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, most notably the promise of reusing formalized infrastructural lemmas between implementations of different systems. Our Coq framework Tealeaves provides libraries of reusable infrastructure for a raw, locally nameless representation and can be extended to other representations in a modular fashion. 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, yielding a recursion combinator for finitary tree-like datatypes.

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 [39] and programming languages [24], 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, 16, 17, 18, 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 makes two contributions.

- We introduce the strict monoidal category $\mathbf{DecTrav}_W$ of decorated-traversable endofunctors on **Set** for some monoid W (Definition 3.16) and define decorated-traversable monads (DTMs) as monoids in this category (Definition 3.17). Examples of decorated-traversable functors come from the signature functors of languages with variable binding; the free monads they generate are DTMs. These structures admit a string-diagram calculus, which aids in equational reasoning.

S. Staton, C. Vasilakopoulou (Eds.):
Applied Category Theory 2023 (ACT2023)
EPTCS 397, 2023, pp. 98–117, doi:10.4204/EPTCS.397.7

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

As with ordinary (strong) monads [29], 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 [15] 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 why the seemingly ad-hoc equational axioms should be considered "correct." This paper justifies the robustness of 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.[1]

Layout. The rest is laid out as follows. Section 2 contains background on first-order representations of variable binding. We recall that abstract syntax trees, parameterized by the data in the leaves, naturally give rise to a (free) monad. For such monads, the Kleisli axiomatization provides a theory of naïve substitution, but this is not expressive enough to define the *capture-avoiding* substitution operations considered by different representations of variables. Section 3 introduces the endofunctor categories $\mathbf{Dec}_W$, **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. Section 6 concludes.

Functors in this paper have type $\mathbf{Set} \to \mathbf{Set}$ and typically represent parameterized container types like lists, binary trees, and abstract syntax trees. We recall that $\mathbf{End}_{\mathrm{Set}}$ is the strict monoidal category whose objects are endofunctors on **Set**, whose arrows are natural transformations, and whose tensor product is given by composition of functors.

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. The first choice is whether to employ a *first-order* or *higher-order* approach. Higher-order strategies represent variable-binding constructors in the object language as higher-order functions in the metatheory; this sidesteps thorny issues like variable capture but does not shed much light on syntax as defined in, say, a compiler. We are interested in things like verified compilers, so we consider a first-order approach. This style is also simple, intuitive, and well supported by general-purpose proof assistants like Coq [36]. Theoretically it lends itself to the theory of initial algebras, the category-theoretic take on structural recursion [11].

A more or less orthogonal question is whether to consider an *intrinsic* or *extrinsic* (also called *raw*) representation. For instance, intrinsically well-scoped terms exist in some context Γ and can only mention free variables declared in Γ, while instrinsically well-typed terms essentially carry around their own typing judgment. The raw approach posits that a single set of terms simply exists, including ones that are ill-formed and untypable in the formal system. Properties like being well-scoped in Γ are then defined post-hoc as predicates on terms by structural recursion. We consider an extrinsic representation, though in future work we could investigate an intrinsic approach.

Finally, one has a choice about how to represent free and bound variables, i.e. the datatype stored in the *leaves* of syntax trees. Encoding strategies go by names like *fully named*, *de Bruijn indices*, *de Bruijn levels*, *locally named*, *locally nameless*, and variations. DTMs capture what is tree-like about syntax

[1] `https://github.com/dunnl/tealeaves`

without saying anything about the type of data in the leaves, 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 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.
```

Figure 1: Syntax of the lambda calculus in Coq

$$\begin{aligned}
\text{bind}\, f\, (\texttt{Var}\, v) &= f\, v \\
\text{bind}\, f\, (\texttt{App}\, t_1\, t_2) &= \texttt{App}\, (\text{bind}\, f\, t_1)\, (\text{bind}\, f\, t_2) \\
\text{bind}\, f\, (\texttt{Lam}\, b\, t) &= \texttt{Lam}\, b\, (\text{bind}\, f\, t)
\end{aligned}$$

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

$$\Sigma^{\lambda}_{B} X \stackrel{def}{=} X \times X + B \times X$$

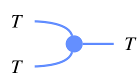
encoding the domain of the two constructors of `term` besides `Var`. $\texttt{term}_B V$ is defined as the least fixpoint $\mu X.\left(V + \Sigma^{\lambda}_{B} X\right)$, i.e. as the smallest solution to the following equation:

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

It is well known that, by its least fixed point construction, a datatype like $\texttt{term}_B$ (for any B) naturally forms a monad. We present monads string-diagrammatically alongside a conventional equational presentation. A general introduction to string diagrams is outside the scope of this paper, but the interested reader may consult [19, 22]. In this paper our calculus depicts a monad T with a blue wire.

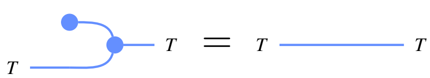
Definition 2.1. *A* monad T *is a functor equipped with two natural transformations*

$\text{ret}^{\mathrm{T}} : \forall (A : \mathbf{Set}), A \to TA$

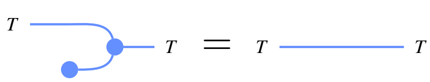


$\text{join}^{\mathrm{T}} : \forall (A : \mathbf{Set}), T(TA) \to TA$

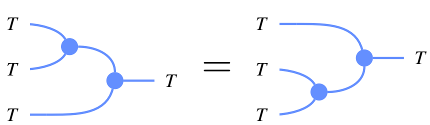
subject to the following laws.



$$\text{join}^{\mathrm{T}} \cdot \text{ret}^{\mathrm{T}} = \text{id} \tag{2.1}$$



$$\text{join}^{\mathrm{T}} \cdot \text{map}^{\mathrm{T}}\left(\text{ret}^{\mathrm{T}}\right) = \text{id} \tag{2.2}$$



$$\text{join}^{\mathrm{T}} \cdot \text{join}^{\mathrm{T}} = \text{join}^{\mathrm{T}} \cdot \text{map}^{\mathrm{T}}\left(\text{join}^{\mathrm{T}}\right) \tag{2.3}$$

The ret^{T} operation constructs a tree from a single leaf—for `term` this is the `Var` constructor. join^{T} flattens a tree-of-trees into a tree by grafting the layers together. map^{T} applies a function to each of the leaves. This presentation is visually pleasing, but fairly abstruse for our purposes. For applications, the following definition is more pragmatic.

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

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

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

$$\text{bind ret} = \text{id} \quad (2.4) \qquad \text{bind } g \cdot \text{bind } f = \text{bind } (\text{bind } g \cdot f) \quad (2.6)$$
$$\text{bind } f \cdot \text{ret} = f \quad (2.5)$$

The equivalence of these definitions is well-known [26].

Lemma 2.3 (Manes, 1976)**.** *Definitions 2.1 and 2.2 are equivalent.*

Figure 2 gives the bind instance for `term`. We note that bind $f\ t$ merely applies f to each variable occurrence in t, replacing it with a subterm. We call this simple replacement operation a *naïve* substitution. (2.4) stipulates that replacing all variables with themselves yields the original t. (2.5) is the definition of bind on `Var`. (2.6) governs the composition of multiple substitutions. The limitations of this naïve notion of substitution become apparent when we turn our attention to situations involving both free and bound variables.

2.1 Variable Encodings

We discuss two exemplary techniques for representing variables.

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 $\texttt{term}_{\mathbb{A}}\mathbb{A}$ corresponds to the following pen-and-paper syntax of lambda terms:

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

Consider 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 λ, 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 [13], natural numbers that describe the "distance" from the occurrence to the abstraction that introduced it, indexing from 0. 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 `fvar` and `bvar` as the names of the left and right injections (respectively) into V.

Because the representation of a bound variable is canonical, there is no need to give arbitrary names to bound variables, hence no need to rename them to avoid conflicts. Lambda abstractions do not need to be annotated with names either, which we formally represent by annotating them with type $B = \mathbf{1} = \{\star\}$, the singleton. This gives the set $\texttt{term}_{\mathbf{1}}\,(\mathbb{A}+\mathbb{N})$, corresponding to the following grammar:

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

A benefit of locally nameless is that substitution of free variables is particularly simple: a variable is free exactly when it is an atom, so it can never be mistaken for a bound variable. Due to this special simplicity,

$$\mathrm{subst}\,x\,u\;(\mathtt{Var}\,v) = \begin{cases} u & \text{if } v = \mathtt{fvar}\,x \\ \mathtt{Var}\,v & \text{else} \end{cases}$$
$$\mathrm{subst}\,x\,u\;(\mathtt{App}\,t_1\,t_2) = \mathtt{App}\;(\mathrm{subst}\,x\,u\,t_1)\;(\mathrm{subst}\,x\,u\,t_2)$$
$$\mathrm{subst}\,x\,u\;(\mathtt{Lam} \star t) = \mathtt{Lam} \star (\mathrm{subst}\,x\,u\,t)$$

(a) Structurally recursive definition

$$\mathrm{subst}\;x\;u\;t = \mathrm{bind}\;\;(\mathrm{subst}_{\mathrm{loc}}\;x\;u)\;\;t$$
$$\mathrm{subst}_{\mathrm{loc}}\;x\;u\;v = \begin{cases} u & \text{if } v = \mathtt{fvar}\;x \\ \mathrm{ret}\;v & \text{else} \end{cases}$$

(b) Definition abstract over a choice of monad

Figure 3: Substitution of atoms in a locally nameless representation

the "correct" notion of substitution for free variables, subst (Figure 3a), happens to be expressible using bind. This operation has the following type, where subst $x\;u\;t$ replaces x in t with u:

$$\mathrm{subst} : \mathbb{A} \to \mathtt{term_1}\,(\mathbb{A}+\mathbb{N}) \to \mathtt{term_1}(\mathbb{A}+\mathbb{N}) \to \mathtt{term_1}\,(\mathbb{A}+\mathbb{N})$$

Figure 3b defines subst in terms of bind and a function $\mathrm{subst}_{\mathrm{loc}}$ that prescribes the "local" effect of substitution on individual occurrences. Decomposing subst like this practical value because $\mathrm{subst}_{\mathrm{loc}}$ does not depend on the particulars of `term`, so this definition is given abstractly over a monad T. This also means we can employ the monad laws to reason about it, exemplified in the following lemma.

Lemma 2.4. *Let T be any monad, let $x \neq y$ be atoms, and let $t[x \mapsto u]$ denote* subst $x\;u\;t$, *defined abstractly in T. Substitution has the following properties:*[2]

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

Lemma 2.4 is easily proven abstractly over T by appealing to equations (2.4)–(2.6). On the other hand, here is a lemma that cannot even be stated, much less 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.

However, subst is not the main operation of locally nameless. That distinction belongs instead to an operation called *opening*, defined in Figure 4a. This operation is used to define β-reduction, with the β-conversion rule taking the form $(\lambda t)u =_\beta t^u$. Here, t^u stands for the opening of t by u, defined by replacing all indices in t previously bound to the outermost λ with u. (Note that the replaced variables are actually de Bruijn indices rather than free variables, hence this is not a substitution of atoms.) Unlike with atoms, the replaced indices do not have to share a common representation, as 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 outermost λ, for instance. Therefore open is defined with an auxiliary function that maintains a count of how many binders we have gone under during

[2] Where x is used as a term, it is understood as the atomic term ret $(\mathtt{fvar}\;x)$. 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.

$$\text{open} : \texttt{term}_{\mathbf{1}}(\mathbb{A}+\mathbb{N}) \to \texttt{term}_{\mathbf{1}}(\mathbb{A}+\mathbb{N}) \to \texttt{term}_{\mathbf{1}}(\mathbb{A}+\mathbb{N})$$
$$\text{open}\ u\ t = \text{open}_0\ u\ t$$

$$\text{open}_n\, u\ (\texttt{Var}\, v) = \begin{cases} u & \text{if } v = \texttt{bvar}\, n \\ \texttt{Var}\, v & \text{else} \end{cases}$$
$$\text{open}_n\, u\ (\texttt{App}\, t_1\, t_2) = \texttt{App}\ (\text{open}_n\, u\, t_1)\ (\text{open}_n\, u\, t_2)$$
$$\text{open}_n\, u\ (\texttt{Lam} \star t) = \texttt{Lam} \star \left(\text{open}_{n+1}\, u\, t\right)$$

(a) Opening a lambda term by u

$$\text{LC} : \texttt{term}_{\mathbf{1}}(\mathbb{A}+\mathbb{N}) \to \mathbf{2}$$
$$\text{LC}\ t = \text{LC}_0\ t$$

$$\text{LC}_n\ (\texttt{Var}\ v) = \begin{cases} \bot & \text{if } v = \texttt{bvar}\ m \text{ and } n \leq m \\ \top & \text{else} \end{cases}$$
$$\text{LC}_n\ (\texttt{App}\ t_1\ t_2) = \text{LC}_n\ t_1\ \wedge\ \text{LC}_n\ t_2$$
$$\text{LC}_n\ (\texttt{Lam}\ \star\ t) = \text{LC}_{n+1}\ t$$

(b) Testing for local closure

Figure 4: Operations on locally nameless terms

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. Unlike open, LC computes a boolean ($\mathbf{2} = \{\top, \bot\}$) instead of a term. To define and reason about LC, one must integrate both concepts above to define decorated-traversable functors and DTMs. Σ_B^λ is an example of a decorated-traversable functor, and $\texttt{term}_B$ is a DTM. As we have shown with Tealeaves [15], 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 present definitions type-theoretically alongside a diagrammatic calculus. For ease of reading, the different sorts of wires in our graphical calculus, which play different roles, are typeset with high-contrast colors.

3.1 Decorations

The category $\mathbf{Dec}_W$ (Definition 3.4) of decorated functors is parameterized by some monoid W, which we take as given. In Tealeaves, W is typically the free monoid $\texttt{list}\ B$, representing the list of the binders in scope at some occurrence. In brief, decorated functors arise from the elementary fact that any monoid W in **Set** forms a unique *bi*monoid—a coherent combination of a monoid and a comonoid on the same set. The "product-with" embedding,

$$X \mapsto (X \times -) : \mathbf{Set} \to \mathbf{End}_{\text{Set}}$$

is strong monoidal,[3] meaning it preserves monoids, comonoids, and indeed bimonoids, making $(\mathrm{W} \times -)$ a *bimonad*. Decorated functors are precisely the right comodules of this bimonad, which, by adapting

[3]As opposed to merely lax or oplax monoidal, not to be confused with tensorial strength.

a construction from abstract algebra (see Section 4.1 of [7]), form a monoidal category. This means we can consider monoids of decorated functors, or decorated monads. Now we step throw this slowly.

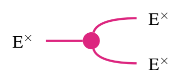
As a first step, consider any set E. It is an exercise in definitions to verify that E is the carrier of exactly one comonoid, the duplication comonoid over E. This structure captures aspects of classical information and its fundamental operations of duplication and deletion.

Definition 3.1. *The* duplication comonoid over E : Set *is given by the following operations.*

$$\begin{aligned} \text{del} &: E \to \mathbf{1} & \text{del}\ e &= \star \\ \Delta &: E \to E \times E & \Delta\ e &= (e,e) \end{aligned}$$

The duplication comonoid induces a comonad on $(E \times -)$ known to functional programmers as the environment or reader comonad. In this paper, these wires, which we think of as carrying "contextual" information, are drawn in red.

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



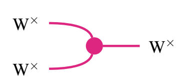
$$\text{extr}^{\text{E}\times} : \forall (A : \mathbf{Set}),\ E \times A \to A \qquad \text{extr}^{\text{E}\times}_A\,(e,a) = a \tag{3.1}$$

$$\text{dup}^{\text{E}\times} : \forall (A : \mathbf{Set}),\ E \times A \to E \times (E \times A) \qquad \text{dup}^{\text{E}\times}_A\,(e,a) = (e,(e,a)) \tag{3.2}$$

The co-Kleisli arrows of the environment comonad have the form $E \times A \to B$. In functional programming, this comonad captures computations $A \to B$ that additionally can read, but not modify, an 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" [37].

Now consider our monoid $W = \langle W, \cdot, 1_W \rangle$. The duplication comonoid exists on the underlying set of W, so in particular $(W \times -)$ is an instance of the reader comonad. Additionally, mirroring the comonoid structure, the monoid on W gives rise to a monad structure on $(W \times -)$ known variously as the writer or logger monad.

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

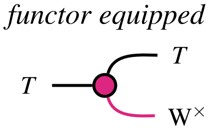


$$\text{ret}^{\text{W}\times} : \forall (A : \mathbf{Set}),\ A \to W \times A \qquad \text{ret}^{\text{W}\times}_A\, a = (1_W, a) \tag{3.3}$$

$$\text{join}^{\text{W}\times} : \forall (A : \mathbf{Set}),\ W \times (W \times A) \to W \times A \qquad \text{join}^{\text{W}\times}_A(w_1,(w_2,a)) = (w_1 \cdot w_2, a) \tag{3.4}$$

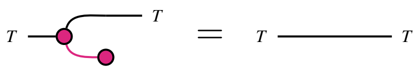
If one thinks about functors as functional data structures, then "decorated" funtors are ones whose elements each occur in a context of type W.

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

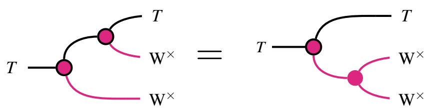


$$\text{dec}^{\text{T}} : \forall (A : \mathbf{Set}), TA \to T\,(W \times A)$$

subject to the following two laws:



$$\mathrm{map}^{\mathrm{T}}\mathrm{extr}^{\mathrm{W}\times} \cdot \mathrm{dec}^{\mathrm{T}} = \mathrm{id} \tag{3.5}$$



$$\mathrm{dec}^{\mathrm{T}} \cdot \mathrm{dec}^{\mathrm{T}} = \mathrm{map}^{\mathrm{T}}\mathrm{dup}^{\mathrm{W}\times} \cdot \mathrm{dec}^{\mathrm{T}} \tag{3.6}$$

Intuitively, (3.5) states that computing the context of every element and immediately deleting it is the same as doing nothing. (3.6) states that computing each context once and making a copy of it is the same as computing each context twice.

Example 3.5. *The functor Σ^{λ}_{B} is decorated by* `list` *B, the free monoid over B. The operation is defined as follows (where by abuse of notation we give constructors of Σ^{λ} the same name as corresponding constructors of* `term`*):*

$$\mathrm{dec}_X : \Sigma^{\lambda}_{B} X \to \Sigma^{\lambda}_{B} (\texttt{list}\, B \times X)$$

$$\begin{aligned} \mathrm{dec}\,(\texttt{App}\, x_1\; x_2) &= \texttt{App}\;([], x_1)\;([], x_2) \\ \mathrm{dec}\,(\texttt{Lam}\, b\; x) &= \texttt{Lam}\, b\;([b], x) \end{aligned}$$

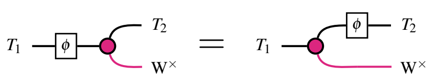
Notation*: $[]$ is the empty list, while $[b]$ is a singleton.*

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.

Technically, we have not yet used the monoid structure assumed of W. A related fact is that we have only defined decorated *functors*, but our term functor T is a monad. How should these structures be related to each other? The answer comes from the recognition that decorated functors form a monoidal category much like $\mathbf{End}_{\mathrm{Set}}$.

Lemma 3.6. *The category $\mathbf{Dec}_W$ of decorated functors is given by the following data:*

- *Objects are endofuntors $T : \mathbf{Set} \to \mathbf{Set}$ paired with a decoration*
- *Morphisms are natural transformations $T_1 \Rightarrow T_2$ that commute with the decorations of T_1 and T_2*



$$\mathrm{dec}^{\mathrm{T}_2} \cdot \phi = \phi \cdot \mathrm{dec}^{\mathrm{T}_1} \tag{3.7}$$

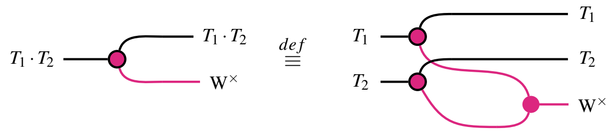
That this constitutes a category is clear. Slightly less obvious is that $\mathbf{Dec}_W$ is a strict monoidal category. Like $\mathbf{End}_{\mathrm{Set}}$, the tensor operation is composition of functors, but we must explain how to decorate the composition. Likewise, the tensor unit is the identity functor, whose decoration must also be defined.

Lemma 3.7. $\mathbf{Dec}_W$ *is a monoidal category by the following data:*

- *The tensor unit is the identity functor paired with the "null" decoration*

$$\mathrm{dec}^{\mathbb{1}} = \mathrm{ret}^{\mathrm{W}\times} \tag{3.8}$$

- *Tensor product is given by composition of functors, with decorations added monoidally*

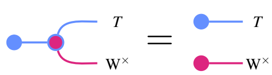


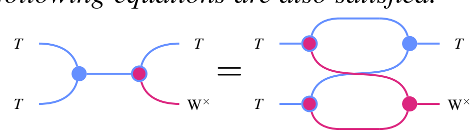
$$\mathrm{dec}^{T_1 \cdot T_2} = \mathrm{map}^{T_1}\left(\mathrm{map}^{T_2}\left(\mathrm{join}^{W^\times}\right) \cdot \mathrm{st}^{T_2}_{W}\right) \cdot \mathrm{dec}^{T_1} \cdot \mathrm{map}^{T_1} \mathrm{dec}^{T_2} \tag{3.9}$$

Above, $\mathrm{st}^{T_2}_{W} : \forall\,(A : \mathbf{Set})\,, W \times T_2\,A \to T_2\,(W \times A)$ is the tensorial strength operation, depicted as crossing a red wire over a functor. That (3.8) and (3.9) satisfy axioms (3.5)—(3.6) is easily verified, as are the laws governing the tensor operation.

Since $\mathbf{Dec}_W$ is a monoidal category, it makes sense to consider monoids in this category. Such a structure must be both an ordinary monad and a decorated functor. The new detail is that the monad operations must also satisfy (3.7), given the operations defined in Lemma 3.7. This yields two additional equations.

Definition 3.8. *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.*





$$\mathrm{dec}^{T} \cdot \mathrm{ret}^{T} = \mathrm{ret}^{T} \cdot \mathrm{ret}^{W^\times} \tag{3.10}$$

$$\mathrm{dec}^{T} \cdot \mathrm{join}^{T} = \mathrm{join}^{T \cdot W^\times} \cdot \mathrm{dec}^{T} \cdot \mathrm{map}^{T}\left(\mathrm{dec}^{T}\right) \tag{3.11}$$

In (3.11), $\mathrm{join}^{T \cdot W^\times}$ is an abbreviation for

$$\mathrm{join}^{T} \cdot \mathrm{map}^{T}\left(\mathrm{map}^{T}\left(\mathrm{join}^{W^\times}\right) \cdot \mathrm{st}^{T}_{W}\right) : \forall (A : \mathbf{Set}), T\,(W \times T\,(W \times A)) \to T\,(W \times A)$$

Indeed, this operation is part of a monad instance on $T \cdot (W \times -)$.

In the context of syntax metatheory, (3.10) states that an atomic term (some `Var` x) has no binders—the context of x is the monoid unit, typically the empty list or the natural number 0. (3.11) 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 one recurses down a syntax tree, as in the recursive operations from Figure 4.

Example 3.9. *The monad* $\mathtt{term}_B$ *is decorated by* $\mathtt{list}\,B$. *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:*

$$\begin{array}{cc} \mathrm{dec} : \mathtt{term}_{\mathbb{A}}\,\mathbb{A} \to \mathtt{term}_{\mathbb{A}}\,(\mathtt{list}\,\mathbb{A} \times \mathbb{A}) & \mathrm{dec} : \mathtt{term}_{\mathbf{1}}\,(\mathbb{A} + \mathbb{N}) \to \mathtt{term}_{\mathbf{1}}\,(\mathbb{N} \times (\mathbb{A} + \mathbb{N})) \\ \lambda x.\lambda y.yx \mapsto \lambda x.\lambda y.([x,y],y)([x,y],x) & \lambda\lambda 01 \mapsto \lambda\lambda(2,0)(2,1) \\ (\lambda x.y\lambda y.z) \mapsto (\lambda x.([x],y)\lambda y.([x,y],z)) & (\lambda 0)\,(\lambda\lambda 1) \mapsto (\lambda\,(1,0))\,(\lambda\lambda(2,1)) \end{array}$$

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

The payoff of this definition will be explained after we consider the separate issue of traversability.

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 [28] defined traversable functors as those equipped with a distributive law over applicative functors (i.e. lax monoidal endofunctors on **Set**). Subsequent work [21, 23] refined the notion by supplying an appropriate set of axioms for this operation.

Definition 3.10. *An* applicative functor *is a set-forming operation* $F : \mathbf{Set} \to \mathbf{Set}$ *with operations*

$$\mathrm{pure}^{\mathrm{F}} : \forall\,(A : \mathbf{Set})\,,\ A \to FA$$
$$(\circledast)^{\mathrm{F}} : \forall\,(A\,B : \mathbf{Set})\,,\ F\,(A \to B) \to FA \to FB$$

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

$$\mathrm{pure}\ \mathrm{id} \circledast a = a \qquad (3.12) \qquad\qquad g \circledast (f \circledast a) = \mathrm{pure}\ \ (\cdot) \circledast g \circledast f \circledast a \qquad (3.14)$$
$$\mathrm{pure}\ f \circledast \mathrm{pure}\ a = \mathrm{pure}\ \ (fa) \qquad (3.13) \qquad\qquad f \circledast \mathrm{pure}\ a = \mathrm{pure}\ \ (f \mapsto fa) \circledast f \qquad (3.15)$$

This class includes the identity functor $\mathbb{1}$ and is closed under composition. An important special case are constant applicatives: these must map all sets to some monoid M, with the operations and axioms coinciding with those of monoids.

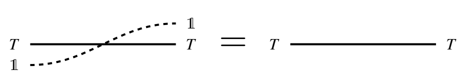
Definition 3.11. *An* applicative morphism $\phi : F \Rightarrow G$ *is a natural transformation between applicative functors that commutes with* pure *and* $(\circledast)$ *in an obvious way.*

Traversable functors are those that distribute over any choice of applicative functor in a well-behaved way.

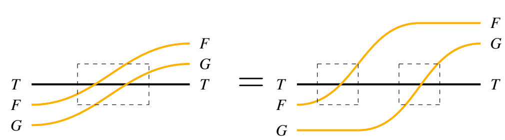
Definition 3.12. *A traversable functor is equipped with an operation*

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

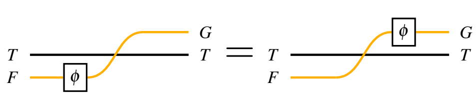
subject to the following axioms (ϕ ranging over applicative morphisms).



$$\mathrm{dist}^{\mathrm{T}}_{\mathbb{1}} = \mathrm{id} \qquad (3.16)$$



$$\mathrm{dist}^{\mathrm{T}}_{\mathrm{F}\cdot\mathrm{G}} = \mathrm{map}^{\mathrm{F}}\left(\mathrm{dist}^{\mathrm{T}}_{\mathrm{G}}\right) \cdot \mathrm{dist}^{\mathrm{T}}_{\mathrm{F}} \qquad (3.17)$$



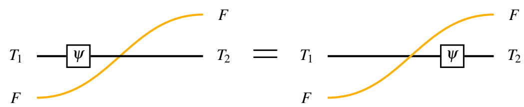
$$\mathrm{dist}^{\mathrm{T}}_{\mathrm{G}} \cdot \mathrm{map}^{\mathrm{T}}\,(\phi_{\mathrm{A}}) = \phi_{\mathrm{A}} \cdot \mathrm{dist}^{\mathrm{T}}_{\mathrm{F}} \qquad (3.18)$$

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 dist reduces to $TM \to M$. 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 [21] pointed out that (3.16) forbids this operation from “skipping” any elements in T, while Jaskelioff and Rypacek [23] pointed out that (3.17) forbids this operation from “double counting” any elements.

Waern [38] defined the monoidal category of traversable functors. An arrow in this category is a natural transformation between traversable functors that commutes with $\text{dist}_{,}$ in an obvious way.

Lemma 3.13 (Category **Trav**). *The category* **Trav** *of traversable functors is given by the following data:*

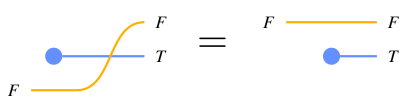
- *Objects are endofunctors T :* **Set** $\to$ **Set** *paired with a distributive law over applicative functors*
- *Morphisms are natural transformations $\psi : T_1 \Rightarrow T_2$ that commute with distribution.*



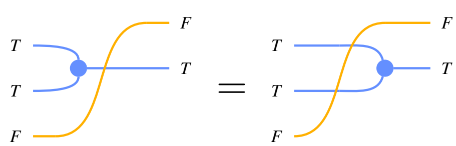
$$\text{dist}_F^{T_2} \cdot \psi = \text{map}^F(\psi) \cdot \text{dist}_F^{T_1} \tag{3.19}$$

The identity functor is trivially traversable, and the composition of traversables is traversable just by composing the distributions. Hence, traversable functors **Trav** forms a monoidal category. As before, we can consider monoids in this category. These are monads that are also traversable and whose monad operations satisfy (3.19).

Definition 3.14. *A* traversable monad *T is a monoid in* **Trav**. *Explicitly, T has the structures of both a traversable functor and a monad and satisfies the following equations:*



$$\text{dist}_F^T \cdot \text{ret}^T = \text{map}^F\left(\text{ret}^T\right) \tag{3.20}$$



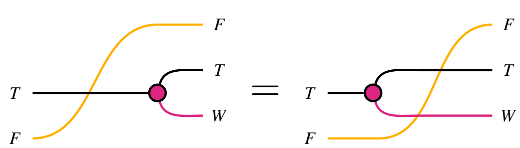
$$\text{dist}_F^T \cdot \text{join}^T = \text{map}^F\left(\text{join}^T\right) \cdot \text{dist}_F^T \cdot \text{map}^T\left(\text{dist}_F^T\right) \tag{3.21}$$

Though the laws appear opaque, for syntax metatheory, (3.20) states that a term formed from ret/`Var` contains only a single variable. (3.21) 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 [15].

3.3 Decorated Traversable Functors

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

Definition 3.15. *A* decorated-traversable *functor is equipped with the structure of both a decorated and traversable functor (Definitions 3.4 and 3.12), subject to the following extra condition:*



$$\text{map}^F\left(\text{dec}^T\right) \cdot \text{dist}_F^T = \text{dist}_F^T \cdot \text{map}^T\left(\text{dist}_F^{W\times}\right) \cdot \text{dec}^T \tag{3.22}$$

Lemma 3.16 (Category $\mathbf{DecTrav}_W$). *The strict monoidal category* $\mathbf{DecTrav}_W$ *of decorated-traversable functors is given by the following data:*

- *Objects are decorated traversable functors*
- *Morphisms are natural transformations satisfying both* (3.7) *and* (3.19).
- *The tensor product is given by composition of decorated-traversable functors, with the identity functor serving as the tensor unit.*

Definition 3.17. *A* decorated traversable monad *(DTM) is a monoid in* $\mathbf{DecTrav}_W$.

The force of Definition 3.17 is that a DTM is simultaneously an instance of Definitions 3.8, 3.14, and 3.16. A self-contained summary of the axioms can be found in the appendix.

4 Kleisli Representation for DTMs

Definition 3.17 is phrased in terms of principled categorical abstractions, but this is not the most convenient presentation when working in a theorem prover. Just proving that a syntax forms a DTM is tedious, requiring five operations and 19 equations. The following Kleisli-style definition, mirroring Definition 2.2, is more economical and more useful to program with.

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

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

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

$$\mathrm{binddt}_{\mathbb{1}} \left(\mathrm{ret}^{\mathrm{T}} \cdot \mathrm{extr}^{\mathrm{W}\times}\right) = \mathrm{id} \tag{4.1}$$

$$\mathrm{binddt}_{\mathrm{F}}\, f \cdot \mathrm{ret}^{\mathrm{T}} = f \cdot \mathrm{ret}^{\mathrm{W}\times} \tag{4.2}$$

$$\mathrm{map}^{\mathrm{F}}\ (\mathrm{binddt}_{\mathrm{G}}\ g) \cdot (\mathrm{binddt}_{\mathrm{F}}\ f) = \mathrm{binddt}_{\mathrm{F}\cdot\mathrm{G}}\ \big(\lambda(w,a).\mathrm{map}^{\mathrm{F}}\ (\mathrm{binddt}_{\mathrm{G}}\ (g \odot w))\, f(w,a)\big) \tag{4.3}$$

$$\phi \cdot \mathrm{binddt}_{\mathrm{F}}\, f = \mathrm{binddt}_{\mathrm{G}}\,(\phi \cdot f) \tag{4.4}$$

In (4.3), $(\odot)$ *is defined* $(g \odot w_1)\,(w_2, b) \overset{def}{=} g\,(w_1 \cdot w_2, b)$.

The following theorem speaks to the robustness of Definition 4.1.

Theorem 4.2. *Definitions 3.17 and 4.1 are equivalent.*

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

$$\mathrm{binddt}_{\mathrm{F}} f \overset{def}{=} \mathrm{map}^{\mathrm{F}} \left(\mathrm{join}^{\mathrm{T}}\right) \cdot \mathrm{dist}^{\mathrm{T}}_{\mathrm{F}} \cdot \mathrm{map}^{\mathrm{T}}\ f \cdot \mathrm{dec}. \tag{4.5}$$

Given ret and binddt, we define the operations of DTMs thus:

$$\begin{aligned} \mathrm{map} f &\overset{def}{=} \mathrm{binddt}_{\mathbb{1}} \left(\mathrm{ret}^{\mathrm{T}} \cdot f \cdot \mathrm{extr}^{\mathrm{W}\times}\right) & \mathrm{dec} &\overset{def}{=} \mathrm{binddt}_{\mathbb{1}} \left(\mathrm{ret}^{\mathrm{T}}\right) \\ \mathrm{join} &\overset{def}{=} \mathrm{binddt}_{\mathbb{1}} \left(\mathrm{extr}^{\mathrm{W}\times}\right) & \mathrm{dist} &\overset{def}{=} \mathrm{binddt}_{\mathrm{F}} \left(\mathrm{ret}^{\mathrm{T}} \cdot \mathrm{extr}^{\mathrm{W}\times}\right) \end{aligned}$$

Besides verifying these definitions satisfy the appropriate equations, 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. The appendix contains a string-diagrammatic derivation of (4.1)—(4.4) (Lemma A.1). ☐

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

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

Like bind, 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 ($\circledast$). Incorporating these aspects greatly expands the range of operations we can define generically.

4.1 Substitution Metatheory

Figure 6 contains generic versions of the opening operation and local closure, relating to Figure 4 as Figure 3b does to 3a. The definition of LC in particular requires full use of the expressiveness of binddt$_,$. Here, **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. As instances of binddt, we can reason about these operations axiomatically.

$$\begin{aligned}
&\text{open}_{\text{loc}} : T\,(\mathbb{A}+\mathbb{N}) \to \mathbb{N} \times (\mathbb{A}+\mathbb{N}) \to \mathrm{T}\,(\mathbb{A}+\mathbb{N}) \\
&\text{open}_{\text{loc}}\, u\ (n, \texttt{fvar}\, a) = \text{ret}\,(\texttt{fvar}\, a) \\
&\text{open}_{\text{loc}}\, u\ (n, \texttt{bvar}\, m) = \begin{cases} u & \text{if } n = m \\ \text{ret}^{\mathrm{T}}\,(\texttt{bvar}\,\mathrm{m}) & \text{else} \end{cases} \\
&\text{open}\, u = \text{binddt}^{\mathrm{T}}_{\mathbb{1}}\ (\text{open}_{\text{loc}}\, u)
\end{aligned}$$

$$\begin{aligned}
&\text{LC}_{\text{loc}} : \mathbb{N} \times (\mathbb{A}+\mathbb{N}) \to \mathbf{2} \\
&\text{LC}_{\text{loc}}\,(n, \texttt{fvar}\, a) = \top \\
&\text{LC}_{\text{loc}}\,(n, \texttt{bvar}\, m) = \begin{cases} \bot & \text{if } n \le m \\ \top & \text{else} \end{cases} \\
&\text{LC} = \text{binddt}^{\mathrm{T}}_{\mathbf{2}}\ \text{LC}_{\text{loc}}
\end{aligned}$$

Figure 6: Generic locally nameless operations for a DTM T

The adequacy of Definition 4.1 for the needs of working metatheorists is an empirical question 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 proven once and for all, while LNgen generates proofs specific to a given signature. Because it relies on heuristics and Ltac [14] (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 we hope to automate in future work.

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.

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 `Ewp` ("extend with policy") which is similar in spirit to, but strictly less expressive than, $\text{binddt}_{,}$. 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, 27, 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 intrinisically 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 [16, 17] have developed a presheaf-theoretic account of syntax. Subsequent work by Power and Tanaka axiomatized and expanded the presheaf-theoretic approach [31, 32]. 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 [18] 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 [30, 12].

Besides LNgen, utilities similar in spirit to Tealeaves include GMeta [25] and Autosubst [34, 35]. 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, 33]. Our $\text{binddt}_{,}$ 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

$$\text{bmap} : \forall (V\ \mathrm{B}_1\ \mathrm{B}_2 : \mathbf{Set}), (B_1 \to B_2) \to \mathtt{term}_{B_1} V \to \mathtt{term}_{B_2} V$$

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, feature 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 might require a distributive law only over commutative applicative functors.

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/S0956796800000186.

[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.

[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.

[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, doi:10.1007/3-540-48168-0_32.

[5] Brian Aydemir & Stephanie Weirich (2010): *LNgen: Tool Support for Locally Nameless Representations*. Technical Report, University of Pennsylvania, Department of Computer and Information Science. Available at https://repository.upenn.edu/handle/20.500.14332/7902.

[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.

[7] Tørris Koløen Bakke (2007): *Hopf Algebras and Monoidal Categories*. Master's thesis, University of Tromsø. Available at https://munin.uit.no/bitstream/handle/10037/1084/finalthesis.pdf.

[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.

[9] Richard Bird & Lambert Meertens (1998): *Nested datatypes*. In: *In MPC'98, volume 1422 of LNCS*, Springer-Verlag, pp. 52–67, doi:10.1007/BFb0054285.

[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] R. M. Burstall (1969): *Proving Properties of Programs by Structural Induction*. *The Computer Journal* 12(1), pp. 41–48, doi:10.1093/comjnl/12.1.41.

[12] 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.

[13] 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:10.1016/1385-7258(72)90034-0.

[14] 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, doi:10.1007/3-540-44404-1_7.

[15] Lawrence Dunn, Val Tannen & Steve Zdancewic (2023): *Tealeaves: Structured Monads for Generic First-Order Abstract Syntax Infrastructure*. In Adam Naumowicz & René Thiemann, editors: *14th International Conference on Interactive Theorem Proving (ITP 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 268, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 14:1–14:20, doi:10.4230/LIPIcs.ITP.2023.14.

[16] 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.

[17] 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.

[18] Marcelo Fiore & Dmitrij Szamozvancev (2022): *Formal Metatheory of Second-Order Abstract Syntax*. *Proc. ACM Program. Lang.* 6(POPL), doi:10.1145/3498715.

[19] Brendan Fong & David I Spivak (2018): *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. doi:10.48550/arXiv.1803.05316.

[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.

[21] Jeremy Gibbons & Bruno Oliveira (2009): *The essence of the Iterator pattern*. *J. Funct. Program.* 19, pp. 377–402, doi:10.1017/S0956796809007291.

[22] Ralf Hinze & Dan Marsden (2023): *Introducing String Diagrams: The Art of Category Theory*. Cambridge University Press, doi:10.1017/9781009317825.

[23] Mauro Jaskelioff & Ondrej Rypacek (2012): *An Investigation of the Laws of Traversals*. *Electronic Proceedings in Theoretical Computer Science* 76, doi:10.4204/EPTCS.76.5.

[24] 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.

[25] 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, doi:10.1007/978-3-642-28869-2_22.

[26] Ernest G. Manes (1976): *Algebraic Theories*. Springer New York, doi:10.1007/978-1-4612-9860-1.

[27] Conor McBride (2005): *Type-Preserving Renaming and Substitution*. Unpublished note.

[28] Conor McBride & Ross Paterson (2008): *Applicative Programming with Effects*. *J. Funct. Program.* 18(1), p. 1–13, doi:10.1017/S0956796807006326.

[29] Eugenio Moggi (1988): *Computational Lambda-Calculus and Monads*. IEEE Computer Society Press, pp. 14–23, doi:10.1109/LICS.1989.39155.

[30] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. 23, pp. 199–208, doi:10.1145/960116.54010.

[31] 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.

[32] John Power & Miki Tanaka (2008): *Category Theoretic Semantics for Typed Binding Signatures with Recursion*. *Fundam. Informaticae* 84, pp. 221–240. Available at http://content.iospress.com/articles/fundamenta-informaticae/fi84-2-05.

[33] 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.

[34] 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, doi:10.1007/978-3-319-22102-1_24.

[35] 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.

[36] The Coq Development Team (2023): *The Coq Proof Assistant*, doi:10.5281/zenodo.8161141.

[37] Tarmo Uustalu & Varmo Vene (2008): *Comonadic Notions of Computation*. *Electronic Notes in Theoretical Computer Science* 203(5), pp. 263–284, doi:10.1016/j.entcs.2008.05.029. Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008).

[38] Love Waern (2019): *Cofree Traversable Functors*. Bachelor's thesis. Available at https://uu.diva-portal.org/smash/record.jsf?pid=diva2:1369180.

[39] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin & Steve Zdancewic (2012): *Formalizing the LLVM Intermediate Representation for Verified Program Transformations*. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, Association for Computing Machinery, New York, NY, USA, p. 427–440, doi:10.1145/2103656.2103709.

A Appendix

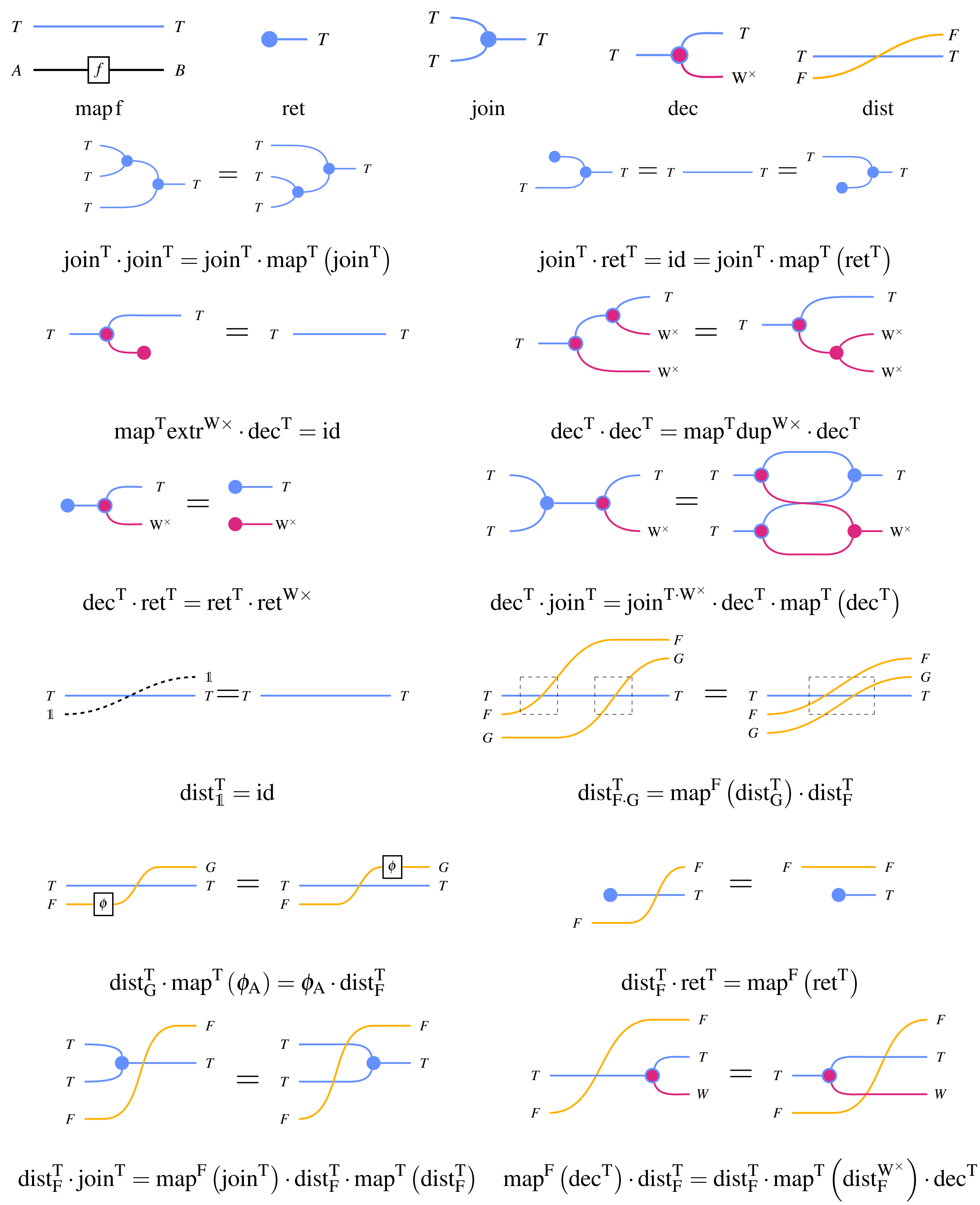
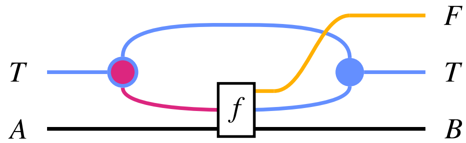


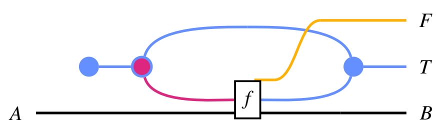
Figure 7: String diagrammatic presentation of DTMs

Lemma A.1. *Every DTM gives rise to a Kleisli-presented DTM according to the following definition of* binddt.

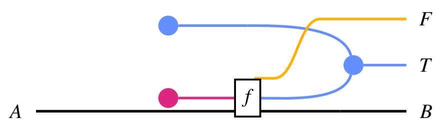


$$\mathrm{binddt}_{\mathrm{F}}\, f = \mathrm{map}^{\mathrm{F}}\left(\mathrm{join}^{\mathrm{T}}\right) \cdot \mathrm{dist}^{\mathrm{T}}_{\mathrm{F}} \cdot \mathrm{map}^{\mathrm{T}}\ f \cdot \mathrm{dec}^{\mathrm{T}}$$

Proof. Proof of Equation (4.1):

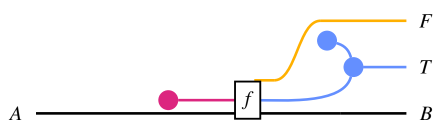


=



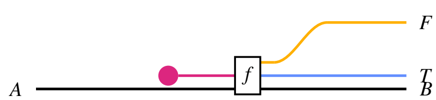
Apply the decoration cup law (3.10).

=



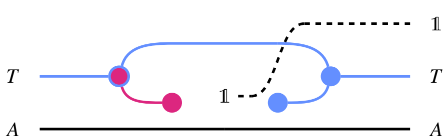
Pull the unit across F (3.20).

=

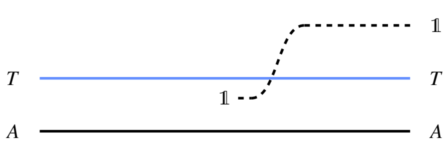


Apply the left monad unit law (2.1).

Proof of Equation (4.2):

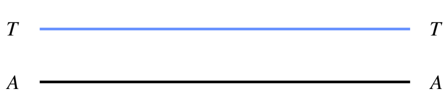


=



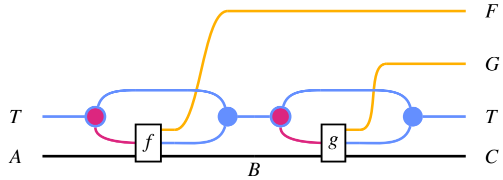
Apply unit and counit laws (3.5) (2.2).

=

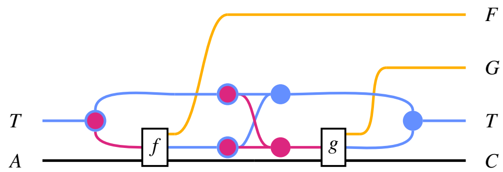


Apply traversal unitary law (3.16).

Proof of Equation (4.3):

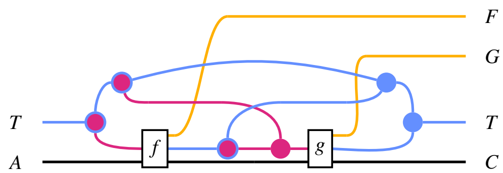


=



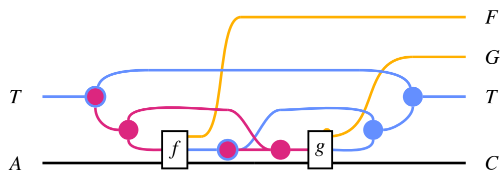
Apply the butterfly law (3.11).

=



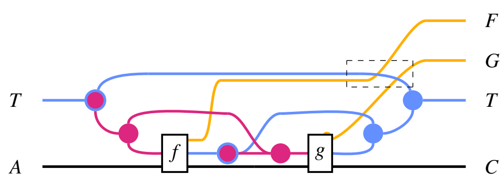
Drag operations past distributions (3.21) (3.22).

=



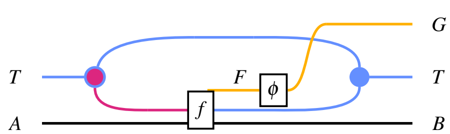
Apply (co)associativity (2.3) (3.6).

=

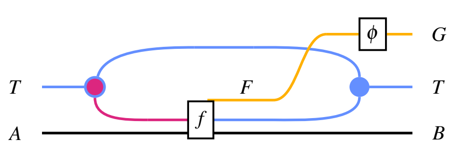


Apply traversal composition law (3.17).

Proof of Equation (4.4):



=



Slide the applicative morphism (3.18).

□