Galois Slicing as Automatic Differentiation

ANONYMOUS AUTHOR(S)

Galois slicing is a technique for program slicing for provenance, developed by Perera and collaborators. Galois slicing aims to explain program executions by demonstrating how to track approximations of the input and output forwards and backwards along a particular execution. In this paper, we explore an analogy between Galois slicing and differentiable programming, seeing the implementation of forwards and backwards slicing as a kind of automatic differentiation. Using the CHAD approach to automatic differentiation due to Vákár and collaborators, we reformulate Galois slicing via a categorical semantics. In doing so, we are able to explore extensions of the Galois slicing idea to quantitative interval analysis, and to clarify the implicit choices made in existing instantiations of this approach.

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.

Existing approaches to program slicing are often tied to particular programming languages or implementations. In this paper we develop a general categorical approach to program slicing, focusing on a particular technique called Galois slicing, where the set of slices of a given value form a lattice of approximations and the forward and backward slicing procedures generate Galois connections between these lattices. Our main contribution is that this approach can be seen as a generalised form of automatic differentiation, with slices of values playing the role of tangents. 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 Galois Program Slicing

Perera and collaborators introduced the idea of *Galois program slicing* as a particular conception of program slicing for provenance, described in several publications [Perera et al. 2012, 2016; Ricciotti et al. 2017]. Galois program slicing (hereafter simply *Galois slicing*) forms the basis of the open source data visualisation tool Fluid [Perera et al. 2025] that allows interactive exploration of programmatically generated visualisations.

At a high level, Galois slicing assumes that, for each possible value that may be input or output by a program, there exists a lattice of *approximations* of that value. For a particular run of a program

ACM XXXX-XXXX/2025/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that takes input x and produces output y, we also get a Galois connection between the lattice of approximations of x and the lattice of approximations of y. The right half of the Galois connection is the "forward direction" taking approximations of the input to approximations of the output; the left half of the Galois connection is the "backward direction" that takes approximations of the output to the least (i.e., most approximate) approximation of the input that gives rise to this output approximation. This becomes *program slicing* by including the source code of the program as part of the input; then, in the backward direction, the least approximation of the input required for an output approximation includes the least part of the program required as well.

Example 1.1. The following program is written in Haskell syntax [Marlow et al. 2010], using a list comprehension to filter a list of pairs of labels and numbers to those numbers with a given label, and then computing the sum of the numbers:

$$\begin{array}{l} \mathrm{query} :: \mathrm{Label} \longrightarrow [(\mathrm{Label}, \mathrm{Int})] \longrightarrow \mathrm{Int} \\ \mathrm{query}\ l\ db = \mathrm{sum}\ [n \mid (l', n) \leftarrow db, l \equiv l'] \end{array}$$

With $db = [(\mathrm{a}, 0), (\mathrm{b}, 1), (\mathrm{a}, 1)]$, we will have $query$ a db and $query$ b db both evaluating to 1.

Now suppose that for a given run of the program, we are interested in which of the numerical parts of the input are used to compute the output for the query parameters $l = \mathrm{a}$ and $l = \mathrm{b}$. We can use Galois slicing to do this. We arrange for the approximations of the input to form the following lattice, where the actual piece of data is at the top and information lost by approximation is represented by $\perp$s:

$$\begin{array}{ccc} & [(\mathrm{a}, 0), (\mathrm{b}, 1), (\mathrm{a}, 1)] & \\ [(\mathrm{a}, \perp), (\mathrm{b}, 1), (\mathrm{a}, 1)] & [(\mathrm{a}, 0), (\mathrm{b}, \perp), (\mathrm{a}, 1)] & [(\mathrm{a}, 0), (\mathrm{b}, 1), (\mathrm{a}, \perp)] \\ [(\mathrm{a}, \perp), (\mathrm{b}, \perp), (\mathrm{a}, 1)] & [(\mathrm{a}, \perp), (\mathrm{b}, 1), (\mathrm{a}, \perp)] & [(\mathrm{a}, 0), (\mathrm{b}, \perp), (\mathrm{a}, \perp)] \\ & [(\mathrm{a}, \perp), (\mathrm{b}, \perp), (\mathrm{a}, \perp)] & \end{array}$$

In both runs of the program, the output approximation lattice looks like this, where 1 is the actual data point that was returned, and $\perp$ indicates that we are approximating this piece of data away:

$$\begin{array}{c} 1 \\ | \\ \perp \end{array}$$

These are not the only choices of approximation lattices that we could have made. For the input, we have chosen a lattice that allows us to "forget" (approximate away) numbers in the input, but not the labels or the structure of the list itself. However, other choices are also useful. Indeed, one of the aims of this work is to clarify how to choose an approximation structure appropriate for different tasks by use of type information. We elaborate on this further in §3.3.

Galois slicing associates with each run of the program a Galois connection telling us how the inputs and outputs are related in that run. The backwards portion $\partial(query\ l)_r$ tells us, given an approximation of the output, what the least approximation of the input is needed to generate that output. In the case of the two runs considered in this example, if we say we are not interested in the output by feeding in the least approximation $\perp$, then we find that we only need the least approximation of the input:

$$\partial(query\ l\ db)_r(\perp) = [(\mathrm{a}, \perp), (\mathrm{b}, \perp), (\mathrm{a}, \perp)]$$

for both $l = \mathrm{a}$ and $l = \mathrm{b}$. If instead we take the greatest approximation of the output (i.e., the output "1" itself), then the two query runs' backwards approximations return different results:

$$\partial(\mathit{query}\,\mathrm{a}\;\mathit{db})_r(1) = [(\mathrm{a},0),(\mathrm{b},\bot),(\mathrm{a},1)]$$
$$\partial(\mathit{query}\,\mathrm{b}\;\mathit{db})_r(1) = [(\mathrm{a},\bot),(\mathrm{b},1),(\mathrm{a},\bot)]$$

Pieces of the input that were *not* used are replaced by $\bot$. As we expect, the run of the query with label a depends on the entries in the database labelled with a, and likewise for the run with label b.

In this case, the forwards portion of the Galois connection tells us, for each approximation of the input, whether or not it is sufficient to compute the output. If we provide insufficient data to compute the output, then we will get an underapproximated output. Here for example we will find that $\partial(\mathit{query}\,\mathrm{a})_f([(\mathrm{a},0),(\mathrm{b},\bot),(\mathrm{a},\bot)]) = \bot$ because we need all the values associated with the label a to compute their sum.

In a simple query like this, it is easy to work out the dependency relationship between the input and output. However, the benefit of Galois slicing, and other language-based approaches, is that it is *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 Galois Slicing and Automatic Differentiation

Previous work on Galois slicing used a special operational semantics to generate a trace of each execution, and then uses that trace to compute the Galois connections described above, by re-running forwards or backwards over the trace. It would be useful to have a denotational account of Galois slicing as well, especially if we could provide a semantics where the backwards analysis is baked in, rather than provided by a separately defined "backwards evaluation" operation. Our thesis, developed in §2 and §3 is that there is a close analogy between Galois slicing and *automatic differentiation* for differentiable programs [Elliott 2018; Siskind and Pearlmutter 2008; Vákár and Smeding 2022], which points to a way to develop such an approach. We have already hinted at this in the description above, but let us now make it explicit.

- For Galois slicing, we assume that every value has an associated lattice of *approximations*. For differentiable programs, every point has an associated vector space of *tangents*.
- For Galois slicing, every program has an associated forward approximation map that takes approximations forward from the input to the output. This map *preserves meets*. 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 Galois slicing, every program has an associated backward approximation map that takes approximations of the output back to least approximations of the input. This map *preserves joins*. For differentiable programs, every program has a reverse derivative that takes tangents of the output to tangents of the input. This map is again *linear*.
- For Galois slicing, the forward and backward approximation maps are related by being a Galois connection. For differentiable programming, the forward and reverse derivatives are related by being each others' transpose.

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

1.3 Outline and Contributions

Galois slicing, as any program slicing technique, essentially rests on an analysis of how programs intensionally explore their input, in addition to their extensional behaviour. Such analysis has been carried out over many years in Domain Theory. In §2, we use ideas from Berry [1979]'s *stable domain theory* and develop an analogy between stable functions and smooth functions from mathematical analysis, where stable functions provide a kind of semantic provenance analysis. In §3, we abstract from stable functions using Vákár *et al.*'s CHAD framework [Lucatelli Nunes and Vákár 2023; Vákár and Smeding 2022] to build models of a higher-order language that automatically compute slices. We apply this to a concrete higher-order language in §4 and demonstrate the use of the model on variations of Example 1.1, highlighting the flexibility of our approach. In particular, we show how type structure can be used to control the approximation lattices associated with data points, something that was "hard coded" in previous presentations of Galois slicing. We prove two correctness properties in §5, relating the higher-order interpretations to first-order ones, proving the crucial Galois connection property. §6 and §7 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 §4.3. Please consult the file `everything.agda` in the supplementary material.

2 Approximations as Tangents

We motivate our approach by showing how to combine ideas from differential geometry and stable domain theory to reconstruct the ideas of Galois slicing 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 Stable Functions as Differentiable Functions

Our thesis is that Galois slicing is a generalised form of differentiable programming, where tangents are not linear approximations of curves but instead are qualitative information approximations of elements. Smooth functions in this setting are Berry's *stable functions* [Berry 1979; Berry and Curien 1982]. We now introduce these concepts and how they relate to Galois slicing.

2.2.1 Domains as a Qualitative Theory of Approximation. Domain theory is a method for defining the semantics of programs that handle infinite data such as functions or infinite streams. Domains are certain partially ordered sets where the ordering denotes a relationship of qualitative information content: if $x \sqsubseteq y$, then y may contain more information than x. For example, if x and y are functions, then y may be defined at more values than x. Infinite objects are understood in terms of their approximations in this sense, and domains are assumed to be closed under least upper bounds (lubs) of directed sets, meaning that any internally consistent collection of elements has a "completion" that contains all the information covered by the set. Programs are interpreted as monotone functions that preserve directed lubs. Monotonicity captures the idea that if the input gets more defined, then the output can get more defined. Preservation of lubs, or *continuity*, states that a function interpreting a program cannot act inconsistently on approximations and their completion, which corresponds to the intuitive idea that a function that is computable cannot look at a non-finite amount of input to produce a finite output. Abramsky and Jung [1995] provide a comprehensive introduction to domain theory.

For the purposes of Galois slicing, we are interested in using approximations not to model computation on infinite objects, but instead for revealing how programs explore their inputs when producing parts of their output. Therefore, we ignore completeness properties of the partially ordered sets we consider.

2.2.2 Bounded Meets and Conditional Multiplicativity. When giving a denotational semantics for sequential programming languages, Scott-continuous functions are too permissive. Famously, Plotkin [1977b]'s Parallel OR (Example 2.5, below) is continuous but does not explore its input in a way consistent with a sequential implementation. Continuous functions can explore their input in a non-deterministic way as long as the result is deterministic. This non-determinism results in functions whose output cannot be assigned a unique minimal input that accounts for it. Therefore, continuous functions are in general incompatible with the central idea in Galois slicing that we should be able to identify a *minimal* part of the input that leads to a part of the output.

Stability is a property that can be required of monotone functions, that was invented by Berry [1979] in an attempt to capture sequentiality. This was unsuccessful (see the gustave function in Example 2.5), but we will see now how it is closely related to the problem of computing the forward and backwards maps of approximations needed in Galois slicing. A textbook description of stable functions in the context of domain theory is given by Amadio and Curien [1998, Chapter 12]. We start with a property that is weaker than stability, but easier to motivate in connection with derivatives of smooth functions.

In a partially ordered set X, for any $x \in X$ the set of elements below x, $\downarrow(x) = \{x' \mid x' \sqsubseteq x\}$, is itself a partially ordered set. These approximations of x we will think of as "tangents" at x, and the whole set $\downarrow(x)$ as the "tangent space". Tangent spaces are vector spaces, so in the partially ordered setting we take elements of $\downarrow(x)$ to be approximations of processes defined at x. As we can add tangents, we assume we can take meets of approximations in $\downarrow(x)$:

Definition 2.1. A *bounded meet poset* is a partially ordered set X where for every $x \in X$, $\downarrow(x)$ is a meet semilattice, with x as the top element.

For the approximation version of the forward derivative of f at x, we take f's restriction to $\downarrow(x)$, taking approximations of the input to approximations of the output. Matching the linearity of the forward derivative of smooth functions, we require that these restrictions preserve meets. This is exactly the definition of *conditionally multiplicative* function from Berry [1979]:

Definition 2.2. A *conditionally multiplicative* (cm) function $f : X \to Y$ is a monotone function such that for all $x \in X$, the restriction $f_x : \downarrow(x) \to \downarrow(f(x))$ preserves meets.

THEOREM 2.3. *Bounded meet lattices and conditionally multiplicative functions form a category* **CM**. *This category has products, coproducts, and exponentials.*

PROOF. *(Formalised in Agda).* See Amadio and Curien [1998, Theorem 12.1.9] for the case when the posets are also cpos. The crucial technical step, identified by Berry [1979], is that the ordering on conditionally multiplicative functions is not the extensional ordering ($f \sqsubseteq_{\mathrm{ext}} g$ iff $\forall x. f(x) \sqsubseteq g(x)$) but instead the stable ordering: $f \sqsubseteq_{\mathrm{st}} g$ iff $f \sqsubseteq_{\mathrm{ext}} g$ and $\forall x, x'. x \sqsubseteq x' \Rightarrow f(x) = f(x') \wedge g(x)$. □

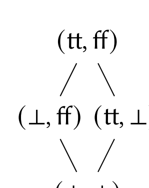
Remark 2 (Chain Rule). It is almost a triviality at this point, but the crucial point is that, for any cm functions $f : X \to Y$ and $g : Y \to Z$, the restriction maps ("forward derivatives") compose according to the chain rule from Remark 1:

$$(g \circ f)_x = g_{f(x)} \circ f_x$$

We will see this phenomenon repeated in the definition of stable functions below.

Example 2.4 (Conditionally Multiplicative Functions). To see the effect of conditional multiplicativity, consider several ways of defining the OR on the lifted booleans $\mathbb{B}_\perp$. Two functions that are cm are the strict and left-short-circuiting ORs[1]:

$$\begin{aligned}
\mathrm{strictOr}(\mathrm{tt}, \mathrm{tt}) &= \mathrm{tt} \\
\mathrm{strictOr}(\mathrm{tt}, \mathrm{ff}) &= \mathrm{tt} \\
\mathrm{strictOr}(\mathrm{ff}, \mathrm{tt}) &= \mathrm{tt} \\
\mathrm{strictOr}(\mathrm{ff}, \mathrm{ff}) &= \mathrm{ff} \\
\mathrm{strictOr}(\perp, _) &= \perp \\
\mathrm{strictOr}(_, \perp) &= \perp
\end{aligned}
\qquad
\begin{aligned}
\mathrm{shortCircuitOR}(\mathrm{tt}, _) &= \mathrm{tt} \\
\mathrm{shortCircuitOR}(\mathrm{ff}, x) &= x \\
\mathrm{shortCircuitOR}(\perp, _) &= \perp
\end{aligned}$$



[1]The clauses in these examples are shorthand for the graph of the function. They are not to be understood as pattern matching clauses in a language like Haskell, where it is not possible to match on $\perp$.

In the poset $\mathbb{B}_\perp^2$, a typical poset of approximations of a fully defined element is shown to the right. For strictOr, any approximation that isn't the fully defined input is mapped to $\perp$, while shortCircuitOr maps the partially defined $(\mathrm{tt}, \perp)$ to tt. Thus, even though these functions operate identically on fully defined inputs, they differ in their *derivatives* on partially defined input, exposing how they explore their arguments differently. That they are cm can be checked by examining their restrictions' behaviour. If we take the approximations $(\perp, \mathrm{ff})$ and $(\mathrm{tt}, \perp)$, then their meet is $(\perp, \perp)$; strictOr maps all three elements to $\perp$, so is cm here since $\perp \wedge \perp = \perp$; and shortCircuitOR has $(\perp, \mathrm{ff}) \mapsto \perp$ and $(\mathrm{tt}, \perp) \mapsto \mathrm{tt}$, the meet of which is $\perp = \mathrm{shortCircuitOR}(\perp, \perp)$. Other combinations can be checked similarly.

Example 2.5 (A non-Conditionally Multiplicative Function). A function that is not cm is Plotkin's Parallel OR [Plotkin 1977a], which short-circuits in both arguments. It returns tt if either argment is tt even if the other argument is not defined:

$$
\begin{aligned}
\mathrm{parallelOR}(\mathrm{tt}, \_\,) &= \mathrm{tt} \\
\mathrm{parallelOR}(\_\,, \mathrm{tt}) &= \mathrm{tt} \\
\mathrm{parallelOR}(\mathrm{ff}, \mathrm{ff}) &= \mathrm{ff} \\
\mathrm{parallelOR}(\perp, \perp) &= \perp
\end{aligned}
$$

We have $\mathrm{parallelOR}(\mathrm{tt}, \perp) \wedge \mathrm{parallelOR}(\perp, \mathrm{tt}) = \mathrm{tt} \wedge \mathrm{tt} = \mathrm{tt}$ but $\mathrm{parallelOR}((\mathrm{tt}, \perp) \wedge (\perp, \mathrm{tt})) = \mathrm{parallelOR}(\perp, \perp) = \perp$, so it is not cm.

Parallel OR is famous because it is not *sequential*, meaning intuitively that it cannot be implemented without running the two arguments in parallel to see if one of them returns tt. The fact that it exists in the standard domain theoretic semantics of PCF means that this semantics is incomplete for reasoning about observational equivalence in PCF. Since Parallel OR is not cm, one might hope that cm-ness is enough to capture sequentiality, and hence potentially give a fully abstract model of PCF. However, the following ternary function $\mathbb{B}_\perp^3 \to \{\top, \perp\}$ is cm but admits no sequential implementation that fixes an order that the arguments are examined in:

$$
\begin{aligned}
\mathrm{gustave}(\mathrm{tt}, \mathrm{ff}, \_\,) &= \top \\
\mathrm{gustave}(\mathrm{ff}, \_\,, \mathrm{tt}) &= \top \\
\mathrm{gustave}(\_\,, \mathrm{tt}, \mathrm{ff}) &= \top \\
\mathrm{gustave}(\_\,, \_\,, \_\,) &= \perp
\end{aligned}
$$

Due to the way that the cases are defined, there is no way of constructing a pair of approximations for which preservation of their meet does not hold. In terms of derivatives, this makes sense in that we are only concerned about the intensional behaviour of a function at a point and its approximations. Parallel OR has two incompatible approximation behaviours at the point $(\mathrm{tt}, \mathrm{tt})$. The gustave function does have consistent behaviour at each approximation for each point.

Example 2.6 (Intervals and Maximal Elements). The set of intervals $\{[l, u] \in \mathbb{R} \times \mathbb{R} \mid l \leq u\}$ ordered by reverse inclusion forms a (Scott) domain [Scott 1970]. The set of maximal elements is exactly $\mathbb{R}$. This domain has been proposed as a model of approximate real number computation [Escardó 1996]. The information approximation reading is intuitive: as intervals move up the order they become tighter, containing more information about the number they are approximating.

Given this reading, it makes sense to wonder if we can use interval approximations as "information tangents" of real numbers, where derivatives take approximating intervals to approximating intervals. Since intervals form a Scott domain, they are bounded complete and hence have bounded meets. However, the addition function on intervals, $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$, is not conditionally multiplicative, as can be easily checked.

A solution is to use the set of intervals with nominated points that they are approximating: $\{[l, x, u] \mid l \leq x \leq u\}$. The ordering now is that $[l_1, x_1, u_1] \sqsubseteq [l_2, x_2, u_2]$ iff $x_1 = x_2$ and $l_1 \leq$

l_2 and $u_2 \leq u_1$. Consequently, the maximal elements are $[x, x, x]$, recovering $\mathbb{R}$ again, but the approximations of each number all form independent sub-lattices. Addition is defined as $[l_1, x_1, u_1] + [l_2, x_2, u_2] = [(l_1 + x_2) \sqcap (l_2 + x_1), x_1 + x_2, (u_1 + x_2) \sqcup (u_2 + x_1)]$, which is conditionally multiplicative. Note how this definition bears a resemblance to the product rule for derivatives, with (in the lower end of the interval) $\sqcap$ replacing $+$.

This example is important to us because it shows that (for total programs) we are separately interested in the maximal elements and their approximations, and that approximations of each maximal element may have to be considered separately.

Relatedly, Edalat and Hackmann [1998] proposed a Scott domain of formal balls on a metric space, where again the maximal elements are the points of the original space. More generally, Gierz et al. [2003, Section V-6] describe *domain environments*, which are domains whose maximal elements are exactly the points of a topological space. We are not aware of any work linking domain environments in general to stable functions. It would be interesting to see whether the situation for intervals, where approximations must be relative to a nominated point, repeats in general when considering conditionally multiplicative functions.

Given these examples, conditional multiplicativity seems to be a reasonable analogue to functions with a well-defined notion of derivative. For Galois slicing we also require an analogue to the reverse derivative, where we map approximations backwards to give the least approximation of the input for a given approximation of the output. In the case of smooth functions, we are always guaranteed a reverse derivative. However, there is not always a best way to map approximations backwards for cm functions, as the following example shows.

Example 2.7 (Is Conditional Multiplicativity Enough?). An example that is conditionally multiplicative, but does not admit a backwards map of approximations (from Amadio and Curien [1998, just before Lemma 12.2.3], originally due to Berry) is given by $\mathrm{unstable} : D \longrightarrow \{\bot \sqsubseteq \top\}$, where $D = \bot \sqsubseteq \cdots \sqsubseteq n \sqsubseteq \cdots \sqsubseteq 1 \sqsubseteq 0$, as $\mathrm{unstable}(\bot) = \bot$ and $\mathrm{unstable}(n) = \top$. This is monotone, and preserves meets in every $\downarrow(x)$. But there is no "best" (i.e., least) input that gives us any finite output.

2.2.3 *Stable functions and L-posets.*

In light of the Example 2.7, we turn to Berry [1979]'s definition of *stable function* that requires the existence of a reverse mapping directly, even without assuming that any meets exist:

Definition 2.8 (Stable function). Let $f : X \longrightarrow Y$ be a monotone function between posets X and Y. The function f is *stable* if for all $x \in X$ and $y \leq f(x)$:

(1) (Existence) there exists an $x_0 \leq x$ such that $y \leq f(x_0)$, and
(2) (Minimality) for any $x_0' \leq x$ such that $y \leq f(x_0')$ then $x_0 \leq x_0'$.

Example 2.9.

(1) The function strictOr is stable. For example, for the input-output pair $(\mathrm{tt}, \mathrm{ff}) \mapsto \mathrm{tt}$, the minimal input that gives this output is exactly $(\mathrm{tt}, \mathrm{ff})$. If we take the approximation $\bot \leq \mathrm{tt}$ of the output, then the corresponding minimal input is $(\bot, \bot)$. The function shortCircuitOR is also stable. For the input-output pair $(\mathrm{tt}, \mathrm{ff}) \mapsto \mathrm{tt}$, the minimal input that gives this input is $(\mathrm{tt}, \bot)$, indicating that the presence of ff in the second argument was not necessary to produce this output. As with strictOr, the minimal input required to produce the output $\bot \leq \mathrm{tt}$ is again $(\bot, \bot)$.
(2) The parallelOR function is not stable. For the input-output pair $(\mathrm{tt}, \mathrm{tt}) \mapsto \mathrm{tt}$, there is no one minimal input that produces this output. We have both $\mathrm{parallelOR}(\mathrm{tt}, \bot) = \mathrm{tt}$ and $\mathrm{parallelOR}(\bot, \mathrm{tt}) = \mathrm{tt}$, which are incomparable and their greatest lower bound $(\bot, \bot)$ gives the output $\bot$.

(3) The gustave function is stable. Despite there being no one minimal input that achieves the output $\top$, each of the minimal inputs that can achieve this output are pairwise incomparable, so for each specific input that gets output $\top$ there is a unique minimal input that achieves it (listed in the first three lines of the definition). In terms of Galois slicing, the gustave function does not present a problem; for any particular run (i.e., input $\mapsto$ output pair), there is an unambiguous minimal input that achieves the output, no matter that it was not achieved by a sequential processing of the input.
(4) As discussed above, unstable is not stable, but is conditionally multiplicative.
(5) The addition function on intervals with nominated points in Example 2.6 is stable. Given the input $[l_1, x_1, u_1], [l_2, x_2, u_2]$ and an approximation $[l, x_1 + x_2, u]$ of the output, the minimal approximations of the input are $[l - x_2, x_1, u - x_2], [l - x_1, x_2, u - x_1]$. We can read this as saying if the output was the maximal element $x_1 + x_2$ but we only require the output to be in the range $[l, x_1 + x_2, u]$, then we can obtain intervals containing the input values that are enough to obtain the desired output approximation *assuming that the other input is kept the same*. Note the analogy to partial derivatives in multi-variable calculus, where the derivative is computed in each variable independently.

Stability has an alternative definition in terms of Galois connections, which will be more useful for what follows. This characterisation is due to Taylor [1999]. We first define Galois connections, which we used informally in Example 1.1.

Definition 2.10 (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.

Lemma 2.11. *A monotone function $f : X \to Y$ is stable if and only if for all $x \in X$, the restriction of $f_x : {\downarrow}(x) \to {\downarrow}(f(x))$ has a left Galois adjoint.*

Proof. If f is stable, then define a left adjoint $f_x^* : {\downarrow}(f(x)) \to {\downarrow}(x)$ by setting $f_x^*(y)$ to be the minimal x_0 required by stability. This is monotone: if $y \leq y'$, then we know that $y \leq y' \leq f(f_x^*(y'))$ by the definition of f_x^*, so $f_x^*(y) \leq f_x^*(y')$ by minimality of $f_x^*(y)$. For the adjointness, let $x' \leq x$ and $y \leq f(x)$. Then if $f_x^*(y) \leq x'$, we have $y \leq f(f_x^*(y)) \leq f(x')$ by monotonicity of f and the first part of stability. In the other direction, if we have $y \leq f(x')$, then by uniqueness we have $f_x^*(y) \leq x'$.

If, for every x, f_x has a left adjoint f_x^*, then for any x', y we have $y \leq f_x(x') \Leftrightarrow f_x^*(y) \leq x'$. So $f_x^*(y)$ is the element that satisfies $y \leq f(f_x^*(y))$, and it is minimal since if $y \leq f_x(x'_0)$ then $f_x^*(y) \leq x'_0$. □

Even though stable functions can be defined on any partially ordered set, in light of the analogy with tangent spaces it makes sense to require that meets, preserved by forward approximation maps, and joins, preserved by backwards approximation maps, exist:

Definition 2.12. An *L-poset* is a partially ordered set X such that for every $x \in X$, the principal downset ${\downarrow}(x)$ is a bounded lattice (i.e., have all finite meets and joins).

This lemma is an instance of standard facts about Galois connections preserving meets and joins:

Lemma 2.13. *For L-posets X and Y, a stable function $f : X \to Y$ preserves meets in its forward part f_x and joins in its reverse part f_x^*.*

The converse to this lemma (that functions that preserve meets in their forward part have a left Galois adjoint) is not true, as was demonstrated by the non-stable function in Example 2.7. In the case when the posets $\downarrow(x)$ are *complete*, and f_x preserves infinitary meets, then we are guaranteed a left Galois adjoint. In that example, the infinite set $\{0, 1, 2, \ldots, n, n-1, \ldots\}$ not including $\bot$ of approximations of 0 does not have a greatest lower bound, so the order is not complete.

THEOREM 2.14. *L-posets and stable functions form a category* **Stable** *with products and coproducts.*

Remark 3 (Chain Rule). As for the "forward derivatives" of conditionally multiplicative functions, the forward and backwards parts of a stable function compose according to the chain rule (c.f. Remark 1):

- $(g \circ f)_x = g_{f(x)} \circ f_x : \downarrow(x) \multimap \downarrow(g(f(x)))$
- $(g \circ f)^*_x = f *_x \circ g^*_{f(x)} : \downarrow(g(f(x))) \multimap \downarrow(x)$

We will use this property in Proposition 3.3 to show that **Stable** embeds into our category of sets-with-approximation.

The category of L-posets and stable functions is not cartesian closed. To make it so, we would need to require that the principal downsets $\downarrow(x)$ are *complete* lattices. Amadio and Curien [1998, Theorem 12.5.10] details the proof. Intuitively, to generate the best approximation of an input value for a function, we need to take the infimum over all possible input values.

Our goal is to model a higher-order language suitable for writing queries on databases as in Example 1.1, so why should we not just take complete L-posets and stable functions as our model of Galois slicing? We have two reasons for moving to a different model in §3:

(1) Even without completeness, in bounded meet posets and L-posets values and their approximations live in the same set. However, in the *total* query language we wish to model in §4, we are not directly interested in the behaviour of programs on approximations as we would be for partial programs with general recursion. (Moreover, in the light of Example 2.6 it is not clear whether approximations for partiality and approximations for stability ought to be the same thing. We discuss this further in §7.) In Example 2.6, maximal elements could be taken to be the "proper values". One idea is to restrict to conditionally multiplicative or stable functions that preserve maximal elements. However, this idea fails at higher order: functions that take maximal elements to maximal elements are not themselves maximal elements. We could devise a category of L-posets with totality predicates (which would pick out maximal elements at first-order) and totality preserving functions, but we prefer a more direct method of separating values proper from their approximations using the *Category of Families* construction as we explain in §3.1.

(2) A more practical reason is that we wish to formalise our construction in the proof assistant Agda [Norell 2007] in order to get an *executable* model of the language in §4. Agda's type theory is both predicative and constructive. Predicativity means that it does not have complete lattices in the classical sense: for a type X we can only get suprema and infima of families in a lower universe level than X. This is not necessarily a problem, as de Jong and Escardó [2021] show how to develop a large amount of domain theory in a predicative setting. However, due to constructivity the lifting construction (due to Escardó and Knapp [2017]) requires a proof of definedness to extract a value. Since we are modelling a total language, we prefer to have a model that computes directly.

2.3 Summary

We have seen that Berry [1979]'s theory of stable functions between suitable partial orders can be seen as form of differentiability with forward and reverse derivatives. In the next section, we describe a model based on these ideas suitable for constructing executable models of total higher order languages.

We end this section with a conjecture. Although we have argued above that there is an analogy between stable functions and smooth functions, we have not stated any mathematical theorems substantiating this. *Tangent Categories* [Cockett and Cruttwell 2014, 2018] are a categorical axiomatisation of the properties of manifolds and smooth functions in terms of the presence of tangent bundles $T(X)$ for every object X, forward derivatives and additivity of tangents. They generalise Cartesian Differential Categories [Blute et al. 2009], which are an axiomatisation of Euclidean spaces and smooth functions.

Conjecture 2.15. *(1) Bounded meet posets and conditionally multiplicative functions form a Tangent category where the tangent bundle $T(X) = \{(x, x') \mid x' \leq x\}$ and addition of tangents is given by meets. (2) L-posets and stable functions form a* reverse Tangent category *[Cruttwell and Lemay 2024].*

As well as codifying exactly what we mean by differentiable structure on partially ordered sets, proving this conjecture would also tell us what higher derivatives mean in this context as well, something that we have not considered above. We will extend this conjecture to our refined model of lattice approximated sets in §3.4.

3 Models of Galois Slicing for a Total Language

The previous section concluded that L-posets and stable functions give a model of Galois slicing analogous to manifolds and smooth functions. However, we noted a conceptual shortcoming of this model, for the purposes of modelling total computations, that proper values and their approximations live in the same category. In this section, we propose a model for total Galois slicing based on the *Category of Families* construction. This construction, and the more general Grothendieck construction, has been previously used by Vákár and collaborators [Vákár and Smeding 2022] to model automatic differentiation for higher-order programs on the reals. We reuse some of their results, and discuss the commonalities as we go.

3.1 The Category of Families Construction

L-posets are partially ordered sets where every principal downset $\downarrow(x)$ is a bounded lattice of approximations/tangents. As we explained in §2.2.3, the shortcoming of this setup is that proper values and their approximations live in the same set. We fix this by changing our model to one where we have sets X of values, and for each $x \in X$, a bounded lattice $\partial X(x)$ of approximations of x. This construction is an instance of the general *Category of Families* construction:

Definition 3.1. Let C be a category. The *Category of Families* over C, $\mathbf{Fam}(C)$, 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, Remark 2, and Remark 3. 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 3.2. *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 conjecture at the end of this section that the analogous construction in our partially ordered setting does not.

For the categories **CM** and **Stable**, we pick the appropriate categories of partial orders and monotone maps:

(1) The category **LatGal** has bounded lattices as objects and Galois connections as morphisms, with the right adjoint going in the "forward" direction. The category $\mathbf{Fam}(\mathbf{LatGal})$ is our preferred model for total functions with Galois slicing. We explore some specific examples in this category in §3.3.
(2) The category **MeetSLat** has meet-semilattices with top as objects and monotone finite meet preserving functions as morphisms. The category $\mathbf{Fam}(\mathbf{MeetSLat})$ provides a model of total functions with "forward derivatives" only.
(3) The category **JoinSLat** has join-semilattices with bottom as objects and monotone finite join preserving functions as morphisms. The category $\mathbf{Fam}(\mathbf{JoinSLat}^{\mathrm{op}})$ provides a model of total functions with "backwards derivatives" only.

We can get an analogous result to Proposition 3.2 for L-posets and stable maps:

Proposition 3.3. *There is a faithful functor* $\mathbf{Stable} \to \mathbf{Fam}(\mathbf{LatGal})$ *that maps an L-poset* X *to* $(X, \lambda x.\downarrow(x))$ *and stable functions* f *to* $(f, \lambda x.(f_x, f_x^*))$. *Likewise, there is a faithful functor* $\mathbf{CM} \to \mathbf{Fam}(\mathbf{MeetSLat})$.

Despite its similarity, this proposition has a lesser status than Proposition 3.2 because it is not clear that the category **Stable** (or **CM**) is a canonical definition of approximable sets and functions with approximation derivatives, as we discussed at the end of §2.2.3. Our working hypothesis is that $\mathbf{Fam}(\mathbf{LatGal})$, where values and their approximations are separated by construction, is a natural model of semantic Galois slicing in a total setting, though we note some shortcomings in Remark 6. We now investigate some categorical properties of this category, with a view to modelling a higher-order total programming language in §4.

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

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

Proposition 3.4. *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 3.5. *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, $\mathbf{Fam}(\mathbf{LatGal})$ has coproducts and finite products, because **LatGal** has products. Similarly for $\mathbf{Fam}(\mathbf{MeetSLat})$. As we shall see below, the products in **LatGal** (and **MeetSLat** and **JoinSLat**) are also coproducts, which is essential to obtaining cartesian closure.

3.2.2 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 $\mathbf{Fam}(C)$. This relies on the definition of *biproducts*, which we discuss below in §3.2.3.

Theorem 3.6 ([Lucatelli Nunes and Vákár 2023]). *(Formalised in Agda). If C has biproducts (Definition 3.9) and all products, then $\mathbf{Fam}(C)$ is cartesian closed[2]. On objects, the internal hom can be given by:*

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

The **Set**-component of $(X, \partial X) \to (Y, \partial Y)$ consists of exactly the morphisms of $\mathbf{Fam}(C)$, rephrased into a single object. When $C = \mathbf{FDVect}$, these are functions with an associated linear map at every point, and when $C = \mathbf{LatGal}$, these are functions with an associated Galois connection 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 **MeetSLat** satisfies the hypotheses of Theorem 3.6, so:

Corollary 3.7. $\mathbf{Fam}(\mathbf{MeetSLat})$ *is cartesian closed and has all coproducts.*

Unfortunately, neither **LatGal** nor **FDVect** satisfy the hypotheses of this theorem, because neither of them have infinite products. We will consider ways to rectify this below in §3.2.4.

Remark 4. 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:

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

However, for the purposes of modelling differentiable programs, this is fatally flawed in that neither **LatGal** nor **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, as we shall see below. We will implicitly use Hermida's construction in our definability proof in §5, where we use a logical relations argument to show that every morphism definable in the higher order language is also first-order definable.

3.2.3 CMon-Categories and 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 shorter definition in categories enriched in commutative monoids:

[2] 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.

Definition 3.8. 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 3.9. In a **CMon**-category a biproduct is an object $X \oplus Y$ together with morphisms

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

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 3.10.

(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 3.11. 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 **LatGal**, right adjoints are summed using meets and left adjoints are summed using joins. The zero maps are given by the constantly $\top$ and constantly $\bot$ functions respectively. Products are given by the cartesian product of the underlying set and the one-element lattice for the terminal/initial/zero object.

(3) **MeetSLat** and **JoinSLat** are both **CMon**-enriched and have finite products similar to **LatGal**.

Remark 5. 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 4.

3.2.4 Discrete Completeness. The second hypothesis of Theorem 3.6 is that the category C has all (i.e., infinite) products. This is required to gather together tangents for all of the points in the domain of the function. Unfortunately, neither **FDVect** nor **LatGal** is complete in this sense.

In the case of **FDVect**, the solution is to expand to the category of all vector spaces **Vect**, where infinite direct products exist. Note that these infinite products are not biproducts because the vector space operations themselves are finitary. This is the solution that Vákár and Smeding [2022] use for the semantics of forward (**Fam**(**Vect**)) and reverse (**Fam**($\mathbf{Vect}^{\mathrm{op}}$)) automatic differentiation for higher order programs. Since the forward and reverse derivatives of a smooth map are intrinsically defined, Vákár and Smeding [2022]'s correctness theorem shows that, for programs with first-order type, the interpretation in **Fam**(**Vect**) correctly yields the forward derivative of the defined function on the reals (and reverse derivative for **Fam**($\mathbf{Vect}^{\mathrm{op}}$)).

For **LatGal** we could expand to the category of complete lattices and Galois connections between them. From a classical mathematical point of view, this would give a model of Galois slicing that would be suitable for reasoning about programs' behaviour and their forward and backward approximations. However, in terms of building an executable model inside the Agda proof assistant, and with an eye toward implementation strategies, we seek a finitary solution. (Note that the solution of moving to complete lattices is very different to moving to arbitrary dimension vector spaces: in the former we have infinitary operations, while the latter still has only finitary operations.)

We will avoid the need for infinitary operations by separating the forward and backward parts of the Galois connections to act independently by moving to the product category $\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}}$. Objects in this category consist of *separate* meet- and join-semilattices and potentially unrelated forward meet-preserving and backward join-preserving maps. We first check that this category satisfies the hypotheses of Theorem 3.6:

Proposition 3.12. $\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}}$ *has biproducts and all products.*

Proof. **MeetSLat** and **JoinSLat** are both **CMon**-enriched and have finite products, as noted above. The opposite of a category with biproducts also has biproducts (by swapping the injections i and projections p), and products of categories with biproducts also have biproducts pointwise. Hence $\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}}$ has biproducts.

MeetSLat has all products, indeed all limits, because it is the category of algebras for a Lawvere theory. Similarly, **JoinSLat** has all coproducts, indeed all colimits, for the same reason. Note that these are very different constructions: elements of a product of meet-semilattices consist of (possibly infinite) tuples of elements, while elements of a coproduct of join-semilattices consist of *finite* formal joins of elements quotiented by the join-semilattice equations. Since **JoinSLat** has all coproducts, $\mathbf{JoinSLat}^{\mathrm{op}}$ has all products, and so $\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}}$ has all products, as required. □

Corollary 3.13. $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$ *is cartesian closed and has all coproducts.*

This corollary means that, assuming a sensible intepretation of primitive types and operations, we can use $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$ to interpret the higher-order language we describe in the next section. We still regard the category $\mathbf{Fam}(\mathbf{LatGal})$ as the reference model of approximable sets with forward and backward approximation maps; the category $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$ is a technical device to carry out the interpretation of higher-order programs. To get interpretations of first-order types and primitive operations, we can embed $\mathbf{Fam}(\mathbf{LatGal})$ into $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$:

Proposition 3.14. *The functor* $H : \mathbf{Fam}(\mathbf{LatGal}) \to \mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$ *is defined on objects as* $H(X, \partial X) = (X, \lambda x.(\partial X(x), \partial X(x)))$. *This functor is faithful and preserves coproducts and finite products.*

With this embedding functor, we will see in Lemma 4.1 that the interpretation of first-order types will be the same up to isomorphism in $\mathbf{Fam}(\mathbf{LatGal})$ and $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$, as long as we interpret the base types as objects in $\mathbf{Fam}(\mathbf{LatGal})$. At higher-order, however, the meet-semilattice and join-semilattice sides of the interpretation will diverge, and it is no longer clear that the interpretation of programs using higher-order functions internally will result in Galois connections. In §5 we will see that every program with first-order type (even if it uses higher-order functions internally) does in fact have an interpretation definable in $\mathbf{Fam}(\mathbf{LatGal})$.

3.3 Semantic Galois Slicing in Fam(LatGal)

Our thesis is that $\mathbf{Fam}(\mathbf{LatGal})$ is a suitable setting for interpreting first-order programs for Galois slicing. The above discussion has been somewhat abstract, so we now consider some examples in the category $\mathbf{Fam}(\mathbf{LatGal})$ and how they relate to Galois slicing.

Spelt out in full, $\mathbf{Fam}(\mathbf{LatGal})$ has as objects $(X, \partial X)$, all pairs of a set X and and for every $x \in X$, a bounded lattice $\partial X(x)$. Morphisms $(X, \partial X) \to (Y, \partial Y)$, are triples $(f, \partial f_f, \partial f_r)$ of functions $f : X \to Y$ and families of monotone maps $\partial f_f : \Pi_{x:X}.\partial X(x) \multimap \partial Y(f\,x)$ ("forward derivative") and $\partial f_r : \Pi_{x:X}.\partial Y(f\,x) \multimap \partial X(x)$ ("reverse derivative"), such that for all x, $\partial f_r(x) \dashv \partial f_f(x)$.

3.3.1 Unapproximated Functions. $\mathbf{LatGal}$ has a terminal (also zero) object $\mathbb{1}$, so there is a functor $\mathrm{Disc} : \mathbf{Set} \to \mathbf{Fam}(\mathbf{LatGal})$ that maps a set X to $(X, \lambda x.\mathbb{1})$ and functions f to morphisms $(f, \lambda\_.\,\mathrm{id}_{\mathbb{1}})$. This functor preserves products and coproducts. Therefore, we can take any sets and functions of interest for modelling primitive types and operations of a programming language and embed, albeit without any interesting approximation information.

3.3.2 Lifting Monad. The operation of adding a new bottom element to a bounded lattice forms part of a monad L on $\mathbf{LatGal}$. This monad extends to a (strong) monad L on $\mathbf{Fam}(\mathbf{LatGal})$ with $L(X, \partial X) = (X, L \circ \partial X)$. The monad L does not affect the points of the original object, but adds a new minimum approximation.

Let $\mathrm{Bool} = \mathrm{Disc}(\{\mathrm{tt}, \mathrm{ff}\})$ be the (unapproximated) embedding of the booleans and $\mathrm{or} : \mathrm{Bool} \times \mathrm{Bool} \to \mathrm{Bool}$ be the (unapproximated) boolean OR function. Using a Moggi-style let notation [Moggi 1991] for morphisms constructed using the Monad structure of L, we can reproduce the functions strictOr and shortCircuitOr functions from Example 2.4 (we also assume an if-then-else operation on booleans, definable from the fact that $\mathbf{Fam}(\mathbf{LatGal})$ has coproducts and Disc preserves them). Both of these expressions define morphisms $L(\mathrm{Bool}) \times L(\mathrm{Bool}) \to L(\mathrm{Bool})$ in $\mathbf{Fam}(\mathbf{LatGal})$:

$$\begin{array}{lcl} \mathrm{strictOr}(x, y) & = & \mathrm{let}\, b_1 \Leftarrow x \,\mathrm{in}\, \mathrm{let}\, b_2 \Leftarrow y \,\mathrm{in}\, \eta(\mathrm{or}(b_1, b_2)) \\ \mathrm{shortCircuitOr}(x, y) & = & \mathrm{let}\, b_1 \Leftarrow x \,\mathrm{in}\, \mathrm{if}\, b_1 \,\mathrm{then}\, \eta(\mathrm{tt}) \,\mathrm{else}\, y \end{array}$$

Examining the morphisms so defined in $\mathbf{Fam}(\mathbf{LatGal})$, we can see that, in the $\mathbf{Set}$ component, they are both exactly the normal boolean-or operation. However, they have different approximation behaviour, reflecting the different ways that they examine their inputs. Let us write $\top, \bot$ for the elements of the approximation lattice at each point of $L(\mathrm{Bool})$, then applying the reverse derivative at $(\mathrm{tt}, \mathrm{tt})$ to the tangent $\top$ reveals which of the inputs contributed to the output for each function:

$$\begin{array}{lcl} (\partial\mathrm{strictOr})_r(\mathrm{tt}, \mathrm{tt})(\top) & = & (\top, \top) \\ (\partial\mathrm{shortCircuitOr})_r(\mathrm{tt}, \mathrm{tt})(\top) & = & (\top, \bot) \end{array}$$

In comparison to the categories $\mathbf{CM}$ and $\mathbf{Stable}$ from §2, we have retained the usage information in the forward and reverse tangents, but we also accurately model totality of the functions. That is, the constantly $\bot$ function is also present in both $\mathbf{CM}$ and $\mathbf{Stable}$, but is not expressible in $\mathbf{Fam}(\mathbf{LatGal})$.

An analogue of the parallelOr function from Example 2.5 is not definable in $\mathbf{Fam}(\mathbf{LatGal})$. We would have to have $(\partial\mathrm{parallelOr})_f(\mathrm{tt}, \mathrm{tt})(\top, \bot) = (\partial\mathrm{parallelOr})_f(\mathrm{tt}, \mathrm{tt})(\bot, \top) = \top$ to reflect the desired property that either of the inputs being tt is enough to determine the output. We also must have $(\partial\mathrm{parallelOr})_f(\mathrm{tt}, \mathrm{tt})(\bot, \bot) = \bot$, to reflect the fact that we will get no information in the output if we required that neither of the inputs is examined. However, this means that $(\partial\mathrm{parallelOr})_f(\mathrm{tt}, \mathrm{tt})$ will not presere meets because $(\top, \bot) \sqcap (\bot, \top) = (\bot, \bot)$ but $\top \neq \bot$.

An analogue of the gustave function from Example 2.5 is definable in $\mathbf{Fam}(\mathbf{LatGal})$, but not using the lifting monad structure as we could for strictOr and shortCircuitOr.

Remark 6. These examples highlight a potential criticism of $\mathbf{Fam}(\mathbf{LatGal})$ as a category for modelling Galois slicing. For shortCircuitOr, we had $(\partial\mathrm{shortCircuitOr})_r(\mathrm{tt}, \mathrm{tt})(\top) = (\top, \bot)$, indicating that the second argument was not needed for computing the output. However, there is no way, in $\mathbf{Fam}(\mathbf{LatGal})$, of turning this into a rigorous statement that the $\mathbf{Set}$-component of this morphism does not actually depend on its second argument. We conjecture that this can be rectified by requiring some kind of additional structure on each object $(X, \partial X)$ consisting of a map

$\Pi_{x:X}.\partial X(x) \to \mathcal{P}(X)$, where $\mathcal{P}$ is the powerset, which identifies for each x which elements are indistinguishable from x at this level of approximation. One would also presumably have to require additional conditions for this to respect the lattice structure and be preserved by morphisms[3].

The L monad provides a controllable way of adding presence/absence approximation points to composite data, and its monad structure makes explicit in the program structure exactly how such approximations are propagated through computations. The fact that there are different choices of this kind of approximation tracking provides freedom to the language implementor to decide what information is worth tracking. The Galois slicing implementations discussed in Perera et al. [2012] and Ricciotti et al. [2017], for example, bake-in an approximation point at every composite type constructor. We will see in §4.4 that this choice can be systematised in our setting by considering a monadic CBN translation to uniformly add L approximation points to composite data types.

3.3.3 An Approximation Object and the Tagging Monad. The L monad provides a way of tagging first-order data with presence and absence information. The object $L(1)$, the lifting of the terminal object in $\mathbf{Fam}(\mathbf{LatGal})$, yields an object that consists purely of presence/absence information: $\mathbb{A} = (1, \lambda\_.\,\{\top, \bot\})$. This object is the carrier of a commutative monoid in $\mathbf{Fam}(\mathbf{LatGal})$, where the forward maps take the meet (both the inputs are required for the output to be present) and the backwards maps duplicate.

Since $\mathbb{A}$ is a monoid, we can define the writer monad $T(X) = \mathbb{A}\times X$ in $\mathbf{Fam}(\mathbf{LatGal})$ which "tags" X values with approximation information. This is similar to the L monad in that it adds approximation information to an object. On discrete objects it agrees with the lifting: $T(\mathrm{Disc}(X)) \cong L(\mathrm{Disc}(X))$. However, on composite data, the two monads give different approximation lattices. Let A and B be sets. Then $T(T(\mathrm{Disc}(A)) \times T(T(\mathrm{Disc}(B))))$ has approximation lattices at (a, b) that are always isomorphic to $\{\top, \bot\}^3$. The corresponding $L(L(\mathrm{Disc}(A)) \times L(\mathrm{Disc}(B)))$ object's approximation lattice at (a, b) is always isomorphic to $(\{\top, \bot\}^2)_\bot$. In terms of usage tracking, the object using the L monad is more appealing. The approximation lattice resulting from the use of the T monad contains apparently nonsensical elements corresponding to "using" one or other components of the product *without using the product itself.* (This arises from the fact that we can project the X out of $\mathbb{A} \times X$ without touching the $\mathbb{A}$.)

This example shows that we have to be careful about how we choose the interpretation of approximable sets in $\mathbf{Fam}(\mathbf{LatGal})$, and again highlights the point we made in Remark 6 that perhaps $\mathbf{Fam}(\mathbf{LatGal})$ does not have quite enough structure to determine "sensible" approximation information. On the other hand, the use of the T monad does have the advantage that the approximation lattices built from discrete sets, products, and coproducts, are always Boolean lattices, meaning that we can take complements of usage information. The ability to take complements of approximations has been used by Perera et al. [2022] to compute *related* outputs, as we discuss in §6.

3.3.4 Approximating Numbers by Intervals. So far, the approximation lattices we have looked at in $\mathbf{Fam}(\mathbf{LatGal})$ have only consisted of those constructed from finite products and lifting, and only track binary usage/non-usage information. Example 2.6 shows how we can go beyond this to get more "quantitative" approximation information. Let the object of reals with interval approximations in $\mathbf{Fam}(\mathbf{LatGal})$ be $\mathbb{R}_{intv} = (\mathbb{R}, \lambda x.\,\{[l, u] \mid l \leq x \leq u\} \cup \{\bot\})$ where the lattices of intervals are reverse ordered by inclusion with $\bot$ at the bottom. Then, following the examples in Example 2.6

[3] This additional structure is reminiscent of the additional structure on *directed containers* defined by Ahman et al. [2012]. They require a map $\Pi_{x:X}.\partial X(x) \to X$ picking out a specific X "jumped to" by some change $\delta x : \partial X(x)$ at x. In our proposed setup, we follow the approximation theme of Galois slicing by having a *set* of things that could be used to replace the original x. We observe that objects of $\mathbf{Fam}(\mathbf{LatGal})$ arising from $\mathbf{Stable}$ are "directed" in the Ahman et al. [2012] sense because the map can pick out the element of the original poset that was approximating x.

and Example 2.9, we can define addition, negation, and scaling by $r \geq 0$:

$$\begin{array}{lcl}
\text{add} & = & (\lambda(x_1, x_2).\, x_1 + x_2, \\
& & \ \lambda(x_1, x_2)\, ([l_1, u_1], [l_2, u_2]).\, [(l_1 + x_2) \sqcap (l_2 + x_1), (u_1 + x_2) \sqcup (u_2 + x_1)] \\
& & \ \lambda(x_1, x_2)\, [l, u].\, ([l - x_2, u - x_2], [l - x_1, u - x_1])) \\
\text{neg} & = & (\lambda x.\, -x, \lambda x\, [l, u].\, [-u, -l], \lambda x\, [l, u].\, [-u, -l]) \\
\text{scale}(r) & = & (\lambda x.\, rx, \lambda x\, [l, u].\, [rl, ru], \lambda x\, [l, u].\, \text{if } r = 0 \text{ then } \bot \text{ else } [\frac{l}{r}, \frac{u}{r}])
\end{array}$$

(we only define the forward and backward maps on intervals, their behaviour on $\bot$ is determined.) Scaling by negative numbers is also possible with swapping of bounds, as is multiplication. We will see an example of the use of these operations in §4.3.

3.4 Summary

We have seen that the category **Fam**(**LatGal**) has enough structure to express useful approximation maps at first-order and that **Fam**(**MeetSLat**×**JoinSLat**$^{\mathrm{op}}$), which is a cartesian closed category with all coproducts, is enough to interpret the total higher-order language we define in the next section with primitive types and operations defined in **Fam**(**LatGal**). However, we are not guaranteed by construction that at first-order type, the interpretations are in fact Galois connections. We will rectify this in §5 using a logical relations construction.

As we did in §2, we end the section with a conjecture relating our categories to Tangent categories.

CONJECTURE 3.15. *(1) The category* **Fam**(**MeetSLat**) *is a Tangent category, with the tangent bundle* $T(X, \partial X) = (\Sigma_{x:X}.\partial X(x), \lambda(x, \delta x).\, \downarrow(\delta x))$. *(2) The category* **Fam**(**LatGal**) *is a reverse Tangent category with the analogous definition of tangent bundle.*

Comparing this conjecture to Conjecture 2.15, we can see that the difference between **CM**, **Stable** and **Fam**(**MeetSLat**), **Fam**(**LatGal**) is that the latter have a separation of points from tangents, somewhat analogous to the situation with manifolds. If this conjecture holds, then contrary to the **Fam**(**FDVect**) representation of manifolds and differentiable maps, we do not throw away information about higher derivatives. It is retained in the order structure of the tangent fibres.

4 Higher-Order Language

To model Galois slicing semantically 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.

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 (§7). 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 $\phi \in \mathsf{Op}^{\rho}_{\rho_1,\ldots,\rho_n}$, a morphism $[\![\phi]\!]_{\mathsf{Op}} : [\![\rho_1]\!]_{\mathsf{PrimTy}} \times \ldots \times [\![\rho_n]\!]_{\mathsf{PrimTy}} \to [\![\rho]\!]_{\mathsf{PrimTy}}$.

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

Fig. 1. Syntax of types and terms

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

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

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

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

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

$$\begin{aligned} [\![\rho]\!] &= [\![\rho]\!]_{\mathsf{PrimTy}} & [\![\sigma \times \tau]\!] &= [\![\sigma]\!] \times [\![\tau]\!] \\ [\![\sigma + \tau]\!] &= [\![\sigma]\!] + [\![\tau]\!] & [\![\sigma \to \tau]\!] &= [\![\sigma]\!] \Rightarrow [\![\tau]\!] \\ [\![\mathbf{1}]\!] &= 1 & [\![\textbf{list}\ \tau]\!] &= \mathsf{List}([\![\tau]\!]) \end{aligned}$$

(a) Interpretation of Types

$$\begin{aligned} [\![\cdot]\!] &= 1 \\ [\![\Gamma, x : \tau]\!] &= [\![\Gamma]\!] \times [\![\tau]\!] \end{aligned}$$

(b) Interpretation of Contexts

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

(c) Terms as morphisms

Fig. 3. Interpretation of types, contexts and terms

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

For the first-order definability result in §5, we will need another interpretation $[\![-]\!]_{fo}$ 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 C and $\mathcal{D}$ are bicartesian and bicartesian closed categories with interpretations of the signature Σ, $F : C \longrightarrow \mathcal{D}$ is a bicartesian functor, and $F([\![\rho]\!]_{\mathsf{PrimTy}}) \cong [\![\rho]\!]_{\mathsf{PrimTy}}$ for all ρ, then for all first-order types τ, $F([\![\tau]\!]_{fo}) \cong [\![\tau]\!]$, and similar for contexts only containing first-order types.*

4.2.1 Interpretation for Higher-Order Galois slicing. 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 **Fam**(**LatGal**). Signatures are first-order, so it does not matter that **Fam**(**LatGal**) is not closed. Any such interpretation can be transported to $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$ along the functor H from Proposition 3.14 because it preserves finite products. We can then interpret the whole language in $\mathbf{Fam}(\mathbf{MeetSLat}\times\mathbf{JoinSLat}^{\mathrm{op}})$.

Interpreting a whole program $\Gamma \vdash t : \tau$ yield morphisms in $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$ which, as in §3.3, are triples $(f, \partial f_f, \partial f_r)$ of the underlying function and the forward and backward approximation maps. However, unlike in **Fam**(**LatGal**), it is not guaranteed that the forward and backward maps even operate on the same lattices, let alone form a Galois connection. Lemma 4.1 guarantees that the lattices agree, but the fact that the pair form a Galois connection is less trivial. We will prove this property in §5.

4.3 Examples

Let the signature $\Sigma_{\mathsf{num}} = (\{\mathsf{num}\}, \{\mathsf{zero} : \mathbf{1} \longrightarrow \mathsf{num}, \mathsf{add} : \mathsf{num} \times \mathsf{num} \longrightarrow \mathsf{num}\})$. This signature suffices to write the simple query function in Example 1.1, where we interpret the Label type as the sum $\mathbf{1} + \mathbf{1}$ and the labels a and b as **inl** () and **inr** (). We consider several interpretations of Σ_{num} in **Fam**(**LatGal**) and their behaviour on the selection-and-sum query from Example 1.1. First, we note the type of the reverse approximation map in this case. The type of the approximation maps depends on the input value. Our example input database was $db = [(\mathrm{a}, 0), (\mathrm{b}, 1), (\mathrm{a}, 1)]$, meaning that the type of the reverse approximation map for this database and any label l is:

$$(\partial\mathrm{query})_r(l, db) :$$
$$\quad \partial[\![\mathsf{num}]\!](\mathrm{query}\ (l, db)) \multimap \mathbb{1} \times (\mathbb{1} \times \partial[\![\mathsf{num}]\!](0)) \times (\mathbb{1} \times \partial[\![\mathsf{num}]\!](1)) \times (\mathbb{1} \times \partial[\![\mathsf{num}]\!](1)) \times \mathbb{1}$$

in the category **JoinSLat**, where $\partial[\![\mathsf{num}]\!](x)$ is the lattice of approximations of the number x determined by our interpretation. In the codomain, the first four $\mathbb{1}$s correspond to the positions of labels in the input, which we are not approximating, and the final $\mathbb{1}$ is the terminator of the list. Note how, even if the $\partial[\![\mathsf{num}]\!](x)$ does not depend on x the type of the output is still dependent on the shape of the input list: type dependency is used in a fundamental way in our interpretation.

(1) If we take $[\![\mathsf{num}]\!]_{\mathsf{PrimTy}} = \mathrm{Disc}(\mathbb{R})$, with $[\![\mathsf{zero}]\!]_{\mathsf{Op}}$ and $[\![\mathsf{add}]\!]_{\mathsf{Op}}$ the embeddings of the usual zero and addition functions via Disc, then the resulting interpretation contains no approximation information. We have $\partial[\![\mathsf{num}]\!](x) = \mathbb{1}$ so the type of $(\partial\mathrm{query})_r$ is trivial.
(2) We take $[\![\mathsf{num}]\!]_{\mathsf{PrimTy}} = L(\mathrm{Disc}(\mathbb{R}))$, using the lifting monad from §3.3.2, with $[\![\mathsf{zero}]\!]_{\mathsf{Op}}$ and $[\![\mathsf{add}]\!]_{\mathsf{Op}}$ defined from the unlifted interpretations above and the monad structure. The type

of the reverse map now becomes, where $\mathbb{2} = \{\top.\bot\}$:

$$(\partial\text{query})_r(l, db) : \mathbb{2} \multimap \mathbb{1} \times (\mathbb{1} \times \mathbb{2}) \times (\mathbb{1} \times \mathbb{2}) \times (\mathbb{1} \times \mathbb{2}) \times \mathbb{1} \tag{2}$$

where every position that corresponds to a number has been tagged with $\top$ for "present" and $\bot$ for "not present". This interpretation recovers the behaviour given in Example 1.1: running the reverse approximation map at the input "a" at approximation $\top$ reveals that only the numbers in the rows tagged with "a" in the input are used, and likewise for "b".

(3) Quantitative approximation information with non-trivial dependency can be obtained by using the interval approximation interpretation from Example 2.6 and §3.3.4. We let $[\![\mathsf{num}]\!]_{\mathsf{PrimTy}} = \mathbb{R}_{intv}$ and interpret addition using the morphism given in §3.3.4. Recall that query $(\mathrm{a}, db) = 1$, so in the reverse direction we must choose an interval containing 1 to discover the largest (i.e. least in the order) intervals that will give rise to this output as *independent* changes to the input. For example, if we pick $[\frac{9}{10}, \frac{11}{10}]$ as the interval, then:

$$(\partial\text{query})_r(\mathrm{a}, db)([\frac{9}{10}, \frac{11}{10}]) = \cdot, (\cdot, [-\frac{1}{10}, \frac{1}{10}]), (\cdot, \bot), (\cdot, [\frac{9}{10}, \frac{11}{10}]), \cdot$$

Thus, to achieve an output within $[\frac{9}{10}, \frac{11}{10}]$, either the first "a" row could be in $[-\frac{1}{10}, \frac{1}{10}]$ or the second one could be in $[\frac{9}{10}, \frac{11}{10}]$, and the number in the "b" row is not relevant.

We have tested these examples on our Agda implementation. See the file `example.agda`.

4.4 Systematic Insertion of Approximation via Moggi's CBN translation

We can now carry out the systematic insertion of approximation points that we foreshadowed in §3.3.2, using Moggi [1991, §3.1]'s monadic CBN translation. We use the T monad from §3.3.3 because it can be defined in terms of the language constructs we already have. This requires that we have a signature Σ that includes a primitive type to be interpreted as the approximation object $\mathbb{A}$ and primitive operations to be interpreted as the monoid operations on this object.

The monadic CBN translation is standard, and entirely determined by the translation on types, so we only define the type translation $\langle\!\langle - \rangle\!\rangle$ here:

$$\begin{array}{lll} \langle\!\langle \rho \rangle\!\rangle = \rho & \langle\!\langle \sigma + \tau \rangle\!\rangle = T(\langle\!\langle \sigma \rangle\!\rangle) + T(\langle\!\langle \tau \rangle\!\rangle) & \langle\!\langle \sigma \times \tau \rangle\!\rangle = T(\langle\!\langle \sigma \rangle\!\rangle) \times T(\langle\!\langle \tau \rangle\!\rangle) \\ \langle\!\langle \mathbf{1} \rangle\!\rangle = \mathbf{1} & \langle\!\langle \sigma \to \tau \rangle\!\rangle = T(\langle\!\langle \sigma \rangle\!\rangle) \to T(\langle\!\langle \tau \rangle\!\rangle) & \langle\!\langle \mathbf{list}\ \tau \rangle\!\rangle = \mathbf{list}\,(T(\langle\!\langle \tau \rangle\!\rangle)) \end{array}$$

Thus, the CBN translation on types inserts a use of the monad $T(X) = \mathbb{A} \times X$ at the point "just underneath" every type former. In this case, we are describing the type of data annotated at every level. Note that our lists here are still "strict", an alternative approach would be to consider "lazy" lists that wrap the tail of every node in a T as well.

We illustrate the effect of the CBN translation on the query example from Example 1.1. Applying $\langle\!\langle - \rangle\!\rangle$ to the type of query yields:

$$T(\text{Label}) \times T(\mathbf{list}(T(T(\text{Label}) \times T(\mathsf{num})))) \to T(\mathsf{num})$$

Thus, every substructure of the input and output has been annotated with usage information. Under interpretation in $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$, the type of the reverse approximation map as a morphism in $\mathbf{JoinSLat}$ at the input db is now (suppressing some "$\times\mathbb{1}$" for readability):

$$(\partial\text{query})_r(l, db) : \mathbb{2} \multimap \mathbb{2} \times (\mathbb{2} \times (\mathbb{2} \times \mathbb{2}) \times (\mathbb{2} \times \mathbb{2}) \times (\mathbb{2} \times \mathbb{2}))$$

Comparing to the type in (2), we now gain much more fine-grained information on which parts of the input are used. When $l = \mathrm{a}$, we have $(\partial\text{query})_r(\mathrm{a}, db)(\top) = (\top, (\top, (\top, \top), (\top, \bot), (\top, \top)))$, indicating that the execution of this query had examined everything except the number in the second entry of the database. With the previous interpretation, we did not have confirmation that the labels in each row were actually required.

5 Correctness of the Higher-Order Interpretation

As we noted at the end of §4.2.1, our interpretation of the higher-order language is in the category $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$, so it is not a priori evident that we get a Galois connection from the interpretation of a program with first-order type (that may use higher-order functions internally). 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[4]).

THEOREM 5.1 (FIORE AND SIMPSON [1999]). *Let C be an extensive bicartesian category, $\mathcal{D}$ be a bicartesian closed category, and $F : C \to \mathcal{D}$ 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 $\hat{F} : C \to \mathbf{GLR}(F)$, such that:*

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

Remark 7. 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 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 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.

THEOREM 5.2. *For all $\Gamma \vdash M : \tau$, with Γ, τ first-order, there exists $g \in \mathbf{Fam}(\mathbf{LatGal})(\llbracket \Gamma \rrbracket_{fo}, \llbracket \tau \rrbracket_{fo})$ such that $H(g) = (\cong) \circ \llbracket \Gamma \vdash M : \tau \rrbracket \circ (\cong)$, with the isomorphisms from Lemma 4.1.*

PROOF. Instantiate Theorem 5.1 with $H : \mathbf{Fam}(\mathbf{LatGal}) \to \mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$. By Proposition 3.14 we know that F preserves finite products and coproducts. The fullness of $\hat{H}$ means that for any morphism $f : \hat{H}(\llbracket \Gamma \rrbracket_{fo}) \to \hat{H}(\llbracket \tau \rrbracket_{fo})$ in $\mathbf{GLR}(H)$ there exists a $g : \llbracket \Gamma \rrbracket_{fo} \to \llbracket \tau \rrbracket_{fo}$ in $\mathbf{Fam}(\mathbf{LatGal})$ such that $H(g) = f$. Since $\mathbf{GLR}(H)$ has enough structure, we can interpret the term M in it to get a morphism $\llbracket \Gamma \vdash M : \tau \rrbracket_{\mathbf{GLR}(H)} : \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$ in $\mathbf{GLR}(H)$. Applying Lemma 4.1 and the fact that the strictness of p means that $p(\llbracket \Gamma \vdash M : \tau \rrbracket_{\mathbf{GLR}(H)}) = \llbracket \Gamma \vdash M : \tau \rrbracket_{\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})}$ yields the result. □

Remark 8. If we modified our base interpretation of semantic Galois slicing as suggested in Remark 6 to give a refined version $\mathcal{G}$ of $\mathbf{Fam}(\mathbf{LatGal})$, then if there is a finite bicartesian functor $\mathcal{G} \to \mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$, an analogous result to Theorem 5.2 still holds.

We can also use Theorem 5.1 to show that the interpretation of the language in the category $\mathbf{Set}$ agrees with the higher-order interpretation in $\mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}})$ on the underlying function at first order. 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.

[4]Our Agda development is complete except for a proof that $\mathbf{Fam}(C)$ has extensive coproducts. We plan to complete this part of the proof before any final version. Moreover, this result does not yet apply to infinitary coproducts, though we believe it is a relatively minor extension to the proof to do so.

Theorem 5.3. *For all $\Gamma \vdash M : \tau$, where Γ and τ are first-order, the underlying function in the interpretation $[\![\Gamma \vdash M : \tau]\!]_{\mathbf{Fam}(\mathbf{MeetSLat}\times\mathbf{JoinSLat}^{\mathrm{op}})}$ is equal to the interpretation $[\![\Gamma \vdash M : \tau]\!]_{\mathbf{Set}}$ in* **Set**.

Proof. Instantiate Theorem 5.1 with the functor $\langle \mathrm{Id}, \pi_1 \rangle : \mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}}) \to \mathbf{Fam}(\mathbf{MeetSLat} \times \mathbf{JoinSLat}^{\mathrm{op}}) \times \mathbf{Set}$ that is the identity in the first component and projects out the underlying function in the second. For each $\Gamma \vdash M : \tau$, we obtain a g such that $g = [\![\Gamma \vdash M : \tau]\!]_{\mathbf{Fam}(\mathbf{MeetSLat}\times\mathbf{JoinSLat}^{\mathrm{op}})}$ and $\pi_1(g) = [\![\Gamma \vdash M : \tau]\!]_{\mathbf{Set}}$. Substituting g yields the result. □

6 Related Work

Stable Domain Theory. Stable Domain Theory was originally proposed by Berry [1979] as a refinement of domain theory aimed at capturing the intensional behaviour of sequential programs, and elaborated on subsequently by Berry and Curien [1982] and Amadio and Curien [1998]. Standard domain-theoretic models interpret programs as continuous functions, preserving directed joins; Berry observed that this continuity condition alone is too permissive to model sequentiality. Stability imposes additional constraints to reflect how functions preserve bounded meets of approximants, effectively requiring that the evaluation of a function respect a specific computational order. Though stable functions do not fully characterise sequentiality, because they admit gustave-style counterexamples (Example 2.5), they remain an appropriate notion for studying the sensitivity of a program to partial data at a specific point.

Our use of Stable Domain Theory diverges from the traditional aim of modelling infinite or partial data, however. Instead, we follow a line of work that uses partiality as a qualitative notion of approximation suitable for provenance and program slicing (discussed in more detail in §6 below). Paul Taylor's characterisation of stable functions via local Galois connections on principle downsets provides the semantic underpinning for the reverse maps used in Galois slicing [Taylor 1999]. Our work builds on these ideas by interpreting Galois slicing as a form of differentiable programming, using the machinery of CHAD to present Galois slicing in a denotational style.

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 (§6 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 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. A connection to stable functions in relation to minimal slices for short-circuiting operations was alluded to in Perera [2013], but not explored. Subsequent work extended the approach to languages with assignment and exceptions [Ricciotti et al. 2017] and concurrent systems, applying Galois slicing to the π-calculus [Perera et al. 2016]. For the π-calculus the analysis shifted from functions to transition relations, considering individual transitions $P \longrightarrow Q$ between configurations P and Q as analogous to the edge between x and $f(x)$ in the graph of f, and building Galois connections between $\downarrow(P)$ and $\downarrow(Q)$. The main difference with the approach presented here is that the earlier work also computes *program slices*, using approximation lattices that represent partially erased programs; we discuss this further in §7 below.

More recent work explored Galois slicing for interactive visualisations. Perera et al. [2022] presented an approach where slicing operates over Boolean algebras rather than plain lattices. In this setting every Galois connection $f \dashv g : A \to B$ has a conjugate $g^\circ \dashv f^\circ : B \to A$, where f° denotes the De Morgan dual $\neg \circ f \circ \neg$ [Jonsson and Tarski 1951]. The provenance analysis can then be composed with its own conjugate to obtain a Galois connection which computes *related outputs* (e.g., selecting a region of a chart and observing the regions of other charts which share data dependencies). Bond et al. [2025] revisited this approach using *dynamic dependence graphs* to decouple the derivation of dependency information from the analyses that make use of it, and observing that to compute the conjugate analysis one can just use the opposite graph.

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. In Conjecture 2.15 and Conjecture 3.15, we have conjectured that the categories we have identified in this paper as models of Galois slicing are Tangent categories. This would clarify the role of higher derivatives in Galois slicing, which we conjecture are related to *program differencing*. 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].

7 Conclusion and Future Work

We have presented a semantic version of Galois slicing, inspired by connections to differentiable programming and automatic differentation, and shown that it can be used to interpret an expressive higher-order language suitable for writing simple queries and data manipulation. Our model elucidates some of the decisions implicitly taken in previous works on Galois slicing (§4.4), and reveals new applications such as approximation by intervals §3.3.4. Our categorical approach admits a modular construction of our model, and the use of general theorems, such as Fiore and Simpson's Theorem 5.1, to prove properties of the interpretation. We have focused on constructions that enable an executable implementation in Agda in this work, but have conjectured that there are connections to established notions of categorical differentiation in Conjecture 2.15 and Conjecture 3.15.

Quantitative slicing and XAI. Explainable AI (XAI) techniques like Gradient-weighted Class Activation Mapping (Grad-CAM) [Selvaraju et al. 2020] use reverse-mode AD selectively to calculate *heat maps* (or *saliency maps*) that highlight input regions contributing to a given classification or other outcome. We would like to investigate quantitative approximation structures where $\top$ represents the original input image and lower elements represent "slices" of the image where individual pixels have been ablated to some degree (partly removed or blurred). This might allow for composing some of these techniques with Galois slicing, for use in hybrid systems such as physical simulations with ML-based parameterisations.

Refinement of the model. As we discussed in Remark 6, there are possible ways that the model we have proposed here could be refined to both remove nonsensical elements of the model, and to augment the model with enough power to prove additional properties such as "functions are insenstive to unused inputs". We are also planning to explore more examples of approximation along the lines of the intervals example in §3.3.4. One route might be to follow Edalat and Hackmann [1998]'s embedding of metric spaces in Scott domains and explore whether metric spaces (which already provide a native notion of approximation) can be embedded in $\mathbf{Fam}(\mathbf{LatGal})$.

General Inductive and Coinductive Types. Lists are the only recursive data type we provided in our source language, so important future work is supporting general inductive and coinductive types. Lucatelli Nunes and Vákár [2023] support automatic differentiation for datatypes defined as the least or greatest fixed points of $\mu\nu$-polynomial functors; we could potentially adopt a similar approach. Full inductive types would allow us to embed an interpreter for a small language; combined with the CBN monadic translation described in §4.4 which uniformly inserts approximation points, we should be able to obtain the program slicing behaviour of earlier Galois slicing work "for free". Coinductive types (e.g. streams) present additional challenges, especially for defining join-preserving backward maps, but also open the door to slicing (finite prefixes of) infinite data sources, with some likely relationship to the problem of dealing with partial or non-terminating computations.

Recursion and Partiality. This work has only examined the case for total programs. Even though we took stable domain theory as our starting point, we did not make any use of directed completeness or similar properties of domains. We expect that an account of recursion in an extension of the framework discussed so far would likely involve families of bounded lattices indexed by DCPOs, where the order of the DCPO would be reflected in embedding-projection relationships between the lattices. Berry [1979]'s bidomains and Laird [2007]'s bistable biorders have separate extensional and stable orders on the same set, in a way that might be similar to Example 2.6.

Source-To-Source Translation Techniques. An interesting alternative to the denotational approach presented here, and to the trace-based approaches used in earlier Galois slicing implementations, 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 Galois slicing would allow slicing to "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.

Samson Abramsky and Achim Jung. 1995. *Domain theory.* Oxford University Press, Inc., USA, 1–168.

Danel Ahman, James Chapman, and Tarmo Uustalu. 2012. When Is a Container a Comonad?. In *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7213)*, Lars Birkedal (Ed.). Springer, 74–88. https://doi.org/10.1007/978-3-642-28729-9_5

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

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.

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.

Tom de Jong and Martín Hötzel Escardó. 2021. Domain Theory in Constructive and Predicative Univalent Foundations. In *29th EACSL Annual Conference on Computer Science Logic (CSL 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 183)*, Christel Baier and Jean Goubault-Larrecq (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:18. https://doi.org/10.4230/LIPIcs.CSL.2021.28

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

Martín H. Escardó and Cory M. Knapp. 2017. Partial Elements and Recursion via Dominances in Univalent Type Theory. In *26th EACSL Annual Conference on Computer Science Logic (CSL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 82)*, Valentin Goranko and Mads Dam (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:16. https://doi.org/10.4230/LIPIcs.CSL.2017.21

Martin Hötzel Escardó. 1996. PCF extended with real numbers. *Theoretical Computer Science* 162, 1 (1996), 79–115. https://doi.org/10.1016/0304-3975(95)00250-2

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.

Gerhard Gierz, Karl Hofmann, Klaus Keimel, Jimmie Lawson, Michael Mislove, and Dana Scott. 2003. *Continuous Lattices and Domains.* https://doi.org/10.1017/CBO9780511542725

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

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

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

James Laird. 2007. Bistable Biorders: A Sequential Domain Theory. *Logical Methods in Computer Science* Volume 3, Issue 2, Article 5 (May 2007). https://doi.org/10.2168/LMCS-3(2:5)2007

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/.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

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, Joseph Bond, Cristina David, Minh Nguyen, and Dominic Orchard. 2025. *Fluid programming language*. https://doi.org/10.5281/zenodo.14637654

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

G.D. Plotkin. 1977a. LCF Considered as a Programming Language. *Theoretical Computer Science* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

Gordon D. Plotkin. 1977b. LCF Considered as a Programming Language. *Theoretical Computer Science* 5 (1977), 223–255.

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.

Dana Scott. 1970. *Outline of a Mathematical Theory of Computation*. Technical Report PRG02. OUCL. 30 pages.

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

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

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