PhD Dissertation

The Programming of Algebra

Mikkel Kragh Mathiesen

Advisor: Fritz Henglein
Co-advisor: Robin Kaarsgaard

2022-06-30

Abstract

We study how abstract algebra can be used as a foundation for programming, in particular for the purpose of query processing. The basic algebraic structures of rings, modules and algebras provide an expressive language for working with generalised multisets. Queries are expressed as linear maps between modules. The theory is presented categorically by means of universal properties and constitutes a promising replacement for classical relational algebra.

Many algebraic hallmarks turn out to be vital. Negative elements provide a uniform method for deleting data and multiplication expresses join of tables. Tensor products provide an efficient non-canonical representation of Cartesian products. Natural isomorphisms inexorably lead to efficient generalised trie representations of finite maps. Ultimately this begets highly efficient techniques for evaluation and in particular demonstrates worst-case output optimality for conjunctive queries.

Going further, we posit that linear algebra can be viewed as an excellent functional logic programming language. From this perspective a potential avenue of improvement becomes visible: finite-dimensional spaces are convenient to work with and—crucially—allow representing linear maps as elements of the tensor product, whereas infinite-dimensional spaces are less convenient in this regard but are necessary to model database schemata with infinite domains. We present a solution in the form of symbolic sums that provide a language of "compact-dimensional" linear algebra. The resulting language is called Algeo and comes with a syntax, type system and axiomatic as well as denotational semantics.

Resumé

Vi undersøger hvordan abstrakt algebra kan anvendes som et fundament for programmering, især i databaseforespørgselsøjemed. Basale algebraiske strukturer som ringe, moduler og algebraer giver anledning til et udtryksfuldt sprog, der kan håndtere generaliserede multimængder. Forespørgsler udtrykkes som lineære afbildninger mellem moduler. Teorien præsenteres kategorisk ved hjælp af universelle egenskaber og udgør en lovende erstatning for klassisk relationel algebra.

Mange algebraiske kendetegn viser sig at være afgørende. Negative elementer tilbyder en ensartet måde at slette data og multiplikation udtrykker sammenfletning af tabeller. Tensorprodukter tilbyder en effektiv ikke-kanonisk repræsentation af kartesiske produkter. Naturlige isomorfier fører uvægerligt til effektive generaliserede *trie*-repræsentationer af endelige afbildninger. I sidste ende afføder det særdeles effektive teknikker til evaluering og viser specielt at afviklingen af konjunktive forespørgsler er *worst-case output optimal*.

Vi hævder ydermere at lineær algebra kan anskues som et glimrende funktionslogikprogrammeringssprog. Med dette perspektiv bliver en forbedringsmulighed synlig: endeligdimensionale rum er belejlige at arbejde med og — helt afgørende — tillader at repræsentere lineære afbildninger som elementer af tensorproduktet, hvorimod uendeligdimensionale rum er forholdsvis mindre belejlige, men er uundværlige til at modellere databaseskemaer med uendelige domæner. Vi præsenterer en løsning i form af symbolske summer som giver anledning til et »kompaktdimensionelt« lineær algebra-sprog. Det resulterende sprog hedder Algeo og er udstyret med syntaks, typesystem og aksiomatisk såvel som denotationel semantik.

Dedication

獅

王

Contents

Notation

Categories	
$f : X \to Y$	f is an arrow between X and Y
$f : X \cong Y$	f is an isomorphism between X and Y
$\textsc{Set}$	Category of sets and functions
$\textsc{Ab}$	Category of Abelian groups and group homomorphisms
$\textsc{Mod}_K$	Category of K-modules and linear maps
$F \dashv G$	Adjunction
Sets	
$A + B$	Disjoint union of sets
$A \times B$	Cartesian product of sets
Modules	
$\mathbf{0}$	Zero module
$U \oplus V$	Biproduct
$U \otimes V$	Tensor product
$\mathbf{F}_K[A]$	Free module
$\mathbf{F}^*_K[A]$	Compact free module
$A \Rightarrow U$	Finite map module
$A \Rightarrow^* U$	Compact map module
$x + y$	Addition
$r \cdot y$	Scalar multiplication
$x \cdot y$	Algebra multiplication
$\#x$	Weight
$x \diamond y$	Inner product
$f^\dagger$	Adjoint
$\{a\}$	Singleton generalised multiset
$*$	Wildcard
Algeo	
x	Metavariable
foo	Concrete variable
τ	Type
a	Atom
b	Base value
d	Duplicable value
e	Expression

$\overline{n}$	Numeric constant
Empty	The empty type
Atom	The type of atoms
Scalar	The type of scalars
$\tau_1 \to \tau_2$	Function type
$\tau_1 \oplus \tau_2$	Sum type
$\tau_1 \otimes \tau_2$	Product type
$x : \tau$	x has type τ
$x^{y:=e}$	x with e substituted for y
$e_1\ e_2$	Function application
$e_1; e_2$	Biased conjunction
$\mathbf{inl}(e)$	Left injection into sum
$\mathbf{inr}(e)$	Right injection into sum
$e_1 \otimes e_2$	Pair
$e_1 \mapsto e_2$	Singleton function
$\emptyset$	Failure/zero
$e_1 \parallel e_2$	Choice/union/sum
$e_1 \bowtie e_2$	Join/intersection/product
$[x : \tau]\, e$	Symbolic sum/variable introduction
$x \Leftrightarrow y$	Comparison/unification/inner product
$x \setminus\!\!\setminus y$	Difference/subtraction
$*_\tau$	Wildcard/sum of all base values of τ
$e^\perp$	Complement
Query Processing	
$(A_1 : a_1, \ldots, A_n : a_n)$	Tuple with value a_i for attribute A_i as used in relational algebra
$\mathbf{cp}_0$	Isomorphism witnessing $0 \Rightarrow U \cong \mathbf{0}$
$\mathbf{cp}_+$	Isomorphism witnessing $(A + B) \Rightarrow U \cong (A \Rightarrow U) \oplus (B \Rightarrow U)$
$\mathbf{cp}_1$	Isomorphism witnessing $1 \Rightarrow U \cong U$
$\mathbf{cp}_\times$	Isomorphism witnessing $(A \times B) \Rightarrow U \cong A \Rightarrow B \Rightarrow U$
Simplification	
1_P	Indicator function, equal to 1 when P is true, otherwise 0
$R^\dagger$	Shallow lookup function for the trie R
$R^\ddagger$	Deep lookup function for the trie R
$R \bowtie S$	Join of R and S, equal to $R \cdot S$
$w(R)$	R with an initial wildcard layer, i.e. $* \mapsto R$
$\hat{f}$	Functorial action of $(A \Rightarrow^* -)$ on f
$\deg R$	Degree of R
$\lvert R \rvert$	Cardinality of R
∂R	Collapse of R
$\langle R \rangle$	Shallow padding of R
$\langle\!\langle R \rangle\!\rangle$	Deep padding of R
$\omega_{C_1,\ldots,C_n}(r_1, \ldots, r_n)$	Worst-case output size

Chapter 1

Introduction

The disciplines of algebra and programming are both mature and interactions between them have been studied for decades in various shapes and forms. What could we possibly hope to offer by combining them yet again? It turns out that they are more closely related than one might think. The quippy version is that linear algebra *is* a programming language, and a very expressive and efficient one to boot.

Previous marriages between algebra and programming fall into two major camps. The first camp studies the structure of programs and use equational reasoning to prove properties and derive optimisations. This is exemplified by the book "Algebra of Programming" [1] whose name the title of this dissertation is a pun of. The second camp studies how to make linear algebra—specifically matrix algebra—computations more convenient to program with. Typically this research results in array languages, array libraries, database extensions with vector types, etc.

Our project ultimately has little overlap with either of these approaches aside from fighting for the same namespace. We focus on some of the most classic of algebraic structures: rings and modules/vector spaces. The basic insight is that modules (and extensions thereof) are not merely objects to be manipulated by existing languages. Module theory—when viewed from the right angle—is actually a capable functional logic language which is particularly well suited for query processing.

This sets the stage, so who are the actors? We will meet the humble "$+$", which is commutative and has an inverse; it plays the rôle of union in our generalisation of set theory. Later we encounter the indomitable "$\otimes$", which is bilinear; it plays the rôle of Cartesian product, but with a twist. The sidekick is the amorphous "$\cdot$", with its many bilinear faces; it plays the rôle of counting and intersection.

We proceed to give a brief tour of the broad strokes of this dissertation.

1.1 Whirlwind Tour

Basics. A ring is a set that behaves roughly like the integers: there are operations "$+$" and "$\cdot$" satisfying properties of associativity, distributivity, etc. A module (or vector space) over a ring R is a set that behaves roughly like R^n: elements can be added and multiplied by *scalars* from R. An algebra over R is a module over R that allows multiplication of arbitrary elements.

Algebraic structures. Modules and algebras in particular will be our focus guided by the insight that elements in a module can be viewed as generalised multisets. Addition corresponds to union and multiplication from the algebra structure serves as intersection. These two operations do not form a lattice as would usually be expected by generalisations of set theory. Instead, any element has an additive inverse which is fundamentally incompatible with idempotence. Negative elements can be thought of as deletion, but due to commutativity of addition it does not matter whether an element is removed before or after it is added.

Synthetic modules. We take a synthetic approach and only concern ourselves with modules that can be built using a handful of constructions. The ring itself is a module. For any set A we can form the *free module* generated by elements $\{a\}$ for each $a \in A$; this is the space of "generalised multisets". Two modules can be combined using $\oplus$ or $\otimes$. The former is a *biproduct* and corresponds to a disjoint union of sets. The latter is a *tensor product* and corresponds to a Cartesian product of sets. Ultimately, all these constructions yield modules that are isomorphic to some free module. Nevertheless, keeping in mind the *intensional* structure of modules turns out to be very beneficial and we should be reluctant to actually apply this isomorphism.

Category theory. Generally, we take a (light) categorical approach and describe all constructions using *universal properties*. This has several advantages: it clarifies how the object behaves and how it can be used, and it makes it possible to postpone the choice of a specific representation. In other words, universal properties describe the *interfaces* of objects at a level of abstraction that allows working with the object while making sure that we do not accidentally depend on the particularities of arbitrary choices.

Pattern matching. When appealing to universal properties we develop a suggestive notation of *patterns* and *copatterns*. For instance, linear maps out of a free module can be written as $f(\{a\}) = \ldots$ by pattern matching on the generator $\{a\}$. When f is applied to a linear combination we appeal to the fact that f is linear *by construction*.

Following this line of development we develop the notion of the *programming of algebra*, the idea that common algebraic structures by themselves provide a programming language. Revealing this connection is a matter of acquiring a suitable notation and perspective.

Programming. Treating syntax formally and giving it a precise semantics, we veer into the territory of programming languages. This gives a formal justification for our notation, but it also naturally leads to ideas about what sort of programs should exist, including programs that are not immediately apparent from the purely algebraic perspective.

One idea is that programs can be run backwards. Many interesting linear maps are isomorphisms and most of the useful isomorphisms are in fact *unitary*, meaning that the inverse coincides with the *adjoint*. Adjoints can be thought of as a "best effort" reverse execution. They are easier and more formulaic to construct than inverses.

Symbolic sums. In standard modules adjoints do not always exist. For instance, given the free module generated by an infinite set the linear map that sends every generator to the scalar 1 has no adjoint. Such an adjoint would have to produce a sum of infinitely many elements.

Extending the language to permit such adjoints we eventually arrive at the idea of symbolic sums, terms of the form $[x : M] \ldots$ for any module M. These terms obey all the usual equalities for sums, but cannot necessarily be simplified. The resulting theory can be thought of as "compact-dimensional linear algebra", since it deals with modules of any dimension while still getting most of the advantages in dealing with finite-dimensional modules, such as an isomorphism between linear maps and tensor products.

Extensional definitions. Eventually, we arrive at a very flexible system for making definitions. To define an object of type τ we take the (symbolic) sum over all τ and impose constraints to filter the desired values. These constraints may be "equations" by using the inner product, but this is merely a matter of convention and practicality. Thus, every definition is extensional in the sense of defining objects by their behaviour.

Query processing. The module structure provides the foundation for query processing. All of traditional relational algebra can be recast using linear maps. Many operations follow from the universal property of free modules: a linear map out of the free module over some set A into another module M is uniquely determined by choosing an element of M for each element of A. If M is also a free module (over some set B) then we can lift any map of sets between A and B to a linear map between the free modules. Using 0 we can also lift any partial function, which suffices to define selection.

A key difference arises when considering projection, since the linear version will preserve multiplicities. For instance, in relational algebra projection the first component of $\{(a, b), (a, c)\}$ gives just $\{a\}$ whereas in linear algebra the result becomes $2 \cdot \{a\}$ to account for the two occurences of a.

We also extend the query language with wildcards. The wildcard $*$ is the unit of the join operator. It is superficially similar to null values from traditional databases, but the wildcard is compatible with *any* value wheras null is compatible with *no* value (not even itself). Having wildcards is convenient—often moreso than nulls—and its rôle as a unit for the join operator avoids issues regarding queries with unconstrained variables.

Conjunctive queries. A well-known class of queries studied in database theory is *conjunctive queries*. Such a query consists of a conjunction of primitive relations (i.e. tables from the database schema) and involves a number of variables. In practice this kind of query arises when joining tables.

Many different algorithms have been developed to evaluate such queries with varying efficiency. The problem is hard in general, and seems to be so fundamentally as SAT-like problems can be encoded. In order to evaluate the relative merits of algorithms complexity measures have been proposed to assess the hardness of a query based on its structure. The relationship between variables and relations in which they occur gives rise to a hypergraph and a measure called the *fractional edge cover* can be used to give sharp bounds on the possible size of outputs of the corresponding query. Algorithms whose runtime is bounded by this worst-case output size are known as *worst-case output optimal*.

When the hypergraph has a cycle the query is called *cyclic*, and these are recognised as particularly hard instances. For instance, given three binary relations Traditional approaches based on constructing a query plan to decide in which order to eliminate the relations are provably suboptimal for cyclic instances. Fairly recent developments have resulted in algorithms that also deal well with cycles.

Algebraic evaluation. Given a query expression, how do we compute the result? This is achieved by *simpliying* the expression until it has a form that can be directly observed as a sum of primitives (e.g. terms of the form $\{a\}$ for the free module). Strategies for simplification of expressions correspond to strategies for evaluating conjunctive queries.

Starting from universal properties and the natural isomorphisms that follow from those, we arrive naturally at an efficient strategy. Certain forms of expressions can be recognised as *generalised tries*, but in our theory this is nothing more than a direct application of the most "obvious" tools at any given point. This approach is largely type-directed, which is yet another reason why flattening every module to a free one would be a bad idea. Effectively, simplification proceeds by eliminating one variable at a time—rather than one relation at a time like traditional query plans—which is similar to how the DPLL algorithm for SAT solving works.

Optimality. The upshot is that algebraic simplification turns out to be worst-case output optimal, which can be seen by an argument that considers inputs padded with extra shadow values. These extra values only increases the input size by a constant factor, but allows each piece of the output to be ascribed to a piece of the input. This argument works directly with the definition of worst-case output size and does not depend on any graph theory, nor does it require computing the actual bounds. The result proven is also strictly more general due to the possible presence of wildcards in the input.

1.2 Attributions

This dissertation is based on and includes material from the following works:

- "Infinite-dimensional Linear Algebra for Efficient Query Processing" [2] (**MT**). Master's thesis.
- "Module Theory and Query Processing" [3] (**MTQP**) together with Fritz Henglein. Rôle: lead author.
- "Polylogic" (**PL**). Rôle: sole author.
- "The Programming of Algebra" [4] (**PoA**) together with Fritz Henglein and Robin Kaarsgaard. Rôle: lead author.
- "Combinatory Adjoints and Differentiation" [5] (**CAD**) together with Martin Elsman, Fritz Henglein, Robin Kaarsgaard and Robert Schenk. Rôle: collaborator.
- "Algeo: An Algebraic Approach to Reversibility" [6] (**AAAR**) together with Fritz Henglein and Robin Kaarsgaard. Rôle: lead author.
- "Worst-case Optimal Algebraic Joins" (**WOAJ**) together with Fritz Henglein. Unpublished. Rôle: lead author.

The dependencies are as follows for each chapter.

- Chapter 2 is mostly based on the first part of **PoA**. The presentation of algebraic structures in the beginning is new.
- Chapter 3 is an adaptation of **AAAR**. The motivation in the beginning as well as the denotational semantics are original to this dissertation.
- Chapter 4 is an adaptation of **PoA**. The motivation in the beginning is entirely new.
- Chapter 5 is an adaptation of **WOAJ** with parts from **PoA** and some additions original to this dissertation.
- Chapter 6 is an adaptation of **MT**. Changes mostly consist of updating notation and removing dependencies on the rest of **MT**.
- Chapter 7 incorporates parts of the related work and discussion sections of **PoA**, **AAAR** and **WOAJ**.

1.3 Contributions

We now list the contributions made in this work. Each contribution is marked with its origin as either coming from one of the publications above or from this dissertation—marked (**D**)—in which case it has not been reviewed yet.

- Linear algebra as a query language (**MTQP**, **PoA**).
  - An algebraic framework based on universal constructions suitable for interpreting database operations (**MTQP**, **PoA**).
  - A "Rosetta Stone" showing how to translate relational algebra into this framework (**PoA**).
  - Extension from semirings to rings (**MTQP**, **PoA**).
  - Symbolic tensor products for efficient representation of Cartesian products (**MTQP**, **PoA**).
  - Algebra multiplication as a join operator (**PoA**).
  - Wildcard elements that function as a unit for the join operator and allow more flexible database modelling such as cofinite sets (**PoA**).
  - Efficient evaluation of queries based on careful simplification of symbolic terms (**PoA**).
  - A working implementation in Haskell that outperforms existing databases on certain queries (**MT**).
- A novel worst-case output optimal algorithm for conjunctive queries (**WOAJ**).
  - An interpretation of join as a combination of embedding and intersection (**PoA**, **WOAJ**).
  - A proof technique for showing worst-case output optimality based on input padding (**WOAJ**).
  - An extension of said algorithm and its optimality proof to deal with wildcards (**WOAJ**).
- The Algeo language (**AAAR**).
  - An algebraic view of functional logic programming (**AAAR**).
  - Symbolic sums for expressing "compact-dimensional" linear algebra (**AAAR**).
  - A very flexible system for giving definitions, subsuming most forms of pattern matching and more (**AAAR**).
  - An axiomatic semantics (**AAAR**).
  - A denotational semantics (**D**).

We do not include any results from **CAD**. However, after perusing this dissertation the reader should be well equipped to read the paper itself.

1.4 Overview

Chapter 2 provides the necessary background knowledge. This mostly consists of a presentation of linear algebra through the lens of category theory. The approach is different from almost all other introductions to the subject, so even a well-versed reader should not skip this section lightly.

Chapter 3 introduces the Algeo language, which provides a formal foundation for the notation developed in Chapter 2. We illustrate the language and give syntax and semantics.

Chapter 4 demonstrates how linear algebra can serve as a query language and how it compares to relational algebra.

Chapter 5 discusses the data structures and algorithms involved in our linear algebra computations. We show how to simplify terms and argue that this simplification procedure satisfies the technical criterion of worst-case output optimality for conjunctive queries.

Chapter 6 presents an approach for implementing an algebraic framework suitable for query processing in Haskell.

Chapter 2

Preliminaries

2.1 Motivation

The main tool for our purposes is linear algebra. Linear algebra has many faces, however, and we will focus on one of them: the theory of modules and linear maps between them viewed through the lens of category theory. In particular, we are not concerned with the "classical" conception of linear algebra as a synonym for matrix algebra; alternatively as the theory of $\mathbb{R}^n$ or $\mathbb{C}^n$.

The categorical approach insists that the objects of study (e.g. modules) are abstract and described only by their behaviour and relationship with each other. Objects can be combined to form new objects, but the exact nature of the construction is left unspecified as much as possible. For instance, we will be able to form the module $M \oplus N$ for any modules M and N. All we need to know about this new module is that there exist injections from and projections onto M and N satisfying certain equations. Compare this with the classical approach: given natural numbers m and n we can consider $\mathbb{R}^m$ and $\mathbb{R}^n$ as well as $\mathbb{R}^{m+n}$. It is possible to define injections and projections by means of index calculations, but there are many ways to do it and none of them are obviously superior or canonical. Yet to proceed we must arbitrarily choose one of them. Care then needs to be taken not to depend on the specifics of the chosen index manipulation scheme. In practice these details are sometimes dismissed and the objects are treated more abstractly with an informal understanding of what is allowed and what is not.

Category theory provides a framework for dismissing details while remaining on a solid mathematical foundation. Each category can be thought of as a mathematical world which determines what we are allowed to do and observe. The very rules of the game prohibit prevent us from depending on details like index manipulation (unless we have explicitly made it part of the structure under consideration).

At the same time, we are interested in computational aspects of our theory. A categorical approach is naturally implementation-agnostic, which gives us complete freedom to choose a strategy. It turns out, however, that a straightforward symbolic representation of the abstract structure is surprisingly efficient. In particular, the derived representation for the tensor product $M \otimes N$ would be nigh impossible to invent by just considering how $\mathbb{R}^m$ and $\mathbb{R}^n$ can be mapped into $\mathbb{R}^{mn}$. We shall also see how a worst-case optimal algorithm for conjunctive queries arises naturally from the abstract structure. Indeed, the categorical avoidance of "premature implementation" often leads to more efficient algorithms and data structures.

2.2 Prerequisites

The reader should be familiar with the idea of algebraic structures such as Abelian groups and rings. Basic category theory is also assumed; alternatively, a good foundation in algebra should be sufficient as well.

Knowledge of linear algebra is not required, as we will introduce the necessary concepts. In fact, too much familiarity with classical linear algebra might even be unhelpful! In that case one should be wary of trying to reduce everything to questions about matrices.

Regarding databases, a passing familiarity should be enough for the most part. The technical discussion of worst-case output optimality might require following references first.

2.3 Structures

We now define the various algebraic structures we will need.

Abelian group. An *Abelian group* A comprises

- A set $|A|$.
- An element $0_A : |A|$.
- A binary operator $(+) : |A| \to |A| \to |A|$.
- A unary operator $(-) : |A| \to |A|$.

subject to the following identities:

- $0_A + x = x$ (identity).
- $x + y = y + x$ (commutativity).
- $(x + y) + z = x + (y + z)$ (associativity).
- $x + (-x) = 0_A$ (inverse cancellation).

Abelian groups include $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ with the usual addition operations. Another notable example is $\mathbb{Z}_m$, integers module m.

Ring. A (commutative) *ring* R is an abelian group that additionally comprises

- An element $1_R : |R|$.
- A binary operator $(\cdot) : |R| \to |R| \to |R|$.

subject to the following identities:

- $1_R \cdot x = x$ (identity).
- $x \cdot y = y \cdot x$ (commutativity).
- $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ (associativity).
- $0_R \cdot x = 0_R$ (cancellation).
- $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$ (distributivity).

All of the examples of Abelian groups given above are in fact also rings with their usual multiplication operations.

Module. Fix a ring R. An R-module M is an abelian group that additionally comprises

- A binary operator $(\cdot) : |R| \to |M| \to |M|$.

subject to the following identities:

- $1 \cdot x = x$ (identity).
- $(r \cdot s) \cdot x = r \cdot (s \cdot x)$ (associativity).
- $0_M \cdot x = 0_M$ (cancellation).
- $(r + s) \cdot x = (r \cdot x) + (s \cdot x)$ (distributivity).

Classical examples of modules include R^n for any ring R with componentwise addition. See Section 2.6 for an overview of the landscape of modules.

Algebra. Fix a ring R. A (commutative) R-algebra M is a module that additionally comprises

- A binary operator $(\cdot) : |M| \to |M| \to |M|$.

subject to the following identities:

- $x \cdot y = y \cdot x$ (commutativity).
- $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ (associativity).
- $0 \cdot x = 0$ (cancellation).
- $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$ (distributivity).

Algebras arise whenever we have a subring R of a ring S in which case S is an R-algebra. Many of the modules that we construct will also have an algebra structure.

Unital algebra. Fix a ring R. A (commutative) unital R-algebra M is an algebra that additionally comprises

- An element $1_M : |M|$.

subject to the following identities:

- $1_M \cdot x = x$ (identity).
- $r \cdot 1_M = 1_M$ (scalar compatibility).

Weighted module. Fix a ring R. A (commutative) weighted R-module M is a module that additionally comprises

- An operator $(\#) : M \to R$.

subject to the following identities:

- $\#0_M = 0_R$ (cancellation).
- $\#(x + y) = \#x + \#y$ (additive distributivity).
- $\#(r \cdot x) = r \cdot \#x$ (multiplicative distributivity).

We will also speak of weighted R-algebras for modules that have both multiplication and weight operators.

General comments.

- When R is a field (i.e. multiplicative inverses exist for nonzero elements) the term *vector space* is often used in place of module. We shall use *module* and *vector space* interchangeably without regard for the nature of R. Similarly, we use *linear algebra* in the broad sense of studying linear maps between structures and not just the subset concerned with classical vector spaces.
- The definitions above are not the most economical, but they have the virtue of making every operation and identity clear. More efficiently, we could define e.g. an algebra as a monoid object in the category of modules.
- We have been explicit about distinguishing the object X from its *carrier*, the set $|X|$. Generally, we will use X to mean either X or $|X|$ when the intent should be clear.
- Only commutative operations are considered, so we shall simply use *ring* to describe what would normally be called a *commutative ring*. The same goes for commutativity of algebras.
- Similarly, what we call *modules* are sometimes called *two-sided modules*, since scalars can also be multiplied on the right by defining $x \cdot r = r \cdot x$. Commutativity of the ring ensures that this works exactly the same as multiplication on the left.

- The $(\cdot)$ operator is overloaded, being used for multiplication in both rings, modules and algebras. Furthermore, we will employ the usual convention that juxtaposition xy means $x \cdot y$. This overloading has two justifications. Firstly, the three operations have different types so disambiguating is always possible. Secondly, there are sufficient identities to ensure that the operations are essentially the same on their common domain.
- Similarly, the subscripts for 0 and 1 will generally be omitted.
- An Abelian group is actually a special case of a module, namely a $\mathbb{Z}$-module. Similarly, a ring is just a unital $\mathbb{Z}$-algebra. Rings must be defined before modules and algebras, though, since the definitions of modules and algebras are parameterised by a ring.

2.4 Category Theory

We now recall some of the basic definitions of category theory.

Category A *category* $\mathfrak{C}$ comprises a collection of objects and for each pair of objects a collection of arrows. We write $X \in \mathfrak{C}$ when X is an object in $\mathfrak{C}$ and $f : X \to Y$ when f is an arrow between objects X and Y. For each object X there should be an identity arrow $\mathrm{id}_X : X \to X$ and any two arrows $f : Y \to Z$ and $g : X \to Y$ can be composed into $f \circ g : X \to Z$. We require that $\circ$ is associative and that id_X is an identity for $\circ$. An arrow $f : X \to Y$ such that $g \circ f = \mathrm{id}_X$ and $f \circ g = \mathrm{id}_Y$ for some $g : Y \to X$ is called an *isomorphism*.

All of the structures we have defined above form categories. As an example take the category Ab of Abelian groups. For any two abelian groups G and H, the arrows between them consist of group homomorphisms, i.e. functions $f : G \to H$ between the underlying sets such that $f(x + y) = f(x) + f(y)$ for all $x, y : G$. In principle we also demand that $f(0) = 0$ and $f(-x) = -f(x)$ (so that f respects *all* the structure of Abelian groups) but these properties are consequences of the previous one, so we do not need to stipulate them directly.

Since arrows in Ab respect the group structure, category theoretic statements in Ab will necessarily be of a purely group theoretic nature and not depend on any particular representational details. For instance, the set $\{a, b\}$ can be made into an (Abelian) group in two ways depending on whether $a+b = a$ or $a + b = b$, although they are the same group *up to isomorphism*. In classical group theory we take great care not to make any statements that could possibly distinguish isomorphic groups. In the category Ab it is quite simply impossible to make any such distinction. Hence, we may think of Ab as “Abelian group theory”.

Functor The notion of a category is a bit too involved to be considered an algebraic structure in the usual sense (i.e. a set equipped with operations obeying axioms), but it nevertheless bears similar consideration. In particular, we want to work with mappings that respect the structure of categories. Such a mapping $F : \mathfrak{C} \to \mathfrak{D}$ is known as a functor. It maps objects to objects and arrows to arrows such that $F(a) : F(X) \to F(Y)$ for every arrow $a : X \to Y$. Respecting the structure entails $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ and $F(a \circ b) = F(a) \circ F(b)$.

For instance, let $F : \textsc{Set} \to \textsc{Ab}$ which maps a set X to the Abelian group whose elements are sum of elements of X subject to all axioms of an Abelian group. Given $f : X \to Y$ the function $F(f) : F(X) \to F(Y)$ maps each component of a sum using f.

Another example is $| \cdot | : \textsc{Ab} \to \textsc{Set}$ which maps each Abelian group to its underlying set, and each homomorphisms to its underlying function.

Adjunctions The two functors F and $|\cdot|$ form what is known as an *adjunction*, written $F \dashv |\cdot|$. In general we require that there is a bijection between $F(X) \to G$ and $X \to |G|$ which is natural (see below) in X and G. For the concrete example this means that defining a homomorphism from the Abelian group generated by X into some group G is equivalent to defining an ordinary map from X into the elements of G. This expresses the idea that F *freely* generates a group, since the generators can be mapped independently of each other. The other direction, $|\cdot|$, is said to be *forgetful* since $\mathfrak{D}$ will usually have more structure than $\mathfrak{C}$ and $|\cdot|$ simply gets rid of the additional structure (e.g. group operations and axioms).

Note that adjoint functors are not inverses. Indeed, in our example $|F(X)|$ is the set consisting of sums of elements of X which is always a larger set when X is finite. Similarly, $F(|G|)$ is in general a different group than just G.

Natural Transformations Given functors $F, G : \mathfrak{C} \to \mathfrak{D}$, a *natural transformation* $\alpha : F \to G$ is a family of arrows $\alpha_X : F(X) \to G(X)$ for each $X : \mathfrak{C}$. Naturality stipulates that $G(f) \circ \alpha_X = \alpha_Y \circ F(f)$ for any $f : X \to Y$. Normally this property is satisfied when α is defined generically for some abstract object X.

For instance, for a given set Y let $\alpha_X : X \times Y \to X$ be the projection onto the first component. This transformation is natural.

2.5 Categories Under Consideration

The two most important categories in our investigation are $\textsc{Set}$ and $\textsc{Mod}_K$ (for some K). Objects in $\textsc{Set}$ are sets and arrows are ordinary maps. Objects in $\textsc{Mod}_K$ are K-modules and arrows are homomorphisms, i.e. maps between the underlying sets that respect all the module operations. It is in fact sufficient to check that $f(x + y) = f(x) + f(y)$ and $f(k \cdot x) = k \cdot f(x)$. Note that this is similar to the situation for Abelian groups and indeed $\textsc{Ab}$ can equivalently be defined simply as $\textsc{Mod}_{\mathbb{Z}}$.

The following discussion appeals to categorical concepts that we will not formally introduce. There are fundamental differences between (bi-)cartesian closed (roughly, set-like) categories and (pre-)additive (roughly, module-like) categories. Intuitively, the former models classical computation and the latter models computation with choice and failure such as nondeterminism, probabilistic programming or query processing. The two notions are incompatible in the sense that any distributive additive category collapses to a single object (since $X \cong X + 0 \cong X \times 0 \cong 0$ for any X).

In a distributive category products and coproducts can be thought of as abstractions of the usual cross product and disjoint union operations on sets. In an additive category finite products and coproducts coincide and are known as biproducts. A product can be thought of as a pairing where the consumer chooses which component to work with; conversely a coproduct is a pairing where the producer chooses the component. A biproduct is then a pairing where the producer and the consumer have equal say in choosing a component (if they disagree the computation simply fails). This behaviour of biproducts is why arrows in additive categories often have pseudo-inverses, since they are fundamentally more like relations than functions.

The conflation of products and coproducts does leave additive categories without a traditional “proper” product type. However, there is often a notion of tensor product (an enriched monoid structure on objects) which fills this gap and interacts with the biproduct structure the same way a product interacts with the coproduct in a distributive category. Tensor products are much weaker and do not provide e.g. first and second projections.

Another important difference concerns how data is copied and deleted. In a distributive category copying is accomplished trivially since pairs are represented as products which by their very nature support duplication. Similarly for deletion. In an additive category where pairs are represented by some tensor product there is no general (natural) way to copy or delete, but particular objects may support it. Often there are many ways to define a copying operation, none of which are canonical. This is also the reason why additive categories appear in categorical treatments of quantum computation.

2.6 Meet the Modules

With the definitions above in hand it is possible to judge for a given set equipped with operations whether it is a module. This is the *analytic* approach. By contrast, we take a *synthetic* approach and present a number of objects that are modules *by construction.* It is by no means a complete set of constructions, but it covers everything that we will need.

Each module is presented with a *universal property*, describing its rôle in the category of modules. The basic principle is that objects in a category can be understood purely by how they relate—via morphisms, in this case linear maps—to other objects. One perspective is that universal properties define *interfaces* to allow abstracting away irrelevant details.

Trivial modules. The simplest module is the zero module $\mathbf{0}$ consisting of just a single 0 element. All operations are trivial and determined by the axioms. It satisfies the following universal property: for all modules U there is a unique linear map $0 : \mathbf{0} \to U$, and a unique linear map $0 : U \to \mathbf{0}$.

The first interesting example of a module is the ring K itself with operations inherited from the ring structure. This is called the *scalar* module. It also has a weighted algebra structure with multiplication inherited from the ring and weight given by the identity map. The scalars satisfy the universal property that for any module U and element $u : U$ there exists a unique linear map $f : K \to U$ such that $f(1) = u$.

Intuitively the property states that any linear map from K is completely determined by what it does to 1. We are therefore justified in defining linear maps by pattern matching on 1, for example defining some $f : K \to K$ by $f(1) = 42$. By linearity $f(r) = f(r \cdot 1) = r \cdot f(1) = r \cdot 42$ for any r, so we have exactly the information we need to determine the value of f at any point.

Free modules. The *free module* over a set A, denoted $\mathbf{F}_K[A]$, is the module generated by elements of A. By *generated* we mean that for every element $a : A$ there is an element $\{a\} : \mathbf{F}_K[A]$ such that the set $\{\{a\}\}_{a\in A}$ of all such elements form an orthonormal basis for $\mathbf{F}_K[A]$. Furthermore, $\mathbf{F}_K[A]$ contains 0 and is closed under addition and scalar multiplication. For instance, there are elements like $3 \cdot \{a\} + 5 \cdot \{b\}$. Two elements of $\mathbf{F}_K[A]$ are equal if and only if they are forced to be equal by the module axioms and properties of K. Hence, $3\cdot\{a\}+5\cdot\{b\} = 5\cdot\{b\}+3\cdot\{a\}$ by commutativity, but $\{a\} \neq 0$ (whenever $1 \neq 0$ in K) since there is no way to show $\{a\} = 0$ from the axioms alone (a formal argument for this is surprisingly tricky, though). In particular $\{\cdot\}$ is injective so $\{a\} = \{b\}$ implies $a = b$.

An element of $\mathbf{F}_K[A]$ is best thought of as a generalised finite multiset. Any such element can be written uniquely as a basis expansion $\sum_{a:A}(r_a \cdot \{a\})$ where the number of nonzero r_a is finite. Depending on the nature of K there is a different interpretation of what "generalised finite multiset" means.

- When K is $\mathbb{F}_2$ elements $\mathbf{F}_K[A]$ are finite sets. A set like $\{a, b, c\}$ is written as $\{a\} + \{b\} + \{c\}$.
- When K is $\mathbb{Z}$, elements of $\mathbf{F}_K[A]$ are finite *polysets*. A polyset like $\{a^3, b^{-2}, c^5\}$ is written as $3\{a\} - 2\{b\} + 5\{c\}$.
- When K is $\mathbb{R}$ elements of $\mathbf{F}_K[A]$ are generalised finite fuzzy sets, whose membership function is not limited to $[0, 1]$. A generalised fuzzy set like $\{a/0.3, b/0.2, c/0.5\}$ is written as $0.3\{a\} + 0.2\{b\} + 0.5\{c\}$.

The free module $\mathbf{F}_K[A]$ satisfies the following universal property: for any module V and function $f : A \to V$ there is a unique linear map $\widehat{f} : \mathbf{F}_K[A] \to V$ such that $\widehat{f} \circ \{\cdot\} = f$. In essence, to define a *linear map* out of $\mathbf{F}_K[A]$ it suffices to identify the target module and define a *map* out of A, and this map can be chosen *freely*. We can think of this as definition by pattern matching, and write linear maps like

$$\begin{aligned} g &: \mathbf{F}_K[\mathbf{Str}] \to \mathbf{F}_K[\mathbf{Str}] \\ g(\{s\}) &= \{\mathbf{reverse}(s)\} \end{aligned}$$

In this case $g = \widehat{f}$ where $f(s) = \{\mathbf{reverse}(s)\}$. Note that due to linearity we also get the equations

$$g(0) = 0 \qquad g(x + y) = g(x) + g(y) \qquad g(r \cdot x) = r \cdot g(x)$$

but we do not need to handle these cases as they are forced by the condition of linearity. Thus, we get to treat $\mathbf{F}_K[A]$ as an inductive type and pretend that it only contains elements of A, even though it does contain many more elements than that.

Finally, we are going to equip $\mathbf{F}_K[A]$ with a bilinear operator and a weight function to make it into a weighted algebra. There is more than one possible choice, but only one makes sense for our purposes:

$$\begin{aligned} \{a\} \cdot \{a\} &= \{a\} \\ \{a\} \cdot \{b\} &= 0 \quad \text{(for } a \neq b\text{)} \\ \#\{a\} &= 1 \end{aligned}$$

Note that we are defining multiplication by pattern matching in each argument, implicitly appealing to the universal property of $\mathbf{F}_K[A]$ twice. This operation can be thought of as a variant of the Kronecker delta function: If the arguments a, b are equal, we return that unique value as a singleton; if they are unequal, we "fail" by returning 0. When applied to sets it computes the set intersection; for multisets and polysets, however, the multiplicities of common elements are multiplied. For instance, for

$$(3\{a\}+2\{b\}+5\{c\})\cdot(7\{b\}+4\{c\}+2\{d\}) = (2\cdot 7)\{b\}+(5\cdot 4)\{c\} = 14\{b\}+20\{c\}$$

The missing component in $\mathbf{F}_K[A]$ being a *unital* weighted algebra is the unit, an element 1 such that $1 \cdot x = x = x \cdot 1$. If A is finite we can take $1 = \sum_{a:A}\{a\}$, but for infinite A this sum is not well-defined. Thus, we now proceed to show how $\mathbf{F}_K[A]$ can be extended to account for this deficiency, which in turn paves the way to efficient representation of certain infinite sets and—eventually—efficient algebraic join computations.

Compact free. We have seen that the free module over a set A does an excellent job of representing finite subsets of A. However, it lacks the ability to represent complements and in particular there is no general way to represent the subset containing every inhabitant of A. Algebraically, the multiplication on $\mathbf{F}_K[A]$ does not have a unit element for infinite A.

To rectify these deficiencies we introduce the *compact free module*, $\mathbf{F}^*_K[A]$, constructed by taking the free module $\mathbf{F}_K[A]$ and adjoining a distinct element

1. We think of 1 as *symbolising* the potentially infinite sum $\sum_{a:A}\{a\}$. However, even when A is finite, 1 is by definition distinct from $\sum_{a:A}\{a\}$.

The choice of never identifying 1 with $\sum_{a:A}\{a\}$ has several advantages. We do not have to know whether A is finite or infinite to decide if 1 is linearly independent from the other generators. It allows a compact symbolic representation of this sum when A is finite but large. And finally it gives us the following universal property: for any module V together with a map $f : A \to V$ and an element $u : V$ there is a unique linear map $\widehat{f} : \mathbf{F}^*_K[A] \to V$ such that $\widehat{f} \circ \{\cdot\} = f$ and $\widehat{f}(1) = u$.

In terms of pattern matching this amounts to having cases for $\{a\}$ and 1. With this addition $\mathbf{F}^*_K[A]$ has a unital weighted algebra structure, where multiplication of generators works just like for $\mathbf{F}_K[A]$ and 1 is the multiplicative unit. Explicitly:

$$\begin{array}{lll} \{a\} \cdot \{a\} = \{a\} & 1 \cdot y = y & \#\{a\} = 1 \\ \{a\} \cdot \{b\} = 0 \quad (\text{for } a \neq b) & x \cdot 1 = x & \#1 = 1 \end{array}$$

This enables us to represent not just finite sets, but also *cofinite* sets: the subsets of A that contain all but a finite number of elements. For example, the set $A \setminus \{a, b\}$ is written as $1 - (\{a\} + \{b\})$. When we introduce tensor products later we will see how even more interesting subsets can be represented compactly in this manner.

Biproduct. The *biproduct* of modules U and V is a module $U \oplus V$ consisting of pairs of elements from U and V with operations defined pointwise. For instance given $(u_1, v_1), (u_2, v_2) : U \oplus V$ we define

$$(u_1, v_1) + (u_2, v_2) = (u_1 + u_2, v_1 + v_2)$$

The biproduct satisfies the universal property of binary products: there are linear maps $p_1 : U \oplus V \to U$ and $p_2 : U \oplus V \to V$ such that for any module W with linear maps $p'_1 : W \to U$ and $p'_2 : W \to V$ there is a unique linear map $(p'_1, p'_2) : W \to U \oplus V$ such that $p_1 \circ (p'_1, p'_2) = p'_1$ and $p_2 \circ (p'_1, p'_2) = p'_2$. The p's are projections and (p'_1, p'_2) is the pair constructor. Operationally:

$$\begin{array}{rcl} p_1(u, v) & = & u \\ p_2(u, v) & = & v \\ (p'_1, p'_2)(w) & = & (p'_1(w), p'_2(w)) \end{array}$$

This universal property gives a way to define linear maps *into* the biproduct by giving the result of projecting each component. In other words, definition by copattern matching. For instance

$$\begin{array}{rcl} f & : & V \to V \oplus V \\ p_1(f(v)) & = & v \\ p_2(f(v)) & = & 3v \end{array}$$

The biproduct also satisfies the universal property of binary coproducts, obtained by simply reversing the direction of all arrows: there are linear maps $i_1 : U \to U \oplus V$ and $i_2 : V \to U \oplus V$ such that for any module W with linear maps $i'_1 : U \to W$ and $i'_2 : V \to W$ there is a unique linear map $[i'_1, i'_2] : U \oplus V \to W$ such that $[i'_1, i'_2] \circ i_1 = i'_1$ and $[i'_1, i'_2] \circ i_2 = i'_2$.

The i's are injections and $[i'_1, i'_2]$ is case analysis. Operationally:

$$\begin{aligned} i_1(u) &= (u, 0) \\ i_2(v) &= (0, v) \\ [i'_1, i'_2](u, v) &= i'_1(u) + i'_2(v) \end{aligned}$$

This universal property gives a way to define linear maps *out of* the biproduct by explaining what happens to either injection. In other words, definition by pattern matching. For instance

$$\begin{aligned} f &: V \oplus V \to V \\ f(i_1(v)) &= 2v \\ f(i_2(v)) &= v \end{aligned}$$

From a programming point of view biproducts are an unfamiliar construction: a data type that supports both pattern matching and copattern matching. They can be treated as either a sum or a product type depending on which is most convenient.

Finally, we describe the canonical algebra structure on the biproduct. Just like for free modules, the motivation for the definition is intersection of multisets. Suppose U and V are algebras. Then so is $U \oplus V$. The definition of $1_{U \oplus V}$ uses copattern matching while $\cdot$ and # use pattern matching.

$$\begin{aligned} p_1(1_{U \oplus V}) &= 1_U \\ p_2(1_{U \oplus V}) &= 1_V \\ i_1(u) \cdot i_1(u') &= i_1(u \cdot u') \\ i_1(u) \cdot i_2(v') &= 0 \\ i_2(v) \cdot i_1(u') &= 0 \\ i_2(v) \cdot i_2(v') &= i_2(v \cdot v') \\ \#i_1(u) &= \#u \\ \#i_2(v) &= \#v \end{aligned}$$

Finite map. Suppose we are given a set A and a module U. The *finite map* module $A \Rightarrow U$ consists of maps $A \to U$ with *finite support*, i.e. maps which produce a nonzero value at finitely many elements of A. It is the module generated by elements of the form $a \mapsto u$ where $a : A$ and $u : U$, subject to the requirement that $a \mapsto \cdot$ is a linear map for any a. Any finite map can be written as $\sum_{a:A}(a \mapsto u_a)$ where $u_a = 0$ for all but finitely many a.

The finite map module satisfies the following universal property: for any module V together with a family of maps $f_a : U \to V$ there exists a unique linear map $\mathbf{case}(f) : (A \Rightarrow U) \to V$ such that $\mathbf{case}(f) \circ (a \mapsto \cdot) = f_a$ for all $a : A$. This allows pattern matching similar to the free module. For example:

$$\begin{aligned} f &: (A \Rightarrow K) \to \mathbf{F}_K[A] \\ f(a \mapsto 1) &= \{a\} \end{aligned}$$

Recall that the use of 1 on the left-hand side is pattern matching on scalars.

Intuitively, we think of $A \Rightarrow U$ as elements of U indexed by A. For instance, suppose we have a multiset of strings and we want to index it by string lengths. The indexed space would be $\mathbb{N} \Rightarrow \mathbf{F}_K[\mathbf{Str}]$ with the indexing being done as follows.

$$\begin{aligned} \text{indexbylength} &: \mathbf{F}_K[\mathbf{Str}] \to (\mathbb{N} \Rightarrow \mathbf{F}_K[\mathbf{Str}]) \\ \text{indexbylength}(\{s\}) &= \mathbf{length}(s) \mapsto \{s\} \end{aligned}$$

We can also go in the other direction and forget the index.

$$\begin{aligned} \mathbf{sum} &: (A \Rightarrow U) \to U \\ \mathbf{sum}(a \mapsto u) &= u \end{aligned}$$

Like the free module the finite map module is a weighted algebra, but lacks a unit when the index set is infinite. Multiplication and weight are defined by:

$$\begin{aligned} (a \mapsto u) \cdot (a \mapsto v) &= a \mapsto (u \cdot v) \\ (a \mapsto u) \cdot (b \mapsto v) &= 0 \quad (\text{for } a \neq b) \\ \#(a \mapsto u) &= \#u \end{aligned}$$

Compact map. Finite maps, like free modules, do not possess a unit element when the index set is infinite. More generally, they do not possess constant mappings that have the same (nonzero) value everywhere. The solution is similar: adjoin distinct elements to account for this deficiency.

Thus, the *compact map* module $A \Rightarrow^* U$ is obtained by adding elements of the form $* \mapsto u$ for any $u : U$. The $*$ represents a *wildcard* which matches anything. Intuitively $* \mapsto u$ symbolises $\sum_{a:A}(a \mapsto u)$ but is by definition always distinct, just like for the compact free module.

The compact map module satisfies the following universal property: for any module V together with a family of maps $f_a : U \to V$ as well as a map $f_* : U \to V$ there exists a unique linear map $\mathbf{case}(f) : (A \Rightarrow^* U) \to V$ such that $\mathbf{case}(f) \circ (a \mapsto \cdot) = f_a$ for all $a : A$ and $\mathbf{case}(f) \circ (* \mapsto \cdot) = f_*$.

The unital weighted algebra structure on compact maps is given as follows.

$$\begin{aligned} &(a \mapsto u) \cdot (a \mapsto v) = a \mapsto (u \cdot v) && 1_{A \Rightarrow^* U} = * \mapsto 1_U \\ &(a \mapsto u) \cdot (b \mapsto v) = 0 \quad (\text{for } a \neq b) && \\ &(a \mapsto u) \cdot (* \mapsto v) = a \mapsto (u \cdot v) && \#(a \mapsto u) = \#u \\ &(* \mapsto u) \cdot (a \mapsto v) = a \mapsto (u \cdot v) && \#(* \mapsto u) = \#u \\ &(* \mapsto u) \cdot (* \mapsto v) = * \mapsto (u \cdot v) && \end{aligned}$$

Lookup in a compact map works like for finite maps, except $*$ is also a valid key. To see how this works in practice, suppose we have an element:

$$x = (* \mapsto 2) + (a \mapsto 3) + (b \mapsto -2)$$

Consider the following lookups:

$$x(*) = 2 \qquad x(a) = 2 + 3 = 5 \qquad x(b) = 2 - 2 = 0 \qquad x(c) = 2$$

It is evident that the $* \mapsto 2$ component of x serves as a *baseline* and a component like $a \mapsto 3$ determines how much the value at a *deviates* from that baseline. In particular, the value at b deviates by exactly the negative of the baseline value, the result being that b is contained 0 times in x when viewed as a polyset. This is in contrast to the more common construct of having finite maps with a *default value* for keys not explicitly listed. In particular, compact map addition is commutative: it does not matter in which order the maps are listed.

Tensor product. The *tensor product* of modules U and V is a module $U \otimes V$ generated by elements of the form $u \otimes v$ with $u : U$ and $v : V$, subject to the requirement that $\otimes$ is bilinear, i.e. linear in each argument separately. More explicitly, linearity in the first argument requires

$$0 \otimes v = 0 \qquad (u_1 + u_2) \otimes v = u_1 \otimes v + u_2 \otimes v \qquad (r \cdot u) \otimes v = r \cdot (u \otimes v)$$

Linearity in the second argument is analogous. In general, any element of the tensor product can be written as a sum of $\otimes$-pairs, i.e. $\sum_i (u_i \otimes v_i)$. Compare this with the biproduct, which is also generated by pairs of elements. An element $\sum_i (u_i, v_i)$ can always be reduced to $(\sum_i u_i, \sum_i v_i)$, since the biproduct pair constructor is linear in both arguments *simultaneously*. The tensor product is linear in each argument *separately* and elements can only be simplified in some circumstances, such as

$$u_1 \otimes v_1 + u_1 \otimes v_2 + u_2 \otimes v_1 + u_2 \otimes v_2 = (u_1 + u_2) \otimes (v_1 + v_2)$$

This kind of simplification can often reduce the size of a term from quadratic to linear, but recognising such opportunities is not easy. Consequently, when a term is in this kind of simplified, compact form we should not expand it unless absolutely necessary!

The tensor product satisfies the following universal property: for any module W and bilinear map $f : U \times V \to W$ there exists a unique linear map $g : U \otimes V \to W$ such that $g \circ \otimes = f$.

The consequence is that any bilinear map can be written as a linear map from the tensor product. It also justifies defining linear maps by pattern matching on $\otimes$, for example:

$$\begin{aligned} f &: U \otimes V \to V \otimes U \\ f(u \otimes v) &= v \otimes u \end{aligned}$$

This is only valid if the definition is linear in each argument separately. In particular we cannot simply define a projection $\pi_1 : U \otimes V \to U$ as $\pi_1(u \otimes v) = u$, as that would not be linear in the second argument. In this respect the tensor product is different from an ordinary product type. If V is equipped with a weighted algebra structure, however, we can define:

$$\begin{aligned} \pi_1 &: U \otimes V \to U \\ \pi_1(u \otimes v) &= \#v \cdot u \end{aligned}$$

The weight serves to dispose of data, which is not possible for an arbitrary module in general. Finally, we equip the tensor product with an algebra structure. Suppose U and V are algebras. All operations are defined pointwise:

$$\begin{aligned} &(u_1 \otimes v_1) \cdot (u_2 \otimes v_2) = (u_1 \cdot u_2) \otimes (v_1 \cdot v_2) \\ &1_{U \otimes V} = 1_U \otimes 1_V \qquad \#(u \otimes v) = \#u \cdot \#v \end{aligned}$$

Note in particular how computing the weight of a tensor product simply reduces to computing the weight of each factor. This saves us from having to expand $u \otimes v$ at all. When u and v are themselves large sums this saves a considerable amount of work.

2.7 Inner Products and Adjoints

Given a weighted algebra M we can define an *inner product* by

$$\begin{aligned} &(\diamond) : M \to M \to K \\ &u \diamond v = \#(u \cdot v) \end{aligned}$$

Intuitively, the inner product is a linear extension of the characteristic function for equality. It measures the degree to which two elements are equal. Indeed, if $M = \mathbf{F}_K[A]$ the definition specialises to:

$$\begin{aligned} \{a\} \diamond \{b\} = 1 \quad \text{when } a = b \\ \{a\} \diamond \{b\} = 0 \quad \text{when } a \neq b \end{aligned}$$

Often the inner product is considered to be part of the structure—such a vector space is called an inner product space—and not a derived operation. For our purposes we need the more powerful weighted algebra structure anyway and have little use for inner products in isolation.

In fact, the weighted algebra structure allows defining inner products of any arity. If we consider the weight map itself as a unary inner product and the map defined above as a binary inner product then the ternary version is defined as (u, v, w) maps to $\#(u \cdot v \cdot w)$ and so on.

With inner products in hand we can give a formal definition of adjoints. Let M and N be weighted algebras and $f : M \to N$ a linear map. An *adjoint* of f is a linear map $f^\dagger : N \to M$ such that

$$f(x) \diamond y = x \diamond f^\dagger(y)$$

for all $x : M$ and $y : N$.

Recalling our intuition for inner products as measuring equality, the definition of adjoint says that f maps x to y to the same degree that $f^\dagger$ maps y to x. In the case of free modules we can let $x = \{a\}$ and $y = \{b\}$; the constraint then guarantees that $f(\{a\}) = r \cdot \{b\}$ precisely when $f^\dagger(\{b\}) = r \cdot \{a\}$. When representing a linear map between finite-dimensional free modules as a matrix the factor r is found at position (a, b) in the matrix. The matrix of the adjoint is found by transposing the matrix.

Given that adjoints look suspiciously similar to inverses, what is the difference? Firstly, adjoints can exist even when inverses do not. For instance, consider a map $f : K \oplus K \to K$ given by $f(\mathbf{inl}(x)) = x$ and $f(\mathbf{inr}(x)) = x$. It has no inverse, but has an adjoint $f^\dagger(x) = \mathbf{inl}(x) + \mathbf{inr}(x)$. Secondly, multiplicities are dealt with differently. Consider a map $f : K \to K$ given by $f(x) = 2x$. The inverse is given by $f^{-1}(x) = \frac{1}{2}x$ while the adjoint is simply f itself. The operation of taking adjoints is linear in the map, while taking inverses has no such property.

When a linear map has an adjoint which is also an inverse it is called *unitary*.

2.8 Functors and Isomorphisms

All of the constructions we have seen are *functors*, structure-preserving maps between categories. Recall that a functor acts not just on objects (sets, modules, etc.), but also on maps between objects. For instance $\mathbf{F}_K[A]$ is functorial in A, so given any map between sets $f : A \to B$ we have $\mathbf{F}_K[f] : \mathbf{F}_K[A] \to \mathbf{F}_K[B]$.

Let f be a map between sets and α, β be linear maps. The functors act as follows:

$$\begin{aligned}
\mathbf{F}_K[f](\{a\}) &= \{f(a)\} \\
\mathbf{F}^*_K[f](\{a\}) &= \{f(a)\} \\
\mathbf{F}^*_K[f](1) &= 1 \\
(\alpha \otimes \beta)(u \otimes v) &= \alpha(u) \otimes \beta(v) \\
(f \Rightarrow^* \alpha)(* \mapsto u) &= * \mapsto \alpha(u) \\
(f \Rightarrow^* \alpha)(a \mapsto u) &= f(a) \mapsto \alpha(u) \\
(f \Rightarrow \alpha)(a \mapsto u) &= f(a) \mapsto \alpha(u)
\end{aligned}$$

Generally, these actions can be derived mechanically and there is only one reasonable choice. The simplicity is deceptive, though, and it is easy to overlook how much one gets for free with a categorical approach. Perhaps the clearest example of this is $\otimes$. It is usually called the Kronecker product, defined as a complicated block matrix expression and relegated to advanced linear algebra courses. The functorial action, by contrast, could not be simpler.

The module constructions we have presented are related in various ways. More precisely, there are a number of *natural isomorphisms*. An isomorphism $\varphi : U \cong V$ is simply a linear map $\varphi : U \to V$ together with an inverse linear

$\mathbf{F}_K[0] \cong \mathbf{0}$	$\mathbf{F}_K[1] \cong K$
$0 \Rightarrow U \cong \mathbf{0}$	$1 \Rightarrow U \cong U$
$A \Rightarrow \mathbf{0} \cong \mathbf{0}$	$A \Rightarrow K \cong \mathbf{F}_K[A]$
$\mathbf{F}_K[A+B] \cong \mathbf{F}_K[A] \oplus \mathbf{F}_K[B]$	$\mathbf{F}_K[A \times B] \cong \mathbf{F}_K[A] \otimes \mathbf{F}_K[B]$
$(A+B) \Rightarrow U \cong (A \Rightarrow U) \oplus (B \Rightarrow U)$	$(A \times B) \Rightarrow U \cong A \Rightarrow B \Rightarrow U$
$A \Rightarrow (U \oplus V) \cong (A \Rightarrow U) \oplus (A \Rightarrow V)$	$A \Rightarrow (U \otimes V) \cong (A \Rightarrow U) \otimes V$
$A \Rightarrow U \cong \mathbf{F}_K[A] \otimes U$	$A \Rightarrow^* U \cong \mathbf{F}^*_K[A] \otimes U$
$\mathbf{F}^*_K[A] \cong \mathbf{F}_K[A] \oplus K$	$A \Rightarrow^* U \cong (A \Rightarrow U) \oplus A$

Table 2.1: A selection of natural isomorphisms

map $\varphi^{-1} : V \to U$. Naturality can be stated precisely using category theory, but for our purposes it is sufficient to think of "natural" as "polymorphic".

When defining an isomorphism φ we write both directions simultaneously using the syntax $p \Leftrightarrow q$ to mean $\varphi(p) = q$ and $p = \varphi^{-1}(q)$. A selection of isomorphisms can be seen in Table 2.1. We draw particular attention to the relationship between free modules and tensor products given by the isomorphism:

$$\begin{array}{rcl} \mathbf{F}_K[A \times B] & \cong & \mathbf{F}_K[A] \otimes \mathbf{F}_K[B] \\ \{(a,b)\} & \Leftrightarrow & \{a\} \otimes \{b\} \end{array}$$

The tensor product of free modules can itself be written as a free module. However, the pattern matching notation belies the true cost of this conversion. The typical element of $\mathbf{F}_K[A] \otimes \mathbf{F}_K[B]$ is a sum of terms like $(\sum_i r_i\{a_i\}) \otimes (\sum_j s_j\{b_j\})$ which converts to $\sum_i \sum_j r_i s_j\{(a_i, b_j)\}$, a quadratic increase in size. Converting back again yields $\sum_i \sum_j r_i s_j(\{a_i\} \otimes \{b_j\})$. This is extensionally equal to the original term, but much larger. Hence, passing through the free module is not free! We will generally prefer to stay on the right side of this isomorphism, only converting when necessary. Fortunately, the isomorphisms shown thus far demonstrate that any polynomial type can be expressed as a module using $\mathbf{0}$, K, $\oplus$ and $\otimes$.

Free Modules Everywhere?

At this point we have seen that any module constructed using any combination of $\mathbf{0}$, K, $\oplus$, $\otimes$, $\mathbf{F}_K[\cdot]$, $\mathbf{F}^*_K[\cdot]$ and $\Rightarrow$ is isomorphic to some free module. Indeed, if K is a field—in which case modules are vector spaces—the classical mathematician would remark that by the Axiom of Choice every vector space is isomorphic to a free one. Why do we not simply consider only free modules then?

Firstly, these isomorphisms only concern the module structure. The algebra structure differs in many cases. For instance, we saw that $\mathbf{F}^*_K[A] \cong \mathbf{F}_K[A] \oplus K$ as modules, but certainly *not* as algebras (if the right-hand side even *has* an algebra structure, which is only the case when A is finite). Secondly, even if two modules are *extensionally* equal, i.e. isomorphic, they need not be *intensionally* equal. The clearest example of this is $\mathbf{F}_K[A \times B] \cong \mathbf{F}_K[A] \otimes \mathbf{F}_K[B]$ where the right-hand side can express certain large terms much more compactly and converting to the left-hand side generally yields asymptotically larger terms.

This distinction is of course invisible from a purely categorical perspective, but we do not have the luxury of such purity. Our goal, ultimately, is computation and category theory is neutral in this regard (it fits well with a constructive treatment, but does not have a concept of computation unless explicitly modelled). Vague appeals to indistinguishability of isomorphic objects is therefore a pitfall best avoided.

2.9 The Adjoint Perspective

For the very categorically inclined the development above can be cast almost purely in terms of adjunctions. This yields an arguably cleaner and certainly more general theory, although one has to beware of venturing too deeply into the depths of "abstract nonsense".

Limits as adjunctions. For categories $\mathfrak{C}$ and $\mathfrak{J}$ let $\Delta_{\mathfrak{J}} : \mathfrak{C} \to \mathfrak{C}^{\mathfrak{J}}$ be the diagonal functor $\Delta_{\mathfrak{J}}(X)(Y) = X$. Objects in the functor category $\mathfrak{C}^{\mathfrak{J}}$ are thought of as diagrams in $\mathfrak{C}$ of shape $\mathfrak{J}$. The left adjoint of $\Delta_{\mathfrak{J}}$, if it exists, assigns to each diagram its colimit. Analogously the right adjoint produces the limit of each diagram. In particular if $\mathfrak{J} = 2$, the category with just two objects, the left adjoint is $+$ (the binary coproduct) and the right adjoint is $\times$ (the binary product).

Distributive categories. A category that has binary products and binary coproducts such that products distribute over coproducts (i.e. $X \times (Y + Z) \cong (X \times Y) + (X + Z)$) is *distributive.*

Additive categories. On the other hand it might be the case that the left and right adjoints of Δ_2 coincide; then (given a few equations relating product projections and coproduct injections) the category has *biproducts*, written $X \oplus Y$. Biproducts are sufficient to define zero arrows as well as addition of arrows. A category where all finite biproducts exist and where the set of arrows between any two objects has the structure of an Abelian group is called *additive.*

Monoidal categories. A category $\mathfrak{C}$ can be equipped with an object I and a functor $\otimes : \mathfrak{C} \times \mathfrak{C} \to \mathfrak{C}$ as well as isomorphisms witnessing left and right identity of I and associativity of $\otimes$. In this case $\mathfrak{C}$ is a *monoidal* category. A functor between monoidal categories is itself monoidal if it preserves I and $\otimes$ up to isomorphism.

The general setting. Now let $\mathfrak{C}$ be a distributive category with a monoidal structure given by 1 and $\times$ and let $\mathfrak{D}$ be an additive category with monoidal structure I and $\otimes$. Furthermore, let $F : \mathfrak{C} \to \mathfrak{D}$ be a monoidal functor which is left adjoint to $U : \mathfrak{D} \to \mathfrak{C}$. This adjunction defines the general setting of our theory.

In particular we get that

- $F(0) \cong 0$ (the empty set is mapped to the empty biproduct)
- $F(1) \cong I$ (singleton sets are mapped to the monoidal unit)
- $F(\coprod_{a:A} B_a) \cong \coprod_{a:A} F(B_a)$ (left adjoints preserve colimits and hence coproducts)
- $F(A \times B) \cong F(A) \otimes F(B)$ (monoidal functors preserve monoidal structure)
- $U(M \oplus N) \cong U(M) \times U(N)$ (right adjoints preserve limits and hence products)

Thus F, as a generalisation of $\mathbf{F}_K[-]$, ties together the whole theory and for a given choice of F there is little to no wiggle room for the rest of the structure.

Instances. Interesting choices for $\mathfrak{C}$, $\mathfrak{D}$ and F include:

- $\mathfrak{C} = \textsc{Set}$ and $\mathfrak{D} = \textsc{Mod}_K$ with $F(X) = \mathbf{F}_K[X]$, our "standard model".
- $\mathfrak{C} = \textsc{Set}$ and $\mathfrak{D} = \textsc{Rel}$ (sets and relations) with $F(X) = X$, the set theory model. All limits and colimits coincide and every morphism has an adjoint.
- $\mathfrak{C} = \textsc{Set}$ and $\mathfrak{D} = \textsc{Span}_{\textsc{Mod}_K}$ (a morphism $X \to Y$ is a span $X \leftarrow Z \to Y$ for some Z) with $F(X) = \mathbf{F}_K[X]$, combines the multiplicities of the standard model with the symmetry of the relational model.
- $\mathfrak{C} = \textsc{Top}$ (topological spaces) and $\mathfrak{D} = \textsc{TopVec}_K$ (topological vector spaces) with $F(X)$ being the free topological vector space generated by X, similar to the standard model but with a topology underneath it all.

Chapter 3

The Algeo Language

3.1 Motivation

In Section 2 we encountered a host of modules. Each module is born with a universal property specifying how to construct linear maps into or out of it. Appealing explicitly to the universal property is cumbersome, so we developed a notation of pattern and copattern matching. Thus, we can write linear maps such as

$$\begin{aligned} &f : M \otimes N \to N \otimes M \\ &f(x \otimes y) = y \otimes x \end{aligned}$$

whose formal definition includes an appeal to the universal property of tensor products with an auxiliary bilinear map $g(x, y) = y \otimes x$. The power of this approach becomes even more apparent when we use nested patterns like

$$\begin{aligned} &\pi_1 : M \otimes \mathbf{F}_K[A] \to M \\ &\pi_1(x \otimes \{a\}) = x \end{aligned}$$

whose formal definition includes multiple appeals to universal properties.

When systematising the notation in this way the contours of a programming language emerges. At this point we can think about how we can make the notation more clear and convenient. For instance, we could write multiplication of biproducts as simply

$$\begin{aligned} &i_1(x) \cdot i_1(y) = i_1(x \cdot y) \\ &i_2(x) \cdot i_2(y) = i_2(x \cdot y) \end{aligned}$$

where omitted cases are implicitly understood to be zero. In general, any single pattern is enough to define a linear map where all other data required by the relevant universal property is set to zero. Functions with multiple patterns are simply the sum of the functions induced by each pattern.

This also permits multiplication of free modules to be defined more elegantly as simply

$$\{a\} \cdot \{a\} = \{a\}$$

omitting the zero case with the awkward side condition of inequality. Note that mentioning a twice on the left-hand side is acceptable, since it ranges over some set A whose elements can be compared for equality.

We can in fact do even better. Multiplication allows us to interpret patterns with multiple occurences of variables that range over any algebra. For example for any algebras M and N we can write

$$m : M \otimes N \otimes M \otimes N \to M \otimes N$$
$$m(x \otimes y \otimes x \otimes y) = x \otimes y$$

which is equivalent to $m(x \otimes y \otimes x' \otimes y') = (x \cdot x') \otimes (y \cdot y')$.

In particular, we get a crisp definition of inner products for any weighted algebra:

$$x \diamond x = \#x$$

Inner products in fact play a vital rôle in our development. When defining isomorphisms in Section 2.8 we noted that most, if not all, of them only need to be written in the forward direction at which point the reverse direction is "obvious". This is not the case for all isomorphisms. After all, finding the inverse of a matrix is not in general a trivial operation.

The saving grace is that we are primarily concerned with a special class of isomorphisms, the *unitary transformations*. Recall that a unitary transformation has both an inverse and an adjoint, and the two coincide. The observation that inverses seems to be easy to find in practice can now be elucidated: adjoints are easy to find and many interesting isomorphisms are in fact unitary transformations.

These considerations suggest that notationally we should be focussing on adjoints. They are easy to derive syntactically, cheap to compute and useful in many cases as partial or pseudo inverses. At this point the main ideas behind Algeo emerges:

- Linear maps should be defined by equations using $\Leftrightarrow$ rather than $=$.
- The $\Leftrightarrow$ operator is just an alias for the inner product.
- Functional patterns are simply adjoints.
- Introduction of variables is orthogonal to defining functions.

3.2 Algeo Tutorial

We now give an introduction to the Algeo language as well as its primary paradigm, *functional logic programming*. At a glance Algeo resembles a typical functional language—and indeed a subset of the language can serve as a perfectly good pure functional language—but the semantics are actually closer to that of a logic language.

3.2.1 Definitions

An Algeo program consists of definitions. Each definition has a type declaration and any number of assertions. For example, we can give the following definition:

$$\mathbf{favcolour} : \mathbf{Atom}$$
$$\mathbf{favcolour} \Leftrightarrow \texttt{green}$$

This introduces a binding **favcolour** of type **Atom** and a single assertion (of type **Scalar**) stating that **favcolour** is pointwise equal with the atom `green`.

Given such a definition we can now evaluate expressions containing **favcolour**. Evaluation works by substituting $[x : \mathbf{Atom}]\, x \Leftrightarrow \texttt{green}; x$ for **favcolour** where x is some fresh name. Let us break down this expression into its parts. The brackets $[x : \mathbf{Atom}]$ introduce a variable x of type **Atom** whose scope extends all the way to the right. The remainder consists of two parts combined with a semicolon, the *biased conjunction* operator. To the left we have a copy of the assertion from the definition and to the right is the variable x.

Semantics of definitions What does it then *mean*? The short version is that **favcolour** is by definition the sum over all atoms subject to the constraint imposed by the assertion in its definition. This is exactly what the substituted term expresses. The long version is as follows. When introducing a variable via $[x : \mathbf{Atom}]$ this "morally" amounts to making an infinite choice between (equivalently, an infinite sum over) all possible atoms, i.e. for any expression e we have the "equality"

$$[x : \mathbf{Atom}]\, e \mathrel{\overset{\cdots}{=}} e^{x:=\texttt{a}} \parallel e^{x:=\texttt{b}} \parallel \cdots \parallel e^{x:=\texttt{green}} \parallel \cdots$$

where $\parallel$ means binary choice and $e^{x:=e'}$ means the expression e with e' substituted for x. We defensively use $\overset{\cdots}{=}$ ("morally equal") rather than $=$ since the right-hand side is technically not a well-formed expression. A formal version will be introduced later.

Specialising the equation to our particular case we get

$$\mathbf{favcolour} \mathrel{\overset{\cdots}{=}} (\texttt{a} \Leftrightarrow \texttt{green}; \texttt{a}) \parallel (\texttt{b} \Leftrightarrow \texttt{green}; \texttt{b}) \parallel \cdots \parallel (\texttt{green} \Leftrightarrow \texttt{green}; \texttt{green}) \parallel \cdots$$

Pointwise equality for *base values*, which include atom constants, reduces to $\overline{1}$ (the scalar value 1) for equal terms and $\overline{0}$ (the scalar value 0) otherwise. Hence, we get

$$\mathbf{favcolour} \mathrel{\overset{\cdots}{=}} (\overline{0}; \texttt{a}) \parallel (\overline{0}; \texttt{b}) \parallel \cdots \parallel (\overline{1}; \texttt{green}) \parallel \cdots$$

where all the elided parts also have zeroes. By the semantics of biased conjunctions this is equal to

$$\mathbf{favcolour} \mathrel{\overset{\cdots}{=}} \emptyset \parallel \emptyset \parallel \cdots \parallel \texttt{green} \parallel \cdots$$

where $\emptyset$ is a polymorphic nullary choice (in this case of type **Atom**). Nullary choice is the identity of binary choice and finally we get

$$\mathbf{favcolour} = \texttt{green}$$

which holds morally *and* formally.

Multiple assertions This is perhaps not too surprising, but take care not to be misled by the trivial nature of this example! Suppose instead that we are not quite sure of our favourite colour. Uncertainty can be represented by having multiple assertions in a definition.

$$\begin{aligned}&\mathbf{favcolour}' : \mathbf{Atom}\\&\mathbf{favcolour}' \Leftrightarrow \texttt{green}\\&\mathbf{favcolour}' \Leftrightarrow \texttt{yellow}\end{aligned}$$

These two assertions together are by definition the same as having a single assertion using binary choice:

$$(\mathbf{favcolour}' \Leftrightarrow \texttt{green}) \parallel (\mathbf{favcolour}' \Leftrightarrow \texttt{yellow})$$

The value of **favcolour′** is then $[x : \mathbf{Atom}]\,(x \Leftrightarrow \texttt{green} \parallel x \Leftrightarrow \texttt{yellow}); x$ Using the same rewriting approach as before we indeed get

$$\mathbf{favcolour}' = \texttt{green} \parallel \texttt{yellow}$$

Hence, our new favourite colour is a choice between green and yellow. There is a simpler way to arrive at this conclusion using a few high-level rewrite rules. Firstly, $\Leftrightarrow$ distributes over $\parallel$ so in general

$$(e_1 \Leftrightarrow e_2) \parallel (e_1 \Leftrightarrow e_2') = e_1 \Leftrightarrow (e_2 \parallel e_2')$$

Secondly, for any variable x of type τ and term e (with no free occurences of x) we have

$$([x : \tau]\, x \Leftrightarrow e; x) = e$$

This is similar to eliminating `let`-expressions in a functional language. Thus, we can reason:

$$[x : \mathbf{Atom}]\,(x \Leftrightarrow \texttt{green} \parallel x \Leftrightarrow \texttt{yellow}); x = \texttt{green} \parallel \texttt{yellow}$$

In general, if we have a definition like

$$\begin{aligned}&d : \tau\\&d \Leftrightarrow e_1\\&\vdots\\&d \Leftrightarrow e_n\end{aligned}$$

we can conlude $d = e_1 \parallel \cdots \parallel e_n$.

The power of indirection The consequences of the definitions made so far should hardly be surprising, even if the method of interpreting them is a bit unusual. Let us now proceed with a potentially confusing definition to make a point.

$$\mathbf{mystery} : \mathbf{Atom}$$
$$(\mathbf{mystery} \parallel \texttt{green}) \Leftrightarrow (\mathbf{mystery} \parallel \texttt{yellow})$$

Note in particular that the *subject* (name being defined, in this case **mystery**) can occur:

- any number of times in the assertion;
- on the right-hand side of $\Leftrightarrow$;
- as an operand of $\parallel$.

The subject is essentially just a free variable in the assertion expression with no constraints imposed on its use. Due to the way definitions are interpreted this causes no trouble, but it does lead to surprises at first.

The simplest way to unravel the above definition is by applying distributivity of $\Leftrightarrow$ on both sides to get

$$\mathbf{mystery} \Leftrightarrow \mathbf{mystery} \parallel \mathbf{mystery} \Leftrightarrow \texttt{yellow} \parallel \texttt{green} \Leftrightarrow \mathbf{mystery} \parallel \texttt{green} \Leftrightarrow \texttt{yellow}$$

As a definition its meaning is then given by

$$[x : \mathbf{Atom}]\,(x \Leftrightarrow x \parallel x \Leftrightarrow \texttt{yellow} \parallel \texttt{green} \Leftrightarrow x \parallel \texttt{green} \Leftrightarrow \texttt{yellow}); x$$

Both ; and $[x : \mathbf{Atom}]$ distribute over $\parallel$.

$$\begin{aligned}&([x : \mathbf{Atom}]\,x \Leftrightarrow x; x) \parallel ([x : \mathbf{Atom}]\,x \Leftrightarrow \texttt{yellow}; x) \parallel\\&([x : \mathbf{Atom}]\,\texttt{green} \Leftrightarrow x; x) \parallel ([x : \mathbf{Atom}]\,\texttt{green} \Leftrightarrow \texttt{yellow}; x)\end{aligned}$$

The second and third alternative can be simplified as seen earlier.

$$([x : \mathbf{Atom}]\,x \Leftrightarrow x; x) \parallel \texttt{yellow} \parallel \texttt{green} \parallel ([x : \mathbf{Atom}]\,\texttt{green} \Leftrightarrow \texttt{yellow}; x)$$

In the first alternative we can exploit the identity $b \Leftrightarrow b = \overline{1}$ for any base value b (variables are considered base values), and in the fourth alternative we have $\texttt{green} \Leftrightarrow \texttt{yellow} = \overline{0}$. Simplifying the biased conjunctions we get:

$$([x : \mathbf{Atom}]\,x) \parallel \texttt{yellow} \parallel \texttt{green} \parallel ([x : \mathbf{Atom}]\,\emptyset)$$

Finally, the first alternative can be written using the more concise wildcard $* = [x : \tau]\,x$ (when τ can be inferred from the context), and the fourth alternative reduces to $\emptyset$. Thus, we arrive at the following:

$$\mathbf{mystery} = * \parallel \texttt{yellow} \parallel \texttt{green}$$

The intuition is that **mystery** is a choice between atoms, where `yellow` and `green` occur twice each and every other atom occurs once. This is different from $*$ alone, which is a choice between every atom occuring just once. Multiplicities matter!

Looking back at the assertion that defines **mystery** this outcome is not what one would expect from trying to “solve the equation” (which is in any case a dubious notion since $\Leftrightarrow$ is not a relation). Nevertheless, the semantics of $\Leftrightarrow$ as a generalisation of equality that respects choice is quite natural.

3.2.2 Functions

So far we have studied definitions of atoms with varying degrees of complexity. Algeo also provides a function type. For instance, the identity function can be written as:

$$\mathbf{id} : \tau \rightarrow \tau$$
$$\mathbf{id}\ \mathbf{x} = \mathbf{x}$$

The variable **x** is free in the assertion. By convention, all top-level free variables are assumed to be introduced by top-level binders (in any order). The explicit version would be

$$[\mathbf{x} : \tau]\, \mathbf{id}\ \mathbf{x} = \mathbf{x}$$

When **id** is referenced in other parts of a program, each use will thus have its own copy of the free variables and be independent of any other use. This is important since function application works differently than in a typical functional language. In general, an application $e_1\ e_2$ of type τ reduces to $[x : \tau]\, e_1 \Leftrightarrow (e_2 \mapsto x); x$. The $\mapsto$ operator is the canonical way of representing a function that maps one base value to some other base value and everything else to $\emptyset$. For instance given $\mathbf{f} = \texttt{green} \mapsto \texttt{yellow}$ of type **Atom** $\rightarrow$ **Atom**, **f** is a function such that $\mathbf{f}\ \texttt{green} = \texttt{yellow}$ and $\mathbf{f}\ a = \emptyset$ for every atom a different from `green`. The operator respects choice so for example $(\texttt{red} \| \texttt{green}) \mapsto (\texttt{yellow} \| \texttt{blue})$ expands to

$$\texttt{red} \mapsto \texttt{yellow} \parallel \texttt{red} \mapsto \texttt{blue} \parallel \texttt{green} \mapsto \texttt{yellow} \parallel \texttt{green} \mapsto \texttt{blue}$$

Function application also respects choice so applying this to an argument will result in a choice between applying each of the alternatives.

We can now rewrite the identity function. Recall that by our method for expanding definitions **id** is equal to

$$[\mathbf{f} : \tau \rightarrow \tau]\, ([\mathbf{x} : \tau]\, \mathbf{f}\ \mathbf{x} \Leftrightarrow \mathbf{x}); \mathbf{f}$$

Using the rule for function application we get

$$[\mathbf{f} : \tau \rightarrow \tau]\, ([\mathbf{x} : \tau]\, ([\mathbf{y} : \tau]\, \mathbf{f} \Leftrightarrow (\mathbf{x} \mapsto \mathbf{y}); \mathbf{y}) \Leftrightarrow \mathbf{x}); \mathbf{f}$$

Pointwise equality respects conjunction and variable introduction (assuming no variables become accidentally free or bound) so we can simplify.

$$[\mathbf{f} : \tau \rightarrow \tau]\, ([\mathbf{x} : \tau]\, [\mathbf{y} : \tau]\, \mathbf{f} \Leftrightarrow (\mathbf{x} \mapsto \mathbf{y}); \mathbf{y} \Leftrightarrow \mathbf{x}); \mathbf{f}$$

When two distinct variables are involved in a pointwise equality we can make progress by unifying them and resolving the equality as true.

$$[\mathbf{f} : \tau \rightarrow \tau]\, [\mathbf{x} : \tau]\, \mathbf{f} \Leftrightarrow (\mathbf{x} \mapsto \mathbf{x}); \mathbf{f}$$

Similarly, when a variable is being compared for equality with some base value ($b_1 \mapsto b_2$ is a base value when b_1 and b_2 are) we can resolve it by substituting that base value for the variable. This gives the reduced form:

$$\mathbf{id} = [\mathbf{x} : \tau]\, \mathbf{x} \mapsto \mathbf{x}$$

Using our previous abuse of notation, if $\tau = \mathbf{Atom}$ we have

$$\mathbf{id} \overset{...}{=} \texttt{red} \mapsto \texttt{red} \parallel \texttt{green} \mapsto \texttt{green} \parallel \texttt{yellow} \mapsto \texttt{yellow} \parallel \texttt{blue} \mapsto \texttt{blue} \parallel \ldots$$

Hence, a function is represented as a relation between inputs and outputs. This is not merely a philosophical observation, but has practical consequences as we shall see now.

Adjoints Every function in Algeo has a pseudo-inverse known as an *adjoint*. We define it as follows:

$$\begin{aligned} &\dagger : (\tau \to \tau') \to (\tau' \to \tau) \\ &\mathbf{f}^\dagger\ (\mathbf{f}\,\mathbf{x}) \Leftrightarrow \mathbf{x} \end{aligned}$$

The superscript notation $\mathbf{f}^\dagger$ is syntactic sugar for $\dagger\ \mathbf{f}$. This assertion can be rewritten into the following form:

$$(\mathbf{x} \mapsto \mathbf{y})^\dagger \Leftrightarrow \mathbf{y} \mapsto \mathbf{x}$$

Thus, the adjoint can also be thought of as a pointwise inverse: each base value mapping is inverted, but choice is respected. For example consider the following function:

$$\begin{aligned} &\mathbf{f} : \mathbf{Atom} \to \mathbf{Atom} \\ &\mathbf{f}\,\texttt{red} \Leftrightarrow \texttt{blue} \\ &\mathbf{f}\,\texttt{green} \Leftrightarrow \texttt{blue} \end{aligned}$$

We have $\mathbf{f} = \texttt{red} \mapsto \texttt{blue} \parallel \texttt{green} \mapsto \texttt{blue}$ so $\mathbf{f}^\dagger = \texttt{blue} \mapsto \texttt{red} \parallel \texttt{blue} \mapsto \texttt{green} = \texttt{blue} \mapsto (\texttt{red} \parallel \texttt{green})$. Clearly, $\mathbf{f}^\dagger$ is not an inverse of $\mathbf{f}$ in the sense of composing to the identity, but it can nevertheless be considered a reverse execution of $\mathbf{f}$ in a more general sense. Indeed, any function written in a classical reversible language can be written in Algeo in which case the adjoint will coincide with the classical inverse. The two concepts diverge only when choice is involved.

Entanglement The definition of adjoints mentions $\mathbf{f}$ twice. In general, when a variable is mentioned multiple times we consider the uses to be *entangled*. The reason is that variables range over base values so any choices must be factored out before substituting a value for the variable.

Consider the difference between $[\mathbf{x} : \tau]\,\mathbf{x} \mapsto \mathbf{x}$ and $([\mathbf{x} : \tau]\,\mathbf{x}) \mapsto ([\mathbf{x} : \tau]\,\mathbf{x})$. The former is the identity function while the latter is a function that maps every possible input to a choice between every possible output.

Note also that each use of a defined term is independent. For instance consider a function that swaps `red` and `green`:

$$\begin{aligned} &\mathbf{swap} : \mathbf{Atom} \to \mathbf{Atom} \\ &\mathbf{swap}\,\texttt{red} \Leftrightarrow \texttt{green} \\ &\mathbf{swap}\,\texttt{green} \Leftrightarrow \texttt{red} \end{aligned}$$

There is a difference between **swap** (**swap** `red`) and $[\mathbf{f}]\,\mathbf{f} \Leftrightarrow \mathbf{swap}; \mathbf{f}\ (\mathbf{f}\ \texttt{red})$. The former reduces to `red` while the latter reduces to $\emptyset$. This can be seen by simply substituting the definition of **swap** to get

$$[\mathbf{f}]\,(\mathbf{f} \Leftrightarrow (\texttt{red} \mapsto \texttt{green} \parallel \texttt{green} \mapsto \texttt{red})); \mathbf{f}\,(\mathbf{f}\,\texttt{red})$$

Distribute the choice:

$$([\mathbf{f}]\,\mathbf{f} \Leftrightarrow (\texttt{red} \mapsto \texttt{green}); \mathbf{f}\,(\mathbf{f}\,\texttt{red})) \parallel ([\mathbf{f}]\,\mathbf{f} \Leftrightarrow (\texttt{green} \mapsto \texttt{red}); \mathbf{f}\,(\mathbf{f}\,\texttt{red}))$$

Replace **f** by its only possible base value:

$$((\texttt{red} \mapsto \texttt{green})\ ((\texttt{red} \mapsto \texttt{green})\ \texttt{red})) \parallel ((\texttt{green} \mapsto \texttt{red})\ ((\texttt{green} \mapsto \texttt{red})\ \texttt{red}))$$

The effect of being forced to make choices globally is now clear; both branches reduce to $\emptyset$.

3.2.3 Products

Algeo also supports a product type $\tau_1 \otimes \tau_2$ with elements of the form $e_1 \otimes e_2$. The astute reader will notice that this sounds suspiciously similar to the function type $\tau_1 \to \tau_2$ which has elements of the form $e_1 \mapsto e_2$. Indeed, the two types are trivially equivalent with $e_1 \otimes e_2$ corresponding to $e_1 \mapsto e_2$, but differ in practice as follows:

- Functions support function application syntax (i.e. $e_1\ e_2$), products do not.
- Functions represent *code*, whereas products represent *data*; their equivalence is another way of saying that in Algeo *code is data is code.*
- A function type $\tau_1 \to \tau_2$ signifies the intention that τ_1 is input and τ_2 is output, whereas the corresponding product type does no such thing. In a hypothetical Quantum Algeo with support for scalars from arbitrary C^*-algebras (e.g. $\mathbb{C}$) this distinction becomes formal as we would need to track dualities explicitly.

3.2.4 Dataflow operations

Weight When a variable occurs by itself it is completely unentangled. Define the *weight* of a term as follows.

$$\begin{aligned} &\# : \tau \to \mathbf{Scalar} \\ &\#\ \mathbf{x} \end{aligned}$$

Intuitively this asserts that # applied to any base value is $\overline{1}$. We could equivalently have written the assertion as $\#\ \mathbf{x} \Leftrightarrow \overline{1}$.

Wildcard The adjoint of # maps $\overline{1}$ to a sum of all terms, i.e. to the wildcard $*$. Indeed we could equivalently define the wildcard in terms of this adjoint:

$$\begin{aligned} *_\tau &: \tau \\ *_\tau &\Leftrightarrow \#^\dagger \, \overline{1} \end{aligned}$$

Join We now introduce an important operator, the *join*:

$$\begin{aligned} \bowtie &: \tau \to \tau \to \tau \\ \mathbf{x} \bowtie \mathbf{x} &\Leftrightarrow \mathbf{x} \end{aligned}$$

Given two base values, if they are equal it returns that unique value, otherwise it fails. For instance

$$(\texttt{red} \parallel \texttt{green} \parallel \texttt{blue}) \bowtie (\texttt{green} \parallel \texttt{blue} \parallel \texttt{black}) = \texttt{green} \parallel \texttt{blue}$$

Compared to $\Leftrightarrow$, which computes *how much* two values are alike, $\bowtie$ computes *where* they are alike.

Duplication Conversely, the function that duplicates base values is defined as follows:

$$\begin{aligned} \mathbf{dup} &: \tau \to \tau \otimes \tau \\ \mathbf{dup}\ \mathbf{x} &\Leftrightarrow \mathbf{x} \otimes \mathbf{x} \end{aligned}$$

It is a matter of perspective if **dup** is the same as or the adjoint of $\bowtie$, due to symmetry. The functionality is in any case the same, namely constraining three base values to be equal.

Linearity A variable is used *linearly* if it is used exactly twice. Intuitively one use is input (typically left of $\Leftrightarrow$) while the other is output (typically right of $\Leftrightarrow$), but the flexibility of definitions means that there is no formal distinction. With help from the operations of #, $*$, $\bowtie$ and **dup** it is in fact possible to rewrite any program such that all variables are used linearly. In this way it possible to make the nonlinear effects explicit.

3.2.5 Booleans

We now give some concrete examples using booleans to reinforce previous concepts and to introduce sum types.

Recall that we have a type of scalars, **Scalar**. Elements have the form $\overline{n}$ where n is a number, though only $\overline{1}$ is a base value. Algeo also has sum types, $\tau_1 \oplus \tau_2$, whose base values are $\mathbf{inl}(b)$ (with b being a base value of τ_1) and $\mathbf{inr}(b)$ (with b being a base value of τ_2).

Take **Bool** to be an alias for **Scalar** $\oplus$ **Scalar**, and define:

$$\begin{aligned} \mathbf{true} &: \mathbf{Bool} & \qquad \mathbf{false} &: \mathbf{Bool} \\ \mathbf{true} &\Leftrightarrow \mathbf{inl}(\overline{1}) & \qquad \mathbf{false} &\Leftrightarrow \mathbf{inr}(\overline{1}) \end{aligned}$$

Negation Negation of booleans can be written using two clauses.

$$\begin{aligned}&\mathbf{not} : \mathbf{Bool} \to \mathbf{Bool}\\ &\mathbf{not\ true} \Leftrightarrow \mathbf{false}\\ &\mathbf{not\ false} \Leftrightarrow \mathbf{true}\end{aligned}$$

The *adjoint* of **not** is then given by:

$$\begin{aligned}&\mathbf{not}^\dagger : \mathbf{Bool} \to \mathbf{Bool}\\ &\mathbf{not}^\dagger\ (\mathbf{not\ x}) \Leftrightarrow \mathbf{x}\end{aligned}$$

Note that this definition implicitly quantifies over $\mathbf{x} : \mathbf{Bool}$ and recall that such a quantification represents a nondeterministic choice of a *base value*. In **Bool** the base values are **true** and **false** so the definition is equal to

$$\mathbf{not}^\dagger\ (\mathbf{not\ true}) \Leftrightarrow \mathbf{true} \parallel \mathbf{not}^\dagger\ (\mathbf{not\ false}) \Leftrightarrow \mathbf{false}$$

By the definition of **not** this reduces to

$$\mathbf{not}^\dagger\ \mathbf{false} \Leftrightarrow \mathbf{true} \parallel \mathbf{not}^\dagger\ \mathbf{true} \Leftrightarrow \mathbf{false}$$

which is equivalent to having two clauses:

$$\mathbf{not}^\dagger\ \mathbf{false} \Leftrightarrow \mathbf{true} \qquad\qquad \mathbf{not}^\dagger\ \mathbf{true} \Leftrightarrow \mathbf{false}$$

We can thus establish that $\mathbf{not}^\dagger = \mathbf{not}$, as expected.

Alternative negation As an example of $\Leftrightarrow$ being merely a conventional—but by no means mandatory—approach to definitions, **not** could equivalently have been defined as:

$$\begin{aligned}&\mathbf{istrue}, \mathbf{isfalse} : \mathbf{Bool} \to \mathbf{Scalar} \qquad && \mathbf{not} : \mathbf{Bool} \to \mathbf{Bool}\\ &\mathbf{istrue\ true} && \mathbf{isfalse}\ (\mathbf{not\ true})\\ &\mathbf{isfalse\ false} && \mathbf{istrue}\ (\mathbf{not\ false})\end{aligned}$$

Binary operations Next, we define conjunction and disjunction:

$$\begin{aligned}&\mathbf{and}, \mathbf{or} : \mathbf{Bool} \to \mathbf{Bool} \to \mathbf{Bool}\\ &\mathbf{and\ true\ x} \Leftrightarrow \mathbf{x}\\ &\mathbf{and\ false} * \Leftrightarrow \mathbf{false}\\ &\mathbf{not}\ (\mathbf{or\ x\ y}) \Leftrightarrow \mathbf{and}\ (\mathbf{not\ x})\ (\mathbf{not\ y})\end{aligned}$$

Multiplicities A nondeterministic choice between copies of the same value like $\mathbf{false} \parallel \mathbf{false}$ can also be written $\overline{2}; \mathbf{false}$. We say that the *multiplicity* of **false** in this result is 2. In general, multiplicities can also be negative so, e.g., $\overline{-1}; \mathbf{false}$ represents -1 occurences of **false**. This can be used to cancel out positive

$$\begin{array}{lcl}
\tau & ::= & \textbf{Atom} \mid \textbf{Empty} \mid \textbf{Scalar} \mid \tau_1 \to \tau_2 \mid \tau_1 \oplus \tau_2 \mid \tau_1 \otimes \tau_2 \\
b & ::= & x \mid a \mid b_1\ b_2 \mid b_1; b_2 \mid \textbf{inl}(b) \mid \textbf{inr}(b) \mid b_1 \otimes b_2 \mid b_1 \mapsto b_2 \\
d & ::= & x \mid a \mid d_1\ d_2 \mid d_1; d_2 \mid \textbf{inl}(d) \mid \textbf{inr}(d) \mid d_1 \otimes d_2 \mid d_1 \mapsto d_2 \mid \emptyset \mid d_1 \bowtie d_2 \\
e & ::= & x \mid a \mid e_1\ e_2 \mid e_1; e_2 \mid \textbf{inl}(e) \mid \textbf{inr}(e) \mid e_1 \otimes e_2 \mid e_1 \mapsto e_2 \mid \emptyset \mid e_1 \bowtie e_2 \\
 & & \overline{n} \mid e_1 \parallel e_2 \mid [x : \tau]\, e
\end{array}$$

Figure 3.1: Syntax of types and terms.

multiplicities. For instance, we have **false** $\parallel$ ($\overline{-1}$; **false**) $= \emptyset$ (an empty result). Thus, negative multiplicities allow another kind of reversal via cancellation. To see this in action, consider the following alternative definition of conjunction:

and $* * \Leftrightarrow$ **false**	Conjunction generally returns **false**
and true true $\Leftrightarrow$ ($\overline{-1}$; **false**)	Not when both arguments are **true**, though
and true true $\Leftrightarrow$ **true**	In that case the result should be in fact be **true**

Generally, functions defined in Algeo are linear in the sense that they respect nondeterminism and multiplicities, corresponding to addition and scalar multiplication, respectively. Even before we know the definition of some function **f** we can say that **f** (**true** $\parallel$ **false**) = **f true** $\parallel$ **f false**. Now suppose that the definition is

$$\mathbf{f\ x} \Leftrightarrow \mathbf{and\ x\ (not\ x)}.$$

It is clear that **f true** = **f false** = **false** and therefore **f** (**true** $\parallel$ **false**) = $\overline{2}$; **false**. Even though **f** uses its argument twice and the argument is a nondeterministic choice between **true** and **false** the two uses of **x** are *entangled* and have to make the same nondeterministic choices.

3.3 Syntax and Semantics

The syntax of types and terms are given in Figure 3.1. Alternatives ($\parallel$) have the lowest precedence. Aggregrations ($[x : \tau] \ldots$) extend all the way to right. We employ the following conventions: τ is a type, e is an expression, d is a duplicable expression (see below for further details), b is a base value, a is an atom, n is a number and x, y and z are variables. Any b is also a d, and any d is also an e.

Intuitively base values represent deterministic computations that yield a value exactly once. Duplicable expressions are deterministic computations that either produce a base value or fail. Expressions in general represent nondeterministic computations that might produce any number of results. Variables are thought of as ranging over base values, although we will sometimes carefully substitute nonbase values.

We now describe the constructs of the language. An axiomatic semantics is given in Section 3.3.2. All operations are *linear* in the sense that they respect failure ($\emptyset$), alternatives ($\|$) and conjunction (;). For instance, $\bowtie$ is linear in each component so in particular $\emptyset \bowtie e = \emptyset$, $(e_1 \| e_2) \bowtie e_3 = (e_1 \bowtie e_3) \| (e_2 \bowtie e_3)$ and $(e_1; e_2) \bowtie e_3 = e_1; (e_2 \bowtie e_3)$. Hence, understanding these operators reduces to understanding their actions on base values.

- $\emptyset$ is *failure*. It aborts the computation.
- $e_1; e_2$ is *biased conjunction*. The first component is evaluated to a base value, which is discarded. The result of the biased conjunction is then the second component.
- $e_1 \bowtie e_2$ is *join*. It computes the intersection of the two arguments. In particular, the intersection of two base values is their unique value when equal and failure otherwise.
- $e_1 \otimes e_2$ is a *pair*.
- $e_1 \mapsto e_2$ is a *mapping*. It is a function that maps every base value in e_1 to every base value in e_2.
- **inl**(e) and **inr**(e) are left and right injections for the $\oplus$ type.
- $e_1 \| e_2$ is *alternative*. It represents a nondeterministic choice between e_1 and e_2.
- $[x : \tau]\, e$ is *aggregation*. It represents a nondeterministic choice of a base value $b : \tau$ which is substituted for x in e.
- $\overline{n}$ is a *number*. It represents a computation that succeeds n times. Note that negative values of n are possible. In general, depending on the choice of ring, n can also be rational or even complex.

We will also need the following syntactic sugar:

$e_1 \Leftrightarrow e_2 = e_1 \bowtie e_2; \overline{1}$	Pointwise unification of e_1 and e_2
$e_1 \setminus\!\!\setminus e_2 = e_1 \| \overline{-1}; e_2$	Collect the results of e_1 but subtract the results from e_2
$*_\tau = [x : \tau]\, x$	Wildcard, acts as the unit for $\bowtie$
$e^\perp = * \setminus\!\!\setminus e$	Everything except e

Beware: some constructs, e.g. $\Leftrightarrow$, use unfamiliar notation. This is done deliberately to show that these constructs represent new and unfamiliar concepts. A good rule of thumb is that the familiar-looking syntax like **inl**(e) means roughly what one would expect, whereas the unfamiliar syntax like $\Leftrightarrow$ has no simple well-known analogue.

In most languages the notion of function embodies both the introduction of variables and the mapping of those variables to some result. In Algeo, by contrast, these are separate concerns. Variable introduction is handled by $[x : \tau]$

whereas mappings are constructed by expressions of the form $e_1 \mapsto e_2$. This separation of concerns is the vital ingredient that makes Algeo so powerful.

Finally, we need to explain how top-level definitions are encoded as expressions. Suppose we define $x : \tau$ by the clauses $e_1, \ldots, e_n$, each of them typeable as $x : \tau \vdash e_i : \textbf{Scalar}$. Intuitively, x refers to (a single component of) the object we are defining.

Given such a top-level definition and a program e that can refer to x the desugared version is e with $[x : \tau]\,(e_1 \,\|\, \cdots \,\|\, e_n)$ substituted for x which we write as:

$$e^{x := [x:\tau]\,(e_1 \| \cdots \| e_n)}$$

This construction works by summing over all basis elements of τ subject to the conditions imposed by $e_1 \,\|\, \cdots \,\|\, e_n$. The sum represents the totality of the object we are defining. Each use of x in the program is replaced with this totality so each copy is independent. Note that $\Leftrightarrow$ is not mentioned and has no special status in this regard; it is merely an operator which happens to be useful for imposing suitable constraints when giving definitions.

Figure 3.2 shows some basic and fundamental functions. While identity and composition are similar to their definition in any functional language, the definition of adjoint $(-^{\dagger})$ seems very strange from a functional perspective and further seems to imply that all functions are injective—which isn't so! The trick to understanding this definition is that f quantifies over base values of the form $b_1 \mapsto b_2$ (and *not* entire functions), while $\mathbf{id}$ masquerades over the sum of all base values of the form $b \mapsto b$. In this way, we could just as well define $(\cdot^{\dagger})$ as $(x \mapsto y)^{\dagger} \circ (x \mapsto y) \Leftrightarrow (x \mapsto x)$ or even the more familiar $(x \mapsto y)^{\dagger} \Leftrightarrow (y \mapsto x)$.

A function $\mathbf{f}$ is *unitary* if $\mathbf{f}^{\dagger} \circ \mathbf{f} = \mathbf{id}$ and $\mathbf{f} \circ \mathbf{f}^{\dagger} = \mathbf{id}$, i.e. if $\mathbf{f}^{\dagger}$ is a two-sided inverse of $\mathbf{f}$. That this is not the case for every function reveals why $\Leftrightarrow$ should not be mistaken for equality! The unitaries include many interesting examples, including all classically reversible functions as well as all quantum circuits.

3.3.1 Type System

The type system is seen in Figure 3.3 and consists of a single judgement $\Gamma \vdash e : \tau$ stating that in type environment Γ the expression e has type τ. These rules should not be surprising, at least for a classical programming language. However, Algeo functions represent linear maps, so why does the type system not track variable use? The reason is that duplication and deletion are relatively harmless operations. Duplicating a value by using a variable multiple times creates *entangled* copies. They still refer to the same bound variables so any nondeterministic choice is made globally for all copies. Unused variables will still be bound in an aggregration and ultimately the multiplicity of the result will be scaled by the dimension of the type. Thus, such variables are not simply forgotten.

$\mathbf{id} : \tau \to \tau$
$\mathbf{id}\ \mathbf{x} \Leftrightarrow \mathbf{x}$

$(\circ) : (\tau_1 \to \tau_2) \to (\tau_2 \to \tau_3) \to (\tau_1 \to \tau_3)$
$(\mathbf{f} \circ \mathbf{g})\ \mathbf{x} \Leftrightarrow \mathbf{f}\ (\mathbf{g}\ \mathbf{x})$

$(-^{\dagger}) : (\tau_1 \to \tau_2) \to \tau_2 \to \tau_1$
$\mathbf{f}^{\dagger} \circ \mathbf{f} \Leftrightarrow \mathbf{id}$

$\mathbf{fst} : \tau_1 \otimes \tau_2 \to \tau_1$
$\mathbf{fst}\ (\mathbf{x} \otimes \mathbf{y}) \Leftrightarrow \mathbf{x}$

$\mathbf{snd} : \tau_1 \otimes \tau_2 \to \tau_2$
$\mathbf{snd}\ (\mathbf{x} \otimes \mathbf{y}) \Leftrightarrow \mathbf{y}$

$\mathbf{case} : (\tau_1 \to \tau) \to (\tau_2 \to \tau) \to \tau_1 \oplus \tau_2 \to \tau$
$\mathbf{case}\ \mathbf{f}\ \mathbf{g}\ \mathbf{inl}(\mathbf{x}) \Leftrightarrow \mathbf{f}\ \mathbf{x}$
$\mathbf{case}\ \mathbf{f}\ \mathbf{g}\ \mathbf{inr}(\mathbf{y}) \Leftrightarrow \mathbf{g}\ \mathbf{y}$

$\mathbf{threewayjoin} : \tau_1 \otimes \tau_2 \to \tau_1 \otimes \tau_3 \to \tau_2 \otimes \tau_3 \to \tau_1 \otimes \tau_2 \otimes \tau_3$
$\mathbf{threewayjoin}\ (\mathbf{x} \otimes \mathbf{y})\ (\mathbf{x}' \otimes \mathbf{z})\ (\mathbf{y}' \otimes \mathbf{z}') \Leftrightarrow (\mathbf{x} \bowtie \mathbf{x}') \otimes (\mathbf{y} \bowtie \mathbf{y}') \otimes (\mathbf{z} \bowtie \mathbf{z}')$

$\mathbf{coinflip} : \mathbf{Atom}$
$\overline{0.5}; \mathbf{coinflip} \Leftrightarrow \texttt{heads}$
$\overline{0.5}; \mathbf{coinflip} \Leftrightarrow \texttt{tails}$

$\mathbf{hadamard} : K \oplus K \to K \oplus K$
$\mathbf{hadamard}\ \mathbf{inl}(\overline{1}) \Leftrightarrow \mathbf{inl}(\overline{1/\sqrt{2}})$
$\mathbf{hadamard}\ \mathbf{inl}(\overline{1}) \Leftrightarrow \mathbf{inr}(\overline{1/\sqrt{2}})$
$\mathbf{hadamard}\ \mathbf{inr}(\overline{1}) \Leftrightarrow \mathbf{inl}(\overline{1/\sqrt{2}})$
$\mathbf{hadamard}\ \mathbf{inr}(\overline{1}) \Leftrightarrow \mathbf{inr}(\overline{-1/\sqrt{2}})$

Figure 3.2: Some basic functions in Algeo (identity, composition and adjoints, projections) and slightly more advanced functions showcasing database, probabistic and quantum programming.

3.3.2 Axiomatic Semantics

We now present the semantics of Algeo as a set of equations between expressions, see Figure 3.4. Equations hold only when well-typed and well-scoped. For instance, $\emptyset = \overline{0}$ implicitly assumes that the $\emptyset$ in question is typed as **Scalar**. The semantics is parametric over the choice of numbers, provided that the numbers form a ring of characteristic 0 (i.e. are the integers or an extension of them) and that for any type τ there is a number $\mathbf{dim}(\tau)$ such that:

$$\begin{aligned} \mathbf{dim}(\tau_1 \oplus \tau_2) &= \mathbf{dim}(\tau_1) + \mathbf{dim}(\tau_2) & \mathbf{dim}(\mathbf{Empty}) &= 0 \\ \mathbf{dim}(\tau_1 \otimes \tau_2) &= \mathbf{dim}(\tau_1) \cdot \mathbf{dim}(\tau_2) & \mathbf{dim}(\mathbf{Scalar}) &= 1 \\ \mathbf{dim}(\tau_1 \to \tau_2) &= \mathbf{dim}(\tau_1) \cdot \mathbf{dim}(\tau_2) \end{aligned}$$

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{}{\Gamma \vdash \overline{n} : \textbf{Scalar}} \quad \frac{}{\Gamma \vdash a : \textbf{Atom}} \quad \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \quad \frac{}{\Gamma \vdash \emptyset : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \parallel e_2 : \tau} \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \bowtie e_2 : \tau} \quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{inl}(e) : \tau_1 \oplus \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{inr}(e) : \tau_1 \oplus \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \mapsto e_2 : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 ; e_2 : \tau} \quad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \quad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash [x : \tau']\, e : \tau}$$

Figure 3.3: The type system of Algeo.

As the "standard model" we propose $\mathbb{Z}[\omega]$, i.e. polynomials over the integers in one variable ω. We define $\textbf{dim}(\textbf{Atom}) = \omega$ together with the equations above.

Operations are defined to be either *linear* or *bilinear*. For an operation o this entails:

$$o(\emptyset) = \emptyset \qquad o([x : \tau]\, e) = [x : \tau]\, o(e)$$
$$o(e_1; e_2) = e_1; o(e_2) \qquad o(e_1 \parallel e_2) = o(e_1) \parallel o(e_2)$$

A binary operator $(- \odot -)$ is bilinear if both $(e_1 \odot -)$ and $(- \odot e_2)$ are linear.

3.3.3 Justification of the Semantics

All axioms are based on intuition from finite-dimensional types, i.e. types whose set of base values is finite. The idea is to extend this to infinite-dimensional types, but with a flavour of "compactness" keeping the properties of finite-dimensionality. While it is possible to aggregate over infinite types, any given expression will only mention a finite number of distinct base values. We avoid contradiction arising from this approach by *not insisting* that every aggregation be reducible. This reveals a possible connection between Algeo and nonstandard analysis.

Most axioms should be uncontroversial, but some deserve elaboration. Perhaps the most unusual one is $[x : \tau]\, e = e^{x:=d} \setminus\!\!\setminus e^{x:=\emptyset} \parallel [y : \tau]\, e^{x:=y \setminus\!\setminus y \bowtie d}$. Usually we will exploit the equality $y \setminus\!\!\setminus y \bowtie d = y \bowtie d^{\perp}$ to get the rule $[x : \tau]\, e = e^{x:=d} \setminus\!\!\setminus e^{x:=\emptyset} \parallel [y : \tau]\, e^{x:=y \bowtie d^{\perp}}$. Firstly, note that d is only used in the substitutions, so some amount of prescience is required to choose a suitable d. The intuition is that we are splitting into cases depending on whether x is equal to d or not. To see how this works in the finite-dimensional case suppose $\tau = \textbf{Scalar} \oplus \cdots \oplus \textbf{Scalar}$ (n copies). Then τ has n distinct base values which we shall refer to as $b_1, \ldots, b_n$. The following reasoning shows how the statement can be shown directly from the other axioms in the finite case. A d

Biased conjunction
$\overline{1}; e = e$
$x; e = e$
$\mathbf{inl}(e_1); e_2 = e_1; e_2$
$\mathbf{inr}(e_1); e_2 = e_1; e_2$
$e_1 \otimes e_2; e_3 = e_1; e_2; e_3$
$e_1 \mapsto e_2; e_3 = e_1; e_2; e_3$

Function application
$(e_1 \mapsto e_2)\ e' = e_1 \bowtie e'; e_2$

Numbers
$\emptyset = \overline{0}$
$\overline{m} \parallel \overline{n} = \overline{m+n}$
$\overline{m}; \overline{n} = \overline{m \cdot n}$
$[x : \tau]\, \overline{1} = \overline{\mathbf{dim}(\tau)}$

Aggregation
$[x : \mathbf{Empty}]\, e = \emptyset$
$[x : \mathbf{Scalar}]\, e = e^{x:=\overline{1}}$
$[x : \tau_1 \oplus \tau_2]\, e = ([y : \tau_1]\, e^{x:=\mathbf{inl}(y)}) \parallel ([y : \tau_2]\, e^{x:=\mathbf{inr}(y)})$
$[x : \tau_1 \otimes \tau_2]\, e = [x_1 : \tau_1]\,[x_2 : \tau_2]\, e^{x:=x_1 \otimes x_2}$
$[x : \tau_1 \to \tau_2]\, e = [x_1 : \tau_1]\,[x_2 : \tau_2]\, e^{x:=x_1 \mapsto x_2}$
$[x : \tau]\, x \bowtie e; x = e$
$[x : \tau]\, e = e^{x:=d} \setminus\!\!\setminus e^{x:=\emptyset} \parallel [y : \tau]\, e^{x:=y \setminus\!\setminus y \bowtie d}$

Extraction

Join
$d \bowtie d = d$
$\mathbf{inl}(e) \bowtie \mathbf{inl}(e') = \mathbf{inl}(e \bowtie e')$
$\mathbf{inr}(e) \bowtie \mathbf{inr}(e') = \mathbf{inl}(e \bowtie e')$
$\mathbf{inl}(e) \bowtie \mathbf{inr}(e') = \emptyset$
$\mathbf{inr}(e) \bowtie \mathbf{inl}(e') = \emptyset$
$(e_1 \otimes e_2) \bowtie (e_1' \otimes e_2') = (e_1 \bowtie e_1') \otimes (e_2 \bowtie e_2')$
$(e_1 \mapsto e_2) \bowtie (e_1' \mapsto e_2') = (e_1 \bowtie e_1') \mapsto (e_2 \bowtie e_2')$
$\langle e_1 \rangle \bowtie \langle e_2 \rangle = (\diamond(e_1 \setminus\!\!\setminus e_2))^{\perp}; \langle e_1 \rangle$

Linearity
$([x : \tau] -)$, $(!-)$,
$\mathbf{inl}(-)$ and $\mathbf{inr}(-)$ are linear
$(-;-)$, $(--)$, $(- \bowtie -)$,
$(- \otimes -)$ and $(- \mapsto -)$ are bilinear

Figure 3.4: Axiomatic semantics of Algeo.

of type τ will either be some b_i or $\emptyset$. Without loss of generality let $d = b_1$ (if $d = \emptyset$ the statement is trivial). We then have:

$$\begin{aligned}
[x : \tau]\, e &= e^{x:=b_1} \parallel e^{x:=b_2} \parallel \ldots \parallel e^{x:=b_n} \\
&= e^{x:=b_1} \setminus\!\!\setminus e^{x:=\emptyset} \parallel e^{x:=\emptyset} \parallel e^{x:=b_2} \parallel \ldots \parallel e^{x:=b_n} \\
&= e^{x:=b_1} \setminus\!\!\setminus e^{x:=\emptyset} \parallel e^{x:=b_1 \bowtie b_1{}^{\perp}} \parallel e^{x:=b_2 \bowtie b_1{}^{\perp}} \parallel \ldots \parallel e^{x:=b_n \bowtie b_1{}^{\perp}} \\
&= e^{x:=b_1} \setminus\!\!\setminus e^{x:=\emptyset} \parallel [y : \tau]\, e^{x:=y \bowtie b_1{}^{\perp}}
\end{aligned}$$

0	$\emptyset$
$x + y$	$x \parallel y$
$n \cdot x$	$\overline{n}; x$
1	$*$
$x \cdot y$	$x \bowtie y$
$\langle x \mid y \rangle$	$x \Leftrightarrow y$

Table 3.1: Linear algebra versus Algeo

3.3.4 Derived Equations and Evaluation

The semantic equations in Figure 3.4 are not reduction rules, although most of them embody some kind of reduction when read from left to right. They can be used for evaluation as well as deriving new equations.

As an example of a derived equation consider $[x : \tau]\, x \bowtie b; e = e^{x:=b}$. This property states that if a variable is unconditionally subject to a join constraint with a base value, we may dispense with the variable and simply substitute that value. The main idea is to case-split on whether or not x equals b. The last line exploits that all base values are left identities for $(;)$.

$$\begin{aligned}[x : \tau]\, x \bowtie b; e &= (x \bowtie b; e)^{x:=b} \mathbin{\backslash\!\backslash} (x \bowtie b; e)^{x:=\emptyset} \parallel [y : \tau]\, (x \bowtie b; e)^{x:=y \bowtie b^{\perp}} \\ &= b \bowtie b; e^{x:=b} \mathbin{\backslash\!\backslash} \emptyset \bowtie b; e^{x:=b} \parallel [y : \tau]\, y \bowtie b^{\perp} \bowtie b; e^{x:=b} \\ &= b; e^{x:=b} \mathbin{\backslash\!\backslash} \emptyset \parallel [y : \tau]\, \emptyset = b; e^{x:=b} = e^{x:=b}\end{aligned}$$

A generalisation of this lemma suggests that we can emulate the usual operational interpretation of logic programming where variables are instantiated based on unification constraints.

Indeed, the version of Algeo presented here (as opposed to the full version [6]) can likely be given an operational semantics similar to the Warren Abstract Machine [7] for Prolog. Purity and lack of recursion avoids many of the problems encountered for Prolog. Multiplicities need to tracked, which is unlikely to be an issue. More seriously, decisions need to made about how canonical the result should be. For instance, should a term equivalent to $\emptyset$ always reduce to $\emptyset$ or should it be allowed to produce something like $e \parallel \overline{-1}; e$? Should nonzero results always be listed in some globally consistent order? We will leave these questions to future work.

3.3.5 Relation to Linear Algebra

Many operations in Algeo are closely related to linear algebra, in particular K-algebras where K is the ring of elements of type **Scalar**. The correspondence can be seen in Table 3.1. Recall the common definition of the adjoint of f as the unique function $f^{\dagger}$ satisfying $\langle f(x) \mid y \rangle = \langle x \mid f^{\dagger}(y) \rangle$ for all x and y. Translating this to Algeo we might write it as $[\mathbf{x}]\, [\mathbf{y}]\, (\mathbf{f\, x} \Leftrightarrow \mathbf{y}) \Leftrightarrow (\mathbf{x} \Leftrightarrow \mathbf{f}^{\dagger}\, \mathbf{y})$, which turns out to be a perfectly good definition that is equivalent to our previous one. This gives a new perspective on what the inner product means in linear algebra.

3.4 Denotational Semantics

We have already given an axiomatic semantics describing how terms can be rewritten. However, how can we be sure that the system is "reasonable"? For instance, it might conceivably be the case that every term is provably equal to 0 due to some creative application of the rewrite rules. A superficially similar class of languages known as algebraic λ-calculi [8, 9] often run into this kind of problem.

One approach often taken for functional languages is to prove *progress* and *preservation* of an operational semantics. Not all of our rewrite rules can be interpreted operationally, though, and the question of progress is in any case more complicated when logic programming is involved.

Instead we provide a denotational semantics, consisting of a suitably chosen *semantic* category and a functor from the *syntactic* category of terms into that category. This settles the question of whether the rewrite system collapses. If two terms are provably equal then their interpretations will be equal as well; since the (image of the functor in the) semantic category is not trivial neither is the syntactic category.

3.4.1 The Category of Terms

We now give the details of how terms and type derivations give rise to a syntactic category. In the category TERM objects are type contexts and morphisms are (tuples of) type derivations. More explicitly, for environments $\Gamma = \tau_1, \dots, \tau_m$ and $\Delta = \tau'_1, \dots, \tau'_n$ a morphism $\varphi : \mathrm{Hom}_{\mathrm{TERM}}(\Gamma, \Delta)$ is a tuple $(\mathcal{T}_i)_{1 \leq i \leq n}$ of type derivations where $\mathcal{T}_i$ derives $\Gamma \vdash e_i : \tau'_i$. The terms e_i are part of the data of the morphism, but can always be inferred from the structure of the derivation.

Intuitively speaking, a morphism $\varphi : \mathrm{Hom}_{\mathrm{TERM}}(\Gamma, \Delta)$ contains the same information as a derivation of $\Gamma \vdash e : \tau'_1 \oplus \cdots \oplus \tau'_n$. In particular, when Δ is a singleton environment, i.e. $\Delta = \tau'$ for some τ', a morphism $\varphi : \mathrm{Hom}_{\mathrm{TERM}}(\Gamma, \tau')$ can be identified with a derivation of $\Gamma \vdash e : \tau'$.

This is not quite enough to represent the language faithfully as a category. We consider terms like $0 + 0$ and 0 to be equal so we should also consider the corresponding morphisms to be equal. The set of morphisms is therefore constructed by taking the set of type derivations and quotienting by equality of terms.

3.4.2 The Category of Spans

In order to construct the semantic category we will need to work with spans. A span of objects X and Y is a diagram of the form:

$$\begin{tikzcd} & A \arrow[dl, "f"'] \arrow[dr, "g"] & \\ X & & Y \end{tikzcd}$$

Making a span thus consists of choosing an apex, A, and two legs, f and g.

Given another span

$$\begin{tikzcd} & B \arrow[dl, "h"'] \arrow[dr, "k"] & \\ Y & & Z \end{tikzcd}$$

how should be composition be defined? We can certainly stitch the two diagrams together:

$$\begin{tikzcd} & A \arrow[dl, "f"'] \arrow[dr, "g"] & & B \arrow[dl, "h"'] \arrow[dr, "k"] & \\ X & & Y & & Z \end{tikzcd}$$

The resulting diagram is clearly not a span, but it is not too far from it either. Supposing we can find a span of A and B the following diagram is possible.

$$\begin{tikzcd} & & C \arrow[dl, "p"'] \arrow[dr, "q"] & & \\ & A \arrow[dl, "f"'] \arrow[dr, "g"] & & B \arrow[dl, "h"'] \arrow[dr, "k"] & \\ X & & Y & & Z \end{tikzcd}$$

The outer edges form a big span with C at the apex and $f \circ p$ and $k \circ q$ as the legs.

To define composition of spans in general we need to know that such a span always exists. For coherency reasons we should also require that the diagram commutes. Hence, we are looking for the *pullback* of g and h.

Together with the identity span

$$\begin{tikzcd} & X \arrow[dl, "\mathrm{id}"'] \arrow[dr, "\mathrm{id}"] & \\ X & & X \end{tikzcd}$$

this constitutes the structure of a category $\textsc{Span}_{\mathfrak{C}}$ for any category $\mathfrak{C}$ with pullbacks. Note that the objects of $\textsc{Span}_{\mathfrak{C}}$ are simply the objects of $\mathfrak{C}$, and any morphism $f : M \to N$ in $\mathfrak{C}$ can be lifted into $\textsc{Span}_{\mathfrak{C}}$ as follows:

$$\begin{tikzcd}
 & M \arrow[dl, "\mathrm{id}"'] \arrow[dr, "f"] & \\
M & & N
\end{tikzcd}$$

Also note that the category of spans has a canonical *dagger structure*, a way to turn any morphism $f : M \to N$ into a morphism $f^\dagger : N \to M$. This is achieved by simply swapping the two legs of the span.

There is an issue regarding equality of morphisms, though. Consider the composition of three spans, associated to the left:

$$\begin{tikzcd}
 & & & E \arrow[dl] \arrow[ddrr] & & & \\
 & & D \arrow[dl] \arrow[dr] & & & & \\
 & A \arrow[dl] \arrow[dr] & & B \arrow[dl] \arrow[dr] & & C \arrow[dl] \arrow[dr] & \\
X & & Y & & Z & & W
\end{tikzcd}$$

And to the right:

$$\begin{tikzcd}
 & & & E' \arrow[ddll] \arrow[dr] & & & \\
 & & & & D' \arrow[dl] \arrow[dr] & & \\
 & A \arrow[dl] \arrow[dr] & & B \arrow[dl] \arrow[dr] & & C \arrow[dl] \arrow[dr] & \\
X & & Y & & Z & & W
\end{tikzcd}$$

The resulting spans are not strictly equal, albeit they are equivalent via an associator. For this reason spans are often considered to form a 2-category, where this higher structure of coherence morphisms is made explicit. We consider it as an ordinary category by taking quotients and considering morphisms only up to equivalence.

For our purposes the base category is always $\textsc{WAlg}_K$, the category of weighted K-algebras and linear maps. Pullbacks always exist here and are quite easy to define. Given $g : A \to Y$ and $h : B \to Y$ as in the composition diagram above we take $C = \{x : A \oplus B \mid g(\mathbf{inl}(x)) = h(\mathbf{inr}(x))\}$ where p and q are the usual projections from the biproduct restricted to C.

Note that the monoidal structure on $\textsc{WAlg}_K$ given by K and $\otimes$ extends cleanly to $\textsc{Span}_{\textsc{WAlg}_K}$. The coherence morphisms are lifted like any other morphism as shown above.

3.4.3 Interpreting Types

We can now begin constructing our functor $\llbracket - \rrbracket$ to give a denotational semantics. The first step is to map objects to objects. Recall that objects in $\textsc{Term}$ are contexts and objects in $\textsc{Span}_{\textsc{Mod}_K}$ are K-modules.

For a general context $\Gamma = \tau_1, \dots, \tau_n$ we let $\llbracket \Gamma \rrbracket = \llbracket \tau_1 \rrbracket \otimes \cdots \otimes \llbracket \tau_n \rrbracket$ so it suffices to define the action on individual types:

$$\begin{aligned}
\llbracket - \rrbracket &: \textsc{Term} \to \textsc{Span}_{\textsc{Mod}_K} \\
\llbracket \mathbf{Atom} \rrbracket &= \mathbf{F}_K[\mathbb{N}] \\
\llbracket \mathbf{Empty} \rrbracket &= 0 \\
\llbracket \mathbf{Scalar} \rrbracket &= K \\
\llbracket \tau_1 \to \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket \otimes \llbracket \tau_2 \rrbracket \\
\llbracket \tau_1 \oplus \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket \oplus \llbracket \tau_2 \rrbracket \\
\llbracket \tau_1 \otimes \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket \otimes \llbracket \tau_2 \rrbracket
\end{aligned}$$

Some of these equations require elaboration.

Atoms need to be interpreted into some countable set, which is arbitrarily chosen to be $\mathbb{N}$.

Functions are mapped to tensor products, because functions are supposed to be just sums of singleton mappings. The introduction of spans is exactly about supporting this choice.

3.4.4 Interpreting Morphisms

With the action of $\llbracket - \rrbracket$ on objects established, we can study its effects on morphisms which is typed as follows:

$$\llbracket - \rrbracket : \mathrm{Hom}_{\textsc{Term}}(\Gamma, \Delta) \to \mathrm{Hom}_{\textsc{Span}_{\textsc{Mod}_K}}(\llbracket \Gamma \rrbracket, \llbracket \Delta \rrbracket)$$

For a general context $\Delta = \tau_1', \dots, \tau_n'$ a morphism has the shape $(\mathcal{T}_1, \dots, \mathcal{T}_n)$ where each $\mathcal{T}_i$ is a derivation of $\Gamma \vdash e_i : \tau_i'$. We let $\llbracket (\mathcal{T}_1, \dots, \mathcal{T}_n) \rrbracket = \llbracket \mathcal{T}_1 \rrbracket \otimes \cdots \otimes \llbracket \mathcal{T}_n \rrbracket$ so it remains to consider the singleton case when $\Delta = \tau'$ and the input is a single type derivation $\mathcal{T}$ of $\Gamma \vdash e : \tau'$.

The definition proceeds by case analysis on $\mathcal{T}$.

Variable

$$\mathcal{T} = \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}$$

The relevant value is picked and the rest of the environment is disposed of using the weight operator.

$$\begin{tikzcd}
 & \llbracket \Gamma \rrbracket \otimes \llbracket \tau \rrbracket \otimes \llbracket \Gamma' \rrbracket \arrow[dl, "\mathrm{id}"'] \arrow[dr, "\lambda(\sigma \otimes v \otimes \sigma').\#\sigma \cdot \#\sigma' \cdot v"] & \\
\llbracket \Gamma \rrbracket \otimes \llbracket \tau \rrbracket \otimes \llbracket \Gamma' \rrbracket & & \llbracket \tau \rrbracket
\end{tikzcd}$$

Atom

$$\mathcal{T} = \frac{}{\Gamma \vdash a : \mathbf{Atom}}$$

A constant atom expression simply disposes of the environment and returns that atom (recall that atoms are considered to be natural numbers in our interpretation).

$$\begin{tikzcd}
 & \llbracket \Gamma \rrbracket \arrow[dl, "\mathrm{id}"'] \arrow[dr, "\lambda\sigma.\#\sigma\cdot\{a\}"] & \\
\llbracket \Gamma \rrbracket & & \mathbf{F}_K[\mathbb{N}]
\end{tikzcd}$$

Application

$$\mathcal{T} = \frac{\overset{\mathcal{T}_1}{\Gamma \vdash e_1 : \tau' \to \tau} \quad \overset{\mathcal{T}_2}{\Gamma \vdash e_2 : \tau'}}{\Gamma \vdash e_1\ e_2 : \tau}$$

By induction we have:

$$\begin{tikzcd}
 & M_1 \arrow[dl, "f_1"'] \arrow[dr, "g_1"] & & & M_2 \arrow[dl, "f_2"'] \arrow[dr, "g_2"] & \\
\llbracket \Gamma \rrbracket & & \llbracket \tau' \rrbracket \otimes \llbracket \tau \rrbracket & \llbracket \Gamma \rrbracket & & \llbracket \tau' \rrbracket
\end{tikzcd}$$

The spans from $\mathcal{T}_1$ and $\mathcal{T}_2$ are combined using a tensor product. In the left leg the environment is duplicated by using multiplication backwards. The specifics of application happens in the right leg where the parameter and the argument are matched using the inner product to compute the multiplicitiy for the result.

$$\begin{tikzcd}
 & & M_1 \otimes M_2 \arrow[dl, "f_1\otimes f_2"'] \arrow[dr, "g_1\otimes g_2"] & & \\
 & \llbracket \Gamma \rrbracket \otimes \llbracket \Gamma \rrbracket \arrow[dl, "(\cdot)"'] & & \llbracket \tau' \rrbracket \otimes \llbracket \tau \rrbracket \otimes \llbracket \tau' \rrbracket \arrow[dr, "\lambda(y\otimes x\otimes y').(y\diamond y')\cdot x"] & \\
\llbracket \Gamma \rrbracket & & & & \llbracket \tau \rrbracket
\end{tikzcd}$$

Sequencing

$$\mathcal{T} = \frac{\overset{\mathcal{T}_1}{\Gamma \vdash e_1 : \tau'} \quad \overset{\mathcal{T}_2}{\Gamma \vdash e_2 : \tau}}{\Gamma \vdash e_1 ; e_2 : \tau}$$

By induction we have:

$$\begin{tikzcd}
 & M_1 \arrow[dl, "f_1"'] \arrow[dr, "g_1"] & & & M_2 \arrow[dl, "f_2"'] \arrow[dr, "g_2"] & \\
\llbracket \Gamma \rrbracket & & \llbracket \tau' \rrbracket & \llbracket \Gamma \rrbracket & & \llbracket \tau \rrbracket
\end{tikzcd}$$

This construction is similar to the previous case. The result of evaluating the first expression is disposed of using the weight operator.

$$\begin{tikzcd}
 & & M_1 \otimes M_2 \arrow[dl, "f_1\otimes f_2"'] \arrow[dr, "g_1\otimes g_2"] & & \\
 & \llbracket\Gamma\rrbracket \otimes \llbracket\Gamma\rrbracket \arrow[dl, "(\cdot)"'] & & \llbracket\tau'\rrbracket \otimes \llbracket\tau\rrbracket \arrow[dr, "\lambda(x\otimes y).\#x\cdot y"] & \\
\llbracket\Gamma\rrbracket & & & & \llbracket\tau\rrbracket
\end{tikzcd}$$

Join

$$\mathcal{T} = \frac{\overset{\mathcal{T}_1}{\Gamma \vdash e_1 : \tau} \quad \overset{\mathcal{T}_2}{\Gamma \vdash e_2 : \tau}}{\Gamma \vdash e_1 \bowtie e_2 : \tau}$$

By induction we have:

$$\begin{tikzcd}
 & M_1 \arrow[dl, "f_1"'] \arrow[dr, "g_1"] & & & M_2 \arrow[dl, "f_2"'] \arrow[dr, "g_2"] & \\
\llbracket\Gamma\rrbracket & & \llbracket\tau\rrbracket & \llbracket\Gamma\rrbracket & & \llbracket\tau\rrbracket
\end{tikzcd}$$

Join uses the same approach to duplicating environments. The join operation itself is implemented using the multiplication from the algebra structure, which results in an elegant symmetric span.

$$\begin{tikzcd}
 & & M_1 \otimes M_2 \arrow[dl, "f_1\otimes f_2"'] \arrow[dr, "g_1\otimes g_2"] & & \\
 & \llbracket\Gamma\rrbracket \otimes \llbracket\Gamma\rrbracket \arrow[dl, "(\cdot)"'] & & \llbracket\tau\rrbracket \otimes \llbracket\tau\rrbracket \arrow[dr, "(\cdot)"] & \\
\llbracket\Gamma\rrbracket & & & & \llbracket\tau\rrbracket
\end{tikzcd}$$

Alternative

$$\mathcal{T} = \frac{\overset{\mathcal{T}_1}{\Gamma \vdash e_1 : \tau} \quad \overset{\mathcal{T}_2}{\Gamma \vdash e_2 : \tau}}{\Gamma \vdash e_1 \parallel e_2 : \tau}$$

By induction we have:

$$\begin{tikzcd}
 & M_1 \arrow[dl, "f_1"'] \arrow[dr, "g_1"] & & & M_2 \arrow[dl, "f_2"'] \arrow[dr, "g_2"] & \\
\llbracket\Gamma\rrbracket & & \llbracket\tau\rrbracket & \llbracket\Gamma\rrbracket & & \llbracket\tau\rrbracket
\end{tikzcd}$$

The result is the sum of the two spans, which is constructed with a biproduct at the apex. Each leg consists of a case analysis to select the appropriate morphism.

$$\begin{tikzcd}
 & M_1 \oplus M_2 \arrow[dl, "{[f_1,f_2]}"'] \arrow[dr, "{[g_1,g_2]}"] & \\
[\![\Gamma]\!] & & [\![\tau]\!]
\end{tikzcd}$$

Aggregation

$$\mathcal{T} = \frac{\overset{\mathcal{T}_1}{\Gamma, x : \tau' \vdash e : \tau}}{\Gamma \vdash [x : \tau']\, e : \tau}$$

By induction we have:

$$\begin{tikzcd}
 & M \arrow[dl, "f"'] \arrow[dr, "g"] & \\
[\![\Gamma]\!] \otimes [\![\tau']\!] & & [\![\tau]\!]
\end{tikzcd}$$

The introduction of the extra variable is dealt with in the left leg by disposing of it.

$$\begin{tikzcd}
 & & M \arrow[dl, "f"'] \arrow[ddr, "g"] & \\
 & [\![\Gamma]\!] \otimes [\![\tau']\!] \arrow[dl, "\lambda(\sigma\otimes x).\#x\cdot\sigma"'] & & \\
[\![\Gamma]\!] & & & [\![\tau]\!]
\end{tikzcd}$$

3.4.5 Coherence

Finally, we need to argue that $[\![-]\!]$ is well-defined. The issue is that when mapping out of a quotient set such as $\mathrm{Hom}_{\mathrm{TERM}}(\Gamma, \Delta)$ it is necessary to also demonstrate that equalities are preserved. A proof of this is future work.

Note that this property would sometimes be referred to as *soundness* of the axiomatic semantics. In our categorical setting this is simply part of what it means for $[\![-]\!]$ to be a well-defined functor. Conversely, there is also the question of *completeness*: can every equality between module elements also be derived using the axiomatic semantics? For the full Algeo language [6] this is certainly not the case, but for the restricted version presented here it is not immediately apparent why it would be impossible; future work is needed to decide this question.

3.5 Applications

Pattern matching. The flexibility of Algeo definitions allows us to define functions in many ways. This clearly includes definition by cases using ordinary pattern matching, but further exploration reveals that many extensions of pattern matching can be encoded as well. Note that in Algeo a pattern is nothing more than an expression written on the left-hand side of $\Leftrightarrow$. Simple examples include $*$ functioning as a wildcard, and definitions in general functioning as pattern synonyms. We list a number of common pattern matching features below along with their representation in Algeo.

- Functional patterns. These can be found in the functional logic language Curry as well as reversible functional languages like Theseus. Suppose $\mathbf{f}$ is defined by $\mathbf{f}\ (\mathbf{g}\ x) \Leftrightarrow e$. When $\mathbf{f}$ is applied to some e', effectively $\mathbf{g}$ will be run in reverse on e' and x bound to each result.

- View patterns. There is a Haskell extension providing patterns of the form $(f \Rightarrow p)$, which matches a value v if p matches $f\ v$. This syntax is definable as a function in Algeo:

$$(\Rightarrow) : (\tau \to \tau') \to \tau' \to \tau$$
$$\mathbf{f} \Rightarrow \mathbf{p} \Leftrightarrow (\mathbf{f}\ \mathbf{v} \Leftrightarrow \mathbf{p}; \mathbf{v})$$

  This is effectively a functional pattern where the function is run in reverse.

- Guard patterns. Some functional languages allow pattern guards like $\mathbf{f}\, p \mid c$ where the interpretation is that p is matched and then the boolean condition c is checked. Algeo, like any other logic language, can of course include conditions (i.e. expressions of type **Scalar**) in the body of a function. However, the flexibility of definitions means that we can write $\mathbf{f}\ (c; p)$ to signify that we consider the condition c to be part of the pattern.

- Alias patterns. Many functional languages support patterns like x **as** p where x is bound to the value matched by the entire pattern p. In Algeo the $\bowtie$ operator furnishes a much more general version of this. A pattern like $e_1 \bowtie e_2$ matches e_1 and e_2 simultaneously. When e_1 is a variable this encodes an alias pattern.

- Alternative patterns. Some languages allow patterns like $(p_1 \mid p_2)$ where p_1 and p_2 are nullary constructors. This is interpreted as equivalent to writing two clauses, one with p_1 and one with p_2. In Algeo any two patterns can be combined using $\parallel$. By linearity $\mathbf{f}\ (e_1 \parallel e_2) \Leftrightarrow e$ means exactly the same as $\mathbf{f}\, e_1 \Leftrightarrow e \parallel \mathbf{f}\, e_2 \Leftrightarrow e$.

- Negative patterns. Generally pattern matching is positive in the sense that patterns describe the shape of the data that we want to match. Matching everything except for some given pattern is typically done with a final default case; this works in functional languages where patterns are ordered and pattern matching works by finding the first match only. Algeo can describe negative patterns directly. Recall that $e^{\perp} = * \mathbin{\backslash\!\backslash} e$ for any expression e. As a pattern this can be interpreted as "everything except e". For example, deciding equality between two values can be defined as follows:

$$\begin{aligned}&\mathbf{eq?} : \tau \to \tau \to \mathbf{Scalar} \oplus \mathbf{Scalar}\\&\mathbf{eq?\ x\ x} \Leftrightarrow \mathbf{inl}(\overline{1})\\&\mathbf{eq?\ x\ x}^{\perp} \Leftrightarrow \mathbf{inr}(\overline{1})\end{aligned}$$

Linear algebra Given that Algeo is built on linear algebra, it will likely come as no surprise that expressing problems from linear algebra is often straightforward. An example of this is matrix multiplication. Given an $m \times n$ matrix A with entries a_{ij}, and a $n \times p$ matrix B with entries b_{ij}, the entries in the $n \times p$ matrix $C = AB$ are given by $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, i.e., summing over all possible ways of going first via A and then via B. In Algeo, a matrix from τ_1 to τ_2 is a value of type $\tau_1 \otimes \tau_2$ (i.e., a weighted sum of pairs of base values of τ_1 and τ_2), and their multiplication is expressed as

$$\begin{aligned}&(\cdot) : \tau_1 \otimes \tau_2 \to \tau_2 \otimes \tau_3 \to \tau_1 \otimes \tau_3\\&(x \otimes y) \cdot (y \otimes z) \Leftrightarrow x \otimes z\end{aligned}$$

where the implicit aggregation over y corresponds to the summation over k in the definition of c_{ij} from before. Another example is the trace (or sum of diagonal elements) of a square $n \times n$ matrix, $\mathrm{tr}(A) = \sum_{n=1} a_{nn}$, which can be slightly cryptically defined without a right hand side:

$$\begin{aligned}&\mathbf{tr} : (\tau \otimes \tau) \to \mathbf{Scalar}\\&\mathbf{tr}\ (x \otimes x)\end{aligned}$$

Again, note how the implicit aggregation over x corresponds to summation in the definition from linear algebra.

Polysets and polylogic. Polysets [4] are a generalisation of multisets which also permit elements to occur a *negative* number of times. This is useful for representing, e.g., (possibly unsynchronised) database states, with elements with *positive* multiplicity representing (pending) data insertions, and elements with *negative* multiplicity representing (pending) data deletions.

Polysets can be represented in Algeo via *polylogic*, an account of propositional logic relying on multiplicities of *evidence* and *counterexamples* (similar to *decisions* as in [10]). Concretely, a truth value in polylogic consists of an amount of evidence (injected to the left) and an amount of counterexamples (injected to the right). For example, $\perp$ has no evidence and a single counterexample, and dually for $\top$, as in

$$\begin{aligned}\perp, \top &\quad : \quad \mathbf{Scalar} \oplus \mathbf{Scalar}\\\perp &\quad = \quad \emptyset \oplus *\\\top &\quad = \quad * \oplus \emptyset\end{aligned}$$

As in [10], negation swaps evidence for counterexamples and vice versa, while the evidence of a conjunction is the join of the evidence of its conjuncts, with everything else counterexamples (disjunction dually):

$$\begin{array}{rcl}
\neg(e \oplus e') & = & e' \oplus e \\
(e_1 \oplus e_1') \wedge (e_2 \oplus e_2') & = & (e_1 \bowtie e_2) \oplus (e_1 \bowtie e_2' \parallel e_1' \bowtie e_2 \parallel e_1' \bowtie e_2') \\
(e_1 \oplus e_1') \vee (e_2 \oplus e_2') & = & (e_1 \bowtie e_2 \parallel e_1 \bowtie e_2' \parallel e_1' \bowtie e_2) \oplus (e_1' \bowtie e_2')
\end{array}$$

A polyset over τ is represented by the type $\tau \oplus \tau$, with all the above definitions generalising directly. That is, a value of this type is an aggregation of evidence (with multiplicity) either for or against each base value of τ. In this way, we can interpret a finite set $\{d_1, \ldots, d_k\}$ by the expression $(d_1 \oplus {d_1}^{\perp}) \vee \cdots \vee (d_k \oplus {d_k}^{\perp})$. Note that in this calculus $\vee$ and $\wedge$ form a lattice-esque structure as opposed to $\parallel$ and $\bowtie$, which form a ring structure.

Chapter 4

Query Processing

4.1 Motivation

Working with data usually involves making *queries*. A query can be thought of as a particular view of a database. For instance, given of database of geographic data we might want to get the complete list of countries together with number of cities for each country. Typically, the countries and the cities will be stored in different *tables* and the query has to suitably *join* them together.

The traditional tools for this job are relational algebra and SQL. On the theoretical side relational algebra provides a theory of sets of tuples with operations like selection, projection and join. On the practical side SQL is a language for working with real database systems inspired by, but deviating from, relational algebra.

Linear algebra provides an alternative to relational algebra as a query language foundation. The most important difference is that using linear algebra gives rise to the idea of generalised multisets. In an ordinary set elements occur either 0 or 1 times. In a multiset elements may occur any number of times, i.e. element multiplicities come from $\mathbb{N}$. In a generalised multiset over a ring R mulplicities can be any value from R. One consequence is that multiplicities can now be negative, which can be used to model deletion.

A broader, but no less important, consequence is that this places us firmly in the domain of *algebra*[1] with all the associated tools and techniques from over a century of mathematical research. The relevant structures—rings, modules, algebras—are well known objects of study. It turns out that all of the basic relational algebra operations have natural algebraic analogues.

[1] Algebra in a broad sense is arguably over two millennia old, but we are referring to modern abstract algebra whose development began in the 19th century.

4.2 Overview

Query languages, like any other programming language, should be

- *efficient*, in that they should produce query results as efficiently as possible in terms of time, space and energy consumed;
- *expressive*, in that they should admit expressing solutions to many problems; and
- *reasonable*, in that they should provide reasoning principles about a query's semantics to support compositional checking of functional and security properties, correctness of transformations and optimisations, query synthesis and more.

These aspects are sometimes presented as irreconcilable, suggesting, for example, that expressivity and reasonableness necessarily come at the cost of efficiency; or they may be studied in isolation of each other (expressiveness and correctness "modulo" efficiency, or efficient query engine implementation with specialised functionality "modulo" concerns what this does to the validity of reasoning principles in the core language). We propose that efficiency, expressiveness and reasoning can mutually support each other via *structure (algebraic and categorical)*.

Intuitively, our approach consists of identifying purely algebraic operations that can be implemented by symbolic simplification that exploits their algebraic properties for run-time efficiency, and factoring expensive exception and equality checking into separate operations that are invoked explicitly (e.g. checking whether a set is empty before deleting from it or whether two elements are equal or not) *only when* this is required, rather than melding the two operations together into a single operation. To give meaning to the result of an exceptionless deletion operation we introduce a data model that encompasses and goes beyond sets and multisets: *polysets*, whose elements carry any integral multiplicity, not only 1 (sets) or positive integers (multisets). Indeed, multiplicities can be replaced by arbitrary rings and even algebras.

We recognise finite polysets as the elements of the free module generated by a (usually infinite) set X over $\mathbb{Z}$. More generally, we show that *modules* over a commutative ring permit a variety of constructions that capture the core operations in query processing, including aggregation. They are all characterised by *universal properties*, which not only justify singling them out as natural primitive operations, but also provide algebraic rewrite rules for run-time efficiency. Overall, our approach of building terms through free structures and then interpreting them using their universal properties is not unlike that taken by algebraic effects (see, e.g., [11]).

Our approach is based on using free modules to represent generalised relations, biproducts for records, copowers for finite maps, tensor products for relational Cartesian products, and compact maps for infinite collections with wildcards. All of these constructions use arbitrary sets and modules to form new modules. We show that operations such as selection, projection, union, and intersection (and more) are not only very natural to express and reason about, but also lead us to highly efficient implementations of operations such as relational Cartesian products and, most significantly, arbitrary natural and outer joins. In the case of joins, the asymptotic efficiency attained is not even achievable using conventional relational query optimisation technology.

Although universal properties characterise these constructions uniquely *extensionally*, some *data structure representations* are better than others. A particular strength of our approach is how it uses prolific symbolic operator representations at run time to avoid unnecessary costly normalisation to a standard data structure, unless a particular context makes it absolutely necessary. For example, representing a tensor product symbolically gives a quadratic space compression and speed-up compared to normalising it to a list of atomic pairs; representing scalar multiplication symbolically eliminates iterated adding; representing additions symbolically facilitates operating on the summands independently, without first turning the tree of additions into a list of leaves. More generally, this also applies to scalar multiplication so that the implementation of a linear map boils down to *folding*, the composition of mapping the leaf elements to the target module and interpreting symbolic scalar multiplication and addition in the target module (which may also use symbolic scalar multiplication and addition).

Efficient evaluation hinges on the algebraic operations and their properties in a crucial way. Unnormalised representations using symbolic constructors let our evaluator invoke specialised evaluation rules depending on the particular way data are used; for example, computing the weight # of a polyset (corresponding to the cardinality of a multiset) contains the clauses $\#(s_1 \otimes s_2) = \#s_1 \cdot \#s_2$ and $\#(t_1 + t_2) = \#t_1 + \#t_2$. In particular, the tensor product $\otimes$ need not first be multiplied out to a quadratically bigger set of pairs of elements from s_1 and s_2. Since the output of a join consists of a data dependent number of tensor products, this is not straightforwardly achievable by static preprocessing of a query.

Selective simplification is explicitly indicated and forced by efficient implementations of explicit natural isomorphisms to modules that correspond to data structures for efficient associative access. This is where tensor products, copowers/finite maps and in particular compact maps, an algebraic generalisation of maps with a nonzero default value, come into play.

Note that this "very lazy" evaluation strategy differs from ordinary lazy evaluation. The former avoids normalisation entirely when opportunity arises while the latter simply postpones normalisation.

4.3 Linear Algebra as a Query Language

We propose the theory of modules and the linear maps between them as an appealing generalised framework for expressing queries. So far we have seen that linear algebra can express a more general class of sets than the usual kinds of sets and multisets employed in query languages; in particular, it is possible to have sets with negative multiplicities, cofinite sets and more. We now present a "Rosetta Stone" showing how the operations of relational algebra have corresponding linear maps on modules.

Selection In relational algebra *selection* restricts a set of tuples to the subset satisfying a given predicate. Suppose $P \subseteq A$. We define $\sigma_P : \mathbf{F}_K[A] \to \mathbf{F}_K[A]$ by

$$\sigma_P(\{a\}) = \{a\} \quad (\text{when } a \in P) \qquad \sigma_P(\{a\}) = 0 \quad (\text{when } a \notin P)$$

The effect is that a generator $\{a\}$ is preserved when $a \in P$ and eliminated otherwise.

Projection In relational algebra *projection* selects a subset of attributes, throwing away those not in the designated subset. Note that due to set semantics this may cause values to collapse, making the resulting set smaller. For example projecting the A component of $\{(A : \texttt{foo}, B : 1), (A : \texttt{foo}, B : 2), (A : \texttt{bar}, B : 3)\}$ yields $\{(A : \texttt{foo}), (A : \texttt{bar})\}$. For a tensor product $\mathbf{F}_K[A] \otimes \mathbf{F}_K[B]$ we define the two projections as follows.

$$\begin{aligned} \pi_1 &: \mathbf{F}_K[A] \otimes \mathbf{F}_K[B] \to \mathbf{F}_K[A] & \pi_1(x \otimes y) &= \#y \cdot x \\ \pi_2 &: \mathbf{F}_K[A] \otimes \mathbf{F}_K[B] \to \mathbf{F}_K[B] & \pi_2(x \otimes y) &= \#x \cdot y \end{aligned}$$

More complicated projections can be constructed using these two projections, the identity map, and the functorial action of $\otimes$. Note that, unlike relational algebra, all such projections preserve multiplicities.

Renaming. In relational algebra *renaming* changes the names of attributes. This makes sure everything is only done "up to choice of names", but it is also sometimes necessary in order to apply a natural join.

We do not use named attributes, but a similar concern arises with regards to the order and parenthesisation of attributes. To this end we have two natural isomorphisms, the *associator* and the *commutator*:

$$\alpha : U \otimes (V \otimes W) \cong (U \otimes V) \otimes W \qquad \beta : U \otimes V \cong V \otimes U$$

Combining these isomorphisms with the functorial action of $\otimes$, we can rearrange arbitrarily as needed.

Union and intersection. Union in relational algebra is just the usual union of sets. We define union as simply $+$. In general this form of union keeps track of multiplicities so for instance $(\{a\} + \{b\}) + (\{b\} + \{c\}) = \{a\} + 2\{b\} + \{c\}$.

In relational algebra *intersection* is typically not mentioned explicitly, but it arises as a special case of join when the two relations have the same set of attributes. In our approach intersection is the primitive upon which join is built. It is simply the product operation from the algebra structure. If $x, y : \mathbf{F}_K[A]$ then $xy : \mathbf{F}_K[A]$ is their intersection. Recall that the algebra product is a bilinear operator so multiplicities are multiplied. For instance $(2\{a\} + 3\{b\})(5\{b\} + 7\{c\}) = (3 \cdot 5)\{b\} = 15\{b\}$.

Cartesian product. Cartesian product in relational algebra is also the usual notion from set theory, and is traditionally viewed as an expensive operation that is best avoided if possible. Our version of the Cartesian product is the tensor product. We view it not as an operation, but as a symbolic term. In particular, we are perfectly comfortable writing down terms like

$$t = (\{a_1\} + \cdots + \{a_m\}) \otimes (\{b_1\} + \cdots + \{b_n\})$$

without insisting that this be expanded to a canonical form as soon as possible. Such an expansion generally incurs a quadratic blow-up in expression size. Depending on what happens to t later we might never have to expand it. For instance, $\pi_2(t)$ can be computed as

$$\pi_2(t) = \#(\{a_1\} + \cdots + \{a_m\}) \cdot (\{b_1\} + \cdots + \{b_n\}) = m \cdot (\{b_1\} + \cdots + \{b_n\})$$

The amount of work done was linear in the size of the symbolic term, and sublinear in the size of the hypothetically expanded form. Also note that even this result is not yet in canonical form—there is no need to distribute the scalar multiplication prematurely.

Static analysis will typically recognise opportunities like a projection immediately applied to a Cartesian product and do appropriate optimisation. By contrast, our approach does it at run time. It does not rely on a sufficiently clever analysis, it guarantees that products are not expanded until necessary and it even allows terms to be stored more efficiently in data structures between operations.

Natural join In relational algebra the *natural join* of two relations is constructed by taking their Cartesian product and keeping only the tuples where both sides agree about the values of shared attributes.

For example, consider the relations

$$\begin{aligned} x = \{&(A : a, B : 1), & y = \{&(B : 2, C : p) \\ &(A : b, B : 2), & &(B : 3, C : q), \\ &(A : c, B : 3) & &(B : 4, C : r)\} \end{aligned}$$

where the notation $(A : a, B : 1)$ denotes a tuple with a value of a for attribute A and 1 for attribute B. Their join is

$$x \bowtie y = \{(A : b, B : 2, C : p), (A : c, B : 3, C : q)\}$$

In our system these two relations would be represented as the vectors x and y given by

$$\begin{aligned} &x : \mathbf{F}_K[\mathbf{Str}] \otimes \mathbf{F}_K[\mathbb{Z}] \qquad && x = \{a\} \otimes \{1\} + \{b\} \otimes \{2\} + \{c\} \otimes \{3\} \\ &y : \mathbf{F}_K[\mathbb{Z}] \otimes \mathbf{F}_K[\mathbf{Str}] \qquad && y = \{2\} \otimes \{p\} + \{3\} \otimes \{q\} + \{4\} \otimes \{r\}\ . \end{aligned}$$

To compute their join we first have to inject them into a common module. This is done by going from $\mathbf{F}_K[\cdot]$ to $\mathbf{F}^*_K[\cdot]$ and adding 1's as necessary.

$$\begin{array}{ll} x' : \mathbf{F}^*_K[\mathbf{Str}] \otimes \mathbf{F}^*_K[\mathbb{Z}] \otimes \mathbf{F}^*_K[\mathbf{Str}] & x' = \{a\} \otimes \{1\} \otimes 1 + \{b\} \otimes \{2\} \otimes 1 + \{c\} \otimes \{3\} \otimes 1 \\ y' : \mathbf{F}^*_K[\mathbf{Str}] \otimes \mathbf{F}^*_K[\mathbb{Z}] \otimes \mathbf{F}^*_K[\mathbf{Str}] & y' = 1 \otimes \{2\} \otimes \{p\} + 1 \otimes \{3\} \otimes \{q\} + 1 \otimes \{4\} \otimes \{r\} \end{array}$$

The join of two elements of the same module is simply their intersection, which is given by multiplication.

$$x' \cdot y' = (\{a\}\otimes\{1\}\otimes 1+\{b\}\otimes\{2\}\otimes 1+\{c\}\otimes\{3\}\otimes 1)\cdot(1\otimes\{2\}\otimes\{p\}+1\otimes\{3\}\otimes\{q\}+1\otimes\{4\}\otimes\{r\})$$

A naive approach to simplification is to apply distributivity bluntly and then simplify using identities for tensor products.

$$\begin{array}{ll} & (\{a\} \otimes \{1\} \otimes 1) \cdot (1 \otimes \{2\} \otimes \{p\}) + (\{a\} \otimes \{1\} \otimes 1) \cdot (1 \otimes \{3\} \otimes \{q\}) + (\{a\} \otimes \{1\} \otimes 1) \cdot (1 \otimes \{4\} \otimes \{r\}) \\ + & (\{b\} \otimes \{2\} \otimes 1) \cdot (1 \otimes \{2\} \otimes \{p\}) + (\{b\} \otimes \{2\} \otimes 1) \cdot (1 \otimes \{3\} \otimes \{q\}) + (\{b\} \otimes \{2\} \otimes 1) \cdot (1 \otimes \{4\} \otimes \{r\}) \\ + & (\{c\} \otimes \{3\} \otimes 1) \cdot (1 \otimes \{2\} \otimes \{p\}) + (\{c\} \otimes \{3\} \otimes 1) \cdot (1 \otimes \{3\} \otimes \{q\}) + (\{c\} \otimes \{3\} \otimes 1) \cdot (1 \otimes \{4\} \otimes \{r\}) \end{array}$$

Next we exploit that $\cdot$ works component-wise on tensor products.

$$\begin{array}{ll} & (\{a\} \cdot 1) \otimes (\{1\} \cdot \{2\}) \otimes (1 \cdot \{p\}) + (\{a\} \cdot 1) \otimes (\{1\} \cdot \{3\}) \otimes (1 \cdot \{q\}) + (\{a\} \cdot 1) \otimes (\{1\} \cdot \{4\}) \otimes (1 \cdot \{r\}) \\ + & (\{b\} \cdot 1) \otimes (\{2\} \cdot \{2\}) \otimes (1 \cdot \{p\}) + (\{b\} \cdot 1) \otimes (\{2\} \cdot \{3\}) \otimes (1 \cdot \{q\}) + (\{b\} \cdot 1) \otimes (\{2\} \cdot \{4\}) \otimes (1 \cdot \{r\}) \\ + & (\{c\} \cdot 1) \otimes (\{3\} \cdot \{2\}) \otimes (1 \cdot \{p\}) + (\{c\} \cdot 1) \otimes (\{3\} \cdot \{3\}) \otimes (1 \cdot \{q\}) + (\{c\} \cdot 1) \otimes (\{3\} \cdot \{4\}) \otimes (1 \cdot \{r\}) \end{array}$$

The multiplications now all involve only generators and 1, so they can all be simplified.

$$\begin{array}{ll} & \{a\} \otimes 0 \otimes \{p\} + \{a\} \otimes 0 \otimes \{q\} + \{a\} \otimes 0 \otimes \{r\} \\ + & \{b\} \otimes \{2\} \otimes \{p\} + \{b\} \otimes 0 \otimes \{q\} + \{b\} \otimes 0 \otimes \{r\} \\ + & \{c\} \otimes 0 \otimes \{p\} + \{c\} \otimes \{3\} \otimes \{q\} + \{c\} \otimes 0 \otimes \{r\} \end{array}$$

All tensor products with a 0 component can be eliminated due to linearity. In the end we get the simplified form:

$$\{b\} \otimes \{2\} \otimes \{p\} + \{c\} \otimes \{3\} \otimes \{q\}$$

This method of simplification is wasteful as we are expanding everything using distributivity only to have most components turn out to be 0. We shall later see that there is a much more efficient approach to simplifying expressions that in particular can solve basic joins like this one in linear time.

Outer join In relational algebra the *outer join* is similar to the natural join. The difference is that tuples from either input which would not occur anywhere in the output are included anyway. The missing attributes are populated with a **null** value.

Consider the previous example of a natural join. The result of the outer join would be as follows.

$$\{(A : a, B : 1, C : \mathbf{null}), (A : b, B : 2, C : p), (A : c, B : 3, C : q), (A : \mathbf{null}, B : 4, C : r)\}$$

There are also *left outer join* and *right outer join* operations, which only include extra tuples from the left and right input respectively.

In our system we use wildcards to achieve a similar effect. Note that though the wildcard bears superficial similarity to **null**, it is in fact the exact opposite: **null** is a special value that agrees with *no value* (including itself), the wildcard is a special value that agrees with *every value*. It could be argued that the main reason **null** is used to fill missing values in relational algebra is that it is the only somewhat applicable tool in the relational toolbox.

Recall that to compute the natural join of x and y we first embed them by adding wildcards to get x' and y'. The left outer join is then given by $x' \cdot (y'+1)$, the right outer join by $(x'+1) \cdot y'$, and the outer join by $(x'+1) \cdot (y'+1)$. By distributivity this is the same as $x' \cdot y' + x'$, $x' \cdot y' + y'$ and $x' \cdot y' + x' + y' + 1$ respectively. One way to think about an expression like $x' \cdot (y'+1)$ is that the added 1 ensures that every component of x' matches with *something*.

For the outer join one might want to use $(x'+1) \cdot (y'+1) - 1$ instead to avoid the last factor of 1. However, our proposed definition generalises more neatly to n-ary outer joins, which can be written simply as $(x_1+1) \cdot (x_n+1)$.

Aggregation In (extensions of) relational algebra *aggregation* computes values like sum or maximum over an attribute. For instance, given the relation $\{(A : p, B : 2), (A : p, B : 3), (A : q, B : 4)\}$ aggregating B by sum would yield $\{(A : p, B : 5), (A : q, B : 4)\}$, while aggregating B by maximum would yield $\{(A : p, B : 3), (A : q, B : 4)\}$.

We take a different view of aggregation, as something that happens implicitly due to module semantics. This was evident when discussing projection, where the discarded attributes are automatically counted with multiplicities. The specifics of aggregation depends on the nature of the ring K.

Take the example above and represent it as follows: $2\{p\} + 3\{p\} + 4\{q\} = (2+3)\{p\}+4\{q\}$. If K is $\mathbb{Z}$ with ordinary arithmetic this reduces to $5\{p\}+4\{q\}$. Note how a tuple such as $(A : p, B : 2)$ was represented as $2\{p\}$ and not as $\{p\} \otimes \{2\}$. We *moved* the attribute into the ring.

Not all aggregations can be expressed directly as a ring structure, though. For instance, integers extended with infinity and equipped with minimum and maximum operations only form a *semiring* since negation is impossible. Our approach can easily work with semirings as well, but as we shall see shortly negation plays an important role and should not be given up so easily.

Instead, we express the aggregation using nested free modules to group data. Each grouping is an element of $\mathbf{F}_{\mathbb{F}_2}[A]$, i.e. an ordinary finite set. The example above would be represented as follows: $\{p\} \otimes \{\{2\} + \{3\}\} + \{q\} \otimes \{\{4\}\}$. Non-linear aggregations like minimum are modelled as functions (*not* linear maps) $\min : \mathbf{F}_{\mathbb{F}_2}[\mathbb{Z}] \to \mathbb{Z}_\infty$ with $\min(\{a_1\} + \cdots + \{a_n\}) = \min\{a_1, \ldots, a_n\}$. Here $\mathbb{Z}_\infty$ is $\mathbb{Z}$ with ∞ adjoined so that $\min\{\}$ is well-defined. Aggregating by minimum on the second attribute we get:

$$(\mathrm{id}_{\mathbf{F}_K[\mathbf{Str}]} \otimes \mathbf{F}_K[\min])(\{p\} \otimes \{\{2\} + \{3\}\} + \{q\} \otimes \{\{4\}\}) = \{p\} \otimes \{2\} + \{q\} \otimes \{4\}$$

In general, nested free modules allow any non-linear function to be included in an otherwise linear query. Besides aggregations, this includes operations like turning negative multiplicities into zeroes.

These considerations demonstrate that linear maps are exactly the maps that can be easily adapted to a distributed setting. The non-linear maps, on the other hand, depend on having the entire dataset at hand and subjected to at least some degree of simplification, which implies that synchronisation is necessary.

Domain computations. Relational algebra proper does not provide a way to transform data, but in practical realisations this is usually possible. For instance, given the relation $\{(A \mapsto \text{foo}), (A \mapsto \text{bar})\}$ we might transform it using a function **upper** that maps strings to uppercase, getting the result $\{(A \mapsto \text{FOO}), (A \mapsto \text{BAR})\}$. In our system this is achieved by the functorial action of $\mathbf{F}_K[\cdot]$. In particular given $\textbf{upper} : \textbf{Str} \to \textbf{Str}$ we have $\mathbf{F}_K[\textbf{upper}] : \mathbf{F}_K[\textbf{Str}] \to \mathbf{F}_K[\textbf{Str}]$. We can therefore apply it to an element as follows: $\mathbf{F}_K[\textbf{upper}](\{\text{foo}\} + \{\text{bar}\}) = \{\textbf{upper}(\text{foo})\} + \{\textbf{upper}(\text{bar})\} = \{\text{FOO}\} + \{\text{BAR}\}$.

Insert and delete. For persistent relations there is the question of how to do updates. Relational algebra is based on set theory, so updates can be done using set union and difference.

In our system all updates are done by addition. Deleting an element amounts to adding its negative. For example, say the database contains $\{a\} + \{b\}$ and we want to add c and delete b. This update is computed as $(\{a\}+\{b\})+(\{c\}-\{b\}) = \{a\} + \{c\}$.

Note that there is no conceptual distinction between *databases* and *database updates*, since deletions are simply represented as negative elements. Furthermore, since $+$ is commutative the order of updates does not matter. For instance, suppose the database only contains $\{a\}$ and we run the update above: $\{a\} + (\{c\} - \{b\}) = \{a\} - \{b\} + \{c\}$. This leaves a database with a negative occurrence of b. If we then insert b afterwards we end up with $\{a\} + \{c\}$ again. In a sense this is the easiest possible conflict resolution strategy: just accept the data from every update and add it all together! Of course, there are times when we might want to ensure that the database does not contain negative multiplicities by, say, setting them all to 0 using an explicit operation. In that case—and that case only—we separate updates into before and after that given point.

Chapter 5

Algebraic Evaluation

5.1 Evaluation by Simplification

We now turn to the question of evaluation. More precisely, the question of *simplifying* module terms. What kind of simplified form do we have in mind; is simplification even necessary? That depends on the context of use, but generally a simplified form involves some of the following:

- No zeroes in additions or multiplications, and no multiplications except $r \cdot \{a\}$ where $r : K$.
- No repeated generators, e.g. $\{a\} + \{b\} + \{a\}$ should be $2\{a\} + \{b\}$.
- No sums of biproducts, e.g. $(u_1, v_1) + (u_2, v_2)$ should be $(u_1 + u_2, v_1 + v_2)$.
- No tensor products where either factor is a sum.

The kind of observation we wish to make dictates which requirements are necessary. For example, the most elementary observation we can make is to ask if a term is equal to 0. In query terms this corresponds to the question of satisfiability. We certainly need to avoid repeated generators as otherwise they might happen to cancel out, e.g. $\{a\} - 2\{a\} + \{a\}$ is a non-trivial representation of 0. On the other hand, if the term is a tensor product we do not need to expand it, since $u \otimes v$ is 0 precisely when either factor is 0.

In order to keep the discussion manageable we will focus on the observation of listing the output, analogous to retrieving a list of rows in the result like in traditional query evaluation. This observation demands that we satisfy all of the constraints listed above. Keep in mind that this only concerns the *final* representation meant for user inspection. When feeding the output of one query as input to the next one there is no reason to obey all of these constraints.

In discussing term simplification it is important to distinguish between *intensional* and *extensional* equality. Two terms are intensionally equal when they are written the same way (modulo renaming). They are extensionally equal when they can be shown to be equal using the presented algebraic identities. For example, $0 + 0 + 0 + 0$ and 0 are extensionally equal which can be argued using the identity $x + 0 = x$ repeatedly. However, they are not intensionally equal and indeed we would consider the latter a more compact version of the former.

5.2 Data Structures

So far we have been following an abstract approach, preferring to leave out detailed descriptions of structures in favour of universal properties. This has the advantage of separating specifications from algorithms and data structures, but postpones the question of whether suitable algorithms and data structures even *exist*. We will proceed to argue that not only is the answer *yes*, but that in fact very efficient methods naturally arise.

5.2.1 Free Structures

In a categorical framework the existence of data structures is often simple; abstract universal constructions generally provide us with at least one example, namely the *free* structure. Free structures simply *generate* all the necessary operations and impose only the equalities required by axioms.

For example, take the tensor product $M \otimes N$ of R-modules M and N. It can be characterised quite succinctly: $(\otimes) : M \times N \to M \otimes N$ is a bilinear operator. That is really all one needs to know about the tensor product. A free structure for the tensor product is then a term algebra consisting of:

- $0 : M \otimes N$
- $(+) : M \otimes N \times M \otimes N \to M \otimes N$
- $(\cdot) : R \times M \otimes N \to M \otimes N$
- $(\otimes) : M \times N \to M \otimes N$

The first three constructors provide the general module structure. The last constructor is simply a copy of the required operator for a tensor product.

To obtain the actual set of elements in $M \otimes N$ we need to impose equalities, namely that $M \otimes N$ is a module ($+$ is commutative, etc.) and that $(\otimes)$ is bilinear (distributive, etc.). The constructed object is thus *by definition* a tensor product.

Another perspective on free constructions is that they are simply *syntax trees*. This highlight another advantage: syntax can be manipulated and if every transformation is justified by an algebraic identity then the resulting term is semantically equivalent. For the tensor product it means that terms like

$$a_1 \otimes b_1 + \ldots + a_1 \otimes b_n + a_2 \otimes b_1 + \ldots + a_2 \otimes b_n + \ldots + a_m \otimes b_1 + \ldots + a_m \otimes b_n$$

with a total of $O(mn)$ components can be rewritten to

$$(a_1 + \ldots a_m) \otimes (b_1 + \ldots + b_n)$$

with a total of $O(m{+}n)$ components, a quadratic improvement in space complexity. In practice these kind of simplifications are hard to discover dynamically. The most useful consequence goes in the other direction, namely that if we have a term written in the latter form we should not expand it to the former unless absolutely necessary.

At the same time the free structure retains all the necessary information to *interpret* it into any other representation: simply walk the syntax tree and evaluate each constructor according to the target representation. For instance, in finite-dimensional linear algebra it is possible to take the flat representation $\mathbb{R}^m \otimes \mathbb{R}^n = \mathbb{R}^{mn}$, where the $(\otimes)$ operator is defined by suitable index juggling. Interpretation then consists of simply performing this index juggling at every occurence of $(\otimes)$, then combining the results according to the uses of 0, $(+)$, $(\cdot)$.

The flat representation has the advantage that it can be stored as a simple array, which is useful for e.g. parallel computing. The disadvantage is that terms are always in what corresponds to the expanded form above using $O(mn)$ space. As noted above going back to the compact form from the expanded form is hard. The rôle of the free structure then emerges: to retain as much information as possible while still being able to be interpreted into other representations later when advantageous.

5.2.2 Finite Maps

Before dealing with simplification of arbitrary terms we also need to consider finite maps. They have a great deal of structure to exploit and also play a vital rôle in making simplification efficient.

Suppose we have some term $x : A \Rightarrow U$ and *assume that we already have a method for simplifying terms from U*. To simplify x we proceed depending on the nature of A. Recall that we have the following isomorphisms[1] at our disposal:

$$\begin{array}{ll} \mathbf{cp}_0 : 0 \Rightarrow U \cong \mathbf{0} & \mathbf{cp}_+ : (A + B) \Rightarrow U \cong (A \Rightarrow U) \oplus (B \Rightarrow U) \\ \mathbf{cp}_1 : 1 \Rightarrow U \cong U & \mathbf{cp}_\times : (A \times B) \Rightarrow U \cong A \Rightarrow B \Rightarrow U \end{array}$$

The idea is that we wrap the term in the appropriate isomorphism. To start, suppose $A = 1$ so $x : 1 \Rightarrow U$. We can then write x as $\mathbf{cp}_1^{-1}(\mathbf{cp}_1(x))$. The term $\mathbf{cp}_1(x)$ is then simplified to get some term $y : U$ and the simplified form of x becomes $\mathbf{cp}_1^{-1}(y)$.

Now suppose $A = A_1{+}A_2$ so $x : (A_1{+}A_2) \Rightarrow U$. We write x as $\mathbf{cp}_+^{-1}(\mathbf{cp}_+(x))$. The term $\mathbf{cp}_+(x)$ reduces to some term $(y_1, y_2) : (A_1 \Rightarrow U) \oplus (A_2 \Rightarrow U)$ (a biproduct can always be reduced to a pair by simple component-wise addition). At this point y_1 and y_2 are finite maps with index sets A_1 and A_2 respectively,

[1]Note the similarity with the usual rules for exponents, albeit with multiple types in play.

so we apply the procedure recursively to get simplified forms z_1 and z_2. The simplified form of x then becomes $\mathbf{cp}_+^{-1}(z_1, z_2)$.

Finally, suppose $A = A_1 \times A_2$ so $x : (A_1 \times A_2) \Rightarrow U$. In this case $\mathbf{cp}_\times(x)$ has type $A_1 \Rightarrow A_2 \Rightarrow U$. Recursively we have a procedure to simplify terms from $A_2 \Rightarrow U$; and given this we also get a procedure to simplify $\mathbf{cp}_\times(x)$ to get some term y. The simplified form of x becomes $\mathbf{cp}_\times^{-1}(y)$.

What about inductive sets? Suppose A is defined as the smallest solution to $F(X) \cong X$ for some functor $F : \textsc{Set} \to \textsc{Set}$. A is then obtained as the colimit of the diagram

$$\emptyset \to F(\emptyset) \to F(F(\emptyset)) \to F(F(F(\emptyset))) \to \cdots$$

where each map is the iterated functorial action of F on the unique map out of the empty set. It can be shown that $\Rightarrow$ as a functor $\textsc{Set} \to (\textsc{Mod}_K \to \textsc{Mod}_K)$ (from sets to endofunctors over modules) is left adjoint to $G : (\textsc{Mod}_K \to \textsc{Mod}_K) \to \textsc{Set}$ given by $G(E) = |E(K)|$. Since left adjoints preserve colimits in general, $\Rightarrow$ preserves colimits in its first argument and we obtain $A \Rightarrow U$ as a colimit of the diagram

$$(\emptyset \Rightarrow U) \to (F(\emptyset) \Rightarrow U) \to (F(F(\emptyset)) \Rightarrow U) \to (F(F(F(\emptyset))) \Rightarrow U) \to \cdots$$

This does not directly define $A \Rightarrow U$ as an inductive set, but it does establish that its elements are finite since any map into $A \Rightarrow U$ must be a map into $F^n(\emptyset) \Rightarrow U$ for some n.

What about sets that are not algebraic data types, i.e. sets defined using a combination of sums, products and induction? Function spaces such as $A = A_1 \to A_2$ are not amenable to simplification in general and we exclude them from consideration. Finally, there are primitives like n-bit integers. These can be handled using specific methods. In the case of finite precision integers an efficient solution is to represent finite maps as Patricia tries. For strings a good choice would be a radix trie.

For primitive types we will simply write the simplified form as $y = \sum_i (a_i \mapsto u_i)$ with the understanding that each $a : A$ occurs at most once, and computing $y(a)$ can be done using work linear in the size of a (which for finite precision integers would be constant). To see all this in action consider the term

$$\begin{array}{lcl} x & : & (\mathbf{Str} \times (\mathbf{Str} + \mathbb{N})) \Rightarrow K \\ x & = & ((a, \mathbf{inl}(p)) \mapsto 1) + ((b, \mathbf{inr}(4)) \mapsto 1) + ((a, \mathbf{inr}(3)) \mapsto 1) + ((a, \mathbf{inl}(p)) \mapsto 1) \end{array}$$

The simplification proceeds as follows:

$$\begin{aligned} x &= ((a, \mathbf{inl}(p)) \mapsto 1) + ((b, \mathbf{inr}(4)) \mapsto 1) + ((a, \mathbf{inr}(3)) \mapsto 1) + ((a, \mathbf{inl}(p)) \mapsto 1) \\ &\qquad \text{apply } \mathbf{cp}_\times \\ &= \mathbf{cp}_\times^{-1}((a \mapsto \mathbf{inl}(p) \mapsto 1) + (b \mapsto \mathbf{inr}(4) \mapsto 1) + (a \mapsto \mathbf{inr}(3) \mapsto 1) + (a \mapsto \mathbf{inl}(p) \mapsto 1)) \\ &\qquad \text{simplify outer finite map using method for strings} \\ &= \mathbf{cp}_\times^{-1}((a \mapsto (\mathbf{inl}(p) \mapsto 1) + (\mathbf{inr}(3) \mapsto 1) + (\mathbf{inl}(p) \mapsto 1)) + (b \mapsto \mathbf{inr}(4) \mapsto 1)) \end{aligned}$$

apply $\mathbf{cp}_+$

$$= \mathbf{cp}_\times^{-1}((a \mapsto \mathbf{cp}_+^{-1}((p \mapsto 1, 0) + (0, 3 \mapsto 1) + (p \mapsto 1, 0))) + (b \mapsto \mathbf{cp}_+^{-1}(0, 4 \mapsto 1)))$$

simplify biproduct by adding pairwise

$$= \mathbf{cp}_\times^{-1}((a \mapsto \mathbf{cp}_+^{-1}((p \mapsto 1) + (p \mapsto 1), 3 \mapsto 1)) + (b \mapsto \mathbf{cp}_+^{-1}(0, 4 \mapsto 1)))$$

simplify inner finite map using methods for strings and integers

$$= \mathbf{cp}_\times^{-1}((a \mapsto \mathbf{cp}_+^{-1}(p \mapsto 2, 3 \mapsto 1)) + (b \mapsto \mathbf{cp}_+^{-1}(0, 4 \mapsto 1)))$$

Using this method the simplified form of a finite map can be found in linear time. Compact maps can be dealt with just as easily by exploiting that $A \Rightarrow^* U \cong (A \Rightarrow U) \oplus U$. These isomorphisms are the basis of generic tries [12, 13].

5.3 Problem Statement

When studying database queries it is necessary to make some assumptions about the context. Examples of possible assumptions include:

- The database has had the opportunity to organise data beforehand (constructing indices, etc.).
- Only a small part of the data needs to be examined to answer the query.
- Only a small part of the output will be needed.

Our investigation primarily concerns *bulk data processing*, transformations from the entire input database to an entire output. In particular, queries are generally expected to take time at least linear in the sum of the input and the output. Since index structures (represented as tries) can be built in linear time this means that asymptotically it does not matter how the database is organised beforehand. A flat list of records suffices, as we simply construct appropriate indices when needed. We argue that this scope is reasonable, since our main contribution is efficient execution of the hardest queries whose complexity is generally much worse than linear anyway.

The problem of efficient execution of relational database queries has been extensively studied. Joins, in particular, pose a challenge. Informally, a k-ary join $\bowtie_{i=1}^{k} R_i$ of relations $R_1, \ldots, R_n$ is a relation R definable by

$$R = \{(x_1, \ldots, x_n) \mid (x_{11}, \ldots, x_{1n_1}) \in R_1 \wedge \ldots \wedge (x_{k1}, \ldots, x_{kn_k}) \in R_k\}$$

where the $x_{i1}, \ldots, x_{in_i}$ are pairwise distinct variables and

$$\{x_1, \ldots, x_n\} = \bigcup_{i=1}^{k} \{x_{i1}, \ldots, x_{in_i}\}$$

In the rest of this chapter we show that the algebraic version of such queries can be executed efficiently (for a certain technical definition of efficient, which generalises the previous notion of worst-case output optimality).

5.3.1 Prior Art

Traditional methods rely on decomposing k-ary joins into binary joins. They carefully create a good *query plan* based on algebraic properties, size estimates and auxiliary data structures (indexes) to minimize the sizes of intermediate results, and execute the query plan using streams of records as data structures for inputs, intermediate results and outputs [14].

While it is possible to evaluate *acyclic* joins in time linear in the size of the query, the input and the output [15, 16] and there are methods that deal with "almost" acyclic joins [17], this is infeasible for cyclic queries since deciding whether the output is empty is NP-hard [18].

To quantify exactly the complexity of a specific join query, the notion of *worst-case optimal complexity* was developed [19, 20], focussing on the *data complexity* of the problem [21]. A join algorithm is *worst-case optimal* if it executes in time linear in N and O, where O is the maximal size of an output of the join query applied to input relations whose sizes sum to N. The size of the query itself is not taken into account.

Bounds for O in terms of N have been found using advanced graph and information theory [22, 23]. In particular, O can be bounded as a function of N by the *fractional edge cover number*, which is the solution of a linear program derived from the form of the join query. Furthermore, that bound is sharp. For example, the set of triangles $\{(x, y, z) \mid (x, y) \in R \wedge (y, z) \in R \wedge (z, x) \in R\}$ is at most of size $O(N^{\frac{3}{2}})$ where $N = |R|$, which is quite unobvious. Every query plan based on binary joins produces an intermediate relation of size $\Theta(N^2)$, for example $\{(x, y, z) \mid (x, y) \in R \wedge (y, z) \in R\}$, and will then, according to the fraction cover bound, *inherently* filter out all but $O(N^{\frac{3}{2}})$ tuples in the final join. Thus computing triangles using binary joins and representing intermediate relations by listing their elements is not worst-case optimal: It expends $\Theta(N^{\frac{1}{2}})$ time for each output triangle.

Recently, worst-case optimal join algorithms have been discovered, which either calculate estimated intermediate output sizes [24, 25] or employ tries on on ordered data [26]. These algorithms have been shown to be strictly beyond the scope of classical query plan optimization since no join query plan [22] or even join-project query plan [19, 20] can achieve worst-case optimal execution time for cyclic query graphs.

5.4 Tries

Recall that $A \Rightarrow^* U$ is like $A \Rightarrow U$ except with additional wildcard entries of the form $* \mapsto x$, which should be thought of as constant maps. This is reflected in the algebra structure on $A \Rightarrow^* U$, which treats wildcards as agreeing with any possible value. In particular, recall that $(a \mapsto x) \cdot (a \mapsto y) = (a \mapsto x) \cdot (* \mapsto y) = (* \mapsto x) \cdot (a \mapsto y) = a \mapsto x \cdot y$ and $(* \mapsto x) \cdot (* \mapsto y) = * \mapsto x \cdot y$.

An element of a weighted module of the form $A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ is called a *trie* over $A_1, \ldots, A_k$. This is our main data structure. It can comfortably represent both inputs and outputs of queries. A trie can be built in linear time from a list of tuples, and lookup is linear in the size of the key [12].

An element of a weighted module of the form $A_1 \to \cdots \to A_k \to K$ is called a *characteristic function* over $A_1, \ldots, A_k$. Intuitively, a characteristic function is a "curried" function that takes its inputs $a_1 \in A_1, \ldots, a_k \in A_k$ one at a time and eventually returns an element of K. We can interpret tries into characteristic functions via *deep lookup*:

$$\begin{aligned}
&(\cdot)^{\ddagger} : (A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K) \to (A_1 \to \cdots \to A_k \to K) \\
&R^{\ddagger}(a_1)\ldots(a_k) = R \diamond (a_1 \mapsto \cdots \mapsto a_k \mapsto 1)
\end{aligned}$$

Thus looking up an element in R is nothing more than comparing R with the corresponding singleton trie.

Characteristic functions can represent arbitrary relations. We shall also need a *shallow lookup* operation that only interprets the topmost copower into a power.

$$\begin{aligned}
&(\cdot)^{\dagger} : (A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K) \to (A_1 \to A_2^* \Rightarrow^* \cdots \Rightarrow^* A_k^* \Rightarrow^* K) \\
&(* \mapsto x)^{\dagger}(a) = x \\
&(a \mapsto x)^{\dagger}(b) = 1_{a=b} \cdot x
\end{aligned}$$

Deep lookup relates to shallow lookup as follows.

$$R^{\ddagger}(x) = R^{\dagger}(x)^{\ddagger}$$

Note that since lookup is linear a lookup with a produces the sum of the a-entry and the $*$-entry, e.g.:

$$((a \mapsto x) + (b \mapsto y) + (* \mapsto z))^{\ddagger}(a) = x + z$$

Here z is the baseline value and x is the deviation of the a-entry from that baseline, while y is ignored since $a \neq b$.

Example 1. *Consider the following table associating paradigms with programming languages.*

Paradigm	***Language***
Functional	*Haskell*
Functional	*ML*
Functional	*Agda*
Imperative	*C++*
Imperative	*Pascal*
OOP	*Java*
OOP	*C++*

If **Str** *is the type of strings, then this table corresponds to a relation* $R \subseteq$ **Str** $\times$ **Str**. *We will, however, represent it as a trie (abbreviating each entry with its first letter):*

$$\begin{aligned}\mathbf{PL} = {} & (F \mapsto (A \mapsto 1) + (H \mapsto 1) + (M \mapsto 1)) \\ & + (I \mapsto (C \mapsto 1) + (P \mapsto 1)) \\ & + (O \mapsto (C \mapsto 1) + (J \mapsto 1))\end{aligned}$$

Observe that $\mathbf{PL}^{\ddagger}(x, y) = 1$ *iff* $(x, y) \in R$. *As a tree structure we draw* **PL** *as shown in Figure 5.1.*

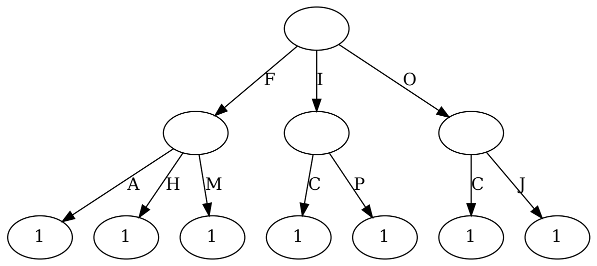

Figure 5.1: **PL** depicted as a tree

Example 2. *Consider the following table associating programmers with languages that they use.*

Name	***Language***
Xander	*Pascal*
Xander	*Java*
Yen	*
Zack	*C++*
Zack	*ML*

Note the use of the wildcard, meaning that Yen uses all conceivable languages. The corresponding trie as a tree structure is seen in Figure 5.2, using dotted lines for wildcards.

Example 3. *Consider the following table associating programmers with paradigms that they tout as superior.*

Name	***Paradigm***
Xander	*Functional*
Yen	*Functional*
Yen	*Imperative*
Zack	*Anything but OOP*

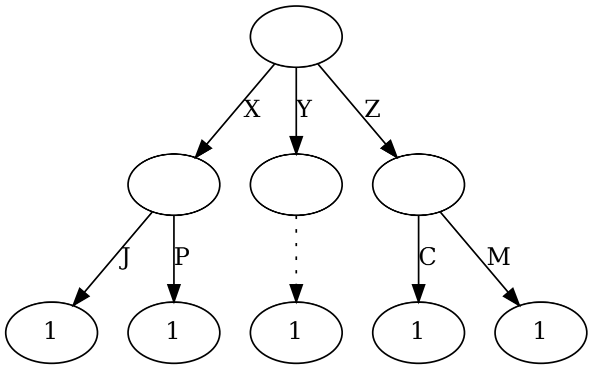

Figure 5.2: **NL** depicted as a tree

What do we mean by "anything but OOP"? We can express Zack's preferences as $\overline{\{O\}}$, the set consisting of all paradigms other than OOP. As a trie this is written using subtraction in $K = \mathbb{Z}$ to cancel out OOP:

$$\begin{aligned}\mathbf{NP} = &(X \mapsto (F \mapsto 1)) \\ &+ (Y \mapsto (F \mapsto 1) + (I \mapsto 1)) \\ &+ (Z \mapsto (* \mapsto 1) + (O \mapsto -1))\end{aligned}$$

See Figure 5.3 for the corresponding tree.

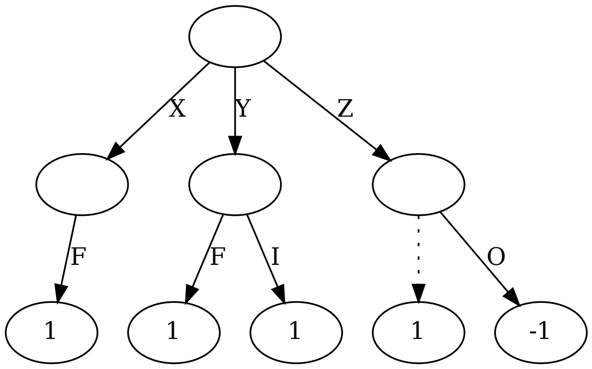

Figure 5.3: **NP** depicted as a tree

In this fashion, tries can represent any relation built from finite relations combined using finite unions, complements and Cartesian products.

5.5 Joins

Classically, the intersection $A \cap B$ of two sets, A and B, is characterized by the logical equivalence

$$x \in A \cap B \Leftrightarrow x \in A \wedge x \in B$$

Analogously, the *algebraic join* of two tries is a bilinear operator

$$\bowtie : (A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K) \times (A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K) \to (A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K)$$

such that

$$(x \bowtie y)^\ddagger = x^\ddagger \cdot y^\ddagger$$

for all $x, y : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$.

We take $x \bowtie y = x \cdot y$, the algebra product. Other choices may be possible for some K, but this definition works in general.

Note that algebraic joins are an extension of traditional relational joins. Say that a trie $R : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ is *standard* if each level of the trie structure of R either contains only $*$-mappings or no $*$-mappings. In other words R represents a relation over a subset of the attributes $A_1, \ldots, A_k$ and contains only finitely many tuples when restricted to this subset.

A join consisting only of standard tries is itself called *standard* and corresponds to the traditional notion of relational join. The output of a standard join is always a standard trie. Hence, algebraic joins contain traditional ones as a special case.

Example 4. *Let* **PL**, **NL** *and* **NP** *be the previously defined relations. We want to compute their join to produce triples* (n, p, l) *satisfying* $\mathbf{PL}(p, l)$, $\mathbf{NL}(n, l)$ *and* $\mathbf{NP}(n, p)$. *In order to line up the attributes correctly we need to insert wildcard layers.*

For any trie R *let* $w(R) = (* \mapsto R)$. *This is sufficient for adding wildcard layers at the first attribute. To insert a layer we use the fact that the copower construction is a functor, i.e. for any linear map* $f : U \to V$ *there is a linear map*

$$\hat{f} : (A \Rightarrow^* U) \to (A \Rightarrow^* V)$$
$$\hat{f}(a \mapsto x) = a \mapsto f(x)$$

We can now define

$$\mathbf{NP}' = \hat{\hat{w}}(\mathbf{NP}) \qquad \mathbf{NL}' = \hat{w}(\mathbf{NL}) \qquad \mathbf{PL}' = w(\mathbf{PL})$$

See Figure 5.4.

The join is then

$$\begin{aligned}\mathbf{NP}' \cdot \mathbf{NL}' \cdot \mathbf{PL}' = {} & (Y \mapsto F \mapsto (A \mapsto 1) + (H \mapsto 1) + (M \mapsto 1)) \\ & + (Y \mapsto I \mapsto (C \mapsto 1) + (P \mapsto 1)) \\ & + (Z \mapsto F \mapsto M \mapsto 1)\end{aligned}$$

See Figure 5.5 for the corresponding tree.

Note that even though two of the inputs were not standard, the result still turned out to be standard. This is not the case in general.

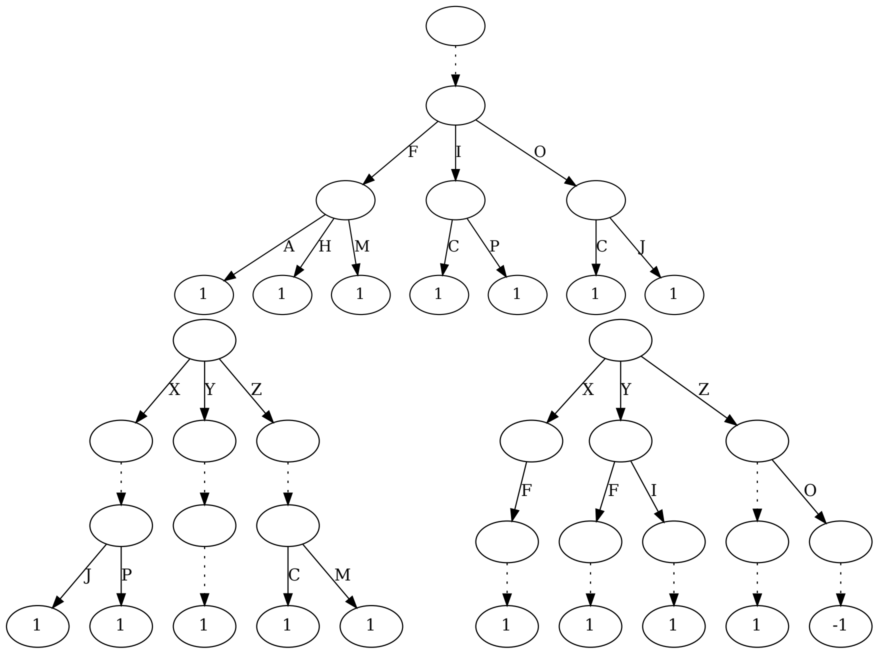

Figure 5.4: $\mathbf{NP}'$, $\mathbf{NL}'$ and $\mathbf{PL}'$ depicted as trees

5.6 Degree and Cardinality

In order to describe the join algorithm and its analysis we shall need a few auxiliary definitions. The *degree* of a trie, written $\deg R$, is defined as the number of nonzero non-$*$ mappings. Note that degree is not a linear function since it treats all nonzero mappings the same.

The *cardinality* of a trie is defined as the number of leaves. More precisely for a trie $R : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ define the cardinality $|R|$ as follows. If $R = 0$ then $|R| = 0$. Otherwise, if $k = 0$ then $|R| = 1$; if $k > 0$ then

$$|R| = |R(*)| + \sum_{a \in A_1} |R(a)|$$

Cardinality is only a linear function for the subset of tries where every leaf 1.

Example 5. *Consider the previous example trie* **NL** *as seen in Figure 5.2. We have*

$$\begin{aligned} \deg \mathbf{NL} &= 3 & |\mathbf{NL}| &= 5 \\ \deg \mathbf{NL}(X) &= 2 & |\mathbf{NL}(X)| &= 2 \\ \deg \mathbf{NL}(Y) &= 0 & |\mathbf{NL}(Y)| &= 1 \\ \deg \mathbf{NL}(Z) &= 2 & |\mathbf{NL}(Z)| &= 2 \end{aligned}$$

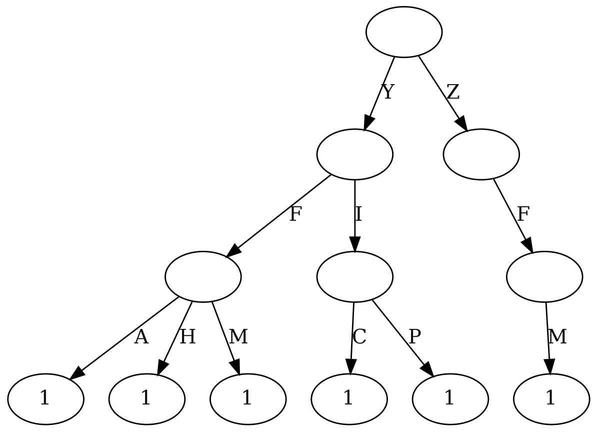

Figure 5.5: $\mathbf{NP}' \bowtie \mathbf{NL}' \bowtie \mathbf{PL}'$ depicted as a tree

5.7 Merging

Assume we are given tries $R_1, \ldots, R_n : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$. We can compute $\bowtie_i R_i$ by the following trie merging algorithm.

1. If $k = 0$ return $R_1 \cdot \ldots \cdot R_n$ using multiplication from K.
2. If $R_i = 0$ for any i return 0 (the empty trie).
3. Sort the tries so $\deg R_1 \leq \cdots \leq \deg R_n$.
4. For $1 \leq i \leq n$:
   (a) For each $a \neq *$ in R_i, generate the mapping
   $$a \mapsto (R_1(*) \cdots R_{i-1}(*) R_i(a) R^\dagger_{i+1}(a) \cdots R^\dagger_n(a))$$
   (b) If $R_i(*) = 0$, break.
5. If $R_j(*) \neq 0$ for all j, generate the mapping $* \mapsto (R_1(*) \cdots R_n(*))$.

The totality of the mappings generated by the loop can be described succinctly as

$$\sum_{1 \leq i \leq n} \sum_{1 \leq p \leq \deg R_i} a_p \mapsto (\bowtie_{1 \leq j < i} R_j(*)) \cdot R_i(a_p) \cdot (\bowtie_{i < j \leq n} R^\dagger_j(a_p))$$

In particular, if R_1 has no wildcard edges this becomes simply

$$\sum_{1 \leq p \leq \deg R_1} a_p \mapsto R_1(a_p) \cdot (\bowtie_{1 < j \leq n} R^\dagger_j(a_p))$$

This amounts to enumerating the edges of R_1 and looking up the corresponding keys in the other tries. Since R_1 has the fewest values to enumerate and since lookup only depends on the key (and is therefore independent of the size of the trie) this minimises work. More generally, sorting the tries ensures that the trie with the smallest degree is chosen at each step of the outer loop.

Theorem 6. *Running the trie merging algorithm on $R_1, \ldots, R_n : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ computes the join $R_1 \bowtie \cdots \bowtie R_n$.*

Proof. By induction on k. For $k = 0$ the result is trivial, so assume $k > 0$. The early return in step 2 is justified by the fact that join is a multilinear function, so any 0 among the inputs leads to a 0 in the output. Sorting in step 3 uses commutativity of the underlying ring, but is otherwise only a matter of efficiency; the rest of this argument does not depend on the tries being in any particular order.

Any trie R can be written as the sum $R' + R^*$ where $R^* = (* \mapsto R(*))$ contains only the $*$ edge and R', being the remainder, contains exactly the non-$*$ edges. The join can be rewritten as follows.[2]

$$\begin{aligned}
R_1 \cdots R_n &= (R_1' + R_1^*)R_2 \cdots R_n \\
&= R_1'R_2 \cdots R_n + R_1^*R_2 \cdots R_n \\
&= R_1'R_2 \cdots R_n + R_1^*(R_2' + R_2^*)R_3 \cdots R_n \\
&= R_1'R_2 \cdots R_n + R_1^*(R_2'R_3 \cdots R_n + R_2^*R_3 \cdots R_n) \\
&= R_1'R_2 \cdots R_n + R_1^*R_2'R_3 \cdots R_n + R_1^*R_2^*R_3 \cdots R_n \\
&\vdots \\
&= \left(\sum_{1 \le i \le n} R_1^* \cdots R_{i-1}^* R_i' R_{i+1} \cdots R_n \right) + R_1^* \cdots R_n^*
\end{aligned}$$

The loop in step 4 computes the summation and step 5 computes the last term. We exploit the fact joins of the form $R_i'R_{i+1} \cdots R_n$ can be computed by enumerating keys of R_i' and filtering by the rest. The break when $R_i(*) = 0$ is justified by the fact that all subsequent iterations will compute a join that involves $R_i(*)$ and so will give an empty result. By induction the recursive calls also compute the join correctly, finishing the proof. □

5.8 The Woes of Join

Computing joins in general is computationally hard. In the abstract this fact can be appreciated by encoding e.g. 3-SAT as a query; each propositional variable becomes a boolean attribute and each clause becomes a relation with seven entries representing the truth table of the clause. Thus, deciding if a query has any output is at least NP-hard.

[2] Recall that we can write $R \bowtie S$ simply by juxtaposition, RS.

Concretely, the challenge faced by join algorithms is that intermediate results can become huge even if the final output is small. Avoiding this problem entirely would probably entail solving P=NP affirmatively which—for the purposes of this discussion—we will assume to be impossible. For an example of a large intermediate result consider:

$$R = \sum_{\substack{1\leq i,j\leq n\\ i+j \text{ odd}}} (a_i \mapsto b_j \mapsto * \mapsto 1)$$

$$S = \sum_{\substack{1\leq i,k\leq n\\ i+k \text{ odd}}} (a_i \mapsto * \mapsto c_k \mapsto 1)$$

$$T = \sum_{\substack{1\leq j,k\leq n\\ j+k \text{ odd}}} (* \mapsto b_j \mapsto c_k \mapsto 1)$$

The join RST consists of elements $a_i \mapsto b_j \mapsto c_k \mapsto x$ where i, j and k have pairwise different parities, i.e. it is empty. Nevertheless after merging the first two attributes the intermediate result is a sum of elements which—up to symmetry—all look similar to

$$a_0 \mapsto b_1 \mapsto (\sum_{\substack{1\leq k\leq n\\ k \text{ odd}}} c_k \mapsto 1) \cdot (\sum_{\substack{1\leq k\leq n\\ k \text{ even}}} c_k \mapsto 1)$$

Only after resolving this innermost join does it become apparent that the output is empty. There are $\Theta(n^2)$ such elements taking $\Theta(n)$ apiece for a total time of $\Theta(n^3)$. If we let $N = \Theta(n^2)$ be the input size the merge algorithm runs in $\Theta(N^{\frac{3}{2}})$, which is superlinear in both input and output sizes.

An alternative approach—corresponding to traditional query methods—would be to compute $U = RS$ first and then compute UT afterwards. The large intermediate result is then U which is a sum of $\Theta(n)$ elements similar to

$$a_0 \mapsto \sum_{\substack{1\leq j,k\leq n\\ j \text{ odd}\\ k \text{ odd}}} b_j \mapsto c_k \mapsto 1$$

Each element has size $\Theta(n^2)$ so a total of $\Theta(n^3)$ is again unavoidable.

However, there is a sense in which the merge algorithm *almost* runs in linear time in the size of the output: if the input had been slightly different, e.g. replacing "odd" with "even" in the definition of T, then the output would be on the order of $\Theta(n^3)$, justifying the $\Theta(n^3)$ time spent to compute it. Such an adjustment of the input does not change its size nor its general shape (T is still a standard trie representing a relation over the second and third attributes).

This observation suggests that if a join algorithm is linear in the size of the largest possible output for any input of a similar shape and size, that algorithm should be considered efficient. It is then said to be *worst-case output optimal* [20]. Our merge algorithm satisfies this criterion, whereas traditional query methods like join-project do not [22] (indeed, worst-case output optimal algorithms are a recent invention in the traditional paradigm based on relational algebra [19, 27, 26], only predating our—simpler, more general—algorithm by a few years).

5.9 Input Padding

In order to reason about worst-case output size we introduce the notion of *padding*. Any trie R can be padded into $\langle\!\langle R \rangle\!\rangle$ by adding elements with canonically chosen attribute values such that the padded version is only larger by a constant factor. By carefully choosing the right padding function we get the property that for any other trie S of a similar shape the intersection $\langle\!\langle R \rangle\!\rangle \cdot \langle\!\langle S \rangle\!\rangle$ will be large even if $R \cdot S$ is small. Arguing that the merge algorithm runs in linear time (in the output size) on padded inputs, we will be able to conclude that is indeed worst-case output optimal.

Assume we have some finite collection of tries $R_1, \ldots, R_n : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ under consideration. Let $\# : \mathbb{N} \to A_i$ be a family of injections whose image is disjoint[3] from the support of all R_i. This will serve as our source for canonical attribute values.

Collapse For any trie R we define its *collapse* ∂R by replacing all non-$*$ edges by $\#1$ and setting all leaves to 1. More formally, define the map $\partial : K \to K$ by:

$$\begin{aligned} \partial 0 &= 0 \\ \partial x &= 1 \quad \text{for } x \neq 0 \end{aligned}$$

Extend it to tries:

$$\partial((\sum_i (a_i \mapsto R_i)) + (* \mapsto R)) = (\#1 \mapsto \bigcup_i \partial R_i) + (* \mapsto \partial R)$$

where union of collapsed tries is defined as

$$((\#1 \mapsto R) + (* \mapsto R')) \cup ((\#1 \mapsto S) + (* \mapsto S')) = (\#1 \mapsto (R \cup S)) + (* \mapsto (R' \cup S'))$$

in the general case and $x \cup y = \max(x, y)$ in the trivial case. The definition of ∂ assumes that all identical attribute values have already been coalesced, i.e. the a_i are all distinct (without this condition we would have $0 = \partial((a \mapsto 1) + (a \mapsto -1)) = \#1 \mapsto (\partial 1 \cup \partial(-1)) = \#1 \mapsto 1$, a contradiction).

Essentially this conflates all attribute values, so e.g. $R = \sum_i (a_i \mapsto * \mapsto b_i \mapsto c_i \mapsto * \mapsto x_i)$ collapses to $\partial R = \#1 \mapsto * \mapsto \#1 \mapsto \#1 \mapsto * \mapsto 1$. Here all the a_i, b_i and c_i have been replaced with $\#1$ for each of those attributes. The collapse of R represents the fact that R can be seen as a ternary relation over the first, third and fourth attributes.

[3]Disjointness is not essential, but it is harmless and makes the argument a little smoother.

Suppose we have some other trie, e.g. $S = * \mapsto * \mapsto * \mapsto * \mapsto * \mapsto 7$ which collapses to $\partial S = * \mapsto * \mapsto * \mapsto * \ \ \mapsto * \mapsto 1$. Then $\partial(R+S) = (\#1 \mapsto * \mapsto \#1 \mapsto \#1 \mapsto * \mapsto 1) + (* \mapsto * \mapsto * \mapsto * \ \ \mapsto * \mapsto 1)$, representing the fact that $R+S$ is both a ternary relation over the first, third and fourth attributes as well as a nullary relation over no attributes. This is a nonstandard trie and therefore not directly expressible using relational algebra.

Note that for (nonzero) standard tries we have $|\partial R| = 1$. In fact, this condition is a succinct way of expressing standardness, and intuitively $|\partial R|$ measures the degree to which R fails to be standard.

Padding Define the *shallow padding* $\langle\cdot\rangle$ of a trie as

$$\langle R\rangle = R \quad \text{for } k = 0$$
$$\langle(* \mapsto R) + \sum_{1\le i\le p}(a_i \mapsto S_i)\rangle = (* \mapsto \partial R) + \sum_{1\le i\le p}(\#i \mapsto \partial S_i)$$

Shallow padding works by replacing all concrete top-level edges of the trie with canonical elements supplied by #. Wildcard edges are left as is. Everything below the top level is collapsed.

Define the *deep padding* of R as

$$\langle\!\langle R\rangle\!\rangle = \sum_{\substack{0\le m\le k\\ S=R(a_1,\ldots,a_m)}} (a_1 \mapsto \cdots \mapsto a_m \mapsto \langle S\rangle)$$

Deep padding works by doing a shallow padding of all subtries. Note that since $S = R(a_1,\ldots,a_m)$ occurs in the union for every path $(a_1,\ldots,a_m)$ we have $R \subseteq \langle\!\langle R\rangle\!\rangle$, meaning that $R(a_1,\ldots,a_m) = 0$ whenever $\langle\!\langle R\rangle\!\rangle(a_1,\ldots a_m) = 0$. This argument exploits that the edges generated by # are disjoint and therefore do not accidentally cancel out with an existing edge.

Let R be as in the example above. In order to understand its deep padding we need to specify a few details about its shape. Suppose there are p distinct values among the a_i, given by $\{\hat{a}_1,\ldots,\hat{a}_p\}$. For any given $\hat{a}_j$ suppose there are q_j distinct values among the b_i in the subtrie $R(\hat{a}_j, *)$, given by $\{\hat{b}_{j,1},\ldots,\hat{b}_{j,q_j}\}$. And finally, for any given $\hat{a}_j$ and $\hat{b}_{j,k}$ suppose there are $r_{j,k}$ distinct values among the c_i in the subtrie $R(\hat{a}_j, *, \hat{b}_{j,k})$, given by $\{\hat{c}_{j,k,1},\ldots,\hat{c}_{j,k,r_{j,k}}\}$.

The deep padding of R is then

$$\begin{aligned}\langle\!\langle R\rangle\!\rangle = (&\sum_{1\le j\le p} \#j \mapsto * \mapsto \#1 \mapsto \#1 \mapsto * \mapsto 1)\\ + (&\sum_{1\le j\le p} \hat{a}_j \mapsto * \mapsto \#1 \mapsto \#1 \mapsto * \mapsto 1)\\ + (&\sum_{\substack{1\le j\le p\\ 1\le k\le q_j}} \hat{a}_j \mapsto * \mapsto \#k \mapsto \#1 \mapsto * \mapsto 1)\end{aligned}$$

$$
\begin{aligned}
&+ (\sum_{\substack{1\le j\le p\\ 1\le k\le q_j\\ 1\le l\le r_{j,k}}} \hat{a}_j \mapsto * \mapsto \hat{b}_{j,k} \mapsto \#l \mapsto * \mapsto 1)\\
&+ (\sum_{\substack{1\le j\le p\\ 1\le k\le q_j\\ 1\le l\le r_{j,k}}} \hat{a}_j \mapsto * \mapsto \hat{b}_{j,k} \mapsto \hat{c}_{j,k,l} \mapsto * \mapsto 1)\\
&+ (\sum_{\substack{1\le j\le p\\ 1\le k\le q_j\\ 1\le l\le r_{j,k}}} \hat{a}_j \mapsto * \mapsto \hat{b}_{j,k} \mapsto \hat{c}_{j,k,l} \mapsto * \mapsto \hat{x}_{j,k,l})
\end{aligned}
$$

To see why this amount of padding is sufficient consider some other trie

$$R' = \sum_i a'_i \mapsto * \mapsto b'_i \mapsto c'_i \mapsto * \mapsto x'_i$$

with the same cardinality. Its padding $\langle\!\langle R' \rangle\!\rangle$ will be similar to $\langle\!\langle R \rangle\!\rangle$ but with different width at each level, say p', q'_j and $r'_{j,k}$. When computing $\langle\!\langle R \rangle\!\rangle \cdot \langle\!\langle R' \rangle\!\rangle$ note that they have the first component of the sum in common and

$$\sum_{i\le j\le \min(p,p')} \#j \mapsto * \mapsto \#1 \mapsto \#1 \mapsto * \mapsto 1$$

will always occur as part of the output. This part of the output will serve to justify the time spent dealing with the first attribute.

A similar argument works for any of the other attributes. For instance, when dealing with the fourth attribute we will be considering the join $\langle\!\langle R \rangle\!\rangle(\hat{a}_j, *, \hat{b}_{j,k}) \cdot \langle\!\langle R' \rangle\!\rangle(\hat{a'}_{j'}, *, \hat{b'}_{j',k'})$ for some specific $a = \hat{a}_j = \hat{a'}_{j'}$ and $b = \hat{b}_{j,k} = \hat{b'}_{j',k'}$. Here, the fourth component of the sum comes to the rescue by ensuring that the output at the very least contains

$$\sum_{1\le l\le \min(r_{j,k}, r'_{j',k'})} a \mapsto * \mapsto b \mapsto \#l \mapsto * \mapsto 1$$

This is the essence of why padding can be used to argue for worst-case output optimality. The rest of this chapter is about proving this result, starting with some minor observations.

Lemma 7. *For any tries $R, S : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$*

1. $|\partial R| \le 2^k$
2. $\deg \langle\!\langle R \rangle\!\rangle \le 2 \cdot \deg R$
3. $\deg \langle R \rangle = \deg R$
4. $|\langle\!\langle R \rangle\!\rangle| \le 2^{k+1} \cdot |R|$
5. *If $R \ne 0$ and $S \ne 0$ then $\partial R \cdot \partial S \ne 0$.*

Proof. 1. For $k = 0$ the result is trivial. Otherwise:

$$\begin{aligned}|\partial R| &= |\partial(R(*) + \sum_i R(a_i))| \\ &= |\partial(R(*))| + |\partial(\sum_i R(a_i))| \\ &\leq 2^{k-1} + 2^{k-1} \\ &= 2^k\end{aligned}$$

2. Follows directly from the definitions of degree and padding.

3. By construction.

4. For $k = 0$ the result is trivial. Otherwise:

$$\begin{aligned}|\langle\!\langle R\rangle\!\rangle| &= |\langle R\rangle + (* \mapsto \langle\!\langle R(*)\rangle\!\rangle) + (\sum_i (a_i \mapsto \langle\!\langle R(a_i)\rangle\!\rangle))| \\ &\leq |\langle R\rangle| + |\langle\!\langle R(*)\rangle\!\rangle| + \sum_i |\langle\!\langle R(a_i)\rangle\!\rangle| \\ &\leq |R| + 2^k \cdot |R(*)| + \sum_i 2^k \cdot |R(a_i)| \\ &= |R| + 2^k(|R(*)| + \sum_i |R(a_i)|) \\ &= |R| + 2^k \cdot |R| \\ &\leq 2^{k+1} \cdot |R|\end{aligned}$$

□

5.10 Worst-case Output Size

Let $C_1, \ldots, C_n$ be collapsed tries, i.e. $C_i = \partial C_i$. Define the worst-case output size

$$\omega_{C_1,\ldots,C_n}(r_1, \ldots, r_n)$$

to be the maximum value of $|R_1 \bowtie \cdots \bowtie R_n|$ over all tries $R_1, \ldots, R_n$ with $\partial R_i \subseteq C_i$ and $|R_i| = r_i$.

In general, we can think of C_i as measuring how complex R_i is allowed to be in terms of wildcard usage. The traditional notion of worst-case output size is recovered when $|C_i| = 1$ for all i. In that case, the C_i function as the algebraic analogue of a database schema.

Example 8. *Consider singleton tries $R_1, \ldots, R_k : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ defined by:*

$$R_1 = a_1 \mapsto * \mapsto * \mapsto \cdots \mapsto * \mapsto * \mapsto 1$$

$$\begin{aligned}
R_2 &= * \mapsto a_2 \mapsto * \mapsto \cdots \mapsto * \mapsto * \mapsto 1 \\
R_3 &= * \mapsto * \mapsto a_3 \mapsto \cdots \mapsto * \mapsto * \mapsto 1 \\
&\vdots \\
R_k &= * \mapsto * \mapsto * \mapsto \cdots \mapsto * \mapsto a_k \mapsto 1
\end{aligned}$$

Let $S_i = R_i + 1$ where $1 = * \mapsto \ldots \mapsto * \mapsto 1$ is the unit of the join algebra. Then $|S_i| = 2$ for all i, but $|\bowtie_i S_i| = 2^k$. In general, the worst-case output size of nonstandard *joins is not limited by the AGM bound [22].*

Lemma 9. *Given positive integers $a_1, \ldots, a_n$ it holds that*

$$\omega_{C_1,\ldots,C_n}(a_1 r_1, \ldots, a_n r_n) \le \prod_i a_i \cdot \omega_{C_1,\ldots,C_n}(r_i, \ldots, r_n)$$

Proof. By symmetry it suffices to show that

$$\omega_{C_1,\ldots,C_n}(ar_1, r_2, \ldots, r_n) \le a \cdot \omega_{C_1,\ldots,C_n}(r_1, r_2, \ldots, r_n)$$

Let $R_1, \ldots, R_n$ be such that $|R_1| = ar_1$, $|R_i| = r_i$ for $i > 1$ and $|R_1 \cdots R_n|$ is maximal. Choose any decomposition $R_1 = \sum_{1 \le j \le a} S_j$ such that $|S_j| = r_1$. By linearity we have

$$R_1 R_2 \cdots R_n = (\sum_{1 \le j \le a} S_j) R_2 \cdots R_n = \sum_{1 \le j \le a} (S_j R_2 \cdots R_n)$$

The cardinality of the right hand side is clearly bounded by $a \cdot \omega_{C_1,\ldots,C_n}(r_1, \ldots, r_n)$ and since $R_1, \ldots, R_n$ was assumed to have maximal output size this is also an upper bound for $\omega_{C_1,\ldots,C_n}(ar_1, r_2, \ldots, r_n)$. □

5.11 Main Theorem

Lemma 10. *Suppose $R_i \subseteq R'_i$ for $1 \le i \le n$. Then computing $\bowtie_i R_i$ is asymptotically no slower than computing $\bowtie_i R'_i$.*

Proof. As long as the R_i and the R'_i tries have the same order when sorted by degree, the result is trivial. It therefore suffices to consider the "discontinuity" when two tries trade places. Suppose $\deg R_i \le \deg R_{i+1}$ and $\deg R'_i \ge \deg R'_{i+1}$. There must be intermediate tries $R_i \subseteq S \subseteq R'_i$ and $R_{i+1} \supseteq T \supseteq R'_{i+1}$ such that $\deg S = \deg T$. Again it suffices to consider the discontinuity at any such pair S and T.

Tracing the execution of the algorithm when S is sorted before T versus the opposite we see that the difference is as follows: in the former case we end up computing $S(a) \cdot T^{\ddagger}(a) = S \cdot (T(a) + T(*))$ and $S(*) \cdot T(a)$, in the latter case we end up computing $S^{\ddagger}(a) \cdot T(a) = (S(a) + S(*)) \cdot T(a)$ and $S(a) \cdot T(*)$. Both of these are at least as fast as computing the three joins $S(a) \cdot T(a)$, $S(a) \cdot T(*)$ and $S(*) \cdot T(a)$ separately. Computing the three separate joins is at the very worst $\frac{3}{2}$ times slower than either of the two approaches (applying the lemma inductively).

Discontinuities where tries trade places may happen more than once, but the number of times is in any case bounded by a function of n. $\square$

Theorem 11. *Let $R_1, \ldots, R_n : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ be arbitrary inputs for the join. Then the trie merging algorithm computes $\bowtie_j \langle\!\langle R_j \rangle\!\rangle$ in time $O(|\bowtie_j \langle\!\langle R_j \rangle\!\rangle|)$.*

Proof. By induction on k. For $k = 0$ the result is trivial. For $k > 0$ assume without loss of generality that $\deg \langle\!\langle R_1 \rangle\!\rangle \leq \cdots \leq \deg \langle\!\langle R_n \rangle\!\rangle$. Consider the work being done at step i in the loop. We can write $\langle\!\langle R_i \rangle\!\rangle$ as

$$\sum_{1 \leq p \leq \deg R_i} (a_p \mapsto \langle\!\langle R_i(a_p) \rangle\!\rangle) + (\#p \mapsto \partial(R_i(a_p)))$$

Looping through the a_p's and looking up $\langle\!\langle R \rangle\!\rangle_j(*)$ for $j < i$ as well as $\langle\!\langle R \rangle\!\rangle_j^\dagger(a_p)$ for $j > i$ takes $O(p)$ work. Looping through the $\#p$'s similarly takes $O(p)$ work.

For each $\#p$ note that since tries are sorted by degree we must in fact have $\partial(R_j(a_p)) \subseteq \langle\!\langle R \rangle\!\rangle_j^\dagger(\#p)$ for $j \geq i$. We must also have $\partial(\langle\!\langle R \rangle\!\rangle_j(*)) \subseteq \langle\!\langle R \rangle\!\rangle_j(*)$ for each $j < i$, otherwise the break condition of the loop would have activated. Hence we have $(\bowtie_{j<i} \partial(\langle\!\langle R \rangle\!\rangle_j(*)))(\bowtie_{j \geq i} \partial(\langle\!\langle R \rangle\!\rangle_j(a_p))) \neq 0$ for each p. This produces p units of output to which we ascribe the $O(p)$ work done.

We still need to account for the work done in recursive calls. The recursive joins involving collapsed tries compute in constant time since the sizes involved are bounded by a constant. For the remaining joins we note that the tries in the recursive call have the right shape (namely, $\langle\!\langle R \rangle\!\rangle$ for some R) to invoke the induction hypothesis. Such a recursive join looks like

$$T_i^p = (\bowtie_{1 \leq j < i} \langle\!\langle R_j(*) \rangle\!\rangle) \cdot \langle\!\langle R_i \rangle\!\rangle(a_p) \cdot (\bowtie_{i < j \leq n} \langle\!\langle R_j \rangle\!\rangle^\dagger(a_p))$$

Applying the induction hypothesis, the cost for computing this for all p for all i is bounded by

$$\sum_{1 \leq i \leq n} \sum_{1 \leq p \leq \deg R_i} |T_i^p|$$

We eliminate the inner sum by instead considering the cardinality of the union.

$$\sum_{1 \leq i \leq n} \left| \sum_{1 \leq p \leq \deg R_i} a_p \mapsto T_i^p \right|$$

The outer sum is eliminated analogously.

$$\left| \sum_{1 \leq i \leq n} \sum_{1 \leq p \leq \deg R_i} a_p \mapsto T_i^p \right|$$

This total is clearly bounded by the total size of the output, completing the proof. $\square$

Corollary 12. *Let $R_i, \ldots, R_n : A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K$ be tries with $\partial R_i \subseteq C_i$. Then trie merging runs in $O(\omega_{C_1,\ldots,C_n}(|R_1|, \ldots, |R_n|))$.*

Proof. By Theorem 11 trie merging computes $\bowtie_j \langle\!\langle R_j \rangle\!\rangle$ in time $O(|\bowtie_j \langle\!\langle R_j \rangle\!\rangle|) = O(\omega_{C_1,\ldots,C_n}(|\langle\!\langle R_1 \rangle\!\rangle|, \ldots, |\langle\!\langle R_n \rangle\!\rangle|))$. Since $R_j \subseteq \langle\!\langle R_j \rangle\!\rangle$ for each j we apply Lemma 10 to show that also $\bowtie R_j$ is computed in time $O(\omega_{C_1,\ldots,C_n}(|\langle\!\langle R_1 \rangle\!\rangle|, \ldots, |\langle\!\langle R_n \rangle\!\rangle|))$. By Lemma 4 and using Lemma 9 we have

$$\begin{aligned}\omega_{C_1,\ldots,C_n}(|\langle\!\langle R_1 \rangle\!\rangle|, \ldots, |\langle\!\langle R_n \rangle\!\rangle|) &\leq \omega_{C_1,\ldots,C_n}(2^{k+1}|R_1|, \ldots, 2^{k+1}|R_n|) \\ &\leq 2^{n(k+1)} \cdot \omega_{C_1,\ldots,C_n}(|R_1|, \ldots, |R_n|)\end{aligned}$$

Consequently, computing $\bowtie_j R_j$ using trie merging takes
$O(\omega(|R_1|, \ldots, |R_n|))$ steps. □

Corollary 13. *Algebraic join is worst-case optimal for standard joins.*

Proof. By Corollary 12 where $|C_i| = 1$ for all i. □

Chapter 6

Implementation

We now present an overview of how to implement the theory using Haskell. The design has roots in the work of Henglein and Larsen [28], but has diverged significantly.

We first give the encoding of modules as a kind, and then illustrate how to represent vectors efficiently. This is followed by an explanation of run-time optimisations and evaluation of homomorphisms. Finally, we discuss some of the finer points of Generalised Algebraic Data Types and Type Families and the importance of natural transformations through their connection with polymorphism.

6.1 Spaces

We begin by declaring an abstract representation of spaces (i.e. modules/algebras):

```
data Space :: Type where
    -- | Scalars.
    K :: Space
    -- | Finite map.
    Copi :: b -> Space -> Space
    -- | Map.
    Pi :: b -> Space -> Space
    -- | Direct sum.
    (:+:) :: Space -> Space -> Space
    -- | Tensor product.
    (:*:) :: Space -> Space -> Space
    -- | Polynomials.
    Poly :: Space -> Space
    -- | Linear maps.
    (:->) :: Space -> Space -> Space

type Free b = Copi b K
type Cofree b = Pi b K
type Dual v = v :-> K
```

On the face of it `Space` is a type and identifiers like `Copi` are terms. However, we are going to use the "Data Kinds" GHC extension, which allows promoting types to kinds and terms to types. This can be used to model categories with `Type` being the category SET of sets and functions[1] and `Space` being MOD_K (for any choice of K). Haskell does not allow direcly talking about objects of types of kind other than `Type`, but we can use the `Space` kind as a domain-specific type system that is interpreted by a suitable functor.

6.2 Vectors

We now turn to the representation of vectors. To this end we use a type `Vec k v` which can be thought of as a family of forgetful functors $|\cdot| : \text{MOD}_K \to \text{SET}$, one for each K. The representation will be dependent on `v`, but in all cases we need closure under addition and scalar multiplication. Hence we use the following definition

```
data Vec k v = Zero
             | Add (Vec k v) (Vec k v)
             | Mul k (Vec k v)
             | Gen (Gen k v)
```

where `Gen k v` is a subset that generates the space (it is not necessarily a basis). The space-dependent part of the representation is defined as a type family:

```
data family Gen k (v :: Space) :: Type
```

Elements of `Gen k v` can be thought of as being in "weak head canonical form" — they contain information specific to `v` though their subterms are not necessarily generators. The separation of vectors into `Vec k v` and `Gen k v` turns out to be very useful since we can handle linearity in a generic manner. It also enables us to express in the type system that a function is defined by its action on generators.

The simplest instance of `Gen k v` is the *scalar* space:

```
newtype instance Gen k K = Scalar k
```

For *direct sums* (i.e. binary coproducts/products) we store each component:

```
data instance Gen k (u :+: v) = DirectSum (Vec k u) (Vec k v)
```

Generators for *tensor products* have a completely analogous declaration:

```
data instance Gen k (u :*: v) = Tensor (Vec k u) (Vec k v)
```

The difference is that homomorphisms act bilinearly on them. Note that we could have defined `Tensor` as taking generators rather than vectors and this would be perfectly adequate for representing any vector since addition and multiplication can be pushed out. Unfortunately, this would force us to always

[1]Haskell does not actually implement the semantics of SET, one of the reasons being that functions are partial. However, the parts of Haskell that we actually employ ultimately stay within the "SET-like" subset.

factor tensor products into a canonical form which incurs a quadratic size increase in the worst case. Depending on what operations we may wish to perform on a tensor product this price does not necessarily need to be paid.

For *finite maps* we could just take the categorical definition literally and define:

```
data instance Gen k (Copi b v) = Inj b
```

This is at odds with how we want to think about finite maps operationally — as database indices allowing efficient querying — and this goal is not served well by what essentially amounts to an unordered list. However, the above definition is the only adequate polymorphic solution. Note that to demonstrate the existence of finite maps we actually need decidable equality (which relies on Excluded Middle) on the index set which is impossible to implement in general. This is not much of a problem since in practice we only need to handle indices which are first order data (i.e. types built using primitives, sums, products and recursion). Therefore we will use an adaptive representation depending on the index type.

A unit index is simply represented by a single vector:

```
newtype instance Gen k (Copi () v) = CopiUnit (Vec k v)
```

Finite precision integers use a patricia tree (named `IntMap` in the Haskell standard library):

```
newtype instance Gen k (Copi Int v) = CopiInt (IntMap (Vec k v))
```

Missing keys are interpreted as yielding a zero.

Sum types are represented by two finite maps, one for lefts and one for rights:

```
data instance Gen k (Copi (Either b c) v) =
       CopiSum (Vec k (Copi b v)) (Vec k (Copi c v))
```

Product types exploit the isomorphism $B \Rightarrow C \Rightarrow V \cong (B \times C) \Rightarrow V$ by currying:

```
data instance Gen k (Copi (b, c) v) = CopiProd (Vec k (Copi b (Copi c v)))
```

This kind of adaptive representation where the structure of a finite map resembles the structure of the domain type is very much like a generic trie [13]. Yet in the context of linear algebra this data structure arises naturally as the most direct realisation of the various factorisation theorems. The common approach of assigning some (perhaps arbitrary) order to the index set and using search trees is, by comparison, less natural since order relations do not occur anywhere in our theory.

Now, how do we implement $(\mapsto) : B \to V \to B \Rightarrow V$ given our choice of representation? Ideally, we want something like:

```
inj :: b -> Vec k v -> Vec k (Copi b v)
```

However, the type signature cannot be implemented because `b` is polymorphic (this is actually a feature as discussed in Section 6.5). To solve this problem, we introduce a type class to identity the subset of types that are usable as finite map indices. Since we are limiting ourselves to first order data the class is aptly named:

```
class Eq b => FirstOrder b where
  inj :: b -> Vec k v -> Vec k (Copi b v)
  intersect :: (Ring k) => (Vec k u -> Vec k v -> Vec k w)
                        -> Vec k (Copi b u) -> Vec k (Copi b v)
                        -> Vec k (Copi b w)
```

The `Eq` superclass is not strictly necessary, but it makes good sense because `Eq` represents the notion of decidable equality (or possibly equivalence) on some type.

Implementations of `inj` are generally straightforward. For instance, products are handled as follows:

```
inj (b, c) x = Gen . CopiProd $ inj b (inj c x)
```

The `intersect` method is crucial for good performance. It takes a merge function and two finite map vectors and produces a new finite map with their common indices and merged values. The special case where one of the finite maps is a singleton yields a lookup function:

```
lookup :: (FirstOrder b) => b -> Vec k (Copi b v) -> Vec k v
lookup b x = mulH `ev` intersect (inj b one) x
```

For primitive types `intersect` uses an appropriate intersection algorithm from the underlying data structure:

```
intersect f x y = Gen . CopiInt $
    IntMap.intersectionWith f (fromCopiInt $ canonicalCopiInt x)
                              (fromCopiInt $ canonicalCopiInt y)
```

It makes use of an auxiliary function for computing the weak head canonical form.

```
canonicalCopiInt :: (Ring k) => Vec k (Copi Int v) -> Gen k (Copi Int v)
```

It is hard to overstate the importance of this approach; from this all of our algorithmic efficiency is derived. By handling data in bulk `intersect` runs in linear time — `canonicalCopiInt` and `IntMap.intersectionWith` both take linear time. Contrast this with a naive expansion by linearity in both arguments incurring a quadratic penalty in the worst case.

Intersection of sums proceeds by separating into lefts and rights (using `partitionCopiSum`) and calling recursively.

```
intersect f x y = Gen $ CopiSum (intersect f xb yb) (intersect f xc yc)
        where
           (xb, xc) = partitionCopiSum x
           (yb, yc) = partitionCopiSum y
```

Intersection of products consists of unraveling the underlying nested finite maps (using `curryCopiProd`) and then calling `intersect` recursively.

```
intersect f x y = Gen . CopiProd $ intersect (intersect f)
                                             (curryCopiProd x)
                                             (curryCopiProd y)
```

We can show that `intersect` runs in linear time in general. Define the size of a term to be the size of its representation when viewed as a tree, counting elements of `Vec k u` and `Vec k v` as unit size (they are essentially opaque pointers).

Lemma 14. *If f runs in time $O(m' + n')$ for all `x'` of size m' and `y'` of size n' and if `x :: Vec k (Copi b u)` has size m and `y :: Vec k (Copi b v)` has size n then `intersect f x y` runs in time $O(m + n)$.*

Proof. By induction on the type `b`. First note that all the auxiliary functions run in linear time and produce output of linear size. For primitive types we rely on the time bound provided by the integer map implementation; see e.g. Okasaki and Gill [29]. The case for sums follows by straightforward induction. Finally, for products we first use induction to establish that `intersect f` runs in linear time, then by induction again we get the desired result. □

At this point it is pertinent to remark on the cost model. We only admit primitive types that can be represented in a finite number of bits and consequently allow efficient indexing by e.g. patricia trees. Large numbers must be represented by, say, a list of words and therefore contribute more to the size of the input. Hence, we are speaking of linearity in the amount of data and not the number of elements. For a more thorough explanation of this distinction see "Generic top-down discrimination for sorting and partitioning in linear time" [30].

For *products* we use the representation suggested by the universal property:

```
data instance Gen k (Pi b v) = PiExt (b -> Vec k v)
```

For *homomorphisms* there is no fully adequate representation. Either we provide a general constructor for promoting functions of type `Vec k u -> Vec k v` to `Hom k (u :-> v)` with no guarantee of linearity; or we provide constructors for specific classes of homomorphisms in such a way that linearity is unavoidable, but we miss out on homomorphisms that are linear for less obvious reasons. We choose to omit a general purpose "unsafe" constructor, since all the linear maps we actually care about can be represented using a reasonable number of constructors.

The type is declared using Generalised Algebraic Data Types (GADT's) allowing each constructor to constrain the type indices:

```
data instance Gen k (u :-> v) where
```

We have completely general homomorphisms like identity and composition:

```
IdH :: Gen k (v :-> v)
ComposeH :: Vec k (v :-> w) -> Vec k (u :-> v) -> Gen k (u :-> w)
```

The basic module operations have constructors too:

```
AddH :: Gen k (v :+: v :-> v)
MulH :: Gen k (K :*: v :-> v)
```

Previously established natural transformations provide a canonical way of mapping out of each type of space:

```
PiH :: Vec k (Copi b (u :-> v)) -> Gen k (Pi b u :-> v)
CopiH :: (FirstOrder b) => Vec k (Pi b (u :-> v)) -> Gen k (Copi b u :-> v)
ScalarH :: Vec k v -> Gen k (K :-> v)
SumH :: Vec k (u :-> w) -> Vec k (v :-> w) -> Gen k (u :+: v :-> w)
TensorH :: Vec k (u :-> v :-> w) -> Gen k (u :*: v :-> w)
```

They cover all possible homomorphisms out of their respective domains, safe for `PiH` which is not in general capable of expressing all homomorphisms from products[2].

In addition, there are a handful of special-case homomorphisms that allow more efficient execution under the right circumstances. For instance, the $\otimes$ functor applied to two functions acts independently on each part of a tensor product:

```
LiftTensorH :: Vec k (u1 :-> v1) -> Vec k (u2 :-> v2)
            -> Gen k (u1 :*: u2 :-> v1 :*: v2)
```

6.3 Run-time Optimisation

A given vector can be represented in many ways. In fact, we have deliberately tried to maximise the number of choices in representation. The idea is that we cannot determine statically how producers and consumers will be connected. Thus, producers construct their output in the locally most convenient format and consumers are then expected to factor their input as necessary. Consumers benefit too if they are able to work directly with the (potentially much smaller) input.

There is a caveat, though. What if a value is shared by many consumers? In that case, there is a possibility that they are all performing the exact same factorisation. This is seldom an asymptotic disadvantage, since most homomorphisms only need to factor the part of the input that they are processing anyway.

Nevertheless, there might be practical efficiency gains to be had by suitably caching the result of factorisations. A simple way would be to expand `Vec k v` with a constructor `Cached (Vec k v) (Vec k v)` where the first element is the producer's preferred form and the second element is some (not necessarily fully) factorised form.

[2]When `b` is countable and `k` is finite it *is* sufficient to represent all *computable* functions!

6.4 Evaluation

The most important function in the implementation is the evaluation map, which represents $\mathrm{ev} : (U \to V) \otimes U \to V$. It is declared as follows:

```
ev :: (Ring k) => Vec k (u :-> v) -> Vec k u -> Vec k v
```

Apart from a case for zero elements, it exploits linearity in the function argument to reach a generator:

```
ev Zero _ = zero
ev _ Zero = zero
ev (Add f g) x = f `ev` x .+. g `ev` x
ev (Mul k f) x = k *. f `ev` x
ev (Gen f) x = evGen f x
```

Evaluation where the function is a generator is performed by an auxiliary function:

```
evGen :: (Ring k) => Gen k (u :-> v) -> Vec k u -> Vec k v
```

Some homomorphisms can be applied directly to non-generator parameters. This primarily includes completely generic homomorphisms such as identity and composition:

```
evGen IdH x = x
evGen (ComposeH f g) x = f `ev` (g `ev` x)
```

Otherwise, we use linearity to reduce the parameter argument to a generator:

```
evGen f (Add x y) = f `evGen` x .+. f `evGen` y
evGen f (Mul k x) = k *. f `evGen` x
evGen f (Gen x) = evGenGen f x
```

Finally, we have an auxiliary function for when both arguments are generators:

```
evGenGen :: (Ring k) => Gen k (u :-> v) -> Gen k u -> Vec k v
```

This is where most of the dynamic optimisation takes place. For instance

```
evGenGen (LiftTensorH f g) (Tensor x y) = f `ev` x .*. g `ev` y
```

where `.*.` is infix tensor product.

6.5 GADT's Versus Type Families

There are generally two ways of programming with adaptive representations in Haskell: GADT's and Type Families. We have chosen to encode `Gen k v` as a type family, while Henglein and Larsen [28] use GADT's. This choice deserves an explanation.

Using GADT's we would have something like

```
data Gen k :: Space -> Type where
  Scalar :: k -> Gen k K
  DirectSum :: Vec k u -> Vec k v -> Gen k (u :+: v)
  Tensor :: Vec k u -> Vec k v -> Gen k (u :*: v)
  ... etc ...
```

An immediate difference is that using GADT's we have a *closed world* — all constructors that are or ever will be are listed in one place. Type families, by contrast, are open to extension. This is of limited use in our case since the kind of spaces is already closed, the main advantage being that new primitive finite map index types can be supported.

The main reason for our choice has do with parametricity, the notion that polymorphic values can only be handled opaquely. In other words, they are natural transformations in HASK[31].

GADT's, being closed, allow pattern matching which provides information about a polymorphic type. For example:

```
getScalar :: Gen k v -> Maybe k
getScalar (Scalar k) = Just k
getScalar _ = Nothing
```

Thus, parametricity does not provide much when GADT's are involved since they come equipped with a too powerful elimination principle. When `Gen k v` is declared as a type family `getScalar` will not type check — we are only allowed to match on a constructor when we know for certain that the type is compatible, such as in the following:

```
getScalar :: Gen k K -> k
getScalar (Scalar k) = k
```

More interestingly when we write a type signature like `f :: Vec k (u :*: v) -> Vec k (v :*: u)` then (providing it is linear) `f` represents a natural transformation in MOD_K. The inability to pattern match on polymorphic values means that functions from type families are harder to write, but easier to reason about.

Of course, sometimes we do want to write unnatural functions! Observe that $\text{dual}_I : (I \Rightarrow V) \to (I \to V)$ is an unnatural embedding. Indeed, we cannot (reasonably) implement a function with such a type signature:

```
dual :: Vec k (Copi b v) -> Vec k (Pi b v)
```

This also makes sense operationally since we only admit first order indices in finite maps. Consequently, the solution is to add the appropriate type class constraint:

```
dual :: (FirstOrder b) => Vec k (Copi b v) -> Vec k (Pi b v)
dual x = Gen . PiExt $ \b -> lookup b x
```

Note that this is not a violation of parametricity. Rather, the meaning of parametricity depends on the type and adding the `FirstOrder` constraint results in a correspondingly weaker property. In summary, the choice of type families does not limit what operations can be implemented, but it enables us to express assumptions clearly and enforces honesty.

Chapter 7

Discussion

7.1 Related Work

We have touched many big areas of research: algebra, databases, language design, semantics and more. Naturally, the magnitude of potentially related work is enormous. The most pertinent references are explained below, separately for each chapter.

7.1.1 Algeo

Algebraic λ-calculi. A related approach to computing with linear algebra is the idea of extending the λ-calculus with linear combinations of terms [8, 9], though such approaches do not provide generalised reversibility in the form of adjoints. Extending the λ-calculus in this way is a delicate ordeal that easily leads to collapse (e.g. $\overline{0} = \overline{1}$) from interactions between sums and fixpoints. Algeo evades these problems by taking a different view of functions, namely that base elements of type $\tau_1 \to \tau_2$ are $b_1 \mapsto b_2$ where b_1 and b_2 are base elements, and function application is linear in *both* arguments (this approach was briefly considered in [9] and discarded due to wanting a strict extension of untyped λ-calculus).

It is possible in Algeo to define an abstraction $\lambda x.e$ as syntactic sugar for $[\widehat{x} : \langle\tau_1\rangle]\,\widehat{x} \mapsto e^{x:=!\widehat{x}}$ of type $\langle\tau_1\rangle \to \tau_2$. The input must be a bag type to model the nonlinearity of function application in algebraic λ-calculi. However, the fixpoint-esque operators definable in Algeo have different semantics than in standard λ-calculus and do not allow the kind of infinite unfolding that so easily leads to paradoxes. The simplest such operator is $\mathbf{fix} : (\tau \to \tau) \to \tau$ defined by $\mathbf{fix}\ (\mathbf{x} \mapsto \mathbf{x}) \Leftrightarrow \mathbf{x}$, which is just a repackaged version of $\bowtie$. To get something approaching the usual concept of fixpoint we again need bags:

$$\mathbf{fix} : \langle\tau \to \tau\rangle \to \tau \qquad\qquad \mathbf{fix\ f} \Leftrightarrow \mathbf{!f}\ (\widehat{\mathbf{!fix}\ \mathbf{f}})$$

Even ignoring the bag operations $\mathbf{fix}$ is not a fixpoint combinator in the λ-calculus sense since $\mathbf{fix}\ e = !e\ (\widehat{!\mathbf{fix}}\ e)$ does not hold in general when e is not a base value. We can try to create a paradox by considering e.g. $e = \mathbf{fix}\ \langle [\mathbf{x} : \tau]\, \mathbf{x} \mapsto \overline{-1}; \mathbf{x} \rangle$. It is indeed the case that $e = (\overline{-1}; e)$ and therefore that $e = 0$, but this only emphasises what we already know: that Algeo is powerful enough to express arbitrary constraints and that recognising 0 is uncomputable in general.

Reversible and functional logic programming. Reversible programming languages have seen a great deal of research in recent years thanks to their applications in surprisingly diverse areas such as debugging [32, 33], robotics [34], discrete event simulation [35] and quantum computing [36, 37, 38]. For this reason, many different styles of reversible programming have been explored, notably imperative [39], object-oriented [40], functional [41, 42, 43], and parallel [43, 32].

The functional logic paradigm of programming was pioneered by languages such as Curry [44, 45] and Mercury [46]. Along with reversible functional programming languages such as Rfun [41], CoreFun [47], and Theseus [48], they have served as inspiration for the design of Algeo. Using the prefix *algebraic* was explored by ALF (Algebraic Logic Functional programming language) [49], albeit this use of *algebra* is—like in so many other cases—different from ours. Unlike Algeo, neither of the conventional functional logic programming languages come with an explicit notion of multiplicity and the added benefits in expressing data of a probabilistic, fractional, or an "inverse" nature, nor do they have adjoints. On the other hand, while the reversible functional languages all have a notion of inversion, their execution models and notions of reversibility differ significantly from those found in Algeo.

Modules, databases, and query languages. Free modules can be seen as a form of generalised multisets. When permitting negative multiplicities, this allows the representation of database table schemas as (certain) free modules, tables as vectors of these free module, and linear maps as operations (e.g., insertion, deletion, search, aggregation, joins, and much more) acting on these tables. The structural theory of modules that led to the development of Algeo, and its relation to database representation and querying, is described in [4].

Abstract Stone duality. Abstract Stone duality (see, e.g., [50]) is a synthetic approach to topology and analysis inspired by Stone's famous duality theorem between categories of certain topological spaces and certain order structures. An interesting feature of abstract Stone duality is that it permits the indirect definition of numbers from a description (i.e., a predicate) via a method known simply as *definition by description*, provided that it can be shown that a description is true for exactly one number. This is not unlike the indirect description of terms in Algeo, though instead of requiring descriptions to be unique, the result of an indirect description in Algeo is instead the *aggregation* over all terms satisfying this description.

7.1.2 Query Processing

Our modules and associative algebras provide a general, established and mathematically deep and well-studied reference framework for structures proposed in the literature, such as provenance semirings [51] and semiring dictionaries [52]. For example, in provenance semirings a K-relation is an element of $\mathbf{F}_K[T_{a_1} \times \ldots \times T_{a_n}]$. Replacing K by $R = \mathbb{N}[X]$, the free commutative semiring generated by X yields queries evaluated over R instead of K. This incorporates provenance for aggregations [53] since $\mathbf{F}_K[T]$ provides folding over arbitrary modules. Our extension with wildcards extends provenance from finite K-relations to infinite K-relations. Further, our approach accounts for probabilistic databases [54] as algebras $\mathbf{F}_{\mathbb{R}}[T]$, with real numbers serving as quasi-probabilities, as well as sets with negative multiplicity (see, e.g., [55, 56]).

Linear algebra has previously been proposed as a framework for interpreting and implementing querying on databases. Notably, typed matrix algebra [57, 58, 59, 60] has been used to provide operations on matrix-shaped data (see also array programming languages such as Futhark [61] and Single Assignment C [62]). Note that our approach works on infinite-dimensional spaces, not only finite-dimensional ones, and it is essential to represent linear maps as functions rather than matrices. More closely related to our work, there are expressive linear algebra-based domain-specific languages/frameworks [63, 64] that provide expressive database and data analytics operations, with evaluation to standard normalised data structures using Kiselyov's tagless final approach. These do not support efficient data structures and execution of *bi*linear maps, however, where we employ symbolic tensor products and efficient intersection, aided by compact maps for both expressiveness and efficiency.

A probabilistic database is database that associates a probability from $[0, 1]$ with each tuple in the relations. There are different treatments, depending on the operations executed on them; see for example [54]. The *measure* of a relation is the sum of probabilities associated with its elements. If the measure is 1, the relation represents a (finite) probability distribution.

The natural setting of probabilistic databases as algebras is $\mathbf{F}_{\mathbb{R}}[T]$, with real numbers serving as quasi-probabilities since addition and subtraction can move them out of $[0, 1]$. It is possible, however, to define a nonlinear operation $x \vee y$, as in fuzzy logic [65]:

$$x \vee y = (x + y) - (xy).$$

It has the property that $x \vee y \in [0, 1]$ if $x, y \in [0, 1]$ and can thus be used to model (nonlinear) union of probabilistic databases.

The problem of efficient execution of relational database queries has been studied extensively, with joins in particular posing a challenge. Traditional methods rely on decomposing k-ary joins into binary joins, carefully creating a good *query plan* in order to minimise the sizes of intermediate results and outputs [14], while our approach instead hinges on sensible and efficient *expression simplification* (see [66] for a formal treatment of this problem). While it is possible to evaluate *acyclic* join queries in time linear in the size of the query, the input, and the output [15, 16] and there are methods that deal with "almost" acyclic join queries [17], this is infeasible for cyclic queries such as the triangle query, as deciding whether the output is empty is NP-hard [18]. To quantify how good a query algorithm is, the notion of *worst-case optimal complexity* was developed [19, 20], focussing on the *data complexity* of the problem, which ignores the size of the query itself [21]. A join algorithm is *worst-case optimal* if it executes in time linear in N and O, where O is the maximal size of an output of the join query applied to input relations whose sizes sum to N.

We have drawn on a great number of categorical concepts, not only to drive generalisation and identify suitable compositional constructs that make for a pleasant framework with strong and useful properties, but also to exploit term constructors as symbolic data structures and exception-less algebraic equalities for their efficient execution. It is difficult to do justice to the many works in category theory, abstract algebra and categorical functional programming we have built on. Most closely related is [67], which presents a framework for relational algebra based on the adjunctions that generate it.

7.1.3 Simplification

We have shown that relational joins are special kinds of algebraic joins over semimodules whose elements can be efficiently represented as tries with a wildcard that provide representing finite and certain infinite relations. We have designed an algebraic join algorithm by merging tries whose worst-case optimality follows from an input padding argument that intuitively provides a shadow element to each input element.

In this fashion we formulate the K-relations framework [51] in terms of associative-commutative K-algebras on semimodules, where join is the multiplication and show that they can be implemented in worst-case optimal time irrespective of the underlying commutative semiring. This is done by trie merging on generalized tries with wildcards that can represent not only finite relations, but also infinite ones. Our algorithm is similar to Leapfrog Join [26] in the sense that it uses tries, but it iterates over tries not in sorted order of the keys (edge labels), but processes trie nodes in order of increasing outdegree. One of the key ideas is to reduce the problem one attribute at a time rather than one join at a time [26, 2]. Precursors to this approach can be found in e.g. approaches to Bayesian network inference [68, 69] and SAT solving [70].

Various generalizations of the join problem have been studied, such as for probabilistic databases [54], aggregating over arbitrary semirings [51] or the general notion of functional aggregate queries [25]. Although usually not emphasized much, various generalizations correspond to different notions of collections: ordinary sets (relational algebra), multisets (bag semantics, or equivalently using multiplicities from the semiring $\mathbb{N}$), polysets [3] (using multiplicities from the ring $\mathbb{Z}$), unnormalized probabilistic sets [54] (using multiplicities from the semiring $\mathbb{R}_+$).

When discussing complexity of data the representation scheme is important. The traditional approach is a simple list of tuples, but more compact structures are possible such as generic tries [13], symbolic Cartesian products with [2, 3] or without symbolic multiplicities [71, 72, 28, 73].

7.2 Future Work

We first describe possible future work for each chapter in isolation and then give a bird's eye view of how the project as a whole could develop.

7.2.1 Algeo

We have presented the reversible functional logic programming language Algeo, described its syntax and type system, and given it semantics in the form of a system of equations. We have illustrated the use of Algeo through applications and examples, and described applications in areas such as database querying and logic programming with an improved notion of negation.

As regards avenues for future research, we consider developing an implementation based on this work to be a logical next step. However, this is not as trivial as it may appear at first glance, as it requires the development of strategies for performing nontrivial rewriting using the equational theory. In particular, we don't believe that there is an obvious optimal evaluation strategy for Algeo, as it would have to optimally solve all expressible problems (e.g., matrix diagonalisation, three-way joins).

An extension to Algeo not considered here is that of *dual types*, reflecting the notion of *dual* modules and vector spaces in linear algebra. To include these would permit Algeo to use multiplicities in the complex numbers, in turn paving the way for using Algeo to express quantum algorithms.

We would also find it interesting to use Algeo to study polylogic (as described in Section 3.5), in particular its use as a reasonable semantics for negation not involving the impure and unsatisfying negation-as-failure known from Prolog. Finally, since Algeo permits aggregating over infinite collections of values, it seems that there is at least some connection to nonstandard analysis and linear algebra (see also [74]) which could be interesting to elaborate. In fact, permitting the use of nonstandard real (or complex) numbers as multiplicities would allow automatic differentiation (see [5] for a recent, combinatory approach to automatic differentiation on Hilbert spaces) to be specified in an exceedingly compact manner, which could lead to further applications in machine learning and optimization.

7.2.2 Query Processing

While logic programming does well in handling positive queries, negation is typically handled in an ad-hoc and unsatisfactory way, for example through *negation as failure*. In our setting, data with *negative* multiplicity is handled no different from data in *positive* quantity: can this give a more satisfactory treatment of negation in logic programming? Further, in the current form, our setting is quite conservative in that it is not capable of describing recursive queries, even if they are ultimately well-behaved. A possible solution to this shortcoming would be to study modules which are somehow topological (e.g., appropriately ordered, or equipped with a norm or inner product), as this could give us access to fixed point theorems directly associated with the semantics of recursion.

7.2.3 Simplification

Generalisation of wildcards. All of our constructions have nice categorical properties. Let us give them the following names:

$$\begin{aligned} W(A) &= A^* \\ T(A_1, \ldots, A_k) &= A_1 \Rightarrow^* \cdots \Rightarrow^* A_k \Rightarrow^* K \\ T^{\mathrm{op}}(A_1, \ldots, A_k) &= A_1 \to \cdots \to A_k \to K \end{aligned}$$

They are in fact functors $W : \mathbf{Set} \to \mathbf{Set}$, $T : \mathbf{Set}^k \to \mathbf{Mod}$ and $T^\circ : \mathbf{Set}^k \to \mathbf{Mod}^{\mathrm{op}}$. Deep lookup is actually a dinatural transformation $(\cdot)^\ddagger : T \circ W^k \dot{\to} T^{\mathrm{op}}$ (where W^k means W applied component-wise to k-tuples). A natural question to ask is for what choices of W and $(\cdot)^\ddagger$ will things "work out".

As remarked earlier there is a redundancy for finite sets where $*$ is not strictly necessary, but still useful for compact representations. This is not a wart to be concealed, but rather an avenue to explore. What happens if we add more redundant symbolic representatives of subsets? Can we extend the scope of problems that algebraic joins can deal with or improve efficiency of existing ones?

More algebra. Let $K = \mathbb{Q}$ and suppose

$$\begin{aligned} R &= (* \mapsto 3) + (a \mapsto 2) \\ S &= (* \mapsto \tfrac{1}{3}) - (a \mapsto \tfrac{2}{15}) \end{aligned}$$

It can be verified that $RS = 1$ so we would be justified in writing $R^{-1} = S$. On the other hand clearly not all nonzero tries have inverses — to wit, any trie that does not represent a cofinite set.

As another example, let K be closed under square roots. Then any trie T has a square root given by

$$\sqrt{T} = (* \mapsto \sqrt{T(*)}) + \sum_a (a \mapsto \sqrt{T(*) + T(a)} - \sqrt{T(*)})$$

This naturally prompts a host of questions. Exactly which set of tries have inverses and what does that mean relationally? What about the largest algebraically closed subset? Can we factorize tries meaningfully? Can we gainfully apply spectral decomposition to linear maps between tries? And so on and so forth.

Functional aggregate queries Since our algebraic approach can easily deal with ordinary joins in a worst-case optimal manner, a natural next step would be to tackle functional aggregate queries (FAQ's)[25]. There is similarity in the use of semirings, though some work is needed to integrate FAQ's with a more principled algebraic foundation such as ours. Nevertheless, if a proper algebraic foundation can be found for this wider class of problems it would not be surprising if a similarly simple worst-case optimal algorithm were to emerge.

7.2.4 Bird's Eye View

The next major step in the research programme is to craft an efficient implementation of Algeo and use that to evaluate queries. As noted previously Algeo programs are hard to evaluate in general, but the subset corresponding to query expressions is significantly simpler. In particular, attaining worst-case output optimality for conjunctive queries should very much be possible. More ambitiously, a parallel implementation of Algeo by compiling to e.g. Futhark [61] would be interesting.

There is also the question of how to go from query evaluation to an actual database system. Theoretically, the framework supports all the basic trappings of a system: schemas, insertion/deletion, etc. Practically, many details need to be worked out. One could see Algeo as similar to Prolog with its natural support for persistent relations. More modern developments like the Cell language [75] can also serve as inspiration.

7.3 Conclusion

The research programme set sail with the ambition of showing that linear algebra makes a compelling alternative to relational algebra as a foundation for query processing. We hoped to make a beautiful theory that also lends itself well to efficiency in practice.

The results have certainly exceeded expectations! Not only did we fulfill our initial objectives, we also made essential algorithmic advances (worst-case optimality of algebraic joins) and developed a new surprisingly powerful programming language (Algeo) to better support the emerging paradigm of algebraic programming.

The conclusion is inescapable: the marriage of abstract algebra and programming is a fruitful endeavour and the journey has just begun.

Bibliography

[1] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall, Inc., USA, 1997.

[2] Mikkel Kragh Mathiesen. Infinite-dimensional linear algebra for efficient query processing. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), August 2016.

[3] Fritz Henglein and Mikkel Kragh Mathiesen. Module theory and query processing (extended abstract). In *Proc. Mathematically Structured Functional Programming (MSFP)*, MSFP '20, 2020.

[4] Fritz Henglein, Robin Kaarsgaard, and Mikkel Kragh Mathiesen. The programming of algebra. In *Proc. 9th Workshop on Mathematically Structured Functional Programming (MSFP)*, pages 71–92, Munich, Germany, April 2022. Electronic Proceedings in Theoretical Computer Science (EPTCS).

[5] Martin Elsman, Fritz Henglein, Robin Kaarsgaard, Mikkel Kragh Mathiesen, and Robert Schenck. Combinatory adjoints and differentiation. In *Proc. 9th Workshop on Mathematically Structured Functional Programming (MSFP)*, pages 1–26, Munich, Germany, April 2022. Electronic Proceedings in Theoretical Computer Science (EPTCS).

[6] Fritz Henglein, Robin Kaarsgaard, and Mikkel Kragh Mathiesen. Algeo: An algebraic approach to reversibility. In Claudio Antares Mezzina and Krzysztof Podlaski, editors, *Reversible Computation*, pages 128–145, Cham, 2022. Springer International Publishing.

[7] David HD Warren. An abstract Prolog instruction set. *Technical report*, 1983.

[8] Lionel Vaux. The algebraic lambda calculus. *Mathematical. Structures in Comp. Sci.*, 19(5):1029–1059, oct 2009.

[9] Pablo Arrighi and Gilles Dowek. Linear-algebraic λ-calculus: Higher-order, encodings, and confluence. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications*, RTA '08, page 17–31, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Robin Kaarsgaard. Condition/decision duality and the internal logic of extensive restriction categories. In *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXV)*, volume 347 of *Electronic Notes in Theoretical Computer Science*, pages 179–202. Elsevier, 2019.

[11] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In *The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35, 2015.

[12] Fritz Henglein and Ralf Hinze. Distributive sorting and searching: From generic discrimination to generic tries. In Chung-chieh Shan, editor, *Proc. 11th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 8301 of *Lecture Notes in Computer Science (LNCS)*, pages 315–332. Springer, December 2013.

[13] Ralf Hinze. Generalizing generalized tries. *J. Funct. Program*, 10(4):327–351, 2000.

[14] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[15] Dan E. Willard. An algorithm for handling many relational calculus queries efficiently. *Journal of Computer and System Sciences*, 65(2):295 – 331, 2002.

[16] Anna Pagh and Rasmus Pagh. Scalable computation of acyclic joins. *Journal of the ACM*, 2006.

[17] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, November 2002.

[18] David Maier, Yehoshua Sagiv, and Mihalis Yannakakis. On the complexity of testing implications of functional and join dependencies. *Journal of the ACM (JACM)*, 28(4):680–695, 1981.

[19] H.Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[20] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[21] Moshe Vardi. The complexity of relational query languages (extended abstract). pages 137–146, 01 1982.

[22] Albert Atserias, Martin Grohe, and Daniel Marx. Size bounds and query plans for relational joins. volume 42, pages 739 – 748, 11 2008.

[23] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)*, 11(1):1–20, 2014.

[24] Hung Q Ngo, Christopher Re, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *arXiv preprint arXiv:1310.3314*, 2013.

[25] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 13–28, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Todd L. Veldhuizen. A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th Int'l Conf. on Database Theory (ICDT)*, pages 96–106, Athens, Greece, March 2014. Openproceedings.org.

[27] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[28] Fritz Henglein and Ken Larsen. Generic multiset programming with discrimination-based joins and symbolic Cartesian products. *Higher-Order and Symbolic Computation (HOSC)*, 23:337–370, 2010. Publication date: November 24, 2011.

[29] Chris Okasaki and Andrew Gill. Fast mergeable integer maps, 1998.

[30] Fritz Henglein. Generic top-down discrimination for sorting and partitioning in linear time. *Journal of Functional Programming*, 22(3):300–374, 2012.

[31] Philip Wadler. Theorems for free! In *Proc. ACM Conf. Functional Programming and Computer Architecture*, pages 347–359, 1989.

[32] James Hoey and Irek Ulidowski. Reversible imperative parallel programs and debugging. In Michael Kirkedal Thomsen and Mathias Soeken, editors, *Reversible Computation*, pages 108–127. Springer, 2019.

[33] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. CauDEr: A causal-consistent reversible debugger for Erlang. In *International Symposium on Functional and Logic Programming (FLOPS 2018)*, pages 247–263. Springer, 2018.

[34] Ulrik Pagh Schultz, Johan Sund Laursen, Lars-Peter Ellekilde, and Holger Bock Axelsen. Towards a domain-specific language for reversible assembly sequences. In Jean Krivine and Jean-Bernard Stefani, editors, *RC 2015*, volume 9138 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2015.

[35] Markus Schordan, David Jefferson, Peter Barnes, Tomas Oppelstrup, and Daniel Quinlan. Reverse code generation for parallel discrete event simulation. In Jean Krivine and Jean-Bernard Stefani, editors, *RC 2015*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2015.

[36] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. From symmetric pattern-matching to quantum control. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2018)*, pages 348–364. Springer, 2018.

[37] Chris Heunen and Robin Kaarsgaard. Quantum information effects. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022.

[38] C. Heunen and R. Kaarsgaard. Bennett and Stinespring, together at last. In *Proceedings 18th International Conference on Quantum Physics and Logic (QPL 2021)*, volume 343 of *Electronic Proceedings in Theoretical Computer Science*, pages 102–118. OPA, 2021.

[39] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Program Manipulation. Proceedings*, pages 144–153. ACM, 2007.

[40] Lasse Hay-Schmidt, Robert Glück, Martin Holm Cservenka, and Tue Haulund. Towards a unified language architecture for reversible object-oriented programming. In *International Conference on Reversible Computation (RC 2021)*, pages 96–106. Springer, 2021.

[41] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In Alexis De Vos and Robert Wille, editors, *Reversible Computation*, pages 14–29. Springer, 2012.

[42] Roshan P James and Amr Sabry. Information effects. *ACM SIGPLAN Notices*, 47(1):73–84, 2012.

[43] Naoki Nishida, Adrián Palacios, and Germán Vidal. A reversible semantics for Erlang. In Manuel V Hermenegildo and Pedro Lopez-Garcia, editors, *Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, pages 259–274. Springer, 2017.

[44] Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[45] Michael Hanus. Functional logic programming: From theory to Curry. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, pages 123–168. Springer, 2013.

[46] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1):17–64, 1996.

[47] Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. CoreFun: A typed functional reversible core language. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation (RC 2018)*, pages 304–321. Springer, 2018.

[48] Roshan P. James and Amr Sabry. Theseus: A high level language for reversible computing. Work-in-progress report, 2014.

[49] ALF: Algebraic logic functional programming language. `https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/impl/fp_lp/alf/0.html`. Accessed 2022-06-30.

[50] Andrej Bauer and Paul Taylor. The Dedekind reals in abstract Stone duality. *Mathematical Structures in Computer Science*, 19(4):757–838, 2009.

[51] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 31–40, New York, NY, USA, June 2007. Association for Computing Machinery.

[52] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *arXiv preprint arXiv:2103.06376*, 2021.

[53] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 153–164, 2011.

[54] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, October 2007.

[55] Daniel Loeb. Sets with a negative number of elements. *Advances in Mathematics*, 91(1):64–74, 1992.

[56] Jacques Carette, Alan P. Sexton, Volker Sorge, and Stephen M. Watt. Symbolic domain decomposition. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *Intelligent Computer Mathematics*, pages 172–188. Springer, 2010.

[57] José Nuno Oliveira and Hugo Daniel Macedo. The data cube as a typed linear algebra operator. In *Proceedings of The 16th International Symposium on Database Programming Languages*, pages 1–11, 2017.

[58] J.N. Oliveira. Towards a linear algebra of programming. *Formal Aspects of Computing*, 24(4):433–458, 2012.

[59] J. Oliveira. Typed linear algebra for weighted (probabilistic) automata. *Implementation and Application of Automata*, pages 52–65, 2012.

[60] H.D. Macedo and J.N. Oliveira. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming*, 2012.

[61] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571. ACM, 2017.

[62] Sven-Bodo Scholz. Single-assignment C — functional programming using imperative style. In *6th International Workshop on Implementation of Functional Languages (IFL'94), Norwich, England, UK*, pages 211–2113. University of East Anglia, Norwich, England, UK, 1994.

[63] Oleg Kiselyov. Reconciling abstraction with high performance: A metaocaml approach. *Foundations and Trends in Programming Languages*, 5(1):1–101, 2018.

[64] Amir Shaikhha and Lionel Parreaux. Finally, a polymorphic linear algebra language (pearl). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[65] Petr Hájek. *Metamathematics of Fuzzy Logic*. Dordrecht, Boston and London: Kluwer Academic Publishers, 1998.

[66] Jacques Carette. Understanding expression simplification. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, page 72–79. ACM, 2004.

[67] Jeremy Gibbons, Fritz Henglein, Ralf Hinze, and Nicolas Wu. Relational algebra by way of adjunctions. *Proceedings of the ACM on Programming Languages: International Conference onf Functional Programming (ICFP)*, 2(ICFP):86, September 2018.

[68] Nevin Lianwen Zhang and David Poole. A simple approach to bayesian network computations, 1994.

[69] Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *J. Artif. Int. Res.*, 5(1):301–328, December 1996.

[70] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[71] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. Fdb: A query engine for factorised relational databases. *Proc. VLDB Endow.*, 5(11):1232–1243, July 2012.

[72] Fritz Henglein. Optimizing relational algebra operations using discrimination-based joins and lazy products. In *Proc. ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation*, pages 73–82, New York, NY, USA, January 18-19 2010. ACM. Also DIKU TOPPS D-report no. 611.

[73] Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):2, 2015.

[74] Stefano Gogioso and Fabrizio Genovese. Infinite-dimensional categorical quantum mechanics. In Ross Duncan and Chris Heunen, editors, *Proceedings 13th International Conference on Quantum Physics and Logic (QPL 2016)*, volume 236 of *Electronic Proceedings in Theoretical Computer Science*. OSA, 2016.

[75] The Cell programming language. `https://www.cell-lang.net`. Accessed 2022-06-30.