Lightweight Monadic Programming in ML

Nikhil Swamy[⋆] Nataliya Guts[†] Daan Leijen[⋆] Michael Hicks[†]

[⋆]Microsoft Research, Redmond [†]University of Maryland, College Park

Abstract

Many useful programming constructions can be expressed as monads. Examples include probabilistic modeling, functional reactive programming, parsing, and information flow tracking, not to mention effectful functionality like state and I/O. In this paper, we present a type-based rewriting algorithm to make programming with arbitrary monads as easy as using ML's built-in support for state and I/O. Developers write programs using monadic values of type $m\ \tau$ as if they were of type τ, and our algorithm inserts the necessary binds, units, and monad-to-monad morphisms so that the program type checks. Our algorithm, based on Jones' qualified types, produces principal types. But principal types are sometimes problematic: the program's semantics could depend on the choice of instantiation when more than one instantiation is valid. In such situations we are able to simplify the types to remove any ambiguity but without adversely affecting typability; thus we can accept strictly more programs. Moreover, we have proved that this simplification is *efficient* (linear in the number of constraints) and *coherent*: while our algorithm induces a particular rewriting, all related rewritings will have the same semantics. We have implemented our approach for a core functional language and applied it successfully to simple examples from the domains listed above, which are used as illustrations throughout the paper.

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages,Theory

Keywords monad, type, rewriting, coherence, coercion

1. Introduction

The research literature abounds with useful programming constructions that can be expressed as monads, which consist of a type constructor m and two operations, *bind* and *unit*:[1]

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

Example monads include parsers [13], probabilistic computations [25], functional reactivity [3, 8], and information flow tracking [26]. In a monadic type system, if *values* are given type τ then *computations* are given type $m\ \tau$ for some monad constructor m. For example, an expression of type $IO\ \ \tau$ in Haskell represents a computation that will produce (if it terminates) a value of type τ but may perform effectful operations in the process. Haskell's `Monad` type class, which requires the `bind` and `unit` operations given above, is blessed with special syntax, the *do notation*, for programming with instances of this class.

Moggi [22], Filinksi [11], and others have noted that ML programs, which are impure and observe a deterministic, call-by-value evaluation order, are inherently monadic. For example, the value $\lambda x.e$ can be viewed as having type $\tau\ \rightarrow\ m\ \ \tau'$: the argument type τ is never monadic because x is always bound to a value in e, whereas the return type is monadic because the function, when applied, produces a computation. As such, call-by-value application and let-binding essentially employ monadic sequencing, but the monad constructor m and the `bind` and `unit` combinators for sequencing are implicit rather than explicit. In essence, the explicit IO monad in Haskell is implicit in ML.

While programming with I/O in ML is lightweight, programming with other monads is not. For example, suppose we are interested in programming *behaviors*, which are time-varying values, as in *functional reactive programs* [3, 8]. With the following notation we indicate that behaviors can be implemented as a monad: expressions of type $Beh\ \ \alpha$ represent values of type α that change with time, and `bindp` and `unitp` are its monadic operations:

$$\mathsf{Monad}(Beh, \texttt{bindb}, \texttt{unitb})$$

As a primitive, function `seconds` has type $unit\ \rightarrow\ Beh\ \ int$, its result representing the current time in seconds since the epoch. An ML program using Beh effectively has two monads: the implicit monad, which applies to normal ML computations, and the user-defined monad Beh. The former is handled primitively but the latter requires the programmer to explicitly use `bindb`, `unitb`, function composition, etc., as in the following example:

```
bindb (bindb (seconds()) (fun s-> unitb (is_even s)))
  (fun y-> unitb (if y then 1 else 2))
```

The type of this entire expression is $Beh\ \ int$: it is time-varying, oscillating between values 1 and 2 every second.

Instead of using tedious explicit syntax, we would like to overload the existing syntax so that monadic constructs are implicit, e.g., as in the following program (call it Q)

```
let y = is_even (seconds()) in
if y then 1 else 2
```

We can see that the programs are structurally related, with a bind corresponding to each let and application, and unit applied to the bound and final expressions.

While ML programming with one monad is tedious, programming with more than one is worse. Along with different binds and units for each monad, the programmer may have to insert calls to

[1] These operations must respect certain laws; cf. Wadler [29] for details.

ICFP'11, September 19–21, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0865-6/11/09. . . $10.00

morphisms, which are functions that lift one monad into another. For example, suppose that along with time-varying expressions like `seconds()` we allowed time-varying probability distributions expressed as a monad $\mathit{BehPrb}\ \alpha$ (we show an example of this in Section 2). Given a morphism from the first monad to the second, i.e., from a time-varying value to a time-varying probability distribution, the programmer must insert calls to it in the right places.

This paper presents an algorithm for automatically converting an ML program, like Q, which makes implicit use of monads, into one that, like P, makes explicit use of monads, with binds, units, and morphisms inserted where needed. Our algorithm operates as part of polymorphic type inference, following an approach similar to Jones [14], which is the foundation of Haskell's type class inference. We use the syntactic structure of the program to identify where binds, units, and morphisms may appear. Type inference introduces a fresh variable for the monad being used at these various points and, as in Jones, we produce qualified types of the form $\forall\bar{\nu}.P \Rightarrow \tau$, where $\bar{\nu}$ contains free type and monad variables, and P contains morphism constraints. We prove our algorithm produces principal types.

As it turns out, our basic scheme of inferring principal types could be encoded within Haskell's type class inference algorithm, e.g., one could define morphisms as instances of a `Morphism` type class and then rely on inference to insert them as necessary. However, this algorithm would reject many useful programs as potentially ambiguous, in particular, those whose types contain quantified variables that appear only in the constraints P but not in the type τ. In the general setting of Haskell, this is sensible, as different instantiations of these variables could produce programs with different semantics. By focusing our attention on monads we sidestep this problem. We can exploit laws governing the behavior of morphisms to ensure that all potential instantiations of such variables are *coherent*; i.e., the program semantics will not change with different instantiations. As such, our algorithm employs a linear-time strategy to eliminate such variables, thus simplifying the constraints P with no adverse effects on typability.

We have implemented our inference algorithm for a core functional language. We demonstrate its utility by applying it to example programs that use monads implementing behaviors, probabilistic computations, and parsers (cited above). We also develop an example using a family of monads for tracking information flows of high- and low-security data [7, 26]. We prove that our rewriting algorithm produces programs accepted by FlowCaml, a dialect of ML that performs information flow tracking [24], and thereby show that rewritten programs are secure (enjoy noninterference [12]).

In summary, this paper presents an extension to ML for monadic programming, providing the following benefits:

- Developers can define their own monads and program with them in a direct style, avoiding the tedium of introducing binds, units, and morphisms manually.
- Monadic operations are inserted automatically in conjunction with a novel type inference algorithm. Our algorithm produces qualified types which describe the monads and morphisms used by a piece of code, usefully revealing the monadic structure of the program to the developer. Exploiting the morphism laws, we efficiently infer general types and produce coherent rewritings.
- The system is quite flexible: we have shown, via our prototype implementation, that it supports many varieties of monads and their combination.

The next section presents an overview of our approach; our main technical results are contained in Sections 3–5; and applications and related work discussed in Sections 6 and 7, respectively.

2. Overview

This section presents an overview of our approach through the development of a few examples. We start by considering a single user-defined monad and see that this is relatively easy to handle. Then, we look at handling multiple monads, which require the use of morphisms. With these examples we illustrate our type inference and rewriting algorithm and discuss its key properties.

2.1 Programming with a single user-defined monad

As mentioned earlier, pure ML computations may be seen as already employing an implicit monad, e.g., covering partiality, state, exceptions, and I/O. We call this implicit monad Bot, since it is the bottom element of our monad hierarchy; in effect, its unit operator is the identity function, and bind is the reverse application. Our goal is to exploit the inherent monadic structure of ML to provide support for programming with multiple, user-defined monads with the same ease as programming, implicitly, with Bot. Of course, we aim to do this without breaking compatibility with ML.

Let us illustrate our idea on an example using the probability monad [25]. Expressions of type $\mathit{Prb}\ \alpha$ describe distributions over values of type α, with `bindp` and `unitp` as its bind and unit combinators, respectively:

$$\mathsf{Monad}(\mathit{Prb}, \mathtt{bindp}, \mathtt{unitp})$$

The probability monad can be used to define probabilistic models. The following program P, based on the classic example due to Pearl [23], is an example of model code we would like to write:

```
let rain = flip .5 in
let sprinkler = flip .3 in
let chance = flip .9 in
let grass_is_wet = (rain || sprinkler) && chance in
if grass_is_wet then rain else fail ()
```

This program uses two functions it does not define:

`flip` : $\mathit{float} \to \mathit{Prb}\ \mathit{bool}$
`fail` : $\forall\alpha.\mathit{unit} \to \mathit{Prb}\ \alpha$

The first introduces distributions: `flip`(p) is a distribution where `true` has probability p and `false` has probability $1-p$. The second, `fail`, represents impossibility.

The first four lines of P define four random variables: `rain` is true when it is raining; `sprinkler` is true when the sprinkler is running; `chance` is explained below; and `grass_is_wet` is true when the grass is wet. The probability distribution for the last is dependent on the distributions of the first three: the grass is wet if either it is raining or the sprinkler is running, with an additional bit of uncertainty due to `chance`: e.g., even with rain, grass under a tree might be dry. The last line of P implements a conditional distribution; i.e., the probability that it is raining given that the grass is wet. Mathematically, this would be represented with notation Pr(`rain` | `grass_is_wet`).

Unfortunately, in ML we cannot write the above code directly because it is not type-correct. For example, the expression `rain || sprinkler` applies the `||` function, which has type $\mathit{bool} \to \mathit{bool} \to \mathit{bool}$, to `rain` and `sprinkler`, which each have type $\mathit{Prb}\ \mathit{bool}$. Fortunately, our system will automatically rewrite P so that it is type-correct, producing the code given below.

```
bindp (flip .5) (fun rain->
bindp (flip .3) (fun sprinkler->
bindp (flip .9) (fun chance->
bindp (unitp ((rain || sprinkler) && chance))
  (fun grass_is_wet->
if grass_is_wet then unitp rain else fail ()))))
```

When there is only one user-defined monad to consider, a rewriting such as this one is entirely syntactic. Roughly, each `let` is replaced by a `bindp` and each let-bound expression that is not already monadic is wrapped with `unitp`. By doing so, we keep to the monadic structure of sequencing implemented primitively by ML for its Bot monad, except we have now interposed the Prb monad. Although not shown above, function applications are handled similarly: we insert `bindp` on both the left- and right-hand sides, thereby echoing the call-by-value semantics of function application in ML.

Under this rewriting semantics, we can give a type to the original program even though it is not typable in ML—our algorithm infers the type $Prb\ bool$ for the source program. The types we infer always have a particular structure that mirrors the monadic structure of ML. As in Moggi's computational lambda calculus, we note that inferred function types always have a monadic type in their co-domain and monadic types never appear to the left of arrows. This corresponds to the following intuition. Since values are always pure, under a call-by-value semantics, function arguments must always be effect free (hence, no monadic types to the left of arrows). Furthermore, since in ML functions can have arbitrary effects, their co-domains are always monadic.

2.2 Programming with multiple monads

Now suppose we wish to program with both probabilities and behaviors (introduced in Section 1). Perhaps we would like the probability of rain to change with time, e.g., according to the seasons. Then we can modify P (call it P') so that the argument to `flip` is a function `rainprb` of type $unit \rightarrow Beh\ float$:

```
let rain = flip (rainprb ()) in ...
```

Again, this program fragment is ill-typed, because `flip` expects a *float* but we have passed it a *Beh float*.

If our rewriting system is to be applied, what should be the type of `rain`? One might expect it to be $Beh\ (Prb\ bool)$, since it is a time-varying distribution. However, this type is nonsensical. Just as ML does not support data structures containing non-values (e.g., those having type $List\ (Bot\ \alpha)$) it does not support computations parameterized by non-values, e.g., expressions of type $Bot\ (Bot\ \alpha)$ and, for that matter, type $Beh\ (Prb\ \alpha)$. Therefore, a programmer must construct a combined monad, $BehPrb$, along with morphisms from the individual monads into the combined one, to ensure that the overall program's semantics makes sense. There are several standard techniques for combining monads [17]; here, we can combine them by defining objects of type $BehPrb\ \tau$ as a stream of distributions over τ, with the obvious morphisms.

$$\begin{array}{l}\mathsf{Monad}(BehPrb, \mathtt{bindbp}, \mathtt{unitbp}) \\ \mathtt{p2bp} : Prb \rhd BehPrb \\ \mathtt{b2bp} : Beh \rhd BehPrb\end{array}$$

To allow automatic type-directed insertion of morphisms, we assume that there is at most one morphism between each pair of monads. In general, a morphism $f_{1,2} : m_1 \rhd m_2$ has the type $\forall\alpha.m_1\ \alpha \rightarrow m_2\ \alpha$.[2] (We implicitly consider the unit operation of monad m as the morphism $Bot \rhd m$.) Given the above morphisms, our system rewrites P' thus:

```
bindpb (bindpb (b2bp (rainprb ()))
         (fun v1-> p2bp (flip v1))) (fun rain->
  p2bp (bindp (flip .3) (fun sprinkler-> ...
```

where `...` is identical to the corresponding part of the rewriting for P, and the final type is $BehPrb\ bool$. Here, the result of `rainprb()` is lifted into *BehPrb float* and then bound to the current value `v1`, which is passed to `flip` to generate a distribution. This value is in turn lifted into $BehPrb\ bool$ and bound to boolean `rain` for the rest of the computation, whose result, of type $Prb\ bool$, is lifted into $BehPrb\ bool$ by application of `p2bp`.

2.3 Properties of type inference and rewriting

Our examples so far have involved inserting known morphisms, binds, and units, producing monomorphic types. But our system is more general in that we can rewrite a function to abstract the monads and morphisms it uses. For such functions our algorithm infers *qualified* types of the form $\forall\bar{\nu}.P \Rightarrow \tau$, where P is a set of constraints $m_1 \rhd m_2$ where the m_i could be constant, known monads, like Beh, or abstracted, unknown monads μ that appear in the bound type variables $\bar{\nu}$ (along with the usual type variables α). For example, the type of

```
let compose f g x = f (g x)
```

inferred by our system is

$$\begin{array}{l}\forall\alpha\beta\gamma\mu_1\mu_2\mu. \\ \quad (\mu_1 \rhd \mu, \mu_2 \rhd \mu) \Rightarrow (\beta \rightarrow \mu_2\ \gamma) \rightarrow (\alpha \rightarrow \mu_1\ \beta) \rightarrow \alpha \rightarrow \mu\gamma\end{array}$$

(where two occurrences of Bot on arrow types have been elided for readability). The rewritten term will take as arguments a representation of the monads μ, μ_1, and μ_2, and two morphism functions corresponding to the two morphism constraints. Section 4 shows that our algorithm infers *principal types*, i.e. most general types.

By restricting the structure of inferred types (e.g., no monadic types to the left of arrows), and by providing only a limited form of subtyping, we obtain morphism constraints P that can be efficiently solved using a simple linear-time procedure. A solution of constraints allows us to instantiate the monadic operators and the morphisms in the elaborated term.

Last but not least, our algorithm enjoys *coherence*: any two rewritings of the same program are semantically equivalent. Said differently, choosing a particular solution does not affect the meaning of the program. Coherence allows us to accept programs that would otherwise be rejected as ambiguous by related systems that employ qualified types for type inference, e.g., Haskell's type-class mechanism. We achieve coherence by taking advantage of the following assumed properties of morphisms:

$$f_{1,2} \circ unit_1 = unit_2 \tag{1}$$

$$f_{1,2}\ (bind_1\ \mathsf{e}_1\ \mathsf{e}_2) = bind_2\ (f_{1,2}\ \mathsf{e}_1)\ (f_{1,2} \circ \mathsf{e}_2) \tag{2}$$

$$f_{2,3} \circ f_{1,2} = f_{1,3} \tag{3}$$

Properties (1–2) are the so-called *morphism laws*, and the third is the transitivity property.

For instance, the combination of Beh, Prb, and $BehPrb$ in the example above leads to constraints that admit several valid solutions. One of these solutions, shown below, directly lifts all the let bindings to the $BehPrb$ monad, instead of lifting parts of the computation to either the Beh or Prb monads.

```
bindbp (bindbp (b2bp (rainprb ()))
         (fun v1-> p2bp (flip v1))) (fun rain->
bindbp (p2bp (flip .3)) (fun sprinkler->
bindbp (p2bp (flip .9)) (fun chance->
bindbp (unitbp ((rain || sprinkler) && chance))
  (fun grass_is_wet->
bindpb (fail ()) (fun f->
unitpb (if_ grass_is_wet then rain else f))))))
```

Using the morphism laws, we can show that the two rewritings are equivalent. However we might argue that the first rewriting, produced by our algorithm, is more precise than this one; intuitively it applies morphisms "as late as possible" and uses the "simplest"

[2] Morphisms (just like the binds and units) are not part of the source program; as such, they are treated specially and not subject to the positivity condition on monadic types.

$$
\begin{array}{llll}
\text{types} & \tau & ::= & \alpha \\
 & & \mid & T\,\tau_1 ... \tau_n \qquad (n \geqslant 0 \text{ is the arity of } T) \\
 & & \mid & \tau_1 \rightarrow m\ \tau_2 \\
 & & & \\
\text{monadic types} & m & ::= & \mu \mid M \\
\text{type variables} & \nu & ::= & \alpha \mid \mu \\
 & & & \\
\text{constraints} & P & ::= & \pi_1, ..., \pi_n \\
\text{constraint} & \pi & ::= & m_1 \rhd m_2 \\
 & & & \\
\text{type schemes} & \sigma & ::= & \forall \bar{\nu}. P \Rightarrow \tau \\
\text{environment} & \Gamma & ::= & \cdot \mid \Gamma, c{:}\sigma \mid \Gamma, x{:}\sigma \\
 & & & \\
\text{values} & v & ::= & x \mid c \mid \lambda x.e \\
\text{expressions} & e & ::= & v \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

Figure 1. Core language syntax.

$$
P \models m \rhd m \ \ \text{(M-Taut)} \qquad \frac{m_1 \rhd m_2 \in P}{P \models m_1 \rhd m_2} \ \ \text{(M-Hyp)}
$$

$$
\frac{P \models m_1 \rhd m_2 \qquad P \models m_2 \rhd m_3}{P \models m_1 \rhd m_3} \ \ \text{(M-Trans)}
$$

$$
\frac{P \vdash \pi_1 \ ... \ P \vdash \pi_n}{P \vdash \pi_1, ..., \pi_n} \ \ \text{(M-Many)}
$$

Figure 2. The constraint entailment relation.

$$
\frac{\begin{array}{cc} \overline{\rho} = \overline{m}, \overline{\tau} & \theta = [\overline{\rho}/\overline{\nu_1}] \\ P, \bar{\pi}_2 \models \theta\bar{\pi}_1 & \bar{\nu}_2 \notin \mathsf{ftv}(\forall \bar{\nu}_1.\, \bar{\pi}_1 \Rightarrow \tau) \end{array}}{P \vdash \forall \bar{\nu}_1.\, \bar{\pi}_1 \Rightarrow \tau \ \geqslant \ \forall \bar{\nu}_2.\, \bar{\pi}_2 \Rightarrow \theta\tau} \ \ \text{(Inst)}
$$

Figure 3. The generic instance relation over type schemes.

monad as long as possible. As such, the types more precisely reveal the monads actually needed by a piece of code.

3. Qualified types for monadic programs

This section describes the formal type rules of our system. Figure 1 gives the syntax of types, constraints, environments, and expressions. Monotypes τ consist of type variables α, full applied type constructors $T\ \tau_1 ... \tau_n$, and function types $\tau_1 \rightarrow m\ \tau_2$. Function arrows can be seen as taking three arguments where the m is the monadic type. We could use a kind system to distinguish monadic types from regular types but, for simplicity, we distinguish them using different syntactic categories. Monadic types m are either monad constants M or monadic type variables μ.

Since types can be polymorphic over the actual monad (which is essential to principal types) we also have monadic constraints π of the form $m_1 \rhd m_2$, which states that a monad m_1 can be lifted to the monad m_2. Type schemes are the usual qualified types [14] where we can quantify over both regular and monadic type variables.

In the expression language, we distinguish between syntactic value expressions v, and regular expressions e. This is in order to impose the value restriction of ML where we can only generalize over let-bound values.

Figure 2 describes the structural rules of constraint entailment, where $P \models \pi$ states that the constraints in P entail the constraint π. The entailment relation is monotone (where $P' \subseteq P$ implies $P \models P'$), transitive, and closed under substitution. We also require that morphisms between the monads form a semi-lattice. This requirement is not essential for type inference but as shown in Section 5 it is necessary for a coherent evidence translation.

Using entailment, we define the generic instance relation $P \vdash \sigma_1 \geqslant \sigma_2$ in Figure 3. This is just the regular instance definition on type schemes where entailment is used over the constraints. In the common case where one instantiates to a monotype, the rule simplifies to:

$$
\frac{\overline{\rho} = \overline{m}, \overline{\tau} \qquad \theta = [\overline{\rho}/\overline{\nu}] \qquad P \models \theta\bar{\pi}}{P \vdash \forall \bar{\nu}.\, \bar{\pi} \Rightarrow \tau \ \geqslant \ \theta\tau} \ \ \text{(Inst-Mono)}
$$

3.1 Declarative type rules

Figure 4 describes the basic type rules of our system; we discuss rewriting in the next subsection. The rules come in two forms: the rule $P \mid \Gamma \vdash v : \sigma$ states that value expression v is well typed with a type σ, while the rule $P \mid \Gamma \vdash e : m\ \tau$ states that expression e is well-typed with a monadic type $m\ \tau$, in both cases assuming the constraints P and type environment Γ. The rule (TI-Bot) allows one to lift a regular type τ into a monadic type $Bot\ \tau$.

The rules for variables, constants, let-bound values, instantiation, and generalization are all standard. The rule for lambda expressions (TI-Lam) requires a monadic type in the premise to get well-formed function types. An expression like $\lambda x.x$ therefore gets type $\alpha \rightarrow Bot\ \alpha$ where the result is in the identity monad.

The application rule (TI-App) and let-rule (TI-Do) lift into an arbitrary result monad. The constraint $\forall i.\ P \models m_i \rhd m$ ensures that all the monads in the premise can be lifted to a common monad m, which allows a type-directed evidence translation to the underlying monadic program.

3.2 Type directed monadic translation

As described in Section 2, we rewrite a source program while performing type inference, inserting binds, units, and morphisms as needed. This translation can be elegantly described using a type directed evidence translation [14]. Since the translation is entirely standard, we elide the full rules, and only sketch how it is done. In Section 5 we do show the evidence translation for the type inference algorithm W since it is needed to show coherence.

Our elaborated target language is System F (but here we leave out type parameters for simplicity). For the declarative rules, we can define a judgment like $P \mid \Gamma \vdash e : m\ \tau \rightsquigarrow \mathsf{e}$ which proves that source term e is given monadic type $m\ \tau$ and elaborated to the well-typed output term e. Similarly, the entailment relation $P \models m_1 \rhd m_2 \rightsquigarrow f$ returns a morphism witness f with type $\forall \alpha.\, m_1\, \alpha \rightarrow m_2\, \alpha$.

As an example, consider the (TI-App) rule. The rule with a type directed translation is defined as:

$$
\frac{\begin{array}{c} P \mid \Gamma \vdash e_1 : m_1\ (\tau_2 \rightarrow m_3\ \tau) \rightsquigarrow \mathsf{e}_1 \quad P \mid \Gamma \vdash e_2 : m_2\ \tau_2 \rightsquigarrow \mathsf{e}_2 \\ \forall i. P \models m_i \rhd m \rightsquigarrow f_i \end{array}}{\begin{array}{c} P \mid \Gamma \vdash e_1\ e_2 : m\ \tau \rightsquigarrow \\ \mathit{bind}_m\ (f_1\, \mathsf{e}_1)\ (\lambda \mathsf{x}{:}(\tau_2 \rightarrow m_3\ \tau).\, \mathit{bind}_m\ (f_2\, \mathsf{e}_2)\ (\lambda \mathsf{y}{:}\tau_2.\, f_3\ (\mathsf{x}\, \mathsf{y}))) \end{array}}
$$

The bind_m evidence comes from the $\mathsf{Monad}(m, \mathit{bind}_m, \mathit{unit}_m)$ constraint. This constraint is left implicit in the type rules since it is always satisfied. Much of the time the morphisms are identity functions and the binding operations will be in the Bot monad which can all be optimized away. An optimizer can make further use of the monad laws to aggressively simplify the target terms.

3.3 Compatibility with ML

Figure 4 is backwards compatible with the ML type system: it accepts any program that is accepted by the standard Hindley-Milner typing rules [5] extended with the value restriction—we

$$\boxed{P\,|\,\Gamma \vdash v : \sigma \quad P\,|\,\Gamma \vdash e : m\ \tau} \qquad \frac{\Gamma(x) = \sigma}{P\,|\,\Gamma \vdash x : \sigma}\ (\text{TI-Var}) \qquad \frac{\Gamma(c) = \sigma}{P\,|\,\Gamma \vdash c : \sigma}\ (\text{TI-Const}) \qquad \frac{P\,|\,\Gamma, x{:}\tau_1 \vdash e : m\ \tau_2}{P\,|\,\Gamma \vdash \lambda x.e : \tau_1 \to m\ \tau_2}\ (\text{TI-Lam})$$

$$\frac{P\,|\,\Gamma \vdash v : \tau}{P\,|\,\Gamma \vdash v : Bot\ \tau}\ (\text{TI-Bot}) \qquad \frac{P\,|\,\Gamma \vdash v : \sigma \qquad P \vdash \sigma \geqslant \tau}{P\,|\,\Gamma \vdash v : \tau}\ (\text{TI-Inst}) \qquad \frac{P, \bar{\pi}\,|\,\Gamma \vdash v : \tau \qquad \bar{\nu} \notin \mathsf{ftv}(\Gamma, P)}{P\,|\,\Gamma \vdash v : \forall \bar{\nu}.\, \bar{\pi} \Rightarrow \tau}\ (\text{TI-Gen})$$

$$\frac{P\,|\,\Gamma \vdash e_1 : m_1\ (\tau_2 \to m_3\ \tau) \qquad P\,|\,\Gamma \vdash e_2 : m_2\ \tau_2 \qquad \forall i. P \models m_i \rhd m}{P\,|\,\Gamma \vdash e_1\ e_2 : m\ \tau}\ (\text{TI-App})$$

$$\frac{P\,|\,\Gamma \vdash v : \sigma \qquad P\,|\,\Gamma, x{:}\sigma \vdash e : m\ \tau}{P\,|\,\Gamma \vdash \mathsf{let}\ x{=}v\ \mathsf{in}\ e : m\ \tau}\ (\text{TI-Let}) \qquad \frac{P\,|\,\Gamma \vdash e_1 : m_1\ \tau_1 \qquad P\,|\,\Gamma, x{:}\tau_1 \vdash e_2 : m_2\ \tau_2 \qquad \forall i. P \models m_i \rhd m}{P\,|\,\Gamma \vdash \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : m\ \tau_2}\ (\text{TI-Do})$$

Figure 4. The basic declarative type rules.

$$\frac{P\,|\,\Gamma \vdash e : m\ \tau \qquad P \models m \rhd m'}{P\,|\,\Gamma \vdash e : m'\ \tau}\ (\text{TI-Lift})$$

Figure 5. The type rules extended with a lifting rule.

write an ML derivation as $\Gamma \vdash_{\text{ML}} e : \tau$. To compare the derivations in both systems, we need to translate regular ML function types to monadic function types, and we define $\langle\tau\rangle$ as:

$$\begin{array}{ll} \langle\alpha\rangle & = \alpha \\ \langle T\ \tau_1 \ldots \tau_n\rangle & = T\ \langle\tau_1\rangle \ldots \langle\tau_n\rangle \\ \langle\tau_1 \to \tau_2\rangle & = \langle\tau_1\rangle \to Bot\ \langle\tau_2\rangle \end{array}$$

We can state compatibility with ML formally as:

Theorem 1 (Compatibility with ML)**.** *For any well-typed ML non-value expression e such that $\Gamma \vdash_{\text{ML}} e : \tau$, we also have a valid monadic derivation in the Bot monad of the form $\emptyset\,|\,\Gamma \vdash e : Bot\ \langle\tau\rangle$. For any well-typed value v where $\Gamma \vdash_{\text{ML}} v : \tau$, we have a monadic derivation of the form $\emptyset\,|\,\Gamma \vdash v : \langle\tau\rangle$.*

The proof is by straightforward induction over typing derivations. We observe that for a standard ML program, we only need the Bot monad which means we can always reason under an empty constraint set $\emptyset$. Assuming empty constraints, the instance relation and generalization rule coincide exactly with the Hindley-Milner rules. The other rules now also correspond directly. We show the case for the App rule as an example. By the induction hypothesis, we can assume the premise $\emptyset\,|\,\Gamma \vdash e_1 : Bot\ \langle\tau_2 \to \tau\rangle$ and the premise $\emptyset\,|\,\Gamma \vdash e_2 : Bot\ \langle\tau_2\rangle$. The first premise is equivalent to $\emptyset\,|\,\Gamma \vdash e_1 : Bot\ (\langle\tau_2\rangle \to Bot\ \langle\tau\rangle)$ by definition. Using the tautology rule of entailment, we can also conclude that $\emptyset \models Bot \rhd Bot$ and therefore we can apply rule (TI-App) to derive $\emptyset\,|\,\Gamma \vdash e_1\ e_2 : Bot\ \langle\tau\rangle$ which is the desired result.

3.4 Extensions

Unfortunately, the basic type rules are fragile with respect to η-expansion. For example, consider the following functions:

$$id\ = \lambda x.x$$

$$\begin{array}{l} iapp : (int \to Beh\ int) \to Beh\ int \quad \texttt{(* given *)} \\ iapp\ = \lambda f.f\ 1 \end{array}$$

The basic rules infer the type of id to be $\forall\alpha.\ \alpha \to Bot\ \alpha$, and we suppose the type of $iapp$ is given by the programmer. With these types both the applications $iapp\ id$ and $iapp\ (\lambda x.x)$ are rejected because id and $\lambda x.x$ have the type $\alpha \to Bot\ \alpha$ where the monadic type Bot doesn't match the expected monad Beh.

However, we can lift the monadic result type by using η-expansion and introducing an application node, e.g. the η-expanded expression $iapp\ (\lambda x.id\ x)$ is accepted since the application rule allows one to lift the result monad to the required Beh monad. Since the monadic types only occur on arrows, the programmer can always use a combination of applications and η-expansions to lift a monadic type anywhere in a type.

Fortunately, such manual η-expansion is rarely required: only when combining higher-order functions where automatic lifting is expected on the result type. The inferred types in the basic system are also often general enough to avoid need of it. For example, without annotation, the inferred principal type for $iapp$ is $\forall\alpha\mu_1\mu_2.\ (\mu_1 \rhd \mu_2) \Rightarrow (int \to \mu_1\ \alpha) \to \mu_2\ \alpha$ where all the given applications are accepted as is without need for η-expansion.

Lifting

Nevertheless, it is possible to make the type rules more robust under η-expansion, where we extend the basic system with a general lifting rule (TI-Lift) given in Figure 5 which allows arbitrary lifting of monadic expressions. For example, the id function in this system has the inferred type $\forall\alpha\mu.\ \alpha \to \mu\ \alpha$. Using this new type, all the applications $iapp\ id$, $iapp\ (\lambda x.x)$, and $iapp\ (\lambda x.id\ x)$ are accepted. The good news is that extending the system with (TI-Lift) is benign: we can still do full type inference and constraint solving as shown in later sections. The bad news is that some inferred types become slightly more complicated. For example, the type for `compose` (given in Section 2.3) would be

$$\begin{array}{l} \forall\alpha\beta\gamma\mu_1\mu_2\mu_3\mu_4\mu.\ (\mu_1 \rhd \mu, \mu_2 \rhd \mu) \Rightarrow \\ \quad (\beta \to \mu_2\ \gamma) \to \mu_3\ ((\alpha \to \mu_1\ \beta) \to \mu_4\ (\alpha \to \mu\gamma)) \end{array}$$

Structural subtyping

To ensure robustness under η-expansion while retaining simple types we could introduce a structural subtyping rule. In particular, besides (TI-Lift) we could also introduce the rule:

$$\frac{P\,|\,\Gamma \vdash e : \tau \qquad P \models \tau \rhd \tau'}{P\,|\,\Gamma \vdash e : \tau'}\ (\text{TI-Subsume})$$

Note that the subsumption constraint is between *types* instead of between monads. The rules for subsumption are:

$$\frac{P \models \tau_1' \rhd \tau_1 \qquad P \models \tau_2 \rhd \tau_2' \qquad P \models m \rhd m'}{P \models \tau_1 \to m\ \tau_2 \rhd \tau_1' \to m'\ \tau_2'}\ (\text{S-Fun})$$

$$P \models \tau \rhd \tau\ (\text{S-Taut})$$

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

$$\frac{\Gamma(x) = \sigma \qquad P \models \sigma \geqslant \tau}{P \,|\, \Gamma \vdash^\star x : \tau}\ \text{(TS-Var)} \qquad \frac{\Gamma(c) = \sigma \qquad P \models \sigma \geqslant \tau}{P \,|\, \Gamma \vdash^\star c : \tau}\ \text{(TS-Const)}$$

$$\frac{P \,|\, \Gamma \vdash^\star v : \tau}{P \,|\, \Gamma \vdash^\bullet v : \mathit{Bot}\ \tau}\ \text{(TS-Bot)} \qquad \frac{P \,|\, \Gamma \vdash^\bullet e_1 : m_1\ (\tau \to m_3\ \tau') \qquad P \,|\, \Gamma \vdash^\bullet e_2 : m_2\ \tau \qquad \forall i.\ P \models m_i \rhd m}{P \,|\, \Gamma \vdash^\bullet e_1\ e_2 : m\ \tau'}\ \text{(TS-App)}$$

$$\frac{P \,|\, \Gamma, x{:}\tau_1 \vdash^\bullet e : m\ \tau_2}{P \,|\, \Gamma \vdash^\star \lambda x.e : \tau_1 \to m\ \tau_2}\ \text{(TS-Lam)} \qquad \frac{P' \,|\, \Gamma \vdash^\star v : \tau' \qquad \sigma = \mathit{Gen}(\Gamma, P' \Rightarrow \tau') \qquad P \,|\, \Gamma, x{:}\sigma \vdash^\bullet e : m\ \tau}{P \,|\, \Gamma \vdash^\bullet \mathsf{let}\ x{=}v\ \mathsf{in}\ e : m\ \tau}\ \text{(TS-Let)}$$

$$\frac{e_1 \neq v \qquad P \,|\, \Gamma \vdash^\bullet e_1 : m_1\ \tau_1 \qquad P \,|\, \Gamma, x{:}\tau_1 \vdash^\bullet e_2 : m_2\ \tau_2 \qquad \forall i.\ P \models m_i \rhd m}{P \,|\, \Gamma \vdash^\bullet \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : m\ \tau_2}\ \text{(TS-Do)}$$

Figure 6. The syntax-directed type rules. The generalization function is defined as: $\mathit{Gen}(\Gamma, \sigma) = \forall(\mathsf{ftv}(\sigma) \setminus \mathsf{ftv}(\Gamma)).\sigma$.

$$\frac{P \,|\, \Gamma \vdash^\star v : \tau}{P \,|\, \Gamma \vdash^\bullet v : m\ \tau}\ \text{(TS-Lift)}$$

Figure 7. The syntax directed type rules extended with a lifting rule which replaces the rule (TS-Bot).

These rules are structural over arrows using the usual co/contra-variant typing.[3] With these rules we can give functions like id and $iapp$ fairly simple types:

$$\begin{aligned} &id : \forall\alpha.\ \alpha \to \mathit{Bot}\ \alpha \\ &iapp : \forall\mu.\ (int \to \mu\ int) \to \mu\ int \end{aligned}$$

And under the subsumption rules we can show that this type for id, $\forall\alpha.\ \alpha \to \mathit{Bot}\ \alpha$, is as general as the type $\forall\alpha\mu.\ \alpha \to \mu\ \alpha$.

Unfortunately, it turns out that it is exceedingly difficult to solve constraints between arbitrary types, as opposed to constraints between just monadic types. Since the subsumption rules give rise to constraints between types, we cannot give a coherent constraint solving strategy that is still complete—we either need to reject certain reasonable programs or we need to solve such constraints too aggressively leading to an incomplete inference algorithm.

Thus, our implementation uses the simple strategy since neither lifting nor subsumption provide a satisfactory improvement.

3.5 Syntax-directed type inference

Figure 6 presents a syntax directed version of the declarative type rules. The rules come in two flavors, one for value expressions $P \,|\, \Gamma \vdash^\star v : \tau$, and one for general expressions $P \,|\, \Gamma \vdash^\bullet e : m\ \tau$. Since each flavor has a unique rule for each syntactical expression, the shape of the derivation tree is uniquely determined by the expression syntax. Just like the Hindley-Milner syntax directed rules, all instantiations occur at variable and constant introduction, while generalization is only applied at let-bound value expressions.

We can show that the syntax directed rules are sound and complete with respect to the declarative rules.

Theorem 2 (The syntax directed rules are sound and complete)**.**
Soundness: *For any derivation $P \,|\, \Gamma \vdash^\star v : \tau$ there exists a derivation $P \,|\, \Gamma \vdash v : \tau$, and similarly, for any $P \,|\, \Gamma \vdash^\bullet e : m\ \tau$ we have $P \,|\, \Gamma \vdash e : m\ \tau$.*
Completeness: *For any derivation on a value expression $P \,|\, \Gamma \vdash v : \sigma$ there exists a derivation $P' \,|\, \Gamma \vdash^\star v : \tau$, such that $\Gamma \vdash (P' \mid \tau) \geqslant (P \mid \sigma)$. Similarly, for any derivation $P \,|\, \Gamma \vdash e : m\ \tau$, there exists a derivation $P' \,|\, \Gamma \vdash^\bullet e : m'\ \tau'$, such that $\Gamma \vdash (P' \mid m'\ \tau') \geqslant (P \mid m\ \tau)$.*

Both directions are proved by induction on the derivations. Following Jones [14], we use an extension of the instance relation in order to define an ordering of polymorphic type schemes and monadic types under some constraint set. We can define this formally as:

$$\frac{\sigma_1 = \forall\overline{\nu}.\, \bar{\pi} \Rightarrow \tau \qquad P_2 \vdash \mathit{Gen}(\Gamma, \forall\overline{\nu}.\, (P_1, \bar{\pi}) \Rightarrow \tau) \geqslant \sigma_2}{\Gamma \vdash (P_1 \mid \sigma_1) \geqslant (P_2 \mid \sigma_2)}$$

$$\frac{\begin{array}{c} \bar{\alpha}, \bar{\mu} = \mathsf{ftv}(m_1, \tau_1, P_1) \setminus \mathsf{ftv}(\Gamma) \qquad \theta = [\overline{m}/\bar{\mu}, \bar{\tau}/\bar{\alpha}] \\ P_2 \models \theta P_1 \qquad P_2 \models \theta m_1 \rhd m_2 \qquad \theta\tau_1 = \tau_2 \end{array}}{\Gamma \vdash (P_1 \mid m_1\ \tau_1) \geqslant (P_2 \mid m_2\ \tau_2)}$$

Besides extending the instance relation to monadic types, the definition of this qualified instance relation allows us specifically to relate derivations in the declarative system that can end in a type scheme σ, to derivations in the syntax directed system that always end in a monotype.

Finally, the syntax directed rules for the declarative type rules extended with the rule (TI-Lift) can be obtained by replacing the rule (TS-Bot) with the rule (TS-Lift) given in Figure 7. This extended system is also sound and complete with respect to the extended declarative rules.

4. Principal types

The standard next step in the development would be to define an algorithmic formulation of the system (including a rewriting to output terms) and then prove that the algorithm is sound and complete with respect to the syntactical rules, thereby establishing the principal types property. Interestingly, we can do this by translation. In particular, we can show that the syntactical rules in Figure 6 directly correspond to the syntactical rules of OML in the theory of qualified types [14]. In the next subsection we prove that for every derivation on an expression e in our syntactical system, there exists an equivalent derivation of an encoded term $[\![e]\!]$ in OML and the other way around. Since OML has a sound and complete type reconstruction algorithm, we could choose to reuse that as is, and thereby get sound and complete type inference (and as a consequence there exist principal derivations).

Unfortunately, the OML type reconstruction algorithm (essentially the Haskell type class inference algorithm) is not satisfactory, as it would reject many useful programs. Intuitively, this is because it conservatively rejects solutions to constraints that are reasonable in light of the morphism laws; since it is unaware of these laws it cannot take advantage of them. The next section develops an algorithm that takes advantage of the morphism laws to be both permissive and coherent.

[3] Note that when a system has higher-kinds, we need to ensure that the arrow is not a first-class type constructor.

4.1 Translation to OML

The translation between our system and OML is possible since we use the same instance and generalization relation as in the theory of qualified types. Moreover, it is easy to verify that our entailment relation over morphism constraints satisfies all the requirements of the theory, namely monotonicity, transitivity, and closure under substitution. The more difficult part is to find a direct encoding to OML terms. First, we are going to assume some primitive terms in OML that correspond to rules in our syntactical system:

$$
\begin{array}{ll}
\mathsf{bot} & : \forall\alpha.\, \alpha \rightarrow \mathit{Bot}\ \alpha \\
\mathsf{do} & : \forall\alpha\beta\mu_1\mu_2\mu.\, (\mu_1 \rhd \mu, \mu_2 \rhd \mu) \\
 & \qquad \Rightarrow \mu_1\ \alpha \rightarrow (\alpha \rightarrow \mu_2\ \beta) \rightarrow \mu\ \beta \\
\mathsf{app} & : \forall\alpha\beta\mu_1\mu_2\mu_3\mu.\, (\mu_1 \rhd \mu, \mu_2 \rhd \mu, \mu_3 \rhd \mu) \\
 & \qquad \Rightarrow \mu_1\ (\alpha \rightarrow \mu_3\ \beta) \rightarrow \mu_2\ \alpha \rightarrow \mu\ \beta
\end{array}
$$

Using these primitives, we can give a syntactic encoding from our expressions into OML terms:

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

We can now state soundness and completeness of our syntactic system with respect to encoded terms in OML, where we write $P \,|\, \Gamma \vdash_{\mathrm{OML}} e : \tau$ for a derivation in the syntax directed inference system of OML (cf. Jones [14], Fig. 4).

Theorem 3 (Elaboration to OML is sound and complete)**.**
***Soundness**: Whenever $P \,|\, \Gamma \vdash^\star v : \tau$ we can also derive $P \,|\, \Gamma \vdash_{\mathrm{OML}} [\![v]\!]^\star : \tau$ in OML. Similarly, when $P \,|\, \Gamma \vdash^\bullet e : m\ \tau$ we have $P \,|\, \Gamma \vdash_{\mathrm{OML}} [\![e]\!] : m\ \tau$.*
***Completeness**: If we can derive $P \,|\, \Gamma \vdash_{\mathrm{OML}} [\![v]\!]^\star : \tau$, there also exists a derivation $P \,|\, \Gamma \vdash^\star v : \tau$, and similarly, whenever $P \,|\, \Gamma \vdash_{\mathrm{OML}} [\![e]\!] : m\ \tau$, we also have $P \,|\, \Gamma \vdash^\bullet e : m\ \tau$.*

The proof of both properties can be done by straightforward induction on terms. As a corollary, we can use the general type reconstruction algorithm W from the theory of qualified types which is shown sound and complete to the OML type rules. Furthermore, it means that our system is sound, and we can derive principal types.

Corollary 4. *The declarative and syntactic type rules admit principal types.*

Again, the same results hold for the extended type rules with the (TI-Lift) and (TS-Lift) rules. The only change needed is that the lifting primitive now needs to be polymorphic to reflect the (TS-Lift) rule, i.e. $\mathsf{bot} : \forall\alpha\mu.\, \alpha \rightarrow \mu\ \alpha$.

4.2 Ambiguous types

Following Theorem 3, we could encode our type inference algorithm using the type class facility of a language like Haskell, employing a morphism type class that provides morphisms between monads. In particular:

```
class Morph m n  where
  lift :: m a -> n a

app :: Morph m1 m, Morph m2 m, Morph m3 m, Monad m
       => m1 (a -> m2 b) -> m3 a -> m b
app mf mx = lift mf >>= \f ->
            lift mx >>= \x ->
            lift (f x)
...
```

Type checking could now be implemented using the syntactical encoding into a Haskell program and running the Haskell type checker. Unfortunately, this approach would not be very satisfactory: it turns out that our particular morphism constraints quickly lead to ambiguous types that cannot be solved by a generic system. In particular, Haskell rejects any types that have variables in the constraints that do not occur in the type (which we call free constraint variables).

Recall our function $\mathit{iapp} : (\mathit{int} \rightarrow \mathit{Beh}\ \mathit{int}) \rightarrow \mathit{Beh}\ \mathit{int}$. The expression $[\![\mathit{iapp}\ (\lambda x.\ \mathit{id}\ (\mathit{id}\ x))]\!]$ has the Haskell type $\forall\mu.\, (\mathit{Morph}\ \mu\ \mathit{Beh}) \Rightarrow \mathit{Beh}\ \mathit{int}$ where the type variable μ only occurs in the constraint but not in the body of the type. Any such type must be rejected in a system like Haskell. In general, there could exist multiple solutions for such free constraint variables where each solution gives rise to a different semantics. A common example in Haskell is the program $\mathit{show}\ [\,]$ with the type $\mathsf{Show}\ \alpha \Rightarrow \mathit{string}$. In this example, choosing to resolve α as char results in the string “”, while any other choice results in $[\,]$.

We were initially discouraged by this situation until we realized that focusing only on morphism constraints confers an advantage: the monad morphism laws allow us to show that any solution for the free constraint variables leads to semantically equivalent programs; i.e., the evidence translations for each solution are coherent.

Moreover, there is an efficient and decidable algorithm for finding a particular “least” solution. At a high-level our algorithm works by requiring that the set of monad constants and morphisms between them form a semi-lattice, with Bot as the least element, where all morphisms satisfy the monad morphism laws. Another requirement that is fulfilled by careful design of the type system is that the only morphism constraints are between monadic type constants or monadic type variables, and never between arbitrary types. We can repeatedly simplify a given constraint graph by eagerly substituting free constraint variables μ with the least upper bound of their lower bounds when these lower bounds are constants. This simple strategy yields a linear-time decision procedure. The next section presents the algorithm in detail and proves coherence.

5. Constraint simplification and coherence

This section presents an algorithmic formulation (a variation on the Hindley-Milner algorithm W) of our syntax-directed type inference system. The previous section established that while the type reconstruction algorithm of Jones can infer principal types, these types are frequently ambiguous and hence programs with these types must be rejected. The contribution of this section is a simple (linear time) procedure that can eliminate some ambiguous variables in the constraints of a type in a *coherent* way. By performing constraint simplification, the types inferred by our algorithm are intentionally not the most general ones. However, simplification allows strictly more programs to be accepted. Moreover, we can show that simplification is *justified*, in that the typability of the program is not adversely effected by the simplified type.

Section 5.1 discusses the key algorithmic typing and rules and illustrates elaboration of source terms to System F target terms. Section 5.2 gives our constraint solving algorithm. Finally, Section 5.3 shows, by appealing to the morphism laws, that our solving algorithm is coherent and does not introduce ambiguity into the semantics of elaborated terms.

5.1 Algorithmic rewriting

The structure of our algorithm W closely follows Jones’ algorithm for qualified types [14], and includes an elaboration into a calculus with first-class polymorphism. We formulate our algorithm in a stylized way to facilitate the proof of coherence. We think of the constraints generated by our system as forming a directed graph, with nodes corresponding to monad type constants and variables,

any type	t	$::=$	$m\ \tau \mid \tau$
constraint	π	$::=$	$\mathsf{Do}(m_1, m_2, m)$
		$\mid$	$\mathsf{App}(m_1, m_2, m_3, m)$
substitution	θ	$::=$	$\cdot \mid \alpha \mapsto \tau \mid \mu \mapsto m \mid \theta\theta'$
target types	t	$::=$	$\nu \mid \mathsf{t}_1\ \mathsf{t}_2 \mid \forall\nu.\mathsf{t} \mid \mathsf{t}_1 \to \mathsf{t}_2$
target terms	e	$::=$	$x \mid c \mid \lambda x{:}\mathsf{t}.\mathsf{e} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \Lambda\alpha.\mathsf{e} \mid \mathsf{e}\ [\mathsf{t}]$

Figure 8. Syntax of constraint bundles and a target language e.

and edges represented by the morphism relation $m \rhd m'$. However, instead of simply producing constraints of the form $m \rhd m'$ as in the syntax-directed system, our algorithm groups related constraints together in "bundles". Constraint bundles come in two flavors, corresponding to the fragments of the typing derivation (and hence bits of program syntax) that induced the constraints. The bundles allow us to reason that edges in constraint graph come in specific kinds of pairs or triples, thus syntactically restricting the shape of the graph and facilitating our coherence proof.

Figure 8 gives the syntax of target terms e and types t, and alters the syntax of constraints π to constraint bundles. As we will see shortly, the bundles arise from corresponding inference rules from Figure 9: $\mathsf{Do}(m_1, m_2, m)$ is induced by the monadic let-binding rule (W-Do) and $\mathsf{App}(m_1, m_2, m_3, m)$ by the (W-App) rule. Substitutions θ map type variables to types, and $\theta_1\theta_2$ denotes substitution composition.

Figure 9 shows the key rules in our algorithm W, expressed as a judgment $P \mid \Gamma \vdash^{\kappa} e : t; \theta \rightsquigarrow \mathsf{e}$ where the constraints P, type t, substitution θ, and target term e are synthesized. As shown in Figure 8, t is either τ or $m\ \tau$; as in the syntax-directed rules, κ denotes one of two modes, $\star$ and $\bullet$; and the target term e is explicitly typed. The substitution θ applies to the free type-level variables (α and μ) in Γ. An invariant of the rules is that $\theta(t) = t$, $\theta(P) = P$, and $\theta(\mathsf{e}) = \mathsf{e}$. For simplicity, we omit types on formal parameters and instantiation of type parameters in elaborated terms e. We also assume that a morphism from a monad m to m' is named $f_{m,m'}$; and the bind and unit of a monad m are bind_m and unit_m. The omitted rules are unsurprising.

Rule (W-Bot) corresponds to the syntactic rule (TS-Bot). It switches modes from $\bullet$ to $\star$ in its premise, produces the monadic type $\mathit{Bot}\ \tau$, and elaborates the term by inserting the unit for Bot.

Rule (W-App) elaborates each sub-term in its first two premises, and in the fourth and fifth premises, computes the most-general unifier θ_3 of the formal parameter type of e_1 and the value type of e_2. We generate an App-constraint bundle which indicates that there is a morphism from each of $\theta_3\theta_2\mu_1$, $\theta_3\theta_1\mu_2$, and $\theta_3\mu'$ to the result monad μ. In the elaborated terms, $f_{\mu_i,\mu}$ stand for morphisms that will be abstracted (or solved) at the nearest enclosing let; similarly the bind_μ are the binds of the result monad. The rule for monadic let-bindings, (W-Do), is nearly identical to (W-App) except that there is one fewer monad variable.

Finally, rule (W-Let) implements generalization. We rewrite the let-bound value v in the first premise, and compute the variables $\bar{\nu}$ over which we can soundly generalize. In the third premise, we compute the variables $\bar{\mu}$ that appear in the constraints P_1 but are not free in the type τ—these variables are candidates for constraint simplification. The judgment $P_1 \overset{\text{solve}(\bar{\mu})}{\longrightarrow} P_1'; \theta'$ simplifies constraints, eliminating the ambiguous type variables $\bar{\mu}$ coherently—this judgment is discussed in the next subsection. The last premise rewrites the body in a context in which x's type is generalized. In the conclusion, we translate to an explicitly typed application form, where the let-bound value is elaborated to generalize over both its constraints and the type variables $\bar{\nu}$.

5.2 Soundness and efficiency of constraint simplification

Intuitively, our algorithm views a constraint set P as a directed graph, where the nodes in the graph are the monad types, and the edges are introduced by the constraint bundles. For example, we view a bundle $\mathsf{Do}(m_1, m_2, m)$ as a graph with vertices for m_1, m_2 and m, and edges from m_1 to m and m_2 to m. In the discussion below, we informally use intuitions from this graphical view of P. For each edge between m and m' in the constraint graph, a solution to P must compute a specific morphism between m and m'.

We start our description of the algorithm with the definition of morphism-induced least-upper bounds. This definition is relative to an initial set of constraints P_0 that define the monad constants and primitive morphisms used to type a source program.

Definition 5 (Least-upper bound)**.** *With respect to an initial context P_0, given a set of monad constants $A = \{M_1, \ldots, M_n\}$, we write* $\mathsf{lub}(A) = M$ *to mean that M is the least upper bound of the monad constants in A, i.e., $\forall i.P_0 \models M_i \rhd M$; and for any M' such that $\forall i.P_0 \models M_i \rhd M'$, we have $P_0 \vdash M \rhd M'$. Although defined with regard to a particular initial context P_0, we write* $\mathsf{lub}(A)$ *for conciseness, leaving P_0 implicit.*

Our constraint simplification algorithm is straightforward. We limit our attention to cycle-free constraint graphs. Whenever a cycle is detected in the constraint graph, we require every variable and constant in the cycle to be identical—a constraint graph with a cycle containing more than one constant cannot be solved and the program is rejected.

Given a cycle-free graph we perform a topological sort and then proceed to simplify the graph starting from the leaves. We consider a variable μ only after all its children have been considered. All variables have lower bounds (in-edges), since variables are introduced by (W-Do) and (W-App) and have lower bounds by construction. Besides, the node corresponding to Bot has an out-edge to every other node. For each variable μ considered, if all its in-edges are from monad constants $A = \{M_1, \ldots, M_n\}$, and if μ has some out-edge (needed for coherence, and discussed in the next sub-section), we assign to μ the constant $\mathit{lub}(A)$, thus eliminating the variable and proceeding to consider the next variable, if any.

Figure 10 presents a set of inference rules that codifies this solving algorithm (omitting the cycle elimination phase, for simplicity). The judgment has the form $P \overset{\text{solve}(\bar{\mu})}{\longrightarrow} P'; \theta$. It considers the free constraint variables $\bar{\mu}$ in P, replacing them with monad constants under certain conditions, returning the residual constraints P' that cannot be simplified further. This judgment ensures that $\mathit{dom}(\theta) \subseteq \bar{\mu}$ and $\theta P' = P'$. Thus, in the (W-Let) rule we apply θ' to the body of e_1 in the conclusion, in effect resolving any free morphism $f_{\mu,\mu'}$ to the specific morphism determined by θ'. Notice that since $\mathit{dom}(\theta) \subseteq \bar{\mu}$, the premises of (W-Let) ensure that we eliminate only those variables appearing in neither the final type nor the context.

The inference rules make use of a few auxiliary functions, defined to the right of Figure 10. First, for a constraint bundle π, function $\mathit{up\text{-}bnd}(\pi)$ is the type of the resulting monad. In contrast, $\mathit{lo\text{-}bnd}(\pi)$ is the set of types in a constraint bundle *from which* we require morphisms. Both of these are lifted to sets of constraints in the natural way. We also define $\mathsf{flowsTo}_\mu\ P$, the set of constraints in P that have μ as an upper bound; $\mathsf{flowsFrom}_\mu\ P$, the set of constraints that have μ as a lower bound.

We now explain the rules in detail. Rule (S-μ) is the workhorse of the algorithm. In the first two premises, it selects some constraint π whose upper bound μ is in the list of variables to be solved, $\bar{\mu}$. The third premise checks that μ has an upper bound; i.e., it is the lower bound of at least one constraint in P (for coherence). The fourth premise defines A, the set of all of μ's lower bounds, and the

$$\boxed{P \mid \Gamma \vdash^{\kappa} e : t ; \theta \rightsquigarrow \mathsf{e}} \qquad \frac{P \mid \Gamma \vdash^{\star} v : \tau ; \theta \rightsquigarrow \mathsf{e}}{P, \pi \mid \Gamma \vdash^{\bullet} v : \mathit{Bot}\ \tau ; \theta \rightsquigarrow (\mathit{unit}_{\mathit{Bot}}\ \mathsf{e})}\ \text{(W-Bot)}$$

$$\frac{\begin{array}{c} P_1 \mid \Gamma \vdash^{\bullet} e_1 : \mu_1\ \tau_1 ; \theta_1 \rightsquigarrow \mathsf{e}_1 \qquad P_2 \mid \Gamma \vdash^{\bullet} e_2 : \mu_2\ \tau_2 ; \theta_2 \rightsquigarrow \mathsf{e}_2 \qquad \mu, \mu', \alpha, \beta \text{ fresh} \\ \theta_2\tau_1 = \theta_3(\alpha \to \mu'\ \beta) \quad \theta_1\tau_2 = \theta_3\alpha \quad P = (\theta_3\theta_2 P_1), (\theta_3\theta_1 P_2), \mathsf{App}(\theta_3\theta_2\mu_1, \theta_3\theta_1\mu_2, \theta_3\mu', \mu) \quad \theta = \theta_1\theta_2\theta_3 \end{array}}{P \mid \Gamma \vdash^{\bullet} e_1\ e_2 : \theta_3(\mu\ \beta) ; \theta \rightsquigarrow \theta(\mathit{bind}_\mu\ (f_{\mu_1,\mu}\ \mathsf{e}_1)\ (\lambda x{:}\_.\mathit{bind}_\mu\ (f_{\mu_2,\mu}\mathsf{e}_2)\ \lambda y{:}\_.(f_{\mu',\mu}(x\ y))))}\ \text{(W-App)}$$

$$\frac{\begin{array}{c} e_1 \neq v \qquad P_1 \mid \Gamma \vdash^{\bullet} e_1 : \mu_1\ \tau_1 ; \theta_1 \rightsquigarrow \mathsf{e}_1 \qquad P_2 \mid \Gamma, x{:}\tau_1 \vdash^{\bullet} e_2 : \mu_2\ \tau_2 ; \theta_2 \rightsquigarrow \mathsf{e}_2 \qquad \mu, \alpha, \beta \text{ fresh} \\ \theta_2\tau_1 = \theta_3(\alpha) \quad \theta_1\tau_2 = \theta_3(\beta) \quad P = (\theta_3\theta_2 P_1), (\theta_3\theta_1 P_2), \mathsf{Do}(\theta_3\theta_2\mu_1, \theta_3\theta_1\mu_2, \mu) \quad \theta = \theta_1\theta_2\theta_3 \end{array}}{P \mid \Gamma \vdash^{\bullet} \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : \theta_3(\mu\ \beta) ; \theta \rightsquigarrow \theta(\mathit{bind}_\mu\ (f_{\mu_1,\mu}\mathsf{e}_1)\ \lambda x{:}\alpha.(f_{\mu_2,\mu}\mathsf{e}_2))}\ \text{(W-Do)}$$

$$\frac{\begin{array}{c} P_1 \mid \Gamma \vdash^{\star} v : \tau ; \theta_1 \rightsquigarrow \mathsf{e}_1 \qquad \bar\nu = \mathsf{ftv}(P_1 \Rightarrow \tau) \setminus \mathsf{ftv}(\Gamma) \qquad \bar\mu = (\mathsf{ftv}(P_1) \setminus \mathsf{ftv}(\tau)) \cap \bar\nu \\ P_1 \overset{\text{solve}(\bar\mu)}{\longrightarrow} P_1' ; \theta' \qquad \sigma = \forall\bar\nu.P_1' \Rightarrow \tau \qquad P_2 \mid \Gamma, x{:}\sigma \vdash^{\bullet} e : m\ \tau ; \theta_2 \rightsquigarrow \mathsf{e}_2 \qquad \theta = \theta_1\theta_2 \end{array}}{P_2 \mid \Gamma \vdash^{\bullet} \mathsf{let}\ x{=}v\ \mathsf{in}\ e : \theta_1(m\ \tau) ; \theta \rightsquigarrow \theta((\lambda x{:}\_.\mathsf{e}_2)\ \Lambda\bar\nu.\mathsf{abstractConstraints}(P_1', \theta'\mathsf{e}_1))}\ \text{(W-Let)}$$

$$\text{where}\quad \begin{array}{lcl} \mathsf{abstractConstraints}((\pi, P), \mathsf{e}) & = & \mathsf{abstractConstraints}(\pi, \mathsf{abstractConstraints}(P, \mathsf{e})) \\ \mathsf{abstractConstraints}(\mathsf{Do}(m_1, m_2, m), \mathsf{e}) & = & \lambda \mathit{bind}_m{:}\_.\lambda f_{m_1,m}{:}\_.\lambda f_{m_2,m}{:}\_.\mathsf{e} \\ \mathsf{abstractConstraints}(\mathsf{App}(m_1, m_2, m_3, m), \mathsf{e}) & = & \lambda \mathit{bind}_m{:}\_.\lambda f_{m_1,m}{:}\_.\lambda f_{m_2,m}{:}\_.\lambda f_{m_3,m}{:}\_.\mathsf{e} \\ \mathsf{abstractConstraints}(\mathsf{Lift}(m), \mathsf{e}) & = & \lambda \mathit{unit}_m{:}\_.\mathsf{e} \end{array}$$

Figure 9. Selected algorithmic rules for elaboration into System F (with types in elaborated terms omitted for readability).

$$\frac{}{P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P ; \cdot} \qquad \frac{\pi, P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta}{\pi, \pi, P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta} \qquad \frac{P_2, P_1 \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta}{P_1, P_2 \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta}$$

$$\text{S-M}\ \frac{P_0 \models \mathit{lub}(\mathit{lo\text{-}bnds}(\pi)) \rhd \mathit{up\text{-}bnd}(\pi) \qquad P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta}{\pi, P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta}$$

$$\text{S-}\mu\ \frac{\begin{array}{c} \mathit{up\text{-}bnd}(\pi) = \mu \qquad \bar\mu = \mu, \bar\mu' \qquad \mathsf{flowsFrom}_\mu\ P \neq \{\} \\ A = \mathit{lo\text{-}bnds}(\pi, \mathsf{flowsTo}_\mu\ P) \qquad \theta = (\mu \mapsto \mathit{lub}(A)) \qquad \theta P \overset{\text{solve}(\bar\mu')}{\longrightarrow} P' ; \theta' \end{array}}{\pi, P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta\theta'}$$

where

$$\begin{array}{lcl} \mathit{up\text{-}bnd}(\mathsf{Lift}(m)) & = & m \\ \mathit{up\text{-}bnd}(\mathsf{App}(\cdot, \cdot, \cdot, m)) & = & m \\ \mathit{up\text{-}bnd}(\mathsf{Do}(\cdot, \cdot, m)) & = & m \\ \mathit{up\text{-}bnds}(P) & = & \bigcup_{\pi \in P}\{\mathit{up\text{-}bnd}(\pi)\} \\ \\ \mathit{lo\text{-}bnd}(\mathsf{Lift}(m)) & = & \mathit{Bot} \\ \mathit{lo\text{-}bnd}(\mathsf{App}(m_1, m_2, m_3, \cdot)) & = & \{m_1, m_2, m_3\} \\ \mathit{lo\text{-}bnd}(\mathsf{Do}(m_1, m_2, \cdot)) & = & \{m_1, m_2\} \\ \mathit{lo\text{-}bnds}(P) & = & \bigcup_{\pi \in P} \mathit{lo\text{-}bnd}(\pi) \end{array}$$

$$\begin{array}{lcl} \mathsf{flowsTo}_\mu\ P & = & \{\pi \mid \pi \in P\ \wedge\ \mu = \mathit{up\text{-}bnd}(\pi)\} \\ \mathsf{flowsFrom}_\mu\ P & = & \{\pi \mid \pi \in P\ \wedge\ \mu \in \mathit{lo\text{-}bnd}(\pi)\} \end{array}$$

Figure 10. $P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta$: Simplifying constraints using the *lub*-strategy.

fifth premise defines the substitution θ as mapping μ to the least upper bound of A. Recall again that *lub*is only defined on constant monads, so A must contain no variables μ'; this requirement essentially forces solving to proceed bottom up, with the leaves of the tree induced by P (following cycle elimination) solved first. Finally, the sixth premise applies the substitution to the constraints P and proceeds to solve them; the final substitution consists of the substitution θ' produced by this recursive step, composed with θ.

Rule (S-M) checks that constraints involving only constants (e.g., those whose μ have been completely substituted for) are consistent with the initial constraint set P_0, in which case they can be dropped. Finally, the first three rules help with bookkeeping, indicating that (1) constraints may remain unsimplified; (2) duplicate constraints may be dropped; and (3) constraints may be permuted. This last rule is non-deterministic, but it can be implemented easily using simple topological sort of a cycle-free constraint constraint graph.

The next definition and the following lemma establish that $P \overset{\text{solve}(\bar\mu)}{\longrightarrow} P' ; \theta$ only produces sound solutions to a constraint set. The proof is straightforward.

Definition 6 (Sound solution)**.** *Given an initial context P_0 and a constraint set P, a solution θ to the constraints P is sound if and only if, for each μ in dom(θ), we have $\{M_1, \ldots, M_n\}$ = lo-bnds(θ($\mathsf{flowsTo}_\mu\ P$)), and $\forall i.P_0 \models M_i \rhd \theta\mu$.*

Lemma 7 (Constraint simplification is sound)**.** *For all $P_0, P, \bar\mu$, P', θ, if we have $P_0 \vdash P \overset{solve(\bar\mu)}{\longrightarrow} P' ; \theta$, then θ is sound for P.* □

Theorem 8 (Constraint solving is linear time)**.** *Given a constraint set P and an initial context P_0, there exists a $O(|P|)$ algorithm to decide whether or not P is* fully solvable*, where a constraint set P is fully solvable iff $\bar\mu = \mathsf{ftv}(P)$ implies $P_0 \vdash P \overset{solve(\bar\mu)}{\longrightarrow} \cdot ; \theta$.*

Proof. (sketch) The algorithm begins by detecting and eliminating cycles in the constraint graph. Doing so is linear in number of vertices and edges of the graph, i.e., $O(|V| + |E|)$, where $|V|$ and $|E|$ are each at most three times the number of $\mathsf{App}(m_1, m_2, m_3, m)$ constraints plus twice the number of $\mathsf{Do}(m_1, m_2, m)$ constraints.

Then the algorithm sorts the graph topologically (also linear).

Then it attempts to eliminate each constraint variable in sorted order. Eliminating a single constraint variable takes constant time. This is because elimination amounts to the cost of computing the *lub*for elements of a finite lattice in P_0 and all *lub*s can be precomputed on P_0 without dependence on P.

After eliminating all eligible constraint variables we are left with either a graph that has variables without upper bounds (in which case we answer "no") or we have a graph with only constants. Checking for upper bounds in a variable-free graph is again linear in the number of constraints, since each ordering can be answered in constant time (again, pre-computed on P_0). □

One would also like to show that our constraint solving algorithm does not solve constraints too aggressively. We define an *improvement* relation on type schemes (following the terminology of Jones [16]), as a lifting of the solving algorithm.

Definition 9 (Improvement of type schemes)**.** *Given a type scheme* $\sigma = \forall\bar{\nu}.P_1 \Rightarrow \tau$ *and a set of type variables* $\bar{\mu} = (\mathsf{ftv}(P_1) \setminus \mathsf{ftv}(\tau)) \cap \bar{\nu}$*. If* $P_1 \overset{solve(\bar{\mu})}{\longrightarrow} P_2; \theta$ *then we say* $\sigma' = \forall\bar{\nu}.P_2 \Rightarrow \tau$ *is an* improvement *of* σ*.*

We might conjecture at first that improvement of types is consistent with the type instantiation relation. That is, if σ' is an improvement of σ, then $P_0 \vdash \sigma' \geq \sigma$ and $P_0 \vdash \sigma \geq \sigma'$. However, by eliminating free constraint variables, our solving algorithm intentionally makes σ' less general than σ. For example, given $\sigma = \forall\mu.(M_1 \rhd \mu, \mu \rhd M_2) \Rightarrow \tau$, (where $\mu \notin \mathsf{ftv}(\tau)$) our algorithm could improve this to $\sigma' = M_1 \rhd M_2 \Rightarrow \tau$, and indeed further to τ, if $P_0 \models M_1 \rhd M_2$. However, for an arbitrary constant μ, it is not that case that $P_0 \models M_1 \rhd \mu, \mu \rhd M_2$, which is what is demanded by the instantiation relation.

Nevertheless, the type improvement scheme is still useful since improvement at generalization points does not impact the typability of the remainder of the program.

Theorem 10 (Improvement is justified)**.** *For all* $P, \Gamma, x, \sigma, \sigma'$, e, m, τ*, if we have* $P \mid \Gamma, x{:}\sigma \vdash e : m\ \tau$*, and* σ' *is an improvement of* σ*, then* $P \mid \Gamma, x{:}\sigma' \vdash e : m\ \tau$*.*

Intuitively, we can see this theorem holds because the improvement of a type $\sigma = \forall\bar{\nu}.\,P_1 \Rightarrow \tau$ to $\forall\bar{\nu}.\,P_2 \Rightarrow \tau$ only effects the free constraint variables: the actual type τ is unchanged and if $P \models P_1$ we always have $P \models P_2$ too. At any instantiation of σ, we can always substitute the improved type since the type τ is the same and the improved constraints P_2 are also entailed if the original constraints P_1 were.

5.3 Coherence

The effectiveness of our constraint-solving strategy stems from our ability to eagerly substitute constraint variable μ with the least upper bound M of all the types that flow to it. Such a technique is not admissible in a setting with general purpose qualified type constraints, particularly when the evidence for constraints (in this case our morphisms, binds and units) has operational meaning. One may worry that by instantiating μ with some $M' \neq M$ where $M \rhd M'$, we may get an acceptable solution to the constraint graph but the meaning of the elaborated programs differs in each case. This section shows that when the monad morphisms satisfy the morphism laws our constraint improvement strategy is coherent, i.e., all admissible solutions to the constraints yield elaborations with the same semantics. So, any specific solution (including the one produced by the *lub*-strategy) can safely be chosen.

Our approach to showing coherence proceeds as follows:

1. Given a constraint set P and a derivation $P_0 \vdash P \overset{\text{solve}(\bar{\mu})}{\longrightarrow} \cdot; \theta$, we call θ the *lub*-solution to P.
2. We can see all other solutions to P as being derived from the *lub*-solution by repeated *local modifications* to the *lub*-solution. A local modification involves picking a single variable μ such that $\theta = \theta'(\mu \mapsto M)$ and considering a solution to the constraint set $\theta' P$ that assigns some other solution $M' \neq M$ to μ; i.e., we have some solution $\theta_1 = \theta'(\mu \mapsto M')$. We can iterate this process, generating the solution θ_{i+1} from θ_i in this manner.
3. We enumerate the ways in which $\theta_i P$ can differ from $\theta_{i+1} P$, considering interactions between pairs of constraint bundles ($\mathsf{App}/\mathsf{App}, \mathsf{Do}/\mathsf{Do}$, Do/App, App/Do, etc.). In each case, since each kind of constraint bundle can be related to the abstract syntax of elaborated programs, we can reason about the differences in semantics that might arise from the θ_i and the θ_{i+1} solutions. We show that when all the morphisms satisfy the morphism laws, that the solutions are indeed equivalent.

Our result applies only to well-formed contexts, a notion defined below. In the definition, requirements (1) and (2) ensure that the monads and their morphisms are well-typed. Requirement (3) ensures that least-upper bounds are defined and that there is a Bot monad. Our notion of term equivalence, written $\mathsf{e}_1 \cong \mathsf{e}_2$, is extensional equality on well-typed, elaborated terms. Clauses (4) and (5) state that this equivalence is axiomatized by the transitivity property and the morphism laws.

Definition 11 (Well-formedness of a context)**.** *The following conditions are required of well-formed contexts,* P_0, Γ*:*

1. *For any pair of monad constants* M *we have* $bind_M$ *and* $unit_M$ *bound as constants in* Γ*, with appropriate types.*
2. *For all* M_1, M_2*, if* $P_0 \models M_1 \rhd M_2$ *then* Γ *contains a constant* f_{M_1,M_2} *bound at the type* $\forall\alpha.M_1\ \alpha \rightarrow M_2\ \alpha$*.*
3. *For any set of monad constants* A*, there exists* M *such that* $\mathsf{lub}(A) = M$ *and* Bot *is a monad constant in* Γ *with* $P_0 \models Bot \rhd M$*, for all* M*.*
4. *We assume that for all* M_1, M_2, M_3*, if* $P_0 \models M_1 \rhd M_2$ *and* $P_0 \models M_2 \rhd M_3$*, then* $f_{M_2,M_3} \circ f_{M_1,M_2} \cong f_{M_1,M_3}$*.*
5. *We assume that for all* $M_1, M_2, \mathsf{e}_1, \mathsf{e}_2, \mathsf{t}_1, \mathsf{t}_2$*, such that* $P_0 \models M_1 \rhd M_2$ *and* $\mathsf{e}_1 : M_1\ \mathsf{t}_1$ *and* $\mathsf{e}_2 : M_2\ \mathsf{t}_2$*, we have* $f_{M_1,M_2}(bind_{M_1}\ \mathsf{e}_1\ \lambda x{:}\mathsf{t}_1.\mathsf{e}_2) \cong bind_{M_2}\ (f_{M_1,M_2}\ \mathsf{e}_1)\ \lambda x{:}\mathsf{t}_1.(f_{M_1,M_2}\ \mathsf{e}_2)$

The following lemma establishes that in well-formed contexts, our algorithm produces well-typed System F terms. The proof is a straightforward induction on the structure of the derivation, where by $\{\![\Gamma]\!\}$ we mean the translation of a source typing context to a System F context.

Lemma 12 (Well-typed elaborations)**.** *Given* Γ *such that* P_0, Γ *is well-formed,* $e, t, \theta, \mathsf{e}, \kappa$*, such that* $P \mid \Gamma \vdash^{\kappa} e : t; \theta \rightsquigarrow \mathsf{e}$*. Then there exists* t *such that* $\{\![\theta\Gamma]\!\} \vdash_F \mathit{abstractConstraints}(P, \mathsf{e}) : \mathsf{t}$*.*

Next, we formalize the notion of a local modification θ' of a valid solution θ to constraint set. Condition (1) identifies the variable μ which is the locus of the modification. Conditions (2) and (3) establish the range of admissible solutions to μ, and condition (4) asserts that the modified solution θ' picks a solution for μ that is different than θ, but still admissible.

Definition 13 (Local modification of a solution)**.** *Given a solution* θ_1 *to a constraint set* (π, P)*, a local modification to* θ_1 *is a solution* θ_2 *for which the following conditions are true:*

1. *There exists a variable* μ *and a constant* M *such that* $\mu = \textit{up-bnd}(\pi)$ *and* $\theta_1 = \theta_1'(\mu \mapsto M)$*.*
2. *There exists* $M^{lo} = \mathsf{lub}(\theta_1'(\textit{lo-bnds}(\mathsf{flowsTo}_\mu\ P))))$*.* M^{lo} *is a lower-bound for* μ*.*
3. *There exists* $\{M_1^{hi}, \ldots, M_n^{hi}\} = \theta_1'(\textit{up-bnds}(\mathsf{flowsFrom}_\mu\ P))$*. Each* M_i^{hi} *is an upper bound for* μ*.*
4. *There exists a monad constant* $M' \neq M$ *such that* $P_0 \vdash M^{lo} \rhd M'$ *and* $\forall i.P_0 \vdash M' \rhd M_i^{hi}$*, such that* $\theta_2 = \theta_1'(\mu \mapsto M')$

Finally, we state and sketch a representative case of the main result of this section: namely, that the *lub*-strategy is coherent when the morphisms form a semi-lattice and satisfy the morphism laws.

Theorem 14 (Coherence of constraint solving)**.**
Given $P_0, \Gamma, P, e, t, \theta, \theta_1, \theta_2, \mathsf{e}, \kappa, \bar{\mu}$, such that

1. *P_0, Γ is well-formed and P is cycle-free.*
2. *For all $\mu \in \bar{\mu}$, the set $\mathsf{flowsFrom}_\mu\ P$ is non-empty, i.e., μ has an upper bound.*
3. *$P \mid \Gamma \vdash^\kappa e : t; \theta \rightsquigarrow \mathsf{e}$.*
4. *There exists θ_1 such that $dom(\theta_1) \subseteq \bar{\mu}$ and θ_1 is a sound solution for P.*
5. *There exists θ_2, a local modification of θ_1.*

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

Proof. (Sketch) Since θ_2 is a local modification, we have (from condition (1) of Definition 13) $\theta_1 = \theta_1'(\mu \mapsto M)$, for some μ, M, θ_1', and $P = \pi, P'$, where $\mu = \textit{up-bnd}(\pi)$ is the modified variable. We proceed by cases on the shape of π.
Case π is an App bundle: We have $\theta_1'\pi = \mathsf{App}(M_1^{lo}, M_2^{lo}, M_3^{lo}, \mu)$ (since from condition (2) of Definition 13, *lub* is only defined on monad constant). To identify the upper bounds of μ, we consider the constraints in $\theta_1'(\mathsf{flowsFrom}_\mu\ P')$, note that all the upper bounds must be constants (from condition (3)), and proceed by cases on the shape of each of the constraints π' in this set.

Sub-case π' is an App bundle: Without loss of generality on the specific position of μ, we have $\theta'\pi' = \mathsf{App}(m_1, \mu, m_3, M^{hi})$, where M^{hi} is an upper-bound of π'. From the shape of the constraints, we reason that we have a source term of the form $e\ (e_1\ e_2)$, that is elaborated to the term shown below, where $\mathsf{e}, \mathsf{e}_1, \mathsf{e}_2$ are the elaboration of the sub-terms.

1. $\textit{bind}_{M^{hi}}\ (f_{m_1,M^{hi}}\ \mathsf{e})\ (\lambda x{:}\_.\textit{bind}_{M^{hi}}$
2. $(f_{\mu,M^{hi}}(\textit{bind}_\mu\ (f_{M_1^{lo},\mu}\ \mathsf{e}_1)\ (\lambda x_1{:}\_.$
3. $\qquad \textit{bind}_\mu\ (f_{M_2^{lo},\mu}\ \mathsf{e}_2)\ (\lambda x_2{:}\_.(f_{M_3^{lo},\mu}(x_1\ x_2))))))$
4. $(\lambda y{:}\_.f_{m_3,M^{hi}}\ (x\ y)))$

Under the solutions θ_1 and θ_2, the inner subterm at lines 2 and 3 (call it $\hat{\mathsf{e}}$) may differ syntactically, i.e., $\theta_1\hat{\mathsf{e}} \neq \theta_2\hat{\mathsf{e}}$. Specifically, the solution θ_1 chooses $\mu \mapsto M$ while θ_2 may choose $\mu \mapsto M'$, for $M \neq M'$. However, using two applications of the morphism laws, (condition (5) of Definition 11), we can show that the $\hat{\mathsf{e}}$ is extensionally equivalent to the term shown below.

2. $\textit{bind}_{M^{hi}}\ (f_{\mu,M^{hi}} \circ f_{M_1^{lo},\mu}\ \mathsf{e}_1)\ (\lambda x_1{:}\_.$
3. $\textit{bind}_{M^{hi}}\ (f_{\mu,M^{hi}} \circ f_{M_2^{lo},\mu}\ \mathsf{e}_2)\ (\lambda x_2{:}\_.$
   $(f_{\mu,M^{hi}} \circ f_{M_3^{lo},\mu}(x_1\ x_2))))$

Appealing to condition (4) of Definition 11, the transitivity property, we get that the term above is extensionally equivalent to the term below (call it $\hat{\mathsf{e}}'$).

2. $\textit{bind}_{M^{hi}}\ (f_{M_1^{lo},M^{hi}}\ \mathsf{e}_1)\ (\lambda x_1{:}\_.$
3. $\textit{bind}_{M^{hi}}\ (f_{M_2^{lo},M^{hi}}\ \mathsf{e}_2)\ (\lambda x_2{:}\_.(f_{M_3^{lo},M^{hi}}(x_1\ x_2))))$

We have $\hat{\mathsf{e}} \cong \hat{\mathsf{e}}'$ and hence $\theta_1\hat{\mathsf{e}} \cong \theta_1\hat{\mathsf{e}}'$. Since, $\mu \notin FV(\hat{\mathsf{e}}')$, we have $\theta_1\hat{\mathsf{e}}' \cong \theta_2\hat{\mathsf{e}}'$, and $\theta_2\hat{\mathsf{e}}' \cong \theta_2\hat{\mathsf{e}}$, thus establishing $\theta_1\hat{\mathsf{e}} = \theta_2\hat{\mathsf{e}}$, as required. □

5.4 Ambiguity and limitations of constraint solving

Our constraint solving procedure is effective in resolving many common cases of free constraint variables in types that would otherwise be rejected as ambiguous by Haskell. However, a limitation of our algorithm is that, for coherence of solving, we require free constraint variables to have some upper bound in the constraint set. (See condition (2) of Theorem 14.) A variable with no upper bound may admit several possible solutions, so the morphisms leading to these solutions differ and result in different program rewritings—our algorithm rejects such a program as ambiguous.

We argue that for typical programs our constraint solving strategy is still effective. Our experience shows that terms with an unbounded constraint variable either consist of 'dead' computations that are never executed, or constitute top-level expressions. In the next paragraph we discuss a particular example program with an ambiguous type, illustrating the former case. To deal with the latter case, top-level expressions should have type annotations. All the examples in this paper are deemed unambiguous by our algorithm provided a top-level annotation. Out of the 5 example programs, 3 programs have types with variables that do not appear in the final type. All of these types would be rejected by OML (or Haskell) as ambiguous but are accepted by our system since they all have constraint variables with several distinct lower bounds and an upper bound, so we can instantiate them with the lub of the lower bounds.

Consider the following example, with a state monad ST and a primitive function `read`: $\textit{int} \to \mathsf{ST}\ \textit{char}$:

```
let g = fun () ->
  let f = fun x -> fun y ->
    let z = read x in read y in
  let w = f 0 in ()
```

Here, the type inferred for `f` is $\forall\mu.\,(\mathsf{ST} \rhd \mu) \Rightarrow \textit{int} \to \textit{Bot}\ \textit{int} \to \mu\ \textit{char}$. Because of the partial application `f 0`, we must give `g` the type $\forall\mu,\mu'.\,(\mathsf{ST} \rhd \mu, \textit{Bot} \rhd \mu') \Rightarrow \textit{unit} \to \mu'\ \textit{unit}$. Here, the constraint variable μ resulting from the partial application of `f` does not appear in the return type, while it appears in the constraints without an upper bound.

Picking an arbitrary solution for μ, say $\mu = \mathsf{ST}$ or $\mu = \textit{IO}$, where $P_0 \models \mathsf{ST} \rhd \textit{IO}$, causes the sub-term `w` to be given different types. This is a source of incoherence, since our extensional equality property is only defined on terms of the same type. However, pragmatically, the specific type chosen for `w` has no impact on the reduction of the program, and we conjecture that in all cases when this occurs, the unbounded constraint variable has no influence on the semantics of the program. As such, our implementation supports a "permissive" mode, so that despite it technically being ambiguous, we can accept the program `g`, and improve its type to $\forall\mu'.\textit{Bot} \rhd \mu', \Rightarrow \textit{unit} \to \mu'\ \textit{unit}$, by solving $\mu = \mathsf{ST}$.

6. Implementation and applications

We have implemented our inference algorithm for the core language of Figure 1 extended with standard features, including conditionals and recursive functions. Our implementation is written in Objective Caml (v3.12) and is about 2000 lines of code. It follows our basic morphism insertion strategy (i.e., Figure 4), but also provides an alternative typing mode that uses the (TI-Lift) rule (Figure 5). All rewritings shown in this paper, modulo minor readability improvements, were produced with our implementation and run against matching monadic libraries.

In this section we present programs using two additional monads, to give further examples of the usefulness of our system: parsing and information flow tracking. For the latter, our technical report [27] further considers a source language extended with mutable references, which for tracking information flow requires *parameterized monads*. We can type rewritten programs using the FlowCaml security type system [24] and thereby prove they are secure.

6.1 Parsing example

A parser can be seen as a function taking an input string and returning its unconsumed remainder along with a result of type α. We can apply this idea directly by implementing a parser as a monad whose type $Par\ \ \alpha$ conveniently hides the input and output strings. Its bind and unit combinators have names `bindp` and `unitp`, respectively. The `token`: $char \rightarrow Par\ unit$ parser parses a particular character, while `choice`: $(unit \rightarrow Par\ \alpha) \rightarrow (unit \rightarrow Par\ \alpha) \rightarrow Par\ \alpha$ returns the result of the first (thunkified) parser if it is successful, and otherwise the result of the second parser.

As an example we shall write a parser that computes the maximum level of nested brackets in an input string:

```
(rec nesting. fun ()->
 let nonempty = fun ()->
   token '[';
   let n  = nesting() in
   token ']';
   let m  = nesting() in
   max (n + 1) m in
 let empty = fun () -> 0 in
 choice (fun ()-> nonempty()) (fun ()-> empty()))()
```

Interpreted as standard ML code, the above program is not type correct: the functions `max` and `+` are typed as $int \rightarrow int \rightarrow int$, which does not match with the type of `n` and `m` of type $Par\ \ int$.

In our system the example is well-typed where the term gets type $Par\ \ int$. The type directed translation automatically inserts the binds for sequencing and units to lift the final result into the parser monad. The actual translation produced by our implementation is:

```
(rec nesting. fun () ->
 let nonempty = (fun ()->
   bindp (token '[') (fun _ ->
   bindp (nesting()) (fun n ->
   bindp (token ']') (fun _ ->
   bindp (nesting()) (fun m ->
   unitp (max (n + 1) m)))))) in
 let empty = fun () -> 0 in
 choice (fun () -> nonempty())
        (fun () -> unitp (empty()))) ()
```

6.2 Information flow

We are interested in enforcing a confidentiality property by tracking information flow. Data may be labeled with a security level, and the target independence property, called noninterference [6, 12], ensures that low-security outputs do not depend on high-security inputs. Ever since Abadi et al. showed how to encode information flow tracking in a dependency calculus [1], a number of monadic encodings have been proposed [4, 7, 19, 26]. We focus on a variant of the `Sec` monad [26] that wraps data protected at some security level for a pure functional subset of ML. In the absence of side effects, we only have to ensure that data with a certain confidentiality level is not disclosed to lower-level adversaries (explicit flows).

Let us consider a simple security lattice $\{\bot \leq L \leq H \leq \top\}$. The information flow monad $SecH$ (resp., $SecL$) tracks data with confidentiality level H (resp., L) with monadic operators `bindh,unith` (resp., `bindl,unitl`). The may-flow relation is expressed via a morphism that permits public data at a protected level:

$$\texttt{labup}: \quad SecL\ \triangleright\ SecH$$

Thus data labeled L may be used in a context expecting data labeled H, but not vice versa.

The following small example computes the interest due for a savings account, and the date of the last payment. Primitive `savings` returns a secret, having type $unit \rightarrow SecH\ float$, `rate` returns a public input having type $unit \rightarrow SecL\ \ float$. `add_interest` is a pure function computing the new amount of the account after adding interest, having type $float \rightarrow float \rightarrow float$. Finally, `current_date` returns the current date, having type $unit \rightarrow int$.

```
  add_interest (savings ()) (rate())
```

The rewriting lifts the low security `rate` to compute the high secrecy update for `savings`. The final type of the entire expression is $SecH\ float$.

```
bindh (savings ()) (fun y ->
bindh (labup (rate ())) (fun z ->
unith (add_interest y z)))}
```

Our extended technical report [27] gives a proof of soundness with respect to FlowCaml for an information flow state monad which subsumes the $Sec*$ monads; therefore they also soundly encode non-interference.

7. Related work

Our work builds on Jones' theory of qualified types [14–16], which ensures principal types and coherence of the type inference for OML and is used to infer Haskell type classes. We adapt this approach for a practical monadic setting. The key difference is that we make the solving procedure aware of morphism laws, in such a way that Jones' restriction on ambiguous types can be removed.

Filinski previously showed that any individual monadic effect can be synthesized from first-class (delimited) continuations and a storage cell [11], and thereby can be expressed in direct style without explicit use of bind and unit. Kiselyov and Shan [18] apply this representation to implement probabilistic programs as an extension to Objective Caml. While our system shares the same goals as these, it uses a different mechanism—type-directed rewriting—to insert monadic operators directly, rather than requiring them to be implemented in terms of continuations.

Filinski also showed how to implement monads in a composable way [9]: given implementations of individual monads, and an order in which they can be *layered* on top of each other, he gives a semantics to their compositions. However, Filinski's representation elides monadic types from terms, complicating program understanding. For example, the type of `seconds` in his system would be $unit \rightarrow int$, not $unit \rightarrow Beh\ int$. Our approach fully integrates monadic types with ML type inference and yields well-typed ML programs, therefore it is hopefully easy to understand by the programmers. Our approach can also be seen as orthogonal, since we leave the implementation of monads to the programmer, treating all monad operators as black boxes. The lattice induced by our morphism declarations corresponds to Filinski's layering structure between monads.

In his latest work on this topic [10], Filinski proposes an operational semantics for composing monads; he reflects monads in the types, as effects, and provides runtime guarantees for well-typed programs, by dynamically inserting the minimal number of binds and units, based on syntactic hints. Orthogonally, we perform type inference from unannotated source code. We make the monadic types, operators, and morphisms explicit in the rewritten program, which gives the programmer the option to review and assess the resulting program. Last but not least our approach supports polymorphism over monads, permitting us to abstract and generalize monads and morphisms. For example, the rewritten `compose` function, whose type is given in Section 2.3, would additionally take as arguments the monads and morphisms used by its body, akin to Haskell's dictionary-passing interpretation of type classes.

We can view our rewriting algorithm as a particular case of a more general strategy for *type-directed coercion insertion*, which supports automatic coercion of data from one type to another, without explicit intervention by the programmer. Most related to our approach is that of Luo [20, 21], which considers coercion insertion as part of polymorphic type inference. In Luo's system rewritings may be ambiguous: when more than one is possible, each may have different semantics. Also, the system does not include qualified types, so coercions may not be abstracted and generalized, hurting expressiveness. Our own prior work addressed the problem with ambiguity by carefully limiting the form and position of coercions [28]. However, we could not scale this approach to a setting with polymorphic type inference, as even the simplest combinations of coercions admitted (syntactic) ambiguity. Our restriction to monads in the present work addresses this issue: we can prove coherence by relying on the syntactic structure of the program to unambiguously identify where combinators should be inserted, and when the choice of combinators is unconstrained, the morphism laws allow us to prove that all choices are equivalent.

Benton and Kennedy developed MIL, the *monadic intermediate language*, as the basis of optimizations in their MLj compiler [2]. They observe, as we do, that ML terms can be viewed as having the structure of our types τ (Figure 1) where monads do not appear to the left of arrows. While our approach performs inference and translation together, their approach suggests an alternative: convert the source ML program into monadic form and then infer the binds, units, and morphisms. We know from our translation to Haskell that this approach can only work by informing the solver of monad morphisms.

8. Conclusions

Monads are a powerful idiom in that many useful programming disciplines can be encoded as monadic libraries. ML programs enjoy an inherent monadic structure, but the monad in question is hardwired to be the I/O monad. We set out to provide a way to exploit this structure so that ML programmers can program against monads of their choosing in a lightweight style.

The solution offered by this paper is a new way to infer monadic types for ML source programs and to elaborate these programs in a style that includes explicit calls into monadic libraries of the programmer's choice. A key consideration of our approach is to provide programmers with a way to reason about the semantics of elaborated programs. We achieve this in two ways. First, the types we infer are informative in that they explicitly indicate the monads involved. And, second, when our system accepts a program, we show that all possible elaborations of a program have the same meaning, i.e., our elaborations are coherent.

We implement our system in a prototype compiler, and evaluate it on a variety of domains. We find our system to be relatively simple, both to implement and to understand and use, and powerful in that it handles many applications of interest.

Acknowledgements The authors would like to thank Gavin Bierman and Matt McCutchen for their early contributions to this work, and to Gavin for comments on this draft. Hicks and Guts were both supported by NSF grant CNS-0905419.

References

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

[2] Nick Benton and Andrew Kennedy. Monads, effects and transformations. In *Electronic Notes in Theoretical Computer Science*, 1999.

[3] Greg Cooper and Shriram 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. *Journal of functional programming*, 15(02):249–291, 2005.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[6] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

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

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

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

[10] A. Filinski. Monads in action. In *POPL*, pages 483–494, 2010.

[11] Andrzej Filinski. Representing monads. In *POPL*, 1994.

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

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

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

[15] Mark P. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, September 1993.

[16] Mark P. Jones. Simplifying and Improving Qualified Types. Technical Report YALEU/DCS/RR-1040, Yale University, June 1994.

[17] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, 1993.

[18] Oleg Kiselyov and Chung chieh Shan. Embedded probabilistic programming. In *DSL*, 2009.

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

[20] Z. Luo. Coercions in a polymorphic type system. *MSCS*, 18(4), 2008.

[21] Z. Luo and R. Kießling. Coercions in Hindley-Milner systems. In *Proc. of Types*, 2004.

[22] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, 1989.

[23] Judea Pearl. Embracing causality in default reasoning (research note). *Artificial Intelligence*, 35(2):259–271, 1988.

[24] F. Pottier and V. Simonet. Information flow inference for ML. *TOPLAS*, 25(1):117–158, 2003.

[25] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

[26] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Haskell*, 2008.

[27] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. Technical Report MSR-TR-2011-039, Microsoft Research, May 2011.

[28] Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *ICFP*, 2009.

[29] Philip Wadler. The essence of functional programming. In *POPL*, 1992.