reverse x) = reverseOp x := rfl
```

While in most cases only `reverse` is useful, in rare cases (section 10.2.6) we will find it convenient to have `reverseOp` (eq. (8.13)) available; the latter is an algebra morphism, `→ₐ[R]`, and so it is sometimes eligible for stronger `ext` lemmas (see chapter 5).

Unlike `involute`, `reverse` is only a `→ₗ[R]` and not a `→ₐ[R]`, so we need to prove how it acts on scalars and multiplication ourselves. Once again, `simp` makes short work of this.

```
@[simp] lemma reverse.map_mul (a b : clifford_algebra Q) :
  reverse (a * b) = reverse b * reverse a :=
by simp [reverse]

@[simp] lemma reverse.commutes (r : R) :
  reverse (algebra_map R (clifford_algebra Q) r) = algebra_map R _ r :=
by simp [reverse]
```

While the above definitions[9] of `involute` and `reverse` give us proofs about their operations on sums and products essentially for free, they miss one key property of these conjugations; the fact that they are involutive, `reverse (reverse x) = x`. One approach we could use to prove this is to set up some point-free equalities of algebra morphisms (as described in section 5.3, and leveraging the extra expressiveness of `reverseOp` over `reverse`); but for the sake of variety and an excuse to expand our toolbox, we shall proceed with a different approach. The approach we choose is the one we might take on paper; claiming "it suffices to consider the effect on pure vectors", which is hiding a more rigorous induction principle.

9.3.4. Induction

The induction principle we seek is theorem 8.3, and rephrased to include slightly more Lean code, can be stated as "If a property `P` holds for the `algebra_map` of `r : R` and the `ι` of `v : V` into `clifford_algebra Q`, and is preserved under addition and multiplication, then it holds for all of `clifford_algebra Q`". As promised in chapter 8, the universal property alone is enough for us to construct this principle.

An outline of the approach is:

1. Show that collectively, the inputs to our inductive principle define a `subalgebra` (`s`) of precisely the elements that satisfy `P`; that is, a subset of the full Clifford algebra which contains zero and one and is closed under addition and multiplication. By doing this, Lean provides us with a type `↥s` which bundles each element of the subalgebra with a proof that it belongs to that subalgebra.

2. Restrict the codomain of `ι Q` to `↥s`, and show that doing so still preserves the fact that vectors square to scalars.

3. Lift this restricted map from `V → ↥s` to `clifford_algebra Q → ↥s`.

4. Apply this lifted map to our input `a`. We show that the value part of this lifting is just `a`, meaning the proof part is `C a`, our proof for an arbitrary element.

The full Lean implementation from [*mathlib#6416*] is shown below.

[9] Which differ from the original versions in [*mathlib#6491*] only by the presence of `reverseOp`.

```
@[elab_as_eliminator]
lemma induction {P : clifford_algebra Q → Prop}
  (h_grade0 : ∀ r, P (algebra_map R (clifford_algebra Q) r))
  (h_grade1 : ∀ x, P (ι Q x))
  (h_mul : ∀ a b, P a → P b → P (a * b))
  (h_add : ∀ a b, P a → P b → P (a + b))
  (a : clifford_algebra Q) :
  P a :=
begin
  -- the arguments are enough to construct a subalgebra, and a mapping into it from M
  let s : subalgebra R (clifford_algebra Q) :=
  { carrier := P, mul_mem' := h_mul, add_mem' := h_add, algebra_map_mem' := h_grade0 },
  let of : clifford_hom Q ↥s :=
  ⟨ (ι Q).cod_restrict s.to_submodule h_grade1,
    λ m, subtype.eq $ ι_sq_scalar Q m ⟩,
  -- the mapping through the subalgebra is the identity
  have of_id : alg_hom.id R (clifford_algebra Q) = s.val.comp (lift Q of),
  { ext,
    simp [of], },
  -- finding a proof is finding an element of the subalgebra
  convert subtype.prop (lift Q of a),
  exact alg_hom.congr_fun of_id a,
end
```

Armed with our new hammer, we find a lot of nail-like lemmas we can easily prove. Below, we show that involution and reverse commute, and each is involutive; that is, $\tilde{\hat{x}} = \hat{\tilde{x}}$, $\hat{\hat{x}} = x$, and $\tilde{\tilde{x}} = x$.

```
@[simp] lemma reverse_involute_commute :
  function.commute (reverse : _ → clifford_algebra Q) involute :=
λ x, by induction x using clifford_algebra.induction; simp [*]

@[simp] lemma involute_involutive :
  function.involutive (involute : _ → clifford_algebra Q) :=
λ x, by induction x using clifford_algebra.induction; simp [*]

@[simp] lemma reverse_involutive :
  function.involutive (reverse : _ → clifford_algebra Q) :=
λ x, by induction x using clifford_algebra.induction; simp [*]
```

9.3.5. The wedge product

The approach described mathematically in section 8.4 leads to the following formal definitions in [*mathlib#11468*]:

```
def changeFormEquiv
    {Q Q' : QuadraticForm R M} {B : BilinForm R M} (h : B.toQuadraticForm = Q' - Q) :
    CliffordAlgebra Q ≃ₗ[R] CliffordAlgebra Q' :=
  sorry

def equivExterior (Q : QuadraticForm R M) [Invertible (2 : R)] :
    CliffordAlgebra Q ≃ₗ[R] ExteriorAlgebra R M :=
  changeFormEquiv (Q := Q) (Q' := 0) (B := QuadraticForm.associated (-Q))) (by simp)
```

where `changeFormEquiv (B := f) h` is the α_f from eq. (8.53), and constructed following the description in that section. The `equivExterior` definition (to_ext in eq. (2.15)) is recovered as a special case, setting one of the two quadratic forms to zero, and choosing `B` as bilinear form associated with `-Q` (which incurs the `Invertible (2 : R)` requirement).

These are supplemented with basic theorems, such as the fact that `equivExterior Q 1 = 1` and `equivExterior Q (ι v) = (ι v)`, along with some more complex statements following [70]; though there are many more important basic theorems in [70] that the author did not attempt to formalize.

As promised in eq. (2.15), this isomorphism provides the wedge product as

```
/-- The wedge product of the Clifford algebra. -/
def wedge [Invertible (2 : R)] (x y : CliffordAlgebra Q) : CliffordAlgebra Q :=
  (equivExterior Q).symm (equivExterior Q x * equivExterior Q y)
```

which is trivially associative as it transports the structure from the exterior algebra. This last definition is currently not in mathlib; without a large surface of API lemmas to make it easy to work with, for now it is better for the end user to write the `equivExterior Q`s explicitly.

9.4. Versors

In [10], the author formalized the versors (from section 2.1.3) as the set of multivectors closed under multiplication (`submonoid`) and scaling, generated from the set of vectors (`set.range (ι Q)`). Formally, this was written

```
def versors := center_submonoid.closure (set.range (ι Q))
```

where `center_submonoid` is a wrapper for a set of elements, built on top of the `submonoid` and `sub_mul_action` structures in mathlib that respectively carry proofs of closure under multiplication and scaling.

Lean provides some useful syntax for working with sub-objects like this. Instead of working with `(x : clifford_algebra Q) (hx : x ∈ versors Q)`, we can write as a shorthand `(v : ↥versors Q)`. The advantage of "bundling" the multivector with its proof of being a versor like this is that operations `v * w` and `k • v` can be defined to automatically produce a bundle containing a proof that their result is also a versor; and indeed, `center_submonoid` does just that.

Armed with our definition, the next step is to once again construct an induction principle. Most of the heavy lifting is done by the `submonoid.closure_induction'` principle in mathlib, which was

developed as part of this paper. The statement of this principle is:

```
/-- If a statement `C` is true for all scalars, all vectors, and all products
of versors which each satisfy `C`, then it is true for all versors. -/
lemma induction_on {C : versors Q → Prop} (v : versors Q)
  (h_scalars : ∀ r : R, C ⟨algebra_map _ _ r, (versors Q).algebra_map_mem r⟩)
  (h_vectors : ∀ m, C ⟨ι Q m, ι_mem m⟩)
  (h_mul : ∀ a b, C a → C b → C (a * b)) :
  C v := sorry
```

The induction principle lets us once again knock out some useful statements with uninteresting proofs. This time, we scratch the surface of Lean's meta-programming framework to avoid repeating a trivial proof:

```
/-- A simple macro tactic that we can reuse between proofs -/
meta def inv_rev_tac : tactic unit :=
`[apply induction_on v,
  { intro r, simp, },
  { intro m, simp, },
  { intros a b ha hb, simp [(versors Q).mul_mem, ha, hb] }]

/-- Involute of a versor is a versor -/
@[simp] lemma involute_mem (v : versors Q) :
  involute (v : clifford_algebra Q) ∈ versors Q := by inv_rev_tac

/-- Reverse of a versor is a versor -/
@[simp] lemma reverse_mem (v : versors Q) :
  reverse (v : clifford_algebra Q) ∈ versors Q := by inv_rev_tac
```

A more interesting application of our induction principle is to prove that the product of a versor and its reverse is a scalar:

```
/-- A versor times its reverse is a scalar -/
lemma mul_self_reverse (v : versors Q) :
  ∃ r : R, (v : clifford_algebra Q) * reverse (v : clifford_algebra Q) = ↑ₐr :=
begin
  with_cases { apply induction_on v },
  case h_grade0 : r {
    refine ⟨r * r, _⟩,
    simp },
  case h_grade1 : m {
    refine ⟨Q m, _⟩,
    simp },
  case h_mul : x y {
    rintros ⟨⟨qx : R⟩, hx⟩ ⟨⟨qy : R⟩, hy⟩, -- results for `x` and `y` by induction
    refine ⟨qx * qy, _⟩,
    simp only [reverse_mul, submonoid.coe_mul, ring_hom.map_mul],
    rw [mul_assoc ↑x, ←mul_assoc ↑y, hy, algebra.commutes, ←mul_assoc, hx], }
end
```

Here we use some more verbose Lean syntax to clearly indicate each branch of the induction. From here, we go on to show that versors have an inverse, and that for a non-trivial algebra[10] over a field in an anisotropic metric, they form a group with zero (i.e. all elements but zero have an inverse):

```
instance
  {K} [field K] {V} [add_comm_group V] [module K V]
  {Q : quadratic_form K V} [nontrivial (clifford_algebra Q)] [f : fact Q.anisotropic] :
    group_with_zero (versors Q) := sorry
```

The work in this section has not made it into mathlib, as the terminology of "versor" is rather specific to geometric algebra. In the context of Clifford algebras, a more typical object of study would be the slightly different "Lipschitz group", which is being developed by other mathlib contributors (with the author's help) in [mathlib#16040; mathlib4#9111].

9.5. Grade selection

As a reminder from chapter 6, an algebra A is said to be graded by an additive monoid I if there exists a family of I-indexed submodules A_i such that $x \in A_i,\ y \in A_j \to xy \in A_{i+j}$, $1 \in A_0$, and every element $a \in A$ has a unique decomposition[11] $a = \sum_i a_i$ where $a_i \in A_i$. Note that this section of chapter 9 diverges from [10]; the original predated the work in chapter 6, and used an inferior solution in terms of `add_monoid_algebra` (a type that was mentioned briefly in chapter 4).

9.5.1. $\mathbb{N}$-grading

The grading described the "grade selection" of eq. (2.5) is that of the exterior algebra, as discussed in section 2.2.6. The formalization of this is almost exactly the same as the one alluded to in example 6.5, but as no explanation was provided there, it shall be provided here instead.

The statement of the result (this time in Lean 4) is

```
abbrev exteriorPower (n : ℕ) : Submodule R (ExteriorAlgebra R M) :=
  LinearMap.range (ι R (M := M)) ^ n

instance gradedAlgebra : GradedAlgebra (exteriorPower R M) :=
```

which amounts to constructing the algebra isomorphism `decomposeAlgequiv : ExteriorAlgebra ≃ₐ[R] ⨁ i, exteriorPower M i` where the reverse map is the obvious one. Constructing the forward map is straightforward; we use theorem 8.1, with a suitable map `ι' : M →ₗ[R] ⨁ i, ↥(exteriorPower R M i)` that is just a trivial repackaging of `ι R : M →ₗ[R] ExteriorAlgebra R M`. Proving that this is a right-inverse to the forward map—that taking an element of $\bigwedge(M)$ apart and putting it back together again is the identity—is another easy consequence via `ext` of theorem 8.2.

[10] As of [mathlib4#7985], `Nontrivial (CliffordAlgebra Q)` can be deduced from the assumptions `[Nontrivial R] [Invertible (2 : R)]`.

[11] Which corresponds to requiring that the submodules A_i span the space, and they satisfy some notion of disjointness.

Proving that it is a *left*-inverse—that taking apart an element of `exteriorPower R M i` results in a piece of a single grade, is rather harder, and requires an induction principle to recurse on the `i` in `exteriorPower R M i`. The statement of this induction principle is

```
@[elab_as_elim]
protected theorem pow_induction_on_left'
    -- For a submodule `M`, a predicate for each power of it, and proofs that:
    (M : Submodule R A) {motive : ∀ n : ℕ, ∀ x : A, x ∈ M ^ n → Prop}
    -- 1) Within the `0`th power, the predicate is satisfies by the center of the algebra
    (hr : ∀ r : R,
      motive 0 (algebraMap _ _ r) (algebraMap_mem r))
    -- 2) Within a the `i`th power, the predicate is closed under addition
    (hadd : ∀ x y : A, ∀ i : ℕ, ∀ hx : x ∈ M ^ i, ∀ hy : y ∈ M ^ n,
      motive i x hx → motive i y hy → motive i (x + y) (add_mem ‹_› ‹_›))
    -- 3) Within the `i+1`th power, the predicate is true for `m * x`,
    --    where `m ∈ M` and `x` is within the `i`th power, if it is true for `x`
    (hmul : ∀ m : A, ∀ hm : m ∈ M, ∀ i : ℕ, ∀ hx : x ∈ M ^ i,
      motive i x hx → motive i.succ (m * x) (mul_mem_mul hm hx)) :
    -- Then for any power `n`, the predicate is true on its members.
    ∀ n : ℕ, ∀ hx : x ∈ M ^ n, motive n x hx := sorry
```

which we instantiate with `A := ExteriorAlgebra R M` and `M := LinearMap.range (ι R (M := M))`. The proof of `pow_induction_on_left'` follows from a similar statement about binary products, rather than powers, of submodules. With this induction principle doing the heavy lifting, the proof in [*mathlib#11542*] is unremarkable.

There is something slightly surprising about this induction principle; the `motive` of the induction is not just quantifying over elements of the underlying type `x : A` and the index of the power `n : ℕ`, but also over *proofs* that `x ∈ M ^ n`, meaning that the statement being proven is itself allowed to be a function of a proof! This turns out to be vital in the face of Lean's dependent types, as our use of the subtype `↥(exteriorPower R M i)` (indicated by the `↥`) means that the statement of our goal really does contain proofs, for example the `hx` on the RHS in:

```
i: ℕ
x: ExteriorAlgebra R M
hx: x ∈ LinearMap.range (ι R) ^ i
⊢ liftι R M x = DirectSum.of (fun i => ↥(exteriorPower R M i)) i { val := x, property := hx }
```

This pattern of dependent induction was largely non-existent in mathlib until the author's contributions in [*mathlib#4984*; *mathlib#11533*; *mathlib#11556*; *mathlib#14219*]; though the trick was worked out with the help of the community in [76].

Together, these pieces conclude the construction of the `gradedAlgebra` instance at the top of section 9.5.1. Selecting the `i : ℕ`th grade of an element `x : CliffordAlgebra Q` then follows as

```
abbrev gradeSelect (x : CliffordAlgebra Q) (i : ℕ) : exteriorPower R M i :=
  decompose (equivExterior Q x) i
```

9.5.2. $\mathbb{Z}_2$-grading

If we take the definition of A graded algebra in its literal sense, then the product in question is not the wedge product, but the geometric product. It may seem like this is problematic, as if we take the product of two vectors $\iota(v)\iota(w)$, then the result is a scalar; even though the axioms of a graded ring suggest that we should obtain an element of degree two. The solution is to pick a different additive monoid to number our grades by; namely $\mathbb{Z}_2$, where $0 = 2$. This recovers the standard split of a Clifford algebra into its even and odd parts.

As a recap from example 6.6, we define these parts in Lean as

```
/-- The even or odd submodule, defined as the supremum of the even or odd powers of
`(ι Q).range`. `evenOdd 0` is the even submodule, and `evenOdd 1` is the odd submodule. -/
def evenOdd (i : ZMod 2) : Submodule R (CliffordAlgebra Q) :=
  ⨆ j : { n : ℕ // ↑n = i }, LinearMap.range (ι Q) ^ (j : ℕ)
```

where `j : { n : ℕ // ↑n = i }` is a sneaky way of saying "`j` is a natural number with parity `i`".

The formal proof that these form a graded algebra, the statement of which is

```
instance gradedAlgebra : GradedAlgebra (evenOdd Q)
```

follows the same approach in section 9.5.1; but now not only do we need `pow_induction_on_left'` to induct on powers of the submodule, but a similar `iSup_induction'` to induct on elements of `⨆ j`, the indexed supremum. These same induction principles also lead to the formal proof of theorem 8.5.

9.6. Constructing specific algebras

Section 2.1.4 draws attention to some particularly useful geometric algebras over the real numbers, which until this point this formalization has been too general to mention. This section will demonstrate how to specialize the general formalization to these specific cases.

In particular, it will show how to set up a definition of Conformal Geometric Algebra[12], $\mathcal{G}(\mathbb{R}^{n+1,1,0})$, which augments the initial vector space $\mathbb{R}^n$ with two extra dimensions spanned by the orthogonal vectors e_+ and e_- where $e_+^2 = 1$ and $e_-^2 = -1$. In practice it can be algebraically convenient to span these extra dimensions instead with null basis vectors n_o, n_∞ where $n_o \cdot n_\infty = 1$, and $n_o^2 = n_\infty^2 = 0$, as is done in [1, Table 13.1]. This approach is desirable because these basis vectors have more geometric meaning, with n_o being associated with the origin, and n_∞ associated with the point at infinity.

We start by defining the conformalized vector space of a real vector space as the triple of (original vector space V, n_0 coefficient, n_∞ coefficient). Note that by doing this we *are* choosing a preferred basis (something we generally wanted to avoid) over the extra dimensions, but we continue to avoid doing so over V.

[12] A construction of PGA in Lean can be done in very much the same way, and is included in the `lean-ga` repository. It is omitted here to avoid repetition.

```
@[derive [add_comm_group, vector_space ℝ]]
def conformalize (V : Type*) [inner_product_space ℝ V] : Type* := V × ℝ × ℝ
```

We proceed by providing linear maps to extract each component:

```
def v : conformalize V →ₗ[ℝ] V := linear_map.fst _ _ _
def c_n0 : conformalize V →ₗ[ℝ] ℝ :=
  (linear_map.fst _ _ _).comp (linear_map.snd _ _ _)
def c_ni : conformalize V →ₗ[ℝ] ℝ :=
  (linear_map.snd _ _ _).comp (linear_map.snd _ _ _)
```

and some definitions to construct conformal vectors:

```
/-- The embedding of direction vectors into `conformalize V`. -/
def of_v : V →ₗ[ℝ] conformalize V := linear_map.inl _ _ _
/-- The n₀ basis vector. -/
def n0 : conformalize V := (0, 1, 0)
/-- The n∞ basis vector. -/
def ni : conformalize V := (0, 0, 1)
```

Finally, we can define the *up* mapping $\text{up}(v) = n_0 + v + \frac{1}{2}\|v\|^2 n_\infty$ and the conformal metric Q, the final pieces needed to construct the Clifford algebra:

```
/-- The embedding of points from `V` to `conformalize V`. -/
def up (x : V) : conformalize V :=
n0 + of_v x + (1 / 2 * ∥x∥^2 : ℝ) • ni

/-- The metric is the metric of `V`' plus an extra term about `n0` and `ni`. -/
def Q : quadratic_form ℝ (conformalize V) :=
(bilin_form_of_real_inner.comp v v).to_quadratic_form
  - (2 : ℝ) • quadratic_form.lin_mul_lin c_n0 c_ni

variables (V)
/-- Define the Conformal Geometric Algebra over `V` . -/
abbreviation CGA := clifford_algebra (Q : quadratic_form ℝ (conformalize V))
```

With our definitions out of the way, our next job is to train the Lean simplifier about trivial combinations of these functions

```
@[simp] lemma v_of_v (x : V) : (of_v x).v = x := rfl
@[simp] lemma c_n0_of_v (x : V) : (of_v x).c_n0 = 0 := rfl
@[simp] lemma c_ni_of_v (x : V) : (of_v x).c_ni = 0 := rfl
-- 6 more lemmas follow combining {v, n0, ni} with {of_v, of_n0, of_ni}
```

From this, we can prove that `Q` has the form we'd expect, that `up` correctly produces null vectors and that the metric between two conformal points is proportional to their distance:

```
@[simp] lemma Q_apply (x : conformalize V) : Q x = ‖x.v‖^2 - 2 * (x.c_n0 * x.c_ni) :=
by simp [Q, inner_self_eq_norm_sq_to_K]

@[simp] lemma Q_up (x : V) : Q (up x) = 0 :=
by simp [up, ←mul_assoc]

lemma Q_polar_up (x y : V) : quadratic_form.polar Q (up x) (up y) = -dist x y ^ 2 :=
sorry
```

Note that while at a glance this last result appears off by a factor of two from the "usual" result that $\mathrm{up}(x) \cdot \mathrm{up}(y) = -\frac{1}{2}\|x - y\|$, this is because Lean's `quadratic_form.polar Q x y` is defined as twice the value of the inner product $x \cdot y$.

These are of course only the most basic of results about CGA, and serve only to demonstrate that the very general definition of `clifford_algebra` does not rule out concrete specializations. To actually become a useful tool for formalizing CGA, many additional `def`s and `lemma`s would be needed that connect equations in `CGA` to result stated using `mathlib`'s geometry library. One such result would be that there is an embedding from `euclidean_geometry.sphere` into `CGA` such that $\mathrm{up}\, x \wedge \mathrm{of\_sphere}\, S = 0 \iff x \in S$.

9.7. Pathological cases

While certainly the most typical description, as mentioned in section 2.2.3 the choice to define Clifford algebras over a quadratic form Q is not universally agreed upon, and a bilinear form B is sometimes used instead (such as in [15]). Through the lens of associative algebras, this definition makes no difference; $\mathcal{G}(V, B)$ is isomorphic as an algebra to $\mathcal{G}(V, x \mapsto B(x, x))$, and indeed the Lean formalization in this thesis makes it trivial to make this true by definition. To do so, we would define `quantum_clifford_algebra B` as a type synonym (section 4.6) of `clifford_algebra B.to_quadratic_form`, copying across the `ring` and `algebra` instances.

Where this B has an effect is in constructing the "canonical" linear isomorphism $\mathcal{G}(V, B) \cong \bigwedge(V)$; for the $\mathcal{G}(V, Q)$ case in eq. (8.62), to solve $B(v, v) = Q(V)$, we chose B (there f) as the (symmetric) bilinear form associated with Q. If we drop the symmetry requirement on B, then we are free to add any antisymmetric bilinear form onto our previous choice of B, as described extensively in [15]. By dropping this requirement, B cannot be determined solely from Q, and so we must carry it around with us in our type.

Assuming we are happy to keep the symmetry requirement, the definition of this associated bilinear form (described in section 2.2.3) includes a factor of $\frac{1}{2} : R$; it is for this reason that our formalization is littered with `Invertible (2 : R)` assumptions. Strictly speaking, this is an unnecessarily strong requirement; the quadratic form

$$Q : \mathbb{Z}^2 \to \mathbb{Z} \coloneqq (x, y) \mapsto x^2 + y^2 \tag{9.1}$$

has an obvious symmetric bilinear form associated with it,

$$B : \mathbb{Z}^2 \to \mathbb{Z}^2 \to \mathbb{Z} \coloneqq (x_1, y_1) \mapsto (x_2, y_2) \mapsto x_1 x_2 + y_1 y_2, \tag{9.2}$$

even though 2 is not invertible in $\mathbb{Z}$. A future refactor could be to replace `Invertible (2 : R)` with a new typeclass along the lines of

```
class HasCanonicalSymmBilinForm (Q : QuadraticForm R M) where
  B : BilinForm R M
  B_symm : B.IsSymm
  B_unique (B' : BilinForm R M) (hB' : B.IsSymm) : B'.toQuadraticForm = Q ↔ B' = B
```

which carries a bilinear form B and a proof that it is the *unique* symmetric bilinear form associated with the quadratic form. This would enable many results in this thesis to generalize to the situations like eq. (9.1).

While more general than `Invertible (2 : R)`, there are still some situations where `HasCanonicalSymmBilinForm Q` is unsatisfiable. We shall look at two examples, both in the cases where R is of characteristic two and so $\frac{1}{2} : R$ is nonsensical due to the fact that $2 = 0$.

9.7.1. Non-unique associated forms

In many cases in characteristic two, symmetric solutions are still available to $B(v, v) = Q(V)$, but they are no longer unique; and so there is no longer an obviously canonical choice of $\mathcal{G}(V, Q) \cong \bigwedge(V)$ available. For instance, over the $\mathbb{F}$-vector space $\mathbb{F}_2 \oplus \mathbb{F}_2$, there are at least two ways to pick B when $Q \coloneqq 0$: the nontrivial solution is $B((x_1, y_1), (x_2, y_2)) \coloneqq x_1 y_2 + x_2 y_1$, which is even symmetric. We can thus formalize that a unique choice of a symmetric B is not possible (in [*mathlib#18146*]) as:

```
/-- `BilinForm.toQuadraticForm` is not injective on symmetric bilinear forms. -/
theorem BilinForm.not_injOn_toQuadraticForm_isSymm.{u} :
    ¬∀ {R M : Type u} [CommRing R] [AddCommGroup M] [Module R M],
      Set.InjOn (toQuadraticForm : BilinForm R M → QuadraticForm R M) {B | B.IsSymm} := by
```

The loss of a canonical $\mathcal{G}(V, Q) \cong \bigwedge(V)$ in characteristic two is another argument in favor of the $\mathcal{G}(V, B)$ spelling, as this preserves canonicity by virtue of encoding the canonical choice of isomorphism in the type itself. However, there are greater perils awaiting us in characteristic two.

9.7.2. Non-existent associated forms, and injectivity of $R \to \mathcal{G}(V, Q)$

If our R-module V is free, then we can at least guarantee that a *non-symmetric* associated bilinear form exists, by choosing an ordered basis v_i on V and writing

$$B(\sum_i a_i v_i, \sum_i b_i v_i) = \sum_i a_i b_i Q(v_i) + \sum_{i<j} a_i b_j \operatorname{polar}[Q](v_i, v_j) \tag{9.3}$$

from which we can show $B(v, v) = Q(v)$ since $Q(\sum_i a_i v_i) = \sum_i a_i^2 Q(v_i) + \sum_{i<j} a_i a_j \operatorname{polar}[Q](v_i, v_j)$, and thus [*mathlib4#14292*]:

```
theorem toQuadraticForm_surjective [Module.Free R M] :
  Function.Surjective (BilinForm.toQuadraticForm : BilinForm R M → QuadraticForm R M)
```

However, in some cases there is no associated bilinear form at all. To reach this conclusion, we shall first investigate whether `Injective (algebraMap R (CliffordAlgebra Q))` holds; that is, whether the scalars inside the algebra are "the same as" the ones in `R`. To a reader used to working only with Clifford algebras where $R \coloneqq \mathbb{R}$, or one used to writing a scalar r without any explicit "casting" between the base ring and a Clifford algebra (discussed in section 9.1), this may seem like a silly question to which the answer is that it surely holds (as respectively, in those cases it does hold, and the choice of notation bakes in the assumption that it holds).

As a warm-up, we can reassure ourselves that this result at least holds for the exterior algebra [*mathlib#5722*] (i.e. when $Q = 0$). To do so, we construct an explicit inverse by invoking $\operatorname{lift}_\wedge[0]$ (the universal property for the exterior algebra), which by virtue of being an algebra morphism is an inverse to `algebraMap` by construction.

For the Clifford algebra it turns out that there are cases where injectivity does not hold! The counterexample [71] provides to demonstrate this is

$$R \coloneqq \mathbb{F}_2[\alpha, \beta, \gamma]/(\alpha^2, \beta^2, \gamma^2), \quad V \coloneqq R^3/(\alpha e_1 + \beta e_2 + \gamma e_3), \quad Q \coloneqq \overline{v} \mapsto v_1^2 + v_2^2 + v_3^2. \tag{9.4}$$

The punchline is that while $\alpha\beta\gamma \neq 0$ in R, we have $\alpha\beta\gamma = 0$ in $\mathcal{G}(V, Q)$; somehow, building the Clifford algebra has shrunk our ring of scalars, and therefore the `algebraMap` we use to formalize the usual "inclusion" is not an inclusion at all.

It is not too hard to formalize the construction of R and V in eq. (9.4); mathlib has a well-developed library of quotients and polynomials, allowing us to write:

```
def rIdeal :
    Ideal (MvPolynomial (Fin 3) (ZMod 2)) :=
  .span { X i * X i | i }

/-- `𝔽₂[α, β, γ] / (α², β², γ²)` -/
def R : Type _ :=
  MvPolynomial (Fin 3) (ZMod 2) ⧸ rIdeal
```

```
def vSubmodule : Submodule R (Fin 3 → R) :=
  letI f : (Fin 3 → R) →ₗ[R] R :=
    α • .proj 0 + β • .proj 1 + γ • .proj 2
  LinearMap.ker f

/-- `R^3 / {αx + βy + γz}` -/
def V : Type _ :=
  (Fin 3 → R) ⧸ vSubmodule
```

Defining Q is a harder, as after defining the quadratic form on R^3, we need to prove the map is well-defined under the quotient (the first `sorry`):

```
/-- The quadratic form (metric) is just euclidean -/
def Q' : QuadraticForm R (Fin 3 → R) :=
  ∑ i, QuadraticForm.sq.comp (.proj i)

/-- 'Q'', lifted to operate on the quotient space 'V'. -/
@[simps!]
def Q : QuadraticForm K V :=
  .ofPolar (fun x => Quotient.liftOn' x Q' sorry) sorry sorry sorry
```

The other three `sorry`s are trivial.

The crux of this first `sorry` (and in the author's opinion, the entire proof in [71]) is showing that when $\alpha v_1 + \beta v_2 + \gamma v_3 = 0$, we have $v_1^2 + v_2^2 + v_3^2 = 0$. This follows by left-multiplying the assumption by each of $\alpha\beta$, $\beta\gamma$, and $\gamma\alpha$ in turn, eliminating two of the three terms due to $\alpha^2 = \beta^2 = \gamma^2 = 0$, and then using an auxiliary result that $\alpha\beta\gamma x = 0 \implies x^2 = 0$. This auxiliary result is the kind of thing that is easy to hand-wave over ("x must contain only α, β, and γ terms"), but the devil ends up being in the details, making it very fiddly to formalize.

The route taken by the author required a substantial amount of infrastructure for dividing (multivariate) polynomials by monomials in [*mathlib#15905*], which provided the definitions

```
def divMonomial (p : MvPolynomial σ R) (s : σ →₀ ℕ) : MvPolynomial σ R := sorry
def modMonomial (x : MvPolynomial σ R) (s : σ →₀ ℕ) : MvPolynomial σ R := sorry
```

and 20 or so obvious theorems about how they interact with each other and `monomial`. In fact, these were first defined for the more general case of `AddMonoidAlgebra`, matching how the scalar actions discussed in section 4.3.4 are constructed. With these operations in place, [*mathlib#18633*] was able to prove the following theorem about ideals generated by monomials:

```
/-- 'x' is in a monomial ideal generated by 's' iff every element of its support dominates one of
the generators. -/
theorem mem_ideal_span_monomial_image {x : MvPolynomial σ R} {s : Set (σ →₀ ℕ)} :
    x ∈ Ideal.span ((fun s => monomial s (1 : R)) '' s) ↔ ∀ xi ∈ x.support, ∃ si ∈ s, si ≤ xi :=
  sorry
```

from which we recover as a corollary a special case.

```
theorem mem_ideal_span_X_image {x : MvPolynomial σ R} {s : Set σ} :
    x ∈ Ideal.span (MvPolynomial.X '' s : Set (MvPolynomial σ R)) ↔
      ∀ m ∈ x.support, ∃ i ∈ s, (m : σ →₀ ℕ) i ≠ 0 := sorry
```

This says "x is in the ideal generated by the variables in s if and only if each of its monomials contains a variable in s"; a seemingly trivial statement mathematically!

The rest of the formalization [*mathlib4#6657*] is largely arithmetic manipulation, so will not be reproduced here.

There are two interesting corollaries that follow from `Injective (algebraMap R (CliffordAlgebra Q))` not holding. The first is

Corollary 9.1. *There cannot be* any *linear isomorphism $\mathcal{G}(V, Q) \cong \bigwedge(V)$ that preserves the scalars*

which was the actual result being shown in [71]; we cannot have `algebraMap` preserve scalars at one end of the isomorphism and not the other! The second is the one we set out to prove in this section,

Corollary 9.2. *Not every quadratic form can be recovered from a bilinear form B with $B(v, v) = Q(v)$,*

which follows in [*mathlib4#9670*] by noting that if such a bilinear form existed, the construction in section 8.4 would contradict corollary 9.1.

9.8. Summary

This chapter formalized a variety of elementary GA operations, and showed how to leverage various existing definitions supplied in mathlib to do so in a way that is concise and integrates with the rest of the library. Notably, it provides the `clifford_algebra Q` (or in Lean 4, `CliffordAlgebra Q`) definition that will be used through the remaining formalizations in this thesis. Unlike the formalizations that exist in other theorem provers, the resulting definition in mathlib is very general, and works even over non-free modules.

Section 9.7.2 contained a formalization of particular interest; one that showed both the extreme generality in which the definition holds, but also that integration into mathlib gives us a broad selection of tools (multivariate-polynomials, quotients, results about non-zero-characteristics) that would not be available had this formalization been standalone.

A complete archive consisting of the formalizations, the translations of parts of existing formalizations, the precise versions of Lean and mathlib that the code is compatible with, and a summary of the various contributions to mathlib made as part of this chapter can be found on GitHub at

`https://github.com/pygae/lean-ga`

Primarily, this chapter has been about setting up useful foundations for further formalization; it leaves plenty on the table in terms of basic results still to be formalized, especially on the interaction between grade selection and the geometric product, such as eq. (2.5).

10

Isomorphisms

> I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.
>
> *(Abraham Maslow)*

A frequently-cited selling point of Clifford algebras for engineers and physicists is that they capture the structure of various other algebraic systems in a single unified language [5, §2; 77, §1; 14]. Informally, we might say something like "the quaternions $\mathbb{H}$ *are* a Clifford algebra"; under the formalization described in this thesis, we interpret that to mean "the quaternions $\mathbb{H}$ *are isomorphic*[1] *as a real-algebra to* a Clifford algebra". In this section, we shall demonstrate how to use Lean to construct these isomorphisms.

There is standard strategy that we will use for every isomorphism $A \cong_R B$ here, which is as follows:

- Use the appropriate universal properties to construct forward ($F : A \to_R B$) and reverse ($F^{-1} : B \to_R A$) algebra morphisms.
- Use the appropriate extensionality lemma (chapter 5) to show that these morphisms compose in either order to give the identity ($F^{-1} \circ F = \mathrm{id}_A$, $F \circ F^{-1} = \mathrm{id}_B$), concluding that together they form an isomorphism.

Section 10.1 works through this strategy on $\mathbb{R}$, $\mathbb{C}$, $\mathbb{H}$, along with the dual numbers and dual quaternions; formalizing universal properties in the process. Section 10.2 addresses isomorphisms relating to Clifford algebras over $\mathbb{C}$, building significant infrastructure around tensor products in the process. Finally, section 10.3 links this strategy with chapter 6 via a formalization of the "graded tensor product".

[1] Or in [1, §1.2.4], "a quaternion [...] is already an element of geometric algebra", suggesting an inclusion rather than an isomorphism.

10.1. Well-known isomorphisms

10.1.1. Reals

We shall start with the easiest isomorphism,

Theorem 10.1. *The real numbers are isomorphic as an $\mathbb{R}$-algebra to $\mathcal{G}(\mathbb{R}^0)$,*

which we will immediately generalize to

Theorem 10.2. *Any commutative ring R is isomorphic as an R-algebra to $\mathcal{G}(R^0)$.*

No universal property is needed for the forward direction; we just use the canonical algebra morphism that any R-algebra is equipped with, $R \to_R \mathcal{G}(R^0)$ (mathlib's `algebraMap`), which we will write as $F \coloneqq r \mapsto r$. For the reverse direction we will need theorem 8.1, which we invoke with the trivial linear map $0 : R^0 \to_R R$; there are no non-zero vectors in $\mathcal{G}(R^0)$, so this is the only possible choice! We will write this as $F^{-1} \coloneqq \text{lift}_{\mathcal{G}}[0]$.

Our first proof obligation is $F^{-1} \circ F = \text{lift}_{\mathcal{G}}[0] \circ (x \mapsto x) = \text{id}_A$. There is only one R-algebra morphism from R and mathlib knows this, so we can prove this simply with `subsingleton.elim _ _`. Our second obligation is $F \circ F^{-1} = (x \mapsto x) \circ \text{lift}_{\mathcal{G}}[0] = \text{id}_A$; applying theorem 8.2 turns this into the easier equality of linear maps $\text{lift}_{\mathcal{G}}[0] \circ \iota = \iota$. This can then also be proven with `subsingleton.elim _ _`, as all linear maps from the zero module are zero!

The final step is performed by mathlib's `alg_equiv.of_alg_hom` function, which assembles all these pieces into a full construction:

```
/-- The Clifford algebra over a 0-dimensional vector space is isomorphic to its scalars. -/
protected def equiv : clifford_algebra (0 : quadratic_form R unit) ≃ₐ[R] R :=
alg_equiv.of_alg_hom
  (clifford_algebra.lift (0 : quadratic_form R unit) $
    ⟨0, λ m : unit, (zero_mul (0 : R)).trans (algebra_map R _).map_zero.symm⟩)
  (algebra.of_id R _)
  (subsingleton.elim _ _)
  (by { ext : 1, exact subsingleton.elim _ _ })
```

We conclude by showing that that reversion (section 9.3.3) and grade involution (section 9.3.3) are the identity operations on $\mathcal{G}(R^0)$. The former follows through the induction principle in section 9.3.4, while the latter can be proven with `by ext; simp` thanks to theorem 8.2.

10.1.2. Complex numbers

> This section is extracted from §7.9 of "Formalizing Geometric Algebra in Lean" [10], instead of appearing with the main part of that work in chapter 9.

The isomorphism for the complex numbers is only very slightly more interesting:

Theorem 10.3. *The complex numbers are isomorphic as an $\mathbb{R}$-algebra to $\mathcal{G}(\mathbb{R}^{0,1})$.*

Intuitively this is still simple; the real axis corresponds to the scalars, while the imaginary axis

corresponds to the single dimension of $\mathbb{R}^{0,1}$. To apply the strategy outlined at the start of chapter 10, we will need a universal property and extensionality lemma for the complex numbers. A suitable pair, from [*mathlib#8105*; *mathlib#8107*], is

Theorem 10.4 (Extensionality for algebra morphisms from the complex numbers)**.** *Given an $\mathbb{R}$-algebra A, and two algebra morphisms $F, G : \mathbb{C} \to_{\mathbb{R}} A$, to show $F = G$ it suffices to show $F(\mathrm{i}) = G(\mathrm{i})$.*

Theorem 10.5 (The universal property of the complex numbers)**.** *Given an $\mathbb{R}$-algebra A, there is a one-to-one correspondence between:*

elements $a : A$ such that $a^2 = -1$	*and*	*algebra morphisms $F : \mathbb{C} \to_{\mathbb{R}} A$.*

The construction of theorem 10.5 is straightforward; we choose $a = F(\mathrm{i})$, and $F(z) = \Re(z) + \Im(z)a$.

To construct the isomorphism in theorem 10.3, we thus pick our forward map as $F := \mathrm{lift}_{\mathbb{C}}[\iota(1)]$, and our reverse map as $F^{-1} := \mathrm{lift}_{\mathcal{G}}[a \mapsto a\mathrm{i}]$. Extensionality leaves us to prove $F^{-1}(F(\mathrm{i})) = F^{-1}(\iota(1)) = 1\mathrm{i} = \mathrm{i}$ and $F(F^{-1}(\iota(1))) = F(1\mathrm{i}) = \iota(1)$, where we see $\iota(1)$ instead of $\iota(v)$ due to the extensionality lemma in theorem 5.2 that is already in mathlib.

The resulting formal construction of theorem 10.3 in [*mathlib#8165*] is stated as follows

```
/-- The quadratic form sending elements to the negation of their square. -/
def Q : quadratic_form ℝ ℝ := -quadratic_form.lin_mul_lin linear_map.id linear_map.id

@[simp] lemma complex.Q_apply (r : ℝ) : complex.Q r = -(r*r) := rfl

/-- The Clifford algebra over `clifford_algebra_complex.Q` is isomorphic as an `ℝ`-algebra to `ℂ`. -/
protected def equiv : clifford_algebra Q ≃ₐ[ℝ] ℂ := sorry

lemma equiv_ι : equiv (ι complex.Q 1) = complex.I := sorry
```

where the two `lemma`s demonstrate that the definitions are the ones we expect.

One advantage of the approach of treating informal "is" as indicating an isomorphism, as discussed at the beginning of chapter 10, is that it does not force us to declare the isomorphism canonical. Indeed, in the example above, we (or another downstream user that wants a different meaning of "is") could just as easily mapped `ι complex.Q 1` to `-complex.I` instead of `complex.I`.

We conclude by showing (as part of [*mathlib#8739*]) that reversion is once again the identity function, but involution corresponds to complex conjugation. We do this in a point-free style (by stating the correspondence in terms of morphism compositions commuting) so that we can once again let extensionality do all the work. For the Lean version of this involution result, the transformation to this point-free style is requested by the `suffices` line in

```
/-- `clifford_algebra.involute` is analogous to `complex.conj`. -/
@[simp] lemma to_complex_involute (c : clifford_algebra Q) :
  to_complex (c.involute) = conj (to_complex c) :=
begin
  suffices : to_complex.comp involute = complex.conj_ae.to_alg_hom.comp to_complex,
  { exact alg_hom.congr_fun this c },
  ext : 2,
  show to_complex (involute (ι Q 1)) = conj (to_complex (ι Q 1)),
  simp only [involute_ι, to_complex_ι, alg_hom.map_neg, one_smul, complex.conj_I],
end
```

and the `ext : 2` applies theorem 10.4 then theorem 5.2, reducing the goal to the statement that appears after `show`.

10.1.3. Dual numbers

The dual numbers are the real numbers adjoined with a symbol ϵ such that $\epsilon^2 = 0$, and can be written in the form $r + d\epsilon$ for $r, d : \mathbb{R}$. We shall write them as $\mathbb{R}[\epsilon]$ to remind ourselves that our coefficients are real, but they are sometimes written $\mathbb{D}$. These were added to mathlib in [*mathlib#10730*] (building upon a more general construction from [mathlib#5109] that we shall discuss in section 10.1.5). A typical motivation for their use in software is that they enable forward automatic differentiation [78, §3]; if a computation $f : \mathbb{R} \to \mathbb{R}$ over the real numbers is suitably extended to execute as $g : \mathbb{R}[\epsilon] \to \mathbb{R}[\epsilon]$ over the dual numbers, then $g(x + \epsilon) = f(x) + \frac{df}{dx}\epsilon$, allowing the derivative to be extracted without any symbolic manipulation. This can be valuable when running optimization algorithms[2].

The connection to Clifford algebras is that

Theorem 10.6. *The dual numbers $\mathbb{R}[\epsilon]$ are isomorphic as an $\mathbb{R}$-algebra to $\mathcal{G}(\mathbb{R}^{0,0,1})$.*

which can be constructed in almost exactly the same way as the construction for theorem 10.3 was in section 10.1.2, by replacing -1 with 0 and i with ϵ in theorems 10.4 and 10.5. The universal property this produces, formalized in [*mathlib#10754*], is

Theorem 10.7 (The universal property of the dual numbers)**.** *Given an $\mathbb{R}$-algebra A, there is a one-to-one correspondence between:*

elements $a : A$ such that $a^2 = 0$	*and*	*algebra morphisms $F : \mathbb{R}[\epsilon] \to_{\mathbb{R}} A$.*

The formal statement of theorem 10.6 in [*mathlib#10730*] is

```
/-- The Clifford algebra over a 1-dimensional vector space with 0 quadratic form is isomorphic to
the dual numbers. -/
protected def equiv {R : Type*} [comm_ring R] :
  clifford_algebra (0 : quadratic_form R R) ≃ₐ[R] R[ε] :=
```

In fact, it was no extra work to define this for the dual numbers `R[ε]` with coefficients an arbitrary

[2] Though in practice, reverse automatic differentiation is often more useful, which does not use $\mathbb{R}[\epsilon]$.

ring R, rather than just over the real numbers.

10.1.4. Quaternions

The quaternions $\mathbb{H}$ are the four-dimensional real division algebra with basis $\{1, i, j, k\}$, such that

$$i^2 = j^2 = k^2 = -1, \quad ij = -ji = k, \quad jk = -kj = i, \quad ki = -ik = j, \quad ijk = -1. \tag{10.1}$$

They are used extensively in computer graphics as a representation of rotations in $\mathbb{R}^3$. Expressed more formally, the non-zero quaternions $\mathbb{H}^\times$ form a double cover of the Lie group $SO(\mathbb{R}^3)$ under the conjugation action[3] $v \mapsto qvq^{-1}$.

When the quaternions are referred to as "included" in geometric algebra, that usually is intended to describe the isomorphism $\mathbb{H} \cong_{\mathbb{R}} \mathcal{G}^+(\mathbb{R}^3)$, where i, j, k correspond to the bivectors e_{23}, e_{31}, e_{12} respectively. That is not the isomorphism we will construct in this section, though it can be recovered via composition with the result in section 8.3.1. Instead, we will show

Theorem 10.8. *The quaternions $\mathbb{H}$ are isomorphic as an $\mathbb{R}$-algebra to $\mathcal{G}(\mathbb{R}^{0,2})$.*

which places i, j, and k in correspondence with the multivectors e_1, e_2, and e_{12} (or more formally, $\iota((1,0))$, $\iota((0,1))$, and the product of these two).

Once again, the strategy at the start of chapter 10 requires we find a universal property and extensionality lemma, this time for the quaternions. These are formalized in [*mathlib#8551*; *mathlib4#9441*], and have statements

Theorem 10.9 (Extensionality for algebra morphisms from the quaternions)**.** *Given an $\mathbb{R}$-algebra A, and two algebra morphisms $F, G : \mathbb{H} \to_{\mathbb{R}} A$, to show $F = G$ it suffices to show $F(\mathrm{i}) = G(\mathrm{i})$, $F(\mathrm{j}) = G(\mathrm{j})$.*

Theorem 10.10 (The universal property of the quaternions)**.** *Given an $\mathbb{R}$-algebra A, there is a one-to-one correspondence between:*

pairs of elements $a_i, a_j : A$		*algebra morphisms*
such that $a_i^2 = a_j^2 = -1$ and $a_i a_j = -a_j a_i$	*and*	*$F : \mathbb{H} \to_{\mathbb{R}} A$.*

We shall write this as $F = \mathrm{lift}_{\mathbb{H}}[a_i, a_j]$.

Note that neither of theorems 10.9 and 10.10 mention k, as respectively: $F(k) = G(k)$ follows from the results on i and j, and taking $a_k = a_i a_j$ is sufficient. The construction of theorem 10.10 should be of no surprise, as it follow the pattern set by theorem 10.5; we can construct $a_i = F(\mathrm{i})$, $a_j = F(\mathrm{j})$, and $F(q) = \Re(q) + \Im_i(q)a_i + \Im_j(q)a_i + \Im_k(q)a_i a_j$.

With these in place, we pick our forward map as $F \coloneqq \mathrm{lift}_{\mathbb{H}}[\iota((1,0)), \iota((0,1))]$, and our reverse map as $F^{-1} \coloneqq \mathrm{lift}_{\mathcal{G}}[(x, y) \mapsto x\mathrm{i} + y\mathrm{j}]$, where the side-conditions of each lift are trivial. When proving these are inverses, extensionality leaves us to show $F^{-1}(F(\mathrm{i})) = F^{-1}(\iota((1,0))) = 1\mathrm{i}+0 = \mathrm{i}$ and similarly for j; and for the other direction, $F(F^{-1}(\iota((1,0)))) = F(\mathrm{i}) = \iota((1,0))$ and similarly

[3] A result that is available as a Lean formalization exercise in [79, §2.4].

for $\iota((0,1))$. Once again, theorem 5.2 is responsible for helping us here, this time in combination with the extensionality lemma for product types (binary direct sums) in section 5.2, by reducing the obligation to show $F(F^{-1}(\iota((x,y)))) = \iota((x,y))$ to the special cases where one of x, y is 1 and the other is 0.

As it turns out, quaternions in mathlib are defined more generally as quaternion *algebras* over an arbitrary commutative ring, replacing eq. (10.1) with

$$i^2 = c_1, \quad j^2 = c_2, \quad k^2 = -c_1c_2, \quad ij = -ji = k, \quad jk = -kj = -c_2 i, \quad ki = -ik = -c_1 j \tag{10.2}$$

for constants $c_1, c_2 : R$. The formalized version of theorem 10.10 in [*mathlib#8670*] is stated in this generality, as it adds no meaningful additional complexity.

We conclude by showing in [*mathlib#8739*] that quaternion conjugation, the map sending $r + xi + yj + zk \mapsto r - xi - yj - zk$, is transformed by the isomorphism of theorem 10.8 to the Clifford conjugate (the composition of grade involution and reversion), which we write as

```
/-- The "Clifford conjugate" maps to the quaternion conjugate. -/
lemma to_quaternion_star (c : clifford_algebra (Q c₁ c₂)) :
  to_quaternion (star c) = star (to_quaternion c) :=
```

The proof follows using our `clifford_algebra.induction` principle from theorem 8.3. This result is also present in the HOL light formalization associated with [75], but [75] never shows theorem 10.8, and instead defines the forward map as a plain function without showing it is a morphism of any kind.

10.1.5. Dual Quaternion

The dual quaternions combine the i, j, k from the quaternions with the ϵ from the dual numbers, to give an 8-dimensional ring. They too have frequent applications in computer graphics, as they contain a subgroup that is a double-cover of the special Euclidean group SE(3), making them the tool of choice for computational representations combining position and orientation.

They can be thought of as either the dual numbers with quaternion coefficients, the quaternions with dual number coefficients, or the tensor product of the quaternions and the dual numbers. The first two of these viewpoints require us to generalize our definitions of $\mathbb{H}$ and $\mathbb{R}[\epsilon]$ in sections 10.1.3 and 10.1.4 to be over coefficients other than $\mathbb{R}$, which thankfully mathlib had already prepared us for. In the case of quaternions with dual number coefficients, the mathlib definition is already sufficient; `ℍ[R]` works for any commutative ring `R`, so we can write `ℍ[R[ε]]` to obtain the quaternions with coefficients in $R[\epsilon]$, and get the appropriate ring structure automatically. In the case of dual numbers with quaternion coefficients, mathlib needed some modification; `R[ε]` required that `R` be a *commutative* ring, which the quaternions `ℍ[R]` are not. Despite this obstruction, we will choose to use this latter "dual numbers with quaternion coefficients" representation, as not only does this put the words in the same order as "dual quaternions" (as `DualNumber (Quaternion R)`), but it also maps more closely to the practice of writing dual quaternions as $p + q\epsilon$.

The standard isomorphism we are after is

Theorem 10.11. *The dual quaternions $\mathbb{H}[\epsilon]$ are isomorphic as an $\mathbb{R}$-algebra to $\mathcal{G}(\mathbb{R}^{0,2,1})$.*

where taking (e_i, e_j, e_0) as the ordered basis for $\mathbb{R}^{0,2,1}$, we identify i with e_i, j with e_j, k with $e_i e_j$, and ϵ with $e_i e_j e_0$. Before we can state this in Lean, we will look at how mathlib's `dual_number` had to be adjusted.

Trivial square-zero extensions over non-commutative rings

Generalizing the definition of `dual_number` in mathlib to non-commutative rings requires us to understand the existing definition:

```
/-- The type of dual numbers, numbers of the form $a + bε$ where $ε^2 = 0$.-/
abbreviation dual_number (R : Type*) : Type* := tsze R R
```

As is typical in mathlib, the definition of the thing we care about is recovered as special case of a more general construction. The construction in question here is the "trivial square-zero extension" (from [mathlib#5109]); namely, the direct sum $R \oplus M$ for a ring R and an R-module M, imbued with a multiplication that identifies the product of any pair of elements in M with zero. We shall write this as $\text{tsze}[R, M]$. This construction is in fact even more useful for automatic differentiation than the dual numbers, as it can be used to differentiate functions of multiple variables[4], for instance by choosing $M = R^2$ (with basis elements ϵ_x and ϵ_y) and lifting a function $f : R \to R \to R$ to $g : R[\epsilon_x, \epsilon_y] \to R[\epsilon_x, \epsilon_y] \to R[\epsilon_x, \epsilon_y]$ such that $g(x + \epsilon_x, y + \epsilon_y) = f(x, y) + \frac{d}{dx} f(x, y)\epsilon_x + \frac{d}{dy} f(x, y)\epsilon_y$.

The version of the trivial square-zero extension in mathlib prior to [*mathlib#18384*] stated this multiplication as

$$(r_1 + m_1)(r_2 + m_2) = r_1 r_2 + r_1 m_2 + r_2 m_1. \tag{10.3}$$

This is correct for when R is *commutative*, but when it is *non-commutative* the last term $r_2 m_1$ is unnatural, and prevents the operation from being associative. This can be resolved by using $m_1 r_2$ instead of $r_2 m_1$ as the final term. The corrected version of this multiplication, as written in Lean, is characterized by

```
@[simp] lemma fst_mul [has_mul R] [has_add M] [has_smul R M] [has_smul Rᵐᵒᵖ M] (x₁ x₂ : tsze R M) :
  (x₁ * x₂).fst = x₁.fst * x₂.fst := rfl
@[simp] lemma snd_mul [has_mul R] [has_add M] [has_smul R M] [has_smul Rᵐᵒᵖ M] (x₁ x₂ : tsze R M) :
  (x₁ * x₂).snd = x₁.fst • x₂.snd + op x₂.fst • x₁.snd := rfl
```

where `fst` is the projection onto the R part of the direct sum, and `snd` is the projection onto the M part. Of interest here are the typeclass arguments `[has_smul R M]` `[has_smul Rᵐᵒᵖ M]`, which using the approach laid out in section 4.7, say that "R acts in different ways on the left and right of M". As a reminder from section 4.7, `op r • m` is our notation for scalar multiplication on the right, mr.

[4] For which [*mathlib4#9510*] is relevant.

With this corrected definition of multiplication, we can now obtain the associative ring structure we desire, stated as

```
instance [ring R] [add_comm_group M] [module R M] [module Rᵐᵒᵖ M] [smul_comm_class R Rᵐᵒᵖ M] :
  ring (tsze R M) :=
```

While the name may suggest otherwise, as discussed in section 4.7.1, `[smul_comm_class R Rᵐᵒᵖ M]` captures that our scalar multiplication is associative in the sense of an R-R-bimodule: in Lean it provides the commutativity statement `r₁ • (op r₂ • m) = op r₂ • (r₁ • m)`, but mathematically it represents $r_1(mr_2) = (r_1 m)r_2$.

With this in place (and some less interesting generalizations in [*mathlib#10729*]), we can state that "the dual numbers with quaternion coefficients are isomorphic to the quaternions with dual number coefficients", as:

```
def dual_number_equiv : ℍ[R[ε]] ≃ₐ[R] (ℍ[R])[ε] := sorry
```

where the actual implementation replacing the `sorry` in [*mathlib#18383*] is a simple unpacking and repacking of coefficients.

Having convinced Lean that `(ℍ[R])[ε]` is a ring and an `R`-algebra, and that the choice between `(ℍ[R])[ε]` and `ℍ[R[ε]]` is a stylistic one of no algebraic consequence, we can write the statement of theorem 10.11 as `CliffordAlgebra Q ≃ₐ[R] ℍ[R][ε]`, for a suitable definition of `Q`.

Universal properties

Proceeding along the usual path for constructing isomorphisms from universal properties, we find that the universal property for the dual numbers in theorem 10.7 is inadequate; it can only provide us an R-algebra morphism from $R[\epsilon]$ for commutative R, not an R-algebra morphism from $A[\epsilon]$ where A is a non-commutative algebra over R. We say the situation is "heterobasic", as there are two different base rings at play here. The universal property that we need here, from [*mathlib4#7934*], is

Theorem 10.12 (The heterobasic universal property of the dual numbers)**.** *Given R-algebras A and B over a commutative ring R, there is a one-to-one correspondence between:*

$$\begin{array}{c|c|c} \text{algebra morphisms } f : A \to_R B \text{ paired with} \\ \text{a choice of element } b : B \text{ such that} \\ b^2 = 0 \text{ and } \forall a,\ f(a)b = bf(a) \end{array} \quad \text{and} \quad \text{algebra morphisms } F : A[\epsilon] \to_R B.$$

We shall write this as $F = \operatorname{lift}_\epsilon[f, b]$.

The construction is such that $F(x + y\epsilon) = f(x) + f(y)b$. From this universal property, we get the extensionality lemma in theorem 10.13.

Theorem 10.13 (Extensionality for heterobasic algebra morphisms from the dual numbers)**.** *Given R-algebras A and B over a commutative ring R, and two algebra morphisms $F, G : A[\epsilon] \to_{\mathbb{R}} B$, to show $F = G$ it suffices to show $F \circ (a : A \mapsto a) = G \circ (a : A \mapsto a)$ and $F \circ (a \mapsto a\epsilon) = G \circ (a \mapsto a\epsilon)$, where the equalities are between algebra morphisms and linear maps from A to B.*

Theorem 10.12 can be stated more generally for the trivial square-zero extension, as

Theorem 10.14 (The heterobasic universal property of the trivial square-zero extension $\text{tsze}[A, M]$)**.** *Given R-algebras A and B over a commutative ring R and an A-A-bimodule M, there is a one-to-one correspondence between:*

algebra morphisms $f : A \to_R B$ paired with a linear map $g : M \to_R B$ such that
$\forall m, g(m)^2 = 0$
$\forall a, \forall m,\ g(am) = f(a)g(m)$
$\forall m, \forall a,\ g(ma) = g(m)f(a)$

and

algebra morphisms $F : \text{tsze}[A, M] \to_R B$.

The construction for theorem 10.14 is such that $F(a + m) = f(a) + g(m)$. We can state theorem 10.13 for the square-zero extension in a similar way (and indeed the formalization does). In the formalization, we derive the forward direction of theorem 10.12 from the forward direction of theorem 10.14 by setting $M = A$ and choosing the $g : A \to_R B$ in theorem 10.14 to be $a \mapsto ab$.

Constructing the isomorphism

The forward direction of our equivalence in theorem 10.11 is the easy direction; we choose $F \coloneqq \text{lift}_{\mathcal{G}}[(a_i, a_j, a_0) \mapsto a_i\mathrm{i} + a_j\mathrm{j} - a_0\mathrm{k}\epsilon]$, where some simple algebra shows that the argument to $\text{lift}_{\mathcal{G}}$ squares to the quadratic form.

It is the reverse direction that justified the work in section 10.1.5; we choose

$$F^{-1} : \mathbb{H}[\epsilon] \to_{\mathbb{R}} \mathbb{G}(\mathbb{R}^{0,2,1}) \coloneqq \text{lift}_{\epsilon}[F_{\mathbb{H}}^{-1}, x_{\epsilon}] \tag{10.4}$$

$$\text{where} \qquad F_{\mathbb{H}}^{-1} : \mathbb{H} \to_{\mathbb{R}} \mathbb{G}(\mathbb{R}^{0,2,1}) \coloneqq \text{lift}_{\mathbb{H}}[\iota(e_i), \iota(e_j)] \tag{10.5}$$

$$x_{\epsilon} : \mathbb{G}(\mathbb{R}^{0,2,1}) \coloneqq \iota(e_i)\iota(e_j)\iota(e_0). \tag{10.6}$$

From our use of theorem 10.12's lift_{ϵ}, we are left with a proof obligation to show that $x_{\epsilon}^2 = 0$ (straightforward), and that x_{ϵ} and $F_{\mathbb{H}}^{-1}(y)$ commute for all $y : \mathbb{H}$ (follows by showing that x_{ϵ} commutes with $\iota(e_i)$ and $\iota(e_j)$). Our use of theorem 10.10's $\text{lift}_{[H]}$ comes with some other straightforward algebraic proof obligations.

Showing that F and F^{-1} are inverses follows again via the `ext` tactic, though unfortunately this still leaves some tedious work to be handled in [*mathlib4#7962*]. It is worth remembering how universal properties have benefitted us here; while we could have taken the naïve approach of constructing the isomorphism by writing down a mapping of the 8 basis vectors, to prove that the map preserves multiplication we would have ended up juggling 64 terms and 16 different

coefficients.

As with section 10.1.4, the result formalized in mathlib by the author is stated more generally for the dual numbers of the quaternion algebra constructed using eq. (10.2), and brings little additional complexity beyond an extra $c_1^{-1}c_2^{-1}$ term in the definition of x_ϵ.

10.2. Complexification

Given a Clifford algebra over a real-vector space such as $\mathcal{G}(\mathbb{R}^3)$, it is natural to ask for a canonical embedding into the Clifford algebra over the corresponding complex-vector space, $\mathcal{G}(\mathbb{C}^3)$. This destination algebra can be thought of as the "complexified" version of $\mathcal{G}(\mathbb{R}^3)$. Over a vector space with a canonical basis such as $\mathbb{R}^3$, the embedding is obvious; we simply embed the coefficients of each basis blade. For instance, the multivector $a + be_{12}$ becomes $(a + 0i) + (b + 0i)e_{12}$, which we might prefer to write as $(a + be_{12}) + 0i$.

While mathematically these viewpoints are easy to dismiss as equivalent, type theory does not give us that luxury for free. $(a + 0i) + (b + 0i)e_{12}$ can be thought of as an element of $\mathcal{G}(\text{complexify}(\mathbb{R}^3))$, where the underlying vector space has been complexified; while $(a + be_{12}) + 0i$ is an element of $\text{complexify}(\mathcal{G}(\mathbb{R}^3))$, where the Clifford algebra has been complexified[5]. We could evade the task of proving their equivalence by taking $\text{complexify}(\mathcal{G}(\mathbb{R}^3)) \coloneqq \mathcal{G}(\text{complexify}(\mathbb{R}^3))$ by definition, but that precludes a more abstract definition of complexify that generalizes.

This more abstract definition is $\text{complexify}(V) = \mathbb{C} \otimes_{\mathbb{R}} V$, namely the real-tensor product of the complex numbers $\mathbb{C}$ and our vector space V. It is straightforward to show that $\text{complexify}(\mathbb{R}^3)$ and $\mathbb{C}^3$ are isomorphic as $\mathbb{C}$-vector spaces, but already some sleight of hand has been performed; we require them to be isomorphic as *quadratic* vector spaces, which requires us first to declare a canonical quadratic form $Q_{\mathbb{C}} : \text{complexify}(V) \to^2 \mathbb{C}$ given a form $Q : V \to^2 \mathbb{R}$.

With this in mind, a formalization target begins to precipitate: constructing the canonical algebra isomorphism between $\mathbb{C} \otimes_{\mathbb{R}} \mathcal{G}(V, Q))$ and $\mathcal{G}(\mathbb{C} \otimes_{\mathbb{R}} V, Q_{\mathbb{C}}))$. Before we embark on this goal, we add one final generalization; instead of working over $\mathbb{R}$ and $\mathbb{C}$, we will work over an arbitrary commutative ring R (where two has an inverse[6]) and commutative R-algebra A. This generalization replaces the concept of "complexifying" with that of "base changing"; we say that $A \otimes_R V$ is the base-change of V from R to A. Our target is thus:

[5] This is analogous to the two spellings of dual quaternions discussed in section 10.1.5.

[6] Which we use as a simplified proxy for "Q has an associated bilinear form", as discussed in section 9.7.

Theorem 10.15 (Base change commutes with the Clifford functor)**.** *Given a commutative ring R, an R-vector space V, a quadratic form $Q : V \to^2 R$, a commutative R-algebra A, and a quadratic form $Q_A : A \otimes_R V \to^2 A$ that satisfies $Q_A(a \otimes v) = a^2 Q(v)$, there is a canonical isomorphism of algebras*

$$A \otimes_R \mathcal{G}(V, Q) \cong_A \mathcal{G}(A \otimes_R V, Q_A) \tag{10.7}$$

that identifies $a \otimes \iota_Q(v)$ with $\iota_{Q_A}(a \otimes v)$ (where the ι_Q is associated with $\mathcal{G}(V, Q)$, and the ι_{Q_A} with $\mathcal{G}(A \otimes_R V, Q_A)$).

10.2.1. Base change of quadratic forms

Conspicuously absent in theorem 10.15 is any mention of how to obtain a suitable Q_A. While "extend Q [...] linearly to a quadratic form Q_A defined by $Q_A(a \otimes v) = a^2 Q(v)$" may be excused in a lecture series [80, §2.2], it is hiding a lot under the rug. In particular, the approach of defining a map on a pure tensor and extending linearly only works for defining a "linear" map from the tensor product, which can be expressed as yet another universal property:

Theorem 10.16 (The universal property of the tensor product)**.** *For any commutative ring R and R-modules M, N, P, there is a one-to-one correspondence between:*

R-linear maps $M \otimes_R N \to_R P$ *and* *R-bilinear maps $M \to_R N \to_R P$.*

The snag is that our Q is not a linear map but a quadratic one, so theorem 10.16 is not applicable to it.

We can make a little progress here by showing the following [*mathlib4#14285*]:

Theorem 10.17 (Extensionality on base-changed quadratic forms)**.** *For a commutative ring R, a commutative R-algebra A, and an R-module M, to show two quadratic forms $Q_1, Q_2 : A \otimes_R M \to A$ are equal, it suffices to show they agree on elements of the form $1 \otimes m$ for $m : M$.*

This reassures us that any such extension is at least unique, but doesn't help with constructing one.

Some doors open for us if we reach for an ordered basis on V. With this, we can appeal to a definition of the tensor product as a quotient of sums of pure 2-tensors under a quotient by a suitable relation (which is precisely the definition used by mathlib), and show that the definition $Q_A(\overline{\sum_i a_i \otimes v_i}) = \sum a_i^2 Q(v_i) + \sum_{i<j} a_i a_j \operatorname{polar}[Q](v_i, v_j)$ is invariant to any choice of representation a_i, v_i. The "suitable relation" here is rather tricky to work with formally, to the point that it is never used again in mathlib once the universal property is available. Alternatively, we can appeal to a representation of quadratic forms as homogeneous multivariate polynomials of degree two, and perform the base change there, as suggested in [81].

Instead, we will proceed in a third direction; by recovering the base change of quadratic forms as a special case of a more general operation, the tensor product of quadratic forms. While it is not immediately clear that this will make things any easier, this more general result is of interest

in other areas of mathematics[7], and so is itself a valuable formalization target.

10.2.2. Tensor products of quadratic forms

We will consider building the tensor product of two quadratic forms $Q_\otimes(u \otimes_R v) = Q_U(u)Q_V(v)$ where $u : U$ and $v : V$. To recover the Q_A of section 10.2.1 satisfying $Q_A(a \otimes v) = a^2 Q(v)$ as a special case, we can set $U = A$ and $Q_U(a) = a^2$. We need to be careful though, as we need Q_U and $Q_\otimes$ to be A-quadratic forms, while Q_V is only R-quadratic; once again, the situation is "heterobasic" as it was in section 10.1.5. Stated precisely then, we have:

Theorem 10.18 (The quadratic form on the tensor product)**.** *For a commutative ring R in which 2 is invertible, a commutative R-algebra A, a quadratic A-module (U, Q_U) and a quadratic R-module (V, Q_V), there is a quadratic A-module $(U \otimes_R V, Q_\otimes)$ such that $Q_\otimes(u \otimes_R v) = Q_U(u)Q_V(v)$.*

A natural extension is show an identification between $Q_\otimes$ and $Q_U \otimes_R Q_V$, which we should expect to be A-linear[8]:

Theorem 10.19 (Tensor products distribute over quadratic forms)**.** *For a commutative ring R in which 2 is invertible, a commutative R-algebra A, an A-module U, and an R-module V, there is an A-linear map of type*

$$\text{Qdistrib} : ((U \to^2 A) \otimes_R (V \to^2 R)) \to_A ((V \otimes_R Q) \to_2 A),$$

such that for Q_U an A-quadratic form on U and Q_V an R-quadratic form on V,

$$\text{Qdistrib}(Q_U \otimes_R Q_V)(u \otimes_R v) = Q_U(u)Q_V(v).$$

We can then prove theorem 10.18 using theorem 10.19 by defining $Q_\otimes = \text{Qdistrib}(Q_U \otimes_R Q_V)$.

When it comes to proving theorem 10.19, we are still obstructed by our inability to apply theorem 10.16. In fact, we now have two obstacles rather than one: we still can't use theorem 10.16 to construct a quadratic form; but as theorem 10.16 requires the two rings to be the same we also can't apply it to Qdistrib as this is an A-linear map from an R-tensor product. We will postpone this new second problem to section 10.2.4.

To solve the first problem, we shall shift from considering quadratic forms to considering bilinear forms (thus incurring our requirement of `invertible (2 : R)`, as discussed in section 9.7[9]).

[7] Notably for the "Witt ring of quadratic forms", for which participants of the "Lean for the Curious Mathematician 2023" workshop attempted a formalization only a few weeks after the author contributed the tensor product of quadratic forms to mathlib.

[8] We will see in more detail what it means for a map from $\otimes_R$ to be A-linear in section 10.2.4.

[9] Strictly speaking, by using eq. (9.3) instead of the associated bilinear form, we can exhange `Invertible (2 : R)` for `Module.Free R M`. However, while this produces a unique tensor product for the special case of a base change thanks to theorem 10.17, it is not clear that the result is invariant to the choice of ordered basis for general tensor products. For consistency with the approaches used elsewhere in this thesis, we shall stick to `Invertible (2 : R)`.

10.2.3. Tensor products of bilinear forms

Having moved our goalposts once again, we are now faced with:

Theorem 10.20 (Tensor products distribute over bilinear forms)**.** *For a commutative ring R and commutative R-algebra A, an A-module U and an R-module V, there is an A-linear map of type*

$$\text{Bdistrib} : ((U \to_A U \to_A A) \otimes_R (V \to_R V \to_R R)) \to_A ((U \otimes_R V) \to_A (U \otimes_R V) \to_A A),$$

such that for B_U an A-bilinear form on U and B_V an R-bilinear form on V,

$$\text{Bdistrib}(B_U \otimes_R B_V)(u_1 \otimes_R v_1, u_2 \otimes_R v_2) = B_U(u)B_V(v).$$

from which we can prove theorem 10.19 by taking $B_U = \text{associated}(Q_U)$, $B_V = \text{associated}(Q_V)$, and extending linearly[10] from $\text{Qdistrib}(Q_U \otimes_R Q_V)(x) = \text{Bdistrib}(B_U \otimes_R B_V)(x, x)$.

We will construct this Bdistrib by assembling a chain of linear maps ($\to_A$) and equivalences ($\cong_A$), instead of attempting to use a single application of theorem 10.16. This is partially motivated by the expectation that using the universal property directly will create lots of tedious work, but also the hope that the component pieces either already exist in mathlib, or would be valuable independent contributions. Our composition proceeds as follows:

$$\begin{aligned}
&(U \to_A U \to_A A) \otimes_R (V \to_R V \to_R R) \\
\cong_A\ &((U \otimes_A U) \to_A A) \otimes_R ((V \otimes_R V) \to_R R) && (10.8)\\
\cong_A\ &\text{Dual}_A (U \otimes_A U) \otimes_R \text{Dual}_R (V \otimes_R V) && (10.9)\\
\to_A\ &\text{Dual}_A ((U \otimes_A U) \otimes_R (V \otimes_R V)) && (10.10)\\
\cong_A\ &\text{Dual}_A ((U \otimes_R V) \otimes_A (U \otimes_R V)) && (10.11)\\
\cong_A\ &(U \otimes_R V) \otimes_A (U \otimes_R V) \to_A A && (10.12)\\
\cong_A\ &(U \otimes_R V) \to_A (U \otimes_R V) \to_A A && (10.13)
\end{aligned}$$

There is a fair amount to unpack here. Equations (10.8) and (10.13) are applications of the (heterobasic[11]) universal property of the tensor product. Equations (10.9) and (10.12) are simply matching against the definition of the dual vector space of a module M, $\text{Dual}_R(M) := M \to_R R$. Equation (10.11) is a statement of four-way commutativity of tensor products, which we shall look at more closely in section 10.2.4. Finally, eq. (10.10) is the statement that the dual vector space of a tensor product is injected into by the tensor product of the dual vector spaces. Prior

[10] Or to avoid that now-suspect phrase, by pre- and post- composing our already-linear Bdistrib with suitable linear maps to convert between bilinear and quadratic forms, as

$$\text{Qdistrib} = (B \mapsto x \mapsto B(x, x)) \circ \text{Bdistrib} \circ (\text{associated} \otimes_R \text{associated}).$$

Note that this $\otimes_R$ is the (heterobasic!) tensor product of *linear maps* defined such that $(f \otimes_R g)(x \otimes_R y) = f(x) \otimes_R g(y)$.

[11] A variant of theorem 10.16 that involves a second compatible ring S.

to the author's contributions, mathlib included all these equivalences, but only for the special case with $R = A$.

In [*mathlib4#6306*], eqs. (10.8) to (10.13) are written as follows

```
def tensorDistrib : BilinForm A M₁ ⊗[R] BilinForm R M₂ →ₗ[A] BilinForm A (M₁ ⊗[R] M₂) :=
  ((TensorProduct.AlgebraTensorModule.tensorTensorTensorComm R A M₁ M₂ M₁ M₂).dualMap
    ≪≫ₗ (TensorProduct.lift.equiv A (M₁ ⊗[R] M₂) (M₁ ⊗[R] M₂) A).symm
    ≪≫ₗ LinearMap.toBilin).toLinearMap
  ∘ₗ TensorProduct.AlgebraTensorModule.dualDistrib R _ _ _
  ∘ₗ (TensorProduct.AlgebraTensorModule.congr
    (BilinForm.toLin ≪≫ₗ TensorProduct.lift.equiv A _ _ _)
    (BilinForm.toLin ≪≫ₗ TensorProduct.lift.equiv R _ _ _)).toLinearMap
```

Note that eq. (10.10) (`TensorProduct.AlgebraTensorModule.dualDistrib`) is the only part of this composition that is not an equivalence. In fact, this line *is* an equivalence when the modules U and V are finite and free, so we can in turn promote theorem 10.20 to an equivalence in this case (and indeed [*mathlib4#6306*] does).

10.2.4. Algebraic towers in tensor products

In eq. (10.11), we need the isomorphism that is a special case[12] of `tensorTensorTensorComm`,

$$(M \otimes_R N) \otimes_A (P \otimes_R Q) \cong_A (M \otimes_A P) \otimes_R (N \otimes_R Q), \tag{10.14}$$

characterized by its action on pure tensors

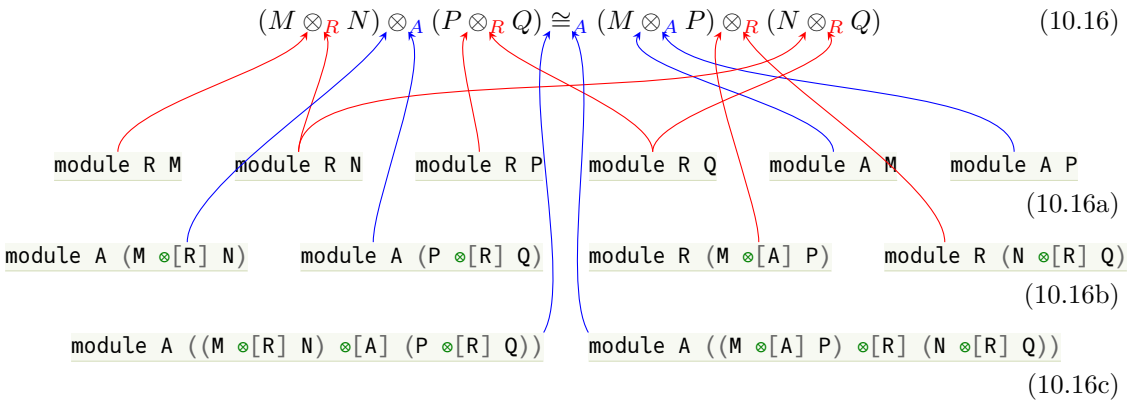
$$(m \otimes n) \otimes (p \otimes q) \longleftrightarrow (m \otimes p) \otimes (n \otimes q), \tag{10.15}$$

for A an R-algebra, and M, N, P, Q modules over suitable rings. We shall see that when formalizing this result, we will make heavy use of some on typeclasses first introduced in section 4.3.5.

Stating four-way commutativity in Lean

For the formal *statement*[13] of eq. (10.14) to make sense, we need all of the following twelve module structures over our two rings (written as Lean `module` instances), for the left- and right-hand side of each of the subscripted operators and relations.

[12] With $M \coloneqq P \coloneqq U$, $N \coloneqq Q \coloneqq V$.
[13] And analogously, the statements of theorems 10.19 and 10.20.



In practice, we would like to assume only the module structures in eq. (10.16a), and set things up such that these structures imply the more complex structures needed by eqs. (10.16b) and (10.16c). The building block we need for this is the following heterobasic instance,

```
instance tensor_product.left_module {R S M N : Type*}
  [comm_semiring R] [semiring S] [add_comm_monoid M] [add_comm_monoid N]
  [module R M] [module R N] [module S M] [smul_comm_class R S M] :
  module S (M ⊗[R] N) := sorry
```

defined such that `s • (m ⊗ₜ n) = (s • m) ⊗ₜ n`. This was generalized in [*mathlib#5430*] from

```
instance tensor_product.module {R M N : Type*}
  [comm_semiring R] [add_comm_monoid M] [add_comm_monoid N]
  [module R M] [module R N] :
  module R (M ⊗[R] N) := sorry
```

The crucial difference is that `left_module` works with two separate rings, `R` and `S`. The `smul_comm_class R S M` assumption is needed to make the operation well-defined on the tensor product.

In the author's initial attempt to generalize this in [*mathlib#5317*], it needed an `is_scalar_tower S R M` assumption instead (in effect, the fact that `S` acts as a subring of `R`). The instance of `module R (M ⊗[A] P)` needed in eq. (10.16b) can be found under this scheme, as here `A` is an `R`-algebra; but it would not work for `module A (P ⊗[R] Q)`, where `R` and `A` have exchanged roles.

If we apply `tensor_product.left_module` to obtain the instances in eq. (10.16b), we find that it matches straightforwardly; the `module` assumptions we need are already present in eq. (10.16a), while the `smul_comm_class` instances can be located as follows:

- `module A (M ⊗[R] N)` needs `smul_comm_class R A M`, which since we have `algebra R A` is the same as `is_scalar_tower R A M`
- `module A (P ⊗[R] Q)` needs `smul_comm_class R A P`, which since we have `algebra R A` is the same as `is_scalar_tower R A P`
- `module R (M ⊗[A] P)` needs `smul_comm_class A R M`, which since we have `algebra R A` is

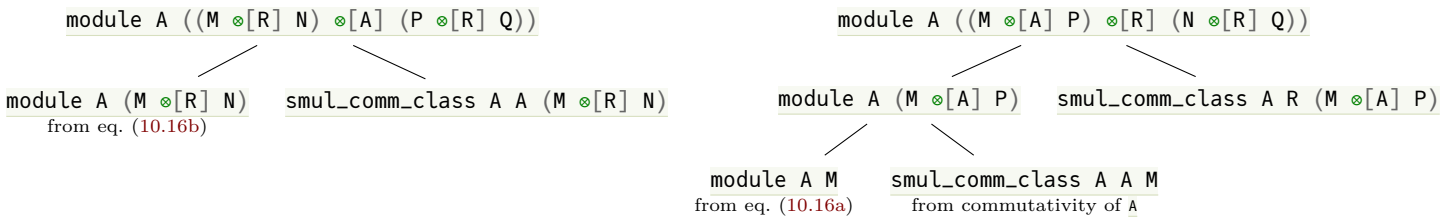


Figure 10.1.: Typeclass resolution for module structures on four-way tensor products

In these trees, children correspond to the assumptions needed when applying `tensor_product.module` to their parent.

the same as `is_scalar_tower R A M`

- `module R (N ⊗[R] Q)` is found automatically

If we do the same thing for eq. (10.16c), we find that not only do we need another module instance of the form we saw in eq. (10.16b) (which is straightforward to solve), but also that the `smul_comm_class` side goals are now rather trickier. The paths to these side goals are shown in fig. 10.1.

To resolve these `smul_comm_class` leaf nodes, the following instance from [*mathlib#19143*] is needed:

```
instance tensor_product.smul_comm_class_left {R S T M N : Type*}
  -- assumptions needed by `M ⊗[R] N`
  [comm_semiring R] [add_comm_monoid M] [add_comm_monoid N] [module R M] [module R N]
  -- assumptions needed by `module S (M ⊗[R] N)`
  [semiring S] [module S M] [smul_comm_class R S M]
  -- assumptions needed by `module S (M ⊗[R] N)`
  [semiring T] [module T M] [smul_comm_class R T M]
  -- assumptions needed for the proof!
  [smul_comm_class S T M] :
  smul_comm_class S T (M ⊗[R] N)
```

This allows us to reduce these nodes to goals of the form `smul_comm_class A A M` and `smul_comm_class A R M`, both of which we already solved above.

Finally then, we are ready to state *the type of* the isomorphism in eq. (10.14) in Lean:

```
def tensor_product.tensor_tensor_tensor_comm (R A M N P Q : Type*) [comm_semiring R]
  [comm_semiring R] [comm_semiring A] [algebra R A]
  [add_comm_monoid M] [module R M] [module A M] [is_scalar_tower R A M]
  [add_comm_monoid N] [module R N]
  [add_comm_monoid P] [module R P] [module A P] [is_scalar_tower R A P]
  [add_comm_monoid Q] [module R Q] :
  (M ⊗[R] N) ⊗[A] (P ⊗[R] Q) ≃ₗ[A] (M ⊗[A] P) ⊗[R] (N ⊗[R] Q) := sorry
```

This *statement* would not have elaborated had we not first defined `tensor_product.left_module` and `tensor_product.smul_comm_class_left`. The `sorry` here is a reminder that despite all this

work, we still haven't even started to *implement* the isomorphism in eq. (10.14)!

Implementing four-way commutativity in Lean

Let us now look at filling the `sorry` above. There are two natural ways to break apart four-way commutativity: by applying three-way right-commutativity on the left, or by applying three-way left-commutativity on the right. Respectively, these break down as:

$$
\begin{aligned}
&(M \otimes_R N) \otimes_A (P \otimes_R Q) && \\
&\cong_A ((M \otimes_R N) \otimes_A P) \otimes_R Q && \text{(10.17a)} \\
&\cong_A ((M \otimes_A P) \otimes_R N) \otimes_R Q && \text{(10.17b)} \\
&\cong_A (M \otimes_A P) \otimes_R (N \otimes_R Q) && \text{(10.17c)}
\end{aligned}
\qquad
\begin{aligned}
&(M \otimes_R N) \otimes_A (P \otimes_R Q) && \\
&\cong_A M \otimes_R (N \otimes_A (P \otimes_R Q)) && \text{(10.18a)} \\
&\cong_A M \otimes_A (P \otimes_R (N \otimes_R Q)) && \text{(10.18b)} \\
&\cong_A (M \otimes_A P) \otimes_R (N \otimes_R Q) && \text{(10.18c)}
\end{aligned}
$$

In each case, the process is to apply an associativity result (eqs. (10.17a) and (10.18a)), then the commutativity result (eqs. (10.17b) and (10.18b)), then another associativity result (eqs. (10.17c) and (10.18c)). The version in eq. (10.18) is how mathlib implements four-way commutativity in the special case when $A = R$. Unfortunately, for the heterobasic case the RHS at eq. (10.18a) is not well-typed, as it requires N to be an A-module, which is not one of our assumptions from eq. (10.16a). So in reality, only one of these two natural approaches (eq. (10.17)) is viable!

The two associativity results we need for eqs. (10.17a) and (10.17c) are in fact different; the first is in general $M \otimes_A (P \otimes_R Q) \cong_A (M \otimes_A P) \otimes_R Q$, while the second is in general $M \otimes_R (P \otimes_R Q) \cong_A (M \otimes_R P) \otimes_R Q$. The choice to keep the colors in that sentence should make it clear that the difference is in where the rings R and A appear. To avoid implementing associativity twice, we need to generalize even further and introduce a third base ring, B.

Associativity is then stated $M \otimes_A (P \otimes_R Q) \cong_B (M \otimes_A P) \otimes_R Q$, from which the two versions we need can be recovered by setting $(R, A, B) \coloneqq (R, A, A)$ and $(R, A, B) \coloneqq (R, R, A)$, respectively. The end result from [*mathlib4#6035*], this time with an implementation, is:

```
def assoc (R A B M P Q : Type*)
    -- B/A/R is a tower of algebras[14]
    [comm_semiring R] [comm_semiring A] [semiring B]
    [algebra R A] [algebra R B] [algebra A B]
    -- M is a module over B/A/R in a compatible way
    [add_comm_monoid M] [module R M] [module A M] [module B M]
    [is_scalar_tower R A M] [is_scalar_tower R B M] [is_scalar_tower A B M]
    -- P is module over A/R in a compatible way
    [add_comm_monoid P] [module R P] [module A P] [is_scalar_tower R A P]
    -- Q is a module over just R
    [add_comm_monoid Q] [module R Q] :
  (M ⊗[A] P) ⊗[R] Q ≃ₗ[B] M ⊗[A] (P ⊗[R] Q) :=
linear_equiv.of_linear
  (lift $ lift $ lcurry R A B P Q _ ∘ₗ mk A B M (P ⊗[R] Q))
  (lift $ uncurry R A B P Q _ ∘ₗ curry (mk R B _ Q))
  (by ext; rfl)
  (by ext; rfl)
```

While rather cryptic by itself due to the point-free computation style (section 5.3), this implementation hints that only the tip of the iceberg has been explored in this section; the linear maps `lcurry` and `uncurry` can be seen to take all three rings R, A, and B as arguments, indicating that they themselves are triply-heterobasic.

For brevity, we will omit the construction needed for eq. (10.17b), which can be found in [*mathlib4#6035*], as it is another even more cryptic point-free computation.

10.2.5. Tensor products of algebras

To make sense of the LHS of eq. (10.7), we need a way to consider $A \otimes_R B$ as an A-algebra when A and B are R-algebras. We can obtain this by combining our result that $A \otimes_R B$ is an A-*module* (in `tensor_product.left_module`) with the standard result that $A \otimes_R B$ is a ring characterized on pure tensors by $(a_1 \otimes b_1)(a_2 \otimes b_2) = a_1 a_2 \otimes b_1 b_2$ and $1 = 1 \otimes 1$. In fact, the collaborative nature of mathlib meant that the author's result was combined with this standard result by another contributor in [mathlib#15241].

There are two more key results we will need to build this isomorphism. The first is an extensionality lemma, which allows us to prove equality of algebra morphisms in terms of a simpler equality. The weak version of this statement is

Theorem 10.21 (Extensionality on algebra morphisms from the tensor product of algebras (weak))**.** *For a trio of R-algebras A, B, and C, and an intermediate R-algebra S such that $A/S/R$ and $C/S/R$ are towers of algebras, we can show equality of S-algebra morphisms f, g between $A \otimes_R B$ and C by showing that they agree on pure tensors; $f(a \otimes b) = g(a \otimes b)$.*

which follows trivially from the analogous statement for extensionality of linear maps.

[14] Interestingly `[is_scalar_tower R A B]` is not actually required here; in fact we only need that they form a tower modulo the kernel of their actions on `M`.

We say this is a "weak" extensionality lemma (in reference to our remarks about theorem 5.3) because it cannot be chained in the way described in section 5.1; if we had an equality on algebra maps $f, g : A \otimes_R (B \otimes_R C) \to_R D$ from a tensor product of three algebras into a fourth, we would not be able to apply theorem 10.21 twice, just as we could not do so for the analogous case of linear maps with theorem 5.3. As a recap: if we try, we find the first application turns an equality of morphisms into an equality of objects D in the codomain to which theorem 10.21 cannot be re-applied, leaving us to show $f(a \otimes x_{bc}) = g(a \otimes x_{bc})$ where $x_{bc} : B \otimes_R C$ is not a pure tensor. To be able to chain this result, we need an extensionality lemma that turns an equality of morphisms into an equality of "simpler" morphisms.

Theorem 10.22 (Extensionality on algebra morphisms from the tensor product of algebras (strong))**.** *For a trio of R-algebras A, B, and C, and an intermediate R-algebra S such that $A/S/R$ and $C/S/R$ are towers of algebras, we can show equality of S-algebra morphisms f, g between $A \otimes_R B$ and C by showing that:*

- *The compositions with the canonical inclusion $A \to A \otimes_R B$ (which is itself an S-algebra morphism) are equal.*
- *The compositions with the canonical inclusion $B \to A \otimes_R B$ (which is an R-algebra morphism) are equal.*

In Lean, this is written as:

```
theorem ext ⦃f g : (A ⊗[R] B) →ₐ[S] C⦄
  (ha : f.comp include_left = g.comp include_left)
  (hb : (f.restrict_scalars R).comp include_right = (g.restrict_scalars R).comp include_right) :
  f = g
```

where `restrict_scalars` is needed to cast f and g from S-algebra morphisms to R-algebra morphisms. Returning to our earlier $f, g : A \otimes_R (B \otimes_R C) \to_R D$ situation but applying theorem 10.22 instead of theorem 10.21, we are left with subgoals of $f(a \otimes 1 \otimes 1) = g(a \otimes 1 \otimes 1)$, $f(1 \otimes b \otimes 1) = g(1 \otimes b \otimes 1)$, and $f(1 \otimes 1 \otimes c) = g(1 \otimes 1 \otimes c)$. Our motivation for needing this strong result is not actually that we want to chain[15] extensionality results when dealing with tensor products like $A \otimes_R (B \otimes_R C)$; but that we want to chain theorem 10.22 with the extensionality results we obtained for the Clifford algebra, theorem 8.2!

The second result we need is a universal property for tensor products of algebras from [82, Proposition II.4.1]:

[15] In the sense of section 5.1.

Theorem 10.23 (The heterobasic universal property for the tensor product of algebras)**.** *For a trio of R-algebras A, B, and C, and an intermediate R-algebra S such that $A/S/R$ and $C/S/R$ are towers of algebras, there is a one-to-one correspondence between:*

$$\begin{array}{c|c|c} \text{pairs of algebra morphisms} & & \text{algebra morphisms} \\ f : A \to_S C \text{ and } g : B \to_R C & \text{and} & F : A \otimes_R B \to_S C. \\ \text{such that } f(a) \text{ and } g(b) \text{ commute} & & \end{array}$$

The forward direction of this correspondence sends the pair (f, g) to $x \mapsto f(x)g(x)$, while the reverse direction sends the morphism F to the pair $(F \circ (a \mapsto a \otimes_R 1), F \circ (b \mapsto 1 \otimes_R b))$. The construction of this was added in [*mathlib4#7409*] as a generalization of an existing result for *commutative* rings. The fact these are inverses follows trivially via `ext` and `simp`, thanks to theorem 10.22.

10.2.6. Base change of Clifford algebras

Having resolved the last issues with the *statement* of eq. (10.7) in section 10.2.5, we can now consider constructing it; this subsection summarizes [*mathlib4#6778*]. We will return to the strategy outlined in chapter 10 that uses universal properties and extensionality.

The forward map

The forward map, with type $A \otimes_R \mathcal{G}(V, Q) \to_A \mathcal{G}(A \otimes_R V, Q_A)$, can be stated in Lean as

```
def ofBaseChange (Q : QuadraticForm R V) :
  A ⊗[R] CliffordAlgebra Q →ₐ[A] CliffordAlgebra (Q.baseChange A) :=
```

To apply theorem 10.23 to construct this, we need a way to consider $\mathcal{G}(A \otimes_R V, Q_A)$ as an R-algebra; until now, we have considered it only as an A-algebra (as Q_A is of type $V \to A$). The easy way to do this is to define the R-algebra structure as the A-algebra structure composed with the canonical map $R \to A$; indeed, this is mathematically fine, but causes trouble in Lean due to the typeclass instance diamonds mentioned in section 4.5.2, especially if $R = \mathbb{Z}$. We instead need to take a more involved approach, adding first the corresponding algebraic structure for free algebras in [*mathlib4#6072*] (where we genuinely can compose with the canonical morphism as long as we are careful) and quotients by rings in [*mathlib4#6066*], from which the results for tensor algebras in [*mathlib4#6073*] and finally Clifford algebras in [*mathlib4#6074*] follow trivially. In all these cases, we are obligated not only to show that there is an R-algebra structure, but that it factors through the existing A-algebra structure (as `is_scalar_tower R A _`).

With these in place, we invoke theorem 10.23 using the piecewise morphisms

$$f : \quad A \to_A \mathcal{G}(A \otimes_R V, Q_A) \coloneqq x \mapsto x \qquad \text{(the canonical embedding)} \tag{10.19}$$

$$g : \mathcal{G}(V, Q) \to_R \mathcal{G}(A \otimes_R V, Q_A) \coloneqq \text{lift}_{\mathcal{G}}[v \mapsto \iota(1 \otimes v)] \tag{10.20}$$

where g incurs a straightforward proof obligation to show that $\iota(a \otimes v)^2 = Q_a(1 \otimes v) = 1^2 Q(v)$,

and theorem 10.23 incurs a trivial one to show that the canonical embedding commutes.

The reverse map

The reverse direction, with statement

```
def toBaseChange (Q : QuadraticForm R V) :
  CliffordAlgebra (Q.baseChange A) →ₐ[A] A ⊗[R] CliffordAlgebra Q
```

is somewhat more challenging. This does not stem from finding the function with which to invoke theorem 8.1; we use the obvious choice $\mathrm{lift}_{\mathcal{G}}[\mathrm{id} \otimes_R \iota]$, which sends $\iota(a \otimes v)$ to $a \otimes \iota(v)$. The challenge arises in proving that this is well-behaved, namely that

$$([\mathrm{id} \otimes_R \iota](v'))^2 = Q_A(v') \tag{10.21}$$

where $v' : A \otimes_R V$ is an arbitrary tensor that may not be a pure tensor. Standard induction on the tensor product leaves us in a mess here, as we do not have an easy way to express the $Q_A(v' + w')$ term that appears in our induction in terms of $Q_A(v')$ and $Q_A(w')$.

We have seen a similar situation before; in the context of `ext`, we found that theorem 10.21 left us with an unpleasant non-pure tensor to deal with, due to being stated in a "weak" way in theorem 10.21 instead of the "strong" in theorem 10.22. Adapting the solution used there, we will resolve our difficulty with eq. (10.21) by rephrasing the $\forall v, f(v)^2 = Q(v)$ condition of theorem 8.1 (where $f : V \to_R A$) to be in terms of an equality of morphisms instead of a universally quantified equality of elements. We will restrict ourselves to the case where 2 is invertible in A, since this is implied by our earlier condition in section 10.2.2 that 2 is invertible in R.

We first note that for $f : V \to_R A$ in a general Clifford algebra as in theorem 8.2, where 2 is invertible in R, the requirement $\forall v, f(v)^2 = Q(v)$ is equivalent to requiring $\forall v, \forall w, f(v)f(w) + f(w)f(v) = 2B(v, w)$, where B is the bilinear form associated with Q. The doubly-quantified version can be interpreted in the language of geometric algebra as saying "the map f preserves the 'inner' product of two vectors". The forward direction of this equivalence follows by writing[16] $f(v)f(w) + f(w)f(v) = f(v + w)^2 - f(v)^2 - f(w)^2$, while the reverse direction is trivial.

This double-quantification is very valuable to us, as it turns an equality where both sides are quadratic, into an equality where both sides are bilinear; so we can instead formalize it as an equality of bilinear maps:

```
theorem forall_mul_self_eq_iff (h2 : IsUnit (2 : A)) (f : M →ₗ[R] A) :
    (∀ x, f x * f x = algebraMap _ _ (Q x)) ↔
      (LinearMap.mul R A).compl₂ f ∘ₗ f + (LinearMap.mul R A).flip.compl₂ f ∘ₗ f =
        Q.polarBilin.toLin.compr₂ (Algebra.linearMap R A) := by
```

The right-hand side is written in a point-free style, and so is rather hard to read; but the motivation

[16] Which bears more than a passing resemblance to the definition of the polar operation on quadratic forms, suggesting a generalization of quadratic *forms* to quadratic *maps* landing in another *module* rather than the base ring R. Other members of the mathlib community are already attempting this generalization in [mathlib4#7569] for unrelated reasons.

```
-- The variables used in the samples below
let f : A ⊗[R] V →ₗ[A] A ⊗[R] CliffordAlgebra Q :=
  TensorProduct.AlgebraTensorModule.map .id (ι Q)
let LHS : A ⊗[R] V →ₗ[A] A ⊗[R] V →ₗ[A] A ⊗[R] CliffordAlgebra Q :=
  (mul R A).compl₂ f ∘ₗ f + (mul R A).flip.compl₂ f ∘ₗ f
let RHS : A ⊗[R] V →ₗ[A] A ⊗[R] V →ₗ[A] A ⊗[R] CliffordAlgebra Q :=
  Q.polarBilin.toLin.compr₂ (Algebra.linearMap R A)
```

```
LHS = RHS                                                    (A ⊗[R] V) →[R] (A ⊗[R] V) →[R] 𝒢(A ⊗[R] V, Q_A)
```
↓ extensionality from a tensor product, theorem 5.5
```
LHS ∘ₗ mk = RHS ∘ₗ mk                                        A →[R] V →[R] (A ⊗[R] V) →[R] 𝒢(A ⊗[R] V, Q_A)
```
↓ extensionality from a ring, theorem 5.2
```
(LHS ∘ₗ mk) 1 = (RHS ∘ₗ mk) 1                                V →[R] (A ⊗[R] V) →[R] 𝒢(A ⊗[R] V, Q_A)
```
↓ extensionality
```
(LHS ∘ₗ mk) 1 v = (RHS ∘ₗ mk) 1 v                            (A ⊗[R] V) →[R] 𝒢(A ⊗[R] V, Q_A)
```
↓ extensionality from a tensor product, theorem 5.5
```
(LHS ∘ₗ mk) 1 v ∘ₗ mk = (RHS ∘ₗ mk) 1 v ∘ₗ mk                A →[R] V →[R] 𝒢(A ⊗[R] V, Q_A)
```
↓ extensionality from a ring, theorem 5.2
```
((LHS ∘ₗ mk) 1 v ∘ₗ mk) 1 = ((RHS ∘ₗ mk) 1 v ∘ₗ mk) 1        V →[R] 𝒢(A ⊗[R] V, Q_A)
```
↓ extensionality
```
((LHS ∘ₗ mk) 1 v ∘ₗ mk) 1 w = ((RHS ∘ₗ mk) 1 v ∘ₗ mk) 1 w    𝒢(A ⊗[R] V, Q_A)
```
↓ dsimp: definitional simplification
```
f (1⊗ₜv)*f (1⊗ₜw) + f (1⊗ₜw)*f (1⊗ₜv) = algebraMap R A ((Q.baseChange A).polar (1 ⊗ₜ v + 1 ⊗ₜ w))
```
↓ further rewriting and substituting f
```
1 ⊗ₜ (ι Q v*ι Q w + ι Q w*ι Q v) = algebraMap R A (Q.polar (v + w))
```

Figure 10.2.: The process taken by the `ext` tactic to turn the bilinear version of the equality in eq. (10.21) into a statement about pure tensors, along with the result of a final stage of cleanup.

Each box is a goal state passed through during the proof, and the expression to the right indicates the type of the equality.

here was not readability, but to make our proof easier. The reason this helps is that in our example, the equality on the right is between bilinear maps $(A \otimes_R V) \to_R (A \otimes_R V) \to_R \mathcal{G}(A \otimes_R V, Q_A)$; this means we can chain the standard extensionality results for linear maps from tensor products (theorem 5.5) and linear maps from rings, and be left with an equality of our function applied to pure tensors with 1, $([\mathrm{id} \otimes_R \iota](1 \otimes v))([\mathrm{id} \otimes_R \iota](1 \otimes w)) = 2B_A(v, w)$. The exact mechanics of this, along with the trivial goal state this leaves us with, are shown in fig. 10.2.

Once again, the fact that the forward and reverse maps are inverses follow trivially from `ext`; this time, this tactic reduces these proofs to showing the maps are inverses on `1 ⊗ₜ[R] ι Q v` and `ι (Q.baseChange A) (1 ⊗ₜ[R] x)`.

Properties

Besides the algebraic properties that we get for free from the fact eq. (10.7) is an isomorphism of algebras, we would also like to prove how it interacts with involution and reversion.

The statements of these results are as follows

```
/-- The involution acts only on the right of the tensor product. -/
theorem toBaseChange_involute (Q : QuadraticForm R V) (x : CliffordAlgebra (Q.baseChange A)) :
    toBaseChange A Q (involute x) =
      TensorProduct.map LinearMap.id (involute.toLinearMap) (toBaseChange A Q x) :=

/-- `reverse` acts only on the right of the tensor product. -/
theorem toBaseChange_reverse (Q : QuadraticForm R V) (x : CliffordAlgebra (Q.baseChange A)) :
    toBaseChange A Q (reverse x) =
      TensorProduct.map LinearMap.id reverse (toBaseChange A Q x) := sorry
```

In both cases we prove these by first stating the point-free version, and using `ext` to prove it. For the `involute` case, this statement is straightforward,

```
theorem toBaseChange_comp_involute (Q : QuadraticForm R V) :
    (toBaseChange A Q).comp involute =
      (Algebra.TensorProduct.map (.id _ _) involute).comp (toBaseChange A Q) := sorry
```

since `involute` is an algebra morphism. For `reverse`, which is only a linear map, this approach doesn't work; the analogous point-free linear map spelling will not allow us to apply theorem 10.22's result about algebra morphisms. If we apply a weaker extensionality result for linear maps, then we are instead left unable to apply extensionality on algebra morphisms from the Clifford algebra, theorem 8.2; and here, there simply is no linear version.

It is here that our `reverseOp : CliffordAlgebra Q →ₐ[R] (CliffordAlgebra Q)ᵐᵒᵖ` from section 9.3.3 finally becomes of value, proving we are willing to go deep into point-free nonsense:

```
theorem toBaseChange_comp_reverseOp (Q : QuadraticForm R V) :
  (toBaseChange A Q).op.comp reverseOp =
    ((Algebra.TensorProduct.opAlgEquiv R A A (CliffordAlgebra Q)).toAlgHom.comp <|
      (Algebra.TensorProduct.map (AlgEquiv.toOpposite A A).toAlgHom reverseOp).comp
      (toBaseChange A Q)) := sorry
```

This `ᵐᵒᵖ` allows us to remain in the world of algebra morphisms (powering up `ext`), but it also comes with a curse; we now need yet another isomorphism to move `ᵐᵒᵖ` through tensor products! The one in question is `opAlgEquiv : Aᵐᵒᵖ ⊗[R] Bᵐᵒᵖ ≃ₐ[S] (A ⊗[R] B)ᵐᵒᵖ`, which is a formalization from [*mathlib4#6555*] of:

Theorem 10.24 (The opposite functor commutes with the tensor product)**.** *For a pair of R-algebras A and B where A is additionally an S-algebra in a compatible way, we have a canonical isomorphism of S-algebras*

$$A^{\mathrm{op}} \otimes_R B^{\mathrm{op}} \cong_S (A \otimes_R B)^{\mathrm{op}} \tag{10.22}$$

which identifies $\mathrm{op}(a) \otimes_R \mathrm{op}(b)$ *with* $\mathrm{op}(a \otimes_R b)$*.*

The construction has been omitted from this thesis as it is uninteresting.

10.3. Direct sums of quadratic vector spaces

As briefly mentioned in section 2.1.4, when working with non-degenerate Clifford algebras over the reals, we typically invoke Sylvester's law of inertia to restrict our vector space to $\mathbb{R}^{p,q}$, without any meaningful loss of generality. Under this parametrization, a complete classification can be obtained through reduction to the table where $p < 8$ and $q < 8$ [68, §4 Table II], and extended through Bott periodicity. This table of *only* 64 entries would be an excellent addition to mathlib's Clifford algebra library, but has been left by the author as an exercise for future contributors.

Instead, in this section we focus on a less useful (but more interesting) approach to handling direct sums of quadratic vector spaces [68, Proposition 1.5; 80, §2.1]:

Theorem 10.25. *For a commutative ring R and two R-modules V and W with associated quadratic forms Q_V and Q_W, we have an isomorphism of algebras*

$$\mathcal{G}(V \oplus W, Q_V \oplus Q_W) \cong \mathcal{G}(V, Q_V) \mathbin{\hat{\otimes}} \mathcal{G}(W, Q_W)$$

which identifies $\iota(v + w)$ *with* $\iota(v) \mathbin{\hat{\otimes}} 1 + 1 \mathbin{\hat{\otimes}} \iota(w)$*,*

where $Q_V \oplus Q_W$ is the direct sum of quadratic forms, and $\hat{\otimes}$ is the $\mathbb{Z}_2$-graded tensor product of graded algebras. We can apply theorem 10.25 to $\mathbb{R}^{p,q}$ by noting that it can be thought of as a direct sum $\mathbb{R}^p \oplus \mathbb{R}^q$, where the left summand has the euclidean form and the right summand has the negation thereof.

Theorem 10.25 could *almost* be used to recover theorem 10.11, since using theorem 10.6 and theorem 10.8 we have

$$\mathbb{H}[\epsilon] \cong_{\mathbb{R}} (\mathbb{H} \otimes_{\mathbb{R}} \mathbb{R}[\epsilon]) \cong_{\mathbb{R}} \left(\mathcal{G}(\mathbb{R}^{0,2}) \otimes_{\mathbb{R}} \mathcal{G}(\mathbb{R}^{0,0,1})\right), \tag{10.23}$$

and we can use theorem 10.25 to obtain

$$\left(\mathcal{G}(\mathbb{R}^{0,2}) \mathbin{\hat{\otimes}}_{\mathbb{R}} \mathcal{G}(\mathbb{R}^{0,0,1})\right) \cong_{\mathbb{R}} \mathcal{G}(\mathbb{R}^{0,2} \oplus \mathbb{R}^{0,0,1}) \cong_{\mathbb{R}} \mathcal{G}(\mathbb{R}^{0,2,1}); \tag{10.24}$$

but to finish the construction we would need an equivalence between the regular and graded tensor products, a task made challenging by the sign flips in multiplication which await us in

eq. (10.26).

10.3.1. Direct sums of quadratic forms

The direct sum of two[17] quadratic forms $Q_V \oplus Q_W$ is defined such that $(Q_V \oplus Q_W)((v, w)) = Q(v) + Q(w)$, which is straightforward to formalize [*mathlib#10939*] as

```
@[simps]
def prod (Q₁ : quadratic_form R M₁) (Q₂ : quadratic_form R M₂) : quadratic_form R (M₁ × M₂) :=
  Q₁.comp (linear_map.fst _ _ _) + Q₂.comp (linear_map.snd _ _ _)
```

where the `@[simps]` automatically generates the characterizing lemma that `Q₁.prod Q₂ m = Q₁ m.1 + Q₂ m.2`. Of particular relevance to theorem 10.25 is the fact that $Q_V \oplus Q_W$ is such that the spaces identified with V and W are orthogonal, and so when embedded into the Clifford algebra, vectors taken from different spaces commute:

$$\iota((v, 0))\iota((0, w)) = -\iota((0, w))\iota((v, 0)) \tag{10.25}$$

To formalize this property, it is helpful to have a definition of orthogonality. For bilinear forms, this definition is obvious; we declare v and w orthogonal if $B(v, w) = 0$. For quadratic forms in the very general case without an associated symmetric bilinear form, we declare v and w orthogonal when $Q(v+w) = Q(v)+Q(w)$ (as opposed to talking about the associated bilinear form satisfying $B(v, w) = 0$), as in practice this is precisely the condition needed by $\iota(v)\iota(w) = -\iota(w)\iota(v)$:

```
def IsOrtho (Q : QuadraticForm R M) (x y : M) : Prop :=
  Q (x + y) = Q x + Q y
```

It is straightforward to show that this is consistent with the definition of orthogonality on bilinear forms, and this was contributed to mathlib along with the definition in [*mathlib4#9141*].

With this in place, we can show that elements of the direct sum of vector spaces are orthogonal with respect to $Q_V \oplus Q_W$ if their components are orthogonal:

```
theorem IsOrtho.prod {Q₁ : QuadraticForm R M₁} {Q₂ : QuadraticForm R M₂}
    {v w : M₁ × M₂} (h₁ : Q₁.IsOrtho v.1 w.1) (h₂ : Q₂.IsOrtho v.2 w.2) :
    (Q₁.prod Q₂).IsOrtho v w :=
  (congr_arg₂ HAdd.hAdd h₁ h₂).trans <| add_add_add_comm _ _ _ _
```

and then the fact that the spaces identified with V and W are orthogonal follows trivially as a special case:

```
@[simp] theorem IsOrtho.inl_inr {Q₁ : QuadraticForm R M₁} {Q₂ : QuadraticForm R M₂}
    (m₁ : M₁) (m₂ : M₂) :
    (Q₁.prod Q₂).IsOrtho (m₁, 0) (0, m₂) := .prod (.zero_right _) (.zero_left _)
```

Equation (10.25) then follows by eliminating the extra term from eq. (2.14) (`ι_mul_comm`).

[17] [*mathlib#10939*] also defines the n-ary direct sum of quadratic forms, but we will not need it in this section.

10.3.2. The tensor product of graded algebras

The real challenge in formalizing theorem 10.25, and the reason that it is interesting, is not the direct sum of quadratic forms $Q_V \oplus Q_W$, but the tensor product of graded algebras $A \hat{\otimes}_R B$. Mathematically, as a module it is "the same as" the regular tensor product, with the same multiplicative unit $1 \otimes 1$, but endowed with a different multiplicative structure which satisfies

$$(a \otimes b)(a' \otimes b') = (-1)^{\deg a' \deg b}(aa') \otimes (bb'), \tag{10.26}$$

where $a, a' : A$ and $b, b' : B$ are homogeneous elements of a single grade, and $\deg \cdot$ is the index of that grade. Intuitively, the sign factor can be thought of accounting for the action of exchanging b and a' one grade at a time.

While for theorem 10.25 we need only consider the case when $\deg \cdot : \mathbb{Z}_2$, we would prefer to generalize to a setting where $\deg \cdot : \mathbb{N}$ is also supported, so that the same construction can be use for the exterior algebra. We could achieve this by following Bourbaki [46, §4.7 Example 2] and replacing $(-1)^{\deg a' \deg b}$ in eq. (10.26) with some arbitrary "commutation factor", but this is overkill for our purposes. Instead, we write the rather strange typeclass assumption `Module ι (Additive ℤˣ)`, which exploits the fact that the lawfulness of a module structure is, through a change in notation swapping `+` for `•`, the same as a lawfulness of a power operator [*mathlib4#7866*].

With this in mind, we are ready to state the typeclass assumptions needed by the graded tensor product:

```
variable {R ι A B : Type*}

variable [CommSemiring ι] [Module ι (Additive ℤˣ)] [DecidableEq ι]
variable [CommRing R] [Ring A] [Ring B] [Algebra R A] [Algebra R B]
variable (𝒜 : ι → Submodule R A) (ℬ : ι → Submodule R B)
variable [GradedAlgebra 𝒜] [GradedAlgebra ℬ]
```

After writing the last line and `Module ι (Additive ℤˣ)`, the Lean error messages provide the context needed to deduce all the earlier lines.

The "inherit some algebraic structure but replace other algebraic structure" scenario we desire here is well-described by the type synonym pattern from section 4.6; we can define the graded tensor product as nominally equal to the tensor product:

```
@[nolint unusedArguments]
def GradedTensorProduct
    (𝒜 : ι → Submodule R A) (ℬ : ι → Submodule R B) [GradedAlgebra 𝒜] [GradedAlgebra ℬ] : Type _ :=
  A ⊗[R] B

@[inherit_doc GradedTensorProduct]
scoped[TensorProduct] notation:100 𝒜 " ᵍ⊗[" R "] " ℬ:100 => GradedTensorProduct R 𝒜 ℬ

variable (R) in
/-- The casting equivalence to move between regular and graded tensor products. -/
def of : A ⊗[R] B ≃ₗ[R] 𝒜 ᵍ⊗[R] ℬ := LinearEquiv.refl _ _
```

The `nolint unusedArguments` is somewhat interesting; without it, Lean complains that we did not actually use $\mathcal{A}$ and $\mathcal{B}$, and so surely we have made a mistake! The pattern here is that of *parametrized* type synonyms, which by attaching extra data to the type (here, the choice of submodules that comprise the grading) permit this data to be available during typeclass search. In our case $\mathcal{A}$ and $\mathcal{B}$ are vital to assembling the multiplication.

With the type and `ᵍ⊗[R]` notation[18] in place, we can copy across the additive structure, module structure, and the definition of 1.

```
instance : AddCommGroupWithOne (𝒜 ᵍ⊗[R] ℬ) :=
  Algebra.TensorProduct.instAddCommGroupWithOne
instance : Module R (𝒜 ᵍ⊗[R] ℬ) := TensorProduct.leftModule
```

It is a convenient coincidence here that `AddCommGroupWithOne`, the mathematically unusual typeclass needed to resolve the instance diamonds for algebras in section 4.5.2, happens to capture both the additive structure and the definition of 1 that we wish to copy!

Multiplicative structure

What remains is to define the multiplication characterized by eq. (10.26), and demonstrate it satisfies the axioms of a ring and an algebra. In practice, it is easiest for us to do this in two phases; by first defining the operation as a bilinear map on the regular tensor product `(⊕ i, 𝒜 i) ⊗[R] (⊕ i, ℬ i)`, and then transferring through the isomorphisms provided by `GradedAlgebra` to obtain the operation on `𝒜 ᵍ⊗[R] ℬ`.

There is one more interesting result about graded tensor products that is worth noting; the canonical braiding of the graded tensor product that exchanges the two arguments:

Theorem 10.26. *For a commutative ring R and two R-algebra A and B graded (over a suitable index) by submodules $\mathcal{A}$ and $\mathcal{B}$, we have an isomorphism of R-algebras*

$$\mathrm{comm} : A \hat{\otimes}_R B \cong B \hat{\otimes}_R A$$

characterized on homogeneous elements $a : \mathcal{A}_i$ and $b : \mathcal{B}_j$ by $\mathrm{comm}(a \otimes b) = (-1)^{ij}(b \otimes a)$.

[18] Using the notation from Bourbaki instead of $\hat{\otimes}$, as the latter renders poorly in some code fonts.

While this is ultimately a distraction for theorem 10.25, it is a convenient waypoint to route through when building the multiplication characterized in eq. (10.26); indeed, it permits us to construct the multiplication by chaining suitable maps.

We start with the type of bilinear maps that our multiplication belongs to,

$$(\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i) \to (\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i) \to (\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i), \tag{10.27}$$

which from the universal property of the tensor product is the same as

$$\cong ((\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i)) \otimes ((\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i)) \to (\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i). \tag{10.28}$$

Via standard results on tensor product of modules, this reassociates to

$$\cong (\bigoplus_i \mathcal{A}_i) \otimes ((\bigoplus_i \mathcal{B}_i) \otimes (\bigoplus_i \mathcal{A}_i)) \otimes (\bigoplus_i \mathcal{B}_i)) \to (\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i), \tag{10.29}$$

where the central term can be passed through comm from theorem 10.26 to give

$$\cong (\bigoplus_i \mathcal{A}_i) \otimes ((\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i)) \otimes (\bigoplus_i \mathcal{B}_i)) \to (\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i). \tag{10.30}$$

Further reassociation gives

$$\cong ((\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{A}_i)) \otimes ((\bigoplus_i \mathcal{B}_i) \otimes (\bigoplus_i \mathcal{B}_i)) \to (\bigoplus_i \mathcal{A}_i) \otimes (\bigoplus_i \mathcal{B}_i), \tag{10.31}$$

which we can construct (non-surjectively, hence the $\leftarrow$) from a pair of bilinear maps on $\bigoplus_i \mathcal{A}_i$ and $\bigoplus_i \mathcal{B}_i$ separately:

$$\leftarrow ((\bigoplus_i \mathcal{A}_i) \to (\bigoplus_i \mathcal{A}_i) \to (\bigoplus_i \mathcal{A}_i)) \otimes ((\bigoplus_i \mathcal{B}_i) \to (\bigoplus_i \mathcal{B}_i) \to (\bigoplus_i \mathcal{B}_i)). \tag{10.32}$$

The two maps needed in eq. (10.32) are precisely the multiplication maps provided by the ring structure on $\bigoplus_i \mathcal{A}_i$, which follow from our treatment of externally-graded rings in section 6.3.3.

Trading one obtuse pile of symbols for another, we can write this in Lean (in a slightly different order) as:

```
/-- The multiplication operation for tensor products of externally `ι`-graded algebras. -/
noncomputable irreducible_def gradedMul :
    letI AB := (⊕ i, 𝒜 i) ⊗[R] (⊕ i, ℬ i)
    AB →ₗ[R] AB →ₗ[R] AB := by
  refine TensorProduct.curry ?_                                              -- eq. (10.27)
  refine map (LinearMap.mul' R (⊕ i, 𝒜 i)) (LinearMap.mul' R (⊕ i, ℬ i)) ∘ₗ ?_      -- eq. (10.32)
  refine (assoc R ..).symm.toLinearMap ∘ₗ .lTensor _ ?_ ∘ₗ (assoc R ..).toLinearMap  -- eqs. (10.28)
  refine (assoc R ..).toLinearMap ∘ₗ .rTensor _ ?_ ∘ₗ (assoc R ..).symm.toLinearMap  --   and (10.30)
  exact (gradedComm ..).toLinearMap                                          -- eq. (10.29)
```

where `gradedComm` is theorem 10.26 written in the externally graded form, with type `(⊕ i, 𝒜 i)`

`⊗[R] (⊕ i, 𝓑 i) ≃ₗ[R] (⊕ i, 𝓑 i) ⊗[R] (⊕ i, 𝓐 i)`.

The definition of `gradedComm` itself is a little easier, and can be built by first distributing $(\bigoplus_i \mathcal{A}_i) \otimes_R (\bigoplus_i \mathcal{B}_i)$ as $\bigoplus_{i,j} \mathcal{A}_i \otimes_R \mathcal{B}_j$, then applying the commutation and sign adjustment componentwise on the direct sum.

The proofs that `gradedMul` satisfies the axioms of a monoid follow the approach described in section 5.3 (and used in section 6.3.3) of restating the results in a point-free style, and applying the `ext` tactic. The application of `ext` used for proving associativity is unusually industrious; the proof begins as

```
theorem gradedMul_assoc (x y z : DirectSum _ 𝓐 ⊗[R] DirectSum _ 𝓑) :
    gradedMul R 𝓐 𝓑 (gradedMul R 𝓐 𝓑 x y) z = gradedMul R 𝓐 𝓑 x (gradedMul R 𝓐 𝓑 y z) := by
  let mA := gradedMul R 𝓐 𝓑
  -- restate as an equality of morphisms so that we can use `ext`
  suffices LinearMap.llcomp R _ _ _ mA ∘ₗ mA =
      (LinearMap.llcomp R _ _ _ LinearMap.lflip <| LinearMap.llcomp R _ _ _ mA.flip ∘ₗ mA).flip by
    exact FunLike.congr_fun (FunLike.congr_fun (FunLike.congr_fun this x) y) z
  ext ixa xa ixb xb iya ya iyb yb iza za izb zb
  sorry
```

where `ext` introduces *twelve* variables (as there are three arguments with two sides apiece, each with an index and a value), and saved us from the alternative approach of manually performing and closing side-goals of *nine* separate `induction`s.

One additional benefit of factoring our definition of multiplication through theorem 10.26 is that it immediately becomes clear that we can restate eq. (10.26) more generally such that it only requires the inner b and a' (and not the outer a and b') to be of homogeneous degree, since we did not split apart the direct sums in eqs. (10.27) to (10.31) except after eq. (10.29); and here, we only did so to the middle two of the four components.

Having built the externally-graded spelling of graded multiplication, it is straightforward to pull it back along the following equivalence:

```
noncomputable def auxEquiv : (𝓐 ᵍ⊗[R] 𝓑) ≃ₗ[R] ((⊕ i, 𝓐 i) ⊗[R] (⊕ i, 𝓑 i)) :=
  let fA := (decomposeAlgEquiv 𝓐).toLinearEquiv
  let fB := (decomposeAlgEquiv 𝓑).toLinearEquiv
  (of R 𝓐 𝓑).symm.trans (TensorProduct.congr fA fB)
```

where the type on the LHS of the `≃ₗ[R]` is the internally-graded spelling, and the type on the RHS is the externally-graded one.

We will find it helpful to provide two more algebraic results; the fact that the maps $a \mapsto a \hat{\otimes} 1$ and $b \mapsto 1 \hat{\otimes} b$ are morphisms of algebras[19]. We provide these as `includeLeft : A →ₐ[R] 𝓐 ᵍ⊗[R] 𝓑` and `includeRight : B →ₐ[R] (𝓐 ᵍ⊗[R] 𝓑)`.

[19] In fact under a suitable graduation of $A \hat{\otimes}_R B$ they are morphisms of *graded* algebras, but we do not go as far as constructing that graduation here.

Universal property

Our "universal property" hammer hasn't failed us yet; so to build the isomorphism in theorem 10.25, we naturally want to go through another universal property. The one that we want, [46, §4.7, Proposition 10 (iii)][20], is very similar to theorem 10.23, and is stated as

Theorem 10.27 (The universal property for the graded tensor product of algebras)**.** *For a trio of R-algebras A, B, and C where A and B are respectively graded (over a suitable index) by submodules $\mathcal{A}$ and $\mathcal{B}$, there is a one-to-one correspondence between:*

$$\begin{array}{c|c|c} \text{pairs of algebra morphisms} & & \\ f_a : A \to_S C \text{ and } f_b : B \to_R C & \text{and} & \text{algebra morphisms} \\ \text{that on homogeneous elements } a : \mathcal{A}_i \text{ and } b : \mathcal{B}_j & & F : A \mathbin{\hat{\otimes}}_R B \to_R C. \\ \text{satisfy } f_a(a) g_b(b) = (-1)^{ij} f_b(b) f_a(a) & & \end{array}$$

The construction is defined such that $F(a \hat{\otimes} b) = f_a(a) f_b(b)$ for homogeneous a, b, $f_a(a) = F(a \hat{\otimes} 1)$, and $f_b(b) = F(1 \hat{\otimes} b)$.

For brevity, we omit the full Lean construction here (unsurprisingly it uses yet-more point-free and `ext` gymnastics), but the full proofs (along with everything else found under section 10.3.2) are now in mathlib due to the author's [*mathlib4#7394*].

The last piece we need is an extensionality lemma, with statement

Theorem 10.28 (Extensionality for the the graded tensor product of algebra)**.** *For a trio of R-algebras A, B, and C where A and B are respectively graded (over a suitable index) by submodules $\mathcal{A}$ and $\mathcal{B}$, to show a pair of algebra morphisms $F, G : A \mathbin{\hat{\otimes}}_R B \to_R C$ are equal, it suffices to show they agree when composed with each of the canonical morphisms $a \mapsto a \hat{\otimes} 1$ and $b \mapsto 1 \hat{\otimes} b$.*

Using the same strategy as we did for theorem 8.2, this follows from pulling the equality back along the equivalence defined by the universal property in theorem 10.27, and equating the two halves of the resulting pairs $(f_a, f_b) = (g_a, g_b)$.

10.3.3. Constructing the isomorphism

By this point the reader is likely rather tired of point-free nonsense, so we shall cut to the characterization of theorem 10.25 rather than showing its full construction.

The forward map, as written in [68, Proposition 1.5], is characterized by:

```
def ofProd : CliffordAlgebra (Q₁.prod Q₂) →ₐ[R] (evenOdd Q₁ ᵍ⊗[R] evenOdd Q₂) := sorry
lemma ofProd_ι_mk (m₁ : M₁) (m₂ : M₂) :
  ofProd Q₁ Q₂ (ι _ (m₁, m₂)) = ι Q₁ m₁ ᵍ⊗ₜ 1 + 1 ᵍ⊗ₜ ι Q₂ m₂ := sorry
```

The construction follows by invoking theorem 8.1 with an inscrutable point-free spelling of

[20] Bourbaki develops things in the generality of n-ary tensors products, which do exist in mathlib [mathlib#5311], but as an entirely parallel development. It would be possible but unrewarding to repeat the entirety of section 10.3 for this n-ary case.

$(m_1, m_2) \mapsto \iota(m_1) \hat{\otimes} 1 + 1 \hat{\otimes} \iota(m_2)$[21]. We are left to prove a slightly fiddly but ultimately simple algebraic statement about the square of this map.

The reverse map is rather more work (and not provided constructively by [68]), and characterized by:

```
def toProd : evenOdd Q₁ ᵍ⊗[R] evenOdd Q₂ →ₐ[R] CliffordAlgebra (Q₁.prod Q₂) := sorry
lemma toProd_ι_tmul_one (m₁ : M₁) : toProd Q₁ Q₂ (ι _ m₁ ᵍ⊗ₜ 1) = ι _ (m₁, 0) := sorry
lemma toProd_one_tmul_ι (m₂ : M₂) : toProd Q₁ Q₂ (1 ᵍ⊗ₜ ι _ m₂) = ι _ (0, m₂) := sorry
```

and follows from an application of theorem 10.27 with the obvious inclusions $\mathcal{G}(V, Q_V) \to \mathcal{G}(V \oplus W, Q_V \oplus Q_W)$ (written as `map (inl Q₁ Q₂)`) and $\mathcal{G}(W, Q_W) \to \mathcal{G}(W \oplus W, Q_V \oplus Q_W)$ (written as `map (inr Q₁ Q₂)`). We then must show these maps satisfy the commutativity requirement of theorem 10.28, and pay what the author can only assume is the price for not having expressed this requirement in a point-free manner[22], via a comparatively grueling 40 line proof, unaided by `ext` and subjected to four fiddly dependent inductions. The statement of this result is:

```
theorem map_mul_map_of_isOrtho_of_mem_evenOdd
    (f₁ : Q₁ →qᵢ Qₙ) (f₂ : Q₂ →qᵢ Qₙ) (hf : ∀ x y, Qₙ.IsOrtho (f₁ x) (f₂ y))
    (m₁ : CliffordAlgebra Q₁) (m₂ : CliffordAlgebra Q₂)
    {i₁ i₂ : ZMod 2} (hm₁ : m₁ ∈ evenOdd Q₁ i₁) (hm₂ : m₂ ∈ evenOdd Q₂ i₂) :
    map f₁ m₁ * map f₂ m₂ = (-1 : ℤˣ) ^ (i₂ * i₁) • (map f₂ m₂ * map f₁ m₁) := by
```

which for our particular case we instantiate with `f₁ := inl Q₁ Q₂` and `f₂ := inl Q₁ Q₂`.

The full proofs are left to [*mathlib4#7644*].

10.4. Summary

This chapter has demonstrated repeatedly, through examples around Clifford algebras, how to use universal properties to construct isomorphisms. While the isomorphisms themselves are far from novel, the approach of constructing them via explicit reference to universal properties is at the very least unusual.

The author is aware of no prior formalizations that cover the content in section 10.1 (let alone future sections of this chapter); [72] concerns itself only with $\mathcal{G}(\mathbb{R}^3)$ so none of the isomorphisms in this chapter can even be stated, [75] constructs a map to the quaternions but is silent about its properties, [74] makes no reference to quaternions at all, and the author could not find any mention of Clifford algebras in Isabelle's "Archive of Formal Proofs". One of the benefits of mathlib is that by having all the formalizations in one place, there is a natural place to contribute links between formalized objects such as these.

The universal properties throughout this chapter are themselves likely also not novel, as while the author was unable to find some of them in literature, their statements are obvious. Theorem 10.14 is seemingly an exception to this, as other formalizations of dual numbers (like

[21] Made more obtuse by the fact the codomain involves the sub-types `evenOdd Qᵢ`.
[22] Which could perhaps be done via theorem 10.26, or by developing the theory of morphisms of graded algebras.

the ones used to build the dual quaternions in the Coq formalization associated with [83]) are not generalized to the trivial square zero extension.

The results in section 10.2 provide a demonstration of the indispensability of the typeclass infrastructure from chapter 4, and also represent broad improvements to the results about mathlib's `tensor_product` by the author.

Finally, the results in section 10.3 provide a demonstration that the infrastructure in chapter 6 is usable in practice. While its results are once again not novel, the use of universal properties to provide an explicit construction is. One consequence of this approach is that theorem 10.25 is stated very generally—we did not assume that our base ring was a field, nor did we even assume that $2 : R$ was invertible—and so our results apply even to the pathological cases explored in section 9.7.

11

Further formalizations

> The numbers formed up and marched past his brain in terrified obedience. Division and multiplication were discovered. Algebra was invented and provided an interesting diversion for a minute or two. And then he felt the fog of numbers drift away, and looked up and saw the sparkling, distant mountains of calculus.
>
> *(Terry Pratchett)*

The author is conscious that Clifford algebras are of relatively niche interest in the space of formal mathematics. In some ways, the benefit of formalizing results about them was as much in the journey (formalizing the fundamental tools in part II and many unremarkable yet essential theorems not mentioned in this thesis) as it was in these destinations. Some of the author's journeys never reached their destination; in some cases because a shorter route was found, and in others, simply because the destination was too far away. These incomplete journeys would not make sense as part of chapter 9 or chapter 10, but some of them are still interesting in their own right.

This chapter presents one of each case; section 11.1 contains a fruitful exploration into alternating maps that proved too indirect to reach the wedge product, while section 11.2 explores formalizations about $\exp(x)$ on objects adjacent to the Clifford algebra. While not useful to the author's results in Clifford algebras, they have already proved useful to other users of mathlib.

11.1. Alternating maps

Before the approach in section 9.3.5 was taken to implement the wedge product, some wrong turns were explored. In particular, one way to define $a \wedge b \wedge c$ (on pure vectors) is as $\frac{1}{3!}(abc - acb + cab - cba + bca - bac)$; that is, by computing the "alternatization" of the geometric product. In general, for a product of n vectors, the scale factor is $\frac{1}{n!}$, and the sign for each term corresponds to the sign of the permutation of the vectors. As remarked in [70, §1], this approach is undesirable

from the perspective of generality, as it requires[1] that our base ring is divisible by all $n!$.

Of course, the fact that this wasn't helpful for the definition of the wedge product does not mean it cannot be formalized, and indeed formalizing it allows future users of mathlib to prove that the two approaches are equivalent. The general construction worth extracting here is that of computing the alternatization of multilinear maps; which in turn, makes it desirable to have a formal definition of alternating maps.

The formalization of `AlternatingMap` in mathlib originates from [*mathlib#5102*], which the author revived and adapted from an earlier version by Zhangir Azerbayev, and is as follows:

```
/-- An alternating map is a multilinear map that vanishes when two of its arguments are equal. -/
structure AlternatingMap
    (R M N ι : Type*) [Semiring R] [AddCommMonoid M] [Module R M] [AddCommMonoid N] [Module R N]
    extends MultilinearMap R (fun _ : ι => M) N where
  /-- The map is alternating: if `v` has two equal coordinates, then `f v = 0`. -/
  map_eq_zero_of_eq' (v : ι → M) (i j : ι) : v i = v j → i ≠ j → toFun v = 0
```

An immediate application of this definition is to prove that the exterior product (that is, the `*` operator for `ExteriorAlgebra R M`) is alternating, which can be stated as

```
/-- The product of `n` terms of the form `ι R m` is an alternating map. -/
def ιMulti (n : ℕ) : AlternatingMap R M (ExteriorAlgebra R M) (Fin n) :=
```

The construction of the map is trivial, though the proof that it is alternating [*mathlib#5124*] requires some slightly fiddly induction.

As for the `alternatization` of a `MultilinearMap`, it can be defined simply as `∑ σ : Perm ι, σ.sign • m.domDomCongr σ`, where `σ.sign` is the sign of a permutation (and predates the author's contributions to mathlib), and `m.domDomCongr σ` from [*mathlib#5136*] is the multilinear map `m` with its arguments permuted by `σ`. It is straightforward to prove that this alternatization satisfies `map_eq_zero_of_eq` and therefore is alternating.

A somewhat unexpected benefit of this work is that it generalized many results about `Matrix.det` (as the determinant is an alternating map in the rows or columns of a matrix), and allowed many existing proofs to be simplified down to a single application of a lemma about alternating maps [*mathlib#6708*]. This is another case where writing things in a point-free style (section 5.3) pays dividends; but as usual, at the cost of locally making things harder to read. The definitions before and after this refactor are shown in listing 11.1.

11.1.1. Products of alternating maps

One more particularly interesting formalization about alternating maps is [*mathlib#5269*], which formalizes a product on alternating maps. This is defined in [84, Proposition 22.24] as

$$(f \wedge g)(u_1, \dots, u_{m+n}) = \sum_{\text{shuffle}(m,n)} \operatorname{sign}(\sigma) f(u_{\sigma(1)}, \dots, u_{\sigma(m)}) g(u_{\sigma(m+1)}, \dots, u_{\sigma(m+n)}), \quad (11.1)$$

[1] Recall that in section 9.3.5, we needed only that 2 be invertible; and as described in section 9.7, even this is stronger than necessary when a bilinear form is available.

```
definition det (M : matrix n n R) : R :=
∑ σ : perm n, σ.sign • ∏ i, M (σ i) i

theorem det_zero_of_row_eq
  (i_ne_j : i ≠ j) (hij : M i = M j) : M.det = 0 :=
sorry -- 12 lines of proof follow
```

```
/-- An `alternating_map` in the rows of the matrix. -/
def det_row_multilinear : alternating_map R (n → R) R n :=
((multilinear_map.mk_pi_algebra R n R).comp_linear_map
 (linear_map.proj)).alternatization

abbreviation det (M : matrix n n R) : R :=
  det_row_multilinear M

lemma det_apply (M : matrix n n R) :
  M.det = ∑ σ : perm n, σ.sign • ∏ i, M (σ i) i :=
multilinear_map.alternatization_apply _ M

theorem det_zero_of_row_eq
  (i_ne_j : i ≠ j) (hij : M i = M j) : M.det = 0 :=
det_row_multilinear.map_eq_zero_of_eq M hij i_ne_j
```

Listing 11.1.: Refactoring `matrix.det` to use `alternatization`

The old code is on the left, and the new code on the right. While the new definition is frankly inscrutable, the power of formal languages is that we can prove to the reader that it is defined correctly (`det_apply`). The new definition makes the long proof of `det_zero_of_row_eq` (and many other results) evaporate.

where shuffle(m, n) consists of all permutations of $[1, m+n]$ such that $\sigma(1) < \cdots < \sigma(m)$ and $\sigma(m+1) < \cdots < \sigma(m+n)$; but this assumes that the indices of the maps are `Fin m` and `Fin n`. The Lean formalization in [*mathlib#5269*] states things more generally, replacing these with arbitrary (finite) indexing types `ιa` and `ιb`.

```
def dom_coprod
  (a : alternating_map R' Mᵢ N₁ ιa) (b : alternating_map R' Mᵢ N₂ ιb) :
  alternating_map R' Mᵢ (N₁ ⊗[R'] N₂) (ιa ⊕ ιb) :=
```

Instead of using the $<$ relation to define shuffle, which is not available on our indices, the formalization works with the type of permutations of `ιa ⊕ ιb`, under the quotient by the relation "differs only by permutations within `ιa` and `ιb`". This `dom_coprod` operation is interesting because it satisfies the following:

```
lemma multilinear_map.dom_coprod_alternization
  (a : multilinear_map R' (λ _ : ιa, Mᵢ) N₁)
  (b : multilinear_map R' (λ _ : ιb, Mᵢ) N₂) :
  (a.dom_coprod b).alternatization =
    a.alternatization.dom_coprod b.alternatization :=
```

where the first `dom_coprod` is the map $v \mapsto a(v_1, \ldots, v_n) \otimes b(v_{n+1}, \ldots, v_{n+m})$.

11.1.2. Further links with the exterior algebra

As a final remark about alternating maps, [*mathlib#14803*] constructs the isomorphism

```
def lift_alternating_equiv :
  (Π i, alternating_map R M N (fin i)) ≃ₗ[R] (exterior_algebra R M →ₗ[R] N) :=
sorry
```

which reads[2] "Choosing an i-argument alternating map for every $i : \mathbb{N}$ is equivalent to choosing a linear map from the exterior algebra". The interpretation is reasonable; we can build a map from the exterior algebra by specifying how the map operates on the blades. As a result, the construction of the reverse map is straightforward.

The construction of the forward map leverages some of the tricks from chapter 8. Writing $\mathrm{Alt}(M^i, N)$ to stand in for `alternating_map R M N (fin i)`, we build the map via an auxiliary construction exploiting the algebra structure of endomorphisms:

$$\mathrm{lift\_alt}_{\mathrm{aux}} : \bigwedge(M) \to ((i : \mathbb{N}) \to \mathrm{Alt}(M^i, N)) \to ((i : \mathbb{N}) \to \mathrm{Alt}(M^i, N)) \tag{11.2}$$

$$\mathrm{lift\_alt}_{\mathrm{aux}}(m : M) := f \mapsto i \mapsto f_{i+1}(m, \dots) \tag{11.2a}$$

Here, $f_{i+1}(m, \dots)$ is the i-argument alternating map formed by fixing the first argument of f_i to m. Effectively, our recursion is applying our functions f_i one grade at a time, and putting the "final result" in f_0; each time we encounter a new vector, we throw out this result (it wasn't final after all!) and remove one argument from all our functions. This informs us how to extract the operator we want:

$$\mathrm{lift\_alt}[f] : ((i : \mathbb{N}) \to \mathrm{Alt}(M^i, N)) \to \bigwedge(M) \to N \tag{11.3}$$

$$\mathrm{lift\_alt}[f] := f \mapsto x \mapsto g_0() \text{ where } g := \mathrm{lift\_alt}_{\mathrm{aux}}(x, f), \tag{11.3a}$$

where the $()$ on g_0 represents the operation of applying a zero-argument alternating map, exploiting the fact that $\mathrm{Alt}(M^0, N)$ and N are isomorphic. For eq. (11.2a) to be well-formed, the universal property of the exterior algebra requires that when $(f \mapsto i \mapsto f_{i+1}(m, \dots))$ is applied to itself with the same m, all the maps vanish. This can be seen easily; applying it twice produces a term of the form $f_{i+2}(m, m, \dots)$, and alternating maps vanish when two of their arguments are equal.

Of course, when M is n-dimensional, we need only pick the maps where $i \leq n$; but formalizing results about Clifford or exterior algebras that only hold over finite-dimensional modules is a project in itself.

11.2. Exponential operators

One of the author's original goals of formalizing Clifford algebras was to be able to formalize results about the multivector exponential operator (shown briefly in table 2.4), which in applications [63, §2.3; 5, eq. (2.120); 1, §7.4.3] typically appears when turning bivectors into rotors to perform

[2] The Π in this syntax is indicating a dependent function; each element of the family of alternating maps is of a different arity.

geometric transformations. The exp operator on multivectors can be defined in much the same way as it is for $\mathbb{R}$ and $\mathbb{C}$, via a power series expansion [68, Chapter I, eq. (2.6)],

$$\exp(x) = \sum_{n=0}^{\infty} \frac{1}{n!} x^n. \tag{11.4}$$

In formalization, we don't just want to define it the "same way", because then we have to rewrite all the proofs in the "same way" as well. Instead, we want a single definition of `exp` that works for as many cases as possible. A candidate for this is the exponential of Banach algebras, introduced by Anatole Dedecker in [mathlib#8576] as

```
variables (𝕂 𝔸 : Type*) [nondiscrete_normed_field 𝕂] [normed_ring 𝔸] [normed_algebra 𝕂 𝔸]
/-- In a Banach algebra `𝔸` over a normed field `𝕂`, `exp 𝕂 𝔸 : 𝔸 → 𝔸` is the exponential map
determined by the action of `𝕂` on `𝔸`. -/
def exp (x : 𝔸) : 𝔸 :=
```

This definition has two main problems.

The first is that it takes a surprising `𝕂` argument, which specifies the field in which to compute the fractions $\frac{1}{n!}$ from eq. (11.4). Mathematically this is effectively redundant, as provided we limit ourselves to `[char_zero 𝕂]` (without which the exponential is not well-defined anyway), we can safely pick `𝕂 := ℚ`. However, it turns out to sometimes be convenient in formalization, as lemmas about `exp` often require a richer choice of field over which `𝔸` is an algebra, such as `𝕂 := ℝ` or `𝕂 := ℂ`; by putting this choice of field in the definition of `exp`, the lemmas can locate it automatically rather than asking the user to choose each time. In the author's opinion this convenience isn't worth the surprise of having an explicit `𝕂` in theorem statements, but attempts to remove the argument in [*mathlib#19244*] and [*mathlib4#8370*] proved awkward.

At any rate, it is the second problem that is particularly obstructive towards the goal of exponentiating multivectors; namely the `[normed_ring 𝔸]` argument. This not only says that `𝔸` must have a submultiplicative norm ($\|AB\| \leq \|A\|\|B\|$, which is a reasonable condition to ensure convergence), but by virtue of the data-carrying definition of `normed_ring` and the use of `[]` around the argument indicating it should be found by typeclass search, also says that the norm ought to be *canonical*. Even if we fight this canonicity requirement, to Lean each choice of norm defines a new `exp` function; that is `@exp 𝕂 𝔸 _ norm1 _` and `@exp 𝕂 𝔸 _ norm2 _` are a priori unrelated functions.

The fact that we need a norm at all in Lean to define eq. (11.4) should be considered suspect; mathlib can write down infinite sums (of convergent sequences) using `tsum`, which for eq. (11.4) results in `∑' n : ℕ, (n !⁻¹ : ℚ) • x ^ n`. To make proofs easier, [mathlib#8576] chose not to use this definition, but to use a more complex definition that leverages existing theory about formal multilinear series. Thankfully, by refactoring formal multilinear series in [*mathlib#13444*; *mathlib#13426*], it was possible to keep these easier proofs while taking on only the weaker typeclass requirements needed by `tsum`.

The end result, after some further simplification in [*mathlib#13987*], was

```
variables (𝕂 : Type*) {𝔸 : Type*}
variables [field 𝕂] [ring 𝔸] [algebra 𝕂 𝔸] [topological_space 𝔸] [topological_ring 𝔸]

/-- `exp 𝕂 : 𝔸 → 𝔸` is the exponential map determined by the action of `𝕂` on `𝔸`. -/
def exp (x : 𝔸) : 𝔸 :=
```

which needs only a suitable topology on the ring `𝔸` (one where addition, multiplication, and negation[3] are continuous). Note that the conditions here are not strong enough to ensure convergence, but that is fine for the same reason that in Lean, the division operator does not require its divisor to be non-zero [37]; those conditions are left to the `theorem`s about `exp`.

Rather than proceeding by defining a canonical topology on the multivectors, which the author struggled to find much literature on, we will instead look at some simpler cases, starting with exponents of matrices[4].

11.2.1. Matrices

Matrices (of finite dimension) have a natural canonical topology, equal to the product or box topology which are one and the same. As a result, the changes to `exp` above mean that we can write `exp 𝕂 M` for a square matrix `M` and Lean is happy.

However, we run into trouble when we try to apply a theorem about `exp`, for instance

```
theorem exp_add_of_commute
  {𝕂 𝔸 : Type*} [is_R_or_C 𝕂] [normed_ring 𝔸] [normed_algebra 𝕂 𝔸] [complete_space 𝔸]
  {x y : 𝔸} (hxy : commute x y) :
  exp 𝕂 (x + y) = exp 𝕂 x * exp 𝕂 y
```

with `𝔸 := matrix n n 𝕂`, as the requirement for a canonical submultiplicative norm on matrices resurfaces.

Ignoring the issue of canonicity for a moment, we clearly still need a formalization of *some* matrix norm to make progress. The easiest norm to formalize is the element-wise infinity norm,

$$\|M\| = \min_{i,j} \|M_{i,j}\|, \tag{11.5}$$

which was added in [mathlib#9379] simply by copying across the norm inherited from product types (whose elements are themselves product types). However, this norm is insufficient for our use-case, as it is not submultiplicative.

A somewhat more useful norm is the Frobenius norm,

$$\|M\|_F = \sqrt{\sum_{i,j} \|M_{i,j}\|^2}, \tag{11.6}$$

added in [*mathlib#13497*]. This too follows easily by copying an existing norm structure; the

[3] Strictly not mathematically necessary here.
[4] Which happens to be one of the targets of the project to formalize an undergraduate mathematics degree in mathlib.

ℓ^2 norm (of the sequence of rows, each of which is itself viewed under the ℓ^2 norm). Unlike eq. (11.5), eq. (11.6) *is* submultiplicative, via Cauchy–Schwarz. In practice, this means that we are restricted to the case when the entries $M_{i,j}$ lie in $\mathbb{R}$ or $\mathbb{C}$, as most of the results we need to prove this are stated in the generality of `is_R_or_C`.

The last norm we should consider is the ℓ^∞ norm,

$$\|M\|_\infty = \sup_i \left(\sum_j \|M_{ij}\| \right), \tag{11.7}$$

added in [*mathlib#13518*]. For the third time in a row, we are in luck; we can build this norm by taking the ℓ^∞ norm of the sequence of rows, each equipped with the ℓ^1 norm. This norm is also submultiplicative; but turns out to be so much more generally than eq. (11.6), as this time the entries $M_{i,j}$ can lie in any normed algebra over a field. This means that it remains valid for matrices of quaternions! Under stronger assumptions on these entries [*mathlib4#9476*] this coincides with the operator norm, $\|M\|_\infty = \sup_{\|x\|_\infty=1}(\|Mx\|_\infty)$.

While both eq. (11.6) and eq. (11.7) are submultiplicative and therefore satisfy `normed_ring`, we also need them to satisfy `normed_algebra`. It was discovered that the definition of `normed_algebra` in mathlib did not permit eq. (11.6) and eq. (11.7) in the special case of 0×0 matrices, due to an erroneous axiom that implied $\|1\| = 1$. This was removed in [*mathlib#13544*], and replaced with the axioms $\|rx\| \le \|r\|\|x\|$; the definition became the simpler "a normed algebra is a normed ring that is also a normed module".

While the infinity norm in eq. (11.7) is certainly the most convenient for us, this isn't sufficient evidence that it is canonical, which would make it the norm that every mathlib user gets automatically. If it `were` canonical, then we would already be done; users of `exp` on matrices could directly use `exp_add_of_commute`.

The fact it is not simply means that we have to duplicate the API:

```
variable {𝕂 m 𝔸 : Type*}
variable [is_R_or_C 𝕂] [fintype m] [decidable_eq m] [normed_ring 𝔸] [normed_algebra 𝕂 𝔸] [complete_
↪  space 𝔸]

lemma matrix.exp_add_of_commute (A B : matrix m m 𝔸) (h : commute A B) :
  exp 𝕂 (A + B) = exp 𝕂 A · exp 𝕂 B :=
begin
  letI : semi_normed_ring (matrix m m 𝔸) := matrix.linfty_op_semi_normed_ring,
  letI : normed_ring (matrix m m 𝔸)      := matrix.linfty_op_normed_ring,
  letI : normed_algebra 𝕂 (matrix m m 𝔸) := matrix.linfty_op_normed_algebra,
  exact exp_add_of_commute h,
end
```

Note here that while we do assume a canonical norm on the *elements* of the matrices `𝔸`, we

do not do so for the matrix itself[5]. Inside the proof, we use the `letI` tactic to locally pretend that the declarations about eq. (11.7) are `instance`s; globally, they are just `def`s and would not therefore not be found in typeclass search. Under this pretence, the `exp_add_of_commute` lemma that previously eluded us is happy to apply.

Overall, this work [*mathlib#13520*] (and its dependencies [*mathlib#13402*; *mathlib#13444*; *mathlib#13488*; *mathlib#13489*; *mathlib#13518*; *mathlib#13534*; *mathlib#13641*; *mathlib#13815*; *mathlib#13918*; *mathlib#13938*; *mathlib#13970*; *mathlib#13971*]) contributed the following facts about the matrix exponential to mathlib (which in turn improved the state in fig. 3.9 of formalizing [38]):

$$\exp(0) = 1 \tag{11.8}$$

$$\exp(A + B) = \exp(A)\exp(B), \qquad \text{when } A \text{ and } B \text{ commute} \tag{11.9}$$

$$\exp(nA) = \exp(A)^n, \qquad \text{when } n : \mathbb{N} \text{ or } n : \mathbb{Z} \tag{11.10}$$

$$\exp(-A) = \exp(A)^{-1} \tag{11.11}$$

$$\exp(UDU^{-1}) = U\exp(D)U^{-1} \tag{11.12}$$

$$A\exp(B) = \exp(B)A, \qquad \text{when } A \text{ and } B \text{ commute} \tag{11.13}$$

$$\exp\left(\begin{bmatrix} A_1 & & \\ & \ddots & \\ & & A_n \end{bmatrix}\right) = \begin{bmatrix} \exp(A_1) & & \\ & \ddots & \\ & & \exp(A_n) \end{bmatrix}, \qquad \text{where } A_i \text{ are either scalars or matrices} \tag{11.14}$$

$$\exp(A^{\mathsf{H}}) = \exp(A)^{\mathsf{H}} \tag{11.15}$$

$$\exp(A^{\mathsf{T}}) = \exp(A)^{\mathsf{T}} \tag{11.16}$$

Prior to the author's work, only the first two already existed, for general Banach algebras. Except for the last two which only makes sense on matrices, the remainder were contributed in general cases first, then creating duplicates specialized to matrices. This duplication is still not ideal, but duplicating every lemma about `exp` for matrices is still much better than duplicating every lemma *for every possible norm* on matrices!

Having practiced on the matrices in section 11.2.1, we can take a small step closer to Clifford algebras by looking at exponentials of dual numbers (section 10.1.3) and quaternions (section 10.1.4), which are at least isomorphic to special cases of Clifford algebras.

11.2.2. Dual numbers

For the dual numbers—or more generally, the trivial square-zero extension $\mathrm{tsze}[R, M]$ that we saw in section 10.1.5—the exponential function is trivial so long as R is commutative; we have $\exp(r + m) = \exp(r) + \exp(r)m$ (and so $\exp(\epsilon) = 1 + \epsilon$). The proof follows by expanding the

[5] This still isn't an ideal situation, as we are still faced with an issue when working with matrices themselves have no canonical choice of norm. A possibly better approach would be to have something like a `submultiplicative_normable` typeclass in mathlib, which would state that the topology can be obtained from a suitable norm, without prescribing which norm is used. This solution entails a fair amount of boilerplate, and the scenario it solves is not one the author needed; so the author has not attempted it.

power series, and noting that when R is commutative[6], $(r+m)^n = r^n + nmr^{n-1}$ (formalized in [*mathlib#18199*], along with other basic algebraic results). Formalizing the result about exp takes a little more work; we first need to set up the topological space structure on $\mathrm{tsze}[R, M]$ (copied from the product topology on its components), and prove a large number of obvious results about continuity of the constructor, projections, and algebraic operators.

Once these are in place, we can prove $\exp(r+m) = \exp(r) + \exp(r)m$. We start by proving results about elements of the power series; we do so separately for the R and M pieces as the latter needs stronger assumptions. Before we can state the theorems, we will need a fairly monstrous set of typeclass assumptions

```
variable [Field 𝕜] [CharZero 𝕜] [Ring R] [AddCommGroup M]
         [Algebra 𝕜 R] [Module 𝕜 M] [Module R M] [Module Rᵐᵒᵖ M]
         [SMulCommClass R Rᵐᵒᵖ M] [IsScalarTower 𝕜 R M] [IsScalarTower 𝕜 Rᵐᵒᵖ M]
         [TopologicalSpace R] [TopologicalSpace M]
         [TopologicalRing R] [TopologicalAddGroup M] [ContinuousSMul R M] [ContinuousSMul Rᵐᵒᵖ M]
```

In summary: we have an additive group `M` which is a left- and right- `R`-module as well as a `𝕜`-module, with all the actions being maximally compatible. The next two lines assure us that all these operations are continuous. As a compromise between the easy commutative case and hard non-commutative case (that we earlier dismissed to a footnote), we prove the result for the special case of the non-commutative case where the particular r and m commute, $rm = mr$, as specified by the `hx` assumption below:

```
@[simp] theorem fst_expSeries (x : tsze R M) (n : ℕ) :
    (expSeries 𝕜 (tsze R M) n fun _ => x).fst = expSeries 𝕜 R n fun _ => x.fst := by
  simp [expSeries_apply_eq]

theorem snd_expSeries_of_smul_comm
    (x : tsze R M) (hx : MulOpposite.op x.fst • x.snd = x.fst • x.snd) (n : ℕ) :
    (expSeries 𝕜 (tsze R M) (n + 1) fun _ => x).snd = (expSeries 𝕜 R n fun _ => x.fst) • x.snd
sorry -- this one is boring, but more slightly work
```

Here, `x.fst` and `x.snd` refer to the r and m pieces of $r+m$.

Working with the R part of the series is easy, as we can see that the R part of the nth term of $\exp(r+m)$ is the nth term of $\exp(r)$. For the M part we need to do slightly more work, as the indices of the terms have shifted by one; we need to state that the *sums* of the series agree

```
theorem hasSum_snd_expSeries_of_smul_comm (x : tsze R M)
  (hx : MulOpposite.op x.fst • x.snd = x.fst • x.snd) {e : R}
  (h : HasSum (fun n => expSeries 𝕜 R n fun _ => x.fst) e) :
  HasSum (fun n => snd (expSeries 𝕜 (tsze R M) n fun _ => x)) (e • x.snd)
```

The `HasSum f a` definition in `mathlib` can be rather confusing at first glance; it doesn't mean "the

[6] When R is not commutative, we have $(r+m)^{n+1} = r^{n+1} + \sum_{i=0}^{n} r^{n-i} m r^i$. In Lean, the summand is written `r ^ (n - i) • op (r ^ i) • m`, once again using the infrastructure in section 4.7, this time to perform the scaling on both sides of `m`. A similar result holds for exp; that $\exp(r+m) = \exp(r) + \int_0^1 \exp(tr)m\exp((1-t)r)\mathrm{d}t$. The author found this somewhat awkward to formalize via the usual analysis proofs [*mathlib#19056*], but the combinatorial proof [*mathlib4#9487*] from [85, §2.1] might be more approachable.

sum of `f` *is* `a`", but rather[7] "the sum of `f` converges to the neighborhood of `a`". This design lets us simultaneously prove both *that* a series converges and *what* it converges to. Here, we resolve difficulties around whether $\exp(r+m)$ converges by ensuring $\exp(r)$ converges. We can combine our results for the R part and the M part as

```
/-- If `exp R x.fst` converges to `e` then `exp R x` converges to `inl e + inr (e • x.snd)`. -/
theorem hasSum_expSeries_of_smul_comm
    (x : tsze R M) (hx : MulOpposite.op x.fst • x.snd = x.fst • x.snd)
    {e : R} (h : HasSum (fun n => expSeries 𝕂 R n fun _ => x.fst) e) :
    HasSum (fun n => expSeries 𝕂 (tsze R M) n fun _ => x) (inl e + inr (e • x.snd)) := by
  have : HasSum (fun n => fst (expSeries 𝕂 (tsze R M) n fun _ => x)) e := by
    simpa [fst_expSeries] using h
  simpa only [inl_fst_add_inr_snd_eq] using
    (hasSum_inl _ <| this).add (hasSum_inr _ <| hasSum_snd_expSeries_of_smul_comm 𝕂 x hx h)
```

where `inl r + inr m` is the formal way we write $r+m$.

To write useful lemmas about `exp 𝕂 x`, it would be reasonable to assume we need to know that the series converges for a given `x`. One option would be to promoting our topological spaces to complete normed Hausdorff (`t2_space`) spaces over the real or complex numbers (and indeed this is what [*mathlib#18200*] did). However, we can exploit the "junk value" design (described in [37]) here; if the series does *not* converge at `x`, then we define `exp 𝕂 x = 0`. This is perfect for our use case, because we can case-split on whether $\exp(r)$ converges: if it does, then we have $\exp(r+m) = \exp(r) + \exp(r)m$ via our results above; if it doesn't, then we know via `fst_expSeries` that $\exp(r+m)$ must also not converge, and so our equation becomes $0 = 0 + 0m$ which is still true!

These results were originally contributed in [*mathlib#18200*], then revised in [*mathlib#19049*] for (partial) non-commutativity considerations, and again in [*mathlib4#9491*] to exploit the use of "junk values".

Unlike in section 11.2.1, we have not yet provided any norm structure on the dual numbers; and so the `exp_add_of_commute` lemma remains out of our grasp. The canonical "norm" on the dual numbers appears to be $\|a + b\epsilon\| = \|a\|$ [86, pg. 292], but this norm has two major flaws that prevent our use of it. The first is that this is not a true norm, but only a seminorm (as there are non-zero elements like ϵ with norm zero); this means that they induce a non-Hausdorff topology, and our exponential series does not converge to a single value[8]! The second is that the topology induced by this seminorm is not equal to the product topology, which is the topology we provided all the theorems for in [*mathlib#18200*]. A better candidate for a norm is $\|r+m\| = \|r\| + \|m\|$, which is submultiplicative, and agrees with the product topology [*mathlib4#9492*]; though it is hard to say whether it is the most canonical[9].

[7] These two notions agree in Hausdorff spaces, written in our case as `[T2Space R] [T2Space M]`.
[8] Indeed, only the real part converges.
[9] Jireh Loreaux proposed the $C^\star$ norm in https://leanprover.zulipchat.com/#narrow/stream/116395-maths/topic/Exponentials.20in.20seminormed.20algebras/near/321870256, in the context of the unitization; a very similar construction to the dual numbers.

11.2.3. Quaternions

For the quaternions, we are in for a treat; mathlib has long-known [mathlib#2339] that they form a normed algebra[10] (with the norm defined by $\|w + x\mathrm{i} + y\mathrm{i} + z\mathrm{i}\| = \sqrt{w^2 + x^2 + y^2 + z^2}$). As a result, `exp ℝ q` works out of the box for a quaternion `q : ℍ[ℝ]`.

The result that we wish to show can be stated in Lean as

```
/-- The closed form for the quaternion exponential on arbitrary quaternions. -/
theorem exp_eq (q : ℍ[ℝ]) :
    exp ℝ q = exp ℝ q.re • (↑(cos ‖q.im‖) + (sin ‖q.im‖ / ‖q.im‖) • q.im)
```

where `q.im` is `q - q.re`, i.e. the pure-imaginary part of the quaternion. The proof is a standard on (so we shall not spend much space on it here), and proceeds by considering the exponential of pure-imaginary quaternions (eliminating the `exp ℝ q.re` term), then collecting alternate terms into terms of the series for cos and sin. Formalizing this hit the slight snag that mathlib did not know what these series were, but with some help from the community this was resolved in [*mathlib#18352*]. The result above, along with various other corollaries like $\|\exp(q)\| = \exp(\Re(q))$ and $\Re(\exp(q)) = \exp(\Re(q))\cos(\|\Im(q)\|)$, were contributed in [*mathlib#18349*]; which in turn required the many basic algebraic results about quaternions in [*mathlib#18413*].

11.3. Summary

Section 11.1 presented a brief summary of the definition of alternating maps in mathlib, and a handful of useful constructions related to them. Notably, it provided another application for the insight gained in chapter 8 regarding universal properties and recursive algorithms. The work here is already being built upon by other contributors; Yury Kudryashov added a definition of *continuous* alternating maps in [mathlib4#5678], and has indicated that the work in section 11.1.1 will be essential to defining products of differential forms.

Section 11.2 outlined the process of ironing out some of the difficulties in working with `exp` in mathlib, and demonstrated how to apply it to the quaternions and the dual numbers. Further work could go on to formalize the results about dual quaternions in [87], and to finish the author's attempt in [*mathlib4#9487*] to formalize the results for non-commutative dual numbers from [85, §2.1].

The issue of canonicity of norms that section 11.2 runs into is one that extends beyond exponential operators; mathlib has the same problem for derivatives, and thus cannot talk about "the" derivative of a matrix (or multivector) function without first specifying a canonical norm. Derivatives are rather more developed in mathlib than exponential operators, and so this is a much thornier issue to resolve; but one that the community is already working on [88].

Of course, the underlying motivation for section 11.2 was the *multivector* exponential; which hinges upon putting at least a topology, if not also a norm, on the Clifford algebra. For now, this

[10] Though the author had to upgrade this to *complete* normed algebra [*mathlib#18347*], i.e. a Banach algebra.

is almost certainly out of reach for mathlib, which is lacking such constructions even on the much more elementary tensor product.

There is a common thread between 11.1 and section 11.2, one which weaves through much of the rest of this thesis; the benefit of the collaborative nature of mathlib. Work by one contributor, even when falling short of its original goal, can be stored, maintained, and indexed as part of mathlib, such that it is at any point immediately ready for a new contributor to discover and resume. These small formalizations add up over time, sometimes quickly: the author had multiple experiences where a partial result he contributed one week was asked for by another user the following week.

12

Conclusions

> If I have not seen as far as others, it is because there were giants standing on my shoulders.
>
> *(Hal Abelson)*

12.1. Key contributions

- Improvements across mathlib's `Matrix` library, in an effort to improve parity with *The Matrix Cookbook*. The most visible contribution here is the new `!![a, b; c, d]` notation for matrices. While not of any use elsewhere in this thesis, this part of mathlib may well be of pedagogical value for new Lean users.
- Significant developments to mathlib's theory of scalar actions, resulting in mathlib gaining non-associative algebra and bimodules, in chapter 4. It also provides an in-depth exploration of typeclass diamonds with examples, supplementing the work in [45] and [44].
- Considerable expansions to the scope of the pre-existing `ext` tactic used in mathlib, along with illumination of the chaining mechanism, in chapter 5.
- A formalization of graded rings that is the first of its kind in any theorem prover, in chapter 6; and a demonstration that it is fit for purpose with multiple examples. This chapter is also responsible for the `SetLike` typeclass in mathlib, which set the stage for many future refactors to algebraic structures.
- A better understanding of the typeclass issues that plagued mathlib during the port from Lean 3 to Lean 4, in chapter 7. The learnings are relevant for theorem provers beyond Lean, though other systems have been fortunate enough to sidestep the issues by coincidence.
- A mental framework for working with universal properties through the lens of functional programming, culminating in the development of a basis-free universal property for the even subalgebra of the Clifford algebra, $\mathcal{G}^+(V, Q)$, in chapter 8.

- A very general formalization of Clifford algebras (matching Bourbaki) that now resides in mathlib, from chapter 9.
- A broad selection of formalized connections between Clifford algebras and other mathematical objects, in chapter 10. This represented significant contributions to mathlib's linear algebra library, including to the trivial square zero extension over non-commutative rings, the tensor products of quadratic forms, and the tensor product of graded rings.
- Various smaller contributions to mathlib, for which chapter 11 presents a few examples:
  - A formalization of alternating maps, which will be essential in future developments of differential forms. Already, this has provided simplifications in the library around matrix determinants.
  - Improvements to the definition of the `exp` operator, and a library of lemmas around it specific to matrices, dual numbers, and quaternions. As part of this, a small selection of matrix norms were added to mathlib.
  - Over 2000 total contributions[1] to the Lean 3 and Lean 4 versions of mathlib, of which only a small minority are directly referenced by this thesis. The author was additionally involved in review of over 5000 contributions from other mathlib contributors.

12.2. Follow-up work

Much of the work in this thesis has been contributed to mathlib on the fly; and so the work of other contributors is already building upon it. This section lists a selection of examples that the author is aware of.

- Sophie Morel has been developing a theory of Grassmannians at `https://github.com/smorel394/ExteriorPowers`, building upon the author's work on `AlternatingMap` (section 11.1) and `ExteriorAlgebra`. This work is already starting to enter mathlib in [mathlib4#9718].
- Jujian Zhang, who co-authored the paper from which chapter 6 is adapted, has further developed the "Proj construction" in [89], which builds heavily upon the author's work on graded rings.
- Jireh Loreaux has expanded mathlib's library of matrix norms described in section 11.2.1, adding the ℓ^2 norm in [mathlib4#9474].
- Ali Ramsey and Kevin Buzzard, through their work on formalizing Hopf algebras [mathlib4#10079], have at the author's suggestion been leveraging much of the `ext` infrastructure (especially on tensor products) set up in chapter 5.

[1] Readers who are logged into GitHub can, at the time of writing, see the full list through this search for `is:pr author:eric-wieser repo:leanprover-community/mathlib repo:leanprover-community/mathlib4 "merged by bors" created:<=2024-02-22`.

- Anne Baanen adopted the author's `SetLike` and early ideas about a corresponding `FunLike` into a major refactor of algebraic structures throughout mathlib, as described in [45, §6.3].

12.3. Future directions

12.3.1. Further changes to scalar actions

Chapter 4 leaves two main directions for further work; a more flexible system for having multiple actions on a type without ambiguity (perhaps using the strategy outlined in section 4.9), and further proliferating support for right actions throughout mathlib (involving the heavy refactor discussed in section 4.7.2 and [*mathlib#7152*]).

12.3.2. Further development of graded algebraic objects

While chapter 6 develops the theory of graded rings and algebras in mathlib, it does not develop the theory of graded *morphisms* between these structures. Constructions such as the one in section 10.3 are in fact isomorphisms of *graded* algebras, which preserve the grade when applied to homogeneous elements.

In the presence of such new theory, we would still need to show that the graded tensor product is itself a graded algebra, before we can strengthen theorem 10.25 into a morphism of graded algebras.

12.3.3. Further comparison between flat and nested structures

Chapter 7 extensively explores a functionality issue with nested structures that provided a major obstacle to the conversion of mathlib from Lean 3 to Lean 4. The obstacle is now fixed, and everything is working correctly with nested structures.

While nested structures are often useful, it is not clear to the author that they always the preferred tool for the job. There are two aspects in which nested structures may still be suboptimal:

API surface Nested inheritance can result in an asymmetric interface to structures. Consider for instance this reduced example that defines morphisms of rings:

```
structure MonoidHom [Monoid A] [Monoid B] where
  toFun : A → B
  map_mul (a b : A) : toFun (a * b) = toFun (a * b)

structure AddMonoidHom [AddMonoid A] [AddMonoid B] where
  toFun : A → B
  map_add (a b : A) : toFun (a + b) = toFun (a + b)

structure RingHom [Ring A] [Ring B] extends MonoidHom A B, AddMonoidHom A B
```

Morally, `RingHom` is a function `toFun` and two proof fields `map_mul` and `map_add`; and indeed for flat inheritance this is exactly the behavior.

For nested structures, the type of constructor `mk` shows that symmetry between `add` and `mul` has been broken:

```
#check RingHom.mk
-- RingHom.mk (toMonoidHom : MonoidHom A B) (map_add : _) : RingHom A B
```

This is hidden from the user when using named field notation such as `{ toFun := _, map_add := _, map_mul := _ }`, but as it is present in the underlying term, it can affect the behavior of tactics like `simp`. Symmetry is again broken if we inspect the `toFun` field,

```
variable [Ring A] [Ring B] (f : RingHom A B)
#check f.toFun                -- f.toFun
#check f.toMonoidHom.toFun    -- f.toFun
#check f.toAddMonoidHom.toFun -- (RingHom.toAddMonoidHom f).toFun
```

where we see that the printer and elaborator have special behavior to pretend that `f.toMonoidHom.toFun` is just `f.toFun`; but the pretense is asymmetric and doesn't apply when viewing the function additively.

Ultimately these symmetry concerns are minor, and the quirks they caused did not obstruct the port from Lean 3 to Lean 4; but in the cases where nested structures are not offering performance benefits, a minor improvement is still an improvement.

Performance One of the main arguments for nested inheritance is that it improves performance: the "preferred" parents have special handling, and the projections to those parents are in some sense "cheaper" operations than projections to other parents, as there is no reconstruction to perform. This is a great optimization for structures with exactly one parent, or even for structures where one parent is clearly more important than another.

Unfortunately, this is far from the way that mathlib's typeclasses are currently structured; for instance, `CommRing` could justifiably have all of `Ring`, `CommSemiring`, `NonUnitalCommRing`, `NonAssocCommRing` as parents, and things get even worse if normed rings are thrown into the mix. This puts mathlib in the awkward position where it has to decide that one mathematical concept is more important than another; only one of these typeclasses can be the "preferred" parent, and experimentation from other contributors seems to suggest that many choices make significant performance trade-offs between formalizations in different branches of mathematics.

In [lean4#2451], Matthew Ballard identified that even in the case of preferred parents, it was easy for users to fall into a nasty performance trap relating to η-expansion; the fix in [lean4#2478] led to significant performance improvements, at the expense of further loss of symmetry between parents.

In [*lean4#2940*], the author has a prototype that introduces `extends flat` syntax, to avoid needing the workaround in section 7.4.2. This can be used to experiment with the effects on flat

inheritance on performance and API surface, the results of which would provide arguments for or against the author's proposal to make flat structures a bona fide language feature in [*lean4#2666*].

12.3.4. Syntactic support for universal properties

In chapter 8, in many places a notation halfway between mathematics and code was used, for example in eq. (8.23) which is reproduced below.

$$\begin{aligned} \operatorname{lift}^+_{\text{aux}}[f] &: \mathcal{G}(V,Q) \to (A \oplus S) \to (A \oplus S) \\ \operatorname{lift}^+_{\text{aux}}[f](v : V) &\coloneqq \quad (a,s) \mapsto (s(v), w \mapsto f(w,v)a), \end{aligned}$$

In principle, it would be possible to translate this directly to Lean notation.

This work was done before Lean 4 and its version of mathlib were ready, and Lean 3 had very little to offer in the direction of custom notation[2]. Now that Lean 4 underpins all of mathlib, it would be possible to explore encoding the above spelling more directly.

Tomáš Skřivan has already made some progress in this direction with the `fun x =>L[ℝ] k * x` notation in [90], which is a step towards being able to write $(a,s) \mapsto (s(v), w \mapsto f(w,v)a)$ and have Lean automatically prove its linearity. In theory, this could be combined with a syntax parser with awareness of universal properties, to form something like the following

```
-- this is not currently valid syntax!
def lift_aux (f : V →ₗ[R] V →ₗ[R] A) :
    CliffordAlgebra Q →ₗ[R] (A × S) →ₗ[R] (A × S)
  | .ι v              ↦ fun (a,  s) ↦ₗ[R] (s v, fun w ↦ₗ[R] f w v * a)
  ι_sq_scalar => by
    sorry
```

where the `sorry` is a placeholder for the proof from eq. (8.26). This is certainly closer to eq. (8.23) than the point-free spellings that this thesis has been corralled towards, though further thought would be required to come up with a general (but not too surprising) notation that works for all the universal properties in this thesis.

12.3.5. Formalizing further elementary results about Clifford algebras

A key result that is missing from the formalizations in this thesis is that the Clifford algebra of a free module is itself a free module. To the author, the obvious way to prove this result is to construct an explicit model for the Clifford algebra; the very approach from section 2.1 that we avoided!

A simplified version of this (for the exterior algebra) was prototyped, under the author's tutoring, by three attendees at LftCM 2023. The version of this adapted for the Clifford algebra might start as follows

[2] The `!![a, b; c, d]` notation added in [*mathlib#14991*] by the author was one of only three `user_notations` in mathlib; the other two were for string formatting.

```
variable {ι R : Type*} [LinearOrder ι] [CommRing R]

/-- Indices of our basis elements -/
abbrev Model.Index : Type _ := {l : List ι // l.Sorted (· < ·) }

/-- The model for a Clifford algebra over the free module `ι →₀ R`,
with `B` describing the scalar product of the basis vectors. -/
def Model (B : ι → ι → R) : Type _ := Model.Index ι →₀ R
```

which defines `Model ι B` as the free module over the appropriate basis elements. This definition is more general than the one discussed in section 9.2.3, as it does not require that the basis be orthogonal with respect to the quadratic form, nor does it require that the index of the basis `ι` be finite. The key challenge here is implementing the multiplicative structure, with statement `Ring (Model ι B)`, which the non-orthogonal basis complicates.

The universal property follows much more simply; at which point we can use the approach from section 8.2.2 to construct an isomorphism between `Model ι B` and `CliffordAlgebra QB` for a suitable `QB : QuadraticForm R (ι →₀ R)` derived from `B`. This is all we need to show that `CliffordAlgebra (Q : QuadraticForm R V` is free whenever `V` is, as free-ness of `V` gives us the isomorphism between `V` and `ι →₀ R`.

A simple consequence would be to show that the Clifford algebra is finite-dimensional when its vector space is; a prerequisite for constructing the "pseudoscalar" $I = \prod_i e_i$ that literature on geometric algebra makes heavy reference to, as the product is nonsensical without a proof that there are finitely many basis vectors.

12.3.6. Improvements to mathlib's calculus library

One of the author's original hopes was to be able to formalize parts of "geometric calculus" from [14]. As discussed in section 11.3, there are currently issues with the definition of derivatives in mathlib that make them challenging to apply to matrices, let alone multivectors; so this goal was not within reach. Another missing piece is integration of differential forms.

12.4. Summary

Despite on the surface being about Clifford algebras, this thesis explored a deep slice of the reality of formalizing mathematics: exploring foundational type theoretic concerns, addressing challenges in elementary algebra, constructing complex yet composable algebraic objects, and adapting written mathematics to exploit the behavior of the theorem proving system. While the formalizations are all in Lean and mathlib, many of the lessons learned are transferrable to other systems, and could inspire design decisions in systems which are yet to be built. With any luck, formalized mathematics is here to stay.

References

[1] Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*. Morgan Kaufmann Series in Computer Graphics. Amsterdam : San Francisco: Elsevier ; Morgan Kaufmann, 2007. 626 pp. ISBN: 978-0-12-374942-0 (cit. on pp. iv, 100, 125, 132, 167).

[2] Eric Wieser and Jujian Zhang. "Graded Rings in Lean's Dependent Type Theory". In: *Intelligent Computer Mathematics*. CICM 2023. Ed. by Kevin Buzzard and Temur Kutsia. Vol. 13467. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 122–137. ISBN: 978-3-031-16680-8 978-3-031-16681-5. DOI: 10.1007/978-3-031-16681-5_8 (cit. on pp. iv, 3, 63).

[3] Eric Wieser. "Multiple-Inheritance Hazards in Dependently-Typed Algebraic Hierarchies". In: *Intelligent Computer Mathematics*. CICM 2023. Ed. by Catherine Dubois and Manfred Kerber. Vol. 14101. Lecture Notes in Computer Science. Cham: Springer, July 21, 2023, pp. 222–236. ISBN: 978-3-031-42753-4. DOI: 10.1007/978-3-031-42753-4_15. arXiv: 2306.00617 [cs.LO] (cit. on pp. iv, 3, 79, 90).

[4] Eric Wieser and Joan Lasenby. "Computing with the Universal Properties of the Clifford Algebra and the Even Subalgebra". In: *Advanced Computational Applications of Geometric Algebra*. ICACGA 2023. Ed. by David W. Silva, Eckhard Hitzer, and Dietmar Hildenbrand. Vol. 13771. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2024, pp. 199–211. ISBN: 978-3-031-34031-4. DOI: 10.1007/978-3-031-34031-4_17 (cit. on pp. v, 3, 95).

[5] Chris Doran and Anthony Lasenby. *Geometric Algebra for Physicists*. Cambridge: Cambridge University Press, 2003. ISBN: 978-0-511-80749-7. DOI: 10.1017/CBO9780511807497. URL: http://ebooks.cambridge.org/ref/id/CBO9780511807497 (visited on 10/09/2019) (cit. on pp. 1, 132, 167).

[6] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. CADE 2015. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21400-9 978-3-319-21401-6. DOI: 10.1007/978-3-319-21401-6_26 (cit. on pp. 1, 66, 80).

[7] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. CADE 2021. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79875-8 978-3-030-79876-5. DOI: 10.1007/978-3-030-79876-5_37 (cit. on pp. 1, 80).

[8] Kevin Hartnett. "Proof Assistant Makes Jump to Big-League Math". In: *Quanta Magazine* (July 28, 2021). URL: https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/ (visited on 02/19/2024) (cit. on p. 2).

[9] Leila Sloman. "'A-Team' of Math Proves a Critical Link Between Addition and Sets". In: *Quanta Magazine* (Dec. 6, 2023). URL: https://www.quantamagazine.org/a-team-of-math-proves-a-critical-link-between-addition-and-sets-20231206/ (visited on 02/19/2024) (cit. on p. 2).

[10] Eric Wieser and Utensil Song. "Formalizing Geometric Algebra in Lean". In: *Advances in Applied Clifford Algebras* 32.3 (July 2022), p. 28. ISSN: 0188-7009, 1661-4909. DOI: 10.1007/s00006-021-01164-1. arXiv: 2306.00617 [cs.LO]. URL: https://link.springer.com/10.1007/s00006-021-01164-1 (visited on 04/23/2022) (cit. on pp. 3, 25, 77, 109, 110, 121, 123, 133).

[11] Eric Wieser. "Scalar Actions in Lean's Mathlib". In: *Workshop Papers of the 14th Conference on Intelligent Computer Mathematics*. CICM 2021. Vol. 3377. Timisoara, Romania: CEUR-WS, Aug. 10, 2021. arXiv: 2108.10700 [cs.LO] (cit. on pp. 3, 36).

[12] Eric Wieser. "Chaining extensionality lemmas in Lean's Mathlib". In: CICM 2024. Lecture Notes in Computer Science. in review (cit. on pp. 3, 55).

[13] Eric Wieser and Joan Lasenby. "Computing with the Universal Properties of the Clifford Algebra and the Even Subalgebra". In: *Advances in Applied Clifford Algebras* (in review) (cit. on pp. 3, 95, 96).

[14] David Hestenes and Garret Sobczyk. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics*. Fundamental Theories of Physics. Springer Netherlands, 1984. ISBN: 978-90-277-1673-6. DOI: 10.1007/978-94-009-6292-7. URL: https://www.springer.com/gp/book/9789027716736 (visited on 05/04/2020) (cit. on pp. 7, 21, 101, 107, 132, 181).

[15] Bertfried Fauser and Rafał Abłamowicz. "On the Decomposition of Clifford Algebras of Arbitrary Bilinear Form". In: *Clifford Algebras and Their Applications in Mathematical Physics*. 5th International Conference on Clifford Algebras and Their Applications in Mathematical Physics. Ed. by Rafał Abłamowicz and Bertfried Fauser. Vol. 18. Progress in Physics. Boston, MA: Birkhäuser Boston, 2000, pp. 341–366. ISBN: 978-1-4612-1368-0. DOI: 10.1007/978-1-4612-1368-0_18 (cit. on pp. 13, 127).

[16] Nicolas Bourbaki. *Algèbre, Chapitre 9*. Réimpression inchangée de l'éd. originale. Eléments de mathématique 2. Berlin: Springer, 2007. ISBN: 978-3-540-35338-6 (cit. on pp. 13, 114).

[17] Zur Izhakian, Manfred Knebusch, and Louis Rowen. "Supertropical Quadratic Forms I". In: *Journal of Pure and Applied Algebra* 220.1 (Jan. 2016), pp. 61–93. ISSN: 00224049. DOI: 10.1016/j.jpaa.2015.05.043. URL: https://linkinghub.elsevier.com/retrieve/pii/S0022404915001589 (visited on 01/17/2024) (cit. on p. 13).

[18] Richard James Wareham. "Computer Graphics Using Conformal Geometric Algebra". University of Cambridge, Nov. 2006. URL: https://rjw57.github.io/phd-thesis/rjw-thesis.pdf (cit. on p. 16).

[19] Pablo Colapinto. "Versor: Spatial Computing with Conformal Geometric Algebra". MA thesis. University of California at Santa Barbara, 2011. URL: http://versor.mat.ucsb.edu (cit. on pp. 16, 17).

[20] D. H. F. Dijkman. "Efficient Implementation of Geometric Algebra". Universiteit van Amsterdam, Oct. 2007. URL: https://dare.uva.nl/search?identifier=627c5dcb-1b7d-4501-ba81-8c5a97db2749 (visited on 01/18/2024) (cit. on p. 16).

[21] Stéphane Breuils, Vincent Nozick, and Laurent Fuchs. "Garamon: A Geometric Algebra Library Generator". In: *Advances in Applied Clifford Algebras* 29.4 (Sept. 2019), p. 69. ISSN: 0188-7009, 1661-4909. DOI: 10.1007/s00006-019-0987-7. URL: http://link.springer.com/10.1007/s00006-019-0987-7 (visited on 08/12/2020) (cit. on pp. 17, 18, 112).

[22] Steven De Keninck. *Ganja.Js*. Zenodo, 2020. DOI: 10.5281/ZENODO.3635774. URL: https://zenodo.org/record/3635774 (cit. on p. 17).

[23] Christian Schwinn, Dietmar Hildenbrand, Florian Stock, and Andreas Koch. "Gaalop 2.0 - A Geometric Algebra Algorithm Compiler". In: 2010 (cit. on pp. 17, 21).

[24] Leandro Augusto Frata Fernandes. *GATL: Geometric Algebra Template Library*. URL: https://github.com/laffernandes/gatl (cit. on p. 17).

[25] Jeremy Ong. *GAL*. 2019. URL: https://github.com/jeremyong/gal (cit. on p. 17).

[26] Alex Arsenovic, Hugo Hadfield, Eric Wieser, Robert Kern, and The Pygae Team. *Pygae/Clifford: V1.3.1*. Version v1.3.1. Zenodo, June 3, 2020. DOI: 10.5281/ZENODO.1453978. URL: https://zenodo.org/record/1453978 (visited on 06/23/2020) (cit. on p. 17).

[27] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array Programming with NumPy". June 17, 2020. arXiv: 2006.10256 [cs, stat]. URL: http://arxiv.org/abs/2006.10256 (visited on 06/23/2020) (cit. on pp. 18, 36, 44).

[28] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*. The Second Workshop. Austin, Texas: ACM Press, 2015, pp. 1–6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162. URL: http://dl.acm.org/citation.cfm?doid=2833157.2833162 (visited on 08/13/2020) (cit. on p. 18).

[29] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AmiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. "SymPy: Symbolic Computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2, 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://peerj.com/articles/cs-103 (visited on 06/23/2020) (cit. on p. 21).

[30] Alan Bromborsky, Utensil Song, Eric Wieser, Hugo Hadfield, and The Pygae Team. *Pygae/Galgebra: V0.5.0*. Version v0.5.0. Zenodo, June 4, 2020. DOI: 10.5281/ZENODO.3857096. URL: https://zenodo.org/record/3857096 (visited on 06/23/2020) (cit. on p. 21).

[31] Robin Lloyd and CNN Interactive Senior Writer. "Metric Mishap Caused Loss of NASA Orbiter". In: *CNN Interactive* (1999), p. 11 (cit. on p. 24).

[32] Stanislas Polu and Ilya Sutskever. *Generative Language Modeling for Automated Theorem Proving*. Sept. 7, 2020. arXiv: 2009.03393 [cs.LG]. preprint (cit. on p. 26).

[33] Sean Welleck and Rahul Saha. *LLMSTEP: LLM Proofstep Suggestions in Lean*. Oct. 27, 2023. arXiv: 2310.18457 [cs.AI]. preprint (cit. on p. 26).

[34] Lawrence C. Paulson and Jasmin Christian Blanchette. "Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers". In: *The 8th International Workshop on the Implementation of Logics*. IWIL 2010. Ed. by Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska. Vol. 2. EPiC Series in Computing. EasyChair, 2012, pp. 1–11. DOI: 10.29007/36dt. URL: https://easychair.org/publications/paper/wV (cit. on p. 26).

[35] Jannis Limperg and Asta Halkjær From. "Aesop: White-Box Best-First Proof Search for Lean". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston MA USA: ACM, Jan. 11, 2023, pp. 253–266. ISBN: 9798400700262. DOI: 10.1145/3573105.3575671 (cit. on p. 26).

[36] *Learning Lean*. URL: https://leanprover-community.github.io/learn.htm (visited on 03/09/2021) (cit. on p. 26).

[37] Kevin Buzzard. *Division by Zero in Type Theory: A FAQ*. Xena. June 5, 2020. URL: https://xenaproject.wordpress.com/2020/07/05/division-by-zero-in-type-theory-a-faq/ (cit. on pp. 29, 169, 173).

[38] Kaare Brandt Petersen and Michael Syskind Pedersen. *The Matrix Cookbook*. Nov. 15, 2012. URL: https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf (visited on 01/14/2024) (cit. on pp. 29, 30, 171, 176).

[39] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. POPL '20: 47th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. New Orleans LA USA: ACM, Jan. 20, 2020, pp. 367–381. ISBN: 978-1-4503-7097-4. DOI: 10.1145/3372885.3373824. URL: https://dl.acm.org/doi/10.1145/3372885.3373824 (visited on 08/21/2020) (cit. on pp. 31–33, 37, 45, 66, 80).

[40] *Mathlib Statistics*. URL: https://leanprover-community.github.io/mathlib_stats.html (visited on 01/17/2021) (cit. on p. 31).

[41] *Undergrad Math in Mathlib*. URL: https://leanprover-community.github.io/undergrad.html (visited on 03/03/2021) (cit. on p. 31).

[42] Floris van Doorn, Gabriel Ebner, and Robert Y. Lewis. "Maintaining a Library of Formal Mathematics". In: *Intelligent Computer Mathematics*. CICM 2020. Ed. by Christoph Benzmüller and Bruce Miller. Vol. 12236. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 251–267. ISBN: 978-3-030-53517-9 978-3-030-53518-6. DOI: 10.1007/978-3-030-53518-6_16. arXiv: 2004.03673 [cs.PL]. URL: http://link.springer.com/10.1007/978-3-030-53518-6_16 (visited on 03/03/2021) (cit. on p. 31).

[43] Anne Baanen, Sander R. Dahmen, Ashvni Narayanan, and Filippo A. E. Nuccio Mortarino Majno Di Capriglio. "A Formalization of Dedekind Domains and Class Groups of Global Fields". In: *Journal of Automated Reasoning* 66.4 (Nov. 2022), pp. 611–637. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-022-09644-0. URL: https://link.springer.com/10.1007/s10817-022-09644-0 (visited on 12/14/2023) (cit. on p. 42).

[44] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. "Competing Inheritance Paths in Dependent Type Theory: A Case Study in Functional Analysis". In: *Automated Reasoning*. IJCAR 2020. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 3–20. ISBN: 978-3-030-51053-4 978-3-030-51054-1. DOI: 10.1007/978-3-030-51054-1_1 (cit. on pp. 45, 70, 86, 176).

[45] Anne Baanen. "Use and Abuse of Instance Parameters in the Lean Mathematical Library". In: *ITP 2022*. Haifa, Israel, May 2, 2022. arXiv: 2202.01629 [cs.LO] (cit. on pp. 45, 67, 73, 75, 80, 82, 85, 87, 92, 176, 178).

[46] Nicolas Bourbaki. *Algebra I, Chapters 1-3*. Elements of Mathematics. Berlin Heidelberg: Springer, 1989. 708 pp. ISBN: 978-3-540-64243-5 (cit. on pp. 64, 65, 157, 161).

[47] The Stacks project authors. *The Stacks Project*. 2022. URL: https://stacks.math.columbia.edu (cit. on p. 65).

[48] Davide Castelvecchi. "Mathematicians Welcome Computer-Assisted Proof in 'Grand Unification' Theory". In: *Nature* 595.7865 (July 1, 2021), pp. 18–19. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/d41586-021-01627-2. URL: http://www.nature.com/articles/d41586-021-01627-2 (visited on 07/24/2022) (cit. on p. 65).

[49] César Domínguez and Julio Rubio. "Effective Homology of Bicomplexes, Formalized in Coq". In: *Theoretical Computer Science* 412.11 (Mar. 2011), pp. 962–970. ISSN: 03043975. DOI: 10.1016/j.tcs.2010.11.016. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397510006493 (visited on 05/27/2022) (cit. on p. 66).

[50] Jeremy Avigad, Steve Awodey, Ulrik Buchholtz, Floris van Doorn, Clive Newstead, Egbert Rijke, and Mike Shulman. *Spectral Sequences in Homotopy Type Theory*. Nov. 2015. URL: https://github.com/cmu-phil/Spectral (cit. on p. 66).

[51] Guillaume Brunerie, Axel Ljungström, and Anders Mörtberg. "Synthetic Integral Cohomology in Cubical Agda". In: *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*. Ed. by Florin Manea and Alex Simpson. Vol. 216. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 11:1–11:19. ISBN: 978-3-95977-218-1. DOI: 10.4230/LIPIcs.CSL.2022.11. URL: https://drops.dagstuhl.de/opus/volltexte/2022/15731 (cit. on p. 66).

[52] Reid Barton, Johan Commelin, Kevin Buzzard, Kenny Lau, and Mario Carneiro. *#maths > CDGAs*. Lean Zulip Chat. June 11, 2019. URL: https://leanprover-community.github.io/archive/stream/116395-maths/topic/CDGAs.html (cit. on pp. 66, 68).

[53] Jacques Carette, William M. Farmer, and Yasmine Sharoda. "Leveraging the Information Contained in Theory Presentations". In: *Intelligent Computer Mathematics*. CICM 2020. Ed. by Christoph Benzmüller and Bruce Miller. Vol. 12236. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 55–70. ISBN: 978-3-030-53517-9 978-3-030-53518-6. DOI: 10.1007/978-3-030-53518-6_4 (cit. on pp. 80, 92).

[54] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. "Hierarchy Builder: Algebraic Hierarchies Made Easy in Coq with Elpi (System Description)". In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Ed. by Zena M. Ariola. Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 34:1–34:21. ISBN: 978-3-95977-155-9. DOI: 10.4230/LIPIcs.FSCD.2020.34. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12356 (cit. on pp. 88, 91, 92).

[55] Andreas Abel. "On Extensions to Definitional Equality in Agda". 10th Agda Implementors' Meeting (Gothenburg, Sweden). Sept. 15, 2009. URL: https://www.cse.chalmers.se/~abela/talkAIM09.pdf (cit. on pp. 88, 92).

[56] Sébastien Gouëzel. *#mathlib4 > Some Observations on Eta Experiment*. Lean Zulip Chat. May 3, 2023. URL: https://leanprover.zulipchat.com/#narrow/stream/287929-mathlib4/topic/Some.20observations.20on.20eta.20experiment/near/355336941 (cit. on p. 90).

[57] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. "Packaging Mathematical Structures". In: *Theorem Proving in Higher Order Logics*. TPHOLs 2009. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 327–342. ISBN: 978-3-642-03358-2 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_23 (cit. on p. 91).

[58] Bas Spitters and Eelis Van Der Weegen. "Type Classes for Mathematics in Type Theory". In: *Mathematical Structures in Computer Science* 21.4 (Aug. 2011), pp. 795–825. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129511000119. URL: https://www.cambridge.org/core/product/identifier/S0960129511000119/type/journal_article (visited on 06/12/2023) (cit. on pp. 91, 92).

[59] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Sept. 28, 2022. URL: https://zenodo.org/record/7118596 (visited on 06/03/2023) (cit. on p. 91).

[60] Kazuhiko Sakaguchi. "Validating Mathematical Structures". In: *Automated Reasoning*. IJCAR 2020. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 138–157. ISBN: 978-3-030-51053-4 978-3-030-51054-1. DOI: 10.1007/978-3-030-51054-1_8 (cit. on p. 92).

[61] Clemens Ballarin. "Exploring the Structure of an Algebra Text with Locales". In: *Journal of Automated Reasoning* 64.6 (Aug. 2020), pp. 1093–1121. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-019-09537-9. URL: http://link.springer.com/10.1007/s10817-019-09537-9 (visited on 06/19/2023) (cit. on p. 92).

[62] Erik Poll and Simon Thompson. "Integrating Computer Algebra and Reasoning through the Type System of Aldor". In: *Frontiers of Combining Systems*. Ed. by Hélène Kirchner and Christophe Ringeissen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 136–150. ISBN: 978-3-540-46421-1 (cit. on p. 92).

[63] Anthony Lasenby, Joan Lasenby, and Richard Wareham. *A Covariant Approach to Geometry Using Geometric Algebra*. F-INFENG/TR-483. Department of Engineering, University of Cambridge, 2004, p. 90. URL: https://pdfs.semanticscholar.org/baba/976fd7f6577eeaa1d3ef488c1db13ec24652.pdf (cit. on pp. 95, 167).

[64] Pertti Lounesto. *Clifford Algebras and Spinors*. 2nd ed. Vol. 286. London Mathematical Society Lecture Note Series. Cambridge University Press, 2001. ISBN: 978-0-511-52602-2. DOI: 10.1017/CBO9780511526022 (cit. on pp. 97, 98).

[65] Claude C. Chevalley. *The Algebraic Theory of Spinors*. Columbia University Press, 1954. ISBN: 978-0-231-89180-6. DOI: 10.7312/chev93056. URL: https://www.degruyter.com/document/doi/10.7312/chev93056/html (visited on 08/09/2022) (cit. on pp. 97, 98).

[66] M G Mahmoudi. "Orthogonal Symmetries and Clifford Algebras". In: *Proceedings - Mathematical Sciences* 120.5 (2010), pp. 535–561. ISSN: 0973-7685. DOI: 10.1007/s12044-010-0050-z. arXiv: 1006.0997 [math.RA] (cit. on p. 101).

[67] Ian R. Porteous. *Clifford Algebras and the Classical Groups*. 1st ed. Cambridge University Press, Oct. 5, 1995. ISBN: 978-0-521-55177-9 978-0-511-47091-2 978-0-521-11802-6. DOI: 10.1017/CBO9780511470912. URL: https://www.cambridge.org/core/product/identifier/9780511470912/type/book (visited on 02/21/2023) (cit. on pp. 104, 106).

[68] H.B. Lawson and M.L. Michelsohn. *Spin Geometry (PMS-38), Volume 38*. Princeton Mathematical Series. Princeton University Press, 1989. ISBN: 978-0-691-08542-5. URL: https://books.google.co.uk/books?id=3d9JkN8w3X8C (cit. on pp. 105, 155, 161, 162, 168).

[69] Rafał Abłamowicz and Pertti Lounesto. "On Clifford Algebras of a Bilinear Form with an Antisymmetric Part". In: *Clifford Algebras with Numeric and Symbolic Computations*. Ed. by Rafał Abłamowicz, Josep M. Parra, and Pertti Lounesto. Boston, MA: Birkhäuser Boston, 1996, pp. 167–188. ISBN: 978-1-4615-8159-8 978-1-4615-8157-4. DOI: 10.1007/978-1-4615-8157-4_11. URL: http://link.springer.com/10.1007/978-1-4615-8157-4_11 (visited on 01/11/2024) (cit. on p. 107).

[70] Darij Grinberg. "The Clifford Algebra and the Chevalley Map- a Computational Approach (Summary Version 1)". June 2016. URL: http://mit.edu/~darij/www/algebra/chevalleys.pdf (cit. on pp. 107, 108, 121, 164).

[71] Darij Grinberg. *Answer to "Clifford PBW Theorem for Quadratic Form"*. MathOverflow. Feb. 9, 2012. URL: https://mathoverflow.net/a/87958/172242 (visited on 01/12/2024) (cit. on pp. 110, 129–131).

[72] Tetsuo Ida, Jacques Fleuriot, and Fadoua Ghourabi. "A New Formalization of Origami in Geometric Algebra". In: *Proceedings of ADG 2016*. Eleventh International Workshop on Automated Deduction in Geometry. Strasbourg, France, June 2016, pp. 117–136. URL: https://hal.inria.fr/hal-01334334 (cit. on pp. 111, 112, 162).

[73] Laurent Fuchs and Laurent Thery. "A Formalization of Grassmann-Cayley Algebra in COQ and Its Application to Theorem Proving in Projective Geometry". In: *Automated Deduction in Geometry, ADG 2010*. Ed. by Julien Narboux Pascal Schreck and Jürgen Richter-Gebert. Vol. 6877. Lecture Notes in Computer Science. Munich, Germany: Springer, July 2010, pp. 51–62. DOI: 10.1007/978-3-642-25070-5_3. URL: https://hal.archives-ouvertes.fr/hal-00657901 (cit. on p. 112).

[74] Laurent Fuchs and Laurent Théry. "Implementing Geometric Algebra Products with Binary Trees". In: *Advances in Applied Clifford Algebras* 24.2 (June 2014), pp. 589–611. ISSN: 0188-7009, 1661-4909. DOI: 10.1007/s00006-014-0447-3. URL: https://hal.inria.fr/hal-01095495 (cit. on pp. 113, 162).

[75] Li-Ming Li, Zhi-Ping Shi, Yong Guan, Qian-Ying Zhang, and Yong-Dong Li. "Formalization of Geometric Algebra in HOL Light". In: *Journal of Automated Reasoning* 63.3 (Oct. 2019), pp. 787–808. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-018-9498-9. URL: http://link.springer.com/10.1007/s10817-018-9498-9 (visited on 08/21/2020) (cit. on pp. 113, 137, 162).

[76] Eric Wieser, Reid Barton, and Kevin Buzzard. *#new Members > Induction on a Submonoid*. Lean Zulip Chat. Nov. 9, 2020. URL: https://leanprover.zulipchat.com/#narrow/stream/113489-new-members/topic/Induction.20on.20a.20submonoid/near/216369226 (cit. on p. 124).

[77] David Hestenes. *New Foundations for Classical Mechanics*. Kluwer Academic Publishers, 1999. ISBN: 978-0-7923-5302-7. URL: http://geocalc.clas.asu.edu/html/NFCM.html (cit. on p. 132).

[78] Dan Piponi. "Automatic Differentiation, C++ Templates, and Photogrammetry". In: *Journal of Graphics Tools* 9 (Jan. 1, 2004). DOI: 10.1080/10867651.2004.10504901 (cit. on p. 135).

[79] Heather Macbeth. "Using Polyrith". In: *Computations in Lean*. July 10, 2022. URL: https://hrmacbeth.github.io/computations_in_lean/02_Using_Polyrith.html#double-cover-of-so-3 (cit. on p. 136).

[80] José Figueroa-O'Farrill. *Spin Geometry*. May 18, 2017. URL: https://empg.maths.ed.ac.uk/Activities/Spin/SpinNotes.pdf (cit. on pp. 142, 155).

[81] Antoine Chambert-Loir. *#Is There Code for X? > Base Change for Bilinear Maps and Quadratic Forms*. Lean Zulip Chat. July 14, 2023. URL: https://leanprover.zulipchat.com/#narrow/stream/217875-Is-there-code-for-X.3F/topic/Base.20change.20for.20bilinear.20maps.20and.20quadratic.20forms/near/375126978 (cit. on p. 142).

[82] Christian Kassel. "Tensor Products". In: *Quantum Groups*. Vol. 155. New York, NY: Springer New York, 1995, pp. 23–38. ISBN: 978-1-4612-6900-7 978-1-4612-0783-2. DOI: 10.1007/978-1-4612-0783-2_2. URL: http://link.springer.com/10.1007/978-1-4612-0783-2_2 (visited on 09/28/2023) (cit. on p. 150).

[83] Reynald Affeldt and Cyril Cohen. "Formal Foundations of 3D Geometry to Model Robot Manipulators". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017. Paris France: ACM, Jan. 16, 2017, pp. 30–42. ISBN: 978-1-4503-4705-1. DOI: 10.1145/3018610.3018629. URL: https://inria.hal.science/hal-01414753 (cit. on p. 163).

[84] Jean Gallier. *Notes on Differential Geometry and Lie Groups*. Department of Computer and Information Science, University of Pennsylvania, June 20, 2011. URL: https://www.cis.upenn.edu/~cis6100/diffgeom-n.pdf (cit. on p. 165).

[85] Ravinder Rupchand Puri. "Algebra of the Exponential Operator". In: *Mathematical Methods of Quantum Optics*. Red. by William T. Rhodes. Vol. 79. Springer Series in Optical Sciences. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 37–53. ISBN: 978-3-642-08732-5 978-3-540-44953-9. DOI: 10.1007/978-3-540-44953-9_2. URL: http://link.springer.com/10.1007/978-3-540-44953-9_2 (visited on 01/06/2024) (cit. on pp. 172, 174).

[86] V. Majerník. "Basic Space-Time Transformations Expressed by Means of Two-Component Number Systems". In: *Acta Physica Polonica A* 86.3 (Sept. 1994), pp. 291–295. ISSN: 0587-4246, 1898-794X. DOI: 10.12693/APhysPolA.86.291. URL: http://przyrbwn.icm.edu.pl/APP/PDF/86/a086z3p01.pdf (visited on 01/25/2024) (cit. on p. 173).

[87] J. M. Selig. "Exponential and Cayley Maps for Dual Quaternions". In: *Advances in Applied Clifford Algebras* 20.3 (3 Oct. 1, 2010), pp. 923–936. ISSN: 1661-4909. DOI: 10.1007/s00006-010-0229-5. URL: https://link.springer.com/article/10.1007/s00006-010-0229-5 (visited on 02/11/2024) (cit. on p. 174).

[88] Yury Kudryashov and Anatole Dedecker. *#maths > Generalizing Deriv to TVS*. Lean Zulip Chat. May 18, 2023. URL: https://leanprover.zulipchat.com/#narrow/stream/116395-maths/topic/generalizing.20deriv.20to.20TVS/near/359284921 (cit. on p. 174).

[89] Jujian Zhang. "Formalising the Proj Construction in Lean". In: *14th International Conference on Interactive Theorem Proving*. ITP 2023. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 35:1–35:17. ISBN: 978-3-95977-284-6. DOI: 10.4230/LIPIcs.ITP.2023.35. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.35 (cit. on p. 177).

[90] Tomáš Skřivan. *Lecopivo/SciLean: Scientific Computing in Lean 4*. URL: https://github.com/lecopivo/SciLean (visited on 02/18/2024) (cit. on p. 180).

Github references

This section contains references to issues and pull requests on GitHub, which is the main place that development of community-owned software such as Lean and mathlib occurs. While not peer-reviewed in a conventional sense, the pull requests (i.e. code contributions) in this section have been reviewed by other contributors, and are only "merged" if they pass this review process. This process can be anything from an instant approval in seconds to a long discussion iterating on code changes that spans months! *Italicized keys* indicate self-references.

[*mathlib#14303*] Eric Wieser. *refactor(linear_algebra/quadratic_form/basic): generalize to semiring*. Reviewed by Adam Topaz. May 2022 (cit. on p. 13).

[sympy#20691] Sudeep Sidhu. *Make inversion of `Matrix` with `MatrixSymbol` as element possible*. Dec. 2020 (cit. on p. 24).

[*mathlib#8560*] Eric Wieser. *feat(matrix/kronecker): Add the Kronecker product*. Reviewed by Filippo A. E. Nuccio and Johan Commelin. Aug. 2021 (cit. on p. 29).

[*mathlib#12767*] Eric Wieser. *feat(linear_algebra/matrix): The Weinstein–Aronszajn identity*. Reviewed by Johan Commelin. Mar. 2022 (cit. on p. 30).

[*mathlib#14991*] Eric Wieser. *feat(data/matrix/notation): add `!![1, 2; 3, 4]` notation*. Reviewed by Anne Baanen. June 2022 (cit. on pp. 29, 180).

[*mathlib#18711*] Eric Wieser. *feat(data/matrix/reflection): lemmas for arbitrary concrete matrices, proved via reflection*. Reviewed by Oliver Nash and Yaël Dillies. Apr. 2023 (cit. on p. 29).

[*mathlib#5124*] Eric Wieser. *feat(linear_algebra/*): Use alternating maps for wedge and determinant*. Reviewed by Anne Baanen. Nov. 2020 (cit. on pp. 31, 165).

[*mathlib#4430*] Eric Wieser. *feat(linear_algebra/clifford_algebra): Add a definition derived from `exterior_algebra.lean`*. Reviewed by Anne Baanen, Adam Topaz, Heather Macbeth, and Utensil Song. Oct. 2020 (cit. on pp. 32, 61, 117).

[mathlib4#3840] Sébastien Gouëzel. *chore(*): tweak priorities for linear algebra*. Reviewed by Scott Morrison, Eric Wieser, and Floris van Doorn. May 2023 (cit. on pp. 32, 83).

[*mathlib4#8395*] Eric Wieser. *feat(Algebra/Module/Hom): `AddMonoid.End` application forms a `Module`*. Reviewed by Oliver Nash. Nov. 2023 (cit. on p. 38).

[*mathlib#8724*] Eric Wieser. *feat(group_theory/group_action): generalize* `mul_action.function_End` *to other endomorphisms*. Reviewed by Anne Baanen. Aug. 2021 (cit. on p. 38).

[*mathlib4#8396*] Eric Wieser. *feat(Algebra/GroupRingAction/Basic):* `RingHom` *application forms a* `MulDistribMulAction`. Reviewed by Oliver Nash. Nov. 2023 (cit. on p. 38).

[*mathlib#8968*] Eric Wieser. *feat(algebra/module/basic): add* `module.to_add_monoid_End`. Reviewed by Johan Commelin. Sept. 2021 (cit. on p. 39).

[mathlib#997] Johan Commelin. *feat(algebra/pointwise): More lemmas on pointwise multiplication*. Reviewed by Sébastien Gouëzel. May 2019 (cit. on p. 40).

[mathlib#12865] Yaël Dillies. *feat(data/finset/pointwise): Missing operations*. Reviewed by Johan Commelin. Mar. 2022 (cit. on p. 40).

[*mathlib#6891*] Eric Wieser. *feat(algebra/module/hom): Add missing smul instances on* `add_monoid_hom`. Reviewed by Aaron Anderson and Anne Baanen. Mar. 2021 (cit. on p. 40).

[mathlib#4784] Yury G. Kudryashov. *refactor(data/polynomial): use* `linear_map` *for* `monomial`, *review* `degree`. Reviewed by Eric Wieser, Bryan Gin-ge Chen, and Johan Commelin. Oct. 2020 (cit. on pp. 40, 41).

[*mathlib#7664*] Eric Wieser. *feat(data/polynomial): generalize* `polynomial.has_scalar` *to require only* `distrib_mul_action` *instead of* `semimodule`. Reviewed by Scott Morrison, Kenny Lau, Sébastien Gouëzel, Anne Baanen, and Johan Commelin. May 2021 (cit. on p. 41).

[*mathlib#4365*] Eric Wieser. *feat(data/monoid_algebra): Allow* `R ≠ k` *in* `semimodule R (monoid_algebra k G)`. Reviewed by Scott Morrison. Oct. 2020 (cit. on p. 41).

[mathlib#284] Scott Morrison. *generalise finsupp.to_module*. Aug. 2018 (cit. on p. 41).

[*mathlib#6533*] Eric Wieser. *feat(data/mv_polynomial/basic): a polynomial ring over an R-algebra is also an R-algebra*. Reviewed by Johan Commelin and Riccardo Brasca. Mar. 2021 (cit. on p. 41).

[mathlib#4770] Yury G. Kudryashov. *chore(group_theory/group_action): introduce* `smul_comm_class`. Reviewed by Johan Commelin, Eric Wieser, and Kevin Buzzard. Oct. 2020 (cit. on p. 42).

[*mathlib#6534*] Eric Wieser. *feat(data/finsupp, algebra/monoid_algebra): add* `is_scalar_tower` *and* `smul_comm_class`. Reviewed by Johan Commelin. Mar. 2021 (cit. on p. 42).

[*mathlib#6614*] Eric Wieser. *feat(data/dfinsupp): add* `is_scalar_tower` *and* `smul_comm_class`. Reviewed by Anne Baanen. Mar. 2021 (cit. on p. 42).

[*mathlib#8965*] Eric Wieser. *chore(field_theory/fixed): reuse existing* `mul_semiring_action.to_alg_hom` *by providing* `smul_comm_class`. Reviewed by Johan Commelin. Sept. 2021 (cit. on p. 42).

[*mathlib#15876*] Eric Wieser. *feat(group_theory/group_action/conj_act):* `smul_comm_class` *instances*. Reviewed by Oliver Nash. Aug. 2022 (cit. on p. 42).

[*mathlib#10262*] Eric Wieser. *feat(linear_algebra/pi_tensor_prod): generalize actions and provide missing* `smul_comm_class` *and* `is_scalar_tower` *instance*. Reviewed by Johan Commelin. Nov. 2021 (cit. on p. 42).

[*mathlib#6542*] Eric Wieser. *feat(data/mv_polynomial/basic): add* `is_scalar_tower` *and* `smul_comm_class` *instances*. Reviewed by Johan Commelin. Mar. 2021 (cit. on p. 42).

[*mathlib#6592*] Eric Wieser. *chore(data/polynomial/basic): add missing* `is_scalar_tower` *and* `smul_comm_class` *instances*. Reviewed by Johan Commelin. Mar. 2021 (cit. on p. 42).

[*mathlib#6139*] Eric Wieser. *chore(algebra/module/pi): add missing* `smul_comm_class` *instances*. Reviewed by Johan Commelin. Feb. 2021 (cit. on p. 42).

[*mathlib#5205*] Eric Wieser. *feat(algebra/module/basic): Add* `smul_comm_class` *instances*. Reviewed by Anne Baanen. Dec. 2020 (cit. on p. 42).

[*mathlib#5369*] Eric Wieser. *feat(algebra/module/basic): Add symmetric* `smul_comm_class` *instances for int and nat*. Reviewed by Kevin Buzzard, Floris van Doorn, and Anne Baanen. Dec. 2020 (cit. on p. 42).

[*mathlib#5509*] Eric Wieser. *fix(algebra/module/basic): Do not attach the* `ℕ` *and* `ℤ` `is_scalar_tower` *and* `smul_comm_class` *instances to a specific definition of* `smul`. Reviewed by Sébastien Gouëzel. Dec. 2020 (cit. on p. 42).

[*mathlib#13174*] Eric Wieser. *fix(algebra/module/basic,group_theory/group_action/defs): generalize nat and int* `smul_comm_class` *instances*. Reviewed by Johan Commelin. Apr. 2022 (cit. on p. 42).

[mathlib#7932] Oliver Nash. *feat(algebra/monoid_algebra): adjointness for the functor* `G ↦ monoid_algebra k G` *when* `G` *carries only* `has_mul`. Reviewed by Eric Wieser. June 2021 (cit. on p. 43).

[mathlib#7416] Kexing Ying. *feat(linear_algebra/quadratic_form): Complex version of Sylvester's law of inertia*. Reviewed by Eric Wieser, Johan Commelin, and Floris van Doorn. Apr. 2021 (cit. on p. 43).

[mathlib#7084] Sébastien Gouëzel. *refactor(*): kill nat multiplication diamonds*. Reviewed by Eric Wieser, Scott Morrison, and Johan Commelin. Apr. 2021 (cit. on pp. 45, 46).

[mathlib#12182] Gabriel Ebner. *feat:* `add_monoid_with_one`, `add_group_with_one`. Reviewed by Eric Wieser, Floris van Doorn, Yury G. Kudryashov, and Yaël Dillies. Feb. 2022 (cit. on p. 46).

[mathlib#14894] Anne Baanen. *chore({algebra,data/rat}): use forgetful inheritance for* `algebra_rat`. Reviewed by Eric Wieser, Riccardo Brasca, and Eric Rodriguez. June 2022 (cit. on p. 46).

[*mathlib#8945*] Eric Wieser. *feat(group_theory/sub{monoid,group}): pointwise actions on* `add_sub{monoid,group}`*s and* `sub{monoid,group,module,semiring,ring,algebra}`*s*. Reviewed by Chris Hughes. Sept. 2021 (cit. on p. 47).

[mathlib#8627] Chris Hughes. *feat(group_theory/group_action/conj_act): conjugation action of groups*. Reviewed by Eric Wieser and Johan Commelin. Aug. 2021 (cit. on p. 47).

[mathlib#8592] Chris Hughes. *feat(group_theory/group_action/subgroup): Conjugation action on subgroups of a group*. Reviewed by Eric Wieser, Yaël Dillies, and Johan Commelin. Aug. 2021 (cit. on p. 47).

[*mathlib#7630*] Eric Wieser. *feat(algebra/opposites): add* `has_scalar (opposite α) α` *instances*. Reviewed by Anne Baanen. May 2021 (cit. on p. 48).

[*mathlib#10543*] Eric Wieser. *feat(group_theory/group_action/defs): add a typeclass to show that an action is central (aka symmetric)*. Reviewed by @Julian-Kuelshammer, Johan Commelin, and Yury G. Kudryashov. Nov. 2021 (cit. on p. 49).

[*mathlib#10720*] Eric Wieser. *chore(algebra/module/submodule): missing* `is_central_scalar` *instance*. Reviewed by Rob Lewis. Dec. 2021 (cit. on p. 49).

[*mathlib#11291*] Eric Wieser. *chore(topology/algebra/module/basic): add* `continuous_linear_map.is_central_scalar`. Reviewed by Oliver Nash. Jan. 2022 (cit. on p. 49).

[*mathlib#11297*] Eric Wieser. *feat(algebra,linear_algebra,ring_theory): more* `is_central_scalar` *instances*. Reviewed by Oliver Nash. Jan. 2022 (cit. on p. 49).

[*mathlib#12248*] Eric Wieser. *feat(measure_theory/function/ae_eq_fun): generalize scalar actions*. Reviewed by Rémy Degenne. Feb. 2022 (cit. on p. 49).

[*mathlib#12272*] Eric Wieser. *chore(topology/continuous_function/bounded): add an* `is_central_scalar` *instance*. Reviewed by Rob Lewis. Feb. 2022 (cit. on p. 49).

[*mathlib#12434*] Eric Wieser. *chore(topology/algebra/uniform_mul_action): add* `has_uniform_continuous_const_smul.op`. Reviewed by Oliver Nash. Mar. 2022 (cit. on p. 49).

[*mathlib#13710*] Eric Wieser. *chore(topology/continuous_function/zero_at_infty): add* `is_central_scalar` *instance*. Reviewed by Frédéric Dupuis. Apr. 2022 (cit. on p. 49).

[*mathlib#15359*] Eric Wieser. *chore(linear_algebra/alternating): add an `is_central_scalar` instance*. Reviewed by Rob Lewis. July 2022 (cit. on p. 49).

[mathlib#18682] Yaël Dillies. *feat(data/finset/pointwise): `a • (s ∩ t) = a • s ∩ a • t`*. Reviewed by Eric Wieser. Mar. 2023 (cit. on p. 49).

[*mathlib#10716*] Eric Wieser. *feat(algebra/algebra/basic): Algebras are bimodules*. Reviewed by Anne Baanen. Dec. 2021 (cit. on p. 50).

[*mathlib#7152*] Eric Wieser. *Algebras should imply both left and right actions*. Sept. 2023 (cit. on pp. 50, 178).

[*mathlib4#8909*] Eric Wieser. *feat(GroupTheory/GroupAction/Opposite): add notation for right and left actions*. Reviewed by Kyle Miller, Johan Commelin, Junyan Xu, and Yaël Dillies. Dec. 2023 (cit. on p. 50).

[mathlib4#5368] Yury G. Kudryashov. *feat: define a type synonym for right action on the domain of a function*. Reviewed by Eric Wieser and Sébastien Gouëzel. June 2023 (cit. on p. 50).

[*mathlib4#6487*] Eric Wieser. *refactor(Data/Matrix): Eliminate `·` notation in favor of `HMul`*. Reviewed by Jon Eugster and Oliver Nash. Aug. 2023 (cit. on p. 52).

[mathlib#104] Simon Hudon. *feat(tactic/ext): new `ext` tactic and corresponding `extensionality`…* Reviewed by Johannes Hölzl and Mario Carneiro. Apr. 2018 (cit. on p. 55).

[mathlib#7029] Greg Price. *chore(algebra/direct_sum_graded): golf proofs*. Reviewed by Eric Wieser and Scott Morrison. Apr. 2021 (cit. on p. 60).

[mathlib#3408] Chris Hughes. *feat(group_theory/semidirect_product): `mk_eq_inl_mul_inr` and `hom_ext`*. Reviewed by Scott Morrison. July 2020 (cit. on p. 61).

[mathlib#3531] Adam Topaz. *feat(linear_algebra/tensor_algebra): Tensor algebras*. Reviewed by Eric Wieser, Scott Morrison, Patrick Massot, Johan Commelin, and Chris Hughes. July 2020 (cit. on p. 61).

[mathlib#4078] Scott Morrison. *feat(algebra/ring_quot): quotients of noncommutative rings*. Reviewed by Eric Wieser, Kenny Lau, and Johan Commelin. Sept. 2020 (cit. on pp. 61, 114, 115).

[mathlib#4741] Yury G. Kudryashov. *chore(*): a few more type-specific ext lemmas*. Reviewed by Johan Commelin and Eric Wieser. Oct. 2020 (cit. on p. 61).

[*mathlib#5484*] Eric Wieser. *feat(group_theory/*): mark some lemmas as ext (about homs out of free constructions)*. Reviewed by Floris van Doorn. Dec. 2020 (cit. on p. 61).

[*mathlib#4297*] Eric Wieser. *feat(linear_algebra/exterior_algebra): Add an exterior algebra*. Reviewed by Anne Baanen and Scott Morrison. Sept. 2020 (cit. on p. 61).

[*mathlib#4821*] Eric Wieser. *feat(data/dfinsupp): Port over the* `finsupp.lift_add_hom` *API*. Reviewed by Johan Commelin. Oct. 2020 (cit. on p. 61).

[*mathlib#5640*] Eric Wieser. *feat(data/zsqrtd/to_real): Add* `to_real`. Reviewed by Johan Commelin, Mario Carneiro, Bryan Gin-ge Chen, and Anne Baanen. Jan. 2021 (cit. on p. 61).

[*mathlib#6105*] Eric Wieser. *refactor(linear_algebra/tensor_product): Use a more powerful lemma for ext*. Reviewed by Johan Commelin. Feb. 2021 (cit. on p. 61).

[*mathlib#6124*] Eric Wieser. *feat(linear_algebra/prod): add an ext lemma that recurses into products*. Reviewed by Johan Commelin. Feb. 2021 (cit. on p. 61).

[*mathlib#8105*] Eric Wieser. *feat(data/complex/module): add* `complex.alg_hom_ext`. Reviewed by Anne Baanen. June 2021 (cit. on pp. 61, 134).

[*mathlib#8641*] Eric Wieser. *feat(linear_algebra/basic, group_theory/quotient_group, algebra/lie/quotient): ext lemmas for morphisms out of quotients*. Reviewed by Oliver Nash and Anne Baanen. Aug. 2021 (cit. on p. 61).

[*mathlib#8783*] Eric Wieser. *feat(algebra/direct_sum): graded algebras*. Reviewed by Kevin Buzzard and Johan Commelin. Aug. 2021 (cit. on pp. 61, 63).

[*mathlib#10730*] Eric Wieser. *feat(linear_algebra/clifford_algebra/equivs): There is a clifford algebra isomorphic to the dual numbers*. Reviewed by Johan Commelin and Rob Lewis. Dec. 2021 (cit. on pp. 61, 135).

[*mathlib#10754*] Eric Wieser. *feat(algebra/triv_sq_zero_ext): universal property*. Reviewed by Johan Commelin. Dec. 2021 (cit. on pp. 61, 135).

[*mathlib#14803*] Eric Wieser. *feat(linear_algebra/clifford_algebra/of_alternating): extend alternating maps to the exterior algebra*. Reviewed by Oliver Nash. June 2022 (cit. on pp. 61, 166).

[*mathlib4#6417*] Eric Wieser. *feat(RingTheory/TensorProduct): heterogenize*. Reviewed by Johan Commelin and Antoine Chambert-Loir. Aug. 2023 (cit. on p. 62).

[*mathlib4#8116*] Eric Wieser. *feat(Data/Polynomial/AlgebraMap): more results for non-commutative polynomials*. Reviewed by Yaël Dillies and Johan Commelin. Nov. 2023 (cit. on p. 62).

[*mathlib#6053*] Eric Wieser. *feat(algebra/direct_sum_graded): endow* `direct_sum` *with a ring structure*. Reviewed by Rob Lewis, Anne Baanen, Scott Morrison, and Johan Commelin. Feb. 2021 (cit. on p. 63).

[*mathlib#9586*] Eric Wieser. *feat(algebra/graded_monoid): Split out graded monoids from graded rings*. Reviewed by Johan Commelin. Oct. 2021 (cit. on p. 63).

[mathlib#10115] Jujian Zhang. *feat(ring_theory/graded_algebra): definition of type class* `graded_algebra`. Reviewed by Eric Wieser, Kevin Buzzard, and Johan Commelin. Nov. 2021 (cit. on p. 63).

[*mathlib#10255*] Eric Wieser. *feat(linear_algebra/tensor_power): the tensor powers form a graded algebra*. Reviewed by Yaël Dillies, Anne Baanen, and Scott Morrison. Nov. 2021 (cit. on p. 73).

[*mathlib#9214*] Eric Wieser. *feat(linear_algebra/direct_sum):* `submodule_is_internal_iff_independent_and_supr_eq_top`. Reviewed by Johan Commelin, Scott Morrison, and Bryan Gin-ge Chen. Sept. 2021 (cit. on p. 75).

[mathlib#11750] Anne Baanen. *feat(*): define subobject classes from submonoid up to subfield*. Reviewed by Eric Wieser, Yaël Dillies, and Riccardo Brasca. Jan. 2022 (cit. on p. 75).

[lean4#2074] Kevin Buzzard. *Typeclass Inference Failure*. Jan. 30, 2023 (cit. on pp. 80, 88).

[lean4#777] Gabriel Ebner. *Definitional eta for structures*. Nov. 9, 2021 (cit. on p. 88).

[lean4#2210] Gabriel Ebner. *Skip proof arguments during unification, and try structure eta last*. May 15, 2023 (cit. on p. 88).

[*mathlib#14619*] Eric Wieser. *feat(linear_algebra/clifford_algebra/fold): Add recursors for folding along generators*. Reviewed by Oliver Nash. June 2022 (cit. on p. 99).

[*mathlib#14790*] Eric Wieser. *feat(linear_algebra/clifford_algebra/even): Universal property and isomorphisms for the even subalgebra*. Reviewed by Johan Commelin. June 2022 (cit. on p. 109).

[*mathlib#11468*] Eric Wieser. *feat(linear_algebra/clifford_algebra): the clifford algebra is isomorphic as a module to the exterior algebra*. Reviewed by Johan Commelin. Jan. 2022 (cit. on pp. 109, 120).

[*mathlib#17833*] Eric Wieser. *refactor(ring_theory/ideal/quotient): extract a* `ring_con` *structure*. Reviewed by Anne Baanen. Dec. 2022 (cit. on p. 114).

[mathlib#4079] Scott Morrison. *refactor(linear_algebra/tensor_algebra): build as a quotient of a free algebra*. Reviewed by Johan Commelin. Sept. 2020 (cit. on p. 115).

[*mathlib#6491*] Eric Wieser. *feat(linear_algebra/clifford_algebra): add definitions of the conjugation operators and some API*. Reviewed by Johan Commelin and Anne Baanen. Mar. 2021 (cit. on p. 119).

[*mathlib#6416*] Eric Wieser. *feat(linear_algebra/{clifford,exterior,tensor}_algebra): add induction principles*. Reviewed by Anne Baanen. Feb. 2021 (cit. on p. 119).

[mathlib4#7985] Richard Copley. *feat: Nontrivial instances for ExteriorAlgebra, CliffordAlgebra*. Reviewed by Eric Wieser. Oct. 2023 (cit. on p. 123).

[mathlib#16040] BillyMiao. *feat(linear_algebra/clifford_algebra/spin_group): Spin Group*. Reviewed by Eric Wieser and Utensil Song. Aug. 2022 (cit. on p. 123).

[mathlib4#9111] Utensil Song. *feat(LinearAlgebra/CliffordAlgebra): port SpinGroup*. Reviewed by Winston Yin, @grunweg, and Eric Wieser. Dec. 2023 (cit. on p. 123).

[*mathlib#11542*] Eric Wieser. *feat(linear_algebra/{tensor,exterior,clifford}_algebra): these algebras are graded by powers of the submodules of their generators*. Reviewed by Johan Commelin. Jan. 2022 (cit. on p. 124).

[*mathlib#4984*] Eric Wieser. *feat(group_theory/sub{monoid,group}): Add* `closure_induction'` *for subtypes*. Reviewed by Anne Baanen. Nov. 2020 (cit. on p. 124).

[*mathlib#11533*] Eric Wieser. *feat(algebra/algebra/operations): remove two hypotheses from* `submodule.mul_induction_on`. Reviewed by Oliver Nash. Jan. 2022 (cit. on p. 124).

[*mathlib#11556*] Eric Wieser. *feat(group_theory/sub{monoid,group}, linear_algebra/basic): add* `supr_induction` *for* `submonoid`, `add_submonoid`, `subgroup`, `add_subgroup`, *and* `submodule`. Reviewed by Johan Commelin. Jan. 2022 (cit. on p. 124).

[*mathlib#14219*] Eric Wieser. *feat(algebra/algebra/operations): add right induction principles for power membership*. Reviewed by Anne Baanen. May 2022 (cit. on p. 124).

[*mathlib#18146*] Eric Wieser. *feat(counterexamples/quadratic_form): symmetric bilinear forms in char 2 do not always inject into quadratic forms*. Reviewed by Johan Commelin. Jan. 2023 (cit. on p. 128).

[*mathlib4#14292*] Eric Wieser. *feat:* `BilinForm.toQuadraticForm` *is surjective in free modules*. June 2024 (cit. on p. 129).

[*mathlib#5722*] Eric Wieser. *feat(linear_algebra/{exterior,tensor,free}_algebra): provide left-inverses for* `algebra_map` *and* `ι`. Reviewed by Anne Baanen. Jan. 2021 (cit. on p. 129).

[*mathlib#15905*] Eric Wieser. *feat(algebra/monoid_algebra): add division by a generator*. Reviewed by Anne Baanen, Damiano Testa, and Kevin Buzzard. Aug. 2022 (cit. on p. 130).

[*mathlib#18633*] Eric Wieser. *feat(ring_theory/mv_polynomial/ideal): lemmas about monomial ideals*. Reviewed by Johan Commelin. Mar. 2023 (cit. on p. 130).

[*mathlib4#6657*] Eric Wieser. *feat(Counterexamples/CliffordAlgebra_not_injective): algebraMap is not always injective*. Reviewed by Kevin Buzzard, Riccardo Brasca, Mario Carneiro, Oliver Nash, and Johan Commelin. Aug. 2023 (cit. on p. 130).

[*mathlib4#9670*] Eric Wieser. *feat(Counterexamples/CliffordAlgebra_not_injective): Some quadratic forms cannot be constructed from bilinear forms*. Reviewed by Antoine Chambert-Loir and Moritz Doll. Jan. 2024 (cit. on p. 131).

[*mathlib#8107*] Eric Wieser. *feat(data/complex/module): add* `complex.lift` *to match* `zsqrtd.lift`. Reviewed by Filippo A. E. Nuccio and Anne Baanen. June 2021 (cit. on p. 134).

[*mathlib#8165*] Eric Wieser. *feat(linear_algebra/clifford_algebra): the reals and complex numbers have isomorphic real clifford algebras*. Reviewed by Anne Baanen. July 2021 (cit. on p. 134).

[*mathlib#8739*] Eric Wieser. *feat(linear_algebra/clifford_algebra/equivs): the equivalences preserve conjugation*. Reviewed by Johan Commelin. Aug. 2021 (cit. on pp. 134, 137).

[mathlib#5109] Kenny Lau. *feat(algebra/triv_sq_zero_ext): trivial square-zero extension*. Reviewed by Johan Commelin and Eric Wieser. Nov. 2020 (cit. on pp. 135, 138).

[*mathlib#8551*] Eric Wieser. *feat(algebra/quaternion_basis): add a quaternion version of* `complex.lift`. Reviewed by Johan Commelin and Anne Baanen. Aug. 2021 (cit. on p. 136).

[*mathlib4#9441*] Eric Wieser. *feat(Algebra/QuaternionBasis): extensionality for algebra morphisms from quaternions*. Reviewed by Oliver Nash. Jan. 2024 (cit. on p. 136).

[*mathlib#8670*] Eric Wieser. *feat(linear_algebra/clifford_algebra/equivs): there is a clifford algebra isomorphic to every quaternion algebra*. Reviewed by Johan Commelin. Aug. 2021 (cit. on p. 137).

[*mathlib4#9510*] Eric Wieser. *feat(Analysis/Calculus/DualNumber): Extending differentiable functions to dual numbers*. Jan. 2024 (cit. on p. 138).

[*mathlib#18384*] Eric Wieser. *refactor(algebra/{dual_number,triv_sq_zero_ext}): support non-commutative rings*. Reviewed by Oliver Nash and Jireh Loreaux. Feb. 2023 (cit. on p. 138).

[*mathlib#10729*] Eric Wieser. *refactor(algebra/triv_sq_zero_ext): generalize and cleanup*. Reviewed by Johan Commelin. Dec. 2021 (cit. on p. 139).

[*mathlib#18383*] Eric Wieser. *feat(algebra/dual_quaternion): two equivalent ways to construct the dual quaternions*. Reviewed by Johan Commelin. Feb. 2023 (cit. on p. 139).

[*mathlib4#7934*] Eric Wieser. *refactor(Algebra/DualNumber): generalize the universal property to non-commutative rings*. Reviewed by Yaël Dillies and Johan Commelin. Oct. 2023 (cit. on p. 139).

[*mathlib4#7962*] Eric Wieser. *feat:* `DualNumber (Quaternion R)` *as a* `CliffordAlgebra`. Oct. 2023 (cit. on p. 140).

[*mathlib4#14285*] Eric Wieser. *feat: `QuadraticForm.baseChange` is unique when it exists*. June 2024 (cit. on p. 142).

[*mathlib4#6306*] Eric Wieser. *feat(LinearAlgebra/BilinearForm/TensorProduct): base change of bilinear forms*. Reviewed by Oliver Nash. Aug. 2023 (cit. on p. 145).

[*mathlib#5430*] Eric Wieser. *feat(linear_algebra/tensor_product,algebra/module/linear_map): Made `tmul_smul` and `map_smul_of_tower` work for int over semirings*. Reviewed by Kevin Buzzard, Johan Commelin, and Kenny Lau. Dec. 2020 (cit. on p. 146).

[*mathlib#5317*] Eric Wieser. *feat(linear_algebra/tensor_product): Inherit smul through `is_scalar_tower`*. Reviewed by Johan Commelin and Bryan Gin-ge Chen. Dec. 2020 (cit. on p. 146).

[*mathlib#19143*] Eric Wieser. *feat(ring_theory/tensor_product): add missing scalar tower instances*. Reviewed by Riccardo Brasca. June 2023 (cit. on p. 147).

[*mathlib4#6035*] Eric Wieser. *feat: heterogenize TensorProduct.congr and friends*. Reviewed by Kevin Buzzard and Johan Commelin. July 2023 (cit. on pp. 148, 149).

[mathlib#15241] Andrew Yang. *feat(ring_theory/tensor_product): `A`-algebra structure on `A' ⊗[R] B`*. Reviewed by Eric Wieser and Riccardo Brasca. July 2022 (cit. on p. 149).

[*mathlib4#7409*] Eric Wieser. *feat(RingTheory/TensorProduct): the universal property of the tensor product of algebras*. Reviewed by Riccardo Brasca. Sept. 2023 (cit. on p. 151).

[*mathlib4#6778*] Eric Wieser. *feat: base change of Clifford algebras*. Reviewed by Oliver Nash. Aug. 2023 (cit. on p. 151).

[*mathlib4#6072*] Eric Wieser. *feat(Algebra/FreeAlgebra): support towers of algebras*. Reviewed by Oliver Nash. July 2023 (cit. on p. 151).

[*mathlib4#6066*] Eric Wieser. *feat: add scalar tower instances for RingQuot and BilinForm*. Reviewed by Oliver Nash. July 2023 (cit. on p. 151).

[*mathlib4#6073*] Eric Wieser. *feat(Algebra/TensorAlgebra): support towers of algebras*. Reviewed by Oliver Nash. July 2023 (cit. on p. 151).

[*mathlib4#6074*] Eric Wieser. *feat(LinearAlgebra/CliffordAlgebra): support towers of algebras*. Reviewed by Oliver Nash. July 2023 (cit. on p. 151).

[mathlib4#7569] Christopher Hoskin. *refactor(LinearAlgebra/QuadraticForm): Generalise `QuadraticForm` to `QuadraticMap`*. Reviewed by Eric Wieser. Oct. 2023 (cit. on p. 152).

[*mathlib4#6555*] Eric Wieser. *feat(LinearAlgebra/TensorProduct/Opposite): `Aᵐᵒᵖ ⊗[R] Bᵐᵒᵖ ≃ₐ[S] (A ⊗[R] B)ᵐᵒᵖ`*. Reviewed by Oliver Nash. Aug. 2023 (cit. on p. 154).

[*mathlib#10939*] Eric Wieser. *feat(quadratic_form/prod): quadratic forms on product and pi types*. Reviewed by Johan Commelin. Dec. 2021 (cit. on p. 156).

[*mathlib4#9141*] Eric Wieser. *feat: define* `QuadraticForm.IsOrtho` *as* `Q (x + y) = Q x + Q y`. Reviewed by Johan Commelin. Dec. 2023 (cit. on p. 156).

[*mathlib4#7866*] Eric Wieser. *refactor(Data/ZMod/IntUnitsPower): generalize ZMod 2 to work for* `Nat` *and* `Int` *too*. Reviewed by Jireh Loreaux. Oct. 2023 (cit. on p. 157).

[mathlib#5311] Frédéric Dupuis. *feat(linear_algebra/pi_tensor_product): define the tensor product of an indexed family of semimodules*. Reviewed by Eric Wieser, Johan Commelin, and Kevin Buzzard. Dec. 2020 (cit. on p. 161).

[*mathlib4#7394*] Eric Wieser. *feat: the graded tensor product*. Reviewed by Yaël Dillies and Rob Lewis. Sept. 2023 (cit. on p. 161).

[*mathlib4#7644*] Eric Wieser. *feat(LinearAlgebra/CliffordAlgebra): isomorphism for direct sums of vector spaces*. Reviewed by Oliver Nash and Mario Carneiro. Oct. 2023 (cit. on p. 162).

[*mathlib#5102*] Eric Wieser. *feat(linear_algebra): Add alternating multilinear maps*. Reviewed by Johan Commelin, Kevin Buzzard, Bryan Gin-ge Chen, and Anne Baanen. Nov. 2020 (cit. on p. 165).

[*mathlib#5136*] Eric Wieser. *feat(linear_algebra/multilinear): Add* `dom_dom_congr`. Reviewed by Bhavik Mehta and Johan Commelin. Nov. 2020 (cit. on p. 165).

[*mathlib#6708*] Eric Wieser. *chore(linear_algebra/determinant): redefine det using* `multilinear_map.alternatization`. Reviewed by Anne Baanen. Mar. 2021 (cit. on p. 165).

[*mathlib#5269*] Eric Wieser. *feat(linear_algebra/alternating): Add* `dom_coprod`. Reviewed by Gabriel Ebner, Johan Commelin, and Bryan Gin-ge Chen. Dec. 2020 (cit. on pp. 165, 166).

[mathlib#8576] Anatole Dedecker. *feat(analysis/normed_space/exponential): define exponential in a Banach algebra and prove basic results*. Reviewed by Patrick Massot. Aug. 2021 (cit. on p. 168).

[*mathlib#19244*] Eric Wieser. *refactor: Remove the K argument from exp*. Nov. 2023 (cit. on p. 168).

[*mathlib4#8370*] Eric Wieser. *refactor(Analysis/NormedSpace/Exponential): remove the* `𝕂` *argument from* `exp`. Nov. 2023 (cit. on p. 168).

[*mathlib#13444*] Eric Wieser. *feat(analysis/normed_space/exponential): Weaken typeclass requirements*. Reviewed by Frédéric Dupuis. Apr. 2022 (cit. on pp. 168, 171).

[*mathlib#13426*] Eric Wieser. *feat(topology/algebra/module/multilinear): relax requirements for* `continuous_multilinear_map.mk_pi_algebra`. Reviewed by Yury G. Kudryashov and Frédéric Dupuis. Apr. 2022 (cit. on p. 168).

[*mathlib#13987*] Eric Wieser. *feat(topology/algebra/monoid): add missing* `has_continuous_const_smul` *instances*. Reviewed by Sébastien Gouëzel. May 2022 (cit. on p. 168).

[mathlib#9379] Heather Macbeth. *chore(analysis/normed_space/*): prereqs for #8611*. Reviewed by Scott Morrison, Yury G. Kudryashov, Eric Wieser, and Oliver Nash. Sept. 2021 (cit. on p. 169).

[*mathlib#13497*] Eric Wieser. *feat(analysis/matrix): define the frobenius norm on matrices*. Reviewed by Johan Commelin. Apr. 2022 (cit. on p. 169).

[*mathlib#13518*] Eric Wieser. *feat(analysis/matrix): provide a* `normed_algebra` *structure on matrices*. Reviewed by Frédéric Dupuis and Yaël Dillies. Apr. 2022 (cit. on pp. 170, 171).

[*mathlib4#9476*] Eric Wieser. *feat(Analysis/Matrix):* `linfty_op_nnnorm` *agrees with the operator norm*. Reviewed by Johan Commelin. Jan. 2024 (cit. on p. 170).

[*mathlib#13544*] Eric Wieser. *fix(analysis/normed_space/basic): allow the zero ring to be a normed algebra*. Reviewed by Bryan Gin-ge Chen and Johan Commelin. Apr. 2022 (cit. on p. 170).

[*mathlib#13520*] Eric Wieser. *feat(analysis/normed_space/matrix_exponential): lemmas about the matrix exponential*. Reviewed by Frédéric Dupuis. Apr. 2022 (cit. on p. 171).

[*mathlib#13402*] Eric Wieser. *feat(analysis/normed_space/exponential): ring homomorphisms are preserved by the exponential*. Reviewed by Frédéric Dupuis and Johan Commelin. Apr. 2022 (cit. on p. 171).

[*mathlib#13488*] Eric Wieser. *feat(analysis/normed_space/exponential): add* `pi.exp_apply`. Reviewed by Johan Commelin. Apr. 2022 (cit. on p. 171).

[*mathlib#13489*] Eric Wieser. *feat(data/matrix/block):* `matrix.block_diagonal` *is a ring homomorphism*. Reviewed by Frédéric Dupuis. Apr. 2022 (cit. on p. 171).

[*mathlib#13534*] Eric Wieser. *feat(topology/uniform_space/matrix): Add the* `uniform_space` *structure on matrices*. Reviewed by Oliver Nash. Apr. 2022 (cit. on p. 171).

[*mathlib#13641*] Eric Wieser. *feat(topology/algebra/matrix):* `matrix.block_diagonal` *is continuous*. Reviewed by Johan Commelin. Apr. 2022 (cit. on p. 171).

[*mathlib#13815*] Eric Wieser. *chore(data/matrix/basic): add lemmas about powers of matrices*. Reviewed by Johan Commelin. Apr. 2022 (cit. on p. 171).

[*mathlib#13918*] Eric Wieser. *feat(data/matrix/block): add* `matrix.block_diag` *and* `matrix.block_diag'`. Reviewed by Johan Commelin. May 2022 (cit. on p. 171).

[*mathlib#13938*] Eric Wieser. *chore(data/matrix/basic): add more lemmas about* `conj_transpose` *and* `smul`. Reviewed by Johan Commelin. May 2022 (cit. on p. 171).

[*mathlib#13970*] Eric Wieser. *feat(data/matrix/basic): even more lemmas about* `conj_transpose` *and* `smul`. Reviewed by Anne Baanen. May 2022 (cit. on p. 171).

[*mathlib#13971*] Eric Wieser. *chore(analysis/normed_space/exponential): replace* `1/x` *with* `x⁻¹`. Reviewed by Yaël Dillies and Johan Commelin. May 2022 (cit. on p. 171).

[*mathlib#18199*] Eric Wieser. *feat(algebra/triv_sq_zero_ext): lemmas about pow, and ring structure*. Reviewed by Anne Baanen. Jan. 2023 (cit. on p. 172).

[*mathlib#19056*] Eric Wieser. *feat(analysis/calculus/fderiv/exp): derivative of* `exp R (A x)` *in non-commutative rings*. Reviewed by Anatole Dedecker. May 2023 (cit. on p. 172).

[*mathlib4#9487*] Eric Wieser. *feat: the exponential of dual numbers over non-commutative rings*. Jan. 2024 (cit. on pp. 172, 174).

[*mathlib#18200*] Eric Wieser. *feat(analysis/normed_space/triv_sq_zero_ext): exponential of dual numbers*. Reviewed by Jireh Loreaux, Damiano Testa, and Frédéric Dupuis. Jan. 2023 (cit. on p. 173).

[*mathlib#19049*] Eric Wieser. *feat(analysis/normed_space/triv_sq_zero_ext): generalize some results to non-commutativity*. Reviewed by Yaël Dillies and Scott Morrison. May 2023 (cit. on p. 173).

[*mathlib4#9491*] Eric Wieser. *feat(Analysis/NormedSpace/TrivSqZeroExt): generalize to topological spaces*. Reviewed by Johan Commelin. Jan. 2024 (cit. on p. 173).

[*mathlib4#9492*] Eric Wieser. *feat(Analysis/NormedSpace/TrivSqZeroExt): The L1 norm*. Reviewed by Jireh Loreaux and Johan Commelin. Jan. 2024 (cit. on p. 173).

[mathlib#2339] Yury G. Kudryashov. *feat(data/quaternion): define quaternions and prove some basic properties*. Reviewed by Johan Commelin, Bryan Gin-ge Chen, Scott Morrison, Mario Carneiro, and Rob Lewis. Apr. 2020 (cit. on p. 174).

[*mathlib#18347*] Eric Wieser. *feat(analysis/quaternion): add* `complete_space`, `module.free`, *and* `module.finite` *instances*. Reviewed by Yaël Dillies and Floris van Doorn. Feb. 2023 (cit. on p. 174).

[*mathlib#18352*] Eric Wieser. *feat(analysis/special_functions/trigonometric/series): express* `cos`/`sin` *as infinite sums*. Reviewed by Jireh Loreaux, Sébastien Gouëzel, Joseph Myers, and Johan Commelin. Feb. 2023 (cit. on p. 174).

[*mathlib#18349*] Eric Wieser. *feat(analysis/normed_space/quaternion_exponential): lemmas about the quaternion exponential*. Reviewed by Yaël Dillies, Jireh Loreaux, and Frédéric Dupuis. Feb. 2023 (cit. on p. 174).

[*mathlib#18413*] Eric Wieser. *chore(algebra/quaternion): missing trivial lemmas*. Reviewed by Yaël Dillies and Oliver Nash. Feb. 2023 (cit. on p. 174).

[mathlib4#5678] Yury G. Kudryashov. *feat: define continuous alternating maps*. Reviewed by Sébastien Gouëzel and Patrick Massot. July 2023 (cit. on p. 174).

[mathlib4#9718] Sophie Morel. *feat(LinearAlgebra/{ExteriorAlgebra,CliffordAlgebra}): Functoriality of the exterior algebra and some lemmas about generation*. Reviewed by Eric Wieser and Matthew Robert Ballard. Jan. 2024 (cit. on p. 177).

[mathlib4#9474] Jireh Loreaux. *feat: provide the ℓ^2 operator norm on matrices*. Reviewed by Eric Wieser. Jan. 2024 (cit. on p. 177).

[mathlib4#10079] Ali Ramsey. *feat(RingTheory): hopf algebra definition*. Reviewed by Eric Wieser, Scott Morrison, and Kevin Buzzard. Jan. 2024 (cit. on p. 177).

[lean4#2451] Matthew Robert Ballard. *RFC: tweak structure instance elaboration to avoid un-needed eta expansion*. Aug. 2023 (cit. on p. 179).

[lean4#2478] Matthew Robert Ballard. *feat: use supplied structure fields left to right and eta reduce terms in structure instance elaboration*. Aug. 2023 (cit. on p. 179).

[*lean4#2940*] Eric Wieser. *feat: Implement* `extends flat`. Nov. 2023 (cit. on p. 179).

[*lean4#2666*] Eric Wieser. *RFC:* `@[flat]` *annotation for names in the* `extend` *clause of a structure*. Oct. 2023 (cit. on p. 180).