Data Provenance as Automatic Differentiation

ROBERT ATKEY, University of Strathclyde, UK
ROLY PERERA*, University of Cambridge, UK

Automatic differentiation (AD) computes the derivative of a program alongside the program itself, as a linear map between tangent spaces, propagated forwards or backwards along an execution. We present a semantic framework that models *data provenance* via the same construction: taking scalars from a commutative semiring of dependency information rather than the real numbers, the derivative of a program becomes a linear map between spaces of approximations of its input and output. The choice of semiring determines the notion of provenance. Over the two-element Boolean algebra, the Jacobian of a program records which input positions each output position may depend on, and composing Jacobians forwards or backwards is *dependency analysis* in the manner of forward- and reverse-mode AD. More generally, over distributive lattices the Jacobian and its transpose propagate dependency information forwards and backwards as a *conjugate pair* of maps; when the lattice is a Boolean algebra, the two directions are moreover related by adjunction, recovering an approach called *Galois slicing*. We interpret a higher-order total functional language in this framework, prove that every program of first-order type denotes such a Jacobian, and instantiate the semiring to obtain dependency tracking (Booleans), automatic differentiation (reals), and quantitative interval provenance (the tropical semiring) as examples. All results are formalised in Agda.

1 Introduction

To audit any computational process, we need robust and well-founded notions of *provenance* to track how data are used. This allows us to answer questions like "Where did these data come from?", "Why are these data in the output?" and "How were these data computed?". Provenance tracking has a wide range of applications, from debugging and program comprehension [Buneman et al. 1995; Cheney et al. 2007] to improving reproducibility and transparency in scientific workflows [Kontogiannis 2008]. *Program slicing*, first proposed by Weiser [1981], is a collection of techniques for provenance tracking that attempts to take a run of a program and areas of interest in the output, and turn them into the subset of the input and the program that were responsible for generating those specific outputs.

Underlying all of these questions is a more basic one: how does the output of a program respond when its input changes? For programs that compute with real numbers, calculus provides a canonical answer: the derivative, a linear map between tangent spaces describing how the output varies as the input varies, with *automatic differentiation* (AD) being a computational technique for computing such derivatives alongside the program itself [Elliott 2018; Siskind and Pearlmutter 2008; Vákár and Smeding 2022]. For data provenance and dependency analysis, where the inputs are database tuples or elements of a data structure, the question is often framed in a more qualitative way: if we perturb the input at a given position, can the output change at all? Posing this question one input at a time yields a vector of Boolean partial "derivatives", and for a program with several outputs, a *Boolean Jacobian*: a matrix whose entry at (j, i) records whether output position j may depend on input position i. Consider multiplication: at the point (x, y), the usual partial derivative $\partial(x \cdot y)/\partial x$ is y, and its Boolean counterpart records that the output can vary with x only when $y \neq 0$. Like their numerical counterparts, Boolean Jacobians compose by matrix multiplication, with conjunction and disjunction playing the role of multiplication and addition. This paper presents an approach to dynamic dependency analysis where such Jacobians are evaluated alongside the

*Also with University of Bristol.

Authors' Contact Information: Robert Atkey, robert.atkey@strath.ac.uk, University of Strathclyde, Glasgow, UK; Roly Perera, roly.perera@cl.cam.ac.uk, University of Cambridge, Cambridge, UK.

program, forwards or backwards, in the manner of forward- and reverse-mode AD, with the two directions related by matrix transposition.

Existing approaches to program slicing are often tied to particular programming languages or implementations. In this paper we develop this analogy into a general categorical approach to data provenance, in which the derivative of a program is a linear map between spaces of approximations of its input and output, with scalars drawn from a commutative semiring of dependency information. The choice of semiring determines the analysis: the reals recover AD itself, while over distributive lattices the derivative and its transpose form a *conjugate pair*, propagating dependency information forwards and backwards; when the lattices are Boolean algebras, the same maps can instead be presented as an adjunction, recovering the Galois slicing of Perera and collaborators. Our categorical approach should provide a suitable setting for enabling "automatic" data provenance for a variety of programming languages, and is easily configured to use alternative approximation strategies, including quantitative forms of slicing.

1.1 Dependencies as Derivatives

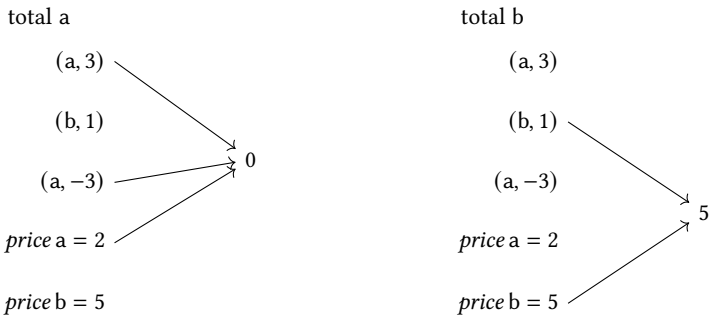
Consider a single run of a program on a particular input. Each part of the output is computed from certain parts of the input, and recording these facts gives the *dependency relation* of the run, relating each output position to the input positions it depends on. This section works through a small example showing how dependency relations behave as derivatives: they have a tabular representation as Boolean matrices, compose using matrix multiplication, and can be evaluated forwards or backwards.

Example 1.1. The following program is written in Haskell syntax [Marlow et al. 2010]. It filters a list of pairs of labels and quantities to those with a given label, and then sums the quantities weighted by a per-label price:

$$\begin{aligned}&\text{total} :: \text{Label} \to [(\text{Label}, \text{Rational})] \to (\text{Label} \to \text{Rational}) \to \text{Rational}\\&\text{total}\ l\ db\ price = \text{sum}\ [price\ l \cdot q \mid (l', q) \leftarrow db, l \equiv l']\end{aligned}$$

With $db = [(\text{a}, 3), (\text{b}, 1), (\text{a}, -3)]$, where the third entry is a refund of the first, and $price\ \text{a} = 2$ and $price\ \text{b} = 5$, we have $\text{total a}\ db\ price = 2 \cdot 3 + 2 \cdot (-3) = 0$ and $\text{total b}\ db\ price = 5$.

Suppose we are interested in how the output depends on the numerical parts of the input, for the query parameters $l = \text{a}$ and $l = \text{b}$. The three quantities in db and the two prices occupy five *positions*, and the output a single position. In the run with $l = \text{a}$, the output is computed from the quantities at the first and third positions and the price of a; in the run with $l = \text{b}$, from the second quantity and the price of b. We depict each input position pointing to the output positions that depend on it:



Tabulating each relation, with a row for each output position and a column for each input position (ordered as the three quantities and then the two prices), gives the *Boolean Jacobian* of the run,

shown here beside the numerical Jacobian of differential calculus, taken at the same point:

$$\partial_{\mathbb{2}}(\text{total a } db\,price) = \begin{pmatrix}1 & 0 & 1 & 1 & 0\end{pmatrix} \qquad \partial(\text{total a } db\,price) = \begin{pmatrix}2 & 0 & 2 & 0 & 0\end{pmatrix}$$

Like their numerical counterparts, these matrices compose. Extensionally, we can think of total l as a sum applied to per-row products applied to a selection, and the Jacobian of the run as the product of the Jacobians of those stages: a chain rule for dependencies. For the run with $l = \text{a}$, the Boolean pipeline is

$$\underbrace{\begin{pmatrix}1 & 1\end{pmatrix}}_{\text{sum}} \underbrace{\begin{pmatrix}1 & 0 & 1\\ 0 & 1 & 1\end{pmatrix}}_{\text{multiply}} \underbrace{\begin{pmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\end{pmatrix}}_{\text{select}} = \begin{pmatrix}1 & 0 & 1 & 1 & 0\end{pmatrix}$$

and the numerical pipeline is

$$\underbrace{\begin{pmatrix}1 & 1\end{pmatrix}}_{\text{sum}} \underbrace{\begin{pmatrix}2 & 0 & 3\\ 0 & 2 & -3\end{pmatrix}}_{\text{multiply}} \underbrace{\begin{pmatrix}1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\end{pmatrix}}_{\text{select}} = \begin{pmatrix}2 & 0 & 2 & 0 & 0\end{pmatrix}$$

Reading right to left: selection maps the five input positions to the three inputs used on this run (the two a-quantities and *price* a); the multiply stage computes the two per-row products, each depending on its quantity (with price as coefficient) and on the price (with quantity as coefficient); and the sum merges the two products. The two pipelines differ in the price column of the multiply stage, where the numerical coefficients 3 and -3 cancel through the sum while their Boolean counterparts join as $1 \vee 1 = 1$.

In the composite Jacobians, the quantity entries of ∂ are the relevant price, since the derivative of a product is the other factor, and the corresponding $\partial_{\mathbb{2}}$-entries record which of them are nonzero, so on multiplication the two Jacobians agree. At the entry for *price* a they differ: $\partial\,\text{total a}/\partial\,price\,\text{a} = 3 + (-3) = 0$, since the refund cancels the purchase and the total genuinely does not depend on that price; the Boolean Jacobian records 1, because dependency information combines by disjunction and there is no negative dependency to cancel with. Boolean Jacobians can thus report more nonzero entries than their numerical counterparts, but never fewer; this over-approximation is inherent to tracking binary dependence.

As in differential calculus, the Jacobian is taken at a point; the run with $l = \text{b}$, a different point, has its own pipeline of stage Jacobians, with composites:

$$\partial_{\mathbb{2}}(\text{total b } db\,price) = \begin{pmatrix}0 & 1 & 0 & 0 & 1\end{pmatrix} \qquad \partial(\text{total b } db\,price) = \begin{pmatrix}0 & 5 & 0 & 0 & 1\end{pmatrix}$$

At this point nothing cancels, and the two Jacobians agree on which entries are nonzero.

In setting up the positions, we chose to let only the numbers (quantities and prices) vary, keeping the labels and the structure of the list itself fixed. Other choices are also useful, and one of the aims of this work is to clarify how to choose a dependency structure appropriate for different tasks by selecting an appropriate semiring of dependency information. We elaborate on this further in §2.

As a matrix, the Boolean Jacobian acts on Boolean vectors indexed by input positions, and this action evaluates dependencies *forwards*: a vector selects input positions, and its image under the Jacobian selects the output positions that depend on them. Such a vector can also be displayed in the shape of the data the program consumes, as an *approximation* of the input, keeping the numbers at selected positions and replacing the rest by $\bot$, with the price function shown as the pair $\langle price\,\text{a}, price\,\text{b}\rangle$. Writing $\partial_{\mathbb{2}}(\text{total } l\ db\,price)_f$ for this forward action, we have $\partial_{\mathbb{2}}(\text{total b } db\,price)_f([(\text{a}, \bot), (\text{b}, 1), (\text{a}, \bot)], \langle \bot, \bot\rangle) = 5$, since the output depends on the entry (b, 1),

whereas $\partial_{\mathbb{2}}\,(\text{total a } db\ price)_f$ applied to the same selection returns $\bot$, because that run consulted no b-labelled entries.

The transpose of the Jacobian evaluates dependencies *backwards*: given a selection of output positions, it returns the input positions on which they depend. Writing $\partial_{\mathbb{2}}\,(\text{total } l\ db\ price)_r$ for this reverse action, the fully approximated output $\bot$ depends on none of the input:

$$\partial_{\mathbb{2}}\,(\text{total } l\ db\ price)_r\,(\bot) = ([(\text{a}, \bot), (\text{b}, \bot), (\text{a}, \bot)], \langle \bot, \bot \rangle)$$

for both $l = \text{a}$ and $l = \text{b}$. If instead we keep the output unapproximated, then the two runs' backwards actions return different results:

$$\begin{aligned}\partial_{\mathbb{2}}\,(\text{total a } db\ price)_r\,(0) &= ([\ (\text{a}, 3)\,,\ (\text{b}, \bot),\ (\text{a}, -3)\ ],\ \langle\ 2\,,\ \bot\ \rangle)\\ \partial_{\mathbb{2}}\,(\text{total b } db\ price)_r\,(5) &= ([\ (\text{a}, \bot),\ (\text{b}, 1)\,,\ (\text{a}, \bot)\ ],\ \langle\ \bot,\ 5\ \rangle)\end{aligned}$$

Pieces of the input that were not used are replaced by $\bot$. As we expect, the run with label a depends on the entries in the database labelled with a, and likewise for the run with label b. Note that the backward action for a retains *price* a, even though the numerical derivative there is 0: read backwards, too, the dependency Jacobian cannot see that contributions cancelled. Asking *how much* an output can change as an input varies, rather than whether it can move at all, calls for a more quantitative choice of scalars; we develop this in §2.3.

In a simple query like this, it is easy to work out the dependency relationship between the input and output. However, the benefit of language-based approaches is that they are *automatic* for all programs, no matter how complex the relationship between input and output. Moreover, by changing what we mean by "approximation" we can compute a range of different information about a program.

1.2 Dependency Analysis and Automatic Differentiation

The backward maps of the previous section compute a form of *data provenance*: for each part of the output, the part of the input needed to explain it. Dependency information of this kind is the common currency of a range of existing techniques, from program slicing [Weiser 1981] to provenance for database queries [Buneman et al. 1995; Cheney et al. 2007], including semiring provenance [Green et al. 2007]. Closest to our setting is the Galois slicing of Perera and collaborators [Perera et al. 2012, 2016; Ricciotti et al. 2017], which organises the forward and backward maps of a run into a Galois connection between lattices of approximations.

Many forms of dynamic provenance analysis, Galois slicing included, work by recording a trace of each execution and computing the backwards analysis by re-running over the trace. A denotational account would instead bake the analysis into the semantics of the program itself, rather than provide it as a separately defined "backwards evaluation" operation. The differential reading of dependency analysis, developed in §2, points to a way to obtain such an account, by analogy with *automatic differentiation* [Elliott 2018; Siskind and Pearlmutter 2008; Vákár and Smeding 2022]; let us make the correspondence explicit.

- For dependency analysis, every value has an associated join-semilattice of *approximations*, with least element $\bot$ (the empty selection). For differentiable programs, every point has an associated vector space of *tangents*.
- For dependency analysis, every program has an associated forward approximation map that takes approximations of the input to the approximations of the output that depend on them. This map *preserves joins and* $\bot$. For differentiable programs, every program has a forward derivative that takes tangents of the input to tangents of the output. The forward derivative map is *linear*, so it preserves addition of tangents and the zero tangent.

- For dependency analysis, every program has an associated backward approximation map that takes approximations of the output back to least approximations of the input that they depend on. This map also *preserves joins and* $\bot$. For differentiable programs, every program has a reverse derivative that takes *cotangents* of the output (linear maps from tangents to scalars) to cotangents of the input. This map is again *linear*.
- In both settings, the forward and backward maps are related by being each other's transpose. The transpose makes sense because the spaces are *self-dual*: tangent vectors pair with tangent vectors via the dot product, identifying cotangents with tangents, and selections of positions pair by the same dot product formula, with joins and meets in place of sums and products.

Given this close connection between dependency analysis and differentiable programming, we can take structures intended for modelling automatic differentiation, such as Vákár's CHAD framework and use them to model dependency analysis. This will enable us to generalise and expand the scope of automatic differentiation to act as a foundation for data provenance in a wider range of computational settings.

1.3 Outline and Contributions

Our main contribution is to show that data provenance can be understood as differentiation over a commutative semiring of dependency information. In §2 we develop the first-order picture, for programs over tuples of atomic data: derivatives as matrices over a semiring S, with the transpose relating forward and backward dependency analysis, and with successively stronger conditions on S yielding join-preserving maps, conjugate pairs and, for Boolean algebras, the Galois connections of Galois slicing; ordinary AD occupies the case $S = \mathbb{R}$.

The following table summarises the correspondence developed over the course of the paper:

	Differentiable programming	**Dependency analysis**
scalars	real numbers	commutative semiring S
space at a point	vector space of tangents	semimodule of approximations over S
identification with the dual	finite dimensionality	chosen self-duality
derivative at a point	Jacobian matrix	S-weighted relation as matrix over S
chain rule	matrix multiplication	matrix multiplication over S
forward mode	push tangents forward	propagate dependencies forward
reverse mode	pull cotangents back	propagate dependencies backwards
forward/backward relation	transpose	transpose via self-dualities

In §3 we extend to sum types using the category of families construction and, following Vákár *et al.*'s CHAD framework [Lucatelli Nunes and Vákár 2023; Vákár and Smeding 2022], build models of a higher-order language in which every program is interpreted together with its family of derivatives. We apply this to a concrete higher-order language in §4 and demonstrate the model on a number of examples in §5, showing how parameterising on S controls the dependency structure associated with data, something that was "hard-coded" in previous presentations of Galois slicing. We prove a correctness property in §6: programs of first-order type have equal interpretations in standard semantics and a dependency tracking semantics, even when they use higher-order functions internally. §7 and §8 discuss additional related and future work.

We have formalised our major results in Agda, resulting in an executable implementation built directly from the categorical constructions that we have used to compute the examples in §5. Please consult the file `everything.agda` in the supplementary material.

2 Tangent Spaces as Semimodules

We motivate our approach by showing how to combine ideas from differential geometry and linear algebra to reconstruct the dependency analyses of §1 in a denotational setting.

2.1 Manifolds, Smooth Functions, and Automatic Differentiation

The general study of differentiable functions takes place on *manifolds*, topological spaces that "locally" behave like an open subset of the Euclidean space $\mathbb{R}^n$. The spaces $\mathbb{R}^n$ themselves are manifolds, but so are "non-flat" examples such as n-spheres and yet more exotic spaces. Every point x in a manifold M has an associated *tangent vector space* $\mathrm{T}_x(M)$ consisting of linear approximations of curves on the manifold passing through x. Each point also has a *cotangent vector space* $\mathrm{T}^*_x(M) = \mathrm{T}_x(M) \multimap \mathbb{R}$. The tangent and cotangent spaces are finite dimensional, so in the presence of a chosen basis they are canonically isomorphic. In the case when the manifold is $\mathbb{R}^n$, then every tangent space is isomorphic to $\mathbb{R}^n$ as well.

Smooth functions f between manifolds M and N are functions on their points that are locally differentiable on $\mathbb{R}^n$. Manifolds and smooth functions form a category **Man**. Each smooth function induces maps of the (co)tangent spaces:

- The *forward derivative* (tangent map, pushforward) f_{*x} is a linear map $\mathrm{T}_x(M) \multimap \mathrm{T}_{f(x)}(N)$. In the Euclidean case when $M = \mathbb{R}^m$ and $N = \mathbb{R}^n$, the tangent map can be represented by the Jacobian matrix of partial derivatives of f at x.
- The *backward derivative* (cotangent map, pullback) f^*_x is a linear map $\mathrm{T}^*_{f(x)}(N) \multimap \mathrm{T}^*_x(M)$. In the Euclidean case, the backward derivative is represented by the transpose of the Jacobian of f at x.

Remark 1 (Chain Rule). A useful property of derivative maps is that they compose according to the chain rule. Suppose that $f : M \to N$ and $g : N \to K$ are smooth functions. Then for any $x \in M$, we have:

- $(g \circ f)_{*x} = g_{*f(x)} \circ f_{*x} : \mathrm{T}_x(M) \multimap \mathrm{T}_{g(f(x))}(K)$
- $(g \circ f)^*_x = f^*_x \circ g^*_{f(x)} : \mathrm{T}^*_{g(f(x))}(K) \multimap \mathrm{T}^*_x(M)$

The chain rule has the practical effect that we can compute derivative maps of f and g independently and compose them, instead of the potentially more difficult task of computing the derivative maps of $g \circ f$. As we shall see below, stable maps also obey a chain rule, and this forms the basis of the general categorical approach to differentiability that we describe in §3.

Computing the forward and backward derivatives of smooth functions f has many applications of practical interest. For example, computation of the reverse derivative is of central interest in machine learning by gradient descent, the main technique used to train deep neural networks [Goodfellow et al. 2016; Rumelhart et al. 1988].

Derivatives can be computed numerically by computing f on small perturbations of its input, or symbolically by examining a closed-form representation of f. However, a more common and practical technique is to use *automatic differentiation*, where a program computing f is instrumented to produce (a representation of) the forward and/or backward derivative as a side-effect of producing the output [Linnainmaa 1976]. This has led to the area of differentiable programming, where programming languages and their implementations are specifically designed to admit efficient automatic differentation algorithms [Abadi et al. 2016; Bradbury et al. 2018; Elliott 2017; Sigal 2024].

2.2 Derivatives over a Commutative Semiring

In the Euclidean case, the differential picture above is plain matrix algebra: with a chosen basis, tangent and cotangent spaces are both $\mathbb{R}^n$, the forward derivative at a point is the Jacobian matrix,

and the backward derivative is its transpose. Once the picture is presented in coordinates like this, the real numbers play no further role beyond supplying the scalars; since dependency information is typically indexed by positions in a value, which then serve as a coordinate basis, our first move is to keep the matrix algebra and change the scalars. Thus throughout this section, we fix a commutative semiring $S = (S, +, 0, \cdot, 1)$, whose elements we think of as atoms of dependency information, with $\cdot$ combining information along a path from an input to an output and $+$ combining information across parallel paths.

Our second move is one of abstraction: rather than axiomatise the matrices of §1 directly, we identify the minimal structure required to capture what made that story work: a space of dependency information for each value, linear maps between spaces for the forward analysis, and an identification of each space with its dual, the extra structure needed to support an analogue of matrix transposition, and hence a backward analysis. The role of vector spaces, which are defined over fields, is played by *semimodules*: commutative monoids $(M, +, 0)$ equipped with an action $a \cdot x$ of the scalars, satisfying the usual laws. Linear maps preserve $+$, 0 and the action, and form a category $\mathbf{SemiMod}(S)$; pointwise addition of linear maps makes each hom-set a commutative monoid, with composition bilinear. Every semimodule has a *dual* $M^* = M \multimap S$ of linear maps into the scalars (categorically, the internal hom into the scalars object, which is the unit of the tensor product of semimodules), and every linear map $f : M \multimap N$ has a *transpose* $f^\top = (- \circ f) : N^* \multimap M^*$, contravariantly.

The transpose is almost the backward analysis we are after: it runs in the opposite direction to f, but between the duals N^* and M^*, the cotangent spaces, rather than between the tangent spaces M and N themselves. What we want is a map $N \multimap M$, running dependency information back through the same spaces the forward map runs it through, as the transposed matrix did in §1. For that, each space must be identified with its dual. Whereas a finite-dimensional vector space always admits such an identification (choose a basis), a semimodule need not; we therefore make the identification part of the data.

Definition 2.1 (Self-dual semimodule). A *self-dual* semimodule is a pair $\langle M, i \rangle$ of a semimodule and a chosen isomorphism $i : M \cong M^*$. The *pairing* induced by i is $\langle x, y \rangle = i(x)(y)$.

The pairing is an S-valued measure of the extent to which x and y overlap; its prototype is the dot product of vectors (the pairing induced by the self-duality of S^n, as we will see below). Extending the analogy, we call x and y *orthogonal* when $\langle x, y \rangle = 0$.

Definition 2.2 (Conjugate). Between self-dual semimodules $\langle M, i \rangle$ and $\langle N, j \rangle$, the *conjugate* of a linear map $f : M \multimap N$ is $f^\dagger = i^{-1} \circ f^\top \circ j : N \multimap M$, the unique linear map satisfying

$$\langle f(x), y \rangle = \langle x, f^\dagger(y) \rangle$$

The conjugate is our backward analysis, and its defining equation makes precise the relationship with the forward one alluded to in §1. Conjugation preserves identities, reverses composition ($g \circ f^\dagger = f^\dagger \circ g^\dagger$) and is involutive, so backward analyses compose according to the chain rule run in reverse, just as reverse derivatives do (Remark 1). Self-dual semimodules and linear maps form a dagger category $\mathbf{SDSemiMod}(S)$. A morphism is not required to relate the chosen self-dualities of its domain and codomain; the choice determines the conjugate, not which maps exist. $\mathbf{SDSemiMod}(S)$ is thus equivalent to the full subcategory of $\mathbf{SemiMod}(S)$ on the semimodules that admit a self-duality.

The matrices of §1 now reappear as concrete instances. The *free* semimodule S^n on n generators has as elements the n-length vectors of scalars, with standard basis vectors e_i and every $v \in S^n$ decomposing as $v = \sum_i v_i \cdot e_i$. Currying the *dot product* $\langle u, v \rangle = \sum_i u_i \cdot v_i$ gives an isomorphism $S^n \cong (S^n)^*$, since pairing with the basis vectors extracts components, $\langle e_i, v \rangle = v_i$; so free semimodules

are self-dual. By basis decomposition, a linear map $S^m \multimap S^n$ is exactly an $n \times m$ matrix over S, with composition as matrix multiplication and the conjugate as the familiar matrix transpose, $(M^\dagger)_{ji} = M_{ij}$.

We write $\mathbf{Mat}(S)$ for the skeletal category whose objects are the natural numbers and whose morphisms $m \to n$ are the $n \times m$ matrices. The Boolean Jacobians from §1 live here: a dependency relation between m input positions and n output positions is exactly a morphism $m \to n$ in $\mathbf{Mat}(\mathbb{2})$, where $\mathbb{2}$ is the Boolean semiring with $+ = \vee$ and $\cdot = \wedge$, and the chain rule for dependencies is composition in $\mathbf{Mat}(\mathbb{2})$, or equivalently *relational* composition under the equivalence of categories $\mathbf{Mat}(\mathbb{2}) \simeq \mathbf{FinRel}$. The dot product can be interpreted as "overlap", becauses $\langle u, v \rangle = \top$ exactly when the selections u and v share a position, i.e. are not disjoint.

The trivial semimodule is a zero object in $\mathbf{SDSemiMod}(S)$, and self-dual semimodules are closed under the *biproduct* $M \oplus N$, an object which is both a product and a coproduct, with the injections conjugate to the projections, $\mathsf{i}_i = \mathsf{p}_i^\dagger$. The coincidence of products and coproducts is characteristic of such linear settings, and means $\mathbf{SDSemiMod}(S)$ on its own cannot interpret programs over sum types; we return to this in §3.

2.3 Dependency Structures over Semirings

The framework so far only asks for commutativity of the scalars, which makes the dual M^* an S-semimodule again, so that the transpose remains a map between S-semimodules. Any commutative semiring S thus yields a category $\mathbf{SDSemiMod}(S)$ with forward derivatives and their conjugates, related through the pairing. When S is the reals, the conjugate is the familiar adjoint of linear algebra. In a dependency-tracking setting, however, the relationships of interest are typically granular: we ask whether, or to what degree, an output depends on an input, and want to compare one answer with another. This calls for an order on dependency information, which additional axioms on S provide.

2.3.1 Conjugate pairs and conjugacy analyses. Suppose first that 1 is an additive top in S ($1 + a = 1$). Addition is then idempotent, meaning $a + a = a$. Idempotent addition induces an order: S becomes a join-semilattice, with $a \leq b$ iff $a + b = b$, joins given by $+$ and least element 0. This structure lifts pointwise to every semimodule, and every linear map automatically preserves joins and 0. Dependency information is now ordered by informativeness: 0 means "no dependency", and the forward and backward analyses are monotone, taking more informative inputs to more informative outputs and vice-versa.

If multiplication is also idempotent ($a \cdot a = a$), then S is a bounded distributive lattice, with multiplication as the meet and distributivity given by the semiring. This opens the way to reading the two analyses in terms of *disjointness*: where meets exist, define $x \# y$ when $x \wedge y = 0$, and if disjointness agrees with the orthogonality given by the pairing, the defining equation of the conjugate (Definition 2.2) says that f and $f^\dagger$ exchange disjointness: $f(x) \# y$ iff $x \# f^\dagger(y)$. Let us therefore consider the case where each semimodule has meets, preserved by the scalar action and agreeing with orthogonality in this sense; meets, unlike joins, do not lift from S, so this is additional structure. Maps that exchange disjointness in this way have a standard name.

Definition 2.3 (Conjugate pair). Join-preserving maps $f : L \to M$ and $g : M \to L$ between bounded lattices form a *conjugate pair* when $f(x) \# y$ iff $x \# g(y)$ for all $x \in L$ and $y \in M$.

The notion originates in relation algebra [Jonsson and Tarski 1951], where the conjugate of composition with a relation is composition with its converse. The scalars S, viewed as a semimodule over itself, qualify automatically for this reading, and biproducts (including the empty biproduct S^0)

inherit the structure componentwise, giving the linear maps between free semimodules S^n, along with their conjugates, as the canonical instances of conjugate pairs.

The relevance to us here is that conjugate pairs also underpin the *cognacy* (common ancestry) analyses of work on linked visualisations [Bond et al. 2025; Perera et al. 2022]. These dependency analyses work by composing a join-preserving dependency analysis with its conjugate, yielding a map sending a selection of outputs to its *related outputs*, those sharing a dependency on some part of the input; selections in one chart can then be linked to "cognate" selections in another.

2.3.2 Galois slicing. Suppose in addition that every scalar has a complement: an operation $\neg$ with $a \wedge \neg a = 0$ and $a \vee \neg a = 1$, making S a Boolean algebra. Negation supports an alternative presentation of conjugates as adjoints. Composing a map with complements on either side preserves its direction but exchanges join-preservation for meet-preservation; the *De Morgan dual* $\neg \circ f^\dagger \circ \neg$ of the conjugate is thus meet-preserving, and is precisely the upper adjoint of f (upper adjoints preserve meets, dually to lower adjoints and joins). Forward and backward analysis therefore form a *Galois connection* between the lattices of approximations.

Definition 2.4 (Galois connection). Suppose X and Y are posets. A *Galois connection* $f \dashv g : X \to Y$ is a pair of monotone functions $f : Y \to X$ and $g : X \to Y$ satisfying $y \leq g(x) \iff f(y) \leq x$ for any $x \in X$ and $y \in Y$. Since a Galois connection is also an adjunction, we refer to f as the left adjoint and g as the right adjoint.

This is precisely the setting of Galois slicing [Perera et al. 2012, 2016; Ricciotti et al. 2017], recovered here as the Boolean instance of the general semimodule picture. Galois connections as such presuppose none of the algebraic structure of this section, being definable between arbitrary posets; De Morgan duality is simply how the adjoint presentation arises here, where the conjugate is the primary notion. One orientation to keep straight: the conjugate as presented here, $f^\dagger$, runs backwards, whereas the meet-preserving map of Galois slicing runs forwards. The two presentations agree because $-^\dagger$ is involutive: instantiating the construction at $f^\dagger$ in place of f gives exactly the connection of Galois slicing, with meet-preserving forward map $\neg \circ f \circ \neg$.

The join-preserving and meet-preserving maps answer different questions. In the forwards direction, the join-preserving map f has a *necessity*-flavoured interpretation: which outputs may depend on a given part of the input. Its meet-preserving counterpart $\neg \circ f \circ \neg$ reads as *sufficiency*: which outputs a given input selection determines outright. Notably, in the Galois slicing work the meet-preserving forward map is only used to establish the existence of the join-preserving backward map; it seems to lack any direct dependency analysis applications of its own.

2.3.3 Perturbation bounds. Dependency analysis over the Booleans records whether an output can change when an input does; Example 1.1 raised the more fine-grained question of how much it can change. Numerical analysis studies this question as *propagation of error*: given bounds on how far each input may be perturbed, derive a bound on the output [Higham 2002]. Propagation of error is again a choice of semiring, this time the *tropical semiring* $(\mathbb{Q}_{\geq 0} \cup \{\infty\}, \min, \infty, +, 0)$ [Simon 1988]. A scalar is a non-negative rational (or ∞) bounding the extent to which a value may change from its value in the run, with ∞ meaning unconstrained. Semiring addition is min, so that combining bounds for parallel paths keeps the tighter bound, and multiplication is numerical +, accumulating bounds along a path. Addition is idempotent, so the analyses are monotone as in §2.3: smaller bounds are more informative, and the semiring zero ∞ means "no information".

A Jacobian entry c now acts as a translation: if the input at that position changes by at most δ, with the others held fixed, the output changes by at most $\delta + c$. Addition's entries are 0: a bound on either argument passes through undisturbed. Multiplication, however, scales perturbations: if

x changes by δ then $x \cdot y$ changes by $|y|\,\delta$, and no translation bounds a scaling; its entries are ∞. This is sound, but reports nothing: propagation of absolute error is blind to multiplication.

Measuring perturbations multiplicatively instead keeps min as the join of bounds but accumulates along paths by numerical product: $(\mathbb{Q}_{\geq 0} \cup \{\infty\}, \min, \infty, \times, 1)$, with scalars now read as factors. This is the multiplicative counterpart of the previous presentation, related to it by logarithms over the reals, and the two are the classical dichotomy between absolute and relative error [Higham 2002, §§1.6–1.7]. Multiplication now has unit entries, since relative perturbations pass through a product unchanged, while an addition $x + y$ acquires the entry $|x|/|x + y|$: the factor by which it amplifies relative error, its (relative) *condition number*.

In the refund example the final sum adds the products 6 and -6, so the amplification is $|6|/|6 + (-6)|$: an infinite condition number, known in numerical analysis as catastrophic cancellation. For non-negative data the factors are at most 1, so sums never amplify; the refund's negative entry makes the infinite factor possible. Where the Boolean analysis could only report a dependency that was not really there, the relative analysis says how badly the bound degrades. The difference between the two is a choice of semiring, and making that choice is part of specifying what the analysis is to observe.

2.3.4 Qualitative derivatives. The Boolean analysis reports a dependency at a cancellation because it forgets signs. The *sign semiring* $\{+, 0, -, ?\}$, the rule-of-signs domain of abstract interpretation [Cousot and Cousot 1977], retains them: multiplication of signs is exact, and the only sum whose sign is not determined is $+$ plus $-$, which is $?$. Taking signs as the scalars gives the *qualitative derivatives* of qualitative reasoning [de Kleer and Brown 1984; Kuipers 1986], an entry reading "increasing this input increases (or decreases, or cannot affect) that output", with $?$ appearing exactly at the potential cancellations: in the refund example, the entry for the price becomes $?$. Collapsing $+$, $-$ and $?$ to 1 recovers the Boolean analysis.

2.3.5 Linearised provenance. Database provenance has its own semiring tradition: in the framework of Green et al. [2007], query results carry annotations from a commutative semiring, with the *free* commutative semiring $\mathbb{N}[X]$ of *provenance polynomials* playing a universal role: its variables name the input tuples, multiplication records their joint use in a derivation, and addition collects the alternative derivations. By freeness, assigning a value to each variable extends uniquely to a semiring homomorphism out of $\mathbb{N}[X]$, so provenance computed once as polynomials determines, by evaluation, the annotation in every other semiring.

Provenance semirings can serve as the scalars of the present framework, but with a limitation: a derivative is linear, so the framework captures *linearised* provenance. Taking $S = \mathbb{N}[X]$, a Jacobian entry is a polynomial recording symbolically how an input position contributes to an output position. Green's semantics assigns an output a polynomial p whose degree records joint use; seeding each input position with a unique variable, our forward analysis instead returns a polynomial linear in the position variables, with the partial derivative $\partial p/\partial x_i$, evaluated at the run's input values, as the coefficient of x_i. Joint use survives only in the coefficients: an input used twice contributes $2x$ where the annotation would record x^2.

Translation along a homomorphism applies to the present framework as well: a semiring homomorphism acts entrywise on Jacobians, and the action is functorial, since matrix composition is built from addition and multiplication. The translation of interest for dependency tracking is "zero-testing" into the Booleans, and whether that is a homomorphism depends on the source semiring. The counting semiring $\mathbb{N}$, itself a provenance semiring recording how many derivations use each input, is *positive*: a sum is zero only when both summands are, and a product only when a factor is; equivalently, zero-testing is a homomorphism, so counting translates exactly to dependency

tracking. The rationals are not positive, since a number and its negation sum to zero, and zero-testing $\mathbb{Q} \to \mathbb{2}$ preserves multiplication exactly but addition only laxly. It still acts entrywise on matrices, preserving identities and transposition, but composition only up to over-approximation: composing Boolean Jacobians can record dependencies absent from the Boolean image of their composite. The refund in Example 1.1 is a case in point: the price genuinely does not matter to the total, yet the composed dependencies say it might.

Remark 2. For most of the semirings in this section the analysis is plainly different in kind from numeric differentiation: it reports use counts, symbolic polynomials or perturbation bounds rather than rates of change. Over the Booleans, however, one might hope for a shortcut: compute the rational Jacobian and then record, for each entry, only whether it is zero. Indeed, there is a potential gain in accuracy: composing Boolean Jacobians may record dependencies that the zero-tested rational Jacobian omits, never the reverse, so the shortcut sometimes gives a sharper answer at the given run. But it also answers a different question. Zero-testing certifies that the output is insensitive to the input at this run, and for a nonlinear operation that is a weaker property than the input being unnecessary: squaring has derivative zero at zero although its output is not constant. The Boolean analysis instead soundly approximates the inputs needed to recompute the output, the guarantee that Galois slicing relies on.

3 Models of Semiring Dependency

For basic differentiable programming, inputs and outputs are over Euclidean spaces where the tangent spaces are uniform and always isomorphic to the original space. It is then possible to, for first order programs at least, to identify a space and its tangent spaces. For semiring dependency analysis we cannot do this. It is not often the case that the types of data the programs operate will be the same as the semiings used for dependency tracking. To apply the semiring settings identified in §2 to a programming language we need an interpretation that attaches semimodules over our chosen semiring to types' interpretations, and linear maps to the interpretations of terms. Given the similarities noted in the introduction between provenance tracking and automatic differentiation, we follow Vákár and collaborators [Lucatelli Nunes and Vákár 2023; Vákár and Smeding 2022] who used the *Category of Families* to build denotational models of automatic differentiation over the reals. We start by recalling the definition of biproducts in categories enriched over commutative monoids, which will be the essential property of modules over semirings that we need.

All of the results of this section have been mechanised in Agda .

3.1 CMon-Categories and Biproducts

As we noted at the end of §2.2, categories of semirings and semimodules have *biproducts*. Loosely stated, biproducts are objects that are both products and coproducts. The concept can be defined in any category, as shown by Karvonen [2020], but for our purposes it will be more convenient to use the more concise definition in categories enriched in commutative monoids:

Definition 3.1. A category C is enriched in **CMon**, the category of commutative monoids, if every homset $C(X, Y)$ is a commutative monoid with $(+, 0)$ and composition is bilinear:

$$f \circ 0 = 0 = 0 \circ f$$
$$(f + g) \circ h = (f \circ h) + (g \circ h) \qquad h \circ (f + g) = (h \circ f) + (h \circ g)$$

In any **CMon**-category we can define what it means to be the biproduct of two objects:

Definition 3.2. In a **CMon**-category a biproduct is an object $X \oplus Y$ together with morphisms

$$X \underset{\mathsf{p}_X}{\overset{\mathsf{i}_X}{\leftrightarrows}} X \oplus Y \underset{\mathsf{p}_Y}{\overset{\mathsf{i}_Y}{\leftrightarrows}} Y$$

satisfying

$$\mathsf{p}_X \circ \mathsf{i}_X = \mathsf{id}_X \qquad \mathsf{p}_Y \circ \mathsf{i}_Y = \mathsf{id}_Y$$
$$\mathsf{p}_Y \circ \mathsf{i}_X = 0_{X,Y} \qquad \mathsf{p}_X \circ \mathsf{i}_Y = 0_{Y,X}$$
$$(\mathsf{i}_X \circ \mathsf{p}_X) + (\mathsf{i}_Y \circ \mathsf{p}_Y) = \mathsf{id}_{X \oplus Y}$$

A zero object is an object that is both initial and terminal.

As the name suggests, biproducts in a category are both products and coproducts:

Proposition 3.3.
(1) A **CMon***-category that has biproducts $X \oplus Y$ for all X and Y also has products and coproducts with $X \times Y = X + Y = X \oplus Y$.*
(2) A **CMon***-category with (co)products also has biproducts, and any initial or terminal object is a zero object.*

Example 3.4. The following are **CMon**-enriched and have finite products, and hence biproducts:
(1) In **FDVect**, morphisms are linear maps and so can be added and have a zero map. Finite products are given by Cartesian products of the underlying sets, with the vector operations defined pointwise.
(2) In **SDSemiMod**(S) and **SemiMod**(S) morphisms can be added and have a zero map. Biproducts are also given by Cartesian product of the underlying sets. The chosen self-duality is preserved because we always have $(M \oplus N)^*$ is a biproduct of M^* and N^*.

Remark 3. A **CMon**-category having biproducts means that its finite products and coproducts coincide, but this does not necessarily extend to infinite products and coproducts. **SemiMod**(S) has infinite products and coproducts, but these do not coincide. For this reason, **SDSemiMod**(S) does not have infinite products because the dual of an infinite product is no longer uniquely isomorphic to an infinite product of duals. **FDVect** also lacks infinite products: the points of a product are tuples of points of the factors, so a product of countably many copies of $\mathbb{R}$ would be infinite-dimensional. This will become relevant when we consider Cartesian closure in §3.3.2.

Remark 4. Categories with zero objects cannot be Cartesian closed without being trivial in the sense of having exactly one morphism between every pair of objects because $C(X, Y) \cong C(1 \times X, Y) \cong C(0 \times X, Y) \cong C(0, X \to Y) \cong 1$. Consequently, we cannot apply the alternative construction of exponentials described in Remark 5.

3.2 The Category of Families Construction

As explained in §2.1, every point in a manifold has an associated vector space of *tangents* at that point, and every smooth function has an associated function mapping points to linear maps between tangent spaces. The *Category of Families* construction abstracts this situation, forgetting any additional structure such as the topology or higher derivatives and replacing the category of real finite-dimensional vector spaces with an arbitrary category:

Definition 3.5. Let C be a category. The *Category of Families* over C, **Fam**(C), has as objects pairs $(X, \partial X)$, where X is a set and $\partial X : X \to C$ is an X-indexed family of objects in C. A morphism $f : (X, \partial X) \to (Y, \partial Y)$ consists of a pair of a function $f : X \to Y$ and a family of morphisms of C, $\partial f : \Pi_{x:X}.\, C(\partial X(x), \partial Y(f\,x))$.

The reason for choosing the **Fam** construction is that composition in this category is an abstract version of the chain rule that we have seen in Remark 1. Composition $f \circ g$ of morphisms f :

$(Y, \partial Y) \to (Z, \partial Z)$ and $g : (X, \partial X) \to (Y, \partial Y)$ in this category is given by normal function composition on the set components, and $\partial(f \circ g)(x) = \partial f(f, x) \circ \partial g(x)$, where the latter composition is in C.

The fact that morphisms in $\mathbf{Fam}(C)$ compose according to a chain rule means that the categories we considered in §2 embed into $\mathbf{Fam}(C)$ for appropriate C. If we let **FDVect** be the category of finite dimensional real vector spaces and linear maps, then:

Proposition 3.6. *There is a faithful functor* $\mathbf{Man} \to \mathbf{Fam}(\mathbf{FDVect})$ *that sends a manifold* M *to* $(M, \lambda x.T_x(M))$*, and each smooth function* f *to* (f, f_*)*, the pair of* f *and its forward derivative.*

A similar result is given by Cruttwell et al. [2022], where Euclidean spaces $\mathbb{R}^n$ and smooth functions are embedded into a category of lenses (the "simply typed" version of the **Fam** construction). As in Vákár and Smeding [2022], the idea is to formally separate functions on points and their forward/reverse tangent maps for the purposes of implementation of automatic differentiation. In the case of smooth maps, this process throws away information on higher derivatives by turning smooth maps into pairs of plain functions and linear functions.

We will use the categories $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$ as our model of semiring-tracking data provenance, analogous to Vákár *et al.*'s use of $\mathbf{Fam}(\mathbf{FDVect})$. The self-duality of objects in $\mathbf{SDSemiMod}(S)$ means that we can take transposes of tangent maps to track reverse provenance. To interpret expressive languages in this setting, we need to ensure it has sufficient categorical structure. Unfortunately, it is the case that $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$ is not Cartesian closed, and so cannot support an interpretation of higher order functions. This motivates the use of the more liberal category $\mathbf{Fam}(\mathbf{SemiMod}(S))$ to interpret higher order programs.

3.3 Categorical Properties of $\mathbf{Fam}(C)$

3.3.1 Coproducts and Products. The categories $\mathbf{Fam}(C)$ are the free coproduct completions of categories C, so they have all coproducts:

Proposition 3.7. *For any* C*,* $\mathbf{Fam}(C)$ *has all coproducts, which can be given on objects by:*

$$\coprod_i (X_i, \partial X_i) = (\coprod_i X_i, \lambda(i, x_i).\, \partial X_i(x))$$

Coproducts in $\mathbf{Fam}(C)$ *are* extensive *[Carboni et al. 1993, Proposition 2.4].*

For $\mathbf{Fam}(C)$ to have finite products, we need C to have finite products:

Proposition 3.8. *If* C *has finite products, then so does* $\mathbf{Fam}(C)$*. On objects, binary products can be defined by:*

$$(X, \partial X) \times (Y, \partial Y) = (X \times Y, \lambda(x, y).\partial X(x) \times \partial Y(y))$$

Since $\mathbf{Fam}(C)$ *is extensive, products and coproducts distribute.*

Using the infinitary coproducts and finite products, we can construct a wide range of other useful semantic models of datatypes in $\mathbf{Fam}(C)$. For example, lists can be constructed as a coproduct

$$\mathrm{List}(X) = \coprod_{n \in \mathbb{N}} X^n \tag{1}$$

where $X^0 = 1$ (the terminal object) and $X^{n+1} = X \times X^n$.

Our category of interest, $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$ has coproducts and finite products, because $\mathbf{SDSemiMod}(S)$ has (bi)products. Similarly for $\mathbf{Fam}(\mathbf{SemiMod}(S))$.

A direct consequence of our chosen construction of products and coproducts in $\mathbf{Fam}(C)$ is:

Lemma 3.9. *The first projection functor* $\pi_1 : \mathbf{Fam}(C) \to \mathbf{Set}$ *preserves all products and coproducts.*

3.3.2 Cartesian Closure. For Cartesian closure of the categories $\mathbf{Fam}(C)$ that we are interested in, we rely on the following theorem of Lucatelli Nunes and Vákár [2023], specialised from their setting with the general Grothendieck construction to $\mathbf{Fam}(C)$.

Theorem 3.10 ([Lucatelli Nunes and Vákár 2023]). *(Formalised in Agda). If C has biproducts (Definition 3.2) and all products, then $\mathbf{Fam}(C)$ is Cartesian closed*[1]*. On objects, the internal hom can be given by:*

$$(X, \partial X) \multimap (Y, \partial Y) = (\Pi_{x:X}.\Sigma_{y:Y}.C(\partial X(x), \partial Y(y)), \lambda f.\Pi_{x:X}.\partial Y(\pi_1(f\,x)))$$

The $\mathbf{Set}$-component of $(X, \partial X) \multimap (Y, \partial Y)$ consists of exactly the morphisms of $\mathbf{Fam}(C)$, rephrased into a single object. When $C = \mathbf{FDVect}$ or $\mathbf{SemiMod}(S)$, these are functions with an associated linear map at every point. A tangent to a function is then defined to be a mapping from points in the domain to tangents in the codomain along the function.

The category $\mathbf{SemiMod}(S)$ satisfies the hypotheses of Theorem 3.10, so:

Corollary 3.11. $\mathbf{SemiMod}(S)$ *is Cartesian closed and has all coproducts.*

Unfortunately, neither $\mathbf{SDSemiMod}(S)$ nor $\mathbf{FDVect}$ satisfy the hypotheses of this theorem, because as we noted above, neither of them have infinite products. So we are forced to use $\mathbf{Fam}(\mathbf{SemiMod}(S))$ to interpret higher order functions, which appears to forego the ability to obtain reverse derivatives by transpose of linear maps afforded by $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$. However, $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$ fully embeds into $\mathbf{Fam}(\mathbf{SemiMod}(S))$, so we can read back interpretations of first-order programs and compute their transpose derivatives.

Proposition 3.12. *There is a functor $H : \mathbf{Fam}(\mathbf{SDSemiMod}(S)) \to \mathbf{Fam}(\mathbf{SemiMod}(S))$ defined on objects as $H(X, \partial X) = (X, \lambda x.U(\partial X(x)))$, where $U : \mathbf{SDSemiMod}(S) \to \mathbf{SemiMod}(S)$ is the functor that forgets the self-dual data. The functor H preserves products (because U does) and all coproducts. It is also full and faithful.*

As we will see in §6, this is no problem for programs of first order type.

Remark 5. There is another construction of internal homs on $\mathbf{Fam}(C)$ arising from the use of fibrations for categorical logical relations, due to Hermida [1999, Corollary 4.12]. If we assume that C is itself Cartesian closed and has all products, then we could construct an internal hom as:

$$(X, \partial X) \multimap (Y, \partial Y) = (X \to Y, \lambda f.\Pi_{x:X}.\, \partial X(x) \to \partial Y(f\,x))$$

However, for the purposes of modelling differentiable programs, this is fatally flawed in that neither $\mathbf{SemiMod}(S)$ nor $\mathbf{FDVect}$ are Cartesian closed, and there is no way of making them so without losing the property of being able to conjunct or add tangents (Remark 4).

4 Higher-Order Language

To demonstrate semiring provenance tracking via automatic differentiation for higher-order programs, we define a simple total functional programming language, extending the simply-typed lambda calculus. The language is parameterised by a signature $\Sigma = (\mathsf{PrimTy}, \mathsf{Op})$ consisting of a set PrimTy of base types ρ and a family of sets $\mathsf{Op}^{\rho}_{\rho_1,\ldots,\rho_n}$ of primitive operations ϕ of arity n over those base types.

[1] More precisely, if C has coproducts then we have a monoidal product on $\mathbf{Fam}(C)$ which is closed by this construction. When these coproducts are in fact biproducts, we get Cartesian closure.

Types				*Terms*			
σ, τ	::=	ρ	primitive type	t, s	::=	x	variable
	|	$\sigma + \tau$	sum		|	$\phi(\vec{t})$	primitive op
	|	$\mathbf{1}$	unit		|	$\mathbf{inl}\ t \mid \mathbf{inr}\ t$	injection
	|	$\sigma \times \tau$	product		|	$\mathbf{case}\ s\ \{x.t_1; y.t_2\}$	case
	|	$\sigma \to \tau$	function		|	$()$	unit
	|	$\mathbf{list}\ \tau$	list		|	(s, t)	pair
					|	$\mathbf{fst}\ t \mid \mathbf{snd}\ t$	projection
					|	$\lambda x.t$	function
					|	$s\ t$	application
					|	$\mathbf{nil} \mid \mathbf{cons}\ s\ t$	nil & cons
					|	$\mathbf{fold}\ s_1\ s_2\ t$	fold

Fig. 1. Syntax of types and terms

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\phi \in \mathrm{Op}^{\rho}_{\rho_1,\ldots,\rho_n} \qquad \Gamma \vdash t_i : \rho_i \quad (\forall i \in \{1..n\})}{\Gamma \vdash \phi(t_1, \ldots, t_n) : \rho} \qquad \frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \mathbf{inl}\ t : \sigma + \tau} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathbf{inr}\ t : \sigma + \tau}$$

$$\frac{\Gamma \vdash s : \sigma + \tau \qquad \Gamma, x : \sigma \vdash t_1 : \tau' \qquad \Gamma, y : \tau \vdash t_2 : \tau'}{\Gamma \vdash \mathbf{case}\ s\ \{x.t_1; y.t_2\} : \tau'} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}} \qquad \frac{\Gamma \vdash s : \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash (s, t) : \sigma \times \tau}$$

$$\frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash \mathbf{fst}\ t : \sigma} \qquad \frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash \mathbf{snd}\ t : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash s : \sigma \to \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash s\ t : \tau} \qquad \frac{}{\Gamma \vdash \mathbf{nil} : \mathbf{list}\ \tau}$$

$$\frac{\Gamma \vdash s : \tau \qquad \Gamma \vdash t : \mathbf{list}\ \tau}{\Gamma \vdash \mathbf{cons}\ s\ t : \mathbf{list}\ \tau} \qquad \frac{\Gamma \vdash s_1 : \tau \qquad \Gamma, x : \sigma, y : \tau \vdash s_2 : \tau \qquad \Gamma \vdash t : \mathbf{list}\ \sigma}{\Gamma \vdash \mathbf{fold}\ s_1\ s_2\ t : \tau}$$

Fig. 2. Well-typed terms over a signature Σ

4.1 Syntax

The syntax is defined in Figure 1. Types includes base types ρ drawn from PrimTy, along with standard type formers for sums, products, functions and lists. Terms include variables, the usual introduction and elimination forms, and primitive operations ϕ.

The language is intentionally minimal: it excludes general recursion, and general inductive or coinductive types, which we will consider in future work (§8). Typing judgments for terms are standard and shown in Figure 2, with the usual rules for products, sums, functions, and lists.

4.2 Semantics

An interpretation of a signature $\Sigma = (\mathsf{PrimTy}, \mathsf{Op})$ can be given in any category C with finite products, and assigns to each base type $\rho \in \mathsf{PrimTy}$ an object $[\![\rho]\!]_{\mathsf{PrimTy}}$ in C, and to each primitive operation $\phi \in \mathrm{Op}^{\rho}_{\rho_1,\ldots,\rho_n}$, a morphism $[\![\phi]\!]_{\mathsf{Op}} : [\![\rho_1]\!]_{\mathsf{PrimTy}} \times \ldots \times [\![\rho_n]\!]_{\mathsf{PrimTy}} \to [\![\rho]\!]_{\mathsf{PrimTy}}$.

Assuming that C is bicartesian closed and has a list object (Equation 1), then we can extend an interpretation of a signature Σ to an interpretation of the whole language over Σ. Figure 3a and Figure 3b define the interpretation of types and contexts as objects of C respectively. Terms are interpreted as morphisms between the interpretations of the context and type, as defined in Figure 3c. We have used the notations π_i for projections, $\langle f, g \rangle$ for pairing, $[f, g]$ for (parameterised)

$$\llbracket \rho \rrbracket = \llbracket \rho \rrbracket_{\mathsf{PrimTy}} \qquad \llbracket \sigma \times \tau \rrbracket = \llbracket \sigma \rrbracket \times \llbracket \tau \rrbracket$$
$$\llbracket \sigma + \tau \rrbracket = \llbracket \sigma \rrbracket + \llbracket \tau \rrbracket \qquad \llbracket \sigma \to \tau \rrbracket = \llbracket \sigma \rrbracket \Rightarrow \llbracket \tau \rrbracket$$
$$\llbracket \mathbf{1} \rrbracket = 1 \qquad \llbracket \mathbf{list}\ \tau \rrbracket = \mathrm{List}(\llbracket \tau \rrbracket)$$

(a) Interpretation of Types

$$\llbracket \cdot \rrbracket = 1$$
$$\llbracket \Gamma, x : \tau \rrbracket = \llbracket \Gamma \rrbracket \times \llbracket \tau \rrbracket$$

(b) Interpretation of Contexts

$$\llbracket x_i \rrbracket = \pi_i \qquad \llbracket \mathbf{fst}\ t \rrbracket = \pi_1 \circ \llbracket t \rrbracket$$
$$\llbracket \phi(t_1, \ldots, t_n) \rrbracket = \llbracket \phi \rrbracket_{\mathrm{Op}} \circ \langle \llbracket t_1 \rrbracket, \ldots, \llbracket t_n \rrbracket \rangle \qquad \llbracket \mathbf{snd}\ t \rrbracket = \pi_2 \circ \llbracket t \rrbracket$$
$$\llbracket \mathbf{inl}\ t \rrbracket = \mathsf{inj}_1 \circ \llbracket t \rrbracket \qquad \llbracket \lambda x.t \rrbracket = \lambda(\llbracket t \rrbracket)$$
$$\llbracket \mathbf{inr}\ t \rrbracket = \mathsf{inj}_2 \circ \llbracket t \rrbracket \qquad \llbracket s\ t \rrbracket = \varepsilon \circ \langle \llbracket s \rrbracket, \llbracket t \rrbracket \rangle$$
$$\llbracket \mathbf{case}\ s\ \{x.t_1; y.t_2\} \rrbracket = [\llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket] \circ \langle \mathsf{id}, \llbracket s \rrbracket \rangle \qquad \llbracket \mathbf{nil} \rrbracket = \mathsf{nil} \circ !_{\llbracket \Gamma \rrbracket}$$
$$\llbracket () \rrbracket = !_{\llbracket \Gamma \rrbracket} \qquad \llbracket \mathbf{cons}\ s\ t \rrbracket = \mathsf{cons} \circ \langle \llbracket s \rrbracket, \llbracket t \rrbracket \rangle$$
$$\llbracket (s, t) \rrbracket = \langle \llbracket s \rrbracket, \llbracket t \rrbracket \rangle \qquad \llbracket \mathbf{fold}\ t_1\ t_2\ s \rrbracket = \mathsf{fold}(\llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket) \circ \langle \mathsf{id}, \llbracket s \rrbracket \rangle$$

(c) Terms as morphisms

Fig. 3. Interpretation of types, contexts and terms

copairing, $!_X$ for morphisms to the terminal object, and λ and ε for currying and evaluation for exponentials.

We have another interpretation $\llbracket - \rrbracket_{fo}$ of the first-order types (those constructed from primitive types, sums, unit and products) in any bicartesian category. Such interpretations are preserved by finite coproduct and coproduct preserving functors:

Lemma 4.1. *If $\mathcal{C}$ and $\mathcal{D}$ are bicartesian and bicartesian closed categories with interpretations of the signature Σ, $F : \mathcal{C} \to \mathcal{D}$ is a bicartesian functor, and $F(\llbracket \rho \rrbracket_{\mathsf{PrimTy}}) \cong \llbracket \rho \rrbracket_{\mathsf{PrimTy}}$ for all ρ, then for all first-order types τ, $F(\llbracket \tau \rrbracket_{fo}) \cong \llbracket \tau \rrbracket$, and similar for contexts only containing first-order types.*

4.2.1 Interpretation in the Semimodule Model. Given the above, we can now interpret the language in any of the bicartesian closed categories with list objects we constructed in §3. Specifically, we assume that we have an interpretation of our chosen signature in $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$. Each base type is assigned a self-dual approximation object, and the first-order type formers preserve self-duality, so every first-order type denotes a self-dual semimodule. Signatures are first-order, so it does not matter that $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$ is not closed. Any such interpretation can be transported to $\mathbf{Fam}(\mathbf{SemiMod}(S))$ along the functor H from Proposition 3.12 because it preserves finite products. We can then interpret the whole language in $\mathbf{Fam}(\mathbf{SemiMod}(S))$.

Interpreting a program $\Gamma \vdash t : \tau$ yields a morphism of $\mathbf{Fam}(\mathbf{SemiMod}(S))$, the underlying function together with its forward derivative. In contrast to many other bidirectional program analysis techniques, a single morphism is enough, because a linear map has a transpose for free. At first-order types, Lemma 4.1 identifies the interpretation with that of the first-order model $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$, whose objects are self-dual, so the forward derivative is a morphism of $\mathbf{Fam}(\mathbf{SDSemiMod}(S))$ whose conjugate $f^\dagger$ is the backward map between the same semimodules.

5 Examples

We now run the interpretation on a range of example programs, over various semirings, with the aim of illustrating the different flavours of analysis the framework supports. The examples are kept small so that each can be mechanised in the accompanying Agda development, with the programs

and their derivatives run inside the type checker and the results checked against the expected values by decidable equality.

Throughout, we work over the signature Σ_{ex}, parameterised by a set of numbers with a distinguished zero, which the examples instantiate with the rationals or the integers. It has sorts num and label; operations add, mult : num × num ⟶ num, a literal for each number, and a literal for each label; and an equality relation on labels, used by the comprehension syntax. In the intended interpretations, the zero literal is a unit for add. Each instance must also interpret each operation of the signature, supplying the function it computes on values together with, at each input value, a linear map between the approximation objects, a semimodule morphism giving the operation's intended derivative at that value.

5.1 Perturbation Bounds

The first two instances demonstrate the perturbation-bound reading of §2.3.3: the analysis reports how much each output can change, rather than whether it can. The two presentations, absolute (§5.1.1) and relative (§5.1.2), run on the database $db = [(\mathrm{a}, 3), (\mathrm{b}, 1), (\mathrm{a}, -3)]$ of Example 1.1. In the absolute presentation the approximation object for a number is the free semimodule S^2, one bound for each direction of change; a relative bound constrains both directions at once, so the relative presentation uses a single scalar.

5.1.1 Absolute bounds. At the min-plus semiring, a pair of scalars (d, u) records that a value may fall by at most d and rise by at most u, so that a number x is known to lie in the interval $[x-d, x+u]$; the pair (∞, ∞) carries no information. Multiplication's derivative carries none either, since no translation bounds a scaling (§2.3.3), so we run a query that selects and sums without multiplying: query $l\ db$ = sum $[q \mid (l', q) \leftarrow db, l \equiv l']$, with query a $db = 0$. Addition's derivative entries are translations by 0, and selection's entries are the semiring's 0 and 1, as before. The backward derivative at this run is a linear map $\partial(\text{query } l\ db)_r : S^2 \multimap S^2 \times S^2 \times S^2$, one factor for each number in the database; the frozen label positions and list terminator contribute trivial factors, omitted here, and because the input is a list, the type depends on the input's shape.

Forwards, with the first a-number known to within bounds $(\frac{1}{2}, 0)$ and the second to within $(\frac{1}{5}, \frac{1}{2})$:

$$\partial(\text{query a } db)_f([(\mathrm{a}, (\tfrac{1}{2}, 0)), (\mathrm{b}, (\infty, \infty)), (\mathrm{a}, (\tfrac{1}{5}, \tfrac{1}{2}))]) = (\tfrac{1}{5}, 0)$$

Bounds for the same position combine by min, reflecting the reading in which one input changes at a time. Backwards, asking for the output to lie within $[-\frac{1}{10}, \frac{1}{10}]$, that is, bounds $(\frac{1}{10}, \frac{1}{10})$:

$$\partial(\text{query a } db)_r(\tfrac{1}{10}, \tfrac{1}{10}) = [(\mathrm{a}, (\tfrac{1}{10}, \tfrac{1}{10})), (\mathrm{b}, (\infty, \infty)), (\mathrm{a}, (\tfrac{1}{10}, \tfrac{1}{10}))]$$

Changing either a-number alone by at most $\frac{1}{10}$ keeps the output within its interval, while the b-number is unconstrained. The backward run here can only route the demand to the used positions, since every entry is 0 or ∞; it would become informative given operations that loosen a bound by a fixed finite amount, such as rounding, whose entry is the translation 1 (δ in, $\delta + 1$ out), or several outputs sharing an input, whose demands combine by min.

5.1.2 Relative bounds. Relative bounds pass through multiplication, so this instance can run the full weighted-sum query of Example 1.1, prices included, on the same database. We instantiate at the min-times semiring of §2.3.3: multiplication's derivative has unit entries, and addition's entries are condition numbers, infinite at exact cancellation. In the run of total b, multiplication behaves well: a 10% bound on the price of b reaches the output intact, and likewise a bound on the selected quantity:

$$\partial(\text{total b } db\ price)_f([(\mathrm{a}, \infty), (\mathrm{b}, \infty), (\mathrm{a}, \infty)], \langle \infty, \tfrac{1}{10} \rangle) = \tfrac{1}{10}$$

Backwards, an output bound of 10% constrains the selected quantity and its price and nothing else:

$$\partial(\text{total b } db\,price)_r(\tfrac{1}{10}) = ([(\text{a}, \infty), (\text{b}, \tfrac{1}{10}), (\text{a}, \infty)], \langle \infty, \tfrac{1}{10} \rangle)$$

The forward run of total a meets the cancellation: its two contributions, $2 \cdot 3$ and $2 \cdot (-3)$, sum to zero, the amplification factor of the final addition is infinite, and no relative bound on either a-quantity, however tight, yields a bound on the output:

$$\partial(\text{total a } db\,price)_f([(\text{a}, \tfrac{1}{10}), (\text{b}, \infty), (\text{a}, \infty)], \langle \infty, \infty \rangle) = \infty$$

5.2 Linearised provenance

At the free semiring $\mathbb{N}[X]$ the analysis computes the linearised provenance of §2.3.5. Each instance must specify the derivative entries of the primitive operations; here every entry is 1, so a Jacobian entry of a run counts the paths from an input position to an output position. On the cancellation run of the weighted-sum query (Example 1.1), seeding each input position with a unique variable:

$$\partial(\text{total a } db\,price)_f([(\text{a}, x_0), (\text{b}, x_1), (\text{a}, x_2)], \langle x_3, x_4 \rangle) = x_0 + x_2 + 2x_3$$

The output polynomial records each selected quantity once and the queried price twice, once per selected row. Backwards, seeding the output with a variable y distributes it by usage:

$$\partial(\text{total a } db\,price)_r(y) = ([(\text{a}, y), (\text{b}, 0), (\text{a}, y)], \langle 2y, 0 \rangle)$$

Other analyses now come by evaluation. Setting every variable to 1 specialises to the counting semiring $(\mathbb{N}, +, 0, \cdot, 1)$: the forward polynomial evaluates to 4, the number of paths from perturbable inputs to the output, and the backward polynomials evaluate to per-position use counts, 2 at the queried price. The count survives the cancellation that zeroes the rational derivative, because $\mathbb{N}$ is positive and counts cannot cancel; zero-testing then sends the count 2 to the Boolean 1 exactly, as positivity guarantees, whereas the same translation applied to the rational entry loses the dependency. The three analyses answer three different questions at the same position: the rational Jacobian reports sensitivity (0), the Boolean Jacobian possible dependence (1), and the counting Jacobian usage (2).

5.3 Signed Saliency

Consider a program that condenses many numeric inputs into a single score, such as a model whose predictions we want to explain. Ours scores a 3×3 grid of numbers against a fixed pattern, like a small image filter applied to one patch of a larger image. The pattern weights the centre positively and the corners negatively, and includes two products:

$$\text{score}\begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix} = x_5 - (x_1 + x_3 + x_7 + x_9) + x_4 x_6 - x_5 x_2$$

A *saliency map* records the influence of each input on the score; gradient-based explanation methods obtain one from the model's derivative [Selvaraju et al. 2020]. Here we compute a *signed* saliency map over the sign semiring of §2.3.4, taking the integers as the numbers: for each cell, whether increasing it raises or lowers the score. The signs could be read off the rational derivative, but each verdict would then hold only at the particular magnitudes of the run. Computing with signs as the derivatives instead gives verdicts that rest on signs alone, so a definite entry holds under any sign-preserving change of magnitude, and ? marks an input whose direction of influence signs alone are insufficient to determine. For addition, the derivative entries are +; for multiplication, the entry for each argument is the sign of the other.

Running on the grid with rows $(1,2,1)$, $(3,5,4)$ and $(1,7,1)$ gives the score 3. Seeding the backward derivative at this run with + distributes it to the inputs:

$$\partial(\text{score})_r(+) = \begin{pmatrix} - & - & - \\ + & ? & + \\ - & 0 & - \end{pmatrix}$$

The four corners lower the score and the two mid-edge cells raise it, via the product x_4x_6; the bottom-middle cell, which the score ignores, gets 0. At the centre the linear term and the product $-x_5x_2$ pull in opposite directions, so the entry is ?. The rational derivative would instead report a negative entry for the centre, a verdict that depends on the magnitude $x_2 = 2$ rather than on its sign.

5.4 Cognacy Queries

The final example introduces no new semiring: it runs over the Booleans, and shows instead how composing the two derivatives of a single run recovers an existing dependency analysis. When outputs overlap in their dependencies, the composite of the backward and forward derivatives sends a selection of outputs to the outputs related to it by a shared dependency: the *cognacy* analysis of linked visualisations [Bond et al. 2025; Perera et al. 2022; Psallidas and Wu 2018]. The *moving average* is the standard example of such overlap. With window two and four inputs,

$$\text{mavg}\,(x_1,x_2,x_3,x_4) = (\tfrac{1}{2}(x_1+x_2),\ \tfrac{1}{2}(x_2+x_3),\ \tfrac{1}{2}(x_3+x_4))$$

Each adjacent pair of outputs shares an input, and the first and last outputs share none. The cognacy analysis is a *relation* on the outputs, derived from the map's input-output relation, so it is useful to present the matrices involved. Running over the Booleans, the forward derivative and its composite with the backward derivative are

$$\partial_{\mathbb{2}}(\text{mavg}) = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad \partial_{\mathbb{2}}(\text{mavg}) \circ \partial_{\mathbb{2}}(\text{mavg})^{\top} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

The composite relates each output to the outputs it shares an input with. At the input $(1,2,4,8)$ the outputs are $(1.5,3,6)$, and applying the composite to the output selection $(1.5,\bot,\bot)$ returns $(1.5,3,\bot)$, highlighting that 1.5 shares an input with 3; the third output stays $\bot$ because its window is disjoint. But whereas in these prior works, cognacy queries like these run over an execution record, here the intermediate structure has been folded away by the chain rule: each derivative is a direct relation on endpoints.

6 Correctness of the Higher-Order Interpretation

The interpretation of the higher-order language at higher types mixes the interpretation of the underlying computation with the interpretation of the semiring dependency matrices (c.f. Theorem 3.10). It is therefore not immediate that intepreting a program in $\mathbf{Fam}(\mathbf{SemiMod}(S))$ will give us the expected interpretation purely on first order data. Note that the fullness of the functor $H : \mathbf{Fam}(\mathbf{SDSemiMod}(S)) \to \mathbf{Fam}(\mathbf{SemiMod}(S))$ does mean that we are guaranteed that the tangent maps are correctly computed. Vákár and Smeding [2022] and Lucatelli Nunes and Vákár [2023] construct custom instances of categorical sconing arguments to prove correctness of their higher-order interpretation with respect to normal differentiation. Instead of doing this, we make use of a general *syntax free* theorem due to Fiore and Simpson [1999]. The proof of this depends on the construction of a Grothendieck Logical Relation over the extensive topology on the category C, but the statement of the theorem does not rely on this. We have formalised this proof in Agda (see `conservativity.agda` in the supplementary material).

Theorem 6.1 (Fiore and Simpson [1999]). *Let $\mathcal{C}$ be an extensive bicartesian category, $\mathcal{D}$ be a bicartesian closed category, and $F : \mathcal{C} \to \mathcal{D}$ a functor preserving finite products and coproducts. Then there is a category $\mathbf{GLR}(F)$ and functors $p : \mathbf{GLR}(F) \to \mathcal{D}$ and $\hat{F} : \mathcal{C} \to \mathbf{GLR}(F)$, such that:*

(1) $\mathbf{GLR}(\mathcal{D}, F)$ is bicartesian closed;
(2) $F = p \circ \hat{F} : \mathcal{C} \to \mathcal{D}$;
(3) The functor p strictly preserves the bicartesian closed structure; and
(4) The functor $\hat{F}$ is full and preserves the bicartesian structure.

Remark 6. Compared to the exact result stated at the end of Fiore and Simpson [1999]'s paper, we have made two modifications, justified by our Agda proof. First, we generalise to the case where $\mathcal{C}$ is not Cartesian closed, and the functor F does not preserve exponentials. Examination of the proof reveals that if this is the case, then $\hat{F}$ also preserves exponentials, but it is not needed for the result stated. Second, Fiore and Simpson restrict to the case when $\mathcal{C}$ is small to be able to construct Grothendieck sheaves on this category. We use Agda's universe hierarchy to simply construct "large" sheaves at the the appropriate universe level.

We use Theorem 6.1 to show that the interpretation of the language in the category **Set** agrees with the higher-order interpretation in $\mathbf{Fam}(\mathbf{SemiMod}(S))$ on the underlying function at first order, as required. This shows that the higher-order interpretation does what we expect in the underlying interpretation of terms, and that the approximation information does not interfere.

Theorem 6.2. *For all $\Gamma \vdash M : \tau$, where Γ and τ are first-order, the underlying function in the interpretation $[\![\Gamma \vdash M : \tau]\!]_{\mathbf{Fam}(\mathbf{SemiMod}(S))}$ is equal to the interpretation $[\![\Gamma \vdash M : \tau]\!]_{\mathbf{Set}}$ in* **Set**.

Proof. Instantiate Theorem 6.1 with the functor

$$\langle \mathrm{Id}, \pi_1 \rangle : \mathbf{Fam}(\mathbf{SemiMod}(S)) \to \mathbf{Fam}(\mathbf{SemiMod}(S)) \times \mathbf{Set}$$

that is the identity in the first component and projects out the underlying function in the second. By Lemma 3.9, this functor preserves products and coproducts. For each $\Gamma \vdash M : \tau$, we obtain a g such that $g = [\![\Gamma \vdash M : \tau]\!]_{\mathbf{Fam}(\mathbf{SemiMod}(S))}$ and $\pi_1(g) = [\![\Gamma \vdash M : \tau]\!]_{\mathbf{Set}}$. Substituting g yields the result. □

7 Related Work

Automatic Differentiation. Automatic differentiation (AD), discussed in §2.1, is the idea of computing derivatives of functions expressed as programs by systematically applying the chain rule. The observation that these derivative computations could be interleaved with the evaluation of the original program is due to Linnainmaa [1976], who showed how the forward derivative f_{*x} of f at a point x could be computed alongside $f(x)$ in a single pass, dramatically improving the efficiency of derivative evaluation over symbolic or numerical differentiation. This insight became the foundation of forward-mode AD, which underpins many optimisation and scientific computing tools, including JAX [Bradbury et al. 2018]. Griewank [1989] showed how the Wengert list, the linear record of assignments used in forward-mode to compute derivatives efficiently, could be traversed in reverse to compute the pullback map. This two-pass approach is the foundation of reverse-mode AD, and closely resembles implementations of Galois slicing (§7 below) that record a trace during forward slicing for use in backward slicing.

More recent approaches to automatic differentiation have emphasised semantic foundations. Elliott [2018] proposed a categorical model of AD that interprets programs as functions enriched with their derivatives, giving a compositional account of differentiation based on duality and linear maps. Vákár and collaborators [Lucatelli Nunes and Vákár 2023; Vákár and Smeding 2022] developed the CHAD framework which inspired this paper, using Grothendieck constructions over

indexed categories to capture both values and their tangents in a compositional semantic structure. These perspectives shed light on the categorical structure of AD and guide the design of systems that generalise AD, including the application to data provenance and slicing explored in this paper.

Galois slicing. Galois slicing was introduced by Perera and collaborators [Perera 2013; Perera et al. 2012] as an operational approach to program slicing for pure functional programs, based on Galois connections between lattices of input and output approximations. Subsequent work extended the approach to languages with assignment and exceptions [Ricciotti et al. 2017] and concurrency [Perera et al. 2016]. The present work is instead denotational, recasting dependency analysis as differentiation over a semiring; the Boolean case recovers Galois slicing, and other semirings give further analyses. Galois slicing is also more general than our framework in two respects that we plan to consider in future work (§8): it computes *program slices* from approximation lattices representing partially erased programs, and it approximates the *shape* of data rather than only the values at fixed positions. More recently, Perera et al. [2022] presented a variant of Galois slicing over Boolean algebras, which unlike plain lattices are complemented, giving each analysis a conjugate. Composing an analysis with its conjugate computes *related outputs*, which they used to support interactive visualisations with linked selections. Bond et al. [2025] revisited this using *dynamic dependence graphs*, computing the conjugate from the opposite graph. The related-outputs analysis is the cognacy relation shown in §5.4, computed here by composing a Boolean Jacobian with its transpose.

Semiring Provenance. Provenance for database queries has its own semiring tradition, originating with the *provenance semirings* of Green et al. [2007]: a query result carries an annotation from a commutative semiring, and the free semiring $\mathbb{N}[X]$ of provenance polynomials is universal, specialising to other semirings by evaluation. Our framework draws on the same semirings but computes a *derivative* rather than an annotation: a Jacobian over the semiring relating input positions to output positions, in place of a value attached to the output. A derivative is a linear map, so the provenance it records is *linearised* (§2.3.5), and joint use appears as a coefficient rather than the degree of a polynomial. Over a non-positive semiring, the Boolean dependency our framework reports is a sound over-approximation, since the exact analysis can cancel a contribution to zero where the Boolean composite still records a dependency. Extending provenance to deletion and difference likewise calls for non-positive annotations [Amsterdamer et al. 2011; Geerts and Poggi 2010; Green et al. 2009].

Discrete and Qualitative Derivatives. Discrete data has its own differential calculi, developed independently of numerical differentiation. The *Boolean differential calculus* of Steinbach and Posthoff [2017] arose in the 1950s from the detection of faults in digital circuits; its *simple derivative* $\partial f/\partial x_i = f|_{x_i=0} \veebar f|_{x_i=1}$, the exclusive-or of f with the input x_i set to 0 and to 1, is 1 exactly when flipping that input flips the output. Exclusive-or and conjunction make the Booleans a *ring* rather than a lattice, so this is a derivative over the two-element field. Because exclusive-or cancels, the derivative is exact, whereas the join of the Boolean lattice underlying Galois slicing (§7) cannot cancel and only reports a possible dependency. The sign domain gives a second such calculus. The rule of signs of abstract interpretation [Cousot and Cousot 1977] and the *qualitative derivatives* of qualitative physics [de Kleer and Brown 1984; Kuipers 1986] ask whether increasing an input increases, decreases, or cannot affect an output; see the sign semiring of §2.3.4. Our account treats the semiring as a parameter, relating these calculi to automatic differentiation and provenance through the choice of scalars.

Tangent Categories and Differential Linear Logic. Tangent categories, due originally to Rosický [1984] and developed by Cockett and Cruttwell [2014, 2018], provide an abstract categorical

framework for reasoning about differentiation, inspired by the structure of the tangent bundle in differential geometry. In a tangent category, each object X is equipped with a tangent bundle $T(X)$, and each morphism $f : X \to Y$ has a corresponding differential map $T(f) : T(X) \to T(Y)$ satisfying axioms analogous to the chain rule and linearity of differentiation. Tangent categories generalise Cartesian differential categories [Blute et al. 2009], which model differentiation over Cartesian closed categories using a syntactic derivative operator. Reverse Tangent categories [Cruttwell and Lemay 2024] further axiomatise the existence of reverse derivatives. Our construction realises differentiation of this kind concretely, over an arbitrary commutative semiring, and we conjecture that the categories underlying it are examples of tangent, or reverse tangent, categories. There are likely links to Differential Linear Logic [Ehrhard and Regnier 2006]. Differential Linear Logic and the Dialectica translation have been used to model reverse differentiation by Kerjean and Pédrot [2024].

8 Conclusion and Future Work

We have presented a semantic account of data provenance as a form of automatic differentiation, representing the dependency of outputs on inputs as a derivative whose Jacobian is a matrix over a commutative semiring. Traditional automatic differentiation, and various program analysis techniques such as Galois slicing, arise as instances. The model interprets an expressive higher-order language for querying and manipulating data, and reveals new applications such as approximation by intervals (§2.3.3). It also opens up an opportunity to integrate traditional symbolic data provenance techniques with the AD-based methods used in explainable AI, such as the saliency maps of Grad-CAM [Selvaraju et al. 2020]. Our categorical approach admits a modular construction of the model, and the use of general theorems, such as Fiore and Simpson's Theorem 6.1, to prove properties of the interpretation. We have focused on constructions that enable an executable implementation in Agda.

Alternative semiring models. The choice of scalar semiring is the principal degree of freedom in our framework, and choices beyond those studied here seem worth pursuing. The perturbation-bound semirings of §5.1 point towards metric approximation: a Lawvere metric space is a category enriched over the min-plus quantale, whose operations are those of our absolute-perturbation semiring, so metric spaces provide a native notion of approximation already partly within reach of the framework. Relating this to Edalat and Hackmann [1998]'s computational model of metric spaces is one promising direction. A second is to replace the setoids carrying values with sets equipped with a semiring-valued equality relation, over a partially ordered semiring with a residual for its multiplication.

Connection to sequentiality and stable functions. In our model the forward derivative carried by a morphism of $\mathbf{Fam}(\mathbf{SemiMod}(S))$ is supplied as part of the interpretation, not determined by the underlying function, so it need not reflect how that function depends on its input. Berry's *stable functions* [Berry 1979; Berry and Curien 1982], by contrast, come equipped with an intrinsic forward and backward derivative. Stability was proposed as a refinement of domain theory aimed at capturing the intensional behaviour of sequential programs, imposing constraints on how functions preserve bounded meets of approximants; although stable functions do not fully characterise sequentiality, because they admit gustave-style counterexamples, they remain an appropriate notion for studying the sensitivity of a program to partial data at a specific point [Amadio and Curien 1998]. Paul Taylor's characterisation of stable functions via local Galois connections on principal downsets provides a semantic underpinning for the reverse maps of Galois slicing [Taylor 1999]. The stable-function story has a tight connection to Galois slicing but a looser one to automatic differentiation, so we intend to develop it in separate work, relating Galois slicing to stable functions

rather than to differentiation. Such a development would also clarify the links to Differential Linear Logic [Ehrhard and Regnier 2006] and to Crubillé's identification of stable functions with power series [Crubillé 2018], relating the qualitative derivatives studied here to the analytic derivatives of that setting.

Shape Approximation. Lists are the only recursive datatype in our source language, so supporting general inductive and coinductive types is important future work, perhaps adapting Lucatelli Nunes and Vákár's [2023] automatic differentiation for $\mu\nu$-polynomial functors. Richer datatypes bring the question of *shape approximation*: approximating not only the values at the positions of a structure but the structure itself, such as the length of a list or the depth of a tree. Shape approximation features in earlier work on Galois slicing [Perera et al. 2012, 2016; Ricciotti et al. 2017], whose approximation lattices represent partially erased data, whereas our model holds the shape of each input fixed. Our Jacobians are spans, and polynomial functors are the shape-dependent generalisation of spans [Gambino and Kock 2013]; the finite polynomials form the Lawvere theory for commutative semirings, the algebra of our scalars. The corresponding datatypes are containers, whose derivative is the type of one-hole contexts, carrying a notion of linear map and a chain rule for both inductive and coinductive types [Abbott et al. 2005]. This already gives a set-theoretic account of shape approximation; weighting positions with our semiring scalars would extend our analyses to varying shapes.

Recursion and Partiality. This work treats only total programs, and makes no use of directed completeness or similar properties of domains. Extending to recursion and partiality would introduce a notion of undefinedness, and hence a definedness order to be reconciled with any approximation order on the same data. Non-termination and infinite data are both partiality phenomena, approximating unbounded computation by finite observation, so potentially the finite prefixes of an infinite or non-terminating computation would fall under the same extension.

Source-To-Source Translation Techniques. An interesting alternative to the denotational approach presented here, and to the trace-based approaches used in some implementations of provenance, would be to develop a source-to-source transformation, in direct analogy with the CHAD approach to automatic differentiation [Lucatelli Nunes and Vákár 2023; Vákár and Smeding 2022]. In their approach, forward and reverse-mode AD are implemented as compositional transformations on source code, guided by a universal property: they arise as the unique structure-preserving functors from the source language to a suitably structured target language formalised as a Grothendieck construction. Adapting this to our setting would allow the analysis to be "compiled in", avoiding the need for a custom interpreter and potentially exposing opportunities for optimisation.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2005. ∂ for Data: Differentiating Data Structures. *Fundamenta Informaticae* 65, 1-2 (2005), 1–28.

Roberto M. Amadio and Pierre-Louis Curien. 1998. *Domains and Lambda-Calculi.* Cambridge University Press, Cambridge.

Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. On the Limitations of Provenance for Queries with Difference. In *3rd Workshop on the Theory and Practice of Provenance (TaPP '11)*. USENIX.

Gérard Berry. 1979. *Modèles complètement adéquat et stables des lambda-calculs typé.* Ph. D. Dissertation. Université Paris VII.

Gérard Berry and Pierre-Louis Curien. 1982. Sequential algorithms on concrete data structures. *Theoretical Computer Science* 20 (07 1982), 265–321. https://doi.org/10.1016/S0304-3975(82)80002-9

R. Blute, Robin Cockett, and R. Seely. 2009. Cartesian differential categories. *Theory and Applications of Categories* 22 (01 2009), 622–672.

Joe Bond, Cristina David, Minh Nguyen, Dominic Orchard, and Roly Perera. 2025. Cognacy Queries over Dependence Graphs for Transparent Visualisations. In *Programming Languages and Systems*, Viktor Vafeiadis (Ed.). Springer Nature Switzerland, Cham, 144–171.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

Peter Buneman, Susan B. Davidson, and Dan Suciu. 1995. Programming Constructs for Unstructured Data. In *Proceedings of the Fifth International Workshop on Database Programming Languages (DBLP-5)*. Springer-Verlag, Berlin, Heidelberg, 12.

Aurelio Carboni, Stephen Lack, and R. F C Walters. 1993. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra* 84, 2 (3 feb 1993), 145–158. https://doi.org/10.1016/0022-4049(93)90035-R

James Cheney, Amal Ahmed, and Umut A. Acar. 2007. Provenance as Dependency Analysis. In *DBPL*. 138–152. https://doi.org/10.1007/978-3-540-75987-4_10

Robin Cockett and Geoffrey Cruttwell. 2014. Differential Structure, Tangent Structure, and SDG. *Applied Categorical Structures* 22 (04 2014). https://doi.org/10.1007/s10485-013-9312-0

Robin Cockett and Geoffrey Cruttwell. 2018. Differential bundles and fibrations for tangent categories. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 60 (2018), 10–92.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, 238–252. https://doi.org/10.1145/512950.512973

Raphaëlle Crubillé. 2018. Probabilistic Stable Functions on Discrete Cones are Power Series. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 275–284. https://doi.org/10.1145/3209108.3209198

Geoffrey Cruttwell and Jean-Simon Pacaud Lemay. 2024. Reverse Tangent Categories. In *32nd EACSL Annual Conference on Computer Science Logic (CSL 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 288)*, Aniello Murano and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:21. https://doi.org/10.4230/LIPIcs.CSL.2024.21

Geoffrey S. H. Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. 2022. Categorical Foundations of Gradient-Based Learning. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 1–28.

Johan de Kleer and John Seely Brown. 1984. A Qualitative Physics Based on Confluences. *Artificial Intelligence* 24, 1–3 (1984), 7–83. https://doi.org/10.1016/0004-3702(84)90037-7

Abbas Edalat and Reinhold Hackmann. 1998. A computational model for metric spaces. *Theoretical Computer Science* 193, 1–2 (feb 1998), 53–73. https://doi.org/10.1016/S0304-3975(96)00243-5

T. Ehrhard and L. Regnier. 2006. Differential interaction nets. *Theoretical Computer Science* 364, 2 (Nov. 2006), 166–195. https://doi.org/10.1016/j.tcs.2006.08.003

Conal Elliott. 2017. Compiling to categories. 1, ICFP, Article 27 (Aug. 2017), 27 pages. https://doi.org/10.1145/3110271

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.* 2, ICFP (July 2018). https://doi.org/10.1145/3236765

Marcelo Fiore and Alex Simpson. 1999. Lambda Definability with Sums via Grothendieck Logical Relations. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 147–161.

Nicola Gambino and Joachim Kock. 2013. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society* 154, 1 (2013), 153–192.

Floris Geerts and Antonella Poggi. 2010. On Database Query Languages for K-relations. *Journal of Applied Logic* 8, 2 (2010), 173–185. https://doi.org/10.1016/j.jal.2009.09.001

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press.

Todd J. Green, Zachary G. Ives, and Val Tannen. 2009. Reconcilable Differences. In *Proceedings of the 12th International Conference on Database Theory (ICDT '09)*. ACM, 212–224. https://doi.org/10.1145/1514894.1514920

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*. 31–40. https://doi.org/10.1145/1265530.1265535

Andreas Griewank. 1989. On Automatic Differentiation. In *Mathematical Programming: Recent Developments and Applications*, Masao Iri and Kunio Tanabe (Eds.). Springer.

Claudio Hermida. 1999. Some properties of Fib as a fibred 2-category. *Journal of Pure and Applied Algebra* 134, 1 (1999), 83–109. https://doi.org/10.1016/S0022-4049(97)00129-1

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Bjarni Jonsson and Alfred Tarski. 1951. Boolean Algebras with Operators. Part I. *American Journal of Mathematics* 73, 4 (1951), 891–939.

Martti Karvonen. 2020. Biproducts without pointedness. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* LXI (2020), 229–238. Issue 3.

Marie Morgane Kerjean and Pierre-Marie Pédrot. 2024. ∂ is for Dialectica. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Tallinn Estonia, 1–13. https://doi.org/10.1145/3661814.3662106

Kostas Kontogiannis. 2008. Challenges and opportunities related to the design, deployment and, operation of Web Services. In *2008 Frontiers of Software Maintenance*. 11–20. https://doi.org/10.1109/FOSM.2008.4659244

Benjamin Kuipers. 1986. Qualitative Simulation. *Artificial Intelligence* 29, 3 (1986), 289–338. https://doi.org/10.1016/0004-3702(86)90073-1

Seppo Linnainmaa. 1976. Taylor expansion of the accumulated rounding error. *BIT* 16, 2 (June 1976), 146–160. https://doi.org/10.1007/BF01931367

Fernando Lucatelli Nunes and Matthijs Vákár. 2023. CHAD for expressive total languages. *Mathematical Structures in Computer Science* 33, 4–5 (2023), 311–426. https://doi.org/10.1017/S096012952300018X

Simon Marlow et al. 2010. Haskell 2010 language report. http://haskell.org/onlinereport/haskell2010/.

Roly Perera. 2013. *Interactive Functional Programming*. Ph. D. Dissertation. University of Birmingham, Birmingham, UK. http://etheses.bham.ac.uk/4209/.

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional Programs That Explain Their Work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/2364527.2364579

Roly Perera, Deepak Garg, and James Cheney. 2016. Causally Consistent Dynamic Slicing. In *Concurrency Theory, 27th International Conference, CONCUR '16 (Leibniz International Proceedings in Informatics (LIPIcs))*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany. https://doi.org/10.4230/LIPIcs.CONCUR.2016.18

Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked Visualisations via Galois Dependencies. *Proc. ACM Program. Lang.* 6, POPL, Article 7 (2022), 29 pages. https://doi.org/10.1145/3498668

Fotis Psallidas and Eugene Wu. 2018. Provenance for Interactive Visualizations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '18)*. ACM. https://doi.org/10.1145/3209900.3209904

Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative Functional Programs That Explain Their Work. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 14 (2017), 28 pages. https://doi.org/10.1145/3110258

J. Rosický. 1984. Abstract tangent functors. *Diagrammes* 12 (1984), JR1–JR11.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988. *Learning representations by back-propagating errors*. MIT Press, Cambridge, MA, USA, 696–699.

Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2020. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. *International Journal of Computer Vision* 128, 2 (feb 2020), 336–359. https://doi.org/10.1007/s11263-019-01228-7

Jesse Sigal. 2024. *Automatic differentiation via effects and handlers*. Ph. D. Dissertation. University of Edinburgh. https://doi.org/10.7488/era/4642

Imre Simon. 1988. Recognizable Sets with Multiplicities in the Tropical Semiring. In *Mathematical Foundations of Computer Science (MFCS 1988) (Lecture Notes in Computer Science, Vol. 324)*. Springer, 107–120.

Jeffrey Siskind and Barak Pearlmutter. 2008. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation* 21 (12 2008), 361–376. https://doi.org/10.1007/s10990-008-9037-1

Bernd Steinbach and Christian Posthoff. 2017. *Boolean Differential Calculus*. Morgan & Claypool.

Paul Taylor. 1999. *Practical Foundations of Mathematics*. Cambridge University Press, Cambridge, UK.

Matthijs Vákár and Tom Smeding. 2022. CHAD: Combinatory Homomorphic Automatic Differentiation. *ACM Trans. Program. Lang. Syst.* 44, 3 (Aug. 2022). https://doi.org/10.1145/3527634

Mark D. Weiser. 1981. Program Slicing. In *ICSE*, Seymour Jeffrey and Leon G. Stucki (Eds.). IEEE Computer Society, 439–449. http://dblp.uni-trier.de/db/conf/icse/icse81.html#Weiser81