Tealeaves: Structured Monads for Generic First-Order Abstract Syntax Infrastructure

Lawrence Dunn
University of Pennsylvania, Philadelphia, PA, USA

Val Tannen
University of Pennsylvania, Philadelphia, PA, USA

Steve Zdancewic
University of Pennsylvania, Philadelphia, PA, USA

Abstract

Verifying the metatheory of a formal system in Coq involves a lot of tedious "infrastructural" reasoning about variable binders. We present Tealeaves, a generic framework for first-order representations of variable binding that can be used to develop this sort of infrastructure once and for all. Given a particular strategy for representing binders concretely, such as locally nameless or de Bruijn indices, Tealeaves allows developers to implement modules of generic infrastructure called *backends* that end users can simply instantiate to their own syntax. Our framework rests on a novel abstraction of first-order abstract syntax called a *decorated traversable monad* (DTM) whose equational theory provides reasoning principles that replace tedious induction on terms. To evaluate Tealeaves, we have implemented a multisorted locally nameless backend providing generic versions of the lemmas generated by LNgen. We discuss case studies where we instantiate this generic infrastructure to simply-typed and polymorphic lambda calculi, comparing our approach to other utilities.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases Coq, category theory, formal metatheory, raw syntax, locally nameless

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.14

Supplementary Material *Software (Source Code)*: `https://github.com/dunnl/tealeaves` archived at `swh:1:dir:213050814e9a7233b944f92c191df8eb360bb84b`

Acknowledgements We wish to thank the anonymous reviewers for their helpful comments and suggestions, and the authors of LNgen for describing their work and experience using LNgen.

1 Introduction

Computer-verified metatheory is increasingly critical for establishing trust in the design and implementation of formal systems [6], which we take to include formal logics, programming languages, query languages, lambda calculi, specification languages, and basically any system with a precise syntax. Formalizing metatheory in a general-purpose proof assistant like Coq requires a lot of tedious reasoning about variable binding. When performed manually, this typically involves the user proving a suite of "infrastructure" lemmas concerned with the properties of capture-avoiding substitution. In practice, if not in principle, this infrastructure is tightly coupled to the exact signature used to generate the syntax, owing to the prolific use of structural recursion and induction on terms. This dependency makes metatheory brittle, prevents reusability, hampers collaboration by users working on different systems, and can make syntax infrastructure more challenging to automate. This paper presents Tealeaves, a Coq framework for building extensible libraries of generic syntax infrastructure that users can *instantiate* to their own syntax, thus facilitating collaboration and reuse. Our framework rests on top of a principled category-theoretic abstraction of raw first-order abstract syntax, that of a *decorated traversable monad*, which we introduce in Section 3.

A wide variety of syntax formalization strategies have been proposed in the literature, and a commensurate number of utilities have been designed to automate syntax infrastructure. Why, then, should we introduce yet another syntax metatheory framework? Generally, frameworks will differ in what sorts of syntax can be handled, what support is required from the proof assistant, and especially the cost of entry to the user. The main novelty of Tealeaves lies in the intersection of three features:

1. **Raw syntax** Tealeaves considers "raw" syntax that is extrinsically typed and scoped, in contrast to work on intrinsically well-scoped, well-typed syntax.
2. **Modular representations** Tealeaves' extensible design is agnostic about how binding is represented, admitting multiple *backend* modules that formalize syntax infrastructure for a particular first-order strategy, such as de Bruijn indices or locally nameless.
3. **Signature-generic** Tealeaves is based on the theory of decorated traversable monads, a set of equations independent of the signature of a particular language. Substitution lemmas proved in the form of reusable Tealeaves backends are proved once and for all, and do not rely on external code generators, which can be slow and fallible.

We briefly summarize some of the more salient points of these features. Sections 4 and 5 offer more detailed comparison to related work.

Extrinsically typed first-order abstract syntax is defined inductively. Notions such as well-scopedness and well-typedness, as well as lemmas about substitution and other operations, are defined *post factum* by structural recursion [10] on terms. This contrasts with intrinsically well-scoped, well-typed syntax [4], which uses the type system of the metatheory (in our case, Coq) to enforce constraints on the embedded syntax, blurring the line between operations' types and their correctness properties. The practical difficulties encountered by the two approaches differ, especially because the intrinsically typed workflow makes heavier use of dependently-typed programming that can be inconvenient to formalize in Coq. The raw approach contrasts with recent work [17] which is formalized in Agda and considers a different category-theoretic abstraction of intrinsically-typed syntax. Both of these styles also contrast with higher-order abstract syntax [26] and representations based on nominal sets [28], which are well-known to require adaptation for use in a general-purpose proof assistant like Coq (see Section 5).

Within the family of first-order approaches, a variety of binding representation strategies are available, with typical examples being de Bruijn indices [13] and locally nameless [11]. At its core, Tealeaves aims to be agnostic about how binding is represented concretely, accommodating multiple representations. Existing utilities for formalizing syntax in Coq, such as Autosubst [32, 36] and LNgen [5], target a specific representation (de Bruijn indices and locally nameless, respectively), and they prove signature-specific lemmas using synthesis or external code generators whereas our lemmas are signature-generic and formalized statically in Coq. Unlike those tools, our design supports variadic binders (i.e. those introducing a variable number of new entities) without modifying the core abstraction.

To achieve the modularity of Tealeaves, it is necessary to have an abstraction of (i.e. interface to) abstract syntax. For us this comes in the form of *decorated traversable monads* (DTMs). Definition 3.1 presents DTMs in terms of a highly expressive combinator `binddt` that we use to define a wide range of syntax-related operations. A previous proof-of-concept Coq framework, GMeta [23], also offers multiple representation strategies formalized generically over syntax, but it lacks a principled abstraction of syntax like DTMs, resorting to proofs by induction on a universe of representable types. One benefit of using DTMs is that the DTM composition law (Equation (3)) yields a fusion law for the composition of any two operations defined with Tealeaves.

To evaluate our framework, we have implemented a locally nameless backend module providing essentially the same infrastructure that a user would generate with LNgen, but whose lemmas are statically verified, generic, and proved using a principled equational theory rather than unverifiable, dynamically-generated proof scripts. We have used this backend to prove type soundness for the simply-typed and polymorphic lambda calculi. The latter especially demonstrates that our framework neatly handles heterogeneous substitution (e.g. substitution of types in terms). We discuss evaluation of Tealeaves in Section 4.

In sum, the contributions of this paper are threefold. (1) We introduce a principled abstraction of raw, first-order abstract syntax, decorated traversable monads, which provides an expressive equational framework for generic reasoning about substitution and related operations. (2) We implement the Coq library Tealeaves, an extensible and modular framework for generic reasoning about syntax, including syntax with many different kinds of variables. (3) We implement a locally nameless Tealeaves backend and use it to formalize progress and preservation lemmas for the two lambda calculi above, evaluating the practicality of our approach.

The rest of this document is laid out as follows. Section 2 explains how a Coq user incorporates Tealeaves into their formal metatheory workflow. Section 3 introduces DTMs and describes how they facilitate generic reasoning. Section 4 evaluates Tealeaves, including a description of our case studies and a feature-wise comparison to the utilities LNgen and Autosubst 1 and 2. Section 5 discusses other related work. Section 6 concludes with our future plans for Tealeaves, especially investigating extensions to the DTM concept.

2 Using Tealeaves

In this section, we examine how Tealeaves fits into the workflow of a formal semanticist working in Coq. As a running example, we consider a formalization of the untyped lambda calculus where variables are represented in the locally nameless style, though Tealeaves accommodates more sophisticated kinds of syntax (see Section 3.4) and can be extended to other representations of variables. It is up to the user what sorts of metatheory they want to develop about the calculus – Tealeaves only provides the syntax infrastructure. The details are unchanged if the user is interested in a typed system because substitution is defined on raw (untyped) terms.

The first-order[1] (or *initial algebraic*) representation of abstract syntax defines terms inductively in the form of *term algebras*. In the simplest case of a single sort of variables, one starts from a base set V of variables and constructs a set $T(V)$ of terms by closing the set under well-sorted applications of constructors. This construction justifies definitions by structural recursion on terms. Figure 1 shows a first-order definition of the syntax of the untyped lambda calculus, called `Lam`, as it would be defined by a user of Tealeaves. To keep the example simple, since we will consider locally nameless variables, the `Abs` case only needs to take the abstraction body as an argument and not a variable name.

Locally nameless is a hybrid strategy mixing Bruijn indices with named variables. A bound variable $n \in \mathbb{N}$ is a natural number that always refers to the n^{th} most recently introduced abstraction, indexing innermost to outermost from 0. A free variable is represented as an `atom`, an abstract type about which we assume only a decidable equality. Figure 2 shows a type `LN` of locally nameless variables as the sum of `nat` and `atom`; this definition is provided by our locally nameless backend. The type of raw lambda terms with locally nameless variables is then `Lam LN`.

[1] "First-order" here refers to the fact that the term constructors do not take Coq-level functions as

```
Inductive Lam (V : Type) : Type :=
  | Var : V -> Lam V
  | Ap  : Lam V -> Lam V -> Lam V
  | Abs : Lam V -> Lam V.
```

Figure 1 Syntax of the untyped lambda calculus.

```
Inductive LN : Type :=
  | Fr : atom -> LN (* free variables *)
  | Bd : nat -> LN. (* bound variables *)
```

Figure 2 The type of locally nameless variables.

As argued by Pollack [29, 30], the main advantage of locally nameless is that there is no possibility of variable capture during substitution and that α-equivalence of expressions coincides with syntactical equality, making this representation more practical in Coq formalizations than a fully named approach as with pen-and-paper. This convenience comes at a mild cost: some terms in `Lam LN` do not correspond to ordinary lambda terms modulo α-equivalence, owing to the possibility of a de Bruijn index n appearing under fewer than $n+1$ abstractions. Such an occurrence is neither free (because it is not an atom) nor bound, so locally nameless substitution lemmas tend to mention a *local closure* predicate ruling out these ill-formed occurrences. Only locally closed terms represent (α-equivalence classes of) ordinary lambda terms.

Without using Tealeaves, most users would not benefit from separating `Lam` and `LN` as shown; they would likely inline `LN` into the definition of `Lam`. We take this approach in Tealeaves mainly so we can exploit the fact that `Lam` is a decorated traversable monad later. Incidentally, this modularity could prove useful to the user who desires to consider more than one representation of variables in the same development, say because one is amenable to formalization in Coq and another is more convenient to program with. As future work, we hope to use Tealeaves to formalize a translation between named and locally nameless variables (see Section 6).

Workflow without Tealeaves

The inductive nature of `Lam` admits a notion of structural recursion on terms, which is used to define operations like capture-avoiding substitution. Our formalization of locally nameless employs five operations: opening one term by another, closing a term by an atom, substitution of a term for a free variable, a free-variable enumeration operation `FV`, and the local closure predicate.[2] The types of these operations are shown in Figure 3. Figure 4 shows an example of how one conventionally defines `FV`.

Users working without tool support must prove a suite of lemmas about these operations, some prototypical examples of which are included in Figure 3. For instance, `subst_fresh` posits that replacing occurrences of an atom `x` with expression `u` in a term `t` leaves `t` unchanged if `x` does not occur in `t`. Such lemmas are needed while developing metatheory about the lambda calculus. They are almost invariably proved by induction on terms.

---

arguments, even if the formal system is higher-order in some other sense.

[2] Local closure can also be given its own `Inductive` definition. In our unified treatment, we prefer to think of the predicate as another operation on terms which happens to return a proposition.

$$\begin{array}{lll} \texttt{open} & : & \texttt{Lam LN} \to \texttt{Lam LN} \to \texttt{Lam LN} \\ \texttt{subst} & : & \texttt{atom} \to \texttt{Lam LN} \to \texttt{Lam LN} \to \texttt{Lam LN} \\ \texttt{close} & : & \texttt{atom} \to \texttt{Lam LN} \to \texttt{Lam LN} \end{array} \qquad \begin{array}{lll} \texttt{FV} & : & \texttt{Lam LN} \to \texttt{list atom} \\ \texttt{LC} & : & \texttt{Lam LN} \to \texttt{Prop} \end{array}$$

$$\texttt{subst\_fresh} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ \texttt{x} \notin \texttt{FV t} \implies \texttt{subst x u t} = \texttt{t}$$

$$\texttt{subst\_spec} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ \texttt{subst x u t} = \texttt{open u (close x t)}$$

$$\texttt{fv\_subst\_upper} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ \texttt{FV (subst x u t)} \subseteq (\texttt{FV t} \setminus \{\texttt{x}\}) \cup \texttt{FV u}$$

$$\begin{aligned} \texttt{open\_inj} : \forall (\texttt{x} : \texttt{atom})(\texttt{u}\,\texttt{t} : \texttt{Lam LN}),\ & \texttt{x} \notin (\texttt{FV t} \cup \texttt{FV u}) \implies \\ & \texttt{open (Var (Fr x)) t} = \texttt{open (Var (Fr x)) u} \implies \texttt{t} = \texttt{u} \end{aligned}$$

Figure 3 Locally nameless operations and some typical infrastructure lemmas.

```
Fixpoint FV (t : Lam LN) : list atom := match t with
  | Var (Fr x) => [x]
  | Var (Bd _) => []
  | Ap t1 t2 => FV t1 ++ FV t2
  | Abs body => FV body
  end.
```

Figure 4 Example definition in Coq of a recursively-defined function `FV`.

This workflow is inherently linearly-ordered as shown in Figure 5: syntax is defined, syntax infrastructure is implemented, then system-specific metatheory is formalized. "System-specific metatheory" can include properties like confluence of the lambda calculus, which is undoubtedly more interesting to the metatheorist than proving dozens of substitution lemmas.

One way a user could save labor is to use a tool like LNgen [5] to generate the infrastructure. LNgen accepts a grammar from the user in the form expected by Ott [33] and generates Coq modules containing lemmas and operations like those in Figure 3. The high-level workflow is unchanged, however: the infrastructure comes after the syntax is defined.

Whether it be implemented by hand or generated automatically, the syntax infrastructure for `Lam` represents a bottleneck in the user's workflow. It is a prerequisite for developing interesting metatheory, but it depends on the definition of `Lam`, so it cannot be formalized as a general-purpose library. The reason for the dependency is that functions defined by recursion (as well as proofs by induction) essentially follow the shape of `Lam`. For instance, adding a new constructor to `Lam` will break `FV` and `subst_fresh` until the user updates them to account for the new constructor or re-executes LNgen.

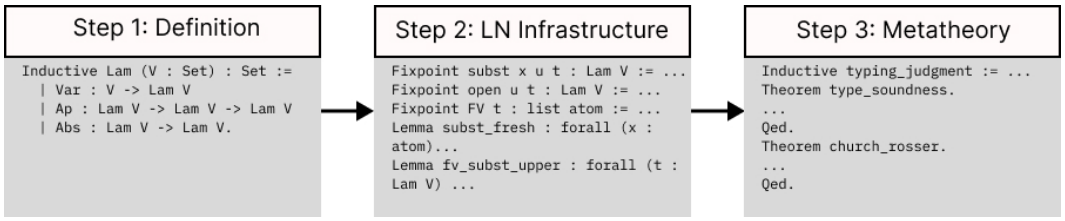

Figure 5 Basic workflow without Tealeaves.

Besides making the infrastructure brittle, this phenomenon implies a user who is formalizing a different syntax cannot reuse the infrastructure for `Lam`. This situation is all the more unfortunate when one realizes that most of the interesting reasoning of locally nameless does not really depend on `Lam` at all. The only interesting case in the proof of `subst_fresh`, for example, is `Var` – the `Ap` and `Abs` cases hold just by induction. Can we do better than this linear workflow?

2.1 The Tealeaves workflow

In a workflow incorporating Tealeaves, the user does not develop the locally nameless syntax infrastructure – we the Tealeaves developers have already implemented it in the form of a reusable Tealeaves *backend* module. Operations and lemmas like those in Figure 3 are provided by this backend, with a caveat: the formalization is generic in the sense that all references to `Lam` are replaced with references to a parameter $\texttt{T} : \texttt{Type} \to \texttt{Type}$. The user's obligation is to *instantiate* the backend to the choice $\texttt{T} = \texttt{Lam}$, which achieves essentially the same effect for the user as if they had constructed the infrastructure themselves. The user benefits as long as it is easier to perform this instantiation than to implement the infrastructure from scratch.

The cost of instantiating the backend is modest: the user must prove that `Lam` forms a *decorated traversable monad* (DTM), a principled category-theoretic concept which Tealeaves defines in the form of a typeclass [34]. All constructs implemented by the backend module are polymorphic over an instance of this typeclass; therefore they can automatically be specialized to any choice of `T`, such as `Lam`, for which a corresponding DTM instance has been registered with Coq's typeclass instance database.

The DTM instantiation process we describe below is for the simplest case when there is one grammatical category (`Lam`) and one sort of variable (arguments to `Var`). Section 3.4 indicates how we generalize this to more complex situations, such as a set of mutually-inductive grammatical categories involving multiple sorts of variables.

Supplying the DTM typeclass instance for `Lam` requires the user to define two operations. The first, which following standard Haskell terminology we call `return` (abbreviated `ret`), represents a coercion from variables to (atomic) terms of the user's syntax. For `Lam`, `ret` is exactly the `Var` constructor.

The more interesting operation is a higher-order function we call `binddt` (`bind` for a `d`ecorated `t`raversable monad). Conceptually, `binddt` acts like a template for defining (some) structurally recursive functions on `Lam`, including context-sensitive substitutions like `open` and "aggregation" operations like `FV`. The type of `binddt`, written in pseudo-Coq notation and specialized to `Lam`, is as follows:

$$\texttt{binddt}\ \texttt{`(Applicative F)}\ \ \texttt{(A B : Type)}\ \ :\ \ \texttt{(nat} \times \texttt{A} \to \texttt{F (Lam B))} \to \texttt{Lam A} \to \texttt{F (Lam B)}$$

We show the definition of `binddt` for `Lam` in Section 3.

Programmers with experience using monads may recognize the previous type as that of the usual `bind` operation extended with two features. First, just like the `traverse` operation of McBride and Paterson [25], the first argument to `binddt` is a choice of applicative functor $\texttt{F} : \texttt{Type} \to \texttt{Type}$. Second, observe that `nat` appears as an input of the function supplied to `binddt` – strictly speaking, one says that `Lam` is a traversable monad decorated *by* the natural numbers under addition. We discuss both of these features in Section 3. To recover the usual `bind` operation, one can instantiate `binddt` at the identity (applicative) functor and apply the projection $\texttt{nat} \times \texttt{A} \to \texttt{A}$.

Once the user defines `binddt`, they must supply a proof that it satisfies the axioms of decorated traversable monads, a set of four equations. These too shall be shown in Section 3.

Altogether, instantiating Tealeaves to `Lam` looks as follows. Note that `ret` and `binddt` are registered as instances of two operational typeclasses [35] called `Return` and `Binddt`. This is just a convenience allowing us to use the notation `ret` and `binddt` throughout Tealeaves and let Coq deduce which DTM instance is being referred to.

```
From Tealeaves Require Import Classes.Kleisli.DTM.
Fixpoint binddt_Lam `{Applicative F} (A B : Type)
    (f : nat * A -> F (Lam B)) (t : Lam A) : F (Lam B) := ...
Instance: Return Lam := Var.
Instance: Binddt nat Lam := binddt_Lam.
Instance: DecoratedTraversableMonad nat Lam.
(* Proofs of the equational axioms of DTMs... *)
Qed.
```

Having bundled all this up into a DTM typeclass instance, the user imports our locally nameless backend, `Tealeaves.Backends.LN`. This module defines all of the operations of locally nameless polymorphically over a choice `T` of DTM (specifically, `T` must be decorated by `nat`). It also supplies polymorphic lemmas. Using Coq's typeclass mechanism, the user can specialize these constructs to their own syntax. We show examples of this specialization below by explicitly passing `(T := Lam)` to each function, but in practice Coq can usually infer the choice of `T` implicitly. These commands will fail with an error message if Coq cannot locate an instance of the `DTM` typeclass for `Lam`.

```
From Tealeaves Require Import Backends.LN.
Check LN.open (T := Lam) : Lam LN -> Lam LN -> Lam LN.
Check LN.subst (T := Lam) : atom -> Lam LN -> Lam LN -> Lam LN.
Check LN.locally_closed (T := Lam) : Lam LN -> Prop.
Check LN.subst_fresh (T := Lam) : forall (t u : Lam LN) (x : atom),
    not (List.In x (FV t)) -> subst x u t = t.
Check LN.subst_spec (T := Lam) : forall (x : atom) (t u : Lam LN),
      subst x u t = open u (close x t).
```

Now that syntax infrastructure for `Lam` is available, the user can proceed with their ordinary workflow, which might consist of defining a type system and developing more interesting metatheory that is specific to the lambda calculus. When the properties of operations like substitution and opening are needed during a proof, the user invokes the corresponding lemma from the backend. The is akin to how one uses the modules that would be generated by LNgen, except no code generation has taken place.

Figure 6 shows a simplified architectural diagram of Tealeaves, which is broadly divided into three parts: the core formalization of DTMs and their properties, the locally nameless backend, and the effort required of the end user. We see again that the user's work logically divides into 3 steps as in Figure 5, but now the second step consists of proving the DTM instance for the user's syntax, while the locally nameless infrastructure is supplied by Tealeaves. We shall explain the rest of the architecture in Section 4.

An inherent limitation of implementing syntax infrastructure as a Coq library is that the backend can only prove a finite number of lemmas about substitution. For locally nameless, it is not entirely clear what constitutes a "complete" set of properties.[3] Suppose a user comes

[3] See Section 4 of [5] for a discussion of this issue in the context of LNgen, along with an informal argument for completeness of their generated lemmas, which our backend also proves.

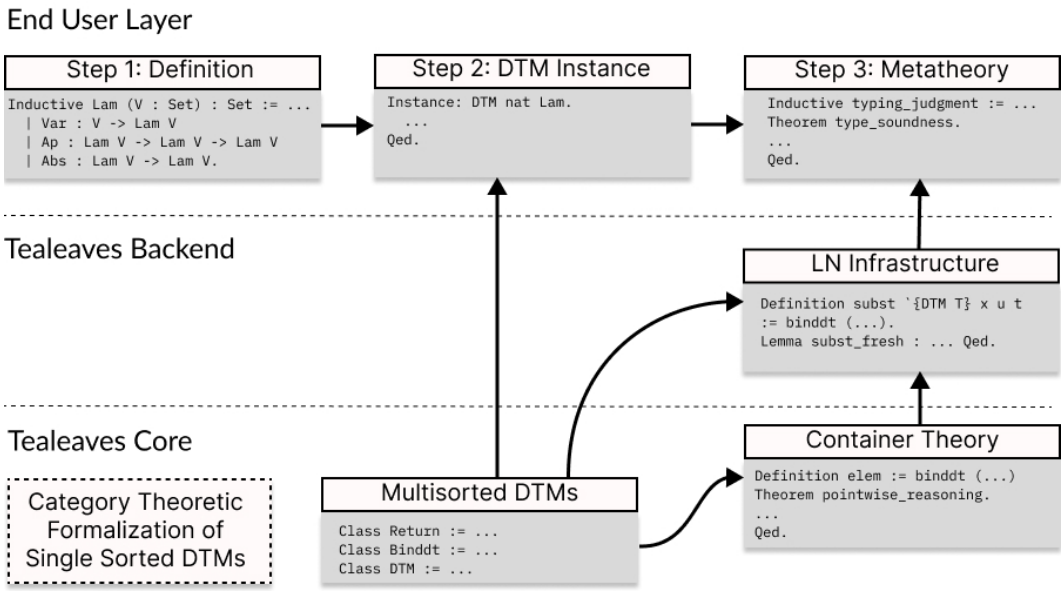

Figure 6 Simplified Tealeaves architecture and user workflow.

across a property of substitution that the backend does not prove. In this case, the user could state their lemma and prove it by induction on `Lam` like usual – Tealeaves does not impede the user's ordinary workflow.[4]

Alternatively, a Tealeaves power user could extend the locally nameless backend with their lemma. Of course, the backend module does not know about `Lam`, so a generic version of their lemma could not be proved by induction on lambda terms. Instead, the proof would have to be developed so as to apply to *any* DTM. Examples of such generic proofs are shown in Section 3.3. The advantage of extending the backend is that the user's effort would be reusable, even to users formalizing other kinds of syntax. This is how Tealeaves facilitates collaboration by users implementing different formal systems. Note that extending the backend with new lemmas requires no programming with Coq's tactic language Ltac [14] or any language outside of Coq itself, unlike the other utilities discussed in Section 4.

3 Decorated Traversable Monads

We now give a high-level intuition for DTMs and their equational theory, starting with the definition of `binddt` for `Lam`. While DTMs can be understood in terms of principled abstractions from monoidal category theory, users of Tealeaves are mostly shielded from this. In this section, we only assume some familiarity with functors and the use of monads to structural functional programs [38]. Like Haskell, we indicate the action of a functor `F` on morphisms as `fmap (A B : Type) : (A → B) → F A → F B`. When it improves clarity, we use subscripts to indicate the implicit values of parametric arguments, and superscripts to disambiguate methods of typeclass instances, e.g. `fmap f` vs. $\texttt{fmap}^{\texttt{F}}_{\texttt{A,B}}\ \texttt{f}$.

[4] The only possible issue is that unfolding the operations exported by the backend will reveal generic constructs that may be challenging to understand, a problem inherent to any generic framework. Future work on Tealeaves could supply custom simplification tactics to hide some of this complexity.

3.1 Proving the DTM instance for `Lam`

The first step required to instantiate Tealeaves to `Lam` is to define `binddt`, which can be thought of as a template for structurally recursive functions. We have seen that the first argument to `binddt` is a choice of applicative functor, an abstraction introduced by McBride and Paterson [25] and used often by functional programmers. For present purposes it suffices to know that an applicative functor $\mathtt{F} : \mathtt{Type} \to \mathtt{Type}$ is characterized by two operations, `pure` and `ap`, whose types are as follows:

$$\begin{aligned} \mathtt{pure} \quad & (\mathtt{A} : \mathtt{Type}) \quad : \quad \mathtt{A} \to \mathtt{F}\ \mathtt{A} \\ \mathtt{ap} \quad & (\mathtt{A}\ \mathtt{B} : \mathtt{Type}) \quad : \quad \mathtt{F}\ (\mathtt{A} \to \mathtt{B}) \to \mathtt{F}\ \mathtt{A} \to \mathtt{F}\ \mathtt{B} \end{aligned}$$

Like monads, applicative functors provide a notion of computational effect, but they are a more general abstraction. Intuitively, `pure` lifts a value into the functor by wrapping it in a trivial effect. `ap` applys effectful functions to effectful values, yielding an effectful value. These operations are subject to unsurprising laws given in [25], but they are not important here. The identity functor, written $\mathbf{I}$, is applicative; applicatives are also closed under composition. An *applicative homomorphism* $\phi : \mathtt{F} \implies \mathtt{G}$ is a natural transformation between applicative functors that commutes with `ap` and `pure` in the obvious way.

A prototypical applicative functor is the datatype `list` of finite lists, interpreted as the effect of non-determinism. `pure a` is the deterministic singleton `[a]`. `ap` applys lists of functions to lists of arguments to get a list of outputs by applying each function to each argument, representing a non-deterministic choice of both. A typical applicative homomorphism would be the transformation that maps a list to the set of its elements. Another important class of examples is given by a constant functor over any monoid, with `pure` and `ap` identified with the unit and multiplication, respectively.

The definition the user should give for `binddt` for `Lam` is as follows. Here, `<*>` is infix notation for `ap`. The helper function `preincr` will be explained below.

```
Fixpoint binddt `{Applicative F} {A B : Type}
  (f : nat * A -> F (Lam B)) : Lam A -> F (Lam B) := match t with
  | Var v => f (0, v)
  | Ap t1 t2 => pure Ap <*> binddt f t1 <*> binddt f t2
  | Abs body => pure Abs <*> binddt (preincr f 1) body
  end.
```

The first non-implicit argument `f` is a *substitution rule* that specifies what should happen at each variable. The role of `binddt` is to apply this substitution rule to each variable in a term. `f` itself takes two arguments. The first, here of type `nat`, represents the number of binders in scope at some variable occurrence, while the second represents the occurrence itself. When `binddt` is specialized to locally nameless case where $\mathtt{A} = \mathtt{B} = \mathtt{LN}$ (recall Figure 2), the second argument to `f` will be either a de Bruijn index or an atom.

The output of `f` has type $\mathtt{F}\,(\mathtt{Lam}\ \mathtt{B})$, representing an expression to replace the occurrence with, with the added flexibility that it may be wrapped in an applicative effect `F`. To account for this effect, the `Ap` and `Abs` cases of `binddt` lift the constructor into `F` with `pure` and replace ordinary function application with effectful application `<*>`. This pattern for working with applicative effects is common enough that there is an established notation of "idiom brackets" [25] (not shown here) to reduce the syntactic clutter.

We draw the reader's attention to the `Var` case: $\mathtt{binddt}\ \mathtt{f}\ (\mathtt{Var}\ \mathtt{v}) = \mathtt{f}\,(0, \mathtt{v})$. This definition is in fact an axiom of DTMs (Equation (1)) and corresponds to the fact that there are no binders in scope in an atomic expression. This may appear to suggest that `f` will only

ever see 0 binders in scope. Actually, `f` is informed about binders using a helper function `preincr` ("precompose increment"), whose definition is $(\texttt{preincr f n})\,(\texttt{n'}, \texttt{v}) = \texttt{f}(\texttt{n} + \texttt{n'}, \texttt{v})$. That `Abs` is a binder is reflected in the recursive call to `binddt`, which modifies `f` with `preincr`. The idea is that when `preincr f 1` is eventually applied to a binding context and a variable, it will increment its binding context before calling `f`. The reader should convince themselves that when the recursion of `binddt f t` bottoms out on a `Var`, the invocation of `f` will be of the form

$$\underbrace{\texttt{preincr}\big(\texttt{preincr}\big(\ldots\big(\texttt{preincr f 1}\big)\ldots\big)\ \texttt{1}\big)\ \texttt{1}}_{n \text{ times}}\,(0, \texttt{v}) = \texttt{f}\,(\texttt{n}, \texttt{v})$$

where $n \in \mathbb{N}$ is the number of `Lam` constructors gone under during recursion.

This scheme is quite general. For example, to extend the lambda calculus with a variadic `Let` construct accepting a list `l` of bound definitions, we can use `preincr f (length l)` to introduce several new entities at once. We can also use monoids other than `nat`. For example, a fully-named representation could use the monoid of finite lists of names under concatenation. The principle limiting what kinds of information one can pass to `f` using `preincr` is that `binddt` must satisfy Equation (3), below, a constraint we discuss further in Section 4.

The final step of instantiating the backend is to prove that `binddt` satisfies a set of four equations. The axioms are given by the following

▶ **Definition 3.1** (DTM, Kleisli-style presentation)**.** *A* traversable monad decorated by a monoid $\langle \texttt{W}, \cdot, 1_{\texttt{W}} \rangle$ *is a type constructor* $\texttt{T} : \texttt{Type} \to \texttt{Type}$ *equipped with operations:*

$$\begin{array}{rlll} \texttt{ret} & (\texttt{A} : \texttt{Type}) & : & \texttt{A} \to \texttt{T A} \\ \texttt{binddt} & (\texttt{Applicative F}) \quad (\texttt{A B} : \texttt{Type}) & : & (\texttt{W} \times \texttt{A} \to \texttt{F}\,(\texttt{T B})) \to \texttt{T A} \to \texttt{F}\ (\texttt{T B}) \end{array}$$

subject to the following four equations:

$$\texttt{binddt}_{\texttt{F}}\ \texttt{f} \circ \texttt{ret} = \texttt{f} \circ \texttt{ret}^{\texttt{W}\times} \tag{1}$$

$$\texttt{binddt}_{\texttt{I}}\ \big(\texttt{ret} \circ \texttt{extract}^{\texttt{W}\times}\big) = \texttt{id}_{\texttt{TA}} \tag{2}$$

$$\begin{aligned} &\texttt{fmap}^{\texttt{F}}\ (\texttt{binddt}_{\texttt{G}}\ \texttt{g}) \circ (\texttt{binddt}_{\texttt{F}}\ \texttt{f}) = \\ &\qquad \texttt{binddt}_{(\texttt{F} \circ \texttt{G})}\ \big(\lambda(\texttt{w}, \texttt{a}).\texttt{fmap}^{\texttt{F}}\,(\texttt{binddt}_{\texttt{G}}\ (\texttt{preincr g w}))\,(\texttt{f}\ (\texttt{w}, \texttt{a}))\big) \end{aligned} \tag{3}$$

$$\phi \circ \texttt{binddt}_{\texttt{F}}\ \texttt{f} = \texttt{binddt}_{\texttt{G}}\ (\phi \circ \texttt{f}) \quad (\textit{for all } \phi : \texttt{F} \implies \texttt{G}\ \textit{applicative hom.}) \tag{4}$$

In Equation (1), $\texttt{ret}^{\texttt{W}\times}$ is defined $\texttt{ret}^{\texttt{W}\times}\texttt{a} = (1_{\texttt{W}}, \texttt{a})$ where $1_{\texttt{W}}$ is the monoid unit. In (2), $\texttt{extract}^{\texttt{W}\times}$ is the projection $\texttt{extract}^{\texttt{W}\times}\,(\texttt{w}, \texttt{a}) = \texttt{a}$. These functions come from the Cartesian-product-with-monoid class of monads (such as used in 2.6 in [38]), known often as the "logging" or "writer" monad. Note that (2) instantiates `binddt` to the identity applicative, while (3) mentions the composition of two applicatives and (4) mentions homomorphisms between two applicatives.

The operations of the locally nameless backend are defined in terms of `binddt` and `ret`, while its lemmas are proved from these four equations only. Notably, this includes properties like `subst_fresh` (recall Figure 3), which is a *conditional* equality, unlike the axioms above. The next two sections show how the axioms of DTMs give rise to high-level properties like `subst_fresh`.

Category theory

Category theorists may wonder if we can give a more "theoretical" definition of DTMs. Law-abiding traversable functors were defined in [21]. Our library extends these to what we call *decorated*-traversable functors and proves the following characterization.

▶ **Theorem 3.2.** *Definition 3.1 is equivalent to a monoid in the category of decorated-traversable functors.*

This theorem is formalized for single-sorted DTMs (Definition 3.1) and a body of general-purpose category theory as shown in Figure 6. This part of Tealeaves is largely separate from the multisorted formalization described in the rest of this paper. A more thorough explanation of this useful perspective shall be forthcoming.

3.2 DTMs as containers

One often has occasion to consider the notion of variable *occurrence*, especially *occurrence in a binding context*. For example, `FV` lists occurrences of free variables, while local closure stipulates that no de Bruijn index $n \in \mathbb{N}$ *occurs under fewer than* $n+1$ *abstractions*. Both operations are more like "aggregations" than "substitutions." `FV` aggregates free variables into a list, while `LC` quantifies over all occurrences, aggregating a set of propositions (one for each occurrence) into a conjunction. It is not so obvious how an equational theory like that of DTMs can incorporate these collection-themed concepts. Tealeaves achieves this by building on a body of work on traversable [25, 18, 21, 8] and shapely [22] functors.

The way to define aggregations is to instantiate the applicative functor `F` to a (constant functor over some) monoid. The most general such choice is the free monoid, i.e. `list`. In particular, we can enumerate occurrences, including their context, as such:[5]

$$\texttt{tolistd} : \texttt{T LN} \to \texttt{list}\ (\mathbb{N} \times \texttt{LN}) \qquad \texttt{tolistd} \overset{def}{\equiv} \texttt{binddt}_{\texttt{list}\,(\mathbb{N}\times\texttt{LN})}\ (\lambda(\texttt{n}, \texttt{v}).\,[(\texttt{n}, \texttt{v})])$$

We also define a context-sensitive notion of variable occurrence ($\in_d$) as a special case of `binddt`. For a term `t` of `Lam LN`, $(\texttt{n}, \texttt{v}) \in_d \texttt{t}$ means a variable `v:LN` occurs somewhere in `t` underneath `n:nat` abstractions.

$$(\texttt{n}, \texttt{v}) \in_d \texttt{t} \overset{def}{\equiv} \texttt{binddt}_{\vee}\ (\lambda(\texttt{n}', \texttt{v}').(\texttt{n}, \texttt{v}) = (\texttt{n}', \texttt{v}'))\ \texttt{t}$$

Here, $\vee$ indicates we instantiate to the monoid of propositions under disjunction. We also provide a version $\texttt{v} \in \texttt{t}$ that checks for occurrences of `v` in any binding context.

Because variable occurrence is defined a special case of `binddt`, we immediately obtain a characterization of how `ret` and `binddt` interact with the occurrence relation.

▶ **Lemma 3.3.** *Equations* (1) *and* (3) *imply the following, respectively.*

$$(\texttt{n}, \texttt{v}_2) \in_d \texttt{ret}\ \texttt{v}_1 \iff \texttt{v}_1 = \texttt{v}_2 \wedge \texttt{n} = 0 \tag{5}$$

$$(\texttt{n}, \texttt{v}_2) \in_d \texttt{binddt}_{\texttt{I}}\ \texttt{f}\ \texttt{t} \iff$$
$$\exists\ \texttt{n}_1\ \texttt{n}_2\ \texttt{v}_1, (\texttt{n}_1, \texttt{v}_1) \in_d \texttt{t} \wedge (\texttt{n}_2, \texttt{v}_2) \in_d \texttt{f}\ (\texttt{n}_1, \texttt{v}_1) \wedge \texttt{n} = \texttt{n}_1 \texttt{+}\ \texttt{n}_2 \tag{6}$$

(5) states that the only variable in an atomic expression occurs with 0 binders in scope. (6) characterizes the set of occurrences in `t` after performing a substitution codified by `f`. If $\texttt{v}_1$ occurs in `t` under $\texttt{n}_1$ binders and $\texttt{v}_1$ is replaced by $\texttt{f}\ (\texttt{n}_1, \texttt{v}_1)$, the occurrences introduced by the subterm have $\texttt{n}_1$ added to their context, in addition to their context as occurrences in $\texttt{f}\ (\texttt{n}_1, \texttt{v}_1)$ – binding context accumulates with tree depth.

[5] Tealeaves generally names context-aware versions of operations with a trailing `d` for *decoration*.

$$\texttt{subst x u} = \texttt{bind}\ (\texttt{subst}_{\text{loc}}\ \texttt{x u}) \qquad \texttt{subst}_{\text{loc}}\ \texttt{x u v} = \begin{cases} \texttt{u} & \text{if } \texttt{v} = \texttt{Fr x} \\ \texttt{ret v} & \text{else} \end{cases}$$

$$\texttt{open u} = \texttt{bindd}\ (\texttt{open}_{\text{loc}}\ \texttt{u}) \qquad \texttt{open}_{\text{loc}}\ \texttt{u}\ (\texttt{n}, \texttt{v}) = \begin{cases} \texttt{Bd (m-1)} & \text{if } \texttt{v} = \texttt{Bd m}, \texttt{m} > \texttt{n} \\ \texttt{u} & \text{if } \texttt{v} = \texttt{Bd n} \\ \texttt{ret v} & \text{else} \end{cases}$$

$$\texttt{close x} = \texttt{fmapd}\,(\texttt{close}_{\text{loc}}\ \texttt{x}) \qquad \texttt{close}_{\text{loc}}\ \texttt{x}\ (\texttt{n}, \texttt{v}) = \begin{cases} \texttt{Bd (m+1)} & \text{if } \texttt{v} = \texttt{Bd m}, \texttt{m} \geq \texttt{n} \\ \texttt{Bd n} & \text{if } \texttt{v} = \texttt{Fr x} \\ \texttt{v} & \text{else} \end{cases}$$

$$\texttt{FV} = \texttt{foldMap}_{\texttt{list}}\ \texttt{FV}_{\text{loc}} \qquad \texttt{FV}_{\text{loc}}\ \texttt{v} = \begin{cases} \texttt{[x]} & \text{if } \texttt{v} = \texttt{Fr x} \\ \texttt{[]} & \text{else} \end{cases}$$

$$\texttt{LC} = \texttt{foldMapd}_{\wedge}\ \texttt{lc}_{\text{loc}} \qquad \texttt{LC}_{\text{loc}}\ (\texttt{n}, \texttt{v}) = \begin{cases} \texttt{n > m} & \text{if } \texttt{v} = \texttt{Bd m} \\ \texttt{True} & \text{else} \end{cases}$$

Figure 7 Locally nameless operations defined as special cases of `binddt`.

Reasoning about syntax often involves conditions on the variable occurrences – for example, `subst_fresh` requires knowledge about the freshness of a given variable. The next theorem gives a pointwise reasoning principle that is used to exploit information about occurrences. This theorem is proved using the coalgebraic presentation of traversability developed in [20].

▶ **Theorem 3.4** (Pointwise reasoning). *Let* $\texttt{T}$ *be a DTM. For all* $t : \texttt{T A}$ *and* $f, g : \texttt{W} \times \texttt{A} \to \texttt{F}\,(\texttt{T B})$ *where* $\texttt{F}$ *is any applicative functor, the following reasoning principle holds.*

$$(\forall\,(\texttt{w} : \texttt{W})\,(\texttt{a} : \texttt{A})\,, (\texttt{w}, \texttt{a}) \in_d \texttt{t} \implies \texttt{f}\ (\texttt{w}, \texttt{a}) = \texttt{g}\ (\texttt{w}, \texttt{a})) \implies \texttt{binddt}_{\texttt{F}}\ \texttt{f t} = \texttt{binddt}_{\texttt{F}}\ \texttt{g t}.$$

3.3 Locally nameless backend

Let $\texttt{T}$ be a DTM decorated by `nat`. We now define the operations of Figure 3 and prove some exemplary lemmas. Our locally backend actually uses multisorted DTMs (defined in the next section), but the basic principles are the same.

The five main operations are defined in Figure 7. On the right, each operation is defined "locally" in terms of its action on individual variable occurrences; three such operations require the number `n` of binders in scope. On the left, each operation is extended to operate on terms using a combinator, all of which are special cases of `binddt`. E.g. `bindd` is `binddt` specialized to the identity applicative, while `fmapd` is like `bindd` for maps rather than substitutions. The `foldMap*` operations instantiate the applicative functor argument of `binddt` to a monoid: `list` uses concentation, while $\wedge$ is shorthand for the monoid of propositions under conjunction. The following properties are proven for all atoms `x:atom` and abstract terms `t, u:T LN`.

▶ **Lemma 3.5** (`subst_fresh`). $\texttt{x} \notin \texttt{FV t} \implies \texttt{subst x u t} = \texttt{t}$.

Proof. Combining Theorem 3.4 with (2) reduces the problem to

$$\forall(\texttt{v} : \texttt{LN}),\ \texttt{v} \in \texttt{t} \implies \texttt{subst}_{\text{loc}}\ \texttt{x u v} = \texttt{ret v},$$

which follows by case analysis on `v` and a lemma $\texttt{y} \in \texttt{FV t} \iff \texttt{Fr y} \in \texttt{t}$ proved by (4). ◀

▶ **Lemma 3.6** (subst_spec). $\texttt{subst x u t} = \texttt{open u (close x t)}$.

Proof. By (3) and (1), $\texttt{open u (close x t)}$ can be fused together to obtain

$$\texttt{bindd } (\lambda(\texttt{n}, \texttt{v}).\texttt{open}_{\texttt{loc}}\ \texttt{u}\ (\texttt{n}, \texttt{close}_{\texttt{loc}}\ \texttt{x}\ (\texttt{n}, \texttt{v})))\ \texttt{t}.$$

One then shows the middle expression is equal to $\lambda(\texttt{n}, \texttt{v}).\texttt{subst}_{\texttt{loc}}\ \texttt{x u v}$ by case analysis. ◀

▶ **Lemma 3.7** (fv_subst_upper). $\texttt{FV}\,(\texttt{subst x u t}) \subseteq (\texttt{FV t} \setminus \{\texttt{x}\}) \cup \texttt{FV u}$.

Proof. By $\texttt{x} \in \texttt{FV t} \iff \texttt{Fr x} \in \texttt{t}$, the problem reduces to

$$\texttt{Fr y} \in \texttt{subst x u t} \implies (\texttt{Fr y} \in \texttt{t} \wedge \texttt{y} \neq \texttt{x}) \vee (\texttt{Fr y} \in \texttt{u})\,.$$

This follows by rewriting the left hand side with (6), and case analysis. ◀

The vast majority of proofs in our locally nameless backend proceed along similar lines: fusing sequential operations with (1) and (3), using (5) and (6) to reason about how operations affect the set of occurrences, applying pointwise reasoning to prove equalities, and case analysis on concrete variables.

3.4 Multisorted DTMs

In practice, few formal systems involve just one sort of variable. For example, polymorphic lambda calculi like System F include both type and term variables. Fixed-point extensions of first-order logic [19], which provide a theoretical foundation for languages like Datalog [2], involve both term and relation variables, as do second-order logics. For such systems it is necessary to consider a generalization of Definition 3.1 that supports parallel substitution of *more than one sort of variable at the same time*. This is because our approach rests on the assumption that all substitution and related operations are special cases of a single operation, so they can always be fused together with an appropriate generalization of (3). This is our motivation for introducing Definition 3.8.

▶ **Definition 3.8** (Multisorted DTM). *Let K be a set of sorts. Let* $\texttt{T} : \texttt{K} \to \texttt{Type} \to \texttt{Type}$ *be a* $\texttt{K}$*-indexed set of type constructors and let* $\texttt{U}$ *be a type constructor. A* traversable T-module decorated by a monoid $\langle \texttt{W}, \cdot, 1_{\texttt{W}} \rangle$ *is defined from the following data:*

$$\begin{array}{rlcl} \texttt{ret} & (\texttt{A} : \texttt{Type}) & : & \forall\,(\texttt{k} : \texttt{K})\,,\ \texttt{A} \to \texttt{T k A} \\ \texttt{binddt} & (\texttt{Applicative F})\ (\texttt{A B} : \texttt{Type}) & : & (\forall(\texttt{k} : \texttt{K}),\ \texttt{W} \times \texttt{A} \to \texttt{F}\,(\texttt{T k B})) \to \texttt{U A} \to \texttt{F}\ (\texttt{U B}) \end{array}$$

subject to conditions generalizing those of Definition 3.1.

For short, we call an instance of Definition 3.8 a $\texttt{K}$-sorted DTM (where "M" technically stands for "right Module.") When $\texttt{K}$ is the unit type and $\texttt{T tt} = \texttt{U}$ ($\texttt{tt}$ being Coq's name for the constructor of unit type), this definition reduces to Definition 3.1. In general, such structures represent a grammatical category $\texttt{U}$ inside which one can substitute any of $|\texttt{K}|$-many different kinds of variables in parallel. Reasoning with multisorted DTMs works as before, but now incorporating case analysis on $\texttt{K}$ as well.

4 Evaluating Tealeaves

Next we discuss how we evaluate Tealeaves. First we describe the size and scope of Tealeaves' core and backend before discussing case studies instantiating Tealeaves with different kinds of syntax. Then we offer a feature-wise comparison to two popular syntax frameworks for Coq, Autosubst versions 1 and 2 [32, 36] and LNgen [5].

Implementation

We recall Figure 6, which shows the division of Tealeaves into two major components:

- Core Tealeaves, which formalizes DTMs and the properties discussed in Section 3.2.
- The locally nameless backend, which develops generic locally nameless syntax infrastructure like that shown in Section 3.3.

The core is a formalization of multisorted DTMs (Definition 3.8). On top of the axioms we implement an additional layer of high-level derived theory like that presented in Section 3.2, the chief export of which is a proof of Theorem 3.4. Additionally, the core includes a general formalization of numerous category-theoretic concepts used to prove Theorem 3.2 for single-sorted DTMs, but this is primarily of theoretical interest; it does not affect end-users and would not be required to port Tealeaves to another proof assistant like Agda. Altogether, as measured by `coqwc`, the core includes about 10,000 lines of specification (including imports, notations, etc.) and 9,000 lines of proof. Of these, the essential parts formalizing multisorted DTMs account for about 2,000 lines of specification of 1,000 of proof.

The locally nameless backend includes a core part independent of DTMs that formalizes basic notions like atoms, sets, and environments. This part consists of about 2000 lines derived from the Metalib library, a component of LNgen, lightly adapted to fit into our more category-theoretic framework. The locally nameless infrastructure, which is parameterized by a DTM instance, consists of about 1000 lines of specification and 650 of proof. The backend export several dozen high-level infrastructural lemmas like the ones generated by LNgen, as well as many other lower-level lemmas. Examples of lemmas include the ones proved in Section 3.3, as well as generalizations that describe the interaction between operations that act on different sorts of variables.

The locally nameless backend supports the claim that the DTM abstraction is adequate for reasoning about raw syntax generically. Next we ask whether this concept is actually useful, i.e. does it save labor, and for which kinds of syntax does it work?

Case studies

So far we have implemented a few different case studies with Tealeaves.

STLC Our first study is a proof of type soundness for the simply-typed lambda calculus (STLC). We use Alectryon [27] to present this file in the form of browser-based tutorial on Equations (1)–(4), demonstrating the general strategy for proving each one. We also provide an alternate version of this tutorial that uses the category-theoretic description of DTMs indicated in Theorem 3.2.

System F In the second study, we instantiate Tealeaves with the syntax of System F before proving type soundness for this system. This makes essential use of multisorted DTMs and the ability of our backend to reason about non-trivial interactions between substitution operations that act on different sorts of variables.

Variadic binding We are developing tutorials demonstrating how to instantiate Tealeaves with languages featuring mutually-inductively defined grammatical categories and variadic binders, such as a `letrec` construct.

The cost to instantiate Tealeaves is to define `binddt` and prove multisorted versions of Equations (1)–(4). These proofs proceed by induction on terms, where each case proceeds by rewriting with laws like those of applicative functors. In the future, we expect to provide automated support for the instantiation process. Happily, three of the DTM axioms are straightforward to prove in most cases, regardless of the user's syntax. Equation (1) defines the behavior of `binddt` on variables and is proved with the `reflexivity` tactic, while (2)

and (4) are straightforward inductive proofs. Equation (3), however, presents a challenge when binding information (i.e. data passed with `preincr`) is computed from an argument which itself is subject to substitution. A key example of this phenomenon is a variadic `let` (or `letrec`) construct that accepts a `list` of definitions, in which case `binddt f` is defined to pass the length of the list to `f` in the `let` body. The bound definitions are themselves subject to substitution with `binddt`, and it is not immediately clear how to prove that this does not change the length of the list, a key requirement of Equation (3) manifest in the two occurrences of `w`. This requires applying the representation theorem for traversals [18], which states that shape is invariant under traversals.

Comparison to other utilities

Three commonly used utilities for automating syntax infrastructure in Coq are Autosubst [32], Autosubst 2 [36], and LNgen [5], all of which involve dynamically generating infrastructure after being provided with a user's syntax. The Autosubst family represent variables as de Bruijn indices, while LNgen generates locally nameless infrastructure. In some ways Tealeaves is more general than these utilities, as the operations they reason about are special cases of `binddt`. Table 1 summarizes the features offered by the utilities.

Autosubst provides tool support for working with de Bruijn indices based on the σ-calculus [1], a version of untyped λ-calculus extended with explicit substitution. Given an `Inductive` definition of a user's syntax, the user calls upon Ltac to synthesize a parallel substitution operation and a small number of equational axioms for this operation. Users invoke a complete decision procedure, `autosubst`, which proves all true equalities between a delineated class of *substitution expressions* from these axioms. Semanticists generate these goals while developing metatheory and call on Autosubst to solve them.

Autosubst provides only ad-hoc support for substitution involving multiple sorts of variables. The limitations of Ltac also prevent their automation from working with mutually-inductively defined grammatical categories. The authors note that the fragile semantics of Ltac mean it is sometimes necessary to manually inspect generated definitions for errors.

Autosubst 2 is an external code generator written in Haskell which accepts a second-order specification of a syntax and generates Coq modules containing proofs of the equations to instantiate an extended calculus that handles multisorted substitution much the same way we do. Compared to the first version, Autosubst 2 handles potentially mutually inductive grammatical categories with multiple kinds of variables. The authors conjecture, but do not prove, that their modified calculus is confluent. Users who modify their syntax must re-execute the external program to reinstantiate the Autosubst library.

The Autosubst family does not provide support for conditional equalities or operations that compute the set of free variables, perhaps because these are not as essential when using de Bruijn indices as when using a locally nameless representation.

LNgen is a code generator that, given an annotated grammar in an Ott-compatible [33] format, generates Coq files containing the operations of locally nameless and proof scripts than synthesize infrastructural lemmas. The scripts proceed by induction and are based on the authors' "knowledge of how such proofs usually go." As with Autosubst 2, modifying the syntax involves re-executing the utility. In private correspondence, the authors of LNgen have reported to us cases of long compile times (about 30 minutes in some cases) and the potential for some proofs to fail, requiring manual intervention from the user. As with Autosubst, this problem is exacerbated by the opaque semantics of Ltac.

Table 1 Features supported by Coq syntax frameworks.

Utility	Representation	Underlying theory	Multisorted	Variadic Binders
Autosubst	de Bruijn	σ-calculus	Ad-hoc	No
Autosubst 2	de Bruijn	σ-calculus	Yes	No
LNgen	Locally nameless	Structural recursion	Yes	No
Tealeaves	Generic	DTMs	Yes	Yes

5 Related work

Besides Autosubst and LNgen, there are syntax metatheory frameworks for Coq that share some of Tealeaves' features but lack the principled theory and flexibility of DTMs.

GMeta [23] is prior art implementing a generic Coq framework for first-order syntax metatheory. Like Tealeaves, it features an extensible architecture supporting multi-sorted syntax and multiple representations of variable binding. However, the implementations differ substantially because GMeta lacks a principled abstraction of syntax like DTMs, considering instead a universe of representable types. In effect, one has a set of type expressions and a denotation mapping these into Coq's types; generic proofs proceed by induction on an expression denoting a type. By contrast, we showed in Section 3.3 how infrastructural lemmas with Tealeaves proceed by the equational theory of DTMs. The user's cost of entry for GMeta is to prove the type of their syntax is representable up to isomorphism, which is supported with automation.

DBlib [31] is a community-maintained Coq library that supports reasoning about de Bruijn indices. Like Tealeaves, it is based on a structured recursion combinator subject to axioms, but these axioms are ad-hoc and not pure equations, whereas (1)–(4) are equations derived from a principled theory of structured monads as manifest in Theorem 3.2. Using results from Section 3.2, it is easy to see that DBlib's axioms are immediate corollaries of DTMs. For instance, their axiom `TraverseVarIsIdentity` can be derived by specializing Theorem 3.4 to $\mathtt{g} = \mathtt{ret} \circ \mathtt{extract}^{\mathtt{W}\times}$ and simplifying with (2).

The application of monads to formal syntax metatheory was proposed by Bellegarde and Hook [7], who considered a combinator `Ewp` ("extension with policy") that is reminiscent of `binddt` but less expressive and lacking an axiomatization. Work building on the monadic approach, typically using a de Bruijn representation, has emphasized well-scoped [9, 4] and well-scoped, well-typed [3] syntax. Fiore and Szamozvancev have recently introduced an Agda framework for well-typed syntax that is inspired by work on presheaf-theoretic models of syntax [16, 15]. The heavy use of dependent types in this work leads to a workflow in which the types of operations are very nearly their own correctness properties, whereas our "raw" approach separates the definition of operations from their metatheory. Investigating the theoretical relation between the two approaches may be an interesting direction for future work.

Two fundamentally different formalization strategies for abstract syntax are higher-order abstract syntax (HOAS) [26] and techniques using nominal sets [28], both of which are closely associated with dedicated-purpose proof assistants. Implementing HOAS in Coq requires using a variation like parametric higher-order abstract syntax (PHOAS) [12]. Nominal sets are generally used with first-class support from the proof assistant, such as in Nominal Isabelle [37].

6 Conclusion and future work

We have presented Tealeaves, a generic Coq framework for reusable syntax metatheory. We showed how a user instantiates Tealeaves by proving their syntax forms a DTM, allowing them to specialize a body of generic infrastructure lemmas to their syntax. We evaluated Tealeaves with case studies instantiating locally nameless infrastructure to languages with multiple sorts of variables, mutually-inductive grammatical categories, and variadic binders.

Tealeaves offers a number of interesting directions for future investigation. Currently we are investigating precisely which kinds of syntax and reasoning work well with Tealeaves. More precisely, we are exploring how the core theory of DTMs can be modified to accommodate more sophisticated situations than raw terms with locally nameless variables or de Bruijn indices.

Well-scoped syntax

We initially sought an abstraction for raw syntax, largely because this representation is simple and commonly used. However, there are convincing theoretical and practical reasons to consider intrinisically well-scoped syntax. We are investigating how to extend DTMs to the well-scoped setting. As a first step, let `LN` be parameterized by a context `ctx` of free variables and by a maximum value `n` for de Bruijn indices using Coq's type `Fin.t` of finite sets, as follows.

```
Inductive LN (ctx : list atom) (n : nat) : Type :=
| Fr : forall (a : atom), In a ctx -> LN ctx n
| Bd : Fin.t n -> LN ctx n.
```

The type of lambda terms is generalized to allow the set of variables to be parameterized by the number of entities in scope (here, `preincr V n` maps `m` to the set `V (n + m)`).

```
Inductive Lam (V : nat -> Type) :=
| Var : V 0 -> Lam V
| Abs : Lam (preincr V 1) -> Lam V
| App : Lam V -> Lam V -> Lam V.
```

The type of locally closed terms with free variables in `ctx` is then `Lam (LN ctx)`. For example, the term `Var (Fr x)` can be given this type if `x ∈ ctx`. On the other hand, the open term `Var (Bd 0)` cannot be given this type, while it can be given type `Lam (preincr (LN ctx) 1)`.

A generalization of Definition 3.1 can be formulated for this situation on paper. An unfortunate limitation of Coq's type theory is that types like `LN ((n + m) + p)` and `LN (n + (m + p))` are not defintionally equal, hence their terms cannot even be compared for equality, obstructing a naïve attempt to formalize this definition in Coq. Further parameterizing `V` by types would also move closer towards the type-preserving approach of McBride [24].

Fully named variables

The Tealeaves repository includes a generalization of Definition 3.1 that additionally takes a binder-renaming operation, with which we intend to implement a fully named Tealeaves backend. With such an extension, our aim is to give a certified change in representation between locally nameless and fully named variables. One use case would be to implement a verified programming language in Coq using locally nameless while allowing programmers to write code with named variables, assured that the change in representation introduces no bugs.

References

1 M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.

3 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *Journal of Functional Programming*, 31:e22, 2021. doi:10.1017/S0956796820000076.

4 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. doi:10.1007/3-540-48168-0_32.

5 Brian Aydemir and Stephanie Weirich. LNgen: Tool Support for Locally Nameless Representations. Technical report, University of Pennsylvania, Department of Computer and Information Science, June 2010. URL: https://repository.upenn.edu/cis_reports/933/.

6 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 50–65, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11541868_4.

7 Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.*, 23(2–3):287–311, December 1994. doi:10.1016/0167-6423(94)00022-0.

8 Richard Bird, Jeremy Gibbons, Stefan Mehner, Janis Voigtländer, and Tom Schrijvers. Understanding idiomatic traversals backwards and forwards. *SIGPLAN Not.*, 48(12):25–36, September 2013. doi:10.1145/2578854.2503781.

9 Richard S. Bird and Ross Paterson. De bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999. doi:10.1017/s0956796899003366.

10 Rod M. Burstall. Proving properties of programs by structural induction. *Comput. J.*, 12(1):41–48, 1969. doi:10.1093/comjnl/12.1.41.

11 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/s10817-011-9225-2.

12 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1411204.1411226.

13 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

14 David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. doi:10.1007/3-540-44404-1_7.

15 Marcelo Fiore. Second-order and dependently-sorted abstract syntax. In *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*, LICS '08, pages 57–68, USA, 2008. IEEE Computer Society. doi:10.1109/LICS.2008.38.

16 Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 193–202. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782615.

17 Marcelo Fiore and Dmitrij Szamozvancev. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498715.

18 Jeremy Gibbons and Bruno Oliveira. The essence of the iterator pattern. *J. Funct. Program.*, 19:377–402, July 2009. doi:10.1017/S0956796809007291.

19 Yuri Gurevich and Saharon Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32(nil):265–280, 1986. doi:10.1016/0168-0072(86)90055-2.

20 Mauro Jaskelioff and Russell O'Connor. A representation theorem for second-order functionals. *Journal of Functional Programming*, 25, February 2014. doi:10.1017/S0956796815000088.

21 Mauro Jaskelioff and Ondrej Rypacek. An investigation of the laws of traversals. *Electronic Proceedings in Theoretical Computer Science*, 76, February 2012. doi:10.4204/EPTCS.76.5.

22 C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In Donald Sannella, editor, *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 1994. doi:10.1007/3-540-57880-3_20.

23 Gyesik Lee, Bruno C. D. S. Oliveira, Sungkeun Cho, and Kwangkeun Yi. GMeta: A generic formal metatheory framework for first-order representations. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 436–455. Springer, 2012. doi:10.1007/978-3-642-28869-2_22.

24 Conor McBride. Type-preserving renaming and substitution. Unpublished note, 2005. URL: http://strictlypositive.org/ren-sub.pdf.

25 Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008. doi:10.1017/S0956796807006326.

26 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988. doi:10.1145/53990.54010.

27 Clément Pit-Claudel. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, pages 155–174, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3426425.3426940.

28 Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003. Theoretical Aspects of Computer Software (TACS 2001). doi:10.1016/S0890-5401(03)00138-X.

29 Randy Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 313–332, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

30 Randy Pollack. Reasoning about languages with binding: Can we do it yet?, February 2006. URL: https://web.archive.org/web/20101122040606/http://homepages.inf.ed.ac.uk/rpollack/export/bindingChallenge_slides.pdf.

31 François Pottier and Coq maintainers. DBlib. https://github.com/coq-community/dblib, 2019.

32 Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Xingyuan Zhang and Christian Urban, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI. Springer-Verlag, August 2015. doi:10.1007/978-3-319-22102-1_24.

33 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1291151.1291155.

34 Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-71067-7_23.

35 Bas Spitters and Eelis Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21, February 2011. doi:10.1017/S0960129511000119.

36 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 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, pages 166–180, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294101.

37 Christian Urban and Christine Tasson. Nominal techniques in isabelle/hol. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, pages 38–53, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11532231_4.

38 Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. Association for Computing Machinery. doi:10.1145/143165.143169.