

# Syntax Monads for the Working Formal Metatheorist

Lawrence Dunn

University of Pennsylvania  
Philadelphia, USA

dunnla@seas.upenn.edu

Val Tannen

University of Pennsylvania  
Philadelphia, USA

val@cis.upenn.edu

Steve Zdancewic

University of Pennsylvania  
Philadelphia, USA

stevez@cis.upenn.edu

Formally verifying the properties of formal systems using a proof assistant requires justifying numerous minor lemmas about capture-avoiding substitution. Despite work on category-theoretic accounts of syntax and variable binding, *raw*, *first-order* representations of syntax, the kind considered by many practitioners and compiler frontends, have received relatively little attention. Therefore applications miss out on the benefits of category theory, such as the deeply attractive promise of reusing formalized infrastructural lemmas between implementations of different systems. Our Coq framework Tealeaves provides libraries of reusable infrastructure for first-order representations of variable binding, such as de Bruijn indices and locally nameless. In this paper we give a string-diagrammatic account of *decorated traversable monads* (DTMs), the key abstraction implemented by Tealeaves. We define DTMs as monoids of structured endofunctors before proving a representation theorem à la Kleisli.

## 1 Introduction

Machine-certified proofs of the properties of programming languages, type theories, and other formal systems are increasingly critical for establishing confidence in the design and implementation of computer systems. Much of this reasoning is overtly concerned with the manipulation of syntactical structures, especially variable-binding constructs, making the representation of these structures a key issue in formal metatheory [6]. As implementations scale in complexity to realistic formalizations of compilers [36] and programming languages [23], often with many kinds of variables, the bookkeeping required to manipulate variables correctly becomes nearly prohibitive.

Category-theoretic accounts of syntax with variable binding (e.g. [8, 17, 18, 19, 2]) offer the tantalizing benefit of formalizing tedious syntax “infrastructure” once and for all over an abstract choice of signature, instead of repeating this effort for the particular syntax of each new system. However, the kind of syntax usually considered by theorists—often intrinsically well-typed with well-scoped de Bruijn indices—is different from what many working semanticists and compilers actually implement. Consequently, the benefits of a principled categorical framework are not yet available to many applications. This work lays the foundations of a category-theoretic account of variable binding as it often looks in practice, with the aim of building certified libraries of generic syntax infrastructure that can be used (and reused) in real-world applications.

**Contributions.** This manuscript contributes two insights into the foundations of *raw*, first-order syntax.

- We introduce the strict monoidal category  $\mathbf{DecTrav}_W$  of decorated-traversable endofunctors on  $\mathbf{Set}$  for some monoid  $W$  (Definition 3.15) and define decorated-traversable monads (DTMs) as monoids in this category (Definition 3.16). Examples of decorated-traversable functors include the signature functors of languages with variable binding; the free monads they generate are DTMs. As a corollary we gain the use of monoidal string diagrams to reason about operations on syntax.

- We prove an equivalence (Theorem 4.2) between monoids in  $\mathbf{DecTrav}_W$  and a Kleisli-style presentation of DTMs (Definition 4.1) that describes a combinator for recursing on abstract syntax trees.

As with ordinary (strong) monads [27], the Kleisli presentation is of more immediate utility from a functional programming or formal metatheory perspective, in part because the definition requires checking fewer axioms. In a previous, tool-oriented paper [16] we introduced Kleisli-presented DTMs and used them to derive generic syntax infrastructure for first-order representations of variable binding in Coq. However, that paper did not explain the categorical nature of DTMs or why the abstraction should be considered “correct.” This paper justifies the axioms by proving their equivalence with a more clearly principled, string-diagrammatic set of axioms. The results in this paper have been formalized in Coq and are available in our GitHub repository.<sup>1</sup>

## 1.1 Layout

The rest is laid out as follows. Section 2 contains background on first-order representations of variable binding. We recall that raw abstract syntax is naturally associated with a (free) monad; for such monads, the Kleisli axiomatization provides a theory of naïve substitution. Kleisli arrows are not expressive enough to define other operations of interest, most notably *capture-avoiding* substitution. Section 3 introduces the endofunctor categories  $\mathbf{Dec}_W$ ,  $\mathbf{Trav}$ , and  $\mathbf{DecTrav}_W$ . Section 4 derives a Kleisli-style characterization of monoids in  $\mathbf{DecTrav}_W$  and explains why this abstraction solves the problems identified in Section 2. Section 5 contrasts our approach with related work. We conclude in Section 6. Appendices A–D elaborate on the graphical calculus and categories defined in this paper.

Unless stated otherwise, the functors in this paper have type  $\mathbf{Set} \rightarrow \mathbf{Set}$ . The reader should think of these as parameterized container types such as lists, binary trees, and abstract syntax trees. We recall that  $\mathbf{End}_{\mathbf{Set}}$  is the strict monoidal category whose objects are endofunctors on  $\mathbf{Set}$ , whose arrows are natural transformations, and whose tensor product is given by composition of functors. Finally, we recall that all endofunctors on  $\mathbf{Set}$  are canonically equipped with a tensorial strength,  $st_{A,B} : A \times FB \rightarrow F(A \times B)$ , given by pairing each element of the container with a copy of the  $A$  value.

## 2 First-order Representations of Variable Binding

The modern formal metatheorist has many options for representing and manipulating terms with variable binding in a proof assistant. A major distinction is whether to employ a *first-order* or *higher-order* approach. A higher-order representation uses functions in the metatheory to represent variable binding in the formalized language; this sidesteps thorny issues like variable capture but is fairly removed from syntax as represented in a compiler. First-order approaches represent terms as inductive datatypes. This style is appealing because it is simple, intuitive, and well supported by general-purpose proof assistants like Coq [11]. Theoretically, it provides familiar tools like initial algebras and structural recursion [12].

Within the family of first-order approaches, one can distinguish between *intrinsic* and *extrinsic* (or *raw*) encodings. An intrinsic encoding uses the metatheory’s type system to enforce static constraints on the syntax of the object theory. Intrinsically well-scoped terms, for example, are parameterized by a context  $\Gamma$  and can have as free variables at most the ones declared in  $\Gamma$ . The more traditional raw approach defines a single set of terms all at once. Properties like being well-scoped in  $\Gamma$  are then defined post-hoc as predicates on terms, rather than inherent to their type.

---

<sup>1</sup><https://github.com/dunml/tealeaves>

We are concerned with raw, first-order representations. Even here one has a choice about how to encode free and bound variables. Encoding strategies go by many names like *fully named*, *de Bruijn indices*, *de Bruijn levels*, *locally named*, *locally nameless*, and variations of these. DTMs are independent of a particular encoding and for now we shall remain agnostic about this choice. Figure 1 displays a first-order definition of the set of raw lambda terms. The only unusual part of this definition is that we parameterize the set of terms by a representation of variables  $V$  and binder annotations  $B$ . These parameters will be fixed by a variable encoding strategy in Section 2.1.

```

Inductive term (B : Set) (V : Set) : Set :=
| Var : V -> term B V
| App : term B V -> term B V -> term B V
| Lam : B -> term B V -> term B V.

```

$\text{bind } f \text{ (Var } v) = fv$   
 $\text{bind } f \text{ (App } t_1 t_2) = \text{App (bind } f t_1) (\text{bind } f t_2)$   
 $\text{bind } f \text{ (Lam } b t) = \text{Lam } b (\text{bind } f t)$

Figure 1: Syntax of the lambda calculus in Coq

Figure 2: bind instance for term

To concentrate on term as functor in  $V$ , we shall typeset  $B$  as a subscript. Associated to the lambda calculus is a signature functor

$$\Sigma_{\lambda, B} X \stackrel{\text{def}}{=} X \times X + B \times X$$

encoding the domain of the two constructors of term besides Var.  $\text{term}_B V$  is defined as the least fixpoint  $\mu X. (V + \Sigma_{\lambda, B} X)$ , i.e. as the smallest solution to the following equation:

$$\text{term}_B V \simeq V + \text{term}_B V \times \text{term}_B V + B \times \text{term}_B V.$$

By its least fixed point construction,  $\text{term}_B$  (for any  $B$ ) naturally forms a monad.

**Definition 2.1.** A monad  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  is a monoid in  $\mathbf{End}_{\mathbf{Set}}$ .

Appendix A unpacks this definition in terms of string diagrams. As a generalized monoid,  $T$  is equipped with a unit  $\text{ret}^T$  and a multiplication  $\text{join}^T$ . For term, the unit is the Var constructor, intuitively representing a coercion from variables to (atomic) terms. The multiplication flattens a  $\text{term}_B(\text{term}_B V)$  into a  $\text{term}_B V$ , witnessing the fact that term is closed under substitution of terms for variables. For our purposes, defining monads as generalized monoids like this is somewhat abstruse; the following presentation is more pragmatic.

**Definition 2.2.** A Kleisli-presented monad  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  is set-forming operation equipped with two polymorphic operations

$$\begin{aligned} \text{ret} & : \forall (A : \mathbf{Set}), A \rightarrow TA \\ \text{bind} & : \forall (A B : \mathbf{Set}), (A \rightarrow TB) \rightarrow TA \rightarrow TB \end{aligned}$$

subject to the following three laws (implicitly universally quantified over all relevant variables).

$$\text{bind ret} = \text{id}_T \quad (2.1) \quad \text{bind } g \cdot \text{bind } f = \text{bind (bind } g \cdot f) \quad (2.3)$$

$$\text{bind } f \cdot \text{ret} = f \quad (2.2)$$

**Lemma 2.3.** Definitions 2.1 and 2.2 are equivalent.

See Lemma A.4 for a string-diagrammatic derivation of the Kleisli presentation.

Figure 2 gives the bind instance for term. We note that  $\text{bind } f t$  merely applies  $f$  to each variable occurrence in  $t$ , replacing each with a subterm. We call this simple replacement operation a *naïve* substitution. (2.1) states that replacing all variables with themselves yields the original  $t$ , while (2.2) is simply the definition of bind on Var, and (2.3) governs the composition of multiple substitutions. The limitations of naïve substitution become apparent as we turn our attention to particular schemes for representing free and bound variables in lambda terms.

## 2.1 Variable Encodings

There are many schemes for representing variables and binding. We discuss two exemplary techniques.

**Fully named** A fully named approach assigns names, represented as atoms  $a \in \mathbb{A}$ , to both free and bound variables, hence  $V = \mathbb{A}$ . Variable-binding constructs are labeled with the names they introduce, so  $B = \mathbb{A}$ . The set  $\text{term}_{\mathbb{A}} \mathbb{A}$  corresponds to the following pen-and-paper syntax of lambda terms:

$$t ::= a | tt | \lambda a.t$$

Recall the main axiom of lambda calculus, the *beta-conversion* rule  $(\lambda x.t_1) t_2 =_{\beta} t_1 \{t_2/x\}$ , where  $t_1 \{t_2/x\}$  stands for the capture-avoiding substitution of  $t_2$  in place of free occurrences of  $x$  in  $t_1$ . For instance:

$$(\lambda x.xz) \{z/x\} =_{\beta} \lambda x.xz \quad (\lambda y.xz) \{z/x\} =_{\beta} \lambda y.zz \quad (\lambda z.xz) \{z/x\} =_{\beta} \lambda y.zy$$

In the first case,  $x$  occurs bound and is not replaced, while in the second and third cases it occurs free and is replaced with  $z$ . In the last case,  $z$  also happens to be the name of the distinct entity introduced by the  $\lambda$ , so a naïve substitution would incorrectly result in the term  $\lambda z.zz$ . Therefore we rename this entity, and all variables bound to it, to a non-conflicting name, say  $y$ . Renaming variables like this complicates a fully named representation, and it also complicates the theory of DTMs. Therefore this manuscript focuses on representations that do not require binder renaming, but see future work in Section 6.

**Locally nameless** The locally nameless strategy represents free variables as atoms, as before, but represents bound variables as de Bruijn indices [14], natural numbers that describe the “distance” from the occurrence to the abstraction that introduced it. For example,  $\lambda x.\lambda y.xyz$  becomes  $\lambda \lambda 10z$ . Thus  $V$  is the (tagged) union  $\mathbb{A} + \mathbb{N}$ . For clarity, we use  $\text{fvar}$  and  $\text{bvar}$  as the names of the left and right injections (respectively) into  $V$ .

Because the representation of a bound variable is canonical, we avoid the need to rename bound variables. Binding constructs need not be annotated at all, which in our framework means annotating them with type  $B = \mathbf{1} = \{\star\}$ , the singleton. The set  $\text{term}_{\mathbf{1}} (\mathbb{A} + \mathbb{N})$  corresponds to the following grammar:

$$t ::= a | n | tt | \lambda t$$

An upside of locally nameless is that substitution of free variables is particularly simple: a variable is free exactly when it is an atom, meaning it can never become bound. Consequently, capture-avoiding substitution of free variables reduces to simply the naïve substitution.<sup>2</sup> as defined in Figure 3a. This has the following type, where  $\text{subst } x \ u \ t$  replaces  $x$  in  $t$  with  $u$ .

$$\text{subst} : \mathbb{A} \rightarrow \text{term}_{\mathbf{1}} (\mathbb{A} + \mathbb{N}) \rightarrow \text{term}_{\mathbf{1}} (\mathbb{A} + \mathbb{N}) \rightarrow \text{term}_{\mathbf{1}} (\mathbb{A} + \mathbb{N})$$

A moment’s reflection shows that  $\text{subst}$  can equivalently be defined as the  $\text{bind}$  of  $\text{subst}_{\text{loc}}$ , shown in Figure 3b, where  $\text{subst}_{\text{loc}}$  defines the “local” effect of substitution on individual variable occurrences. This presentation has practical value.  $\text{subst}_{\text{loc}}$  does not depend on the particulars of  $\text{term}$ , so this definition of  $\text{subst}$  is given entirely abstractly over a monad  $T$ . Moreover, we can use the monad laws to reason about it, exemplified in the following lemma.

**Lemma 2.4.** *Let  $T$  be any monad, and let  $x$  and  $y$  ( $x \neq y$ ) be atoms. Let  $t[x \mapsto u]$  denote  $\text{subst } x \ u \ t$  as defined in Figure 3b. Substitution has the following properties:<sup>3</sup>*

$$x[x \mapsto t] = t \quad t[x \mapsto x] = t \quad t[x \mapsto u_1][y \mapsto u_2] = t[x \mapsto u_1[y \mapsto u_2]; y \mapsto u_2]$$

<sup>2</sup>More precisely, this operation cannot lead to capture assuming the new subterm satisfies a well-formedness property called local closure.

<sup>3</sup>Where  $x$  is used as a term, it is understood as the atomic term  $\text{ret}(\text{fvar } x)$ . In the third equation, the right side mentions the *parallel* substitution that simultaneously replaces all  $x$  with  $u_1[y \mapsto u_2]$  and  $y$  with  $u_2$ .

$$\begin{array}{l}
\text{subst } x \ u \ (\text{Var } v) = \begin{cases} u & \text{if } v = \text{fvar } x \\ \text{Var } v & \text{else} \end{cases} \\
\text{subst } x \ u \ (\text{App } t_1 \ t_2) = \text{App} \ (\text{subst } x \ u \ t_1) \ (\text{subst } x \ u \ t_2) \\
\text{subst } x \ u \ (\text{Lam } \star \ t) = \text{Lam } \star \ (\text{subst } x \ u \ t)
\end{array}
\qquad
\begin{array}{l}
\text{subst } x \ u \ t = \text{bind} \ (\text{subst}_{\text{loc}} \ x \ u) \ t \\
\text{subst}_{\text{loc}} \ x \ u \ v = \begin{cases} u & \text{if } v = \text{fvar } x \\ \text{ret } v & \text{else} \end{cases}
\end{array}$$

(a) Structurally recursive definition

(b) Definition abstract over a choice of monad

Figure 3: Substitution of atoms in a locally nameless representation

Lemma 2.4 is easily proven abstractly over  $T$  by appealing to equations (2.1)–(2.3). On the other hand, here is a lemma that cannot be stated and proven abstractly over  $T$ :

**Lemma 2.5** (fresh-subst). *If an atom  $x$  does not occur in  $t$ , then  $t[x \mapsto u] = t$ .*

Lemma 2.5 cannot be formulated abstractly because we lack a mechanism for defining what it means for an atom to *occur* in a term—occurrence is a predicate, and `bind` does not provide a mechanism for defining predicates. We can of course prove the lemma for `term` in particular by structural recursion, but this is no longer generic over a choice of  $T$  and cannot be shared by users formalizing a different syntax. In order to reason about syntax as a container (of occurrences of variables) like this, we define traversable monads in Section 3.2. This definition admits a generic proof of Lemma 2.5.

To implement  $\beta$ -reduction, a locally nameless formalization must also define the operation of *opening* one term by another, shown in Figure 4a. The  $\beta$ -conversion rule then takes the form  $(\lambda t)u =_{\beta} t^u$ , where  $t^u$  stands for the opening of  $t$  by  $u$ , defined by replacing all indices in  $t$  previously bound to the outermost  $\lambda$  with  $u$ . Note that the replaced variables are de Bruijn indices rather than free variables, hence opening is distinct from substitution of atoms. Unlike the case with atoms, the replaced indices do not all have the same representation: the representation of an index bound to the outer lambda depends on how many other abstractions are in scope at the occurrence—both 0 and 1 in  $\lambda(0\lambda 1)$  point to the outer  $\lambda$ , for instance. Therefore `open` is defined with an auxiliary function that maintains a count of how many binders we have gone under during recursion. In order to define operations that maintain an “accumulator” argument like this, we introduce decorated monads in Section 3.1.

As a final example, some locally nameless terms, e.g.  $\lambda(01)$ , do not correspond to ordinary lambda terms because they have indices (in this example, the 1) that do not “point” to any abstraction. Therefore one restricts attention to terms that are *locally closed*, defined in Figure 4b. Like `open`, `LC` is defined with a helper function that counts the number of binders gone under during recursion. Additionally, we note that `LC` computes a boolean value instead of a term. To define and reason about `LC`, one must integrate both concepts above to define decorated-traversable functors and DTMs.  $\Sigma_{\lambda,B}$  is an example of a decorated-traversable functor, and `termB` is a DTM. As demonstrated by our library `Tealeaves`, this abstraction suffices to prove a large suite of infrastructural lemmas about the operations above.

### 3 Decorated Traversable Functors

We introduce decorated and traversable monads separately before incorporating both to form DTMs. We concentrate on high-level intuitions; Appendices B–D elaborate on the categories and constructs involved. We present definitions type-theoretically alongside a graphical notation (color-coded for ease of reading but unambiguously labelled) defined more thoroughly in the appendices.

$$\begin{array}{l}
\text{open} : \text{term}_1(\mathbb{A} + \mathbb{N}) \rightarrow \text{term}_1(\mathbb{A} + \mathbb{N}) \rightarrow \text{term}_1(\mathbb{A} + \mathbb{N}) \quad \text{LC} : \text{term}_1(\mathbb{A} + \mathbb{N}) \rightarrow \mathbf{2} \\
\text{open } u \ t = \text{open}_0 \ u \ t \quad \text{LC } t = \text{LC}_0 \ t \\
\\
\text{open}_n \ u \ (\text{Var } v) = \begin{cases} u & \text{if } v = \text{bvar } n \\ \text{Var } v & \text{else} \end{cases} \quad \text{LC}_n(\text{Var } v) = \begin{cases} \perp & \text{if } v = \text{bvar } m \text{ and } n \leq m \\ \top & \text{else} \end{cases} \\
\text{open}_n \ u \ (\text{App } t_1 \ t_2) = \text{App} \ (\text{open}_n \ u \ t_1) \ (\text{open}_n \ u \ t_2) \quad \text{LC}_n(\text{App } t_1 \ t_2) = \text{LC}_n \ t_1 \wedge \text{LC}_n \ t_2 \\
\text{open}_n \ u \ (\text{Lam } \star \ t) = \text{Lam } \star \ (\text{open}_{n+1} \ u \ t) \quad \text{LC}_n(\text{Lam } \star \ t) = \text{LC}_{n+1} \ t
\end{array}$$

(a) Opening a lambda term by  $u$  (b) Testing for local closure

Figure 4: Operations on locally nameless terms

### 3.1 Decorations

The category  $\mathbf{Dec}_W$  of decorated functors is parameterized by some monoid  $W$ . Hereafter let  $W$  be given. In Tealeaves,  $W$  is typically the free monoid  $\text{list } B$  (i.e. lists of annotations on binders).

Fans of monoidal category theory will recall the elementary fact that all monoids in  $\mathbf{Set}$  uniquely extend to form bimonoids—a coherent combination of a monoid and comonoid. From this fact, the product-with- $W$  functor ( $W \times -$ ) is itself a bimonoid in  $\mathbf{End}_{\mathbf{Set}}$ , i.e. a *bimonad*. Decorated functors are defined as right coalgebras (or comodules) of this bimonad. By a well-known construction, (co)algebras of bimonoidal objects form a monoidal category (see Appendix B), which naturally leads to the definition of decorated monads as monoids in  $\mathbf{Dec}_W$ . Below we step through this construction more slowly.

As a first step, it is an exercise in definitions to verify that every set  $E$  is the carrier of exactly one comonoid, the duplication comonoid over  $E$ . This structure captures the essence of classical information and its fundamental operations of duplication and deletion.

**Definition 3.1.** *The duplication comonoid over  $E : \mathbf{Set}$  is given by the following operations.*

$$\begin{array}{l}
\text{del} : E \rightarrow \mathbf{1} \quad \text{del } e = \star \\
\Delta : E \rightarrow E \times E \quad \Delta e = (e, e)
\end{array}$$

The duplication comonoid induces a comonad on  $(E \times -)$  known to functional programmers as the environment or reader comonad. More precisely, the embedding

$$E \mapsto (E \times -) : \mathbf{Set} \rightarrow \mathbf{End}_{\mathbf{Set}}$$

is strong monoidal,<sup>4</sup> hence it preserves monoids and comonoids.

**Definition 3.2.** *The environment comonad over  $E : \mathbf{Set}$  is given by the product functor  $(E \times -)$  paired with the following operations of extraction and duplication.*

$$\begin{array}{l}
\text{extr}^{\text{E}\times} : \forall (A : \mathbf{Set}), E \times A \rightarrow A \quad \text{extr}_A(e, a) = a \\
\text{dup}^{\text{E}\times} : \forall (A : \mathbf{Set}), E \times A \rightarrow E \times (E \times A) \quad \text{dup}_A(e, a) = (e, (e, a))
\end{array}$$

The co-Kleisli arrows of the environment comonad have the form  $E \times A \rightarrow B$ . In functional programming, this comonad captures computations  $A \rightarrow B$  that additionally can read, but not modify, an

<sup>4</sup>As opposed to merely lax or oplax monoidal, not to be confused with tensorial strength.

environment of type  $E$ , such as a user-supplied configuration file. This is a classic example of the general intuition that while monads can be used to structure computations with “effects”, comonads represent notions of computation that depend on a “context.” [34].

Let  $W = \langle W, \cdot, 1_W \rangle$  be a monoid. Mirroring the duplication comonoid, the monoid on  $W$  endows  $(W \times -)$  with a monad structure known variously as the writer or logger monad.


**Definition 3.3.** *The writer monad over  $W : \mathbf{Set}$  is given by the product functor  $(W \times -)$  paired with the following operations.*

$$\begin{aligned} \text{ret}^{W \times} &: \forall (A : \mathbf{Set}), A \rightarrow W \times A & \text{ret}_A^{W \times} a &= (1_W, a) \\ \text{join}^{W \times} &: \forall (A : \mathbf{Set}), W \times (W \times A) \rightarrow W \times A & \text{join}_A^{W \times} (w_1, (w_2, a)) &= (w_1 \cdot w_2, a) \end{aligned}$$

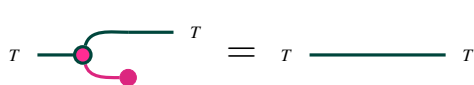
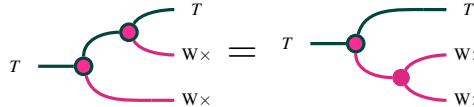
Note that the carrier set  $W$  of the monoid is also equipped with its unique comonoid structure. Crucially, we can say more: the monoid and comonoid operations are all homomorphisms of the other structure (in a precise sense), such that  $W$  is a *bimonoid*. Lifting this up to  $\mathbf{End}_{\mathbf{Set}}$ , any instance of the writer monad  $(W \times -)$  is also an instance of the environment comonad, and in fact  $(W \times -)$  is a bimonad. Appendix B elaborates on this point.

If one thinks about functors as functional data structures, then a decorated functor is one whose elements each occur in a context of type  $W$ . To our knowledge definition is novel.

**Definition 3.4.** *A decorated functor  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  is a right coalgebra of the writer bimonad  $(W \times -)$ . Explicitly, it is a functor equipped with a natural transformation*

$$\text{dec} : \forall (A : \mathbf{Set}), TA \rightarrow T(W \times A)$$


subject to the following equations:

$$\text{map}^T \text{extr}_A^{W \times} \cdot \text{dec}_A^T = \text{id}_{TA} \quad (3.1) \quad \text{dec}_{W \times A} \cdot \text{dec}_A = \text{map}^T \text{dup}_A^{W \times} \cdot \text{dec}_A \quad (3.2)$$

In this paper, the writer bimonad is drawn with a red wire whereas plain functors are shown in black. Intuitively, (3.1) states that computing the context of every element and immediately deleting it is the same as doing nothing. (3.2) states that computing each context once and making a copy of it is the same as computing the context twice.

**Example 3.5.** *The functor  $\Sigma_{\lambda, B}$  is decorated by  $\text{list } B$  (the free monoid over  $B$ ). The operation is defined as follows (where by abuse of notation we give constructors of  $\Sigma_{\lambda}$  the same name as corresponding constructors of  $\text{term}$ ):*

$$\begin{aligned} \text{dec}_X &: \Sigma_{\lambda, B} X \rightarrow \Sigma_{\lambda, B} (\text{list } B \times X) \\ \text{dec}(\text{App } x_1 \ x_2) &= \text{App}([\ ], x_1) ([\ ], x_2) \\ \text{dec}(\text{Lam } b \ x) &= \text{Lam } b ([b], x) \end{aligned}$$

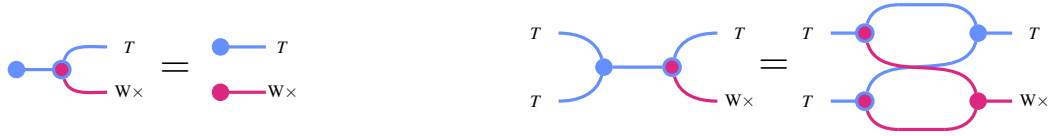
Notation:  $[\ ]$  is the empty list, while  $[b]$  is a singleton.

Intuitively, the decoration in Example 3.5 encodes the policy determining which constructors act as binders in which arguments. The policy states that an abstraction  $\lambda b.x$  adds  $b$  to the binding context of all occurrences in its body, but applications contribute nothing to the binding context of variables.

**Lemma 3.6.** *Decorated functors and decoration-preserving natural transformations form a monoidal category,  $\mathbf{Dec}_W$ . c.f. Appendix, Lemma B.6.*

In this paper, functors equipped with monadic structure are depicted with blue wires. Since  $\mathbf{Dec}_W$  is a monoidal category, it makes sense to generalize monads to decorated monads, i.e. monoidal objects in  $\mathbf{Dec}_W$ . Besides the monad laws and (3.1)–(3.2), decorated monads satisfy two more equations.

**Definition 3.7.** *A decorated monad is a monoid in  $\mathbf{Dec}_W$ . Explicitly, it is equipped with the structures of both a decorated functor and a monad such that the following equations are also satisfied.*



$$\text{dec}_A \cdot \text{ret}_A^T = \text{ret}_{W \times A}^T \cdot \text{ret}_A^{W \times} \quad (3.3) \quad \text{dec}_A \cdot \text{join}_A^T = \text{join}^{T \cdot W \times} \cdot \text{dec}_{T(W \times A)} \cdot \text{map}^T(\text{dec}_A) \quad (3.4)$$

In the context of syntax metatheory, (3.3) states that an atomic term (some  $\text{Var } x$ ) has no binders—the context of  $x$  is the monoid unit, typically the empty list or the natural number 0. (3.4) governs how decoration behaves when we compose constructors to form complex syntax trees. It states that the context of each variable instance is the concatenation of the context contributed by each constructor. That is, binders accumulate as we recurse down a syntax tree.

**Example 3.8.** *The monad  $\text{term}_B$  is decorated by  $\text{list } B$ . The operation annotates each variable with the list of  $B$  values encountered on the unique path from root of the syntax tree to the variable occurrence. We show examples using fully named and locally nameless variables:*

$$\begin{aligned} \text{dec} : \text{term}_\mathbb{A} \mathbb{A} &\rightarrow \text{term}_\mathbb{A} (\text{list } \mathbb{A} \times \mathbb{A}) & \text{dec} : \text{term}_\mathbb{1} (\mathbb{A} + \mathbb{N}) &\rightarrow \text{term}_\mathbb{1} (\mathbb{N} \times (\mathbb{A} + \mathbb{N})) \\ \lambda x. \lambda y. yx &\mapsto \lambda x. \lambda y. ([x, y], y)([x, y], x) & \lambda \lambda 0 1 &\mapsto \lambda \lambda (2, 0)(2, 1) \\ (\lambda x. z) (\lambda x. y \lambda y. z) &\mapsto (\lambda x. ([x], z)) (\lambda x. ([x], y) \lambda y. ([x, y], z)) & (\lambda 0) (\lambda \lambda 1) &\mapsto (\lambda (1, 0)) (\lambda \lambda (2, 1)) \end{aligned}$$

*Note that in the locally nameless example we make the implicit identification  $\text{list } \mathbf{1} \simeq \mathbb{N}$ .*

## 3.2 Traversals

Intuitively, a traversable data structure is a finitary container we can “iterate” [21] over, such as a `list` or `tree` type. McBride and Paterson [26] defined traversable functors as those equipped with a distributive law over applicative functors (i.e. lax monoidal endofunctors on  $\mathbf{Set}$ ). Lacking a suitable set of axioms, the notion was found to be too coarse, and subsequent work [21, 22] refined the notion. Waern [35] defined the category of traversable functors. The notion of a traversable monad appears to be novel.

**Definition 3.9.** *An applicative functor is a lax monoidal functor on  $\mathbf{Set}$ . See Definition C.1.*

In brief, an applicative functor  $F$  is one that “preserves” Cartesian product in the sense that there is a natural transformation  $FA \times FB \rightarrow F(A \times B)$ . This class includes the identity functor  $\mathbb{1}$  and is closed under composition. An important special case is the constant functor mapping all sets to some monoid  $M$  and all functions to  $\text{id}_M$ , in which case the previous operation can be identified with multiplication in  $M$ .

**Definition 3.10.** *An applicative morphism  $\phi : F \rightarrow G$  between applicative functors is one that respects their applicative structure in an obvious way. See Definition C.2.*



**Definition 3.11.** A traversable functor  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  is a functor paired with a family of distributions

$$\begin{array}{c} F \\ \text{---} \\ T \\ \text{---} \\ F \end{array} \quad \text{dist} : \forall (A : \mathbf{Set}), T(FA) \rightarrow F(TA)$$

where  $F$  ranges over applicative functors. This operation is subject to the following laws (with  $\phi$  ranging over applicative morphisms).

$$\text{dist}_{\mathbb{1},A} = \text{id}_{TA} \quad (3.5) \quad \text{dist}_{G,A} \cdot \text{map}^T(\phi_A) = \phi_{TA} \cdot \text{dist}_{F,A} \quad (3.7)$$

$$\text{map}^F(\text{dist}_{G,A}) \cdot \text{dist}_{F,GA} = \text{dist}_{F \cdot G,A} \quad (3.6)$$

The connection between traversability and container-like properties is best exemplified by choosing  $F$  to be a constant functor over a monoid  $M$ . Then, the type of  $\text{dist}$  reduces to  $TM \rightarrow M$ . Intuitively,  $T$  contains a finite number of elements, so that when all elements have type  $M$ , we can combine them together using multiplication in  $M$ . Gibbons and Oliveira pointed out that (3.5) forbids this operation from “skipping” any elements in  $T$ , while Jaskelioff and Rypacek pointed out that (3.6) forbids this operation from “double counting” any elements.

**Lemma 3.12.** Traversable functors and distribution-respecting natural transformations form a monoidal category,  $\mathbf{Trav}$ . c.f. Lemma C.7.

**Definition 3.13.** A traversable monad  $T$  is a monoid in  $\mathbf{Trav}$ . Explicitly,  $T$  has the structures of both a traversable functor and a monad and satisfies the following equations:

$$\begin{array}{c} F \\ \text{---} \\ T \\ \text{---} \\ F \end{array} \quad \text{---} \quad \begin{array}{c} F \\ \text{---} \\ F \\ \text{---} \\ T \end{array} \quad \text{---} \quad \begin{array}{c} F \\ \text{---} \\ F \\ \text{---} \\ T \end{array}$$

$$\text{dist}_{F,A} \cdot \text{ret}_{FA}^T = \text{map}_A^F(\text{ret}_A^T) \quad (3.8)$$

$$\begin{array}{c} F \\ \text{---} \\ T \\ \text{---} \\ F \end{array} \quad \text{---} \quad \begin{array}{c} F \\ \text{---} \\ T \\ \text{---} \\ F \end{array}$$

$$\text{dist}_{F,A}^T \cdot \text{join}_{FA} = \text{map}^F(\text{join}_A) \cdot \text{dist}_{F,A}^{T,T} \quad (3.9)$$

Though the laws appear opaque, for syntax metatheory, (3.8) states that a term formed from  $\text{ret}/\text{Var}$  contains only a single variable. (3.9) implies that substituting a subterm  $u$  for  $x$  in  $t$  adds the occurrences in  $u$  to the set of occurrences of  $t$ . This concept is more thoroughly examined in [16].

### 3.3 Decorated Traversable Functors

For functors that are both traversable and decorated, it is necessary to impose one more condition. For the following definition, we note that  $(W \times -)$  is uniquely traversable.

**Definition 3.14.** A decorated-traversable functor is equipped with the structures of both a decorated and traversable functor, subject to the following condition:

$$\begin{array}{c} F \\ \text{---} \\ T \\ \text{---} \\ F \end{array} \quad \text{---} \quad \begin{array}{c} F \\ \text{---} \\ T \\ \text{---} \\ F \end{array}$$

$$\text{map}^F \text{dec}_A \cdot \text{dist}_{F,A}^T = \text{dist}_{F,W \times A}^T \cdot \text{map}^T(\text{dist}_{F,A}^{W \times}) \cdot \text{dec}_{FA} \quad (3.10)$$

**Definition 3.15.** *Decorated traversable functors and their structure-preserving natural transformations form a strict monoidal category,  $\mathbf{DecTrav}_W$ . c.f. Lemma D.4.*

**Definition 3.16.** *A decorated traversable monad (DTM) is a monoid in  $\mathbf{DecTrav}_W$ .*

The force of Definition 3.16 is as follows. First, as an object in  $\mathbf{DecTrav}_W$ ,  $T$  is equipped with a decoration and traversal satisfying (3.10). Furthermore, it is a monad such that both of the monad operations commute with decorations and traversals. Self-contained equational and string-diagrammatic presentations of this definition are given in the Appendix, Definition D.5.

## 4 Kleisli Representation for DTMs

Definition 3.16 is phrased in terms of principled categorical abstractions and offers the opportunity for string-diagrammatic reasoning. In practice, particularly in a theorem prover, it is tedious to prove a given syntax forms a DTM, as this requires defining five operations subject to a total of 19 equations. The following Kleisli-style definition, mirroring Definition 2.2, is more economical and more useful to program with. First, we define an auxiliary helper function to be used in (4.3).

**Definition 4.1** (DTMs, Kleisli-style). *A Kleisli-presented DTM is a set-forming operation  $T$  equipped with two operations of the following types*

$$\begin{aligned} \text{ret} &: \forall (A : \mathbf{Set}), A \rightarrow TA \\ \text{binddt} &: \forall (F : \mathbf{Applicative}) (A : \mathbf{Set}), (W \times A \rightarrow F(TB)) \rightarrow TA \rightarrow F(TB) \end{aligned}$$

subject to the following laws (where  $\phi$  is quantified over applicative morphisms  $\phi : F \Rightarrow G$ )

$$\text{binddt}_{\perp} (\text{ret} \cdot \text{extr}) = \text{id}_{TA} \quad (4.1)$$

$$\text{binddt}_F f \cdot \text{ret} = f \cdot \text{ret}^{W \times} \quad (4.2)$$

$$\text{map}^F (\text{binddt}_G g) \cdot (\text{binddt}_F f) = \text{binddt}_{F \cdot G} (\lambda (w, a). \text{map}^F (\text{binddt}_G (g \odot w)) f(w, a)) \quad (4.3)$$

$$\phi \cdot \text{binddt}_F f = \text{binddt}_G (\phi \cdot f) \quad (4.4)$$

In (4.3),  $(\odot)$  is defined  $(f \odot w_1)(w_2, a) \stackrel{\text{def}}{=} f(w_1 \cdot w_2, a)$ .

The following theorem speaks to the robustness of Definition 4.1.

**Theorem 4.2.** *Definitions 3.16 and 4.1 are equivalent.*

*Proof.* The  $\text{ret}$  operation is the same for both presentations. Given  $\text{map}$ ,  $\text{join}$ ,  $\text{dec}$ , and  $\text{dist}$ , we define  $\text{binddt}$  as follows:

$$\text{binddt}_F f \stackrel{\text{def}}{=} \text{map}^F (\text{join}_{\mathbb{B}}^T) \cdot \text{dist}_F^T \cdot \text{map}^T f \cdot \text{dec}. \quad (4.5)$$

Given  $\text{ret}$  and  $\text{binddt}$ , we define the operations of DTMs thus:

$$\begin{aligned} \text{map} f &\stackrel{\text{def}}{=} \text{binddt}_{\perp} (\text{ret}^T \cdot f \cdot \text{extr}^{W \times}) & \text{dec} &\stackrel{\text{def}}{=} \text{binddt}_{\perp} (\text{ret}^T) \\ \text{join} &\stackrel{\text{def}}{=} \text{binddt}_{\perp} (\text{extr}^{W \times}) & \text{dist} &\stackrel{\text{def}}{=} \text{binddt}_F (\text{ret} \cdot \text{extr}^{W \times}) \end{aligned}$$

Appendix D.6 contains a string-diagrammatic derivation of the Kleisli presentation. To be an equivalence, we must also verify the other direction and check that starting with either representation and completing a roundtrip returns the original set of operations. A full proof of this fact can be found in our GitHub repository.  $\square$

**Example 4.3.** The  $\text{binddt}$  operation for  $\text{term}_B$  is defined as follows (for any  $f : W \times A \rightarrow F(TB)$ ):

$$\begin{aligned} \text{binddt}_F f (\text{Var } v) &= f([], v) \\ \text{binddt}_F f (\text{App } t_1 t_2) &= \text{pure}^F \text{App} \otimes (\text{binddt}_F f t_1) \otimes (\text{binddt}_F f t_2) \\ \text{binddt}_F f (\text{Lam } b t) &= \text{pure}^F (\text{Lam } b) \otimes (\text{binddt}_F (f \odot [b]) t) \end{aligned}$$

*N.B.*  $\otimes$  is the idiomatic application of applicative functor  $F$ . See Appendix C.

Like  $\text{bind}$ ,  $\text{binddt}$  can be seen as a template for defining structurally recursive operations on abstract syntax trees. However, it is appreciably more expressive, introducing two new features. First, the first argument of  $f$  is now a list of binders in scope at each variable. Second, the output of  $f$  is wrapped in an applicative functor, and all function application is replaced with idiomatic application. Incorporating these aspects greatly expands the range of operations we can define generically.

#### 4.1 Substitution Metatheory

Figure 5 contains generic versions of the opening operation and local closure. These operations are defined for any DTM  $T$ . As instances of  $\text{binddt}$ , we can reason about these operations axiomatically. Note that in the definition of  $\text{LC}$ , the boolean type  $\mathbf{2}$  stands for the constant applicative functor over the monoid  $\langle \mathbf{2}, \wedge, \top \rangle$ , which provides a form of universal quantification over variables.

$$\begin{array}{ll} \text{open}_{\text{loc}} : T(\mathbb{A} + \mathbb{N}) \rightarrow \mathbb{N} \times (\mathbb{A} + \mathbb{N}) \rightarrow T(\mathbb{A} + \mathbb{N}) & \text{LC}_{\text{loc}} : \mathbb{N} \times (\mathbb{A} + \mathbb{N}) \rightarrow \mathbf{2} \\ \text{open}_{\text{loc}} u (n, \text{fvar } a) = \text{ret}(\text{fvar } a) & \text{LC}_{\text{loc}}(n, \text{fvar } a) = \top \\ \text{open}_{\text{loc}} u (n, \text{bvar } m) = \begin{cases} u & \text{if } n = m \\ \text{ret}^T(\text{bvar } m) & \text{else} \end{cases} & \text{LC}_{\text{loc}}(n, \text{bvar } m) = \begin{cases} \perp & \text{if } n \leq m \\ \top & \text{else} \end{cases} \\ \text{open } u = \text{binddt}_{\mathbb{1}}^T(\text{open}_{\text{loc}} u) & \text{LC} = \text{binddt}_{\mathbf{2}}^T \text{LC}_{\text{loc}} \end{array}$$

Figure 5: Generic locally nameless operations for a DTM  $T$

The adequacy of Definition 4.1 for the needs of working metatheorists is an empirical one demonstrated by formalizing generic syntax metatheory with it. For comparison, Weirich and Aydemir previously introduced LNgen [5], a code generator that accepts a grammar and synthesizes files containing locally nameless infrastructure for it in Coq. Using Tealeaves, we were able to formalize all of the infrastructure lemmas defined in [5], as well as others, statically and generically over a choice of arbitrary DTM. We have not found any lemmas of the locally nameless representation that we cannot prove in this fashion. The advantage of Tealeaves over LNgen is that our lemmas are proved once and for all, while LNgen generates proofs specific to a given signature. Because it relies on heuristics and Ltac [15] (Coq’s incompletely specified proof automation language), the authors have reported in private correspondence that LNgen can fail to prove some lemmas. Additionally they have reported long compile times which must be re-endured after any changes to the user’s syntax. These downsides do not apply to Tealeaves because it is a static Coq library rather than a program. The cost of entry is to furnish a proof of (4.1)–(4.4), which is straightforward to automate.

We have also developed a generalization of DTMs for languages with multiple sorts of variables, and re-derived the same locally nameless infrastructure, now extended to reason about operations affecting different sorts of variables. Tealeaves can also be used to formalize de Bruijn indices and levels [14].

## 5 Related Work

Bellegarde and Hook [8] first considered term monads in the context of formal metatheory. They defined substitution for a de Bruijn encoding in terms of a combinator  $E_{wp}$  (“extend with policy”) which is similar in spirit to, but strictly less expressive than,  $bind_{dt}$ . Lacking axioms comparable to (4.1)–(4.4), they were unable to reason about substitution generically.

Subsequent work has generally considered intrinsically well-scoped [4] and well-typed [10, 25, 3] representations using heterogeneous datatypes [9]. Leveraging the metatheory’s type system to constrain object terms will tend to lead to a more dependently-typed style of programming where operations and their correctness properties are woven together. Building on this line of work, Ahrens et al. [2] have recently proposed an intrinsically typed language formalization framework in Coq. The goal of Tealeaves is to support raw syntax, which involves defining operations first and reasoning about them post factum.

Fiore and collaborators [17, 18] have developed a presheaf-theoretic account of syntax. Subsequent work by Power and Tanaka axiomatized and expanded the presheaf-theoretic approach [29, 30]. The basic idea is that intrinsically scoped terms are stratified by a context—the set of all contexts is then used as the indexing category for the presheaves. In our development, syntax is parameterized by types  $V$  and  $B$  for representations of variables and binder annotations. These are fixed by a particular representation strategy (e.g. locally nameless) and one is left with a single set of terms rather than a presheaf. Fiore and Szamozvancev have proposed a intrinsically well-scoped, well-typed, syntax formalization framework in Agda [19] which takes inspiration from the presheaf approach.

Approaches that differ more dramatically from ours include strategies based on nominal sets [20] and variations of higher-order abstract syntax [28, 13].

Besides LNgen, utilities similar in spirit to Tealeaves include GMeta [24] and Autosubst [32, 33]. GMeta is a Coq framework for generic raw, first-order syntax. Like Tealeaves, it is parameterized by a variable encoding strategy. GMeta resorts to proofs by induction on a universe of representable types, while Tealeaves is based on a principled equational theory. Autosubst is an equational framework for reasoning about de Bruijn indices in Coq based on explicit substitution calculi [1, 31]. Our  $bind_{dt}$  can express de Bruijn substitution; it may be enlightening to consider DTMs vis-à-vis these calculi.

## 6 Conclusion and Future Work

We have presented decorated traversable monads, an enrichment of monads on the category of sets that can be used to reason equationally about raw, first-order representations of variable binding.

As presented, DTMs are not equipped with a binder-renaming operation necessary to implement a fully named binding strategy. A first step in this direction is to recognize that  $term$  is also a functor in  $B$  besides  $V$ , yielding an operation

$$bmap : \forall (V B_1 B_2 : \mathbf{Set}), (B_1 \rightarrow B_2) \rightarrow term_{B_1} V \rightarrow term_{B_2} V$$

We are investigating an extension of DTMs that incorporates the functor instance in  $B$ . One intended application is to provide a certified generic translation between a named and locally nameless representation, which could be used as part of a certified compiler, for example.

Imposing a distributive law over all applicative functors imposes an order on variable occurrences, which may be unnecessarily strong. Some process calculi, for example, consider a notion of parallel composition  $|$  such that formulas  $p_1|p_2$  and  $p_2|p_1$  should be taken as syntactically identical. To support quotiented syntax, one could require a distributive law only over some applicative functors, like commutative ones.

## References

- [1] M. Abadi, L. Cardelli, P.-L. Curien & J.-J. Lévy (1991): *Explicit substitutions*. *Journal of Functional Programming* 1(4), p. 375–416, doi:10.1017/S095679680000186.
- [2] Benedikt Ahrens, Ralph Matthes & Anders Mörtberg (2022): *Implementing a Category-Theoretic Framework for Typed Abstract Syntax*. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2022*, Association for Computing Machinery, New York, NY, USA, p. 307–323, doi:10.1145/3497775.3503678. Available at <https://doi.org/10.1145/3497775.3503678>.
- [3] Guillaume Allais, James Chapman, Conor McBride & James McKinna (2017): *Type-and-Scope Safe Programs and Their Proofs*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017*, Association for Computing Machinery, New York, NY, USA, p. 195–207, doi:10.1145/3018610.3018613. Available at <https://doi.org/10.1145/3018610.3018613>.
- [4] Thorsten Altenkirch & Bernhard Reus (1999): *Monadic Presentations of Lambda Terms Using Generalized Inductive Types*. In: *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic, CSL '99*, Springer-Verlag, Berlin, Heidelberg, p. 453–468.
- [5] Brian Aydemir & Stephanie Weirich (2010): *LNgen: Tool Support for Locally Nameless Representations*. Technical Report, University of Pennsylvania, Department of Computer and Information Science.
- [6] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The PoplMark Challenge*. In: *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'05*, Springer-Verlag, Berlin, Heidelberg, p. 50–65, doi:10.1007/11541868\_4. Available at [https://doi.org/10.1007/11541868\\_4](https://doi.org/10.1007/11541868_4).
- [7] Tørris Koløen Bakke (2007): *Hopf Algebras and Monoidal Categories*. Master's thesis, University of Tromsø.
- [8] Françoise Bellegarde & James Hook (1994): *Substitution: A Formal Methods Case Study Using Monads and Transformations*. *Sci. Comput. Program.* 23(2–3), p. 287–311, doi:10.1016/0167-6423(94)00022-0. Available at [https://doi.org/10.1016/0167-6423\(94\)00022-0](https://doi.org/10.1016/0167-6423(94)00022-0).
- [9] Richard Bird & Lambert Meertens (1998): *Nested datatypes*. In: *In MPC'98, volume 1422 of LNCS*, Springer-Verlag, pp. 52–67.
- [10] Richard Bird & Ross Paterson (1999): *de Bruijn notation as a nested datatype*. *Journal of Functional Programming* 9, pp. 77 – 91, doi:10.1017/S0956796899003366.
- [11] Pierre Boutillier, Stephane Glondou, Benjamin Grégoire, Hugo Herbelin, Pierre Letouzey, Pierre-Marie Pédro, Yann Régis-Gianas, Matthieu Sozeau, Arnaud Spiwack & Enrico Tassi (2014): *Coq 8.4 Reference Manual*. Research Report, Inria. Available at <https://hal.inria.fr/hal-01114602>. The Coq Development Team.
- [12] Rod M. Burstall (1969): *Proving Properties of Programs by Structural Induction*. *Comput. J.* 12, pp. 41–48.
- [13] Adam Chlipala (2008): *Parametric Higher-Order Abstract Syntax for Mechanized Semantics*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP '08*, Association for Computing Machinery, New York, NY, USA, p. 143–156, doi:10.1145/1411204.1411226. Available at <https://doi.org/10.1145/1411204.1411226>.
- [14] N.G de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. *Indagationes Mathematicae (Proceedings)* 75(5), pp. 381–392, doi:https://doi.org/10.1016/1385-7258(72)90034-0. Available at <https://www.sciencedirect.com/science/article/pii/1385725872900340>.
- [15] David Delahaye (2000): *A Tactic Language for the System Coq*. In Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 85–95.
- [16] Lawrence Dunn, Val Tannen & Steve Zdancewic (2023): *Tealeaves: Structured monads for generic first-order abstract syntax infrastructure*. To appear.

- [17] M. Fiore, G. Plotkin & D. Turi (1999): *Abstract syntax and variable binding*. In: *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pp. 193–202, doi:10.1109/LICS.1999.782615.
- [18] Marcelo Fiore (2008): *Second-Order and Dependently-Sorted Abstract Syntax*. In: *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science, LICS '08*, IEEE Computer Society, USA, p. 57–68, doi:10.1109/LICS.2008.38. Available at <https://doi.org/10.1109/LICS.2008.38>.
- [19] Marcelo Fiore & Dmitrij Szamozvancev (2022): *Formal Metatheory of Second-Order Abstract Syntax*. *Proc. ACM Program. Lang.* 6(POPL), doi:10.1145/3498715. Available at <https://doi.org/10.1145/3498715>.
- [20] Murdoch J. Gabbay & Andrew M. Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Form. Asp. Comput.* 13(3–5), p. 341–363, doi:10.1007/s001650200016. Available at <https://doi.org/10.1007/s001650200016>.
- [21] Jeremy Gibbons & Bruno Oliveira (2009): *The essence of the Iterator pattern*. *J. Funct. Program.* 19, pp. 377–402, doi:10.1017/S0956796809007291.
- [22] Mauro Jaskielioff & Ondrej Rypacek (2012): *An Investigation of the Laws of Traversals*. *Electronic Proceedings in Theoretical Computer Science* 76, doi:10.4204/EPTCS.76.5.
- [23] Ramana Kumar, Magnus O. Myreen, Michael Norrish & Scott Owens (2014): *CakeML: a verified implementation of ML*. In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pp. 179–192, doi:10.1145/2535838.2535841. Available at <http://doi.acm.org/10.1145/2535838.2535841>.
- [24] Gyesik Lee, Bruno C. D. S. Oliveira, Sungkeun Cho & Kwangkeun Yi (2012): *GMeta: A Generic Formal Metatheory Framework for First-Order Representations*. In Helmut Seidl, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 436–455.
- [25] Conor McBride (2005): *Type-Preserving Renaming and Substitution*. Unpublished note.
- [26] Conor McBride & Ross Paterson (2008): *Applicative Programming with Effects*. *J. Funct. Program.* 18(1), p. 1–13, doi:10.1017/S0956796807006326. Available at <https://doi.org/10.1017/S0956796807006326>.
- [27] Eugenio Moggi (1988): *Computational Lambda-Calculus and Monads*. IEEE Computer Society Press, pp. 14–23.
- [28] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. 23, pp. 199–208, doi:10.1145/960116.54010.
- [29] John Power (2003): *A Unified Category Theoretic Approach to Variable Binding*. In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding, MERLIN '03*, Association for Computing Machinery, New York, NY, USA, p. 1–9, doi:10.1145/976571.976578. Available at <https://doi.org/10.1145/976571.976578>.
- [30] John Power & Miki Tanaka (2008): *Category Theoretic Semantics for Typed Binding Signatures with Recursion*. *Fundam. Informaticae* 84, pp. 221–240.
- [31] Steven Schäfer, Gert Smolka & Tobias Tebbi (2015): *Completeness and Decidability of de Bruijn Substitution Algebra in Coq*. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP '15*, Association for Computing Machinery, New York, NY, USA, p. 67–73, doi:10.1145/2676724.2693163. Available at <https://doi.org/10.1145/2676724.2693163>.
- [32] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions*. In Xingyuan Zhang & Christian Urban, editors: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI, Springer-Verlag.
- [33] Kathrin Stark, Steven Schäfer & Jonas Kaiser (2019): *Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions*. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, Association for Computing Machinery, New York, NY, USA, p. 166–180, doi:10.1145/3293880.3294101. Available at <https://doi.org/10.1145/3293880.3294101>.

- [34] Tarmo Uustalu & Varmo Vene (2008): *Comonadic Notions of Computation*. *Electronic Notes in Theoretical Computer Science* 203(5), pp. 263–284, doi:<https://doi.org/10.1016/j.entcs.2008.05.029>. Available at <https://www.sciencedirect.com/science/article/pii/S1571066108003435>. Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).
- [35] Love Waern (2019): *Cofree Traversable Functors*. Bachelor’s thesis.
- [36] Jianzhou Zhou, Santosh Nagarakatte, Milo Martin & Steve Zdancewic (2012): *Formalizing the LLVM Intermediate Representation for Verified Program Transformations*. 47, pp. 427–440, doi:10.1145/2103621.2103709.

In the Appendices we present a string-diagrammatic account of **DecTrav<sub>W</sub>** and DTMs. Appendix A is a brief and informal introduction to string diagrams for strict monoidal categories (specifically **End<sub>Set</sub>**). B presents decorated functors and **Dec<sub>W</sub>**. C presents traversable functors and **Trav**. D presents **DecTrav<sub>W</sub>** and proves one direction of Theorem 4.2 (the other direction is deeply tedious and is formalized in Tealeaves).

## A String Diagrams for Endofunctor Categories

Here we briefly review monoidal string diagrams, a two-dimensional notation for monoidal categories. The primary advantage of such notation is that it inherently “quotients out” mundane structural equalities. Our library Tealeaves is formalized entirely in Coq, where such quotienting is not automatic and the notation is quite noisy, to the point it is hard to see the forest for the trees. Here, string diagrams provide a useful aid that depicts the important structure behind DTMs.

We work in the category **End<sub>Set</sub>** of endofunctors on the category of sets. The objects of this category are endofunctors  $F : \mathbf{Set} \rightarrow \mathbf{Set}$ , with morphisms given by natural transformations. We recall basic definitions.

**Definition A.1.** A functor is a type constructor  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  equipped with a polymorphic operation

$$\text{map}^T : \forall (A, B : \mathbf{Set}), (A \rightarrow B) \rightarrow TA \rightarrow TB$$

satisfying the following two equations:

$$\text{map}^T \text{id}_A = \text{id}_{TA} \tag{A.1}$$

$$\text{map}^T g \cdot \text{map}^T f = \text{map}^T (g \cdot f) \tag{A.2}$$

**Definition A.2.** A natural transformation  $\phi : F \rightarrow G$  between functors  $F$  and  $G$  is a polymorphic function

$$\phi : \forall (A : \mathbf{Set}), FA \rightarrow GA$$

satisfying the follow equation for all  $f : A \rightarrow B$ :

$$\phi \cdot \text{map}^F f = \text{map}^G f \cdot \phi \tag{A.3}$$

In the notation, functors are depicted as wires:

$$F \text{ ————— } F$$

The composition  $G \cdot F$  of two functors is depicted by putting them next to each other (note that our notation places the outer functor at the top):

$$G \text{ ————— } G$$

$$F \text{ ————— } F$$

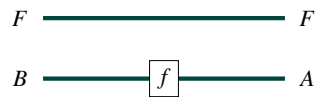
Given a particular set  $A$ , the object  $FA$  can be depicted as shown. (Strictly speaking, we embed the object into  $\mathbf{End}_{\text{Set}}$  as a constant functor.)



A natural transformation  $\phi : G \Rightarrow F$  is depicted as a node along the wire:



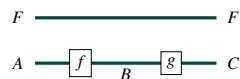
Using the embedding mentioned previously, we can also depict ordinary functions  $f : A \rightarrow B$  as boxes in the same way. (Strictly speaking we view the function as a natural transformation between constant functors.) Given a function  $f : A \rightarrow B$ , the function  $\text{map} f$  is drawn as shown:



The laws of functors are absorbed into the string-diagrammatic notation. For both (A.1) and (A.2), both sides of the equality are depicted exactly the same.

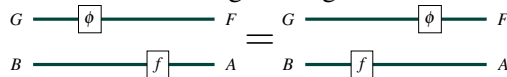


$$\text{map}^T \text{id}_A = \text{id}_{TA}$$



$$\text{map}^T g \cdot \text{map}^T f = \text{map}^T (g \cdot f)$$

Similarly, the defining equation of naturality (Equation (A.3)) is manifest in the fact that boxes slide along wires like beads along a string.



$$\text{map}^G f \cdot \phi = \phi \cdot \text{map}^F f \quad (\text{A.4})$$

The identity functor  $\mathbb{1}$  is not explicitly depicted in the notation. As a special case, natural transformations from the identity functor are depicted as boxes that are not connected to an input wire.



Likewise, transformations to the identity functor are drawn as “terminal” boxes.



## A.1 Monads

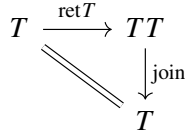
For review and to fix notation, we offer a string-diagrammatic proof that all monads give rise to Kleisli-presented monads. To further cut down on syntactic noise, owing to our frequent use of the monad operations and other special operations later, we depict monads with a special notation. The wire representing a monad  $T$  is drawn in blue, and its operations are depicted as blue circles rather than labelled boxes. Labels on the wires disambiguate these operations for readers without the benefit of reading this manuscript in full color.



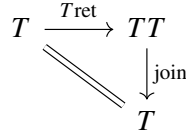
**Definition A.3.** A monad is a functor  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  that forms a monoid in the category of endofunctors. That is, it is equipped with a unit and multiplication operation

$$\text{ret}^T : \mathbb{1} \Rightarrow T \quad \text{join}^T : TT \Rightarrow T$$

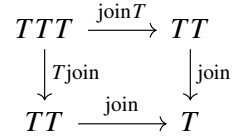
that make the following diagrams commute.



Left identity

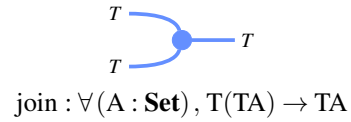
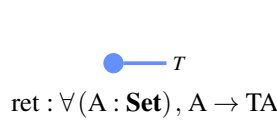


Right identity

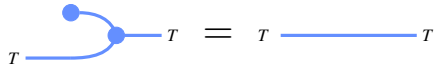


Associativity

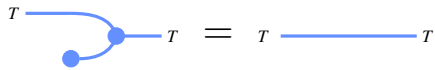
String-diagrammatically, the monoid operations are depicted as follows:



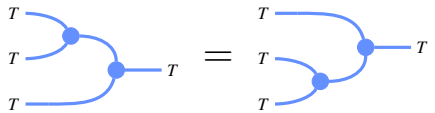
The monoid laws take on the following form:



$$\text{join}_A \cdot \text{ret}_{TA} = \text{id}_{TA} \tag{A.5}$$



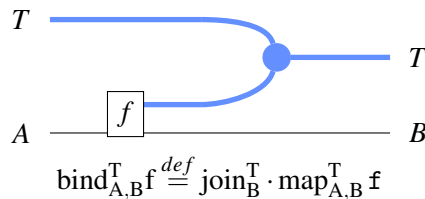
$$\text{join}_A \cdot \text{map}(\text{ret}_A) = \text{id}_{TA} \tag{A.6}$$



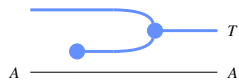
$$\text{join}_A^T \cdot \text{join}_{TA}^T = \text{join}_A^T \cdot \text{map}^T(\text{join}_A^T) \tag{A.7}$$

**Lemma A.4** (Proof of Lemma 2.3). *Every monad gives rise to a Kleisli-presented monad.*

*Proof.* The operation  $\text{ret}$  is the same to both presentations, and  $\text{bind}$  is defined



Proof of (2.1).



$$= \begin{array}{c} T \text{ ————— } T \\ A \text{ ————— } A \end{array}$$

Apply the monad right unit law (A.6).

Proof of (2.2).

$$\begin{array}{c} T \bullet \\ \text{ } \\ A \text{ — } \boxed{f} \text{ — } B \end{array}$$

$$= \begin{array}{c} T \\ \text{ } \\ A \text{ — } \boxed{f} \text{ — } B \end{array}$$

Apply the monad left unit law (A.5).

Proof of (2.3).

$$\begin{array}{c} T \text{ ————— } T \\ \text{ } \\ A \text{ — } \boxed{f} \text{ — } \boxed{g} \text{ — } C \end{array}$$

$$= \begin{array}{c} T \text{ ————— } T \\ \text{ } \\ A \text{ — } \boxed{f} \text{ — } \boxed{g} \text{ — } C \end{array}$$

Bend wires.

$$= \begin{array}{c} T \text{ ————— } T \\ \text{ } \\ A \text{ — } \boxed{f} \text{ — } \boxed{g} \text{ — } C \end{array}$$

Apply monad associativity (A.7).

□

Note that a proof in the other direction (that a Kleisli-presented monad gives rise to a monad) must be given without the aid of string diagrams: to employ the notation requires already knowing that the wires are functors and the morphisms are natural transformations—but in the other direction these are among the things to be shown.

## B The category $\text{Dec}_W$

Let a monoid  $\langle W, \cdot, 1_W \rangle$  be given.

### B.1 The Writer bimonad

The Writer bimonad is defined from the following four natural transformations:

$$\text{extr} : \forall (A : \mathbf{Set}), W \times A \rightarrow A$$

$$\text{ret} : \forall (A : \mathbf{Set}), A \rightarrow W \times A$$

$$\text{dup} : \forall (A : \mathbf{Set}), W \times A \rightarrow W \times (W \times A)$$

$$\text{join} : \forall (A : \mathbf{Set}), W \times (W \times A) \rightarrow W \times A$$

We depict these as red wires with the following shapes:

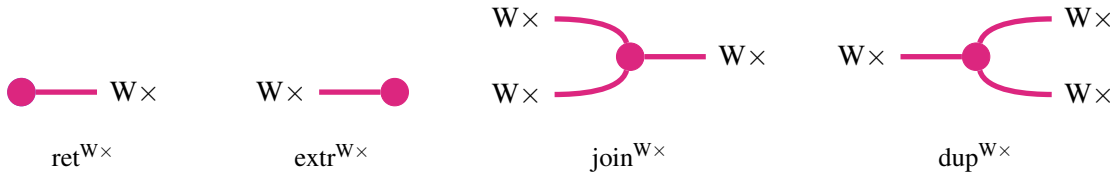


Figure 10: Writer operations

The duplication comonoid on  $W$  induces a comonad structure, while the monoid structure induces a monad. We have the following monad and comonad laws:

(B.1)

Writer associativity

(B.2)

Writer coassociativity

(B.3)

Writer monad unit laws

(B.4)

Writer comonad counit laws

Additionally, the monad and comonad are related by four laws.

- First, a *butterfly law* depicted as such:

(B.5)

Here, the crossing of wires indicates the tensorial strength operation

$$st : \forall (A : \mathbf{Set}), W \times (W \times A) \rightarrow W \times (W \times A)$$

$$\text{st}_A(w_1, (w_2, a)) = (w_2, (w_1, a))$$

The butterfly law is an equivalence between two natural transformations of type  $W \times W \Rightarrow W \times W$ . Let some  $(w_1, w_2)$  be given. On the left of (B.5), we multiply to obtain  $w_1 \cdot w_2$ , then duplicate to obtain

$$(w_1 \cdot w_2, w_1 \cdot w_2).$$

On the right side, we first duplicate pointwise to obtain  $(w_1, w_1, w_2, w_2)$ , swap the middle components to obtain  $(w_1, w_2, w_1, w_2)$ , then multiply the left and right pairs separately to obtain

$$(w_1 \cdot w_2, w_1 \cdot w_2).$$

- Second, we have a *cup law*.

$$\begin{array}{c} W \times \\ W \times \end{array} \begin{array}{c} \bullet \\ \bullet \end{array} = \begin{array}{c} W \times \\ W \times \end{array} \begin{array}{c} \curvearrowright \\ \bullet \\ \curvearrowleft \end{array} \begin{array}{c} \bullet \\ \bullet \end{array} \quad (\text{B.6})$$

This law states that duplicating  $1_W$  is the same as having two copies of  $1_W$ .

- Third, we have a *cap law*.

$$\begin{array}{c} \bullet \\ \bullet \end{array} \begin{array}{c} \curvearrowleft \\ \bullet \\ \curvearrowright \end{array} \begin{array}{c} W \times \\ W \times \end{array} = \begin{array}{c} \bullet \\ \bullet \end{array} \begin{array}{c} W \times \\ W \times \end{array} \quad (\text{B.7})$$

This law states that deleting two values is the same as adding them, then deleting the result.

- Finally, we have a *baton law*.

$$\begin{array}{c} \bullet \\ \bullet \end{array} \begin{array}{c} \text{---} \\ \bullet \end{array} = \quad (\text{B.8})$$

This law states that pairing a value with  $1_W$ , then deleting it, is the same as doing nothing.

In this section, let  $W$  be a monoid. For our applications,  $W$  is ordinarily the free monoid  $\text{list}B$ , or lists of binder annotations.

## B.2 Decorated functors

**Definition B.1.** A functor  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  decorated by  $W$  is a right coalgebra of the writer bimonad as an object of  $\mathbf{End}_{\mathbf{Set}}$ . Explicitly,  $T$  comes equipped with a natural transformation

$$\text{dec} : T \Rightarrow T(W \times -)$$

subject to the following commutativity conditions.

$$\begin{array}{ccc} T & \xrightarrow{\text{dec}} & T(W \times -) \\ & \searrow & \downarrow T \text{extr}^{W \times} \\ & & T \end{array}$$

Countit law

$$\begin{array}{ccc} T & \xrightarrow{\text{dec}} & T(W \times -) \\ \downarrow \text{dec} & & \downarrow T \text{dup}^{W \times} \\ T(W \times -) & \xrightarrow{\text{dec}(W \times)} & T(W \times (W \times -)) \end{array}$$

Coassociativity

String-diagrammatically, the decoration operation is depicted as follows.

$$\begin{array}{c}
 \text{---} T \quad \text{---} T \\
 \quad \quad \quad \curvearrowright \\
 \quad \quad \quad \bullet \\
 \quad \quad \quad \curvearrowleft \\
 \quad \quad \quad \text{---} W \times
 \end{array}$$

$$\text{dec}^T : \forall (A : \mathbf{Set}), TA \rightarrow T(W \times A)$$

The counit and co-associativity law take on the following shapes:

$$\begin{array}{c}
 \text{---} T \\
 \quad \quad \quad \curvearrowright \\
 \quad \quad \quad \bullet \\
 \quad \quad \quad \curvearrowleft \\
 \quad \quad \quad \text{---} T
 \end{array}
 =
 \text{---} T \quad \text{---} T$$

$$\text{map}^T \text{extr}_A^{W \times} \cdot \text{dec}_A = \text{id}_{TA} \quad (\text{B.9})$$

$$\begin{array}{c}
 \text{---} T \\
 \quad \quad \quad \curvearrowright \\
 \quad \quad \quad \bullet \\
 \quad \quad \quad \curvearrowleft \\
 \quad \quad \quad \text{---} W \times
 \end{array}
 =
 \begin{array}{c}
 \text{---} T \\
 \quad \quad \quad \curvearrowright \\
 \quad \quad \quad \bullet \\
 \quad \quad \quad \curvearrowleft \\
 \quad \quad \quad \text{---} W \times
 \end{array}$$

$$\text{dec}_{W \times A} \cdot \text{dec}_A = \text{map}^T \text{dup}_A^{W \times} \cdot \text{dec}_A \quad (\text{B.10})$$

**Definition B.2.** A decoration-preserving morphism  $\phi : F \Rightarrow G$  between decorated functors is a natural transformation that commutes with decorations.

$$\begin{array}{ccc}
 F & \xrightarrow{\phi} & G \\
 \downarrow \text{dec}^F & & \downarrow \text{dec}^G \\
 F(W \times -) & \xrightarrow{\phi(W \times)} & G(W \times -)
 \end{array}$$

Diagrammatically, a decoration-preserving morphism is a natural transformation that can slide past the decoration operation on the  $T$  wire.

$$\begin{array}{c}
 \text{---} T_1 \quad \boxed{\phi} \quad \text{---} T_2 \\
 \quad \quad \quad \curvearrowright \\
 \quad \quad \quad \bullet \\
 \quad \quad \quad \curvearrowleft \\
 \quad \quad \quad \text{---} W \times
 \end{array}
 =
 \begin{array}{c}
 \text{---} T_1 \quad \text{---} T_2 \\
 \quad \quad \quad \curvearrowright \\
 \quad \quad \quad \bullet \\
 \quad \quad \quad \curvearrowleft \\
 \quad \quad \quad \text{---} W \times
 \end{array}$$

$$\text{dec}_A^{T_2} \cdot \phi_A = \phi_{W \times A} \cdot \text{dec}_A^{T_1} \quad (\text{B.11})$$

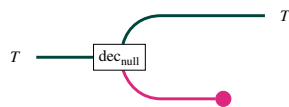
**Definition B.3.** Given any functor  $T$ , the null decoration is defined as the one pairing every element with the monoid unit.

$$\begin{array}{c}
 \text{---} T \\
 \quad \quad \quad \curvearrowright \\
 \quad \quad \quad \boxed{\text{dec}_{\text{null}}} \\
 \quad \quad \quad \curvearrowleft \\
 \quad \quad \quad \text{---} W \times
 \end{array}
 \stackrel{\text{def}}{=}
 \begin{array}{c}
 \text{---} T \\
 \quad \quad \quad \bullet \\
 \quad \quad \quad \text{---} W \times
 \end{array}$$

$$(\text{B.12})$$

For the previous definition to be legitimate, we must validate the counit and coassociativity laws.

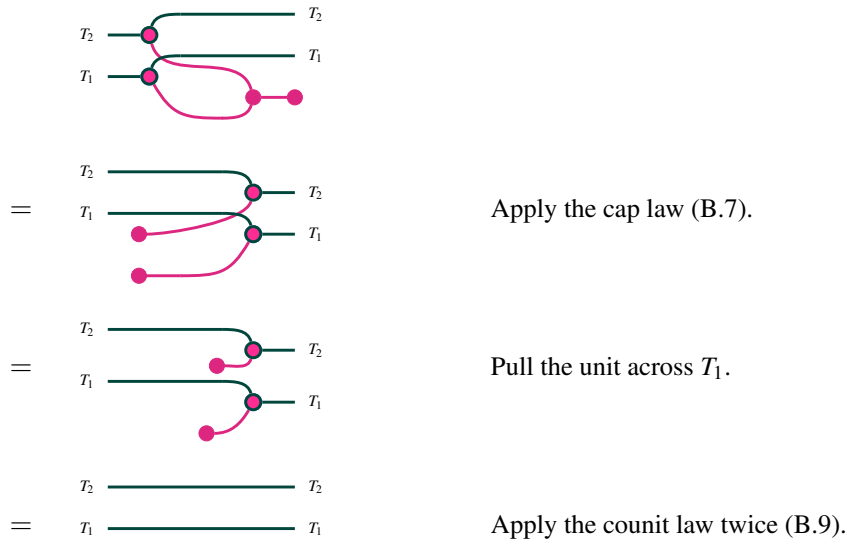
- Proof of counit law (B.9).



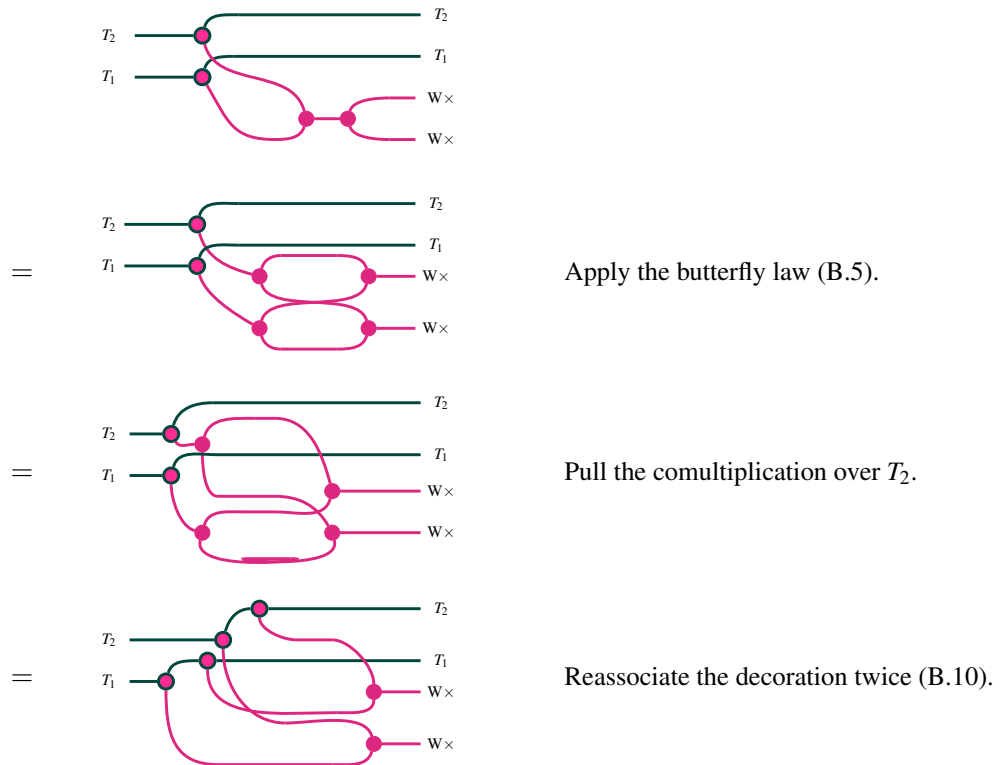


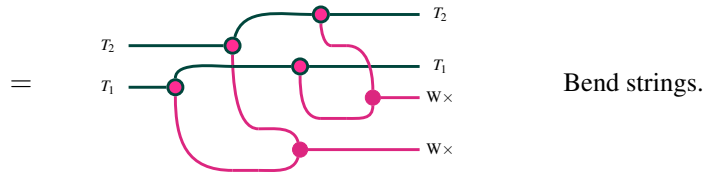
*Proof.* We must validate that (B.14) defines a valid decoration on  $T_2T_1$ .

- Proof of counit law (B.9).



- Proof of coassociativity law (B.10).





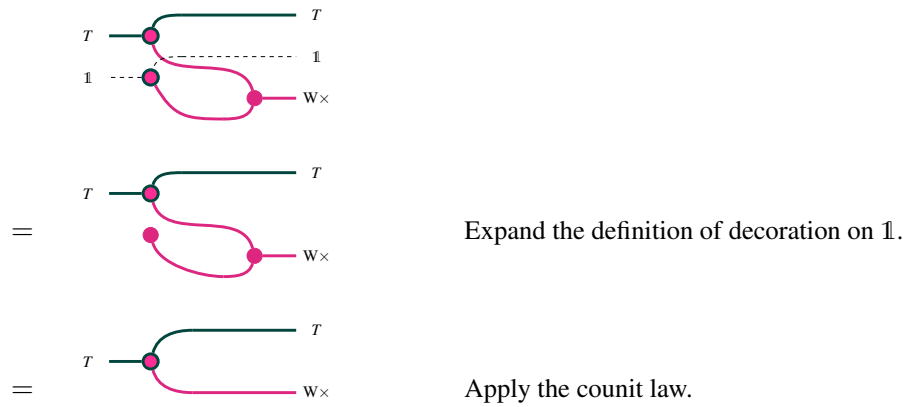
□

**Lemma B.6.** *There is a strict monoidal category  $\mathbf{Dec}_W$  from the following data:*

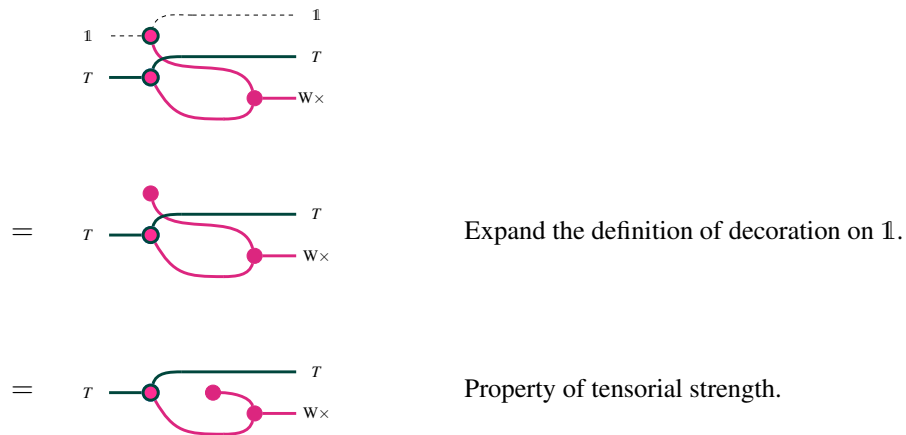
- *Objects are endofunctors  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  paired with a decoration.*
- *Morphisms are natural transformations  $F \Rightarrow G$  that commute with the decorations of  $F$  and  $G$ .*
- *The identity is the identity functor paired with the null decoration.*
- *Tensor product is given by composition, with decorations composed as in Lemma B.5.*

*Proof.*

- The decoration for  $T\mathbb{1}$  is the same decoration operation as on  $T$  itself.



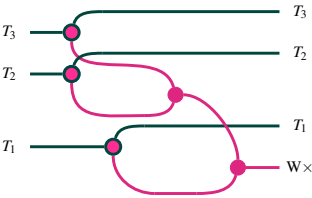
- The decoration for  $\mathbb{1}T$  is the same decoration operation as on  $T$  itself.

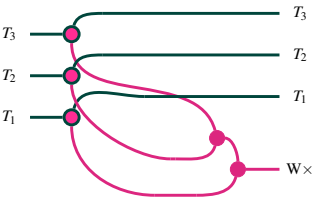





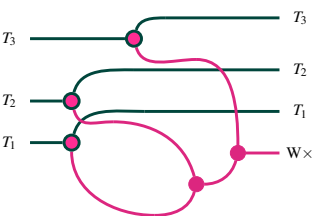
=  Apply the counit law.

- The decoration defined on  $T_3(T_2T_1)$  is the same as the one defined for  $(T_3T_2)T_1$ .



= 

= 

= 

Property of tensorial strength.

Reassociate with the writer monad.

Bend wires.

□

The previous lemma is related to the following well-known fact from abstract algebra: Let  $B$  be a *bialgebra*, i.e. a bimonoid in the monoidal category of vector spaces under tensor product. Then the class of all (left or right) (co-)algebras of  $B$  forms a monoidal category. See, for example, Section 4.1 of [7]. Lemma B.6 applies essentially the same idea in the category of endofunctors, which is only slightly complicated by the fact that  $\mathbf{End}_{\mathbf{Set}}$ , unlike  $\mathbf{Vect}$ , is not symmetric category (as  $FG$  is usually not isomorphic to  $GF$ ). However, the result goes through by using the tensorial strength where the symmetry would ordinarily be used.

**Definition B.7.** A decorated monad is a monoid  $\mathbf{Dec}_W$ . Explicitly, it is a monad  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  that is a decorated functor such that the following diagrams commute.

$$\begin{array}{ccc}
 \mathbb{1} & \xrightarrow{\text{ret}^T} & T \\
 \downarrow \text{ret}^{(W \times)} & & \downarrow \text{dec} \\
 (W \times -) & \xrightarrow{\text{ret}^T} & T(W \times -)
 \end{array}$$

Decoration/unit law

$$\begin{array}{ccc}
 TT & \xrightarrow{\text{dec} \square \text{dec}} & T(W \times (T(W \times -))) & \xrightarrow{T\text{st}^{W,T}(W \times)} & T(T(W \times (W \times -))) \\
 \downarrow \text{join} & & & & \downarrow \text{join}^T \square \text{join}^{W \times} \\
 T & \xrightarrow{\text{dec}} & T(W \times -) & & T(W \times -)
 \end{array}$$

Decoration/join law

These laws correspond to the follow to string diagrams.

$$\text{dec} \cdot \text{ret}^T = \text{ret}^T \cdot \text{ret}^{W \times} \tag{B.15}$$

$$\text{dec}_A \cdot \text{join}_A = \text{join}_{W \times A} \cdot \text{map}^T(\text{shift}_A) \cdot \text{dec}_{T(W \times A)} \cdot \text{map}^T(\text{dec}_A) \tag{B.16}$$

The following lemma is immediate.

**Lemma B.8.** *Let  $T$  be monad and let  $W$  a monoid. Then  $T(W \times -)$  is a monad with unit and multiplication given as follows:*



Figure 17: Composite monad operations

**Lemma B.9.** *A decorated monad is equivalently a decorated functor and a monad whose decoration is a monad homomorphism between  $T$  and  $T(W \times -)$ .*

*Proof.* In light of Lemma B.8, the statement is just another way of reading diagrams (B.15) and (B.16).  $\square$

## C The Category $\mathbf{Trav}$

**Definition C.1.** *An applicative functor, or strong<sup>5</sup> lax monoidal functor, is a type constructor  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  equipped with operations*

$$\begin{aligned} \text{pure}^F &: \forall (A : \mathbf{Set}), A \rightarrow FA \\ (\otimes)^F &: \forall (A B : \mathbf{Set}), F(A \rightarrow B) \rightarrow FA \rightarrow FB \end{aligned}$$

subject to the following equations (note that  $\otimes$  is left-associative).

$$\text{pure id} \otimes a = a \quad (\text{C.1}) \quad g \otimes (f \otimes a) = \text{pure } (\cdot) \otimes g \otimes f \otimes a \quad (\text{C.3})$$

$$\text{pure } f \otimes \text{pure } a = \text{pure } (fa) \quad (\text{C.2}) \quad f \otimes \text{pure } a = \text{pure } (f \mapsto fa) \otimes f \quad (\text{C.4})$$

Here we have defined applicative functors in terms of the  $\otimes$  operation, rather than an inter-definable multiplication operation of type

$$(\otimes)^F : \forall (A B : \mathbf{Set}), FA \times FB \rightarrow F(A \times B).$$

The equivalence of these presentations is discussed in McBride and Paterson [26].

**Definition C.2.** *An applicative morphism  $\phi : F \Rightarrow G$  is a natural transformation that respects the monoidal structure of applicative functors:*

$$\phi (\text{pure}^F a) = \text{pure}^G a \quad (\text{C.5}) \quad \phi (f \otimes a) = \phi f \otimes \phi a \quad (\text{C.6})$$

An important special case of applicative functors are constant functors mapping all sets to some monoid  $M$  and all functions to  $\text{id}_M$ . In this case, both  $(\otimes)$  and  $(\otimes)$  can be identified with the monoid multiplication, while  $\text{pure}$  can be identified with identified with (the constant function returning) the monoid identity element. The applicative functor laws follow from the monoid laws. Unsurprisingly, morphisms between constant applicative functors are simply monoid homomorphisms.

**Definition C.3.** *A traversable functor  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  has a distributive law  $\text{dist} : TF \rightarrow FT$  over applicative functors, natural in  $F$  and respecting the monoidal structure of applicative functors. That is, for all applicative functors  $F_1, F_2$  and applicative morphisms  $F_1 \xrightarrow{\phi} F_2$ , we have the following commutative diagrams.*

$$\begin{array}{ccc} T & \xrightarrow{\text{dist}_I} & T \\ & \searrow \text{id}_T & \nearrow \\ & & T \end{array}$$

Unitality

$$\begin{array}{ccc} TFG & \xrightarrow{\text{dist}_F G} & FTG \\ & \searrow \text{dist}_{F.G} & \downarrow F \text{dist}_G \\ & & FGT \end{array}$$

Linearity

$$\begin{array}{ccc} TF & \xrightarrow{\text{dist}_F^T} & FT \\ \downarrow T\phi & & \downarrow \phi_T \\ TG & \xrightarrow{\text{dist}_G^T} & GT \end{array}$$

Naturality

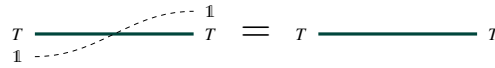
<sup>5</sup>In the sense of tensorial strengths.

Our notation depicts applicative functors as yellow wires. The distributive law is depicted as the ability to cross a yellow wire over a wire representing a traversable functor. (Note that we can cross  $F$  from underneath  $T$  to over it, but we cannot usually go in the other direction.)

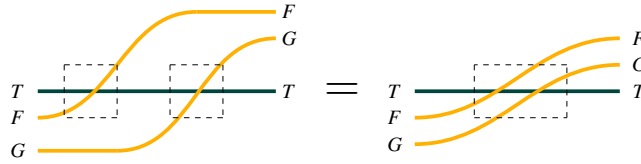


$$\text{dist}^T : \forall (F : \text{Applicative}) (A : \mathbf{Set}), T(FA) \rightarrow F(TA)$$

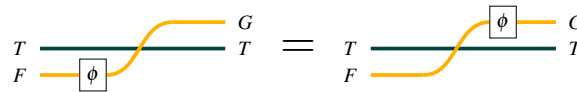
The laws of traversals have the following string-diagrammatic presentation. The first says that crossing the “invisible” identity wire has no effect on  $T$ . For applicative functors  $F$  and  $G$ , the second law states that there is no distinction between considering their composition  $FG$  as an applicative functor and crossing this functor over  $T$ , or crossing the  $F$  and  $G$  wires over  $T$  individually in two steps. The third law is merely a naturality requirement allowing us to slide applicative morphisms along crossed wires.



$$\text{dist}_{\mathbf{1}, A} = \text{id}_{TA} \quad (\text{C.7})$$



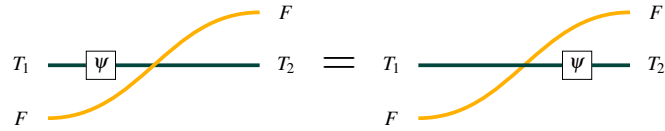
$$\text{map}^F(\text{dist}_{G,A}) \cdot \text{dist}_{F,GA} = \text{dist}_{F,G,A} \quad (\text{C.8})$$



$$\text{dist}_{G,A} \cdot \text{map}^T(\phi_A) = \phi_{TA} \cdot \text{dist}_{F,A} \quad (\text{C.9})$$

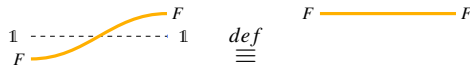
**Definition C.4.** A traversable morphism  $\psi : T_1 \Rightarrow T_2$  is a natural transformation that commutes with distribution.

$$\begin{array}{ccc} T_1 F & \xrightarrow{\text{dist}^{T_1, F}} & F T_1 \\ \downarrow \psi_F & & \downarrow F \psi \\ T_2 F & \xrightarrow{\text{dist}^{T_2, F}} & F T_2 \end{array}$$



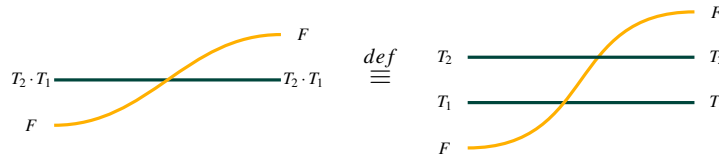
$$\text{dist}_{F,A}^{T_2} \cdot \psi_{FA} = \text{map}^F(\psi_A) \cdot \text{dist}_{F,A}^{T_1} \tag{C.10}$$

**Lemma C.5.** *The identity functor is equipped with a unique traversal that is just the identity function.*



*Proof.* All equations involved are trivial. □

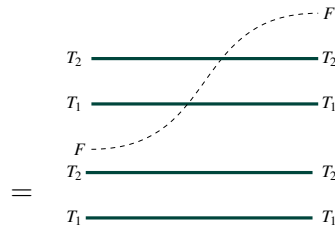
**Lemma C.6.** *Traversable functors are closed under composition by the following construction that distributes F over  $T_2 \circ T_1$  by crossing  $T_1$  and  $T_1$  individually:*



*Proof.*

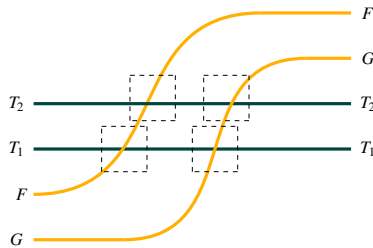
We must validate that (C.7)–(C.9) are satisfied.

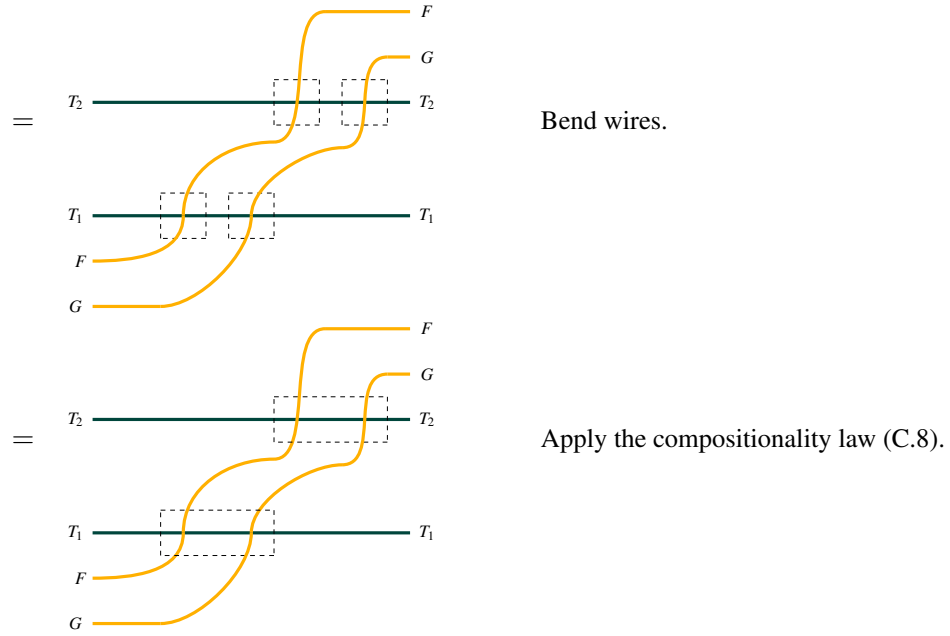
- Distribute over the identity functor



Apply the identity law.

- Proof of composition law Distribute over the composition of two applicative functors





□

**Lemma C.7** (Category **Trav**). *There is a strict monoidal category **Trav** from the following data:*

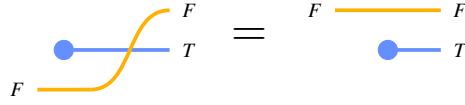
- *Objects are endofunctors  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  paired with a traversal.*
- *Morphisms are natural transformations  $T_1 \Rightarrow T_2$  that commute with traversals.*
- *The identity is the identity functor paired with its unique traversal.*
- *Tensor product is given by composition, with traversals composed as in Lemma C.6.*

*Proof.* This mostly comes down to the previous verification that the identity functor is traversable and that such functors are closed under composition. To form a category, one must check the easy fact that traversable morphisms are also closed under composition. The monoidal structure of this category is inherited from  $\mathbf{End}_{\mathbf{Set}}$ , after verifying the trivial equations stating that the composite distributive law in Lemma C.6 is associative and respects composition with the identity functor. □

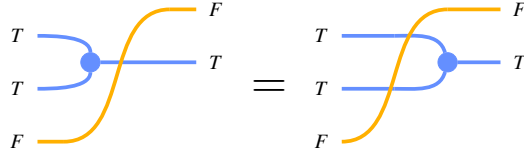
**Definition C.8.** *A traversable monad is a monoid in **Trav**. Explicitly, the requirement is that  $T$  is both a monad and a traversable functor such that the following diagrams commute for all applicative  $F$ .*

$$\begin{array}{ccc}
 F & \xrightarrow{\text{ret}_F} & TF \\
 \searrow F\text{ret} & & \downarrow \text{dist} \\
 & & FT
 \end{array}
 \qquad
 \begin{array}{ccccc}
 TTF & \xrightarrow{T\text{dist}} & TFT & \xrightarrow{\text{dist}T} & FTT \\
 \downarrow \mu_F & & & & \downarrow F\mu \\
 TF & \xrightarrow{\text{dist}} & & & FT
 \end{array}$$

Besides stipulating that  $T$  is both a monad and a traversable functor, the force of Definition C.8 is that  $\text{ret}$  and  $\text{join}$  are required to satisfy (C.9). These laws have the following depictions:



$$\text{dist}_{F,A} \cdot \text{ret}_{FA} = \text{map}^F(\text{ret}_A^T) \tag{C.11}$$



$$\text{dist}_{F,A} \cdot \text{join}_{FA} = \text{map}^F(\text{join}_A) \cdot \text{dist}_{F,TA} \cdot \text{map}^T(\text{dist}_{F,A}) \tag{C.12}$$

## D The Category $\text{DecTrav}_W$

**Definition D.1.** A decorated traversable functor is decorated and traversable functor such that the following diagram commutes.

$$\begin{array}{ccc} TF & \xrightarrow{\text{dist}_F} & FT \\ \downarrow \text{dec}_F & & \downarrow F\text{dec} \\ T(W \times F-) & \xrightarrow{T \text{dist}_F^{W \times}} T(F(W \times -)) \xrightarrow{\text{dist}_F^T(W \times)} & FT(W \times -) \end{array}$$

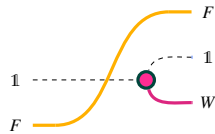
This has the following string-diagrammatic depiction.

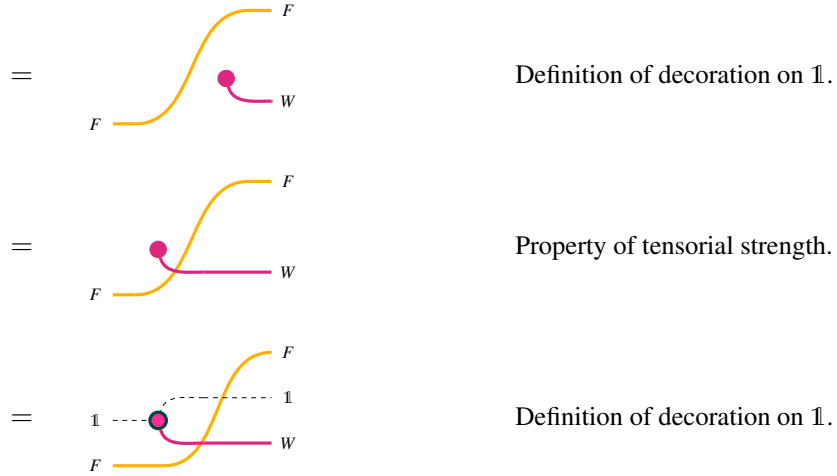


$$\text{map}^F(\text{dec}_A) \cdot \text{dist}_{F,A} = \text{dist}_{F,W \times A} \cdot \text{map}^T(\text{dist}_{F,A}^{W \times}) \cdot \text{dec}_{FA}^T \tag{D.1}$$

**Lemma D.2.** The identity functor is decorated-traversable.

*Proof.* We verify that the identity respects (D.1).

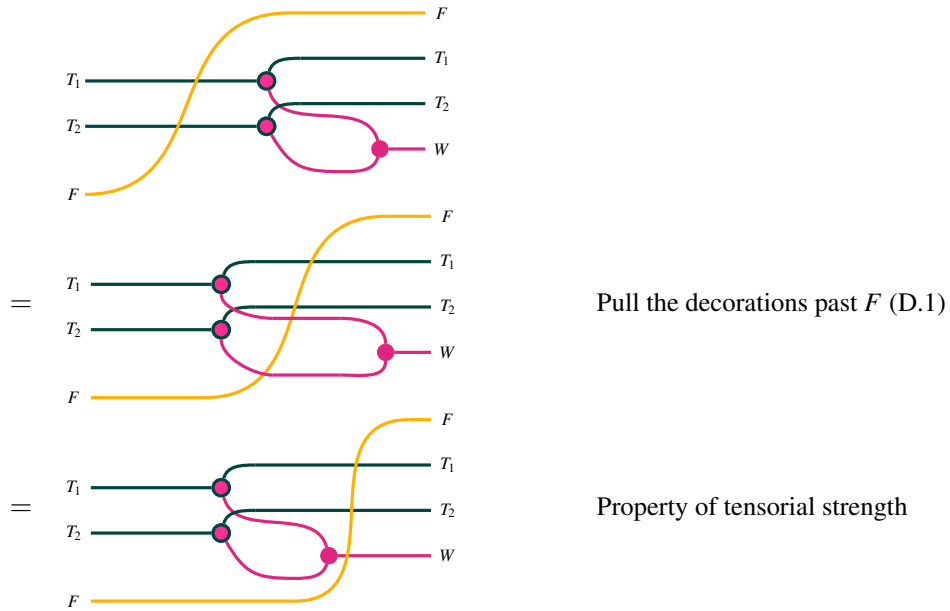




□

**Lemma D.3.** *Decorated traversable functors are closed under composition.*

*Proof.* We verify that composition respects (D.1).



□

**Lemma D.4.** *There is a strict monoidal category  $\mathbf{DecTrav}_W$  from the following data:*

- *Objects are endofunctors  $T : \mathbf{Set} \rightarrow \mathbf{Set}$  paired with a traversal and a decoration.*
- *Morphisms are natural transformations  $T_1 \Rightarrow T_2$  that commute with traversals and decorations.*
- *The identity is the identity functor paired with its unique traversal and the null decoration.*
- *Tensor product is given by composition, with traversals and decorations composed as described above.*



*Proof.* We verified that the identity functor is decorated-traversable and that decorated-traversable functors are closed under composition. The remaining properties are already verified as part of Lemmas B.6 and C.7.  $\square$

### D.1 Decorated traversable monads

**Definition D.5.** A decorated traversable monad is a monoid in  $\mathbf{DecTrav}_W$ .

The force of Definition D.5 can be seen in parts. First, a DTM  $T$  is an object of  $\mathbf{DecTrav}_W$ , hence it is decorated (Definition B.1) and traversable (Definition C.3) and the decoration and traversal commute according to (D.1). Second, the monoid operations  $\text{ret}$  and  $\text{join}$  commute with decoration in the sense of (B.11), for the decoration operations defined in Lemmas B.4 and B.5. Finally, the monoid operations commute with the traversal in the sense of (C.9), according to the traversals defined in Lemmas C.5 and C.6.

In total, a DTM is equipped with 5 operations:

$$\begin{array}{ll}
 \text{map} & (A \ B : \mathbf{Set}) & : (A \rightarrow B) \rightarrow T A \rightarrow T B \\
 \text{ret} & (A : \mathbf{Set}) & : A \rightarrow T A \\
 \text{join} & (A : \mathbf{Set}) & : T(TA) \rightarrow T A \\
 \text{dec} & (A : \mathbf{Set}) & : T A \rightarrow T(W \times A) \\
 \text{dist} & (\text{Applicative } F)(A : \mathbf{Set}) & : T(FA) \rightarrow F(TA)
 \end{array}$$

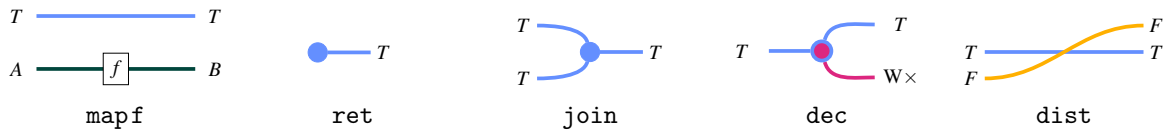


Figure 22: DTM operations

These operations are subject to a total of 19 equations depicted in Figures 23 and 24.

Functor laws	$\text{map}^T \text{id}_A = \text{id}_{TA}$ $\text{map}^T g \cdot \text{map}^T f = \text{map}^T (g \cdot f)$
Naturality	$\text{ret}_B^T \cdot f = \text{map}^T f \cdot \text{ret}_A^T$ $\text{join}_B^T \cdot \text{map}^T (\text{map}^T f) = (\text{map}^T f) \cdot \text{join}_A^T$ $\text{dec}_B^T \cdot \text{map}^T f = \text{map}^T (\text{map}^{W \times} f) \cdot \text{dec}_A^T$ $\text{dist}_B^T \cdot \text{map}^T (\text{map}^F f) = \text{map}^F (\text{map}^T f) \cdot \text{dist}_A^T$
Monad laws	$\text{join}_A^T \cdot \text{ret}_{TA}^T = \text{id}_{TA}$ $\text{join}_A^T \cdot \text{map}^T (\text{ret}_A^T) = \text{id}_{TA}$ $\text{join}_A^T \cdot \text{join}_{TA}^T = \text{join}_A^T \cdot \text{map}^T (\text{join}_A^T)$
Decoration laws	$\text{map}^T (\text{extract}^{W \times}) \cdot \text{dec}_A^T = \text{id}_{TA}$ $\text{dec}_A^T \cdot \text{dec}_A^T = \text{map}^T (\text{dup}^{W \times}) \cdot \text{dec}_A^T$
Decoration/monad coherence	$\text{dec}_A^T \cdot \text{ret}_A^T = \text{ret}_A^T \cdot \text{ret}^{W \times}$ $\text{dec}_A^T \cdot \text{join}_A^T = \text{join}_{W \times A}^T \cdot \text{map}^T (\text{shift}_A^T) \cdot \text{dec}_{T(W \times A)}^T \cdot \text{map}^T (\text{dec}_A^T)$
Traversal laws	$\text{dist}_{I,A}^T = \text{id}_{TA}$ $\text{dist}_{F,G,A}^T = \text{map}^F (\text{dist}_{G,A}^T) \cdot \text{dist}_{F,GA}^T$ $\phi_A \cdot \text{dist}_{F,A}^T = \text{dist}_{G,A}^T \cdot \text{map}^T (\phi_A)$
Traversal/monad coherence	$\text{dist}_{F,A}^T \cdot \text{ret}_{FA}^T = \text{map}^F (\text{ret}_A^T)$ $\text{dist}_{F,A}^T \cdot \text{join}_{FA}^T = \text{map}^F (\text{join}_A^T) \cdot \text{dist}_{F,TA}^T \cdot \text{map}^T (\text{dist}_{F,A}^T)$
Decoration/traversal coherence	$\text{map}^F (\text{dec}_A^T) \cdot \text{dist}_{F,A}^T = \text{dist}_{F,W \times A}^T \cdot \text{map}^T (\text{dist}_{F,A}^{W \times}) \cdot \text{dec}_{FA}^T$

Figure 23: Equational presentation of the axioms of DTMs

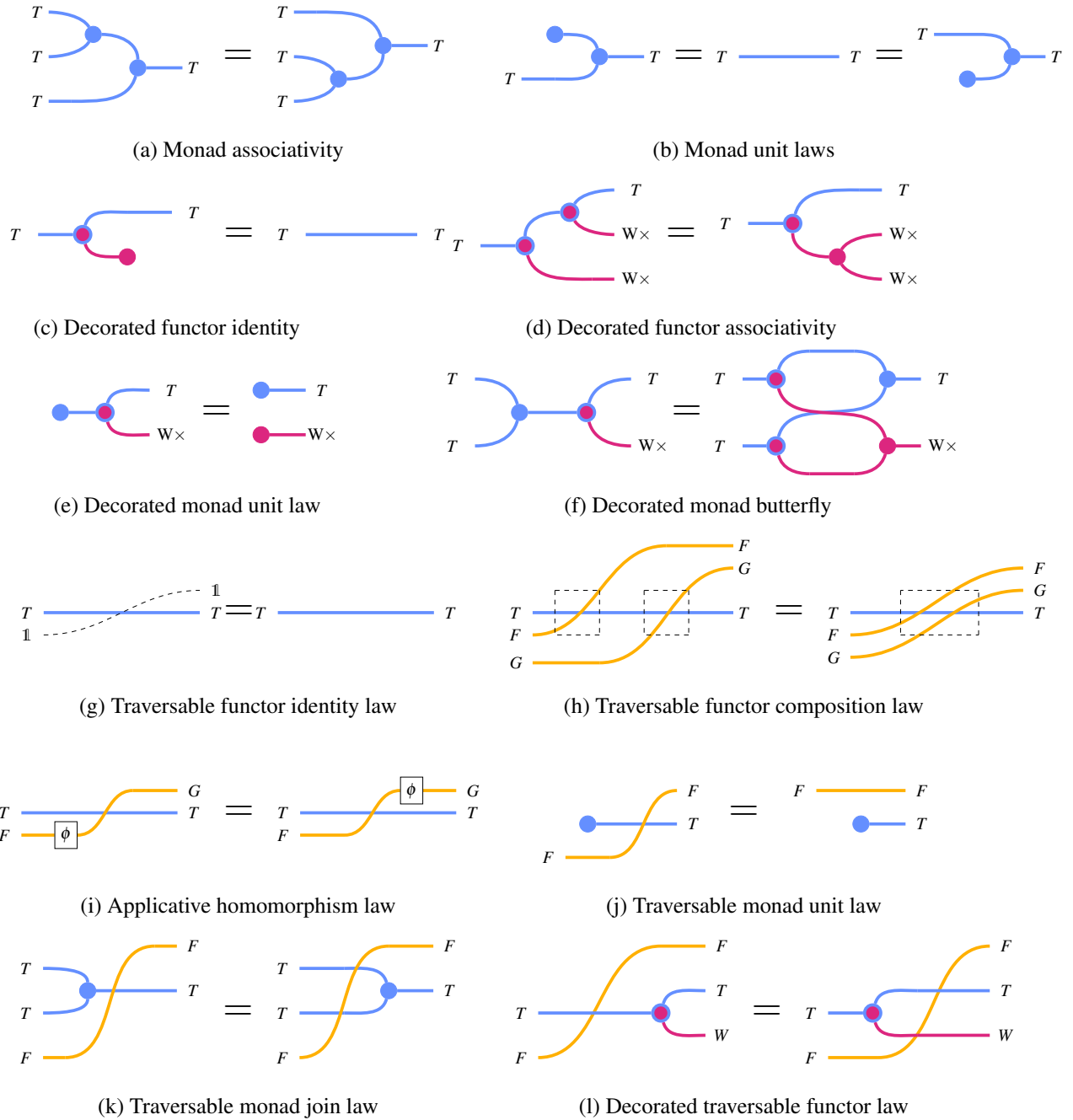
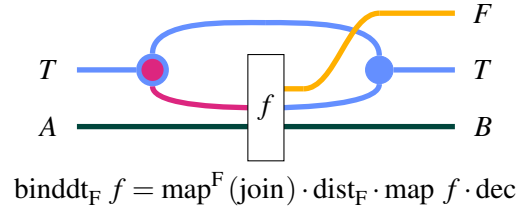


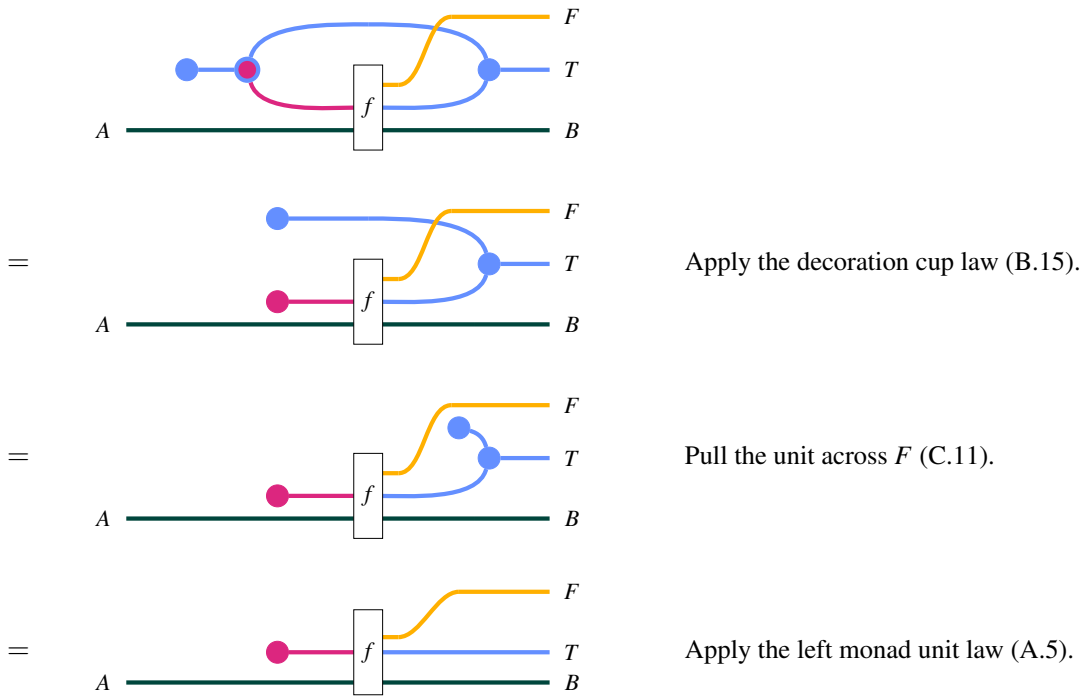
Figure 24: String diagram depictions of the axioms of DTMs

### D.2 Proof of Kleisli Presentation

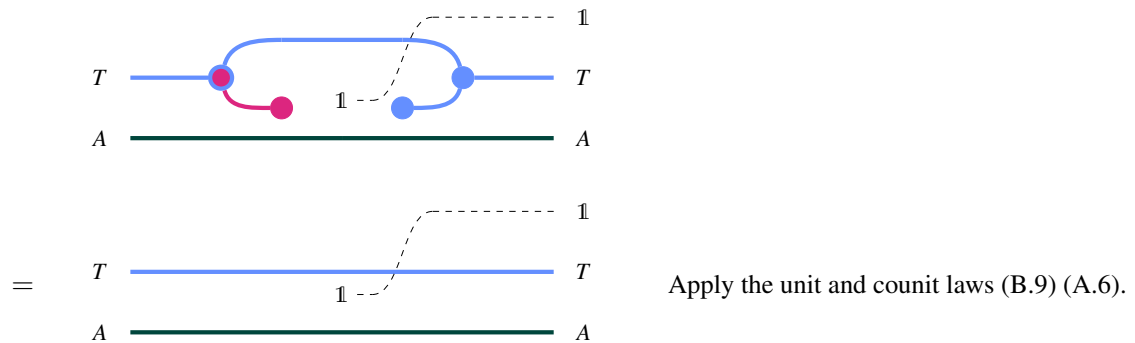
**Lemma D.6.** *Every DTM gives rise to a Kleisli-presented DTM according to the following definition of  $\text{binddt}$ .*



*Proof.* Proof of Equation (4.1):



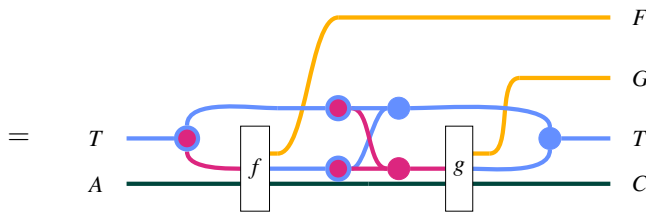
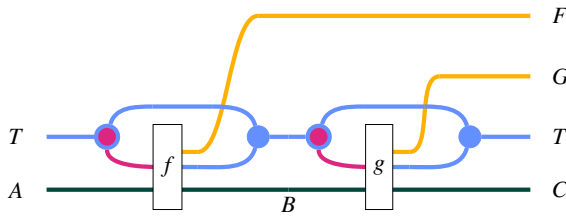
Proof of Equation (4.2):



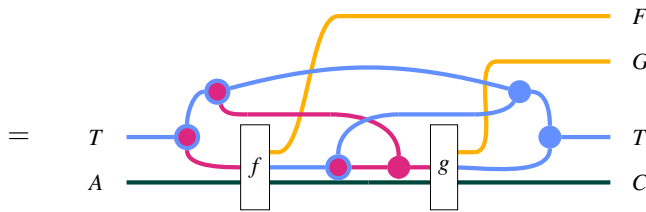


Apply traversal unitary law (C.7).

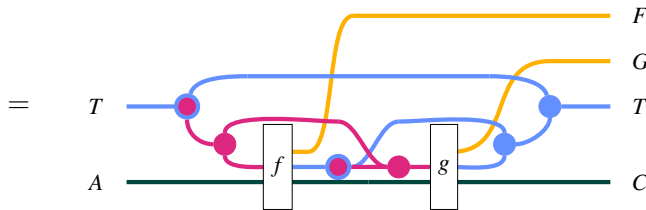
Proof of Equation (4.3):



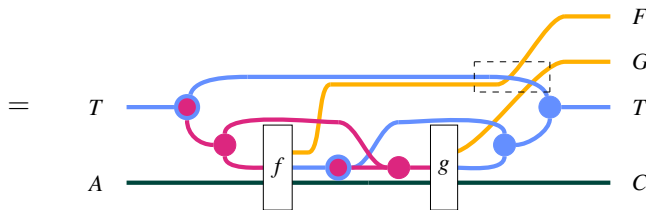
Apply the butterfly law (B.16).



Drag operations past distributions (C.12) (D.1).

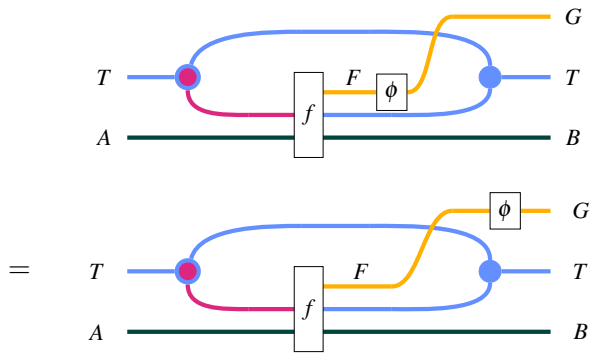


Apply (co)associativity (A.7) (B.10).



Apply traversal composition law (C.8).

Proof of Equation (4.4):



Slide the applicative morphism (C.9).

□