Algeo: An Algebraic Approach to Reversibility

Fritz Henglein[1], Robin Kaarsgaard[2⋆], and Mikkel Kragh Mathiesen[1⋆⋆]

[1] Department of Computer Science, University of Copenhagen (DIKU)
[2] School of Informatics, University of Edinburgh

Abstract. We present Algeo, a functional logic programming language based on the theory of infinite dimensional modules. Algeo is reversible in the sense that every function has a generalised inverse, an *adjoint*, which can be thought of as an inverse execution of the forward function. In particular, when the given function is invertible, the adjoint is guaranteed to coincide with the inverse.

Algeo generalises "ordinary" forward-backward deterministic reversible programming by permitting relational and probabilistic features. This allows functions to be defined in a multitude of ways, which we summarise by the motto that "all definitions are extensional characterisations; all extensional characterisations are definitions."

We describe the syntax, type system, and the axiomatic semantics of Algeo, and showcase novel features of the language through examples.

1 Introduction

Reversible programming languages have seen a great deal of research in recent years thanks to their applications in surprisingly diverse areas such as debugging [10, 15], robotics [19], discrete event simulation [18] and quantum computing [17, 9, 8]. For this reason, many different styles of reversible programming have been explored, notably imperative [22], object-oriented [6], functional [21, 12, 16], and parallel [16, 10].

In this paper, we study reversibility in the context of functional logic programming (see, e.g., [1]). As the name suggests, functional logic programming incorporates aspects associated with both functional programming (e.g., pattern matching, strong typing discipline) and logic programming (e.g., nondeterminism, search), making it capable for tasks such as satisfiability modulo theories, querying, and more. The combination of (very liberal) pattern matching with search means that functions can be defined in an indirect way, which makes certain functions expressible in particularly pleasant and succinct ways.

We present Algeo, a programming language based on the linear algebraic theory of modules. Algeo extends the functional logic paradigm with a notion of reversibility in the form of (Hermitian) *adjoints*, a kind of generalised inverse.

⋆ Supported by DFF–International Postdoctoral Grant 0131-00025B.
⋆⋆ Corresponding author. Supported by DFF Research Grant 8022-00415B.

```
path : ⟨Atom⟩ → ⟨Atom ⊗ Atom⟩ → Atom → Atom → ⟨Atom⟩

path v ∗ p p ⩲ p ⋉ !v                    Start and end agree and are in the vertex set
path v e p q ⩲ (                          The general case
    [r : Atom] [s : Atom]                 Introduce existential variables
    r ⩲ !v;                               Let r be a non-deterministically chosen vertex
    p ⊗ r ⩲ !e;                           Check that there is an edge from p to r
    s ⩲ !(path ⟨!v ⋉ p^⊥⟩ e r q);          Find a path recursively that does not contain p
    (p ∥ s))                              Return either p or the vertex found recursively
```

Fig. 1. Graph search in Algeo. A graph can be represented by a bag of vertices, $\mathbf{v} : \langle \mathbf{Atom} \rangle$, and a bag of edges, $\mathbf{e} : \langle \mathbf{Atom} \otimes \mathbf{Atom} \rangle$, and given a start vertex $\mathbf{p} : \langle \mathbf{Atom} \rangle$ and end vertex $\mathbf{q} : \langle \mathbf{Atom} \rangle$, **path v e p q** nondeterministically returns a vertex in a path from **p** to **q**, further weighted by its number of occurences along any path from **p** to **q**.

These adjoints exist not only for programs which are forward and backward deterministic, but also for arbitary linear maps, which may exhibit nondeterministic behaviours (e.g., relational, probabilistic). Crucially, however, when applied to programs which *are* forward and backward deterministic, the adjoint is guaranteed to coincide with the inverse program.

A unique feature of Algeo is that a value comes equipped with a *multiplicity*. This multiplicity can be taken from any ring, which in turn determines the meaning of this multiplicity. For example, real multiplicities, when restricted to those in the closed unit interval $[0, 1]$, can represent *probabilistic* or *fuzzy* membership. Integral multiplicities, when restricted to nonnegative numbers, can represent *multiset* membership; negative multiplicities provide additive inverse operations, e.g. deleting a row in a database table such that adding it again yields the original table whether or not the row was present to start with. Multiplicities can further be used to give a smooth account of *negation* in logic programming. To properly account for these multiplicities, all functions in Algeo are linear by definition in that they preserve multiplicities. However, since not all useful functions are linear, a form of explicit nonlinearity is also supported, requiring the explicit use of *bags* via bagging $\langle - \rangle$ and extraction $!(-)$ operations. An example of an Algeo program for searching for paths between given vertices in a graph is shown in Figure 1.

Paper outline. Section 2 contains a brief tutorial of the language. We present the syntax and type system of Algeo in Section 3, give it an axiomatic semantics in the form of a system of equations, and illustrate its use through examples. In Section 4, we detail some applications in the use of fixed points, and give an encoding of *polysets* (i.e., sets with integral and possibly negative multiplicities) in Algeo. We describe related work in Section 5, and end with some concluding remarks and directions for future research in Section 6.

2 Algeo by Example

We now give an intuition for Algeo by writing some simple programs. In Algeo **Scalar** and $\oplus$ play the role of unit type and sum type respectively. Take **Bool** to be an alias for $\mathbf{Scalar} \oplus \mathbf{Scalar}$, and define:

$$\begin{aligned} &\mathbf{true} : \mathbf{Bool} && \mathbf{false} : \mathbf{Bool} \\ &\mathbf{true} \coloneqq \mathbf{inl}(*) && \mathbf{false} \coloneqq \mathbf{inr}(*) \end{aligned}$$

Negation of booleans can be written using two clauses.

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

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

$$\begin{aligned} \mathbf{not}^\dagger &: \mathbf{Bool} \to \mathbf{Bool} \\ [\mathbf{x} : \mathbf{Bool}]\, \mathbf{not}^\dagger\ (\mathbf{not}\ \mathbf{x}) &\coloneqq \mathbf{x} \end{aligned}$$

This definition quantifies over $\mathbf{x} : \mathbf{Bool}$. 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

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

where $\parallel$ represents binary nondeterministic choice. By the definition of **not** this reduces to

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

which is equivalent to having two clauses:

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

We can thus establish that $\mathbf{not}^\dagger = \mathbf{not}$, as expected. Note the difference between $=$ and $\coloneqq$. The former is a relation between expressions and the latter behaves as an operator with type $\tau \to \tau \to \mathbf{Scalar}$. A definition of a name x is given by an expression of type **Scalar** which may refer to x. There is no requirement to use $\coloneqq$. In fact **not** could equivalently have been defined as:

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

Next, we define conjunction and disjunction:

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

Let us take the adjoint of **and** with respect to the first argument:

$$\mathbf{and}_1^\dagger : \mathbf{Bool} \to \mathbf{Bool}$$
$$[\mathbf{x} : \mathbf{Bool}]\,\mathbf{and}_1^\dagger(\mathbf{and}\ \mathbf{x}\ *) \equiv\!\!\equiv \mathbf{x}$$

Applying this function we get $\mathbf{and}_1^\dagger\ \mathbf{true} = \mathbf{true}$ and $\mathbf{and}_1^\dagger\ \mathbf{false} = \mathbf{true} \parallel \mathbf{false} \parallel \mathbf{false}$. When the first argument of **and** and its result are both **false** there are two different possibilities for the value of the second argument, so **false** is listed twice.

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 multiplicities. For instance, we have $\mathbf{false} \parallel (\overline{-1}; \mathbf{false}) = \emptyset$ (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:

$\mathbf{and} * * \equiv\!\!\equiv \mathbf{false}$	Conjunction 'usually' returns **false**
$\mathbf{and}\ \mathbf{true}\ \mathbf{true} \equiv\!\!\equiv (\overline{-1}; \mathbf{false})$	Not when both arguments are **true**, though
$\mathbf{and}\ \mathbf{true}\ \mathbf{true} \equiv\!\!\equiv \mathbf{true}$	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 $\mathbf{f}\ (\mathbf{true} \parallel \mathbf{false}) = \mathbf{f}\ \mathbf{true} \parallel \mathbf{f}\ \mathbf{false}$. Now suppose that the definition is

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

It is clear that $\mathbf{f}\ \mathbf{true} = \mathbf{f}\ \mathbf{false} = \mathbf{false}$ and therefore $\mathbf{f}\ (\mathbf{true} \parallel \mathbf{false}) = \overline{2}; \mathbf{false}$. 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.

For cases where this behaviour is undesirable Algeo supports bag types, written $\langle\tau\rangle$ and pronounced 'bag of τ'. Bags are formed by writing an expression in angle brackets, e.g., $\langle\mathbf{true} \parallel \mathbf{false}\rangle$. The bag constructor is explicitly not linear, so $\langle\mathbf{true} \parallel \mathbf{false}\rangle \neq \langle\mathbf{true}\rangle \parallel \langle\mathbf{false}\rangle$. The contents of a bag can be extracted with the $!(-)$ operator.

Consider a version of **f** using bags:

$$\mathbf{g} : \langle\mathbf{Bool}\rangle \to \mathbf{Bool}$$
$$\mathbf{g}\ \mathbf{x} \equiv\!\!\equiv \mathbf{and}\ !\mathbf{x}\ (\mathbf{not}\ !\mathbf{x})$$

We have $\mathbf{g}\ \langle\mathbf{true}\rangle = \mathbf{f}\ \mathbf{true}$ and similarly for **false**. However:

$$\begin{aligned}
&\mathbf{g}\ \langle\mathbf{true} \parallel \mathbf{false}\rangle \\
&= \mathbf{and}\ (\mathbf{true} \parallel \mathbf{false})\ (\mathbf{not}\ (\mathbf{true} \parallel \mathbf{false})) \\
&= \mathbf{and}\ \mathbf{true}\ (\mathbf{not}\ \mathbf{true}) \parallel \mathbf{and}\ \mathbf{true}\ (\mathbf{not}\ \mathbf{false}) \parallel \mathbf{and}\ \mathbf{false}\ (\mathbf{not}\ \mathbf{true}) \parallel \mathbf{and}\ \mathbf{false}\ (\mathbf{not}\ \mathbf{false}) \\
&= \mathbf{false} \parallel \mathbf{true} \parallel \mathbf{false} \parallel \mathbf{false} = \mathbf{true} \parallel \overline{3}; \mathbf{false}
\end{aligned}$$

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

Fig. 2. Syntax of types and terms.

Essentially, the nondeterminism is postponed until $!(-)$ is applied. For **g** this means that the two uses of **!x** are not entangled and the result now has the possibility of being **true**. Adjoints also exist when bags are involved, but can be slightly more complicated. For instance, applying the adjoint of **g** to **true** we get

$$[\mathbf{a} : \langle \mathbf{Scalar} \rangle]\, [\mathbf{b} : \langle \mathbf{Scalar} \rangle]\, !\mathbf{a}; !\mathbf{b}; \langle !\mathbf{a}; \mathbf{true} \parallel !\mathbf{b}; \mathbf{false} \rangle$$

which is the totality of all bags containing **a** copies of **true** and **b** copies of **false** scaled by the product of their multiplicities.

3 Syntax and Semantics

The syntax of types and terms are given in Figure 2. 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 3.2. Most operations, exceptions being $\diamond$ and $\langle\cdot\rangle$, are *linear* in the sense that they respect failure ($\emptyset$), alternatives ($\parallel$) and conjunction ($;$). For instance, $\bowtie$ is linear in each component so in particular $\emptyset \bowtie e = \emptyset$, $(e_1 \parallel e_2) \bowtie e_3 = (e_1 \bowtie e_3) \parallel (e_2 \bowtie e_3)$ 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.
- $\mathbf{inl}(e)$ and $\mathbf{inr}(e)$ are left and right injections for the $\oplus$ type.
- $\langle \tau \rangle$ is the type of bags of τ.
- $\langle e \rangle$ is a *bag*. It collects all the results from e into a single bag. The bag itself is considered a base value.
- $\diamond e$ is an *indicator*. It yields $\overline{0}$ if $e = \emptyset$, otherwise $\overline{1}$. Thus, $\diamond$ is explicitly nonlinear.
- $e_1 \parallel e_2$ 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.
- $!e$ is *extraction*. It extracts the contents of a bag.
- $\overline{n}$ 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:

$e_1 \dddot{=} e_2 = e_1 \bowtie e_2; \overline{1}$	Pointwise unification of e_1 and e_2
$e_1 \setminus\!\setminus e_2 = e_1 \parallel \overline{-1}; e_2$	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
$e_1 \oplus e_2 = \mathbf{inl}(e_1) \parallel \mathbf{inr}(e_2)$	A sum of lefts and rights

Beware: some constructs, e.g. $\dddot{=}$, 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 $\mathbf{inl}(e)$ means roughly what one would expect, whereas the unfamiliar syntax like $\dddot{=}$ 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 (possibly recursive) 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 $x : \tau, \widehat{x} : \langle \tau \rangle \vdash e_i : \mathbf{Scalar}$. Intuitively, x refers to (a single component of) the object we are defining and $\widehat{x}$ refers to the completed definition. The completed definition is used for recursive invocations. Note that $\widehat{x}$ is just a name with no a priori relation to x.

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

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

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

$\mathbf{linv} : \langle \tau_1 \to \tau_2 \rangle \to \langle \tau_2 \to \tau_1 \rangle$
$\langle !(\mathbf{linv}\ \mathbf{f}) \circ !\mathbf{f} \rangle \Doteq \langle \mathbf{id} \rangle$

$\mathbf{rinv} : \langle \tau_1 \to \tau_2 \rangle \to \langle \tau_2 \to \tau_1 \rangle$
$\langle !\mathbf{f} \circ !(\mathbf{rinv}\ \mathbf{f}) \rangle \Doteq \langle \mathbf{id} \rangle$

$\mathbf{inv} : \langle \tau_1 \to \tau_2 \rangle \to \langle \tau_2 \to \tau_1 \rangle$
$\mathbf{inv}\ \mathbf{f} \Doteq \mathbf{linv}\ \mathbf{f} \bowtie \mathbf{rinv}\ \mathbf{f}$

Fig. 3. Some basic functions in Algeo: identity, composition, adjoints, and (left and right) inverses. Note the use of bags $\langle - \rangle$ to contain nonlinearity.

Given such a top-level definition and a program e that can refer to x the desugared version is:

$$[\widehat{x} : \langle \tau \rangle]\, \widehat{x} \Doteq \langle [x : \tau]\, (e_1 \parallel \cdots \parallel e_n); x \rangle; e^{x := !\widehat{x}}$$

The notation $e^{x:=!\widehat{x}}$ means e with $!\widehat{x}$ substituted for x. This construction works by summing over all basis elements of τ subject to the conditions imposed by $e_1 \parallel \cdots \parallel e_n$. The sum is collected into the bag $\widehat{x}$, which represents the totality of the object we are defining. Each use of x in the program is replaced with $!\widehat{x}$ so each copy is independent. This scheme generalises to mutual recursion. Note that $\Doteq$ 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 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 **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 $(x \mapsto y)^\dagger \circ (x \mapsto y) \Doteq (x \mapsto x)$ or even the more familiar $(x \mapsto y)^\dagger \Doteq (y \mapsto x)$.

The use of search and indirect definition is perhaps more powerfully illustrated by **linv** (and, symmetrically, **rinv**) which says that a left inverse to a function f is anything that behaves like it; in other words, any function that, after composition with $!f$ (needed here since f is used more than once), yields the identity (and symmetrically for **rinv**). Even further, **inv** states that a full inverse to f is anything that behaves *both* as a left inverse *and* as a right inverse, using the join operator to intersect the left inverses with the right inverses.

A function **f** is *unitary* if the adjoint is also a two-sided inverse module bagging, i.e. if $\langle \mathbf{f}^\dagger \rangle = \mathbf{inv}\ \langle \mathbf{f} \rangle$. The unitaries include include many interesting examples, including all classically reversible functions as well as all quantum circuits. In these cases we prefer the adjoint, since it does not require the use of bags and is generally easier to work with.

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

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

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \mapsto e_2 : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 ; e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash [x : \tau']\, e : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \diamond e : \mathbf{Scalar}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle : \langle \tau \rangle} \qquad \frac{\Gamma \vdash e : \langle \tau \rangle}{\Gamma \vdash !e : \tau}$$

Fig. 4. The type system of Algeo.

3.1 Type System

The type system is seen in Figure 4 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.

3.2 Axiomatic Semantics

We now present the semantics of Algeo as a set of equations between expressions, see Figure 5. 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:

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

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

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

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
$!\langle e \rangle = e$
$\langle !x \rangle = x$

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

Possibility
$\diamond\emptyset = \overline{0}$
$\diamond(e_1 \parallel e_2) = \diamond e_1 \parallel \diamond e_2 \setminus\!\setminus \diamond e_1; \diamond e_2$ when $e_1 \bowtie e_2 = \emptyset$
$\diamond(e_1; e_2) = \diamond e_1; \diamond e_2$
$\diamond x = \overline{1}$
$\diamond a = \overline{1}$
$\diamond\overline{n} = \overline{1}$ when $n \neq \emptyset$
$\diamond\langle e \rangle = \overline{1}$
$\diamond\mathbf{inl}(e) = \diamond e$
$\diamond\mathbf{inr}(e) = \diamond e$
$\diamond(e_1 \otimes e_2) = \diamond e_1; \diamond e_2$
$\diamond(e_1 \mapsto e_2) = \diamond e_1; \diamond e_2$
$\diamond(\diamond e) = \diamond e$

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

Fig. 5. Axiomatic semantics of Algeo.

Most operations are defined to be either *linear* or *bilinear* (with the notable exceptions of $\diamond$ and $\langle - \rangle$). 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 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. For example, $[x : \langle \mathbf{Scalar} \rangle]\,!x$ has type **Scalar** but cannot be shown to be equal to any expression of the form $\overline{n}$; indeed, we cannot even establish whether it is zero or nonzero. 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 $\tau = \mathbf{Scalar} \oplus \cdots \oplus \mathbf{Scalar}$ (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 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:

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

The possibility operator $\diamond e$ also deserves elaboration. The intuitive description (evaluate e and return $\overline{1}$ if it is not $\emptyset$) sounds similar to negation by failure (evaluate e and return $\overline{1}$ if it *is* $\emptyset$). However, $\diamond e$ is not defined operationally. It has rules for each data constructor as well as $\emptyset$, $\parallel$ and $(;)$, but it does not by itself make progress on e. For example, a subexpression like $\diamond(x \eqqcolon y)$ does not simply succeed with a 'unification' of x and y. Rather, the case-split rule (discussed above) should be applied where either x or y is bound, making a global nondeterministic choice on whether x and y are equal.

The combination $(\diamond e)^{\perp}$ expresses the Algeo version of negation by failure. Compared to the usual notion in logic programming it is pure and not dependent on the evaluation strategy.

Finally, we mention $\langle e_1 \rangle \bowtie \langle e_2 \rangle = (\diamond(e_1 \setminus\!\setminus e_2))^{\perp}; \langle e_1 \rangle$ which describes how to resolve joins of bags. When comparing bags $\langle e_1 \rangle$ and $\langle e_2 \rangle$ we have to decide whether e_1 and e_2 are equal as Algeo expressions. That is the case precisely when $e_1 \setminus\!\setminus e_2 = \emptyset$. If $e_1 = e_2$ then $(\diamond(e_1 \setminus\!\setminus e_2))^{\perp} = \overline{1}$ and the whole right-hand side reduces to $\langle e_1 \rangle$ as expected. If $e_1 \neq e_2$ then the condition fails and the right-hand side reduces to $\emptyset$, again as expected.

This rule is the sole reason why $\diamond$ has to be in the language. The possibility operator could otherwise simply be defined as $\diamond e = (\langle e \rangle \eqqcolon \langle \emptyset \rangle)^{\perp}$, but then the rule for bag joins would not actually make any progress.

3.4 Derived Equations and Evaluation

The semantic equations in Figure 5 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 $(;)$.

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

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

As an example of evaluation consider the problem of calculating how many pairs of atoms are equal and how many are unequal. Equality of x of y can be reified by putting it in a bag, i.e. $\langle x \,\dddot{=}\, y \rangle$. The question can then be answered as follows, assuming the standard model where $\mathbf{dim}(\mathbf{Atom}) = \omega$.

$$\begin{aligned}
&[x : \mathbf{Atom}]\, [y : \mathbf{Atom}]\, \langle x \,\dddot{=}\, y \rangle \\
&= [x : \mathbf{Atom}]\, \langle x \,\dddot{=}\, y \rangle^{y:=x} \setminus\!\setminus \langle x \,\dddot{=}\, y \rangle^{y:=\emptyset} \parallel ([z : \mathbf{Atom}]\, \langle x \,\dddot{=}\, y \rangle^{y:=z \bowtie x^{\perp}}) \\
&= [x : \mathbf{Atom}]\, \langle x \,\dddot{=}\, x \rangle \setminus\!\setminus \langle x \,\dddot{=}\, \emptyset \rangle \parallel ([z : \mathbf{Atom}]\, \langle x \,\dddot{=}\, z \bowtie x^{\perp} \rangle) \\
&= [x : \mathbf{Atom}]\, \langle \overline{1} \rangle \setminus\!\setminus \langle \overline{0} \rangle \parallel ([z : \mathbf{Atom}]\, \langle \overline{0} \rangle) \\
&= [x : \mathbf{Atom}]\, \langle \overline{1} \rangle \setminus\!\setminus \langle \overline{0} \rangle \parallel \overline{\mathbf{dim}(\mathbf{Atom})}; \langle \overline{0} \rangle \\
&= ([x : \mathbf{Atom}]\, \langle \overline{1} \rangle) \setminus\!\setminus ([x : \mathbf{Atom}]\, \langle \overline{0} \rangle) \parallel ([x : \mathbf{Atom}]\, \overline{\mathbf{dim}(\mathbf{Atom})}; \langle \overline{0} \rangle) \\
&= \overline{\mathbf{dim}(\mathbf{Atom})}; \langle \overline{1} \rangle \setminus\!\setminus \overline{\mathbf{dim}(\mathbf{Atom})}; \langle \overline{0} \rangle \parallel \overline{\mathbf{dim}(\mathbf{Atom}) \cdot \mathbf{dim}(\mathbf{Atom})}; \langle \overline{0} \rangle \\
&= \overline{\omega}; \langle \overline{1} \rangle \parallel \overline{\omega^2 - \omega}; \langle \overline{0} \rangle
\end{aligned}$$

Thus, we see that there are ω pairs of equal atoms and $\omega^2 - \omega$ unequal pairs. This corresponds well with the size of the diagonal and off-diagonal respectively of a hypothetical $\omega \times \omega$ matrix.

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 Figure 6. 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

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

Fig. 6. Linear algebra versus Algeo

write it as $[\mathbf{x}]\,[\mathbf{y}]\,(\mathbf{f}\ \mathbf{x} \coloneqq \mathbf{y}) \coloneqq (\mathbf{x} \coloneqq \mathbf{f}^{\dagger}\ \mathbf{y})$, 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.

4 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 $\coloneqq$. 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 are written just like in Curry. For example, suppose $\mathbf{f}$ is defined by $\mathbf{f}\ (\mathbf{g}\ x) \coloneqq e$. 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:

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

  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 $\|$. By linearity $\mathbf{f}\ (e_1 \parallel e_2) \coloneqq e$ means exactly the same as $\mathbf{f}\ e_1 \coloneqq e \parallel \mathbf{f}\ e_2 \coloneqq e$.
- 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 $e^\perp = * \setminus\!\setminus e$ 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:

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

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

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

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.

A much more involved example is (unnormalised) operator diagonalisation, i.e., the representation of an operator F as a composition $F = U^{-1} \circ D \circ U$, where U is an isomorphism, and D is diagonal. We specify this by describing the constraints: U must be an isomorphism, so $U^{-1} \circ U = \mathrm{id}$ and $U \circ U^{-1} = \mathrm{id}$; D must be diagonal, meaning that joining it with the identity should have no effect; and the composition of $U^{-1} \circ D \circ U$ should be F. In Algeo, this becomes

$$\begin{aligned}
&\mathbf{diagonalise} : \langle \tau \to \tau \rangle \to \langle \tau \to \tau \rangle \otimes \langle \tau \to \tau \rangle \otimes \langle \tau \to \tau \rangle \\
&\mathbf{diagonalise}\ \mathbf{f} \coloneqq \mathbf{u} \otimes \mathbf{d} \otimes \mathbf{v}; \\
&\quad \langle !\mathbf{u} \circ !\mathbf{v} \rangle \coloneqq \langle \mathbf{id} \rangle; \\
&\quad \langle !\mathbf{v} \circ !\mathbf{u} \rangle \coloneqq \langle \mathbf{id} \rangle; \\
&\quad \langle !\mathbf{d} \bowtie \mathbf{id} \rangle \coloneqq \mathbf{d}; \\
&\quad \langle !\mathbf{u} \circ !\mathbf{d} \circ !\mathbf{v} \rangle \coloneqq \mathbf{f}
\end{aligned}$$

Notice that this is only slightly different than usual diagonalisation, in that this will find diagonalisations for eigenvectors scaled arbitrarily, rather than (as usual) only of length 1.

Polysets and polylogic. Polysets [7] 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 [14]). 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, $\bot$ has no evidence and a single counterexample, and dually for $\top$, as in

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

As in [14], 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):

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

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 $(d_1 \oplus {d_1}^\bot) \vee \cdots \vee (d_k \oplus {d_k}^\bot)$. Note that in this calculus $\vee$ and $\wedge$ form a lattice-esque structure as opposed to $\parallel$ and $\bowtie$, which form a ring structure.

5 Related Work

Algebraic λ-calculi. A related approach to computing with linear algebra is the idea of extending the λ-calculus with linear combinations of terms [**?**,**?**], 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 [**?**] 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 $\mathbf{fix}\ (\mathbf{x} \mapsto \mathbf{x}) \coloneqq \mathbf{x}$, which is just a repackaged version of $\bowtie$. To get something approaching the usual concept of fixpoint we again need bags:

$$\mathbf{fix} : \langle \tau \to \tau \rangle \to \tau \qquad\qquad \mathbf{fix}\ \mathbf{f} \coloneqq !\mathbf{f}\ (!\widehat{\mathbf{fix}}\ \mathbf{f})$$

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 a 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. The functional logic paradigm of programming was pioneered by languages such as Curry [1, 5] and Mercury [20]. Along with reversible functional programming languages such as Rfun [21], CoreFun [11], and Theseus [13], they have served as inspiration for the design of Algeo. 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 [7].

Abstract Stone duality. Abstract Stone duality (see, e.g., [2]) 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.

6 Conclusion and Future Work

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 4), 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 [4]) which could be interesting to elaborate. In fact, permitting the use of nonstandard real (or complex) numbers as multiplicities would allow automatic differentiation (see [3] 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.

References

1. Antoy, S., Hanus, M.: Functional logic programming. Communications of the ACM **53**(4), 74–85 (2010)
2. Bauer, A., Taylor, P.: The Dedekind reals in abstract Stone duality. Mathematical Structures in Computer Science **19**(4), 757–838 (2009)
3. Elsman, M., Henglein, F., Kaarsgaard, R., Mathiesen, M.K., Schenck, R.: Combinatory adjoints and differentiation (2022), accepted for *Ninth Workshop on Mathematically Structured Functional Programming (MSFP 2022)*, to appear
4. Gogioso, S., Genovese, F.: Infinite-dimensional categorical quantum mechanics. In: Duncan, R., Heunen, C. (eds.) Proceedings 13th International Conference on Quantum Physics and Logic (QPL 2016). Electronic Proceedings in Theoretical Computer Science, vol. 236. OSA (2016)
5. Hanus, M.: Functional logic programming: From theory to Curry. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics: Essays in Memory of Harald Ganzinger. pp. 123–168. Springer (2013)

6. Hay-Schmidt, L., Glück, R., Cservenka, M.H., Haulund, T.: Towards a unified language architecture for reversible object-oriented programming. In: International Conference on Reversible Computation (RC 2021). pp. 96–106. Springer (2021)
7. Henglein, F., Kaarsgaard, R., Mathiesen, M.K.: The programming of algebra (2022), accepted for *Ninth Workshop on Mathematically Structured Functional Programming (MSFP 2022)*, to appear
8. Heunen, C., Kaarsgaard, R.: Bennett and Stinespring, together at last. In: Proceedings 18th International Conference on Quantum Physics and Logic (QPL 2021). Electronic Proceedings in Theoretical Computer Science, vol. 343, pp. 102–118. OPA (2021)
9. Heunen, C., Kaarsgaard, R.: Quantum information effects. Proceedings of the ACM on Programming Languages **6**(POPL) (2022)
10. Hoey, J., Ulidowski, I.: Reversible imperative parallel programs and debugging. In: Thomsen, M.K., Soeken, M. (eds.) Reversible Computation. pp. 108–127. Springer (2019)
11. Jacobsen, P.A.H., Kaarsgaard, R., Thomsen, M.K.: CoreFun: A typed functional reversible core language. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation (RC 2018). pp. 304–321. Springer (2018)
12. James, R.P., Sabry, A.: Information effects. ACM SIGPLAN Notices **47**(1), 73–84 (2012)
13. James, R.P., Sabry, A.: Theseus: A high level language for reversible computing (2014), https://www.cs.indiana.edu/~sabry/papers/theseus.pdf, work-in-progress report
14. Kaarsgaard, R.: 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). Electronic Notes in Theoretical Computer Science, vol. 347, pp. 179–202. Elsevier (2019)
15. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: A causal-consistent reversible debugger for Erlang. In: International Symposium on Functional and Logic Programming (FLOPS 2018). pp. 247–263. Springer (2018)
16. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) Logic-Based Program Synthesis and Transformation (LOPSTR 2016). pp. 259–274. Springer (2017)
17. Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control. In: International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2018). pp. 348–364. Springer (2018)
18. Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.B. (eds.) RC 2015. Lecture Notes in Computer Science, vol. 9138, pp. 95–110. Springer (2015)
19. Schultz, U.P., Laursen, J.S., Ellekilde, L., Axelsen, H.B.: Towards a domain-specific language for reversible assembly sequences. In: Krivine, J., Stefani, J.B. (eds.) RC 2015. Lecture Notes in Computer Science, vol. 9138, pp. 111–126. Springer (2015)
20. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. The Journal of Logic Programming **29**(1), 17–64 (1996)
21. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) Reversible Computation. pp. 14–29. Springer (2012)
22. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation. Proceedings. pp. 144–153. ACM (2007)