Polymonads

Extended version of paper submitted to POPL'13

Nataliya Guts† Michael Hicks† Nikhil Swamy⋆ Daan Leijen⋆ Gavin Bierman⋆

†University of Maryland, College Park ⋆Microsoft Research

Abstract

From their semantic origins to their use in structuring effectful computations, monads are now also used as a programming pattern to structure code in a number of important scenarios, including functional reactivity, information flow tracking and probabilistic computation. However, whilst these examples are inspired by monads they are not strictly speaking monadic but rather something more general. The first contribution of this paper is the definition of a new structure, the polymonad, which subsumes monads and encompasses the monad-like programming patterns that we have observed. A concern is that given such a general setting, a program would quickly become polluted with polymonadic coercions, making it hard to read and maintain. The second contribution of this paper is to build on previous work to define a polymorphic type inference algorithm that supports programming with polymonads using a direct style, e.g., as if computations of type $M\ \tau$ were expressions of type τ. During type inference the program is rewritten to insert the necessary polymonadic coercions, a process that we prove is coherent—all sound rewritings produce programs with the same semantics. The resulting programming style is powerful and lightweight.

1. Monads and more

For most programmers, a monad is an abstract datatype represented by a unary type constructor m and two operations, $bind$ and $unit$ (a.k.a., *return*), with the following signature:

$$\begin{array}{l} bind : \forall\alpha,\beta.\ m\ \alpha \rightarrow (\alpha \rightarrow m\ \beta) \rightarrow m\ \beta \\ unit : \forall\alpha.\ \alpha \rightarrow m\ \alpha \end{array}$$

Implementations of these operations are expected to obey the *monad laws*, which enable reasoning about transformations of monadic programs, both by programmers and by tools such as optimizing compilers [19].

Since the time that Moggi first connected monads to effectful computation [18], monads have proven to be a surprisingly versatile computational structure. Perhaps best known as the foundation of Haskell's support for state, I/O, and other effects, monads have also been used to structure APIs for libraries that implement a wide range of programming tasks, including parsers [13], probabilistic computations [22], functional reactivity [8, 4], and information flow tracking [23].

While conceptually simple, programming directly against a monadic API is impractical. Programmers must insert calls to $bind$ and $unit$ pervasively, and when composing multiple monads, *morphisms* between the monads must also be inserted. However, effective type inference algorithms have been devised to reduce this burden. In the context of Haskell, monadic type inference relies on the mechanisms of typeclasses and a specialized syntax (the do notation) to infer the placement of $bind$s and $unit$s. For ML, our own prior work [24] has shown that the existing let-structure of a call-by-value program can be used to infer the placement of the morphisms in addition to the $bind$s and $unit$s.

The monadic programming pattern is sufficiently appealing that many researchers have developed subtle variations to adapt and apply monads to new problem domains. Examples include Wadler and Thiemann's [27] indexed monad for typing effectful computations; Atkey's parameterized monad [2], which has been used to encode disciplines like regions [15] and session types [21]; Devriese and Piessens' [7] monad-like encodings for information flow controls; Danielsson's [6] counting monad for computational complexity; and many others. Oftentimes these extensions are used to prove stronger properties about computations than would be possible with monads, or to prevent undesirable behavior (such as illegal information flows, memory errors, etc.).

We observe that in each of these cases a family of abstract datatypes $\{m_1, \ldots, m_n\}$ with $bind$ and $unit$-like operations is provided. But, unlike for traditional monads, the $bind$s have signatures of the form

$$\forall\alpha,\beta.m_1\ \alpha \rightarrow (\alpha \rightarrow m_2\ \beta) \rightarrow m_3\ \beta$$

We call binds of this form *non-uniform binds* and refer to the collection of m_i and the binds among them as a *polymonad*.

This paper explores the idea of polymonads, a generalization of monads and monad morphisms. Section 2 defines polymonads precisely, including laws that are analogs of the monad and morphism laws. While the notion of a non-uniform bind has been considered previously [16], the laws underlying the behavior of polymonads have never been articulated. As with monads, these laws are important for reasoning about intuitive program transformations. The laws are also important for type inference, as we explain shortly. We show that every set of monads and monad morphisms can be encoded using polymonads while obeying the polymonad laws.

Next, in Section 3, we present a variety of examples and show them to be polymonads. Our examples include several from the literature mentioned previously, as well as two new polymonadic constructions. The first is an encoding of information flow controls in the presence of side effects, and the second is an encoding of contextual effects [20]. We show that each of our examples obey the polymonad laws.

To make polymonads easier to program with, in Section 4 we develop a novel type inference and rewriting algorithm for an ML-like, call-by-value programming language. Rather than write non-uniform binds in programs directly, the programmer can use expressions of type $m\,\tau$ as if they were of type τ, thus programming in a direct style. Our algorithm will automatically infer which non-uniform binds are needed and where they should be placed. While in Haskell the do notation must be used to identify where $bind$s and $unit$s should be placed (and type class inference determines which binds/units to insert), for ML no special syntax is needed. General-

constructors	$\mathcal{M}$	$::=$	$Id \mid M, \mathcal{M}$
signature	$\Sigma_{\mathcal{M}}$	$::=$	$\mathsf{b}_{Id,Id,Id} \mid \mathsf{b}{:}s, \Sigma_{\mathcal{M}}$
bind type	s	$::=$	$\forall\alpha\beta.\, m_1\alpha \to (\alpha \to m_2\beta) \to m_3\beta$

Figure 1. Syntax of a polymonadic signature

izing our prior work [24], normal let bindings and applications naturally identify where binds should be placed when programming with (poly)monadic computations, thanks to ML's call-by-value, fixed evaluation order. We prove that our algorithm enjoys principal types.

One might wonder why a special algorithm is needed: could not a standard type inference algorithm based on OML [14], the framework of qualified types on which Haskell's typeclass inference algorithm is based, be applied to polymonads? Indeed, this has been done for the typeclass-based encoding of polymonads in Kmett's `Control.Monad.Parameterized` package for Haskell [16]. But this approach has two important problems.

First, because vanilla type class inference is unaware of the polymonadic laws many simple programs are rejected. For example, we cannot use `Control.Monad.Parameterized` to check that the program `\x. id (id x)` has the type $\alpha \to m\ \alpha$ for some polymonad m. Haskell infers type class constraints for this program that could allow ambiguous instantiations: different instantiations of certain type variables could result in different program semantics. In Section 5, we prove a coherence result for our type inference algorithm for polymonadic programs, extending our prior result for monadic programs. We show that by appealing to the polymonad laws, instantiations that OML would conservatively deem as ambiguous can in fact be accepted, as there is no possibility for ambiguity of a program's semantics. Thus our algorithm permits many useful functions that would otherwise be rejected.

Second, principal types turn out to be exceedingly hard to read. Even simple terms have a large number of quantified variables and constraints. However, as we show in Section 6, by applying constraint simplifications justified by the polymonad laws we can *improve* types to make them much simpler without negatively impacting their generality.

Section 7 describes a simple prototype implementation we have built to check the examples in the paper.

Concurrently with our work, Tate [25] developed a semantic construction he calls *productors*. After some interaction, we discovered that polymonads match a specialization of productors applied to what Tate calls *semi-strict* effect systems. Tate proves that such effect systems are quite general, and thus polymonads are equally general. As polymonads are focused on practical programming, while productors are focused on semantics, our two papers provide complementary views of an exciting, underlying generalization of monads. We discuss more related work in Section 8.

2. Polymonads

We begin by defining polymonads from the perspective of a programmer, who may think of them as a set of abstract types with a collection of operations which, when taken together, must respect a specific set of equations. To help provide intuition as to the implications of these equations, we prove that every collection of monad and monad morphisms also induces a polymonad. The increased expressive power of polymonads is put on display in the next section, which incorporates them into a simple programming language and presents a series of examples.

2.1 Syntax

Figure 1 defines the signature $\Sigma_{\mathcal{M}}$ of a set $\mathcal{M}$ of polymonadic abstract types. The set $\mathcal{M}$ contains unary type constructors M, including a distinguished type constructor Id. We expect all the type constructors occurring in $\Sigma_{\mathcal{M}}$ to be in $\mathcal{M}$. We write m as a metavariable ranging over monadic type constructors. When the set of constructors is irrelevant or clear from the context, we refer to the signature as Σ, rather than $\Sigma_{\mathcal{M}}$.

The signature represents a map from names b to types s where each type has the shape $\forall\alpha\beta.m_1\alpha \to (\alpha \to m_2\beta) \to m_3\beta$. We refer to each b as a *bind*, for its relation to the monadic bind operation [26], although, unlike a monadic bind, a polymonadic bind involves three abstract types. We also refer to the signature Σ as the *bind set*. Syntactically, we require the bind set Σ to contain an element $\mathsf{b}_{Id,Id,Id}$ which implicitly has the type $\forall\alpha\beta.Id\alpha \to (\alpha \to Id\beta) \to Id\beta$. One may think of $Id\tau$ as a synonym for τ, in which case (adhering to the laws below) $\mathsf{b}_{Id,Id,Id}$ is reverse apply.

We use the following shorthands:

- $(m_1, m_2) \rhd m$ is the type $\forall\alpha\beta.m_1\alpha \to (\alpha \to m_2\beta) \to m\beta$.
- $(m_1, m_2) \rhd m \in \Sigma$ means $\exists\mathsf{b}.\mathsf{b}{:}(m_1, m_2) \rhd m \in \Sigma$. This notation is unambiguous as we assume that binds b with the same type s have the same semantics.
- We write $\mathsf{b}_{m_1,m_2,m} \in \Sigma$ to mean $\mathsf{b}{:}(m_1, m_2) \rhd m \in \Sigma$.
- We define unit_m, with type $\forall\alpha.\alpha \to m\alpha$, to be $\lambda x.\mathsf{b}_{Id,Id,m}\ x\ (\lambda z.z)$, and write $\mathsf{unit}_m \in \Sigma$ to mean $(Id, Id) \rhd m \in \Sigma$.
- We write $m_1 \succ m_2$ to mean $(m_1, Id) \rhd m_2$ or $(Id, m_1) \rhd m_2$.

2.2 Polymonad laws

We impose several requirements on Σ for it to define a valid polymonad. First, we require each bind operation to be given an *interpretation* in an underlying typed lambda calculus. We choose System F [11] as the underlying calculus in this paper, although for clarity, we omit explicit type abstraction and application in System F terms. Thus, we write $\mathsf{b}_{m_1,m_2,m_3}\ e_1\ e_2$ to mean the application in System F of the interpretation of b_{m_1,m_2,m_3} to two terms $e_1 : m_1\ \tau$ and $e_2 : \tau \to m_2\tau'$. Equivalent formulations in other calculi are also feasible.

Any valid interpretation of a bind set Σ is expected to respect certain equations between well-typed terms, where we interpret equality as β,η-equivalence. We also require a bind set to be closed, in the sense that the presence of certain binds in Σ mandate the existence in Σ of other, related binds. These requirements are expressed by the six polymonad laws below (which we explain in detail shortly); we write $\models \Sigma$ when a signature satisfies these laws.

Functorial law: For all $m \in \mathcal{M}$, we require $(m, Id) \rhd m \in \Sigma$. That is, every m comes equipped with a *map* function.

Left identity: For all m_1, m_2, e, k, if $\mathsf{b}_{m_1,m_2,m_2} \in \Sigma$ and $\mathsf{unit}_{m_1} \in \Sigma$, then $\mathsf{b}_{m_1,m_2,m_2}\ (\mathsf{unit}_{m_1}\ e)\ k = k\,e$.

Right identity: For all m_1, m_2, e, if $\mathsf{b}_{m_1,m_2,m_1} \in \Sigma$ and $\mathsf{unit}_{m_2} \in \Sigma$, then $\mathsf{b}_{m_1,m_2,m_1}\ e\ \mathsf{unit}_{m_2} = e$.

Associativity:

(1) For all m_1, m_2, m_3, m_{123},
$\exists m_{12}.\{\mathsf{b}_{m_1,m_2,m_{12}}, \mathsf{b}_{m_{12},m_3,m_{123}}\} \subseteq \Sigma$
if and only if
$\exists m_{23}.\{\mathsf{b}_{m_1,m_{23},m_{123}}, \mathsf{b}_{m_2,m_3,m_{23}}\} \subseteq \Sigma$.

(2) For all $m_1, m_2, m_3, m_{12}, m_{23}, m_{123}, e_1, k_2, k_3$,
if $\{\mathsf{b}_{m_1,m_2,m_{12}}, \mathsf{b}_{m_{12},m_3,m_{123}}, \mathsf{b}_{m_1,m_{23},m_{123}}, \mathsf{b}_{m_2,m_3,m_{23}}\} \subseteq \Sigma$,
then $\mathsf{b}_{m_{12},m_3,m_{123}}\,(\mathsf{b}_{m_1,m_2,m_{12}}\ e_1\ k_2)\,k_3 = \mathsf{b}_{m_1,m_{23},m_{123}}\ e_1\,(\lambda x.\,\mathsf{b}_{m_2,m_3,m_{23}}\ (k_2\ x)\,k_3)$.

$$
\begin{array}{llll}
\text{monad sig.} & \mathcal{S} & ::= & (D_{Id}, D_1, ..., D_n, f_{M_1,M_1'}, ..., f_{M_k,M_k'}) \\
\text{monad} & D & ::= & (M, unit_M, bind_M), f_{M,M} \\
\text{morphism} & f_{M,N} & : & M \rhd N
\end{array}
$$

$$
\frac{M \in \mathcal{S}}{\mathcal{S} \models M \rhd M} \qquad \frac{M \in \mathcal{S}}{\mathcal{S} \models Id \rhd M} \qquad \frac{f : M_1 \rhd M_2 \in \mathcal{S}}{\mathcal{S} \models M_1 \rhd M_2}
$$

$$
\frac{\mathcal{S} \models M_1 \rhd M_2 \quad \mathcal{S} \models M_2 \rhd M_3}{\mathcal{S} \models M_1 \rhd M_3}
$$

$$
\begin{array}{ll}
bind_M\ (unit_M\ e)\ k\ =\ k\ e & (i) \\
bind_M\ e\ unit_M\ =\ e & (ii) \\
bind_M\ (bind_M\ e\ k_1)\ k_2 = & \\
\qquad bind_M\ e\ (\lambda x.\ bind_M\ (k_1\ x)\ k_2) & (iii) \\
f_{M_1,M_2}(unit_{M_1}\ e)\ =\ unit_{M_2}\ e & (iv) \\
f_{M_1,M_2}\ (bind_{M_1}\ e\ k)\ = & \\
\qquad sbind_{M_2}\ (f_{M_1,M_2}\ e)\ (f_{M_1,M_2} \circ k) & (v) \\
f_{M_2,M_3} \circ f_{M_1,M_2}\ = f_{M_1,M_3} & (vi) \\
f_{M,M}\ e\ =\ e & (vii)
\end{array}
$$

Figure 2. Monads

Paired morphisms law: For all $m_1, m_2 \in \mathcal{M}$, if $(m_1, Id) \rhd m_2 \in \Sigma$ then $(Id, m_1) \rhd m_2 \in \Sigma$, and vice versa.

Composition closure law: For all $m_1, m_2, m_3, m_1', m_2', m_3'$, if $(m_1, m_2) \rhd m_3 \in \Sigma$ and if $\{m_1' \succ m_1, m_2' \succ m_2, m_3 \succ m_3'\} \subseteq \Sigma$, then $(m_1', m_2') \rhd m_3' \in \Sigma$.

We choose these laws for two reasons. First, we aim for polymonads to be a generalization of monads, yielding reasoning principles with which programmers are already familiar, e.g., associativity of bind operations, which provides justification for many simple program transformations. This motivates our choice of the first four laws.

Second, we aim to exploit the polymonad laws to devise an coherent constraint solving procedure for type inference. Thus, when certain binds can be defined unambiguously in terms of other binds, we require these to be present in the bind set. This is the motivation behind the last two laws.

For example, consider the paired morphism law. We think of both $(m_1, Id) \rhd m_2$ and $(Id, m_1) \rhd m_2$ as *morphisms* between m_1 and m_2 and write $m_1 \succ m_2$ for a morphism between m_1 and m_2. Given a $\mathsf{b}_{m_1,Id,m_2} \in \Sigma$ one can easily define b_{Id,m_1,m_2} as $\lambda x.\lambda f.\mathsf{b}_{m_1,Id,m_2}(fx)(\lambda x.x)$. It is easy to check that this is the only interpretation of b_{Id,m_1,m_2} consistent with the other laws. A similar construction yields b_{m_1,Id,m_2} from b_{Id,m_1,m_2}.

The composition closure law allows morphisms to be composed with binds. For example, given $\mathsf{b}_1{:}m_1' \succ m_1$, $\mathsf{b}_2{:}m_2' \succ m_2$, $\mathsf{b}_3{:}m_3 \succ m_3'$, and $\mathsf{b}{:}(m_1, m_2) \rhd m_3$, one can define $\mathsf{b}_{m_1',m_2',m_3'}$ as $\lambda x.\lambda f.\mathsf{b}_3\ (\mathsf{b}\ (\mathsf{b}_1\ x\ (\lambda y.y))(\lambda x.\mathsf{b}_2\ (f x)\ (\lambda y.y)))\ (\lambda y.y)$, where we apply b_1 and b_2 contravariantly, and b_3 covariantly. As with the paired morphism law, it is easy to check that this construction is the only way to compose the binds and morphisms in a manner compatible with the rest of the laws, so we simply require these binds to be present.

2.3 Comparison to conventional monads

Here we formalize the relationship between conventional monads and polymonads.

Monad families. We define a family of monads using a *monadic signature* $\mathcal{S}$, defined in Figure 2. A signature consists of a series of monad definitions D and a series of morphisms $f_{M,N}$ between monads M and N. The relation $\mathcal{S} \models M \rhd N$ indicates that according to signature S we can convert a monad M into a monad N, either reflexively, via a morphism, or via morphism composition. We also assume the existence of a monad Id whose semantics is the identity for $unit_{Id}$ and reverse apply for $bind_{Id}$; as such $unit_M$ for all M can be viewed as a morphism from Id to M.

A monadic signature is well-formed, denoted $\models \mathcal{S}$, if morphisms in $\mathcal{S}$ only refer to monads also defined in $\mathcal{S}$, and if the morphisms and monadic operators satisfy the laws given at the bottom fo Figure 2. Observe that the polymonad laws are structurally similar the conventional monad laws. In particular, making all indices $m_1, ...m_6$ equal to a single M maps the polymonad laws *left identity*, *right identity*, and *associativity* to the monad laws (i)—(iii), respectively. In fact, we can prove that all monads are polymonads.

Translating signatures. We write $\langle S \rangle$ to denote the polymonad $\mathcal{M}, \Sigma_{\mathcal{M}}$ equivalent to monadic signature S, where $\mathcal{M} = \{M \mid (M, unit_M, bind_M) \in S\}$ and $\Sigma_{\mathcal{M}} = Clos(S)$, defined as follows:

$$
\begin{aligned}
Clos(S) = \{ & \mathsf{b}_{M_1 M_2 M} : (M_1, M_2) \rhd M \mid \\
& S \models M_1 \rhd M \ \wedge\ S \models M_2 \rhd M\}
\end{aligned}
$$

Given the definitions of its binds and morphisms, we can define each of the polymonadic binds $\mathsf{b}_{M_1 M_2 M} \in Clos(S)$ as follows:

$$
\mathsf{b}_{M_1 M_2 M}\ x\ g = bind_M\ (f_{M_1,M}\ x)\ (\lambda y.\ f_{M_2,M}\ (g\ y))
$$

That the morphisms $f_{M_1,M}$ and $f_{M_2,M}$ are present in S follows directly from the definition of $Clos(S)$.

We can prove that polymonad and monad laws coincide for polymonadic representation of monads.

Lemma 1. *$\models S$ iff $\models \langle S \rangle$ for all S.*

Proof. See Appendix A. ☐

Example As an example of this translation, consider the signature $S = (D_{Id}, (M, bind_M, unit_M), f_{M,M})$. The corresponding polymonad is $\mathcal{M} = M, Id$ and

$$
\begin{aligned}
\Sigma_{\mathcal{M}} = {} & \mathsf{b}_{MMM} : (M, M) \rhd M,\ \mathsf{b}_{Id,Id,M} : (Id, Id) \rhd M, \\
& \mathsf{b}_{Id,M,M} : (Id, M) \rhd M,\ \mathsf{b}_{M,Id,M} : (M, Id) \rhd M, \\
& \mathsf{b}_{Id,Id,Id} : (Id, Id) \rhd Id
\end{aligned}
$$

We can define the polymonad binds using bind_M and unit_M as follows (after simplifying terms produced by the definition above):

$$
\begin{aligned}
& \mathsf{b}_{M,M,M} = bind_M \\
& \mathsf{b}_{Id,Id,M}\ x\ f = unit_M\ (f\ x) \\
& \mathsf{b}_{Id,M,M}\ x\ f = bind_M\ (unit_M\ x)\ f = f\ x \\
& \mathsf{b}_{M,Id,M}\ x\ f = bind_M\ x\ (\lambda y.unit_M\ (f\ y))
\end{aligned}
$$

2.4 Categorical foundations

Whilst the focus of our work is on the programmatic aspects of polymonads, we have developed some categorical analysis of polymonads. Our categorical model consists of a collection of functors (modeling the type constructors) and over this a collection of natural transformations of the form $T_1(T_2(A)) \to T_3(A)$ where the T_i are taken from the collection of functors. The collection of natural transformations must satisfy a number of conditions that are the categorical analogs of the polymonad laws from Section 2.2. Some details of our categorical model appear in Appendix D. Interestingly, almost identical categorical constructions have been independently proposed by Tate as models of his generalized effects framework [25]. (Indeed we are grateful to Tate for spotting some shortcomings in our initial model.)

3. Programming with polymonads

The previous section defines a core, abstract definition for polymonads. In this section, we define λPM, a lambda calculus with support for polymonadic programming. λPM integrates polymonads with constructs familiar from the polymorphic lambda calculus. Notably, rather than insisting on having only unary type constants in $\mathcal{M}$, we will permit polymonadic type constructors to have additional parameters, and to carry constraints on these parameters within types.

We develop a series of λPM examples demonstrating the usefulness of polymonads. With our type inference algorithm, detailed in the next section, programmers write λPM programs in direct style, our algorithm infers polymonadic types and then elaborates the typed source programs into System F with explicit applications of the polymonadic bind operations.

3.1 λPM: A language for polymonadic programming

Figure 3 presents the syntax of λPM, a call-by-value, polymorphic lambda calculus. Its term language is standard—we have variables x, constants c, undecorated λ-abstractions, function application and let-bindings. We expect λPM programs to be written in direct style, as computations of type $m\,\tau$ are simply expressions of type τ. We then infer where and which bind operations must be inserted to typecheck a λPM program against a polymonadic interface.

To make inference feasible, we rely crucially on λPM's call-by-value structure. Following our prior work on monadic programming for ML, we restrict the shape of types assignable to a λPM program by separating value types τ from the types of polymonadic computations $m\ \tau$. The co-domain of every function is required to be a computation type $m\ \tau$, although pure functions can be typed $\tau \to \tau'$, which is a synonym for $\tau \to Id\ \tau'$. We also include types $T\ \bar{\tau}$ for fully applied type constructors, e.g., $\mathsf{list}\ int$.

Programs can also be given type schemes σ that are polymorphic in their polymonads, e.g., $\forall\alpha\mu\beta.(\alpha \to \mu\beta) \to \alpha \to \mu\beta$. Here, the variable α ranges over all value types τ, while μ ranges over computation types m. Type schemes may also be qualified by a set of bind constraints, P. For example, $\forall\alpha\mu.(\mu, Id) \rhd M \Rightarrow (int \to \mu\ int) \to M\ int$ is the type of a function that abstracts over a morphism $\mu \succ M$.

λPM is parameterized by a set of polymonad type constructors $\mathcal{M}$, where each constructor $M/k \in \mathcal{M}$ is a $(k+1)$-ary type constructor (unlike in Section 2 where we only considered unary constructors). For example, we may write polymonadic types like $\mathsf{ST}\ h\ int$, indexing the state monad $\mathsf{ST}/1$ with a phantom type h for a heap variable, as is common in a language like Haskell. We often omit the arity k for brevity. Note, our metavariable m for polymonadic types now includes both polymonad variables μ, as well as polymonadic constants $M\bar{\tau}$ applied to a sequence of type indices. Our intention is that type indices are *phantom*, meaning that they are used as a type-level representation of some property of the polymonad's current state, but a polymonadic bind's implementation does not depend on them. For example, we would expect that binds would treat objects of type $\mathsf{ST}\,h\,\tau$ uniformly, for all h; different values of h would be used to statically prevent unsafe operations like double-frees or dangling pointer dereferences. If an object has different states that would affect the semantics of binds, the programmer can use different constructors M for each state (rather than different type indices and the same constructor).

As before, a bind set $\Sigma_{\mathcal{M}}$ is a map from bind names b to their types s. However, unlike in Section 2, where we required Σ to be specified intensionally as a set, here, we allow an extensional definition of Σ using a language of *theory constraints* Φ. λPM's type system is parametric in the choice of theory constraints Φ. This generality allows us to encode a variety of prior monad-like systems with λPM.

values	v	$::=$	$x \mid c \mid \lambda x.e$
expressions	e	$::=$	$v \mid e_1\ e_2 \mid \mathsf{let}\ x\!=\!e_1\ \mathsf{in}\ e_2$
types	τ	$::=$	$() \mid \alpha \mid T\ \bar{\tau} \mid \tau_1 \to m\ \tau_2$
type schemes	σ	$::=$	$\forall\bar{\nu}.P \Rightarrow \tau$
monadic types	m	$::=$	$\mathrm{M} \mid \mu$
ground monads	M	$::=$	$M\ \bar{\tau}$
type variables	ν	$::=$	$\alpha \mid \mu$
bind constraint	π	$::=$	$(m_1, m_2) \rhd m$
bind constraints	P	$::=$	$\cdot \mid \pi, P$
substitutions	θ	$::=$	$\cdot \mid \mu \mapsto m \mid \alpha \mapsto \tau \mid \theta, \theta$
environment	Γ	$::=$	$\cdot \mid \Gamma, c{:}\sigma \mid \Gamma, x{:}\sigma$
Parameterized by:			
k-ary constructors	$\mathcal{M}$	$::=$	$Id/0 \mid M/k, \mathcal{M}$
bind set	$\Sigma_{\mathcal{M}}$	$::=$	$\mathsf{b}_{Id,Id,Id} \mid \mathsf{b}{:}s, \Sigma_{\mathcal{M}}$
bind types	s	$::=$	$\forall\bar{\alpha}.\Phi \Rightarrow (\mathrm{M}_1, \mathrm{M}_2) \rhd \mathrm{M}_3$
theory constraints	Φ		
theory entailment	$\models$: $2^{\Sigma \times P \times \theta \times \bar{b}}$		

Figure 3. Syntax of λPM

For example, to model Wadler and Thiemann's [27] indexed monad, which represents a type and effect system, we can introduce a polymonadic constructor $W/1$, and use $W\ \epsilon\ \tau$ to represent a computation that produces a τ-result after exhibiting effects contained within the set ϵ. To specify a polymonadic bind for W, we can use λPM's bind type $\forall\alpha,\beta,\epsilon_1,\epsilon_2,\epsilon_3.(\epsilon_3 = \epsilon_1 \cup \epsilon_2) \Rightarrow (W\ \epsilon_1, W\ \epsilon_2) \rhd W\ \epsilon_3$. Here, we have instantiated the theory Φ to include equality and set operators like $\cup$, and the bind type indicates (informally) that when composing two computations, the effects are additive.

To interpret the theory constraints, λPM requires a theory entailment relation $\models$, where elements of this relation are written $\Sigma \models \theta P \leadsto \bar{\mathsf{b}}$. This states that for each $\pi_i \in P$, the bind $\mathsf{b}_i{:}\theta\pi_i$ is provided by Σ, for some substitution θ of the free variables of P. We write $\Sigma \models \theta P$ when we do not care about the elaborated binds. The solving of inferred bind constraints in λPM is complete modulo the completeness of a decision procedure for the entailment relation $\models$. Note, however, that the type schemes σ for a λPM program are entirely independent of the choice of the theory—Φ constraints never appear in a type scheme σ.

Of course, we require the entailment relation to still define a polymonad, i.e., $\models$ is admissible if and only if the set $\{\mathsf{b}_{m_1,m_2,m} \mid \Sigma \models \cdot(m_1, m_2) \rhd m \leadsto \mathsf{b}\}$ satisfies the polymonad laws.

3.2 Parameterized monads

Our first example shows that Atkey's parameterized monad [2] is a polymonad and illustrates how it can be used to program safe communication protocols. Atkey proposes an abstract data type A, a ternary type constructor with two operations, $unitA$ and $bindA$, with the signature shown below.

$$unitA : \forall\alpha\phi. \quad \alpha \to A\,\phi\,\phi\,\alpha$$
$$bindA : \forall\alpha\beta\phi\gamma\psi.\ A\,\phi\,\gamma\,\alpha \to (\alpha \to A\,\gamma\,\psi\,\beta) \to A\,\phi\,\psi\,\beta$$

The type constructor $A\ p\ q\ \tau$ can be thought of (informally) as the type of a computation producing a τ-typed result, with a pre-condition p and a post-condition q. The $bindA$ operator matches the post-condition parameter of the first computation with the pre-condition parameter of the composing function, producing a computation having the pre-condition of the former and the post-condition of the latter. The $unitA$ operator lifts a pure computation into a parameterized monad with the same pre- and post-condition.

Notice that $A\ p\ q$ is not a monad, for all indexes p and q—a unit is only available when the indexes are the same, and the type indices vary in the bind operator. However, Atkey's construction can be seen as a polymonad with the signature given below.

$$
\begin{array}{rcl}
\mathcal{M} & = & Id, A/2 \\
\Sigma & = & \mathsf{b}_{Id,Id,Id}, \\
& & \mathsf{mapA} : \forall \phi, \psi.\ (A\ \phi\ \psi, Id) \rhd A\ \phi\ \psi, \\
& & \mathsf{appA} : \forall \phi, \psi.\ (Id, A\ \phi\ \psi) \rhd A\ \phi\ \psi, \\
& & \mathsf{unitA} : \forall \phi.\ (Id, Id) \rhd A\ \phi\ \phi, \\
& & \mathsf{bindA} : \forall \phi\gamma\psi.\ (A\,\phi\,\gamma,\ A\,\gamma\,\psi) \rhd A\,\phi\,\psi
\end{array}
$$

The bind set Σ includes mapA, the functorial map over A (as required by the functorial law). The paired morphism law requires us to include appA (since it is dual to mapA). We include unitA, the analog of Atkey's $unitA$—note that the polymonad laws do not require us to provide a unit for every instance of A. Finally, we have bindA, the analog of $bindA$—of course, the varying type constructors are natural with polymonads.

The composition closure law requires us to close the bind set Σ under the composition of a morphism and a bind. Here, we have one non-trivial morphism, i.e., unitA : $\forall \phi. Id \succ A\ \phi\ \phi$. Composing unitA with bindA we get binds of the form $\forall \phi\psi.\ (A\,\phi\,\psi, Id) \rhd A\,\phi\,\psi$ and $\forall \phi\psi.\ (Id, A\,\phi\,\psi) \rhd A\,\phi\,\psi$. These have exactly the same form as mapA and appA, so Σ is already closed. Note, by instantiating the indexes, one can see Σ as an infinite set of binds. Also, observe that in this example the theory constraints Φ are empty; however, as we will see shortly, we still require an entailment relation for bind constraints over the empty theory.

Session types. To check the remaining polymonad laws, we need to instantiate the abstract type A, provide interpretations for each bind, and check that they satisfy the necessary equations. As an example, we choose Pucella and Tov's encoding of session types [21]. This provides a way to safely program two-party communication protocols. In what follows, we write $Sess$ as a synonym for A, to emphasize the connection to session types.

The type $Sess\,\phi\,\gamma\,\alpha$ represents a computation involved in a two-party session which starts in protocol state ϕ and completes in state γ, returning a value of type α. We begin by instantiating the language of phantom type indices to describe protocol states and transitions. Concretely, we use the type index $send\,\gamma\,\alpha$ to denote a protocol state that requires a message of type α to be sent, and then transitions to γ. Similarly, the type index $recv\,\psi\,\beta$ denotes the protocol state in which once a message of type β is received, the protocol transitions to ψ. We also use the index end to denote the protocol end state. The signatures of two primitive operations for sending and receiving messages captures this behavior.

$$
\begin{array}{l}
\mathsf{send} : \forall \alpha\gamma.\, \alpha \to Sess\,(send\,\gamma\,\alpha)\,\gamma\,() \\
\mathsf{recv} : \forall \alpha\gamma.\, () \to Sess\,(recv\,\gamma\,\alpha)\,\gamma\,\alpha
\end{array}
$$

The concrete representation of the abstract type $Sess\ \phi\ \gamma\ \alpha$ is simple. Assuming that the underlying language provides support for primitive effects like I/O, $Sess\ \phi\ \gamma\ \alpha$ is just a synonym for α. Under this interpretation, all five bind operations correspond to reverse function application—it is easy to check that this interpretation satisfies the left- and right-identities and the associativity law.

Using these definitions, consider the following λPM program that implements one side of a simple protocol that sends a message x, waits for an integer reply y, and returns y+1.

let go = λx. **let** _ = send x **in** incr (recv ())

Type inference in λPM is closely related to the algorithm of Jones' OML [14], a variant of which is also implemented by Haskell. In Section 4, we show how λPM programs can be embedded in OML, and how, based on OML's principal types property, we can compute a principal type of λPM programs. Using this strategy, we compute the following principal type for go:

$$
\begin{array}{l}
\forall \alpha, \beta, \gamma, \mu_{10}, \mu_2, \mu_8, \mu_9, \mu_5, \mu_3, \mu_6, \mu_7, \mu_6, \mu_{11}, \\
(\mu_2, \mu_8) \rhd \mu_{10}, \\
(\mu_5, Id) \rhd \mu_9, \\
(Id, Sess\,(send\ \ \alpha\,\beta)\alpha) \rhd \mu_3, \\
(Id, \mu_9) \rhd \mu_8), \\
(Id, \mu_3) \rhd \mu_2), \\
(Id, Sess\,(recv\ \ \gamma\ int)\,\gamma) \rhd \mu_6), \\
(Id, \mu_6) \rhd \mu_5), \\
\Rightarrow\ (\beta \to \mu_{10}\ int)
\end{array}
$$

Inference also produces a rewritten term that contains, or abstracts, the needed binds. For this example, the term starts with a sequence of λ abstractions, one for each of the seven bind constraints. If we imagine these are numbered b1 ... b7 then the main body looks as follows

```
λx. b1 (b5 send (λ y. b3 x y))
       (λ _. b4 incr
             (λ z. b2 (b7 recv (λ w. b6 () w)) z))
```

While maximally general, the principal type of go is also unreadable! Worse yet, OML (and Haskell) reject this type as ambiguous. The reason is that the constraint set contains many variables that do not appear in the final type $\beta \to \mu_{10}\ int$. A general-purpose solver for such constraints (not being aware of the polymonad laws) assumes that the particular instantiation of these variables could influence the semantics of the program, and so requires a programmer to explicitly instantiate each of these variables.

Thankfully, by exploiting the polymonad laws, λPM can do much better. In Section 5, we show that for a given result type, all possible solutions to a set of polymonad constraints are coherent, i.e., they have the same semantics. Next, in Section 6, we show how the polymonad laws allow us to aggresively improve types, eliminating constraints that we show cannot affect typability. Using our improvement procedure, the type of go becomes slightly more readable (as does the rewritten term, which we elide):

$$
\begin{array}{l}
\forall \alpha, \beta, \gamma, \mu_{10}. \\
(Sess\,(send\ \ \alpha\,\beta)\,\alpha, Sess\,(recv\ \ \gamma\ int)\,\gamma) \rhd \mu_{10}, \\
(Sess\,(recv\ \ \gamma\ int)\,\gamma, Id) \rhd Sess\,(recv\ \ \gamma\ int)\,\gamma, \\
(Id, Sess\,(recv\ \ \gamma\ int)\,\gamma) \rhd Sess\,(recv\ \ \gamma\ int)\,\gamma, \\
(Id, Sess\,(send\ \ \alpha\,\beta)\,\alpha) \rhd Sess\,(send\ \ \alpha\,\beta)\,\alpha \\
\Rightarrow\ (\beta \to \mu_{10}\ int)
\end{array}
$$

This type is still rather unwieldy. But, note that all but the first constraint are tautologies. By the functorial and paired morphism laws, we know that for every instantiation of α, β, γ, there is guaranteed to be a bind in Σ of the form $(m, Id) \rhd m$ and $(Id, m) \rhd m$. Thus, we can further simplify the type to the one shown below.

$$
\begin{array}{l}
\forall \alpha, \beta, \gamma, \mu_{10}. \\
(Sess\,(send\ \ \alpha\,\beta)\,\alpha), (Sess\,(recv\ \ \gamma\ int)\,\gamma)) \rhd \mu_{10} \\
\Rightarrow\ (\beta \to \mu_{10}\ int)
\end{array}
$$

Finally, when at the top-level a programmer calls go 0, and instantiates the result type, say, to $Sess\ end\ end$, we obtain a constraint $\pi\ =\ (Sess\,(send\ \ \alpha\,\beta)\alpha), (Sess\,(recv\ \ \gamma\ \ int)\,\gamma)) \rhd Sess\ end\ end$. To complete typing go 0, we make use of the decision procedure $\models$ to solve this constraint. For this particular example, without any theory constraints, the relation $\models$ is a simple unification-based procedure that can compute the substitution $\theta = \alpha \mapsto (recv\ \ end\ \ int), \gamma \mapsto end$, such that bindA $\in \Sigma$ can be instantiated to $\theta\pi$.

3.3 Polymonadic information flow controls

Several researchers have proposed type systems or libraries with a monad-like structure to implement information flow controls [7, 23, 17, 5, 1]. These controls allow a programmer to indicate that some program inputs are secrets, and that some outputs are public. The goal is to ensure that the public outputs are independent of the secret inputs—a property called noninterference [12]. In this section, we develop a polymonad, IST, suitable for enforcing information flow controls in stateful programs.

Our encoding has several elements. First, as usual, we introduce a lattice of security labels $l \in \{L, H\}$, each a nullary type constructor. We assume $L < H$ in a lattice ordering, indicating that data labeled H is more secret than data labeled L.

Next, we introduce a type of integer references,[1] $\mathit{intref}\ l$, a reference to an integer value labeled l. Finally, we have a polymonadic type constructor IST which takes two phantom type indices. In particular, $IST\, pc\, l\, \tau$ classifies a computation that potentially writes to references labeled l and returns a τ-result that is labeled l. For instance, $IST\ H\ L$ is the type of a computation that may write to secret storage cells while computing a public value.

To make this notion precise, we introduce a lattice theory Φ and use it in the definition of the bind set below.

$$
\begin{array}{lcl}
\mathcal{M} & = & IST/2 \\
\Phi & ::= & \tau \le \tau \mid \Phi, \Phi \\
\Sigma & = & \mathsf{b}_{Id,Id,Id}, \\
 & & \mathsf{appIST} : \forall pc, l.(Id, IST\ pc\ l) \rhd IST\ pc\ l, \\
 & & \mathsf{mapIST} : \forall pc, l.(IST\ pc\ l, Id) \rhd IST\ pc\ l, \\
 & & \mathsf{bIST} : \forall pc_1, l_1, pc_2, l_2, pc_3, l_3. \\
 & & \qquad l_1 \le pc_2, l_1 \le l_3, l_2 \le l_3, pc_3 \le pc_1, pc_3 \le pc_2 \\
 & & \qquad \Rightarrow (IST\ pc_1\ l_1, IST\ pc_2\ l_2) \rhd IST\ pc_3\ l_3
\end{array}
$$

When composing a computation $IST\ pc_1\ l_1\ \alpha$ with a function $\alpha \to IST\ pc_2\ l_2\ \beta$, the type of bIST requires $l_1 \le pc_2$ to prevent the second computation from leaking information about its l_1-secure α-typed argument into a reference cell that is less that l_1-secure. Dually, the next two constraints ensures that the β-typed result of the composed computation is at least as secure as the results of each component. The last two constraints ensure that the effect lower bound of the composed computation is a lower bound of the effects of each component.

To interpret the theory constraints, we instantiate $\models$ to a standard theory of lattice contraints. Specifying this decision procedure is orthogonal to the purposes of this paper. However, for a finite lattice of labels, an easy decision procedure simply constructs the finite enumeration of binds obtained by instantiating bIST in all ways consistent with the lattice constraints Φ. Deciding whether a particular bind constraint π is derivable then simply amounts to testing the membership of π in the finite enumeration, for some substitution θ of the free variables in π.

To implement IST, we must give it a representation in System F and provide an interpretation for bIST and check that it satisfies the polymonad laws. One choice is for IST to be the state monad augmented with phantom label indexes, as shown below.

```
heap = list (int ∗ int) (∗ an assoc. list for a heap ∗)
intref l = int (∗ intref l is a synonym of int ∗)
IST αβγ= heap → (γ ∗ heap) (∗ state monad with phantom indexes ∗)
bIST = λc f h = let (x,h) = c h in f x h
```

We can then give polymonadic signatures to functions for allocating, reading, and writing references, as shown below.

$$
\begin{array}{lcl}
\mathsf{alloc} & : & \forall pc.\ \mathit{int} \to IST\ pc\ pc\ (\mathit{intref}\ pc) \\
\mathsf{read} & : & \forall l.\ \mathit{intref}\ l \to IST\ H\ l\ \mathit{int} \\
\mathsf{write} & : & \forall l\ \mathit{intref}\ l \to \mathit{int} \to IST\ l\ L\ ()
\end{array}
$$

With these definitions in hand, we can write the following λPM program, infer a principal type for it (very verbose, as with sessions), and improve its type using the polymonad laws to the one shown below.

λPM source: **let** add_interest = λsavings. λinterest.
  let current = read savings **in**
  write savings (current + interest)

improved type: $\forall l, \mu.(IST\ H\ l, IST\ l\ L) \rhd \mu \Rightarrow \mathit{intref}\ l \to \mathit{int} \to \mu\,()$

Consider applying add_interest secret_ac 100, where secret_ac : $\mathit{intref}\ l$ is a reference to a bank account at a secrecy level l, for some l. We now have a constraint $(IST\ H\ l, IST\ l\ L) \rhd \mu$ to solve. If we instantiate the variable μ to $IST\ l\ l$ which produces type $\forall l.\mathit{intref}\ l \to \mathit{int} \to IST\ l\ l\ ()$, we must check that $(IST\ H\ l, IST\ l\ L) \rhd IST\ l\ l$ exists, and it does: the theory constraints on bIST are satisfied since the resulting pc label $l \le l$ and $l \le H$; and, likewise, the resulting confidentiality label $l \ge L$ and $l \ge l$. Intuitively, the type of add_interest tells us that it is a function that writes to its argument reference and returns a value that is at least as secret as the contents of that reference. This type is precise as far as its effect lower bound goes, but it is a little imprecise in that its result value is (), hence uninformative, and so could be given the label L. Overall, polymonadic information flow tracking provides a convenient syntactic way of eliminating information leaks in a program, but, being syntactic, it is necessarily imprecise.

3.4 Contextual type and effect systems

We have already sketched an encoding of Wadler and Thiemann's [27] type and effect systems as polymonad. As our final example, we show that a recent type and effect system for *contextual effects* [20] (which subsumes traditional type and effects) is a polymonad. We define a polymonad $CE\ \alpha\ \epsilon\ \omega\ \tau$, for the type of a computation which itself has effect ϵ and produces a value of type τ, and appears in a context in which the prior computations have effect α and whose subsequent computations have effect ω.

As with our information flow encoding, we start by describing a language of type indices to describe effect sets. Indices are sets of atomic effects $a_1 \dots a_n$, with $\emptyset$ the empty effect, $\top$ the effect set that includes all other effects, and $\cup$ the union of two effects. We also introduce theory constraints for subset relations and extensional equality on sets, with the obvious interpretation.

$$
\begin{array}{lll}
\text{types} & \tau & ::= \ \cdots \mid a_1 \dots a_n \mid \emptyset \mid \top \mid \tau_1 \cup \tau_2 \\
\text{theory constraints} & \Phi & ::= \ \tau \subseteq \tau' \mid \tau = \tau' \mid \Phi, \Phi
\end{array}
$$

The following binds capture the tracking of contextual effects:

$$
\begin{array}{lcl}
\mathcal{M} & = & Id, CE/3 \\
\Sigma & = & \mathsf{b}_{Id,Id,Id}, \\
 & & \mathsf{unitce} : (Id, Id) \rhd CE\ \top\ \emptyset\ \top \\
 & & \mathsf{bindce}_{Id} : \forall \alpha_1, \alpha_2, \epsilon_1, \epsilon_2, \omega_1, \omega_2. \\
 & & \quad (\alpha_2 \subseteq \alpha_1, \epsilon_1 \subseteq \epsilon_2, \omega_2 \subseteq \omega_1) \Rightarrow \\
 & & \quad (Id, CE\ \alpha_1\ \epsilon_1\ \omega_1) \rhd CE\ \alpha_2\ \epsilon_2\ \omega_2 \\
 & & \mathsf{bindce} : \forall \alpha_1, \epsilon_1, \omega_1, \alpha_2 \epsilon_2, \omega_2, \epsilon_3. \\
 & & (\epsilon_2 \cup \omega_2 = \omega_1, \epsilon_1 \cup \alpha_1 = \alpha_2, \epsilon_1 \cup \epsilon_2 = \epsilon_3) \Rightarrow \\
 & & \quad (CE\ \alpha_1\ \epsilon_1\ \omega_1, CE\ \alpha_2\ \epsilon_2\ \omega_2) \rhd CE\ \alpha_1\ \epsilon_3\ \omega_2
\end{array}
$$

The bind unitce lifts a computation into a contextual effect monad with empty effect and any prior or future effects. The bind bindce_{Id} expresses that it is safe to consider an additional effect for the current computation (the ϵs are covariant), and fewer effects for the prior and future computations (αs and ωs are contravariant). Finally, bindce composes two computations such that the future effect of the first computation includes the effect of the second one, provided that the prior effect of the second computation includes the first computation; the effect of the composition includes both effects, while the prior effect is the same as before the first

[1] A generalization to polymorphic references is feasible, but we use integer references here to keep our heap model simple.

computation, and the future effect is the same as after the second computation.

As with Atkey's monad, it is easy to check that Σ is closed under the composition with the single non-trivial morphism bindce_{Id}. However, unlike our other examples, we do not provide an example of programming with contextual effects. Typing any non-trivial example requires an implementation of a decision procedure for the set theory we have introduced—something that our current prototype lacks. In the future, we aim to extend our work to the setting of an SMT-based dependently typed programming language, where we hope to make use of the theory support of the underlying solver to efficiently decide polymonadic theory constraints.

4. Type inference for λPM

This section presents a formalization of type inference for λPM. We defer to the next section how type inference is (also) drives the conversion of the direct-style source program to an elaborated program containing the needed polymonadic binds. We prove that our type system enjoys principal types via a sound and complete translation to Jones' OML [14] (the theoretical foundation of Haskell type classes), that type inference produces principal types. However, this is not the end of the story—the next two sections show that appealing to the polymonadic laws makes it possible to safely accept what might otherwise be deemed ambiguous programs, and to simplify principal types without reducing their generality.

4.1 Type rules

Figure 4 gives the syntax-directed type rules of our language. As shown in earlier work [24] it is straightforward to give declarative rules too. There are two forms of rules, one for values, $P \mid \Gamma \vdash v : \tau$ and one for expressions $P \mid \Gamma \vdash e : m\ \tau$ (which have a polymonadic type). The rules state that under an environment Γ, and predicates P, the value v or expression e have type τ or $m\ \tau$ respectively.

The rules are all straightforward: Rules (TS-Var) and (TS-Const) look up a variable or constant in the environment and return an instantiated type scheme. Note how the rule (TS-Inst) ensures that the constraints in the type scheme are entailed by the assumed predicates P. The entailment relation is defined as:

$$\frac{\pi \in P}{P \models \pi} \qquad \frac{\Sigma \models \cdot\pi}{P \models \pi} \qquad \frac{P \models \pi_1 \quad \ldots \quad P \models \pi_n}{P \models \pi_1, \ldots, \pi_n}$$

Basically, a predicate π must be implied either by the bind set Σ, or be an element of the assumed predicates P.

The rule (TS-Lam) types lambda abstractions while (TS-Id) lifts a value into the identity monad. Rule (TS-Let) generalizes over values in the usual way.

Rule (TS-App) ensures that there exists a bind expression that can apply the function (with monadic type m_3) to the argument (m_2) taking it to some monadic type m_4, and that there exist a bind that can combine the evaluation of the function (of type m_1) with that result monad (m_4) into a final monadic type m_5. Rule (TS-Do) is similar, but requires only the existence of a bind between the bound expression and the body of the let.

4.2 Principal types

The type rules admit principal types, and there exists an efficient type inference algorithm that finds such types. The way we show this is by a translation of polymonadic terms (and types) to terms (and types) in OML [14] and prove this translation is sound and complete: a polymonadic term is well-typed if and only if its translated OML term has an equivalent type.

We encode terms in our language into OML as shown in Figure 5. We rely on three primitive OML terms that force the typing

$$\begin{array}{ll}
\mathsf{id} & : \forall \alpha.\ \alpha \to Id\ \alpha \\
\mathsf{do} & : \forall \alpha\beta\mu_1\mu_2\mu.\ ((\mu_1,\mu_2) \rhd \mu) \\
 & \quad\ \Rightarrow \mu_1\ \alpha \to (\alpha \to \mu_2\ \beta) \to \mu\ \beta \\
\mathsf{app} & : \forall \alpha\beta\mu_1\mu_2\mu_3\mu_4\mu.\ ((\mu_1,\mu_4) \rhd \mu, (\mu_2,\mu_3) \rhd \mu_4) \\
 & \quad\ \Rightarrow \mu_1\ (\alpha \to \mu_3\ \beta) \to \mu_2\ \alpha \to \mu\ \beta
\end{array}$$

$$\begin{array}{lll}
[\![x]\!]^\star & = x & \\
[\![c]\!]^\star & = c & \\
[\![\lambda x.e]\!]^\star & = \lambda x.[\![e]\!] & \\
 & & \\
[\![v]\!] & = \mathsf{id}\ [\![v]\!]^\star & \\
[\![e_1\ e_2]\!] & = \mathsf{app}\ [\![e_1]\!]\ [\![e_2]\!] & \\
[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] & = \mathsf{let}\ x = [\![v]\!]^\star\ \mathsf{in}\ [\![e]\!] & \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = \mathsf{do}\ [\![e_1]\!]\ [\![\lambda x.e_2]\!]^\star & (\text{with } e_1 \neq v)
\end{array}$$

Figure 5. Translation of λPM to OML

of the terms to generate the same constraints as our type system does: id for lifting a pure term, do for typing a do-binding, and app for typing an application. Using these primitives, we encode values and expressions of our system into OML.

We write $P \mid \Gamma \vdash_{\text{OML}} e : \tau$ for a derivation in the syntax directed inference system of OML (cf. Jones [14], Fig. 4).

Theorem 2 (Elaboration to OML is sound and complete)**.**
Soundness: *Whenever $P \mid \Gamma \vdash v : \sigma$ we can also derive $P \mid \Gamma \vdash_{\text{OML}} [\![v]\!]^\star : \sigma$ in OML. Similarly, when $P \mid \Gamma \vdash e : m\ \tau$ we have $P \mid \Gamma \vdash_{\text{OML}} [\![e]\!] : m\ \tau$.*
Completeness: *If we can derive $P \mid \Gamma \vdash_{\text{OML}} [\![v]\!]^\star : \sigma$, there also exists a derivation $P \mid \Gamma \vdash v : \sigma$, and similarly, whenever $P \mid \Gamma \vdash_{\text{OML}} [\![e]\!] : m\ \tau$, we also have $P \mid \Gamma \vdash e : m\ \tau$.*

The proof is by straightforward induction on the typing derivation of the term. It is important to note that our system uses the same instantiation and generalization relations as OML which is required for the induction argument. Moreover, the constraint entailment over bind constraints, also satisfies the monotonicity, transitivity and closure under substitution properties required by OML.

As a corollary of the above properties, we have that our system admits principal types via the general-purpose OML type inference algorithm.

5. Coherence

By the result in the previous section, we could perform type inference and rewriting for polymonads by using OML's algorithm, which is essentially the Haskell type class inference algorithm. Unfortunately, this algorithm is not satisfactory, as it would reject many useful programs. Translated to our setting, the limitation of this algorithm is that it rejects any term whose type $\forall\bar{\nu}.P \Rightarrow \tau$ binds a variable $\mu \in \bar{\nu}$ such that μ appears in a constraint $\pi \in P$ but does not appear in the final type τ. We call such variables *open variables* and such constraints *open constraints*. Many polymonadic terms have open constraints; one example was given in Section 3.2.

The reason such terms are rejected in OML is that instantiations of open variables have operational effect—the instantiations determine which *evidence* will be used when evaluating a term (in our setting, the evidence is the binds), and different evidence could have different operational effect.

However, it turns out for polymonads we do not have the same problem: all possible solutions to open constraints will produce terms having the same semantics; i.e., the solutions are *coherent*. To show coherence, we have to show that the programs resulting from evidence translation (i.e. how we insert binds) are operationally equivalent no matter which solution we choose for open

$$\boxed{P \,|\, \Gamma \vdash v : \tau \quad P \,|\, \Gamma \vdash e : m\,\tau}$$

$$\frac{\Gamma(x) = \sigma \qquad P \models \sigma \geqslant \tau}{P \,|\, \Gamma \vdash x : \tau}\ \text{(TS-Var)} \qquad \frac{\Gamma(c) = \sigma \qquad P \models \sigma \geqslant \tau}{P \,|\, \Gamma \vdash c : \tau}\ \text{(TS-Const)} \qquad \frac{\theta = [\bar{m}/\bar{\mu}][\bar{\tau}/\bar{\alpha}] \qquad P \models \theta P_1}{P \models \forall \bar{\mu}\bar{\alpha}.\, P_1 \Rightarrow \tau \,\geqslant\, \theta\tau}\ \text{(TS-Inst)}$$

$$\frac{P \,|\, \Gamma, x{:}\tau_1 \vdash e : m\ \tau_2}{P \,|\, \Gamma \vdash \lambda x.e : \tau_1 \to m\ \tau_2}\ \text{(TS-Lam)} \qquad \frac{\begin{array}{cc} P \,|\, \Gamma \vdash e_1 : m_1\ (\tau_2 \to m_3\ \tau) & P \,|\, \Gamma \vdash e_2 : m_2\ \tau_2 \\ P \models (m_1, m_4) \rhd m_5 & P \models (m_2, m_3) \rhd m_4 \end{array}}{P \,|\, \Gamma \vdash e_1\ e_2 : m_5\ \tau}\ \text{(TS-App)}$$

$$\frac{P \,|\, \Gamma \vdash v : \tau}{P \,|\, \Gamma \vdash v : \mathit{Id}\ \tau}\ \text{(TS-Id)} \qquad \frac{\begin{array}{cc} P \,|\, \Gamma \vdash e_1 : m_1\ \tau_1 & P \,|\, \Gamma, x{:}\tau_1 \vdash e_2 : m_2\ \tau_2 \\ e_1 \neq v & P \models (m_1, m_2) \rhd m_3 \end{array}}{P \,|\, \Gamma \vdash \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : m_3\ \tau_2}\ \text{(TS-Do)} \qquad \frac{\begin{array}{c} P' \,|\, \Gamma \vdash v : \tau' \\ \sigma = \mathit{Gen}(\Gamma, P' \Rightarrow \tau') \\ P \,|\, \Gamma, x{:}\sigma \vdash e : m\ \tau \end{array}}{P \,|\, \Gamma \vdash \mathsf{let}\ x{=}v\ \mathsf{in}\ e : m\ \tau}\ \text{(TS-Let)}$$

Figure 4. Syntax-directed type rules for λPM. The generalization function is defined as: $\mathit{Gen}(\Gamma, \sigma) = \forall(\mathsf{ftv}(\sigma) \setminus \mathsf{ftv}(\Gamma)).\sigma$

constraints. Unfortunately, doing such a proof directly on the evidence translation for the syntax directed system is difficult: since the rules allow arbitrary instantiations for monadic constraints it is hard to relate the resulting evidence terms semantically. Instead we define a more algorithmic version of the type rules that let us reason about coherence in a more structural and syntactic way.

The remainder of this section presents our proof. First we present a more algorithmic treatment of type inference that shows how source terms are elaborated to target terms containing binds. Next, we define two additional properties needed for coherence, and argue why they are easy to satisfy. We conclude with the presentation of the actual proof.

5.1 Type inference with elaboration

Figure 6 defines a more algorithmic version of the syntax directed type rules of Figure 4. The inference rules are $P \mid \Gamma \vdash e : \tau \rightsquigarrow \mathsf{e}$ and $P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$ where the constraints P, type τ (or $m\ \tau$) and target term e are synthesized. This definition combines a syntax-directed presentation of Hindley-Milner type inference, where the details of unification are elided, with an algorithmic presentation of polymonadic constraint generation and solving.

In particular, in rule (TA-Var) and (TA-Const), the substitutions for the type variables are still chosen arbitrarily as in any standard presentation of Hindley-Milner style type rules. However, the monadic variables are explicitly substituted with fresh monadic variables where the constraints of the type scheme are returned in the predicates P. Similarly, in rule (TA-App) and (TA-Do), the intermediate monadic types are now represented by fresh monadic type variables. Effectively, this ensures that the evidence for each bind is always inserted and solved at specific syntactical locations which is essential to doing the coherence proof. Finally, (TA-Let) performs generalization. It uses a relation $P \xrightarrow{\text{simplify}(\bar{\mu})} P'; \theta$ to optionally simplify constraints P by using substitution θ to solve (some) monad variables $\bar{\mu}$ with any remaining, unsolved constraints in P'. Solved variables are no longer generalized. We present our particular simplification algorithm in Section 6.

We have not done a full proof yet, but we conjecture that the algorithmic rules of Figure 6 are sound and complete with respect to the syntax directed rules of Figure 4.

The following lemma establishes that our algorithm produces well-typed System F terms. By $\{\![\Gamma]\!\}$ we mean the (straightforward) translation of a source typing context to a System F context.

Lemma 3 (Well-typed elaborations)**.** *Given $e, \tau, m, \mathsf{e}, \Gamma$ such that if either $P \mid \Gamma \vdash e : \tau \rightsquigarrow \mathsf{e}$ or $P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$ then there exists t such that $\{\![\Gamma]\!\} \vdash_F \mathit{abs}(P, \mathsf{e}) : \mathsf{t}$.*

Proof. See Appendix B. □

Once we prove $P \,|\, \Gamma \vdash e : m\ \tau \rightsquigarrow \mathsf{e}$ and instantiate the final type $m\,\tau$, we must solve the constraints P to produce bind terms $\bar{\mathsf{b}}$ that we use to execute e. We can do this using the given polymonadic entailment relation $\Sigma \models \theta P \rightsquigarrow \bar{\mathsf{b}}$ based on Σ's theory Φ. Then we simply execute $\mathsf{e}\ \bar{\mathsf{b}}$. We discuss our implementation of solving in Section 7.

5.2 Additional properties

With the evidence translation in place, we can now precisely state what we mean by coherence: suppose we have $P \,|\, \Gamma \vdash e : m\ \tau \rightsquigarrow \mathsf{e}$. Then if θ and θ' are both solutions to the constraints P, where a solution is a mapping of constraints π to particular binds $\mathsf{b}_{m_1,m_2,m_3} \in \Sigma$, then both solutions, applied to e, will yield terms with an identical operational effect. Our proof of coherence works by starting with θ and showing how we can iteratively transform it into θ', all the while proving that each intermediate step does not affect the semantics of the rewritten term. For this proof to work we rely on polymonads satisfying two additional properties.

First, the semantics of a particular bind in Σ must be independent of any type indexes in m; i.e., type indices are *operationally irrelevant*. That is, as mentioned in Section 3.1, for computations $M\ \bar{\tau}$ the type indices $\bar{\tau}$ are meant to be phantom, and thus not influence a bind's semantics. To express this idea formally, let $(\!|\mathsf{e}|\!)$ represent the type *erasure* of a System F term, so that all type annotations, type abstractions, and type applications are dropped. Then we expect polymonads to satisfy the following property

Param: For all $M_1, M_2, M_3, \bar{\tau}_1, \bar{\tau}_2, \bar{\tau}_3, \bar{\tau}'_1, \bar{\tau}'_2, \bar{\tau}'_3$,

$$(\!|\mathsf{bind}_{M_1\ \bar{\tau}_1, M_2\ \bar{\tau}_2, M_3\ \bar{\tau}_3}\ e\ k|\!) \cong (\!|\mathsf{bind}_{M_1\ \bar{\tau}'_1, M_2\ \bar{\tau}'_2, M_3\ \bar{\tau}'_3}\ e\ k|\!)$$

Here, we write $\cong$ to mean β/η equality for the untyped lambda calculus (which should match the semantics of the original System F terms). The (Param) property is useful for establishing that intermediate solutions will not change a term's semantics when only the type indices are changing. All of our examples from Section 3 satisfy this property.

Second, we require that the polymonad be *smooth* in that for each pair of constructors $M_1, M_2 \in \mathcal{M}$, we can compute $M = \mathit{smooth}(M_1, M_2)$ where there exist binds involving M for those binds involving M_1 and/or M_2:

Definition 4 (Smooth polymonad)**.** *A* smooth *polymonad $\Sigma_{\mathcal{M}}$ is one for which we can define a function smooth on polymonad constructors $M \in \mathcal{M}$, written $\mathit{smooth}(M, M') = M^\star$. This function has the following properties.*

$$\boxed{P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e} \qquad P \mid \Gamma \vdash v : \tau \rightsquigarrow \mathsf{v}}$$

$$\frac{\Gamma(x) = \forall\bar{\mu},\bar{\alpha}.P \Rightarrow \tau \quad \theta = [\bar{\mu}'/\bar{\mu}][\bar{\tau}/\bar{\alpha}] \quad \bar{\mu}' \text{ fresh}}{\theta P \mid \Gamma \vdash x : \theta\tau \rightsquigarrow \mathsf{app}(x, \theta P)}\ \text{(TA-Var)} \qquad \frac{\Gamma(c) = \forall\bar{\mu},\bar{\alpha}.P \Rightarrow \tau \quad \theta = [\bar{\mu}'/\bar{\mu}][\bar{\tau}/\bar{\alpha}] \quad \bar{\mu}' \text{ fresh}}{\theta P \mid \Gamma \vdash c : \theta\tau \rightsquigarrow \mathsf{app}(c, \theta P)}\ \text{(TA-Const)}$$

$$\frac{P \mid \Gamma, x{:}\tau_1 \vdash e : m\,\tau_2 \rightsquigarrow \mathsf{e}}{P \mid \Gamma \vdash \lambda x.e : \tau_1 \rightarrow m\,\tau_2 \rightsquigarrow \lambda x{:}\tau_1.\mathsf{e}}\ \text{(TA-Lam)} \qquad \frac{\begin{array}{c} P_1 \mid \Gamma \vdash e_1 : m_1\,(\tau_2 \rightarrow m_3\,\tau) \rightsquigarrow \mathsf{e}_1 \quad P_2 \mid \Gamma \vdash e_2 : m_2\,\tau_2 \rightsquigarrow \mathsf{e}_2 \\ \mu_4, \mu_5 \text{ fresh} \quad P = P_1, P_2, (m_1, \mu_4) \rhd \mu_5, (m_2, m_3) \rhd \mu_4 \end{array}}{P \mid \Gamma \vdash e_1\ e_2 : \mu_5\,\tau \rightsquigarrow \mathsf{b}_{m_1,\mu_4,\mu_5}\ \mathsf{e}_1\ (\lambda x{:}\_.\ \mathsf{b}_{m_2,m_3,\mu_5}\ \mathsf{e}_2\ x)}\ \text{(TA-App)}$$

$$\frac{P \mid \Gamma \vdash v : \tau \rightsquigarrow \mathsf{v}}{P \mid \Gamma \vdash v : Id\,\tau \rightsquigarrow (\mathsf{unit}_{Id}\ \mathsf{v})}\ \text{(TA-Id)} \qquad \frac{\begin{array}{c} e_1 \neq v \quad P_1 \mid \Gamma \vdash e_1 : m_1\,\tau_1 \rightsquigarrow \mathsf{e}_1 \quad P_2 \mid \Gamma, x{:}\tau_1 \vdash e_2 : m_2\,\tau_2 \rightsquigarrow \mathsf{e}_2 \\ \mu_3 \text{ fresh} \quad P = P_1, P_2, (m_1, m_2) \rhd \mu_3 \end{array}}{P \mid \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \mu_3\,\tau_2 \rightsquigarrow \mathsf{b}_{m_1,m_2,\mu_3}\ \mathsf{e}_1\ (\lambda x{:}\_.\ \mathsf{e}_2)}\ \text{(TA-Do)}$$

$$\frac{\begin{array}{c} P' \mid \Gamma \vdash v : \tau' \rightsquigarrow \mathsf{v} \quad \bar{\mu},\bar{\alpha} = \bar{\nu} = \mathsf{ftv}(P' \Rightarrow \tau') \setminus \mathsf{ftv}(\Gamma) \\ P' \xrightarrow{\text{simplify}(\bar{\mu}\setminus\mathsf{ftv}(\tau'))} P''; \theta \quad \bar{\nu}' = \bar{\nu} \setminus dom(\theta) \\ \sigma = \forall\bar{\nu}'.P'' \Rightarrow \tau' \quad P \mid \Gamma, x{:}\sigma \vdash e : m\,\tau \rightsquigarrow \mathsf{e} \end{array}}{P \mid \Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : m\,\tau \rightsquigarrow (\lambda x{:}\_.\ \mathsf{e})\ \mathsf{abs}(P'', \theta\mathsf{v})}\ \text{(TA-Let)}$$

where

$$\begin{array}{lcl} \mathsf{app}(\mathsf{e}, (P, (m_1, m_2) \rhd m_3)) & = & \mathsf{app}(\mathsf{e}, P)\ \mathsf{b}_{m_1,m_2,m_3} \\ \mathsf{app}(\mathsf{e}, \cdot) & = & \mathsf{e} \\ \mathsf{abs}(((m_1, m_2) \rhd m, P), \mathsf{e}) & = & \lambda\mathsf{b}_{m_1,m_2,m}{:}\_.\ \mathsf{abs}(P, \mathsf{e}) \\ \mathsf{abs}(\cdot, \mathsf{e}) & = & \mathsf{e} \end{array}$$

Figure 6. Algorithmic syntax-directed type rules with evidence translation

1. *Given* $M_1^\star = smooth(M_1, M_1')$ *and* $M_2^\star = smooth(M_2, M_2')$.
   if $\Sigma \models (M_1\ \_, M_2\ \_) \rhd M_3\ \_$ *and*
   $\Sigma \models (M_1'\ \_, M_2'\ \_) \rhd M_3\ \_$
   then there exist type indices $\bar{\tau}_1, \bar{\tau}_2, \bar{\tau}_3, \bar{\tau}_1', \bar{\tau}_2', \bar{\tau}_3'$ *such that*
   $\Sigma \models (M_1^\star\ \bar{\tau}_1, M_2^\star\ \bar{\tau}_2) \rhd M_3\ \bar{\tau}_3$ *and*
   $\Sigma \models (M_1^\star\ \bar{\tau}_1', M_2^\star\ \bar{\tau}_2') \rhd M_3'\ \bar{\tau}_3'$
2. *Given* $M^\star = smooth(M, M')$.
   If $\Sigma \models (M_1\ \_, M_2\ \_) \rhd M\ \_$ *or*
   $\Sigma \models (M_1\ \_, M_2\ \_) \rhd M'\ \_$
   then there exist type indices $\bar{\tau}_1, \bar{\tau}_2, \bar{\tau}'$ *such that*
   $\Sigma \models (M_1\ \bar{\tau}_1, M_2\ \bar{\tau}_2) \rhd M^\star\ \bar{\tau}'$.
3. $smooth(M, M) = M$.

Intuitively, $smooth(M_1, M_2)$ is like the least upper bound of M_1 and M_2. This property is useful for generating small modifications to a solution θ to bring it closer, in a coherent fashion, to solution θ'.

We believe that requiring $smooth$ places little practical limitation on polymonads. In particular, when writing programs with just one polymonadic constructor M, the function is trivial: $smooth(M, M) = M$, $smooth(Id, M) = M$, $smooth(M, Id) = M$, and $smooth(Id, Id) = Id$. When using multiple constructors M_1 and M_2, we must already define how computations interact. For example, suppose we were programming with sessions and state, with constructors $Sess/2$ and $\mathsf{ST}/0$. We would probably define a third constructor $SessST/2$ that represents a communicating computation that also modifies the heap, and we would define a bind for lifting $Sess\,\alpha\,\beta\,\tau$ computations into $SessST\,\alpha\,\beta\,\tau$ computations, and one for lifting $\mathsf{ST}\,\tau$ computations into $SessST\,\alpha\,\beta\,\tau$ computations. By the (Param) property, these liftings will not depend on type indices, and so they are morally monad-style morphisms $Sess \succ SessST$ and $\mathsf{ST} \succ SessST$. Such morphisms, when arranged to form a lattice, easily satisfy the requirements of $smooth$.

5.3 Proof of coherence

Now we present the final formal details of the proof of coherence. First, some notation: We view constraints P as a directed graph. Nodes are polymonads m, i.e., either monad type constants $M\,\bar{\tau}$

$$\begin{array}{lcl} \textit{up-bnd}((\cdot,\cdot) \rhd m) & = & m \\ \textit{up-bnds}(P) & = & \bigcup_{\pi \in P}\{\textit{up-bnd}(\pi)\} \\ & & \\ \textit{lo-bnd}((m_1, m_2) \rhd \cdot) & = & (m_1, m_2) \\ \textit{lo-bnds}(P) & = & \bigcup_{\pi \in P}\{\textit{lo-bnd}(\pi)\} \\ & & \\ \textit{flowsTo}_P\ \mu & = & \{\pi \mid \pi \in P\ \wedge\ \mu = \textit{up-bnd}(\pi)\} \\ \textit{flowsFrom}_P\ \mu & = & \{\pi \mid \pi \in P\ \wedge \\ & & \quad \exists m'.\ \textit{lo-bnd}(\pi) = (\mu, m') \\ & & \quad \vee\ \textit{lo-bnd}(\pi) = (m',\ mu)\} \end{array}$$

Figure 7. Constraints P as graphs

(where the $\bar{\tau}$ could contain type variables), or monad variables μ. Each bind constraint $(m_1, m_2) \rhd m$ induces two edges, a *left edge* $m_1 \longrightarrow m$ and a *right edge* $m_2 \longrightarrow m$. We say the *upper bound* of such a constraint is m, and the lower bound is the pair of monads (m_1, m_2); we can think of the former as the constraint's *output* and the latter as its *input*. These notions are defined formally in Figure 7, along with their obvious liftings to constraint sets P. The figure also defines *flowsTo*$_P$ μ—the set of constraints in P that have μ as an upper bound—and *flowsFrom*$_P$ μ—the set of constraints that have μ as a lower bound.

For a given constructor $M/k \in \mathcal{M}$ not all instantiations of M's type indices may be legal in a given bind. For example, for session types we could not legally define a bind with type $(A\,\phi\,\gamma,\ \ A\,\phi\,\psi) \rhd A\,\phi\,\psi$ if ϕ and γ were different. However, because of (Param), we view the semantics of any bind as due to the default "operational" bind for every triple of polymonad constructors, irrespective of their parameters.

For the sole purpose of coherence proof, which is concerned only with run-time semantics, it is safe to assume the existence of more binds, which only differ in polymonad parameters from the existing binds. We thus *saturate* the initial signature with more binds, provided that they never appear in actual solutions but only used to establish congruence of the actual solutions by transitivity. Saturation is defined formally as follows:

Definition 5 (Saturated signature)**.** *Given a bindset Σ, its saturated signature $\Sigma_{sat} = \langle\!| \Sigma |\!\rangle$, is defined as follows:*

$$\begin{array}{lcl} \langle\!| \mathsf{b}_{Id,Id,Id} |\!\rangle & = & \mathsf{b}_{Id,Id,Id} \\ \langle\!| \mathsf{b} : \forall\bar\alpha.\Phi \Rightarrow (M_1\,\bar\tau_1, M_2\,\bar\tau_2) \rhd M_3\,\bar\tau_3, \Sigma |\!\rangle & = & \\ \quad \mathsf{b} : \forall\bar\beta_1\bar\beta_2\bar\beta_3.(M_1\,\bar\beta_1, M_2\,\bar\beta_2) \rhd M_3\,\bar\beta_3, \langle\!| \Sigma |\!\rangle & & \end{array}$$

Finally, following two more definitions, here is our general coherence result for polymonads.

Definition 6 (Well-formedness of a signature)**.** *A well-formed signature Σ is one that satisfies the polymonad laws, the (Param) property, and is smooth.*

Definition 7 (Ground solution)**.** *A solution θ to constraints P is* ground *for Σ if and only if co-domain(θ) contains only ground polymonads M or ground types τ, and $\Sigma \models \theta\pi$ for all $\pi \in P$.*

Theorem 8 (Coherence)**.**
Given $\Sigma, \Gamma, P, e, m, \tau, \theta_1, \theta_2, \mathsf{e}, \bar\mu, \bar\alpha$, such that

1. *Σ is well-formed and P is cycle-free.*
2. *$P \mid \Gamma \vdash e : \tau \rightsquigarrow \mathsf{e}$ or $P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$.*
3. *There exist open variables $\bar\mu, \bar\alpha = ftv(P) \backslash ftv(\tau)$ (or $\bar\mu, \bar\alpha = ftv(P) \backslash ftv(m\,\tau)$) such that for all $\mu \in \bar\mu$, the sets flowsTo$_P$ μ and flowsFrom$_P$ μ are non-empty, i.e., each $\mu \in \bar\mu$ has a lower and an upper bound.*
4. *θ_1 and θ_2 are ground solutions for P such that $\theta_1(\nu) = \theta_2(\nu)$ for all $\nu \notin \bar\alpha, \bar\mu$.*

Then, $\theta_1\mathsf{e} = \theta_2\mathsf{e}$.

Proof. The full proof is shown in Appendix C. Because θ_1 and θ_2 are ground solutions for P under Σ, they are also ground under the saturated signature Σ_{sat}. The proof first constructs a solution θ_3 by combining monad variable substitutions from θ_2 and type variable substitutions from θ_1; this is a ground solution under Σ_{sat}. By repeated appeal to the (Param) property for every subterm that differs between $\theta_2\mathsf{e}$ and $\theta_3\mathsf{e}$ we get that $\theta_2\mathsf{e} = \theta_3\mathsf{e}$.

The rest of the proof proceeds by iterating over the constraints solved differently by θ_1 and θ_3. Because P is cycle-free we can consider each $\pi \in P$ in reverse topological order. We maintain an invariant that each π considered has a (ground) upper bound. At each step we construct two new ground solutions θ_1' and θ_3' that only differ from θ_1 and θ_3 in the substitution of the lower bounds for the current constraint π. We assign by $smooth(\textit{lo-bnds}(\theta_1\pi), \textit{lo-bnds}(\theta_3\pi))$ to these bounds in both θ_1' and θ_3'. We prove these are ground solutions due to the properties of $smooth$, and they must have the same semantics as the terms with θ_1 and θ_3 applied, which follows from a corollary of the polymonadic associativity property. We set $\theta_1 = \theta_1'$ and $\theta_3 = \theta_3'$ and continue. At the last step the solutions are exactly the same, so $\theta_1\mathsf{e} = \theta_3\mathsf{e}$ follows. □

The statement of Theorem 8 requires that there are no cycles in P. It is easy to show the type inference system presented in Figure 6 satisfies this requirement, provided that no type scheme constraints in the typing environment have cycles in them. However cycles would arise if we extended our language to support recursion. Our preliminary investigations suggest we can deal with cycles by annotating recursive definitions with ground types (thus ensuring the upper bound requirement of the proof). Reasoning about coherence of programs with arbitrary recursion is future work.

6. Simplification

In this section we present our simplification algorithm which allows to simplify types prior to generalizing them in (TA-Let) (Figure 6)

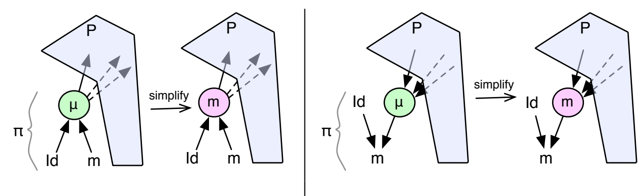
$$\frac{}{P \xrightarrow{\mathsf{simplify}(\bar\mu)} P; \cdot}$$

$$\frac{\pi, P \xrightarrow{\mathsf{simplify}(\bar\mu)} P'; \theta}{\pi, \pi, P \xrightarrow{\mathsf{simplify}(\bar\mu)} P'; \theta} \qquad \frac{P_2, P_1 \xrightarrow{\mathsf{simplify}(\bar\mu)} P'; \theta}{P_1, P_2 \xrightarrow{\mathsf{simplify}(\bar\mu)} P'; \theta}$$

$$\text{S-}\Uparrow\ \frac{\begin{array}{cc} \textit{up-bnd}(\pi) = \mu & \bar\mu = \mu, \bar\mu' \\ \multicolumn{2}{c}{\textit{lo-bnd}(\pi) = (Id, m) \vee \textit{lo-bnd}(\pi) = (m, Id)} \\ \textit{flowsFrom}_P\ \mu \neq \{\} & \textit{flowsTo}_P\ \mu = \{\} \\ \theta = (\mu \mapsto m) & \theta P \xrightarrow{\mathsf{simplify}(\bar\mu')} P'; \theta' \end{array}}{\pi, P \xrightarrow{\mathsf{simplify}(\bar\mu)} \theta\pi, P'; \theta, \theta'}$$

$$\text{S-}\Downarrow\ \frac{\begin{array}{cc} \multicolumn{2}{c}{\bar\mu = \mu, \bar\mu'} \\ \multicolumn{2}{c}{\textit{lo-bnd}(\pi) = (Id, \mu) \vee \textit{lo-bnd}(\pi) = (\mu, Id)} \\ \textit{flowsFrom}_P\ \mu = \{\} & \textit{flowsTo}_P\ \mu \neq \{\} \\ \theta = (\mu \mapsto \textit{up-bnd}(\pi)) & \theta P \xrightarrow{\mathsf{simplify}(\bar\mu')} P'; \theta' \end{array}}{\pi, P \xrightarrow{\mathsf{simplify}(\bar\mu)} \theta\pi, P'; \theta, \theta'}$$

Figure 8. Constraint simplification



(a) Rule S-⇑ (b) Rule S-⇓

Figure 9. Visual depiction of simplification (Fig. 8)

while eliminating open variables. Simplification makes types easier to read while not reducing their generality.

Figure 8 presents inference rules for the judgment $P \xrightarrow{\mathsf{simplify}(\bar\mu)} P'; \theta$, which states that constraints P can be simplified to constraints P' (of the same cardinality) according to substitution θ which has a domain that is subset of monad variables $\bar\mu$. When referenced in (TA-Let), $\bar\mu$ is the list of open variables which appear in constraints P but not the let-bound term's final type. The goal here is to eliminate as many of these variables as possible.

The first rule is the identity rule, performing no solving. The second rule drops duplicate constraints. The third rule permits constraints to be reordered. The last two perform the real work.

Rule S-⇑ solves monad variable μ with monad m for the situation depicted in Figure 9(a). Here, we have a single constraint π whose upper bound is an open variable μ, and whose lower bounds are some monad m and Id; it has no other lower bounds in P. If μ also has an upper bound in P then we may substitute μ with its lower bound. This rule is justified by appealing to the polymonad laws. First, the substitution will convert $(Id, m) \rhd \mu$ into $(Id, m) \rhd m$ but this creates no additional burden on typing since this identity morphism is always guaranteed to exist by the Functorial Law and the Paired Morphisms law. Second, the substitution will convert all constraints $(\mu, m_1) \rhd m_2$ in P to $(m, m_1) \rhd m_2$. Any context that could have satisfied the original constraints can also satisfy these new constraints by the composition closure law: since $m \succ \mu$ and $(\mu, m_1) \rhd m_2$ then Σ must also contain $(m, m_1) \rhd m_2$. Note that this reasoning, and the rule, ap-

$$\text{S-Const}\ \frac{\Sigma \models \cdot P \qquad P' \xrightarrow{\text{simplify}(\bar{\mu})} P''; \theta'}{P, P' \xrightarrow{\text{simplify}(\bar{\mu})} P''; \theta, \theta'}$$

$$\text{S-Meta}\ \frac{\pi = (m, Id) \rhd m \ \vee \pi = (Id, m) \rhd m \qquad P \xrightarrow{\text{simplify}(\bar{\mu})} P'; \theta}{\pi, P \xrightarrow{\text{simplify}(\bar{\mu})} P'; \theta}$$

Figure 10. Constraint hiding

plies whether we start with $\pi = (Id, m) \rhd \mu$ or $\pi = (m, Id) \rhd \mu$ (we have depicted only the first of these).

Rule S-$\Downarrow$ follows similar reasoning, but in the reverse direction. The solution $\mu \mapsto m$ again produces an identity morphism, which always exists, and alters existing constraints $(m_1, m_2) \rhd \mu$ to be $(m_1, m_2) \rhd m$. The latter constraints are again justified by composition closure; i.e., since $\mu \succ m$ and $(m_1, m_2) \rhd \mu$ then Σ must also contain $(m_1, m_2) \rhd m$.

Example. Recall the session types example we gave in Section 3.2. We apply (S-$\Uparrow$) several times to simplify the principal type to the more readable one. For example, we apply it to the sixth constraint $(Id, \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma) \rhd \mu_6)$ and induce the substitution $\mu_6 \mapsto \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma)$ which is applied to the seventh constraint, making it eligible for simplification with (S-$\Uparrow$) too (substituting for μ_5). The process continues to the second constraint and the fourth, and then stops after substituting $\mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma)$ for μ_8 in the first constraint. We can also apply (S-$\Uparrow$) to the third constraint, which has a different session type, which then propagates to the fifth constraint and the first. At this point we have the following set of constraints:

$$
\begin{array}{l}
(\mathit{Sess}\,(\mathit{send}\ \ \alpha\,\beta)\alpha, \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma) \rhd \mu_{10},\\
(\mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma), Id) \rhd \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma),\\
(Id, \mathit{Sess}\,(\mathit{send}\ \ \alpha\,\beta)\alpha) \rhd \mathit{Sess}\,(\mathit{send}\ \ \alpha\,\beta)\alpha,\\
(Id, \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma) \rhd \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma)),\\
(Id, \mathit{Sess}\,(\mathit{send}\ \ \alpha\,\beta)\alpha) \rhd \mathit{Sess}\,(\mathit{send}\ \ \alpha\,\beta)\alpha),\\
(Id, \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma) \rhd \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma)),\\
(Id, \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma)) \rhd \mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma)),
\end{array}
$$

Three of these constraints are duplicates so we can drop them, leaving us with the simpler type given in Section 3.2. We can further improve this type, as we show shortly.

Pleasingly, this process yields a simpler type that can be used in the same contexts as the original principal type, so we are not compromising the generality of the code by simplifying its type.

Lemma 9 (Simplification improves types)**.**
Given σ and σ' where σ is $\forall\bar{\nu}.P \Rightarrow \tau$ and σ' is an improvement *of σ', having form $\forall\bar{\nu}'.P' \Rightarrow \tau$ where $P \xrightarrow{\text{simplify}(\bar{\mu})} P'; \theta$ and $\bar{\nu}' = \bar{\nu} - dom(\theta)$. Then for all $P'', \Gamma, x, e, m, \tau$, if $P'' \mid \Gamma, x : \sigma \vdash e : m\,\tau$ such that $\Sigma \models P''$ then there exists some P''' such that $P''' \mid \Gamma, x : \sigma' \vdash e : m\,\tau$ and $\Sigma \models P'''$.*

Proof. The proof is by induction on the derivation $P'' \mid \Gamma, x : \sigma \vdash e\ :\ m\,\tau$. Most cases are by assumption or induction, with the interesting one being (TA-Var) where the variable in question is x, and we know that all of the constraints are solvable according to the reasoning we used to justify the simplifications, above. □

6.1 Hiding variables and constraints

While constraint simplification is effective, it can leave behind uninteresting constraints. Along with the first and third rules from Figure 8, we can apply the rules in Figure 10 to *hide* uninteresting constraints (along with their type variables, if they only appear in these constraints). Rule (S-Const) allows us to drop constraints that refer only to constant binds, i.e., those proved with an empty substitution. The intuition is that constant binds could just be inlined in the function that requires them, so they do not communicate any useful information. Rule (S-Meta) drops constraints of the form $(m, Id) \rhd m$ and $(Id, m) \rhd m$ because, by the Functorial and Paired Morphism laws, they must exist for all polymonads m. The constraints are thus not communicating useful information.

Applying these rules to our session types example, we can drop all but the first constraint resulting in a far simpler final type.

$$
\begin{array}{l}
\forall\alpha,\beta,\gamma,\mu_{10}.\\
\quad (\mathit{Sess}\,(\mathit{send}\ \ \alpha\,\beta)\alpha), (\mathit{Sess}\,(\mathit{recv}\ \ \gamma\ \mathit{int})\,\gamma)) \rhd \mu_{10} \quad \Rightarrow\\
(\beta \to \mu_{10}\ \mathit{int})
\end{array}
$$

Lemma 9 still holds even when adding in these rules. The structure of the proof is unchanged, and the (TA-Var) case will be augmented to justify the additional simplifications. Note that if we were to include these rules in Simplification directly, the *elaborated* terms would no longer be type correct (we would break Lemma 3) since placeholders for binds inserted in the elaborated term v in the (TA-Let) rule would no longer be abstracted. Therefore we simply imagine these extra constraints being hidden by the IDE.

7. Implementation

We have implemented our type inference algorithm for the simple language given in Figure 3. Our prototype implementation uses the empty theory Φ, and thus constraints must be encoded by enumeration. For example, for the IST polymonad from Section 3.3 we define several binds instead of the single bIST. In addition to IST, we have implemented all of the monad examples from our prior paper [24], the session types example from Section 3.2 and an example of Danielsson's [6] computational complexity monad. We would require non-empty theories for the remaining examples.

Because we have no separate theory, we implement entailment $\Sigma \models \theta\pi \rightsquigarrow \bar{\mathsf{b}}$ by unifying π with some $\mathsf{b}{:}s \in \Sigma$ but with its quantified type variables replaced with fresh (unification) variables. Since there may be many $\mathsf{b}{:}s$ against which we can unify π (producing substitution θ), we may need to consider each possibility. Thus, to solve a set of constraints $P = \pi, P'$, we consider each possible substitution θ for π, then attempt to solve the constraints $\theta P'$, composing the result with θ until we find one that works (or fail).

The worst-case complexity of the algorithm is

$$O(\sum_{k=1}^{|P|}(|\Sigma|^k))$$

The number of substitutions considered at each step is $|\Sigma|$ when all bind types $\mathsf{b}{:}s$ can be unified with π. (This will happen, for example, when $\pi = (\mu_1, \mu_2) \rhd \mu_3$.) Thus the performance of the algorithm crucially depends on the number of the monad variables in π, so the order in which the constraints is handled is important. Our prototype employs topological sort for cycle-free subsets of constraints. In this case both lower bounds of a constraint are ground and we only have to solve its upper bound.

When polymonads form a join lattice (as was true of monads in our prior work [24]) we can use a simpler algorithm that performs linearly in the number of constraints, in the absence of cycles in the constraint graph. Because lubs exist for any set of polymonad instances, instead of computing all sound solutions θ we can solve every monad variable to the lub of the lower bounds of constraints that flow to this variable.

8. Related work

A variety of past work has aimed to refine the conventional notion of monads. Several examples, including Atkey's parameterized monads [2], Wadler and Thiemann's indexed monads [27], and applications thereof, were cited in the introduction and given in Section 3. Each of these constructions can be viewed as an instance of a polymonad. Filliâtre [10] proposed *generalized monads* as a means to more carefully reason about effects in a monadic style, and his work bears a close resemblance to Wadler and Thiemann's. Generalized monads can also be seen as instances of polymonads—it is easy to show that the polymonad laws imply Filliâtre's six required identities. Conversely, it is clear that some useful examples cannot be expressed using any of these prior refinements to monads; for example, our IST polymonad cannot be expressed due to its exclusion of certain (information-flow-violating) compositions. Thus polymonads provide greater expressive power.

Kmett's `Control.Monad.Parameterized` Haskell package [16] provides a typeclass for non-uniform binds, with the goal of generalizing monadic programming. One key limitation is that Kmett's bind $(m_1, m_2) \rhd m_3$ must be *functionally dependent*; i.e., m_3 must be a function of m_1 and m_2. As such, it is not possible to program morphisms between different monadic constructors, i.e., the pair of binds $(m_1, Id) \rhd m_2$ and $(m_1, Id) \rhd m_3$ would be forbidden, so there would be no way to convert from m_1 to m_2 and from m_1 to m_3 in the same program. Polymonads do not have this limitation. Kmett does not discuss laws that should govern the proper use of non-uniform binds.

Another line of past work has focused on making monadic programming easier. Haskell's `do` notation exposes the structure of a monadic computation, and typeclass inference can determine which binds and units should be used, but the placement of morphisms is left to the programmer. The problem is that the use of morphisms (e.g., if defined as a typeclass) would frequently lead to open type variables, which Haskell's typeclass inference deems ambiguous. Inference with Kmett's class has the same problems.

For ML, our own prior work [24] showed that no additional notation is needed: left-to-right, call-by-value evaluation order makes the order of operations well-defined, so the syntactic structure of the program indicates where binds, units, and even morphisms should be placed. Moreover, we proved that the monad laws ensured that open variables could solved arbitrarily without affecting semantics, and so there was no need to reject programs with open types. Our present paper generalizes the approach of this prior paper in two ways, first by applying it to the more general polymonadic construction, and second by showing more rigorously how to reason about two arbitrary, different solutions. Interestingly, the present work arose when we discovered we could not write IST as a monad, since a monad would require the existence of binds that could violate an information flow property.

As mentioned in the introduction and Section 2, concurrently with our work Tate developed a general semantic framework called *productors* for describing *producer effect* systems [25]. This framework turns out to match our definition of polymonads in many important cases. Our work is complementary to Tate's in that we consider practical programming concerns (i.e., type inference and rewriting) in a higher-order functional programming language, whereas he focuses on semantic foundations.

9. Conclusions

We have presented *polymonads*, a generalization of prior monad-like programming idioms. We have shown polymonads to be useful, using them to encode a variety of prior programming constructions. We have also shown how to facilitate programming with them in a direct style—the programmer can use a polymonadic computation $m\ \tau$ as if it were of type τ and our novel type inference and rewriting algorithm will insert the necessary coercions. Rewritten programs are *coherent*: all solutions to variables not present in the final type will induce the same semantics. Pleasingly, our algorithm produces general types that are nevertheless simple.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL*, volume 26, pages 147–160, 1999.

[2] R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009.

[3] R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3 & 4):335–376, 2009.

[4] G. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, 2006.

[5] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of functional programming*, 15(02):249–291, 2005.

[6] N. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL*, pages 133–144, 2008.

[7] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *TLDI*, pages 59–72, 2011.

[8] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[9] A. Filinski. Representing layered monads. In *POPL*, pages 175–188, 1999.

[10] J.-C. Filliâtre. A theory of monads parameterized by effects, 1999.

[11] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VI I, 1972.

[12] J. Goguen and J. Meseguer. Security policy and security models. In *Symposium on Security and Privacy*, pages 11–20, 1982.

[13] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *JFP*, 8(4), 1998.

[14] M. P. Jones. A theory of qualified types. In *ESOP*, 1992.

[15] O. Kiselyov and C. Shan. Lightweight monadic regions. In *ACM SIGPLAN Notices*, volume 44, pages 1–12. ACM, 2008.

[16] E. Kmett. `Control.Monad.Parameterized` package. `http://hackage.haskell.org/packages/archive/monad-param/0.0.4/doc/html/Control-Monad-Parameterized.html`, 2012.

[17] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *CSFW*, pages 16–27, 2006.

[18] E. Moggi. Computational lambda-calculus and monads. In *LICS*, 1989.

[19] Monad laws. `http://www.haskell.org/haskellwiki/Monad_Laws`, 2012.

[20] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, 2008.

[21] R. Pucella and J. Tov. Haskell session types with (almost) no class. In *ACM SIGPLAN Notices*, volume 44, pages 25–36. ACM, 2008.

[22] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

[23] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Haskell*, 2008.

[24] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP*, 2011.

[25] R. Tate. The sequential semantics of producer effect systems, 2012.

[26] P. Wadler. The essence of functional programming. In *POPL*, 1992.

[27] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, January 2003.

A. Proof of Lemma 1

We separately prove the two directions of the implication. First we show that for all S such that $\langle S \rangle = \mathcal{M}, \Sigma_{\mathcal{M}}$, if $\models S$ then $\models \langle S \rangle$. (Note that $\mathsf{bind}_{Id,Id,Id} \in \Sigma_{\mathcal{M}}$ as required since $S \models Id \rhd Id$.)

Proof. Let S be a monadic signature. Suppose that $\models S$. To show $\models \Sigma_{\mathcal{M}}$ we prove the six polymonad laws hold for all the binds defined by $\Sigma_{\mathcal{M}} = \mathit{Clos}(S)$.

Functorial law: For all $M \in \mathcal{M}$, we have that $S \models Id \rhd M$ and $S \models M \rhd M$ so by $\mathit{Clos}(S)$ we must have that $(M, Id) \rhd M \in \Sigma_{\mathcal{M}}$.

Left identity: Suppose that for some $M_i, M_j \in \mathcal{M}$, we have

$$\{\mathsf{bind}_{M_i,M_j,M_j}, \mathsf{unit}_{M_i}\} \in \Sigma_{\mathcal{M}}$$

$$\begin{array}{lll} & \mathsf{bind}_{M_i,M_j,M_j}\ (\mathsf{unit}_{M_i}\ e)\ k & \\ = & \mathit{bind}_{M_j}\ (f_{M_i,M_j}\ (\mathit{unit}_{M_i}\ e))\ (\lambda x.\, f_{M_j,M_j}\ (k\ x)) & \text{def} \\ = & \mathit{bind}_{M_j}\ (f_{M_i,M_j}\ (\mathit{unit}_{M_i}\ e))\ x & \text{(vii)} \\ = & \mathit{bind}_{M_j}\ (\mathit{unit}_{M_j}\ e)\ k & \text{(iv)} \\ = & k\ e & \text{(i)} \end{array}$$

Right identity: Suppose that we have

$$\{\mathsf{bind}_{M_i,M_j,M_i}, \mathsf{unit}_{M_j}\} \in \Sigma_{\mathcal{M}}$$

$$\begin{array}{lll} & \mathsf{bind}_{M_i,M_j,M_i}\ e\, \mathsf{unit}_{M_j} & \\ = & \mathit{bind}_{M_i}\ (f_{M_i,M_i}\ e)\ (\lambda x.\, f_{M_j,M_i}\ (\mathit{unit}_{M_j}\ x)) & \text{def} \\ = & \mathit{bind}_{M_i}\ e\ (\lambda x.\, f_{M_j,M_i}\ (\mathit{unit}_{M_j}\ x)) & \text{(vii)} \\ = & \mathit{bind}_{M_i}\ e\, \mathit{unit}_{M_i} & \text{(iv)} \\ = & e & \text{(ii)} \end{array}$$

Associativity: **(1)** ($\Rightarrow$) direction. Suppose that we have

$$\mathsf{bind}_{M_1,M_2,M_{12}} \in \Sigma_S \text{ and } \mathsf{bind}_{M_{12},M_3,M_{123}} \in \Sigma_S$$

FroM the definition of closure, we have $M_1 \rhd M_{12} \in S$, $M_2 \rhd M_{12} \in S$, $M_{12} \rhd M_{123} \in S$, and $M_3 \rhd M_{123} \in S$. Then we can choose $M_{23} = M_{123}$, and easily show $\mathsf{bind}_{M_2,M_3,M_{23}} \in \Sigma_S$ and $\mathsf{bind}_{M_1,M_{23},M_{123}} \in \Sigma_S$. Indeed, $M_1 \rhd M_{123} \in S$ and $M_2 \rhd M_{123} \in S$ follow by transitivity, $M_3 \rhd M_{123} \in S$ is an assumption, and $M_{123} \rhd M_{123} \in S$ is an axioM. The direction ($\Leftarrow$) is analogous.

(2) We want to show that

$$\begin{array}{ll} & \mathit{bind}_{M_{12},M_3,M_{123}}\ (\mathit{bind}_{M_1,2,M_{12}}\ e_1\ k_2)\ k_{M_3} \\ = & \mathit{bind}_{M_1,M_j,M_{123}}\ e_1\ (\lambda z.\, \mathit{bind}_{2,M_3,M_j}\ (k_2\ e_1)\ k_{M_3}) \end{array}$$

provided that all the mentioned binds exist. We are going to show that both the left and the right sides of this equality are equivalent to the same expression, as shown in Figure 11.

Paired morphisms: Suppose $(M_1, Id) \rhd M_2 \in \Sigma_{\mathcal{M}}$, then by def. of $\mathit{Clos}(S)$ we must have that $S \models M_1 \rhd M_2$ and $S \models Id \rhd M_2$ which, by def. of $\mathit{Clos}(S)$ implies $(Id, M_1) \rhd M_2 \in \Sigma_{\mathcal{M}}$. Same argument goes in the reverse direction.

Composition closure: Suppose $(M_1, M_2) \rhd M_3 \in \Sigma_{\mathcal{M}}$. By def of closure, we know $S \models M_1 \rhd M_3$ and $S \models M_2 \rhd M_3$. Suppose $M_2' \succ M_2$, $M_1' \succ M_1$, and $M_3 \succ M_3'$; this implies that $(M_2', Id) \rhd M_2 \in \Sigma_{\mathcal{M}}$ and thus that $S \models M_2' \rhd M_2$; we similarly know that $S \models M_1' \rhd M_1$ and $S \models M_3 \rhd M_3'$. But since $S \models$ is transitive, we have $S \models M_2' \rhd M_3'$ and $S \models M_1' \rhd M_3'$ which implies $(M_1', M_2') \rhd M_3' \in \mathit{Clos}(S)$. □

Now we show that for all S such that $\langle S \rangle = \mathcal{M}, \Sigma_{\mathcal{M}}$ then $\models \langle S \rangle$ implies $\models S$.

Proof. To show $\models S$ we prove that monad laws (i-vi) hold for all the binds in S such that $\mathit{Clos}(S) = \Sigma_{\mathcal{M}}$. For each law, we start by assuming the given binds/units/morphisms are defined in S, which implies the corresponding existence of binds in $\Sigma_{\mathcal{M}}$. Existence of other binds used in proofs of laws (iv)–(vi) is justified at the end.

$$\begin{array}{lr} (i) & \\ \quad \mathit{bind}_M\ (\mathit{unit}_M\ e)\ k & \\ = \mathsf{bind}_{M,M,M}\ (\mathsf{unit}_M\ e)\ k & \text{def} \\ = k\ e & \text{left id} \end{array}$$

$$\begin{array}{lr} (ii) & \\ \quad \mathit{bind}_M\ e\ \mathit{unit}_M & \\ = \mathsf{bind}_{M,M,M}\ e\ \mathsf{unit}_M & \text{def} \\ = e & \text{right id} \end{array}$$

$$\begin{array}{lr} (iii) & \\ \quad \mathit{bind}_M\ (\mathit{bind}_M\ e\ k_1)\ k_2 & \\ = \mathsf{bind}_{M,M,M}\ (\mathsf{bind}_{M,M,M}\ e\ k_1)\ k_2 & \text{def} \\ = \mathsf{bind}_{M,M,M}\ e\ (\lambda x.\ \mathsf{bind}_{M,M,M}\ (k_1\ x)\ k_2) & \text{Assoc} \\ = \mathit{bind}_M\ e\ (\lambda x.\ \mathit{bind}_M\ (k_1\ x)\ k_2) & \text{def} \end{array}$$

$$\begin{array}{lr} (iv) & \\ \quad f_{M_1,M_2}\ (\mathit{unit}_{M_1}\ e) & \\ = \mathsf{bind}_{M_1,Id,M_2}\ (\mathsf{bind}_{Id,Id,M_1}\ e\ \mathsf{unit}_{Id})\ \mathsf{unit}_{Id} & \text{def} \\ = \mathsf{bind}_{Id,Id,M_2}\ e\ (\lambda x.\ \mathsf{bind}_{Id,Id,Id}\ (\mathsf{unit}_{Id}\ x)\ \mathsf{unit}_{Id}) & \text{Assoc} \\ = \mathsf{bind}_{Id,Id,M_2}\ e\ \mathsf{unit}_{Id} & \text{left id} \\ = \mathit{unit}_{M_2}\ e & \text{def} \end{array}$$

$$\begin{array}{lr} (v) & \\ \quad \mathit{bind}_{M_2}\ (f_{M_1,M_2}\ e)\ (f_{M_1,M_2} \circ k) & \\ = \mathsf{bind}_{M_2,M_2,M_2}\ (\mathsf{bind}_{M_1,Id,M_2}\ e\ \mathsf{unit}_{Id}) & \\ \quad (f_{M_1,M_2} \circ k) & \text{def} \\ = \mathsf{bind}_{M_1,M_2,M_2}\ e\ (\lambda x. & \\ \quad \mathsf{bind}_{Id,M_2,M_2}\ (\mathsf{unit}_{Id}\ x)\ (f_{M_1,M_2} \circ k)) & \text{Assoc} \\ = \mathsf{bind}_{M_1,M_2,M_2}\ e\ (f_{M_1,M_2} \circ k) & \text{left id} \\ = \mathsf{bind}_{M_1,M_2,M_2}\ e\ (\lambda x. \mathsf{bind}_{M_1,Id,M_2}\ (k\ x)\ \mathsf{unit}_{Id}) & \text{def} \\ = \mathsf{bind}_{M_1,Id,M_2}\ (\mathsf{bind}_{M_1,M_1,M_1}\ e\ k)\ \mathsf{unit}_{Id} & \text{Assoc} \\ = f_{M_1,M_2}\ (\mathit{bind}_{M_1}\ e\ k) & \text{def} \end{array}$$

$$\begin{array}{lr} (vi) & \\ \quad f_{M_2,M_3} \circ f_{M_1,M_2} & \\ = \lambda x.\mathsf{bind}_{M_2,Id,M_3}\ (\mathsf{bind}_{M_1,Id,M_2}\ x\ \mathsf{unit}_{Id})\ \mathsf{unit}_{Id} & \text{def} \\ = \lambda x.\mathsf{bind}_{M_1,Id,M_3}\ x\ (\lambda y. & \\ \quad \mathsf{bind}_{Id,Id,Id}\ (\mathsf{unit}_{Id}\ y)\ \mathsf{unit}_{Id}) & \text{Assoc} \\ = \lambda x.\mathsf{bind}_{M_1,Id,M_3}\ x\ \mathsf{unit}_{Id} & \text{left id} \\ = f_{M_1,M_3} & \text{def} \end{array}$$

$$\begin{array}{lr} (vii) & \\ \quad f_{M,Id,M}\ e\, \mathsf{unit}_{Id} & \text{def} \\ = e & \text{right id} \end{array}$$

For (iv) and (v), we know that $S \models Id \rhd M_2$ and $S \models M_2 \rhd M_2$ and $S \models M_1 \rhd M_2$, and thus

$$\{\mathsf{bind}_{Id,Id,M_2}, \mathsf{bind}_{Id,M_2,M_2}, \mathsf{bind}_{M_1,M_2,M_2}\} \subseteq \mathit{Clos}(S)$$

For (vi), we have $S \models Id \rhd M_3$ and $S \models M_1 \rhd M_3$ which implies $\mathsf{bind}_{1,Id,3} \in \mathit{Clos}(S)$.

□

B. Well-typed elaborations

This section includes the full definitions and proof of Lemma 3.

$$
\begin{array}{lll}
& bind_{M_{12},M_3,M_{123}}\,(bind_{M_1,M_2,M_{12}}\,e_1\,k_2)\,k_3 & \\
= & bind_{M_{123}}\,(f_{M_{12},M_{123}}\,(bind_{M_{12}}\,(f_{M_1,M_{12}}\,e_1)\,(\lambda x.\,f_{M_2,M_{12}}\,(k_2\,x)))\,(\lambda y.f_{M_3,M_{123}}\,(k_3\,y)) & \text{def of } Clos \\
= & bind_{M_{123}}\,(bind_{M_{123}}\,(f_{M_{12},M_{123}}\,(f_{M_1,M_{12}}\,e_1))\,(\lambda x.\,f_{M_{12},M_{123}}\,(f_{M_2,M_{12}}\,(k_2\,x))))\,(\lambda y.f_{M_3,M_{123}}\,(k_3\,y)) & \text{(v)} \\
= & bind_{M_{123}}\,(bind_{M_{123}}\,(f_{M_1,M_{123}}\,e_1)\,(\lambda x.\,(f_{M_2,M_{123}}\,(k_2\,x))))\,(\lambda y.f_{M_3,M_{123}}\,(k_3\,y)) & \text{(vi)} \\
= & bind_{M_{123}}\,(f_{M_1,M_{123}}\,e_1)\,(\lambda z.\,bind_{M_{123}}\,(f_{M_2,M_{123}}\,(k_2\,z))\,(\lambda y.f_{M_3,M_{123}}\,(k_3\,y))) & \text{(vi)} \\
\\
& bind_{M_1,M_j,M_{123}}\,e_1\,(\lambda z.\,bind_{M_2,M_3,M_j}\,(k_2\,e_1)\,k_3) & \\
= & bind_{M_{123}}\,(f_{M_1,M_{123}}\,e_1)\,(\lambda z.\,f_{M_j,M_{123}}\,(bind_{M_j}\,(f_{M_2,M_j}\,(k_2\,z))\,(\lambda y.\,f_{M_3,M_j}\,(k_3\,y)))) & \text{def of } Clos \\
= & bind_{M_{123}}\,(f_{M_1,M_{123}}\,e_1)\,(\lambda z.\,(bind_{M_{123}}\,(f_{M_j,M_{123}}\,(f_{M_2,M_j}\,(k_2\,z)))\,(\lambda y.\,f_{M_j,M_{123}}\,(f_{M_3,M_j}\,(k_3\,y))))) & \text{by (v)} \\
= & bind_{M_{123}}\,(f_{M_1,M_{123}}\,e_1)\,(\lambda z.\,(bind_{M_{123}}\,((f_{M_2,M_{123}}\,(k_2\,z))\,(\lambda y.\,(f_{M_3,M_{123}}\,(k_3\,y)))) & \text{by (v)}
\end{array}
$$

Figure 11. Associativity case for proof in Lemma 1

Notations and auxliary results

$$
\begin{array}{lcl}
\{\![\cdot]\!\} = \cdot & = & \\
\{\![\Gamma, x : \forall\bar{\nu}.P \Rightarrow \tau]\!\} & = & \{\![\Gamma]\!\}, x : \forall\bar{\nu}.\mathsf{abstyp}(P,\tau) \\
\mathsf{abstyp}(.,\tau) & = & \tau \\
\mathsf{abstyp}((\pi,P),\tau) & = & \pi \rightarrow \mathsf{abstyp}(P,\tau) \\
\mathsf{Abs}((m_1,m_2) \rhd m, P) & = & \mathsf{b}_{m_1,m_2,m} : (m_1,m_2) \rhd m, \mathsf{Abs}(P) \\
\mathsf{Abs}(\cdot) & = & \cdot
\end{array}
$$

We assume the following property about simplification:

Definition 10 (Type-preserving simplification)**.** *Constraint simplification is type-preserving when for all $P, P', \bar{\mu}, \theta$,*

if $P \xrightarrow{simplify(\bar{\mu})} P'; \theta$ then $dom(\theta) \subseteq \bar{\mu}$ and $\forall \pi.\, \pi \in P \Leftrightarrow \theta\pi \in P'$.

The following lemma says that if two multisets of constraints correspond to the same set, then their corresponding typing environment are equivalent:

Lemma 11. *For all $\Gamma, P, P', \mathsf{e}, \mathsf{t}$, if $\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash \mathsf{e} : \mathsf{t}$ and $\forall\pi.\,\pi \in P \Leftrightarrow \pi \in P'$, then $\{\![\Gamma]\!\}, \mathsf{Abs}(P') \vdash \mathsf{e} : \mathsf{t}$*

The proof is straightforward by induction on the derivation. We also use this auxiliary lemma to relate different functions of P:

Lemma 12 (Abstraction and application of evidence)**.** *For all Γ, P, e, t, we have*

1. $\{\![\Gamma]\!\} \vdash_F \mathit{abs}(P,\mathsf{e}) : \mathsf{abstyp}(P,\mathsf{t})$ *iff* $\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F \mathsf{e} : \mathsf{t}$
2. $\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F \mathit{app}(P,\mathsf{e}) : \mathsf{t}$ *iff* $\{\![\Gamma]\!\} \vdash_F \mathsf{e} : \mathsf{abstyp}(P,\mathsf{t})$

The proof is straightforward by induction on the derivation.

Proof of Lemma 3

Proof. Suppose that there are $e, \tau, m, \mathsf{e}, \Gamma$ such that either $P \mid \Gamma \vdash e : \tau \rightsquigarrow \mathsf{e}$ or $P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$. We want to show that $\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F \mathsf{e} : \tau$, or $\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F \mathsf{e} : m\,\tau$.

The proof is by induction on the derivation $P \mid \Gamma \vdash e : \tau \rightsquigarrow \mathsf{e}$ (or $P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$).

Sub-case TA-Var: We want to show that

$$\{\![\Gamma]\!\}, \mathsf{Abs}(\theta P) \vdash_F \mathsf{app}(x, \theta P) : \theta\tau\,\bar{\mu}'\,\bar{\tau}$$

From the hypothesis of (TA-Var) we know that $\Gamma(x) = \forall\bar{\mu},\bar{\alpha}.P \Rightarrow \tau$, and there is $\theta = [\bar{\mu}'/\bar{\mu}][\bar{\tau}/\bar{\alpha}]$, and $\bar{\mu}'$ fresh. So in the translated environment we get $\{\![\Gamma]\!\}(x) = \forall\bar{\mu},\bar{\alpha}.\mathsf{abstyp}(P,\tau)$. We can instantiate this type with θ and get $\{\![\Gamma]\!\} \vdash_F x : \mathsf{abstyp}(\theta\,P, \theta\tau\,\bar{\mu}'\,\bar{\tau})$, and then conclude using Lemma 12.2.

Sub-case TA-Lam and TA-Val: By induction.

Sub-case TA-Do: We need to show that

$$\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F \mathsf{b}_{m_1,m_2,\mu_3}\,\tau_1\,\tau_2\,\tau_1(\lambda x : \mathsf{t}_1.\mathsf{e}_2) : m\,\tau_2$$

when by induction hypothesis we have

$$P = P_1, P_2, (m_1, m_2) \rhd \mu_3$$

$$\{\![\Gamma]\!\}, \mathsf{Abs}(P_1) \vdash_F \mathsf{e}_1 : m_1\,\tau_1$$

$$\{\![\Gamma]\!\}, x : \tau_1, \mathsf{Abs}(P_2) \vdash \mathsf{e}_2 : m_2\,\tau_2$$

Since $(m_1, m_2) \rhd \mu_3 \in P$, by definition $\mathsf{Abs}(P)$ contains the binding $\mathsf{b}_{m_1,m_2,\mu_3} : (m_1,m_2) \rhd \mu_3$. We can type the goal using instantiation and application rules of $\vdash_F$.

Sub-case TA-Let: We want to show that

$$\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F (\lambda x{:}\forall\bar{\nu}'.\mathsf{abstyp}(P'',\tau').\,\mathsf{e})\ \mathsf{abs}(P'',\theta\mathsf{v}) : m\,\tau$$

By induction hypothesis we know

$$
\begin{array}{ll}
(IH1) & \{\![\Gamma]\!\}, \mathsf{Abs}(P') \vdash_F \mathsf{v} : \tau' \\
(IH2) & \{\![\Gamma, x{:}\forall\bar{\nu}'.\mathsf{abstyp}(P'',\tau')]\!\}, \mathsf{Abs}(P) \vdash_F \mathsf{e} : m\,\tau
\end{array}
$$

when

$$
\begin{array}{l}
\bar{\mu},\bar{\alpha} = \bar{\nu} = \mathsf{ftv}(P' \Rightarrow \tau') \setminus \mathsf{ftv}(\Gamma) \\
P' \xrightarrow{\text{simplify}(\bar{\mu} \setminus \mathsf{ftv}(\tau'))} P''; \theta \\
\bar{\nu}' = \bar{\nu} \setminus dom(\theta)
\end{array}
$$

Using (IH2) we type the function:

$$
\begin{array}{c}
\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F (\lambda x{:}\forall\bar{\nu}'.\mathsf{abstyp}(P'',\tau').\,\mathsf{e}) : \\
(\forall\bar{\nu}'.\mathsf{abstyp}(P'',\tau')) \rightarrow m\,\tau
\end{array}
$$

We apply θ to (IH1) and note that $\mathsf{dom}(\theta) \cap (\mathsf{ftv}(\tau') \cup \mathsf{ftv}(\Gamma)) = \emptyset$) by Definition 10. So we get

$$\{\![\Gamma]\!\}, \theta\mathsf{Abs}(P') \vdash_F \theta\mathsf{v} : \tau'$$

Since $\forall\pi.\,\pi \in P'' \Leftrightarrow \theta P'$, we apply Lemma 11, and get $\{\![\Gamma]\!\}, \mathsf{Abs}(P'') \vdash_F \theta\mathsf{v} : \tau'$ We can weaken the typing environment with $\mathsf{Abs}(P)$ and get

$$\{\![\Gamma]\!\}, \mathsf{Abs}(P), \mathsf{Abs}(P'') \vdash_F \theta\mathsf{v} : \tau'$$

So we can type the argument of the application in the goal:

$$\{\![\Gamma]\!\}, \mathsf{Abs}(P) \vdash_F \mathsf{abs}(P'',\theta\mathsf{v}) : \mathsf{abstyp}(P'',\tau')$$

So the function application in the goal can be given type $m\,\tau$. $\square$

C. Definitions and proofs for Section 5

The statement of Theorem 8 assumes that the constraints generated by our type inference algorithm are cycle-free. We show that this property always holds under some restrictions on the shape of type schemes in the typing environment. Intuitively, this holds because our language does not include recursion.

Definition 13 (Cycle-free type environment)**.** *We say that there is a path from m to m' in P when there exist m_1, m_2 such that*

$$(m, m_1) \rhd m_2 \in P \text{ or } (m_1, m) \rhd m_2 \in P$$

and either $m_2 = m'$ or there is a path from m_2 to m'. We say that there is a cycle in P when there exists m, and a path from m to m. Γ is cycle-free when for all $x : \forall\bar{\nu}.P \Rightarrow \tau$, P is cycle free.

Lemma 14 (Type inference produces cycle-free constraints)**.** *For all* $\Gamma, P, e, m, \tau, \mathsf{e}$*, if* Γ *is cycle-free and* $\Gamma|P \vdash e : \tau \rightsquigarrow \mathsf{e}$ *(or* $\Gamma|P \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$*), then* P *is cycle-free.*

Proof. By induction on the typing derivation $\Gamma|P \vdash e : \tau \rightsquigarrow \mathsf{e}$ (or $\Gamma|P \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$).

Case TA-Var and TA-Const: Γ must contain the scheme $\forall\bar{\mu}, \bar{\alpha}.P' \Rightarrow \tau$ and the resulting constraints are $P = [\bar{\mu}'/\bar{\mu}][\bar{\tau}/\bar{\alpha}]P'$ for some fresh $\bar{\mu}'$. By assumption. P' has no cycles and the substitution only introduces fresh variables, so P has no cycles.

Sub-case TA-Lam and TA-ID: Induction hypothesis applies.

Sub-case TA-App and TA-Do: The resulting substitution combines constraints from subterms, which are cycle-free by the induction hypothesis, with new constraints that only add edges to fresh variables, so no cycles are created.

Sub-case TA-Let: The final constraints are the same as in the second inductive hypothesis. We can apply the induction hypothesis, because the type environment extended with the type scheme for the let bound term is still cycle free. The derivation of the bound term produces cycle free constraints P' which then get simplified, and simplification does not introduce new constraints, so no cycles. □

Coherence for local modifications We show coherence by iteratively constructing ground solutions that have fewer differences and preserve semantics of the elaboration. We rely on a coherence of local modifications, to show that at every step of our construction we can *smoothly* modify the lower bounds of a constraint, and preserve groundness of the solution.

The proof of local coherence relies on the following lemma:

Lemma 15. *For all polymonad instances* $m_1 ... m_7$*, and* m_3'*,* m_6'*,*

$$\{\mathsf{bind}_{m_3 m_6 m_7}, \mathsf{bind}_{m_1 m_2 m_3}, \mathsf{bind}_{m_4 m_5 m_6} \\ \mathsf{bind}_{m_3' m_6' m_7}, \mathsf{bind}_{m_1 m_2 m_3'}, \mathsf{bind}_{m_4 m_5 m_6'}\} \in \Sigma$$

implies

$$\mathsf{bind}_{m_3 m_6 m_7}\ (\mathsf{bind}_{m_1 m_2 m_3}\ e_1\ k_2)(\lambda x.\mathsf{bind}_{m_4 m_5 m_6}\ e_4\ k_5) = \\ \mathsf{bind}_{m_3' m_6' m_7}\ (\mathsf{bind}_{m_1 m_2 m_3'}\ e_1\ k_2)(\lambda x.\mathsf{bind}_{m_4 m_5 m_6'}\ e_4\ k_5)$$

Proof. From the Associativity(1) polymonad law we know that there exist monads m_8, m_9 such that

$$\{\mathsf{bind}_{m_1 m_8 m_7}, \mathsf{bind}_{m_9 m_5 m_8}, \mathsf{bind}_{m_2 m_4 m_9}\} \in \Sigma$$

Applying Associativity(2) to the lhs of the goal we get

$$\mathsf{bind}_{m_3 m_6 m_7}\ (\mathsf{bind}_{m_1 m_2 m_3}\ e_1\ k_2)(\lambda x.\mathsf{bind}_{m_4 m_5 m_6}\ e_4\ k_5) = \\ \mathsf{bind}_{m_1 m_8 m_7}\ e_1 \\ (\lambda y.\mathsf{bind}_{m_9 m_5 m_8}\ (\mathsf{bind}_{m_2 m_4 m_9}\ (k_2\ y)\ (\lambda z.e_4))\ k_5)$$

Applying Associativity(2) to the rhs of the goal we also get

$$\mathsf{bind}_{m_3' m_6' m_7}\ (\mathsf{bind}_{m_1 m_2 m_3'}\ e_1\ k_2)(\lambda x.\mathsf{bind}_{m_4 m_5 m_6'}\ e_4\ k_5) = \\ \mathsf{bind}_{m_1 m_8 m_7}\ e_1 \\ (\lambda y.\mathsf{bind}_{m_9 m_5 m_8}\ (\mathsf{bind}_{m_2 m_4 m_9}\ (k_2\ y)\ (\lambda z.e_4))\ k_5)$$

The goal follows by transitivity of $=$.

□

Next, we formalize the notion of a local modification θ_2 of a ground solution θ_1 to constraint set.

Definition 16 (Local modification of a solution)**.** *Given a ground solution* θ_1 *to a constraint set* (π, P) *a local modification to* θ_1 *is a ground solution* θ_2 *for which the following conditions are true:*

1. θ_1*lo-bnd*$(\pi) \neq \theta_2$*lo-bnd*(π)*;*
2. $\theta_1 P = \theta_2 P$ *and* θ_1*up-bnd*$(\pi) = \theta_2$*up-bnd*(π)*;*
3. *for all* $\mu \in$ *lo-bnd*(π)*, if* $\theta_1\mu \neq \theta_2\mu$ *then flowsTo*$_P\ \mu \neq \{\}$.

Conditions (1) and (2) establish that the substitutions differ for the variables that occur in *lo-bnd*(π), and coincide on all other variables. Condition (3) states that the variables whose values differ in θ_1 and θ_2 must have incoming edges in P.

We suppose that the constraints generated at type inference are labeled with the name of the typing rule where they have been introduced, for example $(m_1, m_2) \overset{\mathsf{Do}}{\rhd} m_3$ denotes a bind introduced by the rule (TA-Do).

Theorem 17 (Local coherence)**.**
Given $\Sigma_0, \Gamma, P, e, m, \tau, \theta_1, \theta_2, \mathsf{e}$*, such that*

1. *Signature* Σ *is well-formed.*
2. $P \mid \Gamma \vdash e : \tau \rightsquigarrow \mathsf{e}$ *or* $P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$*.*
3. θ_1 *is a ground solution for* P*, and* θ_2 *is a local modification of* θ_1*.*
4. $\mathsf{dom}(\theta_1) \subseteq (ftv(P) \setminus (ftv(\tau))$ *(or* $\mathsf{dom}(\theta_1) \subseteq (ftv(P) \setminus (ftv(m\,\tau))$*)*

Then, $\theta_1\mathsf{e} = \theta_2\mathsf{e}$*.*

Proof. (Sketch) Since θ_2 is a local modification of θ_1, we have (from condition (1) of Def. 16) that $P = \pi, P'$ and θ_1*lo-bnd*$(\pi) \neq \theta_2$*lo-bnd*(π). We proceed by cases on the shape of π.

Case π is an Do bundle: We have $\theta_1\pi = (m_L, m_R) \overset{\mathsf{Do}}{\rhd} m_U$ and $\theta_2\pi = (m_L', m_R') \overset{\mathsf{Do}}{\rhd} m_U$. From Condition (3) of Definition 16, if $m_L \neq m_L'$, then it's a variable μ_L, and *flowsTo*$_P\ \mu_L \neq \{\}$. Similarly, if $m_R \neq m_R'$, then it's a variable μ_R, and *flowsTo*$_P\ \mu_R \neq \{\}$.

We proceed by cases on the shape of each of the constraints (π', π'') in sets *flowsTo*$_P\ \mu_L$ and *flowsTo*$_P\ \mu_R$, respectively.

Sub-case π' and π'' are Do bundles: From examining the typing rule TA-Do, we have $\theta_1\pi' = (m_1, m_2) \overset{\mathsf{Do}}{\rhd} \mu_L$ and $\theta_2\pi' = (m_1, m_2) \overset{\mathsf{Do}}{\rhd} \mu_L'$ for some m_1, m_2; we similarly have $\theta_1\pi'' = (m_1, m_2) \overset{\mathsf{Do}}{\rhd} \mu_R$ and $\theta_2\pi'' = (m_1', m_2') \overset{\mathsf{Do}}{\rhd} \mu_R'$ for some m_1', m_2'. From the shape of the constraints, we reason that we have a source term e of this form:

$$(\mathsf{let}\ y = (\mathsf{let}\ z = e_3\ \mathsf{in}\ e_4)\ \mathsf{in}\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$$

that is elaborated to the terms shown below, where $\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3$, and e_4 are the elaborated of the sub-terms. The solution θ_1 yields the following term $\hat{\mathsf{e}}$:

$$\mathsf{bind}_{m_L, m_R, m_U}\ (\mathsf{bind}_{m_1, m_2, m_L}\mathsf{e}_3\ \ (\lambda z{:}\_.\ \mathsf{e}_4)) \\ (\lambda y{:}\_.\ \mathsf{bind}_{m_1', m_2', m_R}\mathsf{e}_1\ (\lambda x{:}\_.\ \mathsf{e}_2))$$

whereas the solution θ_2 yields the following term:

$$\mathsf{bind}_{m_L', m_R', m_U}\ (\mathsf{bind}_{m_1, m_2, m_L'}\mathsf{e}_3\ \ (\lambda z{:}\_.\ \mathsf{e}_4)) \\ (\lambda y{:}\_.\ \mathsf{bind}_{m_1', m_2', m_R'}\mathsf{e}_1\ (\lambda x{:}\_.\ \mathsf{e}_2))$$

The equality of the terms $\theta_1\hat{\mathsf{e}}$ and $\theta_2\hat{\mathsf{e}}$ follows from Lemma 15.

Case Other cases: To do. □

Proof of coherence (Theorem 8)

Proof. Suppose that for some $\Sigma, \Gamma, P, e, m, \tau, \theta_1, \theta_2, \mathsf{e}, \bar{\mu}, \bar{\alpha}$, we have

1. Σ is well-formed and P is cycle-free.
2. $P \mid \Gamma \vdash e : \tau \rightsquigarrow \mathsf{e}$ or $P \mid \Gamma \vdash e : m\,\tau \rightsquigarrow \mathsf{e}$.
3. There exist open variables $\bar{\mu}, \bar{\alpha} = ftv(P) \backslash ftv(\tau)$ (or $\bar{\mu}, \bar{\alpha} = ftv(P) \backslash ftv(m\,\tau)$) such that for all $\mu \in \bar{\mu}$, the sets *flowsTo*$_P\ \mu$ and *flowsFrom*$_P\ \mu$ are non-empty, i.e., each $\mu \in \bar{\mu}$ has a lower and an upper bound.
4. θ_1 and θ_2 are ground solutions for P such that $\theta_1(\nu) = \theta_2(\nu)$ for all $\nu \notin \bar{\alpha}, \bar{\mu}$.

Since θ_1 and θ_2 are ground solutions for P under the original signature Σ, they are also ground solutions under the saturated signature Σ_{sat}. We can construct a solution θ_3 with $\mathsf{dom}(\theta_3) = \mathsf{dom}(\theta_2)$, such that it assigns the same monads as θ_2 to monad variables: $\forall\mu \in \bar{\mu}.\theta_3\mu = \theta_2\mu$, and the same types as θ_1 to type variables: $\forall\alpha \in \bar{\alpha}.\,\theta_3\alpha = \theta_1\alpha$. By construction, solutions θ_3 and θ_2 only differ for $\bar{\alpha}$, and thus θ_3 is also ground under the saturated signature Σ_{sat}. By successively applying the (Param) property (required to hold by Condition 1 of Definition 6) to every constraint that contains a variable from $\bar{\alpha}$, we show that $\theta_3\mathsf{e} = \theta_2\mathsf{e}$.

Now we prove that $\theta_1\mathsf{e} = \theta_3\mathsf{e}$ by induction on the number of monad variables whose values differ in the solutions. Notationally, we write $\theta\backslash A$ to be the substitution with domain $\mathsf{dom}(\theta) \setminus A$ such that $(\theta\backslash A)(\nu) = \theta(\nu)$ for all $\nu \in \mathsf{dom}(\theta\backslash A)$.

The rest of the proof is by iteration on the number n of constraints in P with variables μ in their lower bounds and which are solved differently by θ_1 and θ_3:

$$\bar{\pi} = \{\pi \mid \pi \in P, ftv(\textit{lo-bnd}(\theta_1\pi)) \cap \bar{\mu} \neq \{\}, \theta_1\pi \neq \theta_3\pi\}$$

Because P is cycle-free, we can sort it topologically. For the proof we number the constraints $\bar{\pi}$ in reverse topological order:

$$\forall i, j.\ \textit{up-bnd}(\pi_i) \in \textit{lo-bnd}(\pi_j) \implies i > j \quad (\text{Topo-}\pi)$$

At each iteration i we construct a θ_1^i and θ_3^i for which the following **invariant** holds:

1. θ_1^i and θ_3^i are ground solutions to P under the saturated signature Σ_{sat}
2. $\theta_1^i\mathsf{e} \cong \theta_1\mathsf{e}$ and $\theta_3^i\mathsf{e} \cong \theta_3\mathsf{e}$
3. $\forall k.\ 1 \leq k \leq i \implies \theta_1^i\pi_k = \theta_3^i\pi_k$
4. $\forall k.\ i+1 \leq k \leq n \implies \theta_1^i\pi_k = \theta_1\pi_k \wedge \theta_3^i\pi_k = \theta_3\pi_k$
5. $\forall\pi \in P.\ \nexists k.\pi_k = \pi \implies \theta_1^i\pi = \theta_3^i\pi$

Base case: if $n = 0$, the two solutions do not differ and the lemma holds trivially. Now suppose that the invariant holds for i iterations. We show how to construct θ_1^{i+1} and θ_2^{i+1}.

Let us consider the constraint $\pi_{i+1} = (m_L, m_R) \rhd m_U$. From its definition we know that $ftv(\pi_{i+1}) \cap \bar{\mu} \neq \{\}$, so we consider all positions where open variables may appear, and the values assigned to them by the substitutions θ_1^{i+1} and θ_3^{i+1}.

From (Topo-π) we have that if there is a π_k s.t. $m_U \in \textit{lo-bnd}(\pi_k)$ then $i+1 > k$. From condition (3) of the invariant, we have that $\theta_1^k\pi_k = \theta_2^k\pi_k$. So m_U is either a constant, or a variable that has the same value in θ_1^k and θ_2^k: $\theta_1^{i+1}m_U = \theta_1^k m_U$. If there is no such π_k, then we can draw the same conclusion since we are given that all open μ variables have an upper bound.

We consider the case when m_L and m_R are equal to some open variables μ_L and μ_R, resp., and have different values in θ_1 and θ_3. From Condition (2) of Definition 6 we know that there is a pair $m_L^\star$ and $m_R^\star$ such that $(m_L^\star, m_R^\star) = \textit{smooth}((\theta_1\mu_L, \theta_1\mu_R), (\theta_3\mu_L, \theta_3\mu_R))$. We note that if either m_L or m_R is a constant, $m_L^\star$ or $m_R^\star$ will be equal to the same constant, respectively, as *smooth* preserves constants by definition. So from here on we assume m_L is some variable μ_L and m_R is some variable μ_R.

Now we define θ_1^{i+1} and θ_3^{i+1} as follows:

$$\begin{aligned}
\theta^{i+1} &= (\mu_L \mapsto m_L^\star)\ (\mu_R \mapsto m_R^\star)\\
\theta_1^{i+1} &= \theta^{i+1}\ \theta_1^i\backslash\{\mu_L, \mu_R\}\\
\theta_3^{i+1} &= \theta^{i+1}\ \theta_3^i\backslash\{\mu_L, \mu_R\}
\end{aligned}$$

The last three conditions of the invariant follow directly from this construction. Next we show why θ_1^{i+1} and θ_2^{i+1} are ground solution, as in Definition 7. Since the co-domain of θ_1^{i+1} and θ_3^{i+1} is constructed from the co-domain of θ_1^i and θ_3^i, which are themselves ground solutions, and from the results of *smooth* function, we know that co-domain of θ_1^{i+1} and θ_3^{i+1} only contains ground polymonads and types. Now we have to show that $\Sigma_{sat} \vDash \theta_1^{i+1}\pi$ and $\Sigma_{sat} \vDash \theta_3^{i+1}\pi$ for all $\pi \in P$.

From the induction hypothesis we know that θ_1^i and θ_3^i are ground; so $\Sigma_{sat} \vDash \theta_1^i\pi$ and $\Sigma_{sat} \vDash \theta_3^i\pi$ for all $\pi \in P$.

We consider different cases for all the constraints $\pi \in P$.

Sub-case $\pi = \pi_{i+1}$: The first property of lubs applies: since we have $\theta_1^i((m_L, m_R) \rhd m_U$ and
$\theta_3^i((m_L, m_R) \rhd m_U$ and
the invariant establishes that $\theta_1^i m_U = \theta_1^{i+1}m_U = \theta_3^i m_U = \theta_3^{i+1}m_U$, so we have
$\theta_1^{i+1}((m_L, m_R) \rhd m_U)$ and
$\theta_3^{i+1}((m_L, m_R) \rhd m_U)$.

Sub-case π **such that** ***up-bnd***$(\pi) \in$ ***lo-bnds***(π_{i+1}): Second property of lubs: for all m, m' if $(m, m') \rhd \theta_1^i m_L$ or $(m, m') \rhd \theta_3^i m_L$ then $(m, m') \rhd \theta^{i+1}m_L$. Similarly, for all m, m' if $(m, m') \rhd \theta_1^i m_R$ or $(m, m') \rhd \theta_3^i m_R$ then $(m, m') \rhd \theta^{i+1}m_R$.

Sub-case Other $\pi \in P$: are unchanged, so they are still entailed.

So the solutions θ_1^{i+1} and θ_3^{i+1} are both ground. Now we prove that the second condition of the invariant holds for θ_1^{i+1} and θ_3^{i+1}. By construction, θ_1^{i+1} (and θ_3^{i+1} respectively) are local modifications of θ_1^i (and θ_3^i). By Theorem 17, we know that $\theta_1^i\,\mathsf{e} = \theta_1^{i+1}\,\mathsf{e}$ and $\theta_3^i\,\mathsf{e} = \theta_3^{i+1}\,\mathsf{e}$. By transitivity of $=$, $\theta_1^{i+1}\,\mathsf{e} = \theta_1\,\mathsf{e}$ and $\theta_3^{i+1}\,\mathsf{e} = \theta_3\,\mathsf{e}$.

At the end of the iterations, we have $\theta_1^n\,\mathsf{e} = \theta_3^n\,\mathsf{e}$. By transitivity, $\theta_1\,\mathsf{e} = \theta_1^n\,\mathsf{e} = \theta_3^n\,\mathsf{e} = \theta_3\,\mathsf{e}$. We combine this result with $\theta_3\,\mathsf{e} = \theta_2\,\mathsf{e}$, we get $\theta_1\,\mathsf{e} = \theta_2\,\mathsf{e}$. □

D. Categorical foundations

In this section we give some preliminary details of our categorical study of polymonads. Whilst the main focus of this paper is the generalization of monads for programming, there has been considerable semantic work on generalizing monads; in particular, we mention Filinski's layered monads [9] and Atkey's parameterised monads [2].

Definition 18 (Polymonads)**.** *Given a (cartesian) category, $\mathbb{C}$, a polymonad is given by two collections:*

1. *A collection of endofunctors over $\mathbb{C}$:*

$$\mathbf{T} = \{T_i \colon \mathbb{C} \to \mathbb{C} \mid i \in \{0..n\}\}$$

where we require each endofunctor T_i to come equipped with a strength $\tau_{A,B}^i \colon A \times T_i(B) \to T_i(A \times B)$ that satisfies the following two diagrams.

$$\begin{tikzcd}
T_i(A) & \\
1 \times T_i(A) \arrow[u, "\mathsf{snd}"] \arrow[r, "\tau_A^i"'] & T_i(1 \times A) \arrow[ul, "T_i(\mathsf{snd})"']
\end{tikzcd}$$

$$\begin{tikzcd}
(A \times B) \times T_i(C) \arrow[rr, "\tau_{A\times B,C}^i"] \arrow[d, "\alpha"'] & & T_i((A \times B) \times C) \arrow[d, "T_i(\alpha)"] \\
A \times (B \times T_i(C)) \arrow[r, "Id\times\tau_{B,C}^i"'] & A \times T_i(B \times C) \arrow[r, "\tau_{A,B\times C}^i"'] & T_i(A \times (B \times C))
\end{tikzcd}$$

By convention in any given collection, T_0 is always the identity functor.

2. *A collection of natural transformations:*

$$\circledast_{\mathbf{T}} = \{\star^j_A \colon T_{j_2}(T_{j_1}(A)) \to T_{j_3}(A) \mid j \in \{0..m\}, \\ T_{j_1} \in \mathbf{T}, T_{j_2} \in \mathbf{T}, T_{j_3} \in \mathbf{T}\}$$

where the collection must satisfy the (strong) polymonad laws. Again, it is by convention that $\star^0$ is the identity natural transformation between identity functors, i.e. $\star^0_A \colon T_0(T_0(A)) \to T_0(A)$.

A collection $\circledast_{\mathbf{T}}$ is said to be polymonadic if it satisfies the following.

(a) $\forall T \in \mathbf{T}. \exists \star^1 \in \circledast_{\mathbf{T}}$ *such that* $\star^1 \colon T(T_0(A)) \to T(A)$ *and* $\star^1 = Id$.

(b) $\forall \star^1 \in \circledast_{\mathbf{T}}$ *such that* $\star^1_A \colon T_1(T_0(A)) \to T_2(A)$ *then* $\exists \star^2 \in \circledast_{\mathbf{T}}$ *such that* $\star^2_A \colon T_0(T_1(A)) \to T_2(A)$ *and* $\star^1 = \star^2$*, and vice versa.*

(c) $\forall \star^1, \star^2 \in \circledast_{\mathbf{T}}$ *such that* $\star^1_A \colon T_1(T_2(A)) \to T_3(A)$ *and* $\star^2_A \colon T_1(T_2(A)) \to T_3(A)$ *then* $\star^1 = \star^2$.

(d) $\forall \star^1, \star^2, \star^3, \star^4 \in \circledast_{\mathbf{T}}$ *if* $\star^1 \colon T_2(T_1(A)) \to T_3(A)$, $\star^2 \colon T_0(T'_1(A)) \to T_1(A)$, $\star^3 \colon T_0(T'_2(A)) \to T_2(A)$ *and* $\star^4 \colon T_0(T_3(A)) \to T'_3(A)$ *then* $\exists \star^5 \in \circledast_{\mathbf{T}}$ *such that* $\star^5 \colon T'_2(T'_1(A)) \to T'_3(A)$.

(e) $\forall \star^1, \star^2, \star^3, \star^4 \in \circledast_{\mathbf{T}}$ *such that* $\star^1_A \colon T_1(T_2(A)) \to T_4(A)$, $\star^2_A \colon T_4(T_3(A)) \to T_5(A)$, $\star^3_A \colon T_2(T_3(A)) \to T_6(A)$ *and* $\star^4_A \colon T_1(T_6(A)) \to T_5(A)$*, the following diagram commutes.*

$$\begin{tikzcd}
T_1(T_2(T_3(A))) \arrow[r, "\star^1_{T_3(A)}"] \arrow[d, "T_1(\star^3_A)"'] & T_4(T_3(A)) \arrow[d, "\star^2_A"] \\
T_1(T_6(A)) \arrow[r, "\star^4_A"'] & T_5(A)
\end{tikzcd}$$

A polymonadic collection $\circledast_{\mathbf{T}}$ is said to be strong if it in addition satisfies the following laws.

(a) $\forall \star^i_A \colon T_2(T_1(A)) \to T_3(A) \in \circledast_{\mathbf{T}}$ *the following diagram commutes.*

$$\begin{tikzcd}
A \times T_3(B) \arrow[rr, "\tau^3_{A,B}"] & & T_3(A \times B) \\
A \times T_2(T_1(B)) \arrow[u, "Id \times \star^i_B"] \arrow[r, "\tau^2_{A,T_1(B)}"'] & T_2(A \times T_1(B)) \arrow[r, "T_2(\tau^1_{A,B})"'] & T_2(T_1(A \times B)) \arrow[u, "\star^i_{A \times B}"']
\end{tikzcd}$$

(b) $\forall \star^i_A \colon T_0(T_0(A)) \to T_1(A) \in \circledast_{\mathbf{T}}$ *the following diagram commutes.*

$$\begin{tikzcd}
A \times B \arrow[d, "Id \times \star^i_A"'] \arrow[dr, "\star^i_{A \times B}"] & \\
A \times T_1(B) \arrow[r, "\tau^1_{A,B}"'] & T_1(A \times B)
\end{tikzcd}$$

The definition is relatively straightforward. The polymonad law (a) requires that there is an identity natural transformation between every functor in the endofunctor set. Polymonad law (b) essentially ensures that the identity functor acts like the unit in the monoid where functor composition acts as the multiplication. Polymonad law (c) reflects the coherence restriction of our setting that there can only be only bind operation between any three functors. Polymonad law (d) is the analog of the composition closure law defined in Section 2.2.

Polymonad law (e) is the generalization of the associativity law for monads, and laws (f) and (g) are the generalizations of the strong monad laws.

It is interesting to note that we do not require the generalization of monad triangle laws. In fact, these are derivable.

Lemma 19. *For every polymonad, the following diagrams commute.*

1. For $\star^1 \colon T_0(T_0(A)) \to T_2(A)$, $\star^2 \colon T_1(T_2(A)) \to T_3(A)$ *and* $\star^3 \colon T_0(T_1(A)) \to T_3(A)$

$$\begin{tikzcd}
T_1(A) \arrow[r, "T_1(\star^1_A)"] \arrow[dr, "\star^3_A"'] & T_1(T_2(A)) \arrow[d, "\star^2_A"] \\
& T_3(A)
\end{tikzcd}$$

2. For $\star^1 \colon T_0(T_0(A)) \to T_1(A)$, $\star^2 \colon T_1(T_2(A)) \to T_3(A)$ *and* $\star^3 \colon T_0(T_2(A)) \to T_3(A)$

$$\begin{tikzcd}
T_2(A) \arrow[r, "\star^1_{T_2(A)}"] \arrow[dr, "\star^3_A"'] & T_1(T_2(A)) \arrow[d, "\star^2_A"] \\
& T_3(A)
\end{tikzcd}$$

Proof. 1. In the following diagram, the square commutes as it is an instance of the polymonad law (d). Laws (a) and (b) combine to enforce that $\star^4_A \colon T_1(T_0(A)) \to T_1(A)$ both exists in the polymonad and is the identity natural transformation. Hence the left triangle commutes.

$$\begin{tikzcd}
T_1(A) \arrow[r, equal] \arrow[dr, equal] & T_1(T_0(T_0(A))) \arrow[r, "T_1(\star^1_A)"] \arrow[d, "\star^4_{T_0(A)}"] & T_1(T_2(A)) \arrow[d, "\star^2"] \\
& T_1(T_0(A)) \arrow[r, "\star^3(A)"'] & T_3(A)
\end{tikzcd}$$

2. In the following diagram, the polymonad laws (a) and (b) combine to enforce that $\star^4_A \colon T_0(T_3(A)) \to T_3(A)$ both exists in the polymonad and is the identity natural transformation. The square commutes as it is an instance of the polymonad law (d).

$$\begin{tikzcd}
T_2(A) \arrow[r, equal] & T_0(T_0(T_2)) \arrow[r, "\star^1_{T_2(A)}"] \arrow[d, "T_0(\star^3)"] & T_1(T_2(A)) \arrow[d, "\star^2_A"] \\
& T_0(T_3(A)) \arrow[r, "\star^4"'] & T_3(A)
\end{tikzcd}$$

□

Atkey [3] proposed an generalization of a strong monad where the functor is no longer an endofunctor but a functor $T \colon \mathbb{S}^{\mathrm{op}} \times \mathbb{S} \times \mathbb{C} \to \mathbb{C}$, where $\mathbb{S}$ is an additional category whose objects are intended to denote states and morphisms denote state transitions.

Theorem 20. *Every $\mathbb{S}$-parameterized monad (T, η, μ) on $\mathbb{C}$ [3] is a polymonad.*

By currying the functor $T \colon \mathbb{S}^{\mathrm{op}} \times \mathbb{S} \to (\mathbb{C} \to \mathbb{C})$, the polymonad functor and natural transformation collections can be defined by enumeration over the objects in $\mathbb{S}$. Checking that the polymonad laws are satisfied is routine.