Polymonadic programming

Mike Hicks[2], Gavin Bierman[1], Nataliya Guts[2], Daan Leijen[1], and Nik Swamy[1]

[1] Microsoft Research
[2] University of Maryland, College Park

Abstract. Monads are a popular tool for the working functional programmer to structure effectful computations. This paper presents *polymonads*, a generalization of monads. Polymonads give the familiar monadic bind the more general type $\forall a, b.\ \mathsf{L}\ a \to (a \to \mathsf{M}\ b) \to \mathsf{N}\ b$, to compose computations with three different kinds of effects, rather than just one. Polymonads subsume monads and parameterized monads, and can express other constructions, including precise type-and-effect systems and information flow tracking; more generally, polymonads correspond to Tate's *productoid* semantic model. We show how to equip a core language (called $\lambda_{\underline{\mathrm{PM}}}$) with syntactic support for programming with polymonads. Type inference and elaboration in $\lambda_{\underline{\mathrm{PM}}}$ allows programmers to write polymonadic code directly in an ML-like syntax—our algorithms compute principal types and elaborate programs (to System F) wherein the binds appear explicitly. Furthermore, we prove that the elaboration is *coherent*: no matter which (type-correct) binds are chosen, the elaborated program's semantics will be the same. Pleasingly, the inferred types are easy to read: the polymonad laws justify (sometimes dramatic) simplifications, but with no effect on the type's generality.

1 Introduction

Since the time that Moggi first connected them to effectful computation [17], *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 [10], probabilistic computations [20], and functional reactivity [6, 3].

Monads (and morphisms between them) are not a panacea, however, and so researchers have proposed various extensions. Examples include Wadler and Thiemann's [24] indexed monad for typing effectful computations; Atkey's parameterized monad [2], which has been used to encode disciplines like regions [14] and session types [19]; Devriese and Piessens' [5] monad-like encodings for information flow controls; and many others. Oftentimes these extensions are used to prove stronger properties about computations than would be possible with monads, e.g., to prove the absence of information leaks or memory errors.

Unfortunately, these extensions do not enjoy the same status as monads in terms of language support. For example, the conveniences that Haskell provides for monadic programs (e.g., the `do` notation combined with type-class inference) do not apply to these extensions. One might imagine adding specialized support for each of these extensions on a case-by-case basis, but a unifying construction into which all of them, including normal monads, fit is clearly preferable.

This paper proposes just such a unifying construction, making several contributions. Our first contribution is the definition of a *polymonad*, a new way to structure effectful computations. Polymonads give the familiar monadic bind (having type $\forall a, b.\ \mathsf{M}\ a \to (a \to \mathsf{M}\ b) \to \mathsf{M}\ b$) the more general type $\forall a, b.\ \mathsf{L}\ a \to (a \to \mathsf{M}\ b) \to \mathsf{N}\ b$. That is, a polymonadic bind can compose computations with three different types to a monadic bind's one. Section 2 defines polymonads formally, along with the *polymonad laws*, which we prove are a generalization the monad and morphism laws. To precisely characterize their expressiveness, we prove that polymonads correspond to Tate's *productoids* [23] (Theorem 1), a recent semantic model general enough to capture most known effect systems, including all the constructions listed above.[3]

Whereas Tate instantiates his model only in a first-order setting, we show how to make polymonads work in a higher-order language. Our second contribution is the definition of $\lambda_{\underline{\mathrm{PM}}}$ (Section 3), an ML-like programming language well-suited to programming with polymonads. We work out several examples in $\lambda_{\underline{\mathrm{PM}}}$, including novel polymonadic constructions for stateful information flow tracking, contextual type and effect systems [18], and session types.

Our examples are made practical by $\lambda_{\underline{\mathrm{PM}}}$'s support for type inference and elaboration, which allows programs to be written in a familiar ML-like notation while making no mention of the bind operators. Enabling this feature, our third contribution (Section 4) is an instantiation of Jones' theory of qualified types [11] to $\lambda_{\underline{\mathrm{PM}}}$. In a manner similar to Haskell's type class inference, we show that type inference for $\lambda_{\underline{\mathrm{PM}}}$ computes *principal types* (Theorem 2). Our inference algorithm is equipped with an elaboration phase, which translates source terms to System F by inserting binds where needed. We prove that elaboration is *coherent* (Theorem 3), meaning that when inference produces constraints that could have several solutions, when these solutions are applied to the elaborated terms the results will have equivalent semantics, thanks to the polymonad laws. This property allows us to do better than Haskell, which does not take such laws into account, and so needlessly rejects programs it thinks might be ambiguous. Moreover, as we show in Section 5, the polymonad laws allow us to dramatically simplify types, making them far easier to read without compromising their generality. A prototype implementation of $\lambda_{\underline{\mathrm{PM}}}$ is available from the first author's web page and has been used to check all the examples in the paper.

Put together, our work lays the foundation for providing practical support for advanced monadic programming idioms in typed, functional langaues.

2 Polymonads

Polymonads generalize a setting of many monads and morphisms between them.

[3] In fact, we discovered the same model concurrently with Tate and independently of him, though our presentation here has benefited from conversations with him.

Definition 1. *A **polymonad** $(\mathcal{M}, \Sigma)$ consists of (1) a collection $\mathcal{M}$ of unary type constructors, with a distinguished element $\mathsf{Id} \in \mathcal{M}$, such that $\mathsf{Id}\ \tau = \tau$, and (2) a collection, Σ, of* bind *operators such that the laws given in Figure 1 hold.*

In the figure, we write the type of bind operators using the shorthand $(\mathsf{M}, \mathsf{N}) \rhd \mathsf{P}$, instead of $\forall \mathsf{a}\ \mathsf{b}.\ \mathsf{M}\ \mathsf{a} \to (\mathsf{a} \to \mathsf{N}\ \mathsf{b}) \to \mathsf{P}\ \mathsf{b}$. We also write $\mathsf{M} \hookrightarrow \mathsf{N}$ as a shorthand for $(\mathsf{M}, \mathsf{Id}) \rhd \mathsf{N}$ (to evoke its use as a kind of monad morphism).

(Functor) For all $\mathsf{M} \in \mathcal{M}$, there exists $(\mathsf{M}, \mathsf{Id}) \rhd \mathsf{M} \in \Sigma$
(Identity) For all $\mathsf{M} \in \mathcal{M}$, if $\mathsf{bind}\colon (\mathsf{M}, \mathsf{Id}) \rhd \mathsf{M} \in \Sigma$ then for all $\mathsf{m}\colon \mathsf{M}\ \tau$,
$\mathsf{bind}\ \mathsf{m}\ (\lambda \mathsf{y}.\mathsf{y}) = \mathsf{m}$
(Paired morphisms) For all $\mathsf{M}, \mathsf{N} \in \mathcal{M}$,
$\mathsf{bind}_1\colon (\mathsf{M}, \mathsf{Id}) \rhd \mathsf{N} \in \Sigma \iff \mathsf{bind}_2\colon (\mathsf{Id}, \mathsf{M}) \rhd \mathsf{N} \in \Sigma$ where
$\mathsf{bind}_2\ \mathsf{v}\ \mathsf{f} = \mathsf{bind}_1\ (\mathsf{f}\ \mathsf{v})\ (\lambda \mathsf{y}.\mathsf{y})$.
(Associativity (1)) For all $\mathsf{M}, \mathsf{N}, \mathsf{P}, \mathsf{T} \in \mathcal{M}$.
$\exists \mathsf{R}.\{(\mathsf{M}, \mathsf{N}) \rhd \mathsf{R}, (\mathsf{R}, \mathsf{P}) \rhd \mathsf{T}\} \subseteq \Sigma \iff \exists \mathsf{S}.\{(\mathsf{N}, \mathsf{P}) \rhd \mathsf{S}, (\mathsf{M}, \mathsf{S}) \rhd \mathsf{T}\} \subseteq \Sigma$.
(Associativity (2)) If $\mathsf{bind}_1\colon (\mathsf{M}, \mathsf{N}) \rhd \mathsf{P} \in \Sigma, \mathsf{bind}_2\colon (\mathsf{P}, \mathsf{R}) \rhd \mathsf{T} \in \Sigma$,
$\mathsf{bind}_3\colon (\mathsf{M}, \mathsf{S}) \rhd \mathsf{T} \in \Sigma$, and $\mathsf{bind}_4\colon (\mathsf{N}, \mathsf{R}) \rhd \mathsf{S} \in \Sigma$ then
for all $m\colon \mathsf{M}\ \tau$, $f\colon \tau \to \mathsf{N}\ \tau'$ and $g\colon \tau' \to \mathsf{R}\ \tau''$
$\mathsf{bind}_2\ (\mathsf{bind}_1\ m\ f)\ g = \mathsf{bind}_3\ m\ (\lambda x.\mathsf{bind}_4\ (f\ x)\ g)$
(Closure) If $\{(\mathsf{M}, \mathsf{N}) \rhd \mathsf{P}, \mathsf{S} \hookrightarrow \mathsf{M}, \mathsf{T} \hookrightarrow \mathsf{N}, \mathsf{P} \hookrightarrow \mathsf{U}\} \subseteq \Sigma$, then $(\mathsf{S}, \mathsf{T}) \rhd \mathsf{U} \in \Sigma$.

Fig. 1. Polymonad laws

The (Functor) law simply asks for a bind of type $\mathsf{M}\ \mathsf{a} \to (\mathsf{a} \to \mathsf{Id}\ \mathsf{b}) \to \mathsf{M}\ \mathsf{b}$ for every type constructor M. Rearranging this type and treating the Id type operator as an identity yields the type $(\mathsf{a} \to \mathsf{b}) \to \mathsf{M}\ \mathsf{a} \to \mathsf{M}\ \mathsf{b}$. So, this condition is no more than asking for a map operator, as one would expect from monads.

The (Identity) law simply requires that a bind corresponding to a map operator preserves identities (this is a requirement of a functor). The (Paired Morphisms) law amounts to a coherence condition that all monad morphisms can be re-expressed as binds.

The (Associativity (2)) law is the familiar associativity law for monads generalized for both our more liberal typing for bind operators and for the fact that we have a *collection* of binds rather than a single bind. The (Associativity (1)) law essentially guarantees a coherence property for associativity, namely that it is always possible to complete an application of the (Associativity (2)) law.

The (Closure) law is another coherence-like property that ensures closure under composition of monad morphisms with binds.

The first thing to check is that every monad is a polymonad.

Lemma 1. *If* $(\mathsf{M}, \mathsf{map}, \mathsf{unit}, \mathsf{bind})$ *is a monad then* $(\{\mathsf{M}, \mathsf{Id}\}, \{b_1, b_2, b_3, b_4\})$ *is a polymonad where* $b_1 = \lambda x\colon \mathsf{M}\ \mathsf{a}.\lambda f\colon \mathsf{a} \to \mathsf{Id}\ \mathsf{b}.\mathsf{map}\ f\ x$, $b_2 = \lambda x\colon \mathsf{Id}\ \mathsf{a}.\lambda f\colon \mathsf{a} \to \mathsf{M}\ \mathsf{b}.f\ x$, $b_3 = \mathsf{bind}$, $b_4 = \lambda x\colon \mathsf{Id}\ \mathsf{a}.\lambda f\colon \mathsf{a} \to \mathsf{Id}\ \mathsf{a}.\mathsf{unit}\ x$.

Definition 1 may look a little austere, but there is a simple refactoring that yields something more directly resembling monads and morphisms. Appendix A gives this definition and proves the following theorem.

Signatures$(\mathcal{M}, \Sigma)$:	k-ary constructors	$\mathcal{M}$	$::= \cdot \mid \mathsf{M}/k, \mathcal{M}$
	ground constructor	M	$::= \mathsf{M}\ \overline{\tau}$
	bind set	Σ	$::= \cdot \mid \mathsf{b}{:}s, \Sigma$
	bind specifications	s	$::= \forall \bar{a}.\Phi \Rightarrow (\mathrm{M}_1, \mathrm{M}_2) \rhd \mathrm{M}_3$
	theory constraints	Φ	
Terms:	values	v	$::= x \mid c \mid \lambda x.e$
	expressions	e	$::= v \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$
			$\mid\ \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{letrec}\ f = v\ \mathsf{in}\ e$
Types:	monadic types	m	$::= \mathrm{M} \mid \rho$
	value types	τ	$::= a \mid T\ \overline{\tau} \mid \tau_1 \to m\ \tau_2$
	type schemes	σ	$::= \forall \bar{a}\bar{\rho}.P \Rightarrow \tau$
	bag of binds	P	$::= \cdot \mid \pi, P$
	bind type	π	$::= (m_1, m_2) \rhd m_3$

Fig. 2. $\lambda\underline{\text{PM}}$: Syntax for signatures, types, and terms

Theorem 1. *Every polymonad gives rise to a productoid, and every productoid (that satisfies some mild additional conditions) is a polymonad.*

Productoids have been recently proposed as a new categorical foundation for effectful computation [23]. Tate demonstrates the expressive power of productoids by showing how they subsume other proposed extensions to monads [24, 7, 2]. This theorem shows polymonads can be soundly interpreted using productoids, and that they have similar expressive power. We give examples of this expressive power in subsequent sections.

3 Programming with polymonads

This section presents $\lambda\underline{\text{PM}}$, an ML-like language for programming with polymonads, along with several examples of its use. Figure 2 presents $\lambda\underline{\text{PM}}$'s syntax.

Polymonadic signatures. A $\lambda\underline{\text{PM}}$ *polymonadic signature* $(\mathcal{M}, \Sigma)$ amends Definition 1 in two ways. Firstly, each element M of $\mathcal{M}$ may be *type-indexed*—we write M/k to indicate that M is a $(k+1)$-ary type constructor (we sometimes omit k for brevity). For example, constructor $W/1$ could represent an effectful computation so that $W\ \epsilon\ \tau$ characterizes computations of type τ that have effect ϵ. We write M to denote *ground constructors*, which are monadic constructors applied to all their type indexes; e.g., $W\ \ \epsilon$ is ground. Secondly, a bind set Σ is not specified intensionally as a set, but rather extensionally using a language of *theory constraints* Φ. In particular, Σ is a list of mappings $\mathsf{b}{:}s$ where s contains a triple $(\mathrm{M}_1, \mathrm{M}_2) \rhd \mathrm{M}_3$ along with constraints Φ, which determine how the constructors may be instantiated. For example, a mapping $\mathsf{sube} : \forall \varepsilon_1, \varepsilon_2.\, \varepsilon_1 \subseteq \varepsilon_2 \Rightarrow (W\ \varepsilon_1, \mathsf{Id}) \rhd W\ \varepsilon_2$ specifies the set of binds involving type indexes $\varepsilon_1, \varepsilon_2$ such that the theory constraint $\varepsilon_1 \subseteq \varepsilon_2$ is satisfied.

$\lambda\underline{\text{PM}}$'s type system is parametric in the choice of theory constraints Φ, which allows us to encode a variety of prior monad-like systems with $\lambda\underline{\text{PM}}$, as we show shortly. To interpret a particular set of constraints, $\lambda\underline{\text{PM}}$ requires a theory entailment relation $\models$. Elements of this relation, written $\Sigma \models \pi \rightsquigarrow \mathsf{b}; \theta$, state that there exists $\mathsf{b}{:}\forall\bar{a}.\Phi \Rightarrow (\mathrm{M}_1, \mathrm{M}_2) \rhd \mathrm{M}_3$ in Σ and a substitution θ' such that $\theta\pi = \theta'(\mathrm{M}_1, \mathrm{M}_2) \rhd \mathrm{M}_3$, and the constraints $\theta'\Phi$ are satisfiable.[4] Here, θ is a substitution for the free (non-constant) variables in π, while θ' is an instantiation of the abstracted variables in the bind specification. The interpretation of Σ is thus the set $\{\mathsf{b}{:}\pi \mid \Sigma \models \pi \rightsquigarrow \mathsf{b}; \cdot\}$. Signature $(\mathcal{M}, \Sigma)$ is a polymonad if this set satisfies the polymonad laws (where each ground constructor is treated distinctly).

Terms and types. $\lambda\underline{\text{PM}}$'s term language is standard. $\lambda\underline{\text{PM}}$ programs do not explicitly reference binds, but are written in *direct style*, treating computations $\mathrm{M}\ \tau$ as if of type τ. Type inference determines where and which bind operations to insert (or abstract) to type check a program against a polymonadic interface.

To make inference feasible, we rely crucially on $\lambda\underline{\text{PM}}$'s call-by-value structure. Following our prior work on monadic programming for ML [22], we restrict the shape of types assignable to a $\lambda\underline{\text{PM}}$ program by separating value types τ from the types of polymonadic computations $m\ \tau$. Here, metavariable m may be either a ground constructor M or a polymonadic type variables ρ. The co-domain of every function is required to be a computation type $m\ \tau$, although pure functions can be typed $\tau \rightarrow \tau'$, which is a synonym for $\tau \rightarrow \mathsf{Id}\ \tau'$. We also include types $T\ \bar{\tau}$ for fully applied type constructors, e.g., $\mathsf{list}\ \mathit{int}$.

Programs can also be given type schemes σ that are polymorphic in their polymonads, e.g., $\forall a, b, \rho.(a \rightarrow \rho\, b) \rightarrow a \rightarrow \rho\, b$. Here, the variable a ranges over all value types τ, while ρ ranges over computation types m. Type schemes may also be qualified by a set P of bind constraints π. For example, $\forall\rho.(\rho, \mathsf{Id}) \rhd \mathsf{M} \Rightarrow (\mathit{int} \rightarrow \rho\ \mathit{int}) \rightarrow \mathsf{M}\ \mathit{int}$ is the type of a function that abstracts over a bind having shape $(\rho, \mathsf{Id}) \rhd \mathsf{M}$. Notice that π triples may contain functorial type variables ρ while specification triples $s \in \Sigma$ may not. Moreover, Φ constraints never appear in σ, which is thus entirely independent of the choice of the theory.

3.1 Polymonadic information flow controls

Polymonads are appealing because they can express many interesting constructions as we now show.

Figure 3 presents a polymonad IST, which implements *stateful* information flow tracking [5, 21, 16, 4, 1]. The idea is that some program values are secret and some are public, and no information about the former should be learned by observing the latter—a property called noninterference [9]. In the setting of IST, we are worried about leaks via the heap.

Heap-resident storage cells are given type *intref* l where l is the secrecy label of the referenced cell. Labels $l \in \{L, H\}$ form a lattice with order $L \sqsubset H$. A program is acceptable if data labeled H cannot flow, directly or indirectly, to computations or storage cells labeled L. In our polymonad implementation, L and H are just types T (but only ever serve as indexes), and the lattice ordering is implemented by theory constraints $l_1 \sqsubseteq l_2$ for $l_1, l_2 \in \{L, H\}$.

[4] We write $\Sigma \models \pi; \theta$ when we do not care about the elaborated binds.

Signature:

$\mathcal{M} \; = \; \mathsf{IST}/2$

$\Phi \; ::= \; l_1 \sqsubseteq l_2 \mid \Phi_1, \Phi_2$

$\Sigma \; = \; \mathsf{bId}: \quad \mathsf{Id} \hookrightarrow \mathsf{Id},$

$\quad \mathsf{unitIST}: \forall p, l.\, \mathsf{Id} \hookrightarrow \mathsf{IST}\ p\ l,$

$\quad \mathsf{mapIST}: \forall p_1, l_1, p_2, l_2.\, p_2 \sqsubseteq p_1, l_1 \sqsubseteq l_2 \Rightarrow \mathsf{IST}\ p_1\ l_1 \hookrightarrow \mathsf{IST}\ p_2\ l_2,$

$\quad \mathsf{appIST}: \forall p_1, l_1, p_2, l_2.\, p_2 \sqsubseteq p_1, l_1 \sqsubseteq l_2 \Rightarrow (\mathsf{Id}, \mathsf{IST}\ \mathsf{p}_1\ \mathsf{l}_1) \rhd \mathsf{IST}\ \mathsf{p}_2\ \mathsf{l}_2,$

$\quad \mathsf{bIST}: \forall p_1, l_1, p_2, l_2, p_3, l_3.\, l_1 \sqsubseteq p_2, l_1 \sqsubseteq l_3, l_2 \sqsubseteq l_3, p_3 \sqsubseteq p_1, p_3 \sqsubseteq p_2 \Rightarrow (\mathsf{IST}\ \mathsf{p}_1\ \mathsf{l}_1, \mathsf{IST}\ \mathsf{p}_2\ \mathsf{l}_2) \rhd \mathsf{IST}\ \mathsf{p}_3\ \mathsf{l}_3$

Types and auxiliary functions:

$\tau: \quad \ldots \mid \mathit{intref}\ \tau \mid L \mid H$

$\mathsf{read}: \forall l.\, \mathit{intref}\ l \to \mathsf{IST}\ H\ l\ \mathit{int}$

$\mathsf{write}: \forall l.\, \mathit{intref}\ l \to \mathit{int} \to \mathsf{IST}\ l\ L\ ()$

Example program:

```
let add_interest = λsavings. λinterest.
 let currinterest = read interest in
 if currinterest > 0 then
   let currbalance = read savings in
   let newbalance =
     currbalance + currinterest in
   write savings newbalance
 else ()
```

Fig. 3. Polymonad IST, implementing stateful information flow control

The polymonadic constructor $\mathsf{IST}/2$ uses secrecy labels for its type indexes. A computation with type $\mathsf{IST}\ p\ l\ \tau$ potentially writes to references labeled p and returns a τ-result that has security label l; we call p the *write label* and l the *output label*. Function read reads a storage cell, producing a $\mathsf{IST}\ H\ l\ \mathit{int}$ computation—the second type index l matches that of l-labeled storage cell. Function write writes a storage cell, producing a $\mathsf{IST}\ l\ L\ ()$ computation—the first type index l matches the label of the written-to storage cell. H is the most permissive write label and so is used for the first index of read, while L is the most permissive output label and so is used for the second index of write.

Aside from the identity bind bId, implemented as reverse apply, there are four kinds of binds. Units $\mathsf{unitIST}\ p\ l$ lift a normal term into an IST computation. Binds $\mathsf{mapIST}\ p\ l$ lift a computation into a more permissive context (i.e., p_2 and l_2 are at least as permissive as l_1 and l_2), and $\mathsf{appIST}\ p\ l$ do likewise, and are implemented using mapIST: $\mathsf{appIST}\ p\ l = \lambda x.\lambda f.\mathsf{mapIST}\ p\ l\ (f\ x)\ (\lambda x.x)$. Finally, binds bIST compose a computation $\mathsf{IST}\ p_1\ l_1\ \alpha$ with a function $\alpha \to \mathsf{IST}\ p_2\ l_2\ \beta$. The constraints ensure safe information flow: $l_1 \sqsubseteq p_2$ prevents the second computation from leaking information about its l_1-secure α-typed argument into a reference cell that is less than l_1-secure. Dually, the constraints $l_1 \sqsubseteq l_3$ and $l_2 \sqsubseteq l_3$ ensure that the β-typed result of the composed computation is at least as secure as the results of each component. The final constraints $p_3 \sqsubseteq p_1$ and $p_3 \sqsubseteq p_2$ ensure that the write label of the composed computation is a lower bound of the labels of each component.

Proving $(\mathcal{M}, \Sigma)$ satisfies the polymonad laws is straightforward. The functor, identity, and paired morphism laws are relatively easy to prove. The Associativity(1) law is more tedious: we must consider all possible pairs of binds that can properly compose. This reasoning involves consideration of the theory constraints as implementing a lattice, and so would work for any lattice of labels, not just H and L. In all, there were ten cases to consider. We prove the Associativity(2) law for the same ten cases. This proof is straightforward as the implementation of IST ignores the indexes: read, write and various binds are just as in a normal state monad, while the indexes serve only to prevent illegal flows.

Finally, proving Closure is relatively straightforward—we start with each possible bind shape and then consider correctly-shaped flows into its components; in all there were eleven cases.

Example. The lower right of Figure 3 shows an example use of IST. The $\mathsf{add\_interest}$ function takes two reference cells, $\mathsf{savings}$ and $\mathsf{interest}$, and modifies the former by adding to it the latter if it is non-negative. (This example is a bit contrived for illustration purposes which will be clear later.) Notice that expressions of type $\mathsf{IST}\ p\ l\ \tau$ are used as if they merely had type τ—this is happening in the branch on $\mathsf{currinterest}$, for example. The program is rewritten during type inference to insert, or abstract, the necessary binds so that the program type checks. This process results in the following type for $\mathsf{add\_interest}$:[5]

$$\begin{array}{l}\forall \rho_6, \rho_{27}, a_1, a_2.P \Rightarrow \mathit{intref}\ a_1 \rightarrow \mathit{intref}\ a_2 \rightarrow \rho_{27}\ () \\ \text{where } P = (\mathsf{Id}, \mathsf{Id}) \rhd \rho_6, (\mathsf{IST}\ H\ \mathsf{a}_1, \mathsf{IST}\ \mathsf{a}_1\ L) \rhd \rho_6, (\mathsf{IST}\ H\ \mathsf{a}_2, \rho_6) \rhd \rho_{27}\end{array}$$

The rewritten version of $\mathsf{add\_interest}$ starts with a sequence of λ abstractions, one for each of the bind constraints in P. If we imagine these are numbered $\mathsf{b1}$ ... $\mathsf{b3}$, e.g., where $\mathsf{b1}$ is a bind with type $(\mathsf{Id}, \mathsf{Id}) \rhd \rho_6$, then the term looks as follows (notation ... denotes code elided for simplicity):

$$\begin{array}{l}\lambda\mathsf{savings.}\ \lambda\mathsf{interest.}\ \mathsf{b3}\ (\mathsf{read\ interest}) \\ \quad (\lambda\ \mathsf{currinterest.}\ \textbf{if}\ \mathsf{currinterest} > 0\ \textbf{then}\ (\mathsf{b2}\ ...)\ \textbf{else}\ (\mathsf{b1}\ ()\ (\lambda\ \mathsf{z.\ z})))\end{array}$$

In a program that calls $\mathsf{add\_interest}$, the bind constraints will be solved, and actual implementations of these binds will be passed in for each of b_i (using a kind of dictionary-passing style as with Haskell type classes).

Looking at the type of $\mathsf{add\_interest}$ we can see how the constraints prevent improper information flows. In particular, if we tried to call $\mathsf{add\_interest}$ with $a_1 = L$ and $a_2 = H$, then the last two constraints become $(\mathsf{IST}\ H\ L, \mathsf{IST}\ L\ L) \rhd \rho_6, (\mathsf{IST}\ H\ H, \rho_6) \rhd \rho_{27}$, and so we must instantiate ρ_6 and ρ_{27} in a way allowed by the signature in Figure 3. While we can legally instantiate $\rho_6 = \mathsf{IST}\ L\ l_3$ for any l_3 to solve the second constraint, there is then no possible instantiation of ρ_{27} that can solve the third constraint. After substituting for ρ_6, this constraint has the form $(\mathsf{IST}\ H\ H, \mathsf{IST}\ L\ l_3) \rhd \rho_{27}$, but this form is unacceptable because the H output of the first computation could be leaked by the L side effect of the second computation. On the other hand, all other instantiations of a_1 and a_2 (e.g., $a_1 = H$ and $a_2 = L$ to correspond to a secret savings account but a public interest rate) do have solutions and do not leak information.

The type for $\mathsf{add\_interest}$ is not its principal type, but an *improved* one. As it turns out, the principal type is unreadable, with 19 bind constraints! Fortunately, Section 5 shows how some basic rules can greatly simplify types without reducing their applicability, as has been done above. Moreover, our coherence result (given in the next section) assures that the corresponding changes to the elaborated term do not depend on the particular simplifications: the polymonad laws ensure all such elaborations will have the same semantics.

[5] This and other example types were generated by our prototype implementation.

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

Types and theory constraints:

$$
\begin{array}{ll}
\tau & ::= ... \mid \{A_1\}...\{A_n\} \mid \emptyset \mid \top \mid \tau_1 \cup \tau_2 \\
\Phi & ::= \tau \subseteq \tau' \mid \tau = \tau' \mid \Phi, \Phi
\end{array}
$$

Auxiliary functions:

$$
\begin{array}{l}
\mathsf{read} :\ \forall \alpha, \omega, r.\ \mathit{intref}\ r \rightarrow \mathsf{CE}\ \alpha\ \{r\}\ \omega\ \mathit{int} \\
\mathsf{write} : \forall \alpha, \omega, r.\ \mathit{intref}\ r \rightarrow \mathit{int} \rightarrow \mathsf{CE}\ \alpha\ \{r\}\ \omega\ ()
\end{array}
$$

Fig. 4. Polymonad expressing contextual type and effect systems

3.2 Contextual type and effect systems

Wadler and Thiemann [24] showed how a monadic-style construct can be used to model type and effect systems. Polymonads can model standard effect systems, but more interestingly can be used to model *contextual effects* [18], which augment traditional effects with the notion of *prior* and *future* effects of an expression within a broader context. As an example, suppose we are using a language that partitions memory into *regions* $R_1, ..., R_n$ and reads/writes of references into region R have effect $\{R\}$. Then in the context of the program $\mathsf{read}\ r_1; \mathsf{read}\ r_2$, where r_1 points into region R_1 and r_2 points into region R_2, the contextual effect of the subexpression $\mathsf{read}\ r_1$ would be the triple $[\emptyset; \{R_1\}; \{R_2\}]$: the prior effect is empty, the present effect is $\{R_1\}$, and the future effect is $\{R_2\}$.

Figure 4 models contextual effects as the polymonad $\mathsf{CE}\ \alpha\ \epsilon\ \omega\ \tau$, for the type of a computation with prior, present, and future effects α, ϵ, and ω, respectively. Indices are sets of atomic effects $\{A_1\}...\{A_n\}$, 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. As an example source of effects, we include read and write functions on references into regions r. The bind unitce lifts a computation into a contextual effect monad with empty effect and any prior or future effects. The binds appce and mapce express 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 computation, and the future effect is the same as after the second computation.

3.3 Parameterized monads, and session types

Finally, we show $\lambda_{\underline{\mathrm{PM}}}$ can express Atkey's parameterized monad [2], which has been used to encode disciplines like regions [14] and session types [19]. The type

$$
\begin{array}{ll}
\mathcal{M} = \mathsf{Id}, A/2 & \textit{Types:} \\
\Sigma \;\; = \mathsf{bId} : (\mathsf{Id}, \mathsf{Id}) \rhd \mathsf{Id}, & \tau ::= \cdots \mid \textit{send}\,\tau_1\,\tau_2 \mid \textit{recv}\,\tau_1\,\tau_2 \mid \textit{end} \\
\quad\;\; \mathsf{mapA} : \forall p, r.\ (A\ p\ r, \mathsf{Id}) \rhd A\ p\ r, & \\
\quad\;\; \mathsf{appA} : \forall p, r.\ (\mathsf{Id}, A\ p\ r) \rhd A\ p\ r, & \textit{Auxiliary functions:} \\
\quad\;\; \mathsf{unitA} : \forall p.\ (\mathsf{Id}, \mathsf{Id}) \rhd A\ p\ p, & \mathsf{send}\ : \forall a, q.\, a\ \to A\,(\textit{send}\,q\,a)\,q\,() \\
\quad\;\; \mathsf{bindA} : \forall p, q, r.\ (A\,p\,q,\ A\,q\,r) \rhd A\,p\,r & \mathsf{recv}\ : \forall a, q.\,()\ \to A\,(\textit{recv}\,q\,a)\,q\,a
\end{array}
$$

Fig. 5. Parameterized monad for session types, expressed as a polymonad

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.

As a concrete example, Figure 5 gives a polymonadic expression of Pucella and Tov's notion of session types [19]. The type $A\,p\,q\,\tau$ represents a computation involved in a two-party session which starts in protocol state p and completes in state q, returning a value of type τ. The key element of the signature Σ is the bindA, which permits composing two computations where the first's post-condition matches the second's precondition. We use the type index $\textit{send}\,q\,\tau$ to denote a protocol state that requires a message of type τ to be sent, and then transitions to q. Similarly, the type index $\textit{recv}\,r\,\tau$ denotes the protocol state in which once a message of type τ is received, the protocol transitions to r. We also use the index *end* to denote the protocol end state. The signatures of two primitive operations for sending and receiving messages capture this behavior.

As an example, the following $\lambda\underline{\text{PM}}$ program implements one side of a simple protocol that sends a message x, waits for an integer reply y, and returns y+1.

$$
\textbf{let}\ \mathsf{go} = \lambda\mathsf{x}.\ \textbf{let}\ \_ = \mathsf{send}\ \mathsf{x}\ \textbf{in}\ \mathsf{incr}\ (\mathsf{recv}\ ())
$$
$$
\text{Simplified type: } \forall a, b, q, \rho.\,(A\,(\textit{send}\ \ a\,b)\,a,\ A\,(\textit{recv}\ \ q\ \textit{int})\,q)) \rhd \rho \Rightarrow\ (b \to \rho\ \textit{int})
$$

There are no specific theory constraints for session types: constraints simply arise by unification and are solved as usual when instantiating the final program (e.g., to call go 0).

4 Coherent type inference for $\lambda\underline{\textbf{pm}}$

Mike says: Add this sentence somewhere here We write $(\mathrm{P}, \mathrm{Q}) \rhd \mathrm{R} \in \Sigma$ to mean $\Sigma \models (\mathrm{P}, \mathrm{Q}) \rhd \mathrm{R} \rightsquigarrow \mathsf{b}; \cdot$.

This section defines our declarative type system, together with type inference and a coherence result. Figure 6 show a syntax-directed type system for $\lambda\underline{\text{PM}}$, organized into two main judgments. The value-typing judgment $P \,|\, \Gamma \vdash v : \tau \rightsquigarrow \mathsf{e}$ types a value v in an environment Γ (binding variables x and constants c to type schemes) at the type τ, provided the constraints P are satisfiable. Moreover, it *elaborates* the value v into a lambda term e that explicitly contains binds, lifts, and evidence passing (as shown in Section 3.1. However, note that the elaboration is independent and we can read just the typing rules by igoring the elaborated terms. The expression-typing judgment $P \,|\, \Gamma \vdash e : m\ \tau \ \rightsquigarrow \mathsf{e}$ is similar, except that it yields a computation type. Constraint satisfiability is

$$\boxed{P \models P'} \qquad \frac{\forall \pi \in P'.\pi \in P \vee \pi \in \Sigma}{P \models P'} \text{ (TS-Entail)}$$

$$\boxed{P \models \sigma \geqslant \tau \rightsquigarrow \mathsf{f}} \qquad \frac{\theta = [\bar{\tau}/\bar{a}][\bar{m}/\bar{\rho}] \qquad P \models \theta P_1}{P \models (\forall \bar{a}\bar{\rho}.\, P_1 \Rightarrow \tau) \geqslant \theta\tau \rightsquigarrow \mathsf{app}(\theta P_1)} \text{ (TS-Inst)}$$

$$\boxed{P \,|\, \Gamma \vdash v : \tau \rightsquigarrow \mathsf{e}} \qquad \frac{v \in \{x, c\} \qquad P \models \Gamma(v) \geqslant \tau \rightsquigarrow \mathsf{f}}{P \,|\, \Gamma \vdash v : \tau \rightsquigarrow \mathsf{f}\, v} \text{ (TS-XC)}$$

$$\frac{P \,|\, \Gamma, x{:}\tau_1 \vdash e : m\ \tau_2 \rightsquigarrow \mathsf{e}}{P \,|\, \Gamma \vdash \lambda x.e : \tau_1 \to m\ \tau_2 \rightsquigarrow \lambda x.\mathsf{e}} \text{ (TS-Lam)}$$

$$\boxed{P \,|\, \Gamma \vdash e : m\ \tau \rightsquigarrow \mathsf{e}} \qquad \frac{P \,|\, \Gamma \vdash v : \tau \rightsquigarrow \mathsf{e}}{P, \mathsf{Id} \hookrightarrow m \,|\, \Gamma \vdash v : m\ \tau \rightsquigarrow \mathsf{b}_{\mathsf{Id},\mathsf{Id},m}\ \mathsf{e}\ (\lambda x.x)} \text{ (TS-V)}$$

$$\frac{\begin{array}{c} P_1 \,|\, \Gamma, x{:}\tau \vdash v : \tau \rightsquigarrow \mathsf{e}_1 \qquad (\sigma, \mathsf{e}_2) = \mathit{Gen}(\Gamma, P_1 \Rightarrow \tau,\ \mathsf{e}_1) \\ P \,|\, \Gamma, x{:}\sigma \vdash e : m\ \tau' \rightsquigarrow \mathsf{e}_3 \end{array}}{P \,|\, \Gamma \vdash \mathsf{letrec}\ x{=}v\ \mathsf{in}\ e : m\ \tau' \rightsquigarrow \mathsf{letrec}\ x{=}\mathsf{e}_2\ \mathsf{in}\ \mathsf{e}_3} \text{ (TS-Rec)}$$

$$\frac{\begin{array}{c} P_1 \,|\, \Gamma \vdash v : \tau \rightsquigarrow \mathsf{e}_1 \qquad (\sigma, \mathsf{e}_2) = \mathit{Gen}(\Gamma, P_1 \Rightarrow \tau,\ \mathsf{e}_1) \\ P \,|\, \Gamma, x{:}\sigma \vdash e : m\ \tau' \rightsquigarrow \mathsf{e}_3 \end{array}}{P \,|\, \Gamma \vdash \mathsf{let}\ x{=}v\ \mathsf{in}\ e : m\ \tau' \rightsquigarrow \mathsf{let}\ x{=}\mathsf{e}_2\ \mathsf{in}\ \mathsf{e}_3} \text{ (TS-Let)}$$

$$\frac{\begin{array}{cc} P \,|\, \Gamma \vdash e_1 : m_1\ \tau_1 \rightsquigarrow \mathsf{e}_1 & P \,|\, \Gamma, x{:}\tau_1 \vdash e_2 : m_2\ \tau_2 \rightsquigarrow \mathsf{e}_2 \\ e_1 \neq v & P \models (m_1, m_2) \rhd m_3 \end{array}}{P \,|\, \Gamma \vdash \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : m_3\ \tau_2 \rightsquigarrow \mathsf{b}_{m_1,m_2,m_3}\ \mathsf{e}_1\ (\lambda x.\, \mathsf{e}_2)} \text{ (TS-Do)}$$

$$\frac{\begin{array}{cc} P \,|\, \Gamma \vdash e_1 : m_1\ (\tau_2 \to m_3\ \tau) \rightsquigarrow \mathsf{e}_1 & P \,|\, \Gamma \vdash e_2 : m_2\ \tau_2 \rightsquigarrow \mathsf{e}_2 \\ P \models (m_1, m_4) \rhd m_5 & P \models (m_2, m_3) \rhd m_4 \end{array}}{P \,|\, \Gamma \vdash e_1\ e_2 : m_5\ \tau \rightsquigarrow \mathsf{b}_{m_1,m_4,m_5}\ \mathsf{e}_1\ (\mathsf{b}_{m_2,m_3,m_4}\ \mathsf{e}_2)} \text{ (TS-App)}$$

$$\frac{\begin{array}{cc} P \,|\, \Gamma \vdash e_1 : m_1\ \mathsf{bool} \rightsquigarrow \mathsf{e}_1 & P \,|\, \Gamma \vdash e_2 : m_2\ \tau \rightsquigarrow \mathsf{e}_2 \\ P \,|\, \Gamma \vdash e_3 : m_3\ \tau \rightsquigarrow \mathsf{e}_3 & P \models m_2 \hookrightarrow m, m_3 \hookrightarrow m, (m_1, m) \rhd m' \end{array}}{\begin{array}{c} P \,|\, \Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : m'\ \tau \\ \rightsquigarrow \mathsf{b}_{m_1,m,m'}\ \mathsf{e}_1\ (\lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ \mathsf{b}_{m_2,\mathsf{Id},m}\ \mathsf{e}_2\ (\lambda x.x)\ \mathsf{else}\ \mathsf{b}_{m_3,\mathsf{Id},m}\ \mathsf{e}_3\ (\lambda x.x)) \end{array}} \text{ (TS-If)}$$

$$\begin{array}{ll} \mathit{Gen}(\Gamma, P \Rightarrow \tau, \mathsf{e}) & = (\forall(\mathsf{ftv}(P \Rightarrow \tau) \setminus \mathsf{ftv}(\Gamma)).P \Rightarrow \tau,\ \mathsf{abs}(P, \mathsf{e})) \\ \mathsf{abs}(((m_1, m_2) \rhd m_3, P), \mathsf{e}) & = \lambda \mathsf{b}_{m_1,m_2,m_3}\, .\, \mathsf{abs}(P, \mathsf{e}) \\ \mathsf{abs}(\cdot, \mathsf{e}) & = \mathsf{e} \\ \mathsf{app}(P, (m_1, m_2) \rhd m_3)) & = \lambda f.\, \mathsf{app}(P)\ (f\ \mathsf{b}_{m_1,m_2,m_3}\ ) \\ \mathsf{app}(P, \cdot) & = \lambda x.\, x \end{array}$$

Fig. 6. Syntax-directed type rules for $\lambda_{\underline{\text{PM}}}$, where Σ is an implicit parameter.

defined by $P \models P'$, which states that P' is satisfiable under the hypothesis P if $P' \subseteq P \cup \Sigma$.

The rule (TS-XC) types a variable or constant at an instance of its type scheme in the environment. The instance relation for type schemes $P \models \sigma \geq$

$\tau \rightsquigarrow$ f is standard: it instantiates the bound variables, and checks that the abstracted constraints are entailed by the hypothesis P. The elaborated f term supplies the instantiated evidence using the app rule. The rule (TS-Lam) is straightforward where the bound variable is given a value type and the body a computation type.

The rule (TS-V) allows a value $v : \tau$ to be used as an expression by lifting it to a computation type $m\ \tau$, so long as there exists a morphism (or unit) from the Id functor to m. In the elaborated term we use $\mathsf{b}_{\mathsf{Id},\mathsf{Id},m}$ to lift explicitly to monad m. Note that for all evidence we make up names ($\mathsf{b}_{\mathsf{Id},\mathsf{Id},m}$) based on the constraint ($\mathsf{Id} \hookrightarrow m$). This simplifies our presentation but in a real implementation one usually names each constraint explicitly [13]. We use the name $\mathsf{b}_{m_1,\mathsf{Id},m_2}$ for morphism constraints $m_1 \hookrightarrow m_2$, and use b_{m_1,m_2,m_3} for general bind constraints $(m_1, m_2) \rhd m_3$.

(TS-Rec) types a recursive let-binding by typing the definition v at the same (mono-)type as the letrec-bound variable f. When typing the body e, we generalize the type of f using a standard generalization function $Gen(\Gamma, P \Rightarrow \tau,\ \mathsf{e})$, which closes the type relative to Γ by generalizing over its free type variables. However, in constrast to regular generalization, we return both a generalized type, as well as an elaboration of e that takes all generalized constraints as explicit evidence parameters (as defined by rule abs). (TS-Let) is similar, although somewhat simpler since there is no recursion involved.

(TS-Do) is best understood by looking at its elaboration: since we are in a call-by-value setting, we interpret a **let**-binding as forcing and sequencing two computations using a single bind where e_1 is typed monomorphically.

(TS-App) is similar to (TS-Do), where, again, since we use call-by-value, in the elaboration we sequence the function and its argument using two bind operators, and then apply the function. (TS-If) is also similar, since we sequence the expression e in the guard with the branches. As usual, we require the branches to have the same type. This is achieved by generating morphism constraints, $m_2 \hookrightarrow m$ and $m_3 \hookrightarrow m$ to coerce the type of each branch to a functor m before sequencing it with the guard expression.

4.1 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 [11] 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. OML's type inference algorithm is known to enjoy principal types, so a corollary of our translation is that principal types exist for our system too.

We encode terms in our language into OML as shown in Figure 7. We rely on four primitive OML terms that force the typing of the terms to generate the same constraints as our type system does: ret for lifting a pure term, do for typing a do-binding, app for typing an application, and cond for conditionals. Using these primitives, we encode values and expressions of our system into OML.

$$
\begin{array}{ll}
[\![x]\!]^\star & = x \\
[\![c]\!]^\star & = c \\
[\![\lambda x.e]\!]^\star & = \lambda x.[\![e]\!] \\
\\
[\![v]\!] & = \texttt{ret}\ [\![v]\!]^\star \\
[\![e_1\ e_2]\!] & = \mathsf{app}\ [\![e_1]\!]\ [\![e_2]\!] \\
[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] & = \mathsf{let}\ x = [\![v]\!]^\star\ \mathsf{in}\ [\![e]\!] \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = [\![e_1]\!]\ \texttt{>>=}\ [\![\lambda x.e_2]\!]^\star \qquad (\text{when } e_1 \neq v) \\
[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] & = \texttt{cond}\ [\![e_1]\!]\ \lambda().[\![e_2]\!]\ \lambda().[\![e_3]\!] \\
[\![\mathsf{letrec}\ f = v\ \mathsf{in}\ e]\!] & = \texttt{letrec} f = [\![v]\!]^\star\ \texttt{in}\ [\![e]\!]
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{ret} & : \forall \alpha\rho.\,(\mathsf{Id} \hookrightarrow \rho) \Rightarrow \alpha \to \rho\ \alpha \\
\mathsf{do} & : \forall \alpha\beta\rho_1\rho_2\rho.\,((\rho_1, \rho_2) \rhd \rho) \Rightarrow \rho_1\ \alpha \to (\alpha \to \rho_2\ \beta) \to \rho\ \beta \\
\mathsf{app} & : \forall \alpha\beta\rho_1\rho_2\rho_3\rho_4\rho.\,((\rho_1, \rho_4) \rhd \rho, (\rho_2, \rho_3) \rhd \rho_4) \Rightarrow \rho_1\ (\alpha \to \rho_3\ \beta) \to \rho_2\ \alpha \to \rho\ \beta \\
\mathsf{cond} & : \forall \alpha\rho_1\rho_2\rho_3\rho\rho'.\,(\rho_2 \hookrightarrow \rho, \rho_3 \hookrightarrow \rho, (\rho_1, \rho) \rhd \rho') \\
& \quad \Rightarrow \rho_1\ \mathsf{bool} \to (() \to \rho_2\ \alpha) \to (() \to \rho_3\ \alpha) \to \rho'\ \alpha
\end{array}
$$

Fig. 7. Type inference for $\lambda_{\underline{\text{PM}}}$ via elaboration to OML

We write $P \,|\, \Gamma \vdash_{\text{OML}} e : \tau$ for a derivation in the syntax directed inference system of OML (cf. Jones [11], Fig. 4).

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

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, our system admits principal types via the general-purpose OML type inference algorithm.

4.2 Ambiguity

Seeing the previous OML translation, one might think we could directly translate our programs into Haskell since Haskell uses OML style type inference. Unfortunately, in practice, Haskell would reject many useful programs. In particular, Haskell[6] rejects as ambiguous any term whose type $\forall\bar{\alpha}.P \Rightarrow \tau$ includes a variable α that occurs free in P but not in τ – we call such type variables *open*. Haskell, in its generality, must reject such terms since the instantiation of an open variable can have operational effect, while at the same time, since the variable does not

[6] The actual ambiguity rule in Haskell is more involved due to functional dependencies and type families but that does not affect our results.

appear in τ, the instantiation for it can never be uniquely determined by the context in which the term is used. A common example is the term $\mathsf{show}\ .\ \mathsf{read}$ with the type $(\mathsf{Show\ a}, \mathsf{Read\ a}) \Rightarrow \mathsf{String} \rightarrow \mathsf{String}$, where a is open. Depending on the instantiation of a, the term may parse and show integers, or doubles, or characters, etc.

Rejecting all types that contain open variables works well for type classes, but it would be unacceptable for $\lambda_{\underline{\mathrm{PM}}}$. Many simple terms have principal types that contain open variables. For example, the term $\backslash \mathsf{f\ x} \rightarrow \mathsf{f\ x}$ has the type $\forall \alpha \beta m_1 m_2 m_3.((\mathsf{Id}, m_1) \rhd\ m_2, (\mathsf{Id}, m_2) \rhd\ m_3) \Rightarrow\ (\alpha \rightarrow m_1\ \beta) \rightarrow \alpha \rightarrow m_3\ \beta$ where the type variable m_2 is open.

A major contribution of this paper is that we show that for our specific bind constraints, we can relax this rule and solve much more aggressively. In particular, by appealing to the monadic laws, we can prove that programs with open type variables in bind constraints are indeed unambigiuous. Even if there are many possible instantiations, the semantics of each instantiation is equivalent. This *coherence* result is at the essence of making programming with polymonads practical and next section treats this in detail.

4.3 Coherence

The main result of this section (Theorem 3) establishes that for a certain class of polymonads, the ambiguity check of OML can be weakened to accept more programs while still ensuring that programs are coherent. Thus, for this class of polymonads, programmers can reliably view our syntax-directed system as a specification without being concerned with the details of how the type inference algorithm is implemented or how programs are elaborated.

The proof of Theorem 3 is a little technical—the following roadmap summarizes the structure of the development.

- We define the class of *principal* polymonads for which unambiguous typing derivations are coherent. All polymonads that we know of are principal.
- Given $P \,|\, \Gamma \vdash e : t \rightsquigarrow \mathsf{e}$ (with $t \in \{\sigma, m\ \tau\}$), the predicate $\mathsf{unambiguous}(P, \Gamma, t)$ characterizes when the derivation is unambiguous. This notion requires interpreting P as a graph G_P, and ensuring (roughly) that all open variables in P have non-zero in/out-degree in G_P.
- A *solution* S to a constraint graph with respect to a polymonad $(\mathcal{M}, \Sigma)$ is an assignment of polymonad constructors $M \in \mathcal{M}$ to the variables in the graph such that each instantiated constraint is present in Σ. We give an equivalence relation on solutions such that $S_1 \cong S_2$ if they differ only on the assignment to open variables in a manner where the composition of binds still compute the same function according to the polymonad laws.
- Finally, given $P \,|\, \Gamma \vdash e : t \rightsquigarrow \mathsf{e}$ and $\mathsf{unambiguous}(P, \Gamma, t)$, we prove that all solutions to P that agree on the free variables of Γ and t are in the same equivalence class.

While Theorem 3 enables our type system to be used in practice, this result is not the most powerful theorem one can imagine. Ideally, one might like a theorem of the form $P \,|\, \Gamma \vdash e : t \rightsquigarrow \mathsf{e}$ and $P' \,|\, \Gamma \vdash e : t \rightsquigarrow \mathsf{e}'$ implies $\mathsf{e} \cong \mathsf{e}'$, given that both P and P' are satisfiable. However, a result of this form is out of our reach, at present. There are at least two difficulties. First, a coherence result of this form is unknown for qualified type systems in a call-by-value setting. In an unpublished paper, Jones [12] proves a coherence result for OML, but his techique only applies to call-by-name programs. Jones also does not consider reasoning about coherence based on an equational theory for the evidence functions, i.e., the binds in our case. So, proving the ideal coherence theorem would require both generalizing Jones' approach to call-by-value and then extending it with support for equational reasoning about evidence. In the meantime, Theorem 3 provides good assurance and lays the ground for future work in this direction.

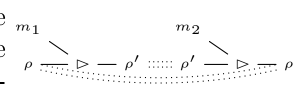
Defining and analyzing principality. Our notion of principal polymonads corresponds to Tate's principalled productoids. Informally, in a principal polymonad, if there is more than one way to combine pairs of computations in the set F (e.g., combining compuations in m and m' to either M_1 or to M_2), then there must be a "best" way to combine them. This best way is called the principal join of F, and all other ways to combine the functors are related to the principal join by morphisms. All the polymonadic libraries we have encountered so far are principal polymonads. It is worth emphasizing that principality does not correspond to functional dependency—it is perfectly reasonable to combine m and m' in multiple ways, and indeed, for applications like sub-effecting, this expressiveness is important. We only require that there be an ordering among the choices.

Definition 2 (Principal polymonad). *A polymonad* $(\mathcal{M}, \Sigma)$ *is a* principal polymonad *if and only if for any set* $F \subseteq \mathcal{M}^2$, *and any* $\{\mathsf{M}_1, \mathsf{M}_2\} \subseteq \mathcal{M}$ *such that* $\{(\mathsf{m}, \mathsf{m}') \rhd \mathsf{M}_1 \mid (m, m') \in F\} \subseteq \Sigma$ *and* $\{(\mathsf{m}, \mathsf{m}') \rhd \mathsf{M}_2 \mid (m, m') \in F\} \subseteq \Sigma$, *then there exists* $\mathsf{M} \in \mathcal{M}$ *such that* $\{\mathsf{M} \hookrightarrow \mathsf{M}_1, \mathsf{M} \hookrightarrow \mathsf{M}_2\} \subseteq \Sigma$, *and* $\{(\mathsf{m}, \mathsf{m}') \rhd \mathsf{M} \mid (m, m') \in F\} \subseteq \Sigma$. *We call* M *the principal join of* F *and write it as* $\bigsqcup F$

Definition 3 (Graph-view of a constraint-bag P). *A graph-view* $G_P = (V, A, E_{\rhd}, E_{eq})$ *of a constraint-bag* P *is a graph consisting of a set of vertices* V, *a vertex assignment* $A : V \to m$, *and a set of directed edges* $E_{\rhd}$, *and a set of undirected edges* E_{eq}, *where:*

- $V = \{\pi.0, \pi.1, \pi.2 \mid \pi \in P\}$, *i.e., each constraint contributes three vertices.*
- $\forall \pi.i \in V.A(\pi.i) = m_i$ *when* $\pi = (\mathsf{m}_0, \mathsf{m}_1) \rhd \mathsf{m}_2$.
- $E_{\rhd} = \{(\pi.0, \pi.2), (\pi.1, \pi.2) \mid \pi \in P\}$
- $E_{eq} = \{(v, v') \mid v, v' \in V \ \wedge \ v \neq v' \wedge \exists \rho.\rho = A(v) = A(v')\}$

Notation We make use of a pictorial notation for graph views, distinguishing the two flavors of edges in a graph. Each constraint $\pi \in P$ induces two edges in $E_{\rhd}$. These edges are drawn with solid lines, with a triangle for orientation. Unification constraints arise from correlated variable occurrences in multiple constraints—we depict these with double dotted lines. For example, the pair of constraints $(\mathsf{m}_1, \rho) \rhd \rho', (\mathsf{m}_2, \rho') \rhd \rho$ contributes four unification edges, two for ρ and two for ρ'. We show its graph view alongside.



Unification constraints reflect the dataflow in a program. Referring back to Figure 6, in a principal derivation using (TS-App), correlated occurrences of unification variables for m_4 in the constraints indicates how the two binds operators compose. The following definition captures this dataflow and shows how to interpret the composition of bind constraints using unification edges as a lambda term (in the obvious way).

Definition 4 (Functional view of a flow edge). *Given a constraint graph $G = (V, A, E_{\triangleright}, E_{eq})$, an edge $\eta = (\pi.2, \pi'.i) \in E_{eq}$, where $i \in \{0, 1\}$ and $\pi \neq \pi'$ is called a* flow edge. *The flow edge η has a functional interpretation $F_G(\eta)$ defined as follows:*

If $i = 0$, $F_G(\eta) = \lambda(x{:}A(\pi.0)\ a)\ (y{:}a \to A(\pi.1)\ b)\ (z{:}b \to A(\pi'.1)\ c).$
$\qquad \mathsf{bind}_{A(\pi'.0),A(\pi'.1),A(\pi'.2)}(\mathsf{bind}_{A(\pi.0),A(\pi.1),A(\pi.2)}\ x\ y)\ z$

If $i = 1$, $F_G(\eta) = \lambda(x{:}A(\pi'.0)\ a)\ (y{:}a \to A(\pi.0)\ b)\ (z{:}b \to A(\pi.1)\ c).$
$\qquad \mathsf{bind}_{A(\pi'.0),A(\pi'.1),A(\pi'.2)}\ x\ (\lambda a.\mathsf{bind}_{A(\pi.0),A(\pi.1),A(\pi.2)}\ (y\ a)\ z)$

We can now define our ambiguity check—a graph is unambiguous if it contains a sub-graph that has no cyclic dataflows, and where open variables only occur as intermediate variables in a sequence of binds.

Definition 5 (Unambiguous constraints). *Given $G_P = (V, A, E_{\triangleright}, E_{eq})$, the predicate* $\mathsf{unambiguous}(P, \Gamma, t)$ *holds if and only if there exists $E'_{eq} \subseteq E_{eq}$, such that in the graph $G' = (V, A, E_{\triangleright}, E'_{eq})$ all of the following are true.*

1. *For all $\pi \in P$, there is no path from $\pi.2$ to $\pi.0$ or $\pi.1$.*
2. *For all $v \in V$, if $A(v) \in \mathsf{ftv}(P) \setminus \mathsf{ftv}(\Gamma, t)$, then there exists a flow edge that is incident on v.*

We call G' a core *of G_P.*

Definition 6 (Solution to a constraint graph). *For a polymonadic $(\mathcal{M}, \Sigma)$, a solution to a constraint graph $G = (V, A, E_{\triangleright}, E_{eq})$, is a vertex assignment $S : V \to \mathcal{M}$ such that all of the following are true.*

1. *For all $v \in V$, if $A(v) \in \mathcal{M}$ then $S(v) = A(v)$*
2. *For all $(v_1, v_2) \in E_{eq}$, $S(v_1) = S(v_2)$.*
3. *For all $\{(\pi.0, \pi.2), (\pi.1, \pi.2)\} \subseteq E_{\triangleright}$, $(\mathsf{S}(\pi.0), \mathsf{S}(\pi.1)) \triangleright \mathsf{S}(\pi.2) \in \Sigma$.*

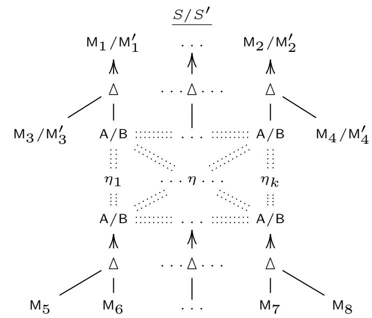
We say that two solutions S_1 and S_2 to G agree on *ρ if for all vertices $v \in V$ such that $A(v) = \rho$, $S_1(v) = S_2(v)$.*

Our notion of equivalence of two solutions captures the idea that the differences in the solutions are only to the internal open variables while not impacting the overall function computed by the binds in a constraint. It is easy to check that $\cong_P$ is an equivalence relation.

Definition 7 (Equivalence of solutions). *Given a polymonad $(\mathcal{M}, \Sigma)$ and constraint graph $G = (V, A, E_{\triangleright}, E_{eq})$, two solutions S_1 and S_2 to G are equivalent with respect to a set of variables P (denoted $S_1 \cong_P S_2$) if and only if S_1 and S_2 agree on P and for each vertex $v \in V$ such that $S_1(v) \neq S_2(v)$ for all flow edges η incident on v, $F_{G_1}(\eta) = F_{G_2}(\eta)$, where $G_i = (V, S_i, E_{\triangleright}, E_{eq})$.*

Theorem 3 (Coherence). *For all principal polymonads, derivations* $P|\Gamma \vdash e : t \rightsquigarrow \mathsf{e}$ *such that* $\mathsf{unambiguous}(P, \Gamma, t)$*, and for any two solutions* S *and* S' *to* G_P *that agree on* $P = FTV(\Gamma, t)$*, we have* $S \cong_P S'$*.*

(Sketch; full version in appendix) The main idea is to show that all solutions in the core of G_P are in the same equivalence class (the solutions to the core include S and S'). The proof proceeds by induction on the number of vertices at which S and S' differ. For the main induction step, we take vertices in topological order, considering the least (in the order) set of vertices Q, all related by unification constraints, and whose assignment in S is A and in S' is B, for some $\mathsf{A} \neq \mathsf{B}$. The vertices in Q are shown in the graph alongside, all connected to each other by double dotted lines (unification constraints), and their neighborhood is shown as well. Since vertices are considered in topological order, all the vertices below Q in the graph have the same assignment in S and in S'. We build solutions S_1 and S_1' from S and S' respectively, that instead assign the principal join $\mathsf{J} = \bigsqcup\{(\mathsf{M}_5, \mathsf{M}_6), \ldots, (\mathsf{M}_7, \mathsf{M}_8)\}$ to the vertices in Q, where $S_1 \cong_P S_1'$ by the induction hypothesis. Finaly, we prove $S \cong_P S_1$ and $S' \cong_P S_1'$ by showing that the functional interpretation of each of the flow edges η_i are equal according to the polymonad laws, and conclude $S \cong_P S'$ by transitivity.



5 Simplification and solving

Before running a program, we must solve the constraints produced during type inference, and apply the appropriate evidence for these constraints in the elaborated program. We also perform *simplification* on constraints prior to generalization to make types easier to read, but without compromising their utility.

A simple syntactic transformation on constraints can make inferred types easier to read. For example, we can hide duplicate constraints, identity morphisms (which are trivially satisfiable), and constraints that are entailed by the signature. More substantially, we can find instantiations for open variables in a constraint set before generalizing a type (and at the top-level, before running a program). To do this, we introduce below a modified version of (TS-Let) (from Figure 6); a similar modification is possible for (TS-Rec).

$$\frac{\begin{array}{cc} P_1 \,|\, \Gamma \vdash v : \tau \rightsquigarrow \mathsf{e}_1 & \bar{\rho}, \bar{a} = \mathsf{ftv}(P_1 \Rightarrow \tau) \setminus \mathsf{ftv}(\Gamma) \\ P_1 \xrightarrow{\text{simplify}(\bar{\rho} \setminus \mathsf{ftv}(\tau))} \theta \quad (\sigma, \mathsf{e}_2) = Gen(\Gamma, \theta P_1 \Rightarrow \tau, \mathsf{e}_1) & P \,|\, \Gamma, x{:}\sigma \vdash e : m\ \tau' \rightsquigarrow \mathsf{e}_3 \end{array}}{P \,|\, \Gamma \vdash \mathsf{let}\ x{=}v\ \mathsf{in}\ e : m\ \tau' \rightsquigarrow \mathsf{let}\ x{=}\mathsf{e}_2\ \mathsf{in}\ \mathsf{e}_3}$$

This rule employs the judgment $P \xrightarrow{\text{simplify}(\bar{\rho})} \theta$, defined in Figure 8, to simplify constraints by eliminating some open variables in P (via the substitution θ)

$$
\text{S-}\Uparrow\,\frac{\begin{array}{c}\pi = (\mathsf{Id},\mathsf{m}) \rhd \rho \;\vee\; \pi = (\mathsf{m},\mathsf{Id}) \rhd \rho \\ \rho \in \bar{\rho} \qquad \mathit{flowsFrom}_{P,P'}\ \rho \neq \{\} \\ \mathit{flowsTo}_{P,P'}\ \rho = \{\}\end{array}}{P, \pi, P' \xrightarrow{\text{simplify}(\bar{\rho})} \rho \mapsto m}
\qquad
\text{S-}\Downarrow\,\frac{\begin{array}{c}\pi = (\mathsf{Id},\rho) \rhd \mathsf{m} \;\vee\; \pi = (\rho,\mathsf{Id}) \rhd \mathsf{m} \\ \rho \in \bar{\rho} \qquad \mathit{flowsFrom}_{P,P'}\ \rho = \{\} \\ \mathit{flowsTo}_{P,P'}\ \rho \neq \{\}\end{array}}{P, \pi, P' \xrightarrow{\text{simplify}(\bar{\rho})} \rho \mapsto m}
$$

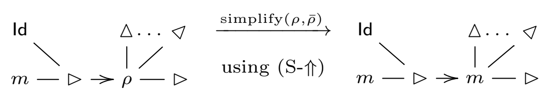
$$
\text{S-}\sqcup\,\frac{\begin{array}{c}F = \mathit{flowsTo}_P\ \rho \\ m \in F \Rightarrow m = \mathrm{M} \\ \text{for some M}\end{array}}{P \xrightarrow{\text{simplify}(\bar{\rho})} \rho \mapsto \bigsqcup F}
\qquad
\frac{\begin{array}{c}P \xrightarrow{\text{simplify}(\bar{\rho})} \theta \\ \theta P \xrightarrow{\text{simplify}(\bar{\rho})} \theta'\end{array}}{P \xrightarrow{\text{simplify}(\bar{\rho})} \theta'\theta}
\qquad
\frac{}{P \xrightarrow{\text{simplify}(\bar{\rho})} \cdot}
$$

$$
\text{where}\quad \begin{array}{l} \mathit{flowsTo}_P\ \rho \quad\;\; = \{\, (m_1, m_2) \mid (\mathsf{m}_1, \mathsf{m}_2) \rhd \rho \in P \,\} \\ \mathit{flowsFrom}_P\ \rho = \{\, m \mid \exists m'.\ \ \pi \in P \ \wedge\ (\pi = (\rho, \mathsf{m}') \rhd \mathsf{m} \ \vee\ \pi = (\mathsf{m}', \rho) \rhd \mathsf{m}) \,\} \end{array}
$$

Fig. 8. Eliminating open variables in constraints

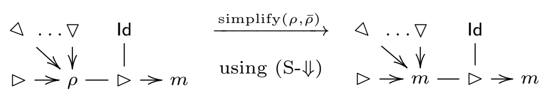
before type generalization. There are three main rules in the judgment (S-$\Uparrow$), (S-$\Downarrow$) and (S-$\sqcup$), while the last two simply take the transitive closure.

Rule (S-$\Uparrow$) solves monad variable ρ with monad m for the depicted situation:

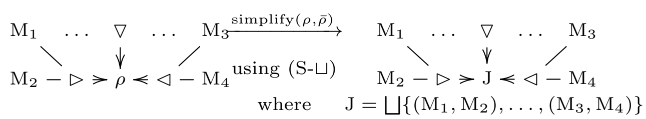


Here, we have a constraint $\pi = (\mathsf{Id}, \mathsf{m}) \rhd \rho$, where the only edges directed inwards to ρ are from Id and m, although there may be many out-edges from ρ. (The case where $\pi = (\mathsf{m}, \mathsf{Id}) \rhd \rho$ is symmetric.) Such a constraint can always be solved without loss of generality using an identity morphism, which, by the polymonad laws is guaranteed to exist. Moreover, for a closed polymonad, any solution to the constraints that chooses $\rho = m'$, for some $m' \neq m$ could also have just as well have chosen $\rho = m$. Thus, this rule does not impact solvability of the costraints.

Rule S-$\Downarrow$ follows similar reasoning, but in the reverse direction:



Finally, we have the rule (S-$\sqcup$), which exploits the properties of a principal relational polymonad:



Here we have a variable ρ such that all its in-edges are from pairs of ground constructors M_i, so we can simply apply the join function to compute a solution for ρ. For a closed principal relational polymonad, if such a solution exists, this simplification does not impact solvability of the rest of the constraint graph.

Example. Recall the information flow example we gave in Section 3.1, in Figure 3. Its principal type is the following, which is hardly readable:

$$\begin{array}{l}\forall \bar{\rho}_i, a_1, a_2.P_0 \Rightarrow \mathit{intref}\ a_1 \to \mathit{intref}\ a_2 \to \rho_{27}\ ()\\ \text{where } P_0 = (\mathsf{Id}, \rho_3) \rhd \rho_2, (\mathsf{Id}, \mathsf{IST}\ H\ \mathsf{a}_2) \rhd \rho_3, (\rho_{26}, \mathsf{Id}) \rhd \rho_4, (\mathsf{Id}, \mathsf{Id}) \rhd \rho_4,\\ \qquad (\rho_8, \rho_4) \rhd \rho_6, (\mathsf{Id}, \rho_9) \rhd \rho_8, (\mathsf{Id}, \mathsf{Id}) \rhd \rho_9, (\rho_{11}, \rho_{25}) \rhd \rho_{26},\\ \qquad (\mathsf{Id}, \rho_{12}) \rhd \rho_{11}, (\mathsf{Id}, \mathsf{IST}\ H\ \mathsf{a}_1) \rhd \rho_{12}, (\rho_{17}, \rho_{23}) \rhd \rho_{25}, (\rho_{14}, \rho_{18}) \rhd \rho_{17},\\ \qquad (\mathsf{Id}, \mathsf{Id}) \rhd \rho_{18}, (\mathsf{Id}, \rho_{15}) \rhd \rho_{14}, (\mathsf{Id}, \mathsf{Id}) \rhd \rho_{15}, (\rho_{20}, \rho_{24}) \rhd \rho_{23},\\ \qquad (\mathsf{Id}, \mathsf{IST}\ \mathsf{a}_1\ L) \rhd \rho_{24}, (\mathsf{Id}, \rho_{21}) \rhd \rho_{20}, (\mathsf{Id}, \mathsf{Id}) \rhd \rho_{21}.\end{array}$$

After applying (S-$\Uparrow$) and (S-$\Downarrow$) several times, and then hiding redundant constraints, we simplify P_0 to P which contains only three constraints. If we had fixed a_1 and a_2 (the labels of the function parameters) to H and L, respectively, we could do even better. The three constraints would be $(\mathsf{IST}\, H\, L, \rho_6) \rhd \rho_{27}, (\mathsf{Id}, \mathsf{Id}) \rhd \rho_6, (\mathsf{IST}\, H\, H, \mathsf{IST}\, H\, L) \rhd \rho_6$. Then, applying (S-$\sqcup$) to ρ_6 we would get $\rho_6 \mapsto \mathsf{IST}\, H\, H$, which when applied to the other constraints leaves only $(\mathsf{IST}\, H\, L, \mathsf{IST}\, H\, H) \rhd \rho_{27}$, which cannot be simplified further, since ρ_{27} appears in the result type.

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 2 (Simplification improves types). *For a closed, principal relational polymonad, given σ and σ' where σ is $\forall \bar{a}\bar{\rho}.P \Rightarrow \tau$ and σ' is an* improvement *of σ, having form $\forall \bar{a}'\bar{\rho}'.\theta P \Rightarrow \tau$ where $P \xrightarrow{\mathit{simplify}(\bar{\rho})} \theta$ and $\bar{a}'\bar{\rho}' = (\bar{a}\bar{\rho}) - \mathit{dom}(\theta)$. Then for all $P'', \Gamma, x, e, m, \tau$, if $P'' \,|\, \Gamma, x{:}\sigma \vdash e : m\,\tau$ such that $\models P''$ then there exists some P''' such that $P''' \,|\, \Gamma, x{:}\sigma' \vdash e : m\,\tau$ and $\models P'''$.*

Note that our $\xrightarrow{\text{simplify}(\bar{\rho})}$ relation is non-deterministic in the way it picks constraints to analyze, and also in the order in which rules are applied. In practice, for an acyclic constraint graph, one could consider nodes in the graph in topological order and, say, apply (S-$\sqcup$) first, since, if it succeeds, it eliminates a variable. For principal relational polymonads and acyclic constraint graphs, this process would always terminate.

However, if unification constraints induce cycles in the constraint graph, simply computing joins as solutions to internal variables may not work. This should not come as a surprise. In general, finding solutions to arbitrary polymonadic constraints is undecidable, since, in the limit, they can be used to encode the correctness of programs with general recursion. Nevertheless, simple heuristics such as unrolling cycles in the constraint graph a few times may provide good mileage, as would the use of domain-specific solvers for particular polymonads, and such approaches are justified by our coherence proof.

6 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 [24], 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 [8] 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 [15] provides a type class for bind-like operators that have a signature resembling our $(m_1, m_2) \rhd m_3$. One key limitation is that Kmett's binds must be *functionally dependent*: 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 $(m_1, \mathsf{Id}) \rhd m_2$ and $(m_1, \mathsf{Id}) \rhd m_3$ 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. Kmett also does not permit polymorphic units—he requires units into Id, which may later be lifted. But this only works for first-order code before running afoul of Haskell's ambiguity restriction. Polymonads do not have either 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 type class 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 type class) would frequently lead to open type variables, which Haskell's type class inference deems ambiguous. Inference with Kmett's class has the same problems. For ML, we have already discussed our own prior work [22] in detail throughout the paper.

Our work was carried out concurrently with Tate's work on *productors*, which describe the sequential composition of effects [23]. We have proved that polymonads are a *productoid*, and that with some mild restrictions, all productoids are polymonads. Tate proves a coherence result in a first-order imperative setting establishing that sequential compositions of productor joins are fully associative. Our coherence result is different in that it proves that all well-typed elaborations of a higher-order program with branching, function application, and recursion have the same semantics. Tate also does not address type inference.

7 Conclusions

This paper has presented *polymonads*, a generalization of monads and morphisms. Polymonads, by virtue of their relationship to Tate's *productoids*, are extremely powerful, subsuming monads, parameterized monads, and several other interesting constructions. Thanks to supporting algorithms for (principal) type inference, (provably coherent) elaboration, and (generality-preserving) simplification, this power comes with strong supports for the programmer. Like monads before them, we believe polymonads can become a useful and important element in the functional programmer's toolkit.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL*, 1999.
2. R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3 & 4):335–376, 2009.
3. G. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, 2006.
4. K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *J. Funct. Program.*, 15(02):249–291, 2005.
5. D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *TLDI*, 2011.
6. C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, 1997.
7. A. Filinski. Representing layered monads. In *POPL*, pages 175–188, 1999.
8. J.-C. Filliâtre. A theory of monads parameterized by effects, 1999.
9. J. Goguen and J. Meseguer. Security policy and security models. In *Symposium on Security and Privacy*, 1982.
10. G. Hutton and E. Meijer. Monadic Parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.
11. M. P. Jones. A theory of qualified types. In *ESOP*, 1992.
12. M. P. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, Sept. 1993.
13. M. P. Jones. Simplifying and Improving Qualified Types. Technical Report YALEU/DCS/RR-1040, Yale University, June 1994.
14. O. Kiselyov and C. Shan. Lightweight monadic regions. In *Haskell Symposium*, 2008.
15. E. Kmett. `Control.Monad.Parameterized` package. On `Hackage` repository, 2012.
16. P. Li and S. Zdancewic. Encoding information flow in Haskell. In *CSFW*, 2006.
17. E. Moggi. Computational lambda-calculus and monads. In *LICS*, 1989.
18. 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.
19. R. Pucella and J. Tov. Haskell session types with (almost) no class. In *Haskell Symposium*, 2008.
20. N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, 2002.
21. A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell*, 2008.
22. N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP*, 2011.
23. R. Tate. The sequential semantics of producer effect systems. In *POPL*, 2013.
24. P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4:1–32, 2003.

A Polymonads are productoids and vice versa

Given a polymonad $(\mathcal{M}, \Sigma)$, we can construct a 4-tuple $(\mathcal{M}, U, L, B)$ as follows:

(Units) $U = \{(\lambda x.\mathsf{bind}\ x\ (\lambda y.y))\colon \mathsf{a} \to \mathsf{M\ a} \mid \mathsf{bind}\colon (\mathsf{Id},\mathsf{Id}) \rhd \mathsf{M} \in \Sigma\}$,
(Lifts) $L = \{(\lambda x.\mathsf{bind}\ x\ (\lambda y.y))\colon \mathsf{M\ a} \to \mathsf{N\ a} \mid \mathsf{bind}\colon \mathsf{M} \hookrightarrow \mathsf{N} \in \Sigma\}$,
(Binds) The set $B = \Sigma - \{\mathsf{bind} \mid \mathsf{bind}\colon (\mathsf{Id},\mathsf{Id}) \rhd \mathsf{M}$ or $\mathsf{bind}\colon (\mathsf{M},\mathsf{Id}) \rhd \mathsf{N} \in \Sigma\}$.

It is fairly easy to show that the above structure satisfies generalizations of the familiar laws for monads and monad morphisms.

Theorem 4. *Given a polymonad $(\mathcal{M},\Sigma)$, the induced 4-tuple $(\mathcal{M},U,L,B)$ satisfies the following properties.*

(Left unit) $\forall \mathsf{unit} \in U, \mathsf{bind} \in B.$ *if* $\mathsf{unit}\colon \forall \mathsf{a}.\ \mathsf{a} \to \mathsf{M\ a}$ *and* $\mathsf{bind}{:}(\mathsf{M},\mathsf{N}) \rhd \mathsf{N}$ *then* $\mathsf{bind}\ (\mathsf{unit}\ e)\ f = f(e)$ *where* $e\colon \tau$ *and* $f\colon \tau \to \mathsf{N}\ \tau'$.
(Right unit) $\forall \mathsf{unit} \in U, \mathsf{bind} \in B.$ *if* $\mathsf{unit}\colon \forall \mathsf{a}.\ \mathsf{a} \to \mathsf{N\ a}$ *and* $\mathsf{bind}{:}(\mathsf{M},\mathsf{N}) \rhd \mathsf{M}$ *then* $\mathsf{bind}\ m\ (\mathsf{unit}) = m$ *where* $m\colon \mathsf{M}\ \tau$.
(Associativity) $\forall \mathsf{bind}_1, \mathsf{bind}_2, \mathsf{bind}_3, \mathsf{bind}_4 \in B.$ *if* $\mathsf{bind}_1 : (\mathsf{M},\mathsf{N}) \rhd \mathsf{P}$, $\mathsf{bind}_2 : (\mathsf{P},\mathsf{R}) \rhd \mathsf{T}$, $\mathsf{bind}_3 : (\mathsf{M},\mathsf{S}) \rhd \mathsf{T}$, *and* $\mathsf{bind}_4 : (\mathsf{N},\mathsf{R}) \rhd \mathsf{S}$ *then* $\mathsf{bind}_2\ (\mathsf{bind}_1\ m\ f)\ g = \mathsf{bind}_3\ m\ (\lambda x.\mathsf{bind}_4\ (f\ x)\ g)$ *where* $m\colon \mathsf{M}\ \tau$, $f\colon \tau \to \mathsf{N}\ \tau'$ *and* $g\colon \tau' \to \mathsf{R}\ \tau''$
(Morphism 1) $\forall \mathsf{unit}_1, \mathsf{unit}_2 \in U, \mathsf{lift} \in L.$ *if* $\mathsf{unit}_1\colon \forall \mathsf{a}.\ \mathsf{a} \to \mathsf{M\ a}$, $\mathsf{unit}_2\colon \forall \mathsf{a}.\ \mathsf{a} \to \mathsf{N\ a}$ *and* $\mathsf{lift}\colon \forall \mathsf{a}.\ \mathsf{M\ a} \to \mathsf{N\ a}$ *then* $\mathsf{lift}\ (\mathsf{unit}_1\ e) = \mathsf{unit}_2\ e$ *where* $e\colon \tau$.
(Morphism 2) $\forall \mathsf{bind}_1, \mathsf{bind}_2 \in B, \mathsf{lift}_1, \mathsf{lift}_2, \mathsf{lift}_3 \in L.$ *if* $\mathsf{bind}_1 : (\mathsf{M},\mathsf{P}) \rhd \mathsf{S}$, $\mathsf{bind}_2 : (\mathsf{N},\mathsf{Q}) \rhd \mathsf{T}$, $\mathsf{lift}_1\colon \forall \mathsf{a}.\ \mathsf{M\ a} \to \mathsf{N\ a}$, $\mathsf{lift}_2\colon \forall \mathsf{a}.\ \mathsf{P\ a} \to \mathsf{Q\ a}$ *and* $\mathsf{lift}_3\colon \forall \mathsf{a}.\ \mathsf{S\ a} \to \mathsf{T\ a}$ *then* $\mathsf{lift}_3\ (\mathsf{bind}_1\ m\ f) = \mathsf{bind}_2\ (\mathsf{lift}_1\ m)\ (\lambda x.\mathsf{lift}_2\ (f\ x))$ *where* $m\colon \mathsf{M}\ \tau$ *and* $f\colon \tau \to \mathsf{P}\ \tau'$.

Now we show how this definition can be used to relate polymonads to Tate's *productoids* [23]. The definition of a productoid is driven by an underlying algebraic structure: the effectoid [23, Theorem 1]. An effectoid $(E, U, \leq, \mapsto)$ is a set E, with an identified subset $U \subseteq E$ and relations $\leq \subseteq E \times E$ and $(\_;\_) \mapsto \_ \subseteq E \times E \times E$, that satisfies six monoid-like conditions. It is possible to show that a polymonad directly induces an effectoid structure and hence a productoid.

Lemma 3. *Given a polymonad $(\mathcal{M},U,L,B)$ we can define an effectoid $(E, U, \leq, (\_;\_) \mapsto \_)$ as follows.*

$$E = \mathcal{M} \qquad U = \{\mathsf{M} \mid \mathsf{unit}\colon \mathsf{a} \to \mathsf{M\ a} \in U\}$$
$$\leq\ = \{(\mathsf{M},\mathsf{N}) \mid \mathsf{lift}\colon \mathsf{M\ a} \to \mathsf{N\ a} \in L\} \qquad (\_;\_) \mapsto \_\ = \{(\mathsf{M},\mathsf{N},\mathsf{P}) \mid (\mathsf{M},\mathsf{N}) \rhd \mathsf{P} \in B\}$$

Lemma 4. *Every polymonad gives rise to a productoid.*

Proof. We have shown that a polymonad gives rise to an effectoid. Given an effectoid $(E, U, \leq, (\_;\_) \mapsto \_)$ a productoid is defined as a collection of functors indexed by the collection E, and three collections of natural transformations indexed by the three relations. These functors and natural transformations are required to satisfy five addition properties [23, Theorem 2]. The five properties are the five properties of Theorem 4, so the proof is immediate.

Interestingly, we can identify conditions where the opposite direction also holds.

Lemma 5. *A productoid* $(\mathbf{C}, \{F_e \colon \mathbf{C} \to \mathbf{C}\}_{e \in E}, \{\eta \colon 1 \Rightarrow F_e\}_{e \in U}, \{\mu \colon F_{e_1} \circ F_{e_2} \Rightarrow F_{e_3}\}_{(e_1;e_2) \mapsto e_3}, \{\sigma \colon F_{e_1} \Rightarrow F_{e_2}\}_{e_1 \leq e_2})$ *that in addition satisfies the following conditions gives rise to a polymonad.*

1. $\mathsf{Id} \in E$ *and* $F_{\mathsf{Id}} = 1$
2. $\mathsf{Id} \in U$
3. *For all* $e \in E$, $(e;\mathsf{Id}) \mapsto e$
4. *For all* $e \in E, \mu \colon F_e \circ 1 \Rightarrow F_e = \mathsf{Id}$
5. $(e_1;e_2) \mapsto e \ \wedge \ e_1' \leq e_1 \ \Rightarrow \ (e_1';e_2) \mapsto e$
6. $(e_1;e_2) \mapsto e \ \wedge \ e_2' \leq e_2 \ \Rightarrow \ (e_1;e_2') \mapsto e$

These additional conditions are fairly mild: (1)-(4) simply ensure that the Id element is interpreted as the identity functor. Conditions (5)-(6) are also quite straightforward; certainly if the category is cartesian closed then the extra natural transformations are always defined.