Lightweight Monadic Programming in ML

Nikhil Swamy — Microsoft Research
Nataliya Guts — University of Maryland
Daan Leijen — Microsoft Research
Michael Hicks — University of Maryland

Microsoft Research Technical Report

MSR-TR-2011-039

This is a extended version of a shorter paper of the same title. It differs from the short paper in the following regard.

The short version of the paper includes a section 6.2 in which we show how to encode information flow controls while accounting for leaks through side effects. We indicated that in order to support this encoding, mild extensions (in the form of parameterized monads) to the core type system described in earlier sections of the paper would be necessary. We have since realized that these extensions while feasible, are not as straightforward as we thought.

This version of the paper includes a reworked section 6.2. We now present an information flow encoding, suitable for use with pure programs, that can be handled by our current system with no extensions at all.

In Appendix A, we discuss the encoding of information flow with side effects in detail. We consider encodings in terms of parameterized monads, and sketch the shape of an extension to the core system to handle these parameterized monads.

Appendix B (also not present in the short paper, for space reasons) formalizes the soundness of the parameterized-monad information flow encoding via a translation to MLIF, the core language of FlowCaml. This result establishes that our information flow encoding correctly enforces a noninterference property for programs that include side effects.

Abstract

Many useful programming constructions can be expressed as monads. Examples include probabilistic computations, time-varying expressions, parsers, and information flow tracking, not to mention effectful features 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 typechecks. Our algorithm is based on Jones' qualified types and enjoys three useful properties: (1) principal types, i.e., the rewriting we perform is the most general; (2) coherence, i.e., thanks to the monad and morphism laws, all instances of the principal rewriting have the same semantics; (3) decidability; i.e., the solver for generated constraints will always terminate. Throughout the paper we present simple examples from the domains listed above. Our most complete example, which illustrates the expressive power of our system, proves that ML programs rewritten by our algorithm to use the information flow monad are equivalent to programs in FlowCaml, a domain-specific information flow tracking language.

1. Introduction

The research literature abounds with interesting programming constructions that can be expressed as monads. Examples include parsers [10], probabilistic computations [21], functional reactivity [7, 3], and information flow tracking [22]. 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 $\mathsf{IO}\ \tau$ in Haskell represents a computation that will produce a value of type τ but may perform effectful operations in the process. Haskell's `Monad` type class is blessed with special syntax, the *do notation*, for programming with instances of this class.

Moggi [19], Filinksi [8], 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 (`fun x->` e) can be viewed as having type $\tau \to m\ \tau'$: 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 an implicit Id monad 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 computations whose value varies with time, as in *functional reactive programs* [7, 3]. Behaviors can be implemented as a monad: expressions of type $\mathsf{Beh}\ \alpha$ represent values of type α that change over time, and `bindp` and `unitp` are its monadic operations:

$$\begin{aligned}&\mathsf{Monad}(\mathsf{Beh}, \mathtt{bindb}, \mathtt{unitb})\\ &\mathtt{bindb} : \forall\alpha,\beta.\mathsf{Beh}\ \alpha \to (\alpha \to \mathsf{Beh}\ \beta) \to \mathsf{Beh}\ \beta\\ &\mathtt{unitb} : \forall\alpha.\alpha \to \mathsf{Beh}\ \alpha\end{aligned}$$

As a primitive, function `seconds` has type $unit \to \mathsf{Beh}\ float$, its result representing the current time in seconds since the epoch. An ML program using Beh effectively has two monads: the Id 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 `bind`, `unit`, function composition, etc. Instead we would prefer to overload the existing syntax, e.g., to write the following (call it Q)

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

The type of this entire expression is $\mathsf{Beh}\ int$: it is time-varying, oscillating between values 1 and 2 every second. Using monad operations directly, we might write

```
bindb (bindb (seconds()) (fun s-> unitb (is_even s)))
  (fun y-> unitb (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.

Filinski [8] showed that individual monads could be encoded using continuations and with this encoding programmers could write Q directly. However, we find his treatment unsatisfactory for two reasons. First, some of the monadic type structure is hidden, making programs harder to reason about; e.g., the type of `seconds` in his system would be $unit \to int$, not $unit \to \mathsf{Beh}\ int$. Second, his encoding combines monads implicitly rather than explicitly. We would prefer that programmers specify how monads ought to be combined by writing *morphisms* to *lift* one monad into another. For example, suppose that in addition to time-varying expressions like `seconds()` we allowed time-varying probability distributions expressed as a monad $\mathsf{BehPrb}\ \alpha$ (we show an example of this in the next section). Writing a morphism from the first monad to the second, i.e., from a time-varying value to a time-varying probability distribution, ensures there is no mismatch with the representation.

This paper presents a type-based rewriting algorithm that takes an ML program and automatically translates the implicit use of monads into explicit uses, inserting binds, units, and morphisms where needed. Our algorithm does this in conjunction with polymorphic type inference, following an approach similar to Jones' qualified type inference [11]. We interpret ML types according to the intuition given above: we never have monadic types in negative positions, and moreover we never apply type constructors (including monad constructors) to monadic types, to ensure constructed types always classify values. 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 in solving for these variables our algorithm produces principal types; i.e., it finds the most general solution. Moreover, we show that our solution is *coherent*: when more than one rewriting is possible, each is behaviorally equivalent to the rewriting we actually produce.

Interestingly, while our algorithm bears close resemblance to Haskell's algorithm for type class inference (which follows that of Jones' OML [11]), our setting confers some useful advantages. For example, in Haskell one could define morphisms as instances of a `Morphism` type class and then rely on inference to insert them as necessary. However, the standard algorithm would reject many useful programs as potentially ambiguous. By contrast, our algorithm knows when constraints arise from morphisms, and can exploit morphism laws to find more solutions it knows will be coherent, while still ensuring decidability—in fact, we prove our constraint solving algorithm completes in linear time.

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 [6, 22, 16, 4]. We prove that our rewriting algorithm produces programs accepted by FlowCaml, a dialect of ML that performs information flow tracking [20], and thereby show that rewritten programs enjoy the security property of noninterference [5, 9]. As these examples show, our system can be viewed as an extension to ML in which programmers can easily define domain-specific languages using monadic encodings without suffering syntactic overhead. Moreover, they can reason easily about the semantics of the DSL since rewritings are coherent.

This paper is organized as follows. Section 2 presents an overview of our approach using several examples. Section 3 presents a declarative and syntactic formulation of our type system. Section 4 shows that our system can be translated into OML, and versa, and thereby is sound and has principal types. As OML's type inference algorithm is unsatisfactory, we develop our own algorithm in Section 5 and prove that it runs in linear time and produces coherent solutions. Section 6 develops further examples, including our information flow monad example. We end with a discussion of related work (Section 7) and conclude (Section 8).

2. Overview

This section presents an overview of approach through the development of a few examples.

Probability monad example Let us illustrate our idea on an example using the probability monad [21]. Expressions of type $\mathsf{Prb}\ \alpha$ describe distributions over values of type α, with `bindp` and `unitp` as its bind and unit combinators, respectively:

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

As mentioned earlier, the pure ML computations may be seen as a monad Id whose unit operator is the identity function, and bind is the reverse application (in the examples we may directly apply them to simplify the notation).

The probability monad can be used to define probabilistic models. The following program P 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_wet = (rain || sprinkler) && chance in
let f = fail () in
if grass_wet then rain else f
```

This program uses two functions it does not define:

$$\begin{array}{lcl} \mathtt{flip} & : & \mathit{float} \rightarrow \mathsf{Prb}\ \mathit{bool} \\ \mathtt{fail} & : & \forall\alpha.\mathit{unit} \rightarrow \mathsf{Prb}\ \alpha \end{array}$$

The first introduces distributions: $\mathtt{flip}(p)$ 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 two: 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 precipitation, 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 $\mathit{bool} \rightarrow \mathit{bool} \rightarrow \mathit{bool}$, to `rain` and `sprinkler`, which each have type $\mathsf{Beh}\ \mathit{bool}$. Fortunately, our system will automatically rewrite P to the following type-correct code:[1]

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

Notice that roughly each `let` is replaced by a `bindp` and each let-bound expression that is not already monadic is wrapped with `unitp`.

Time-varying probability monad example Now suppose we wish to program with both behaviors (see Section 1) and probabilities. 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 the `flip` is a function `rainprb` of type $\mathit{unit} \rightarrow \mathsf{Beh}\ \mathit{float}$:

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

Again, this program fragment is ill-typed, because `flip` expects a *float* but we have passed it a $\mathsf{Beh}\ \mathit{float}$. If our rewriting system is to be applied, what should be the type of `rain`? One might expect it to be $\mathsf{Beh}\ (\mathsf{Prb}\ \ \mathit{bool})$, since it is a time-varying distribution. However, this would not correspond to the monadic structure of the source program—only one monad can unambiguously characterize the effects of a computation (essentially for the same reason that types such as $\mathit{list}\ \ (m\ \ \alpha)$ would be problematic). Therefore, we require that monad types only appear in positive positions, and may not be the type argument of other monad types. As such the 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 [13]; here, we can combine them by tupling, with the obvious morphisms:

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

In general, a morphism $f_{1,2} : m_1 \rhd m_2$ has the type $\forall\alpha.m_1\ \alpha \rightarrow m_2\ \alpha$ and is expected to satisfy the morphism laws, namely:

$$\begin{array}{c} f_{1,2} \circ \mathit{unit}_1 = \mathit{unit}_2 \\ f_{1,2}\ (\mathit{bind}_1\ \mathsf{e}_1\ \mathsf{e}_2) = \mathit{bind}_2\ (f_{1,2}\ \mathsf{e}_1)\ (f_{1,2} \circ \mathsf{e}_2) \\ f_{2,3} \circ f_{1,2} = f_{1,3} \end{array}$$

With this additional information, our system rewrites P' as follows:

```
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 $\mathsf{BehPrb}\ \ \mathit{bool}$. Here, the result of `rainprb()` is lifted into $\mathsf{BehPrb}\ \mathit{float}$ and then bound to the current value `v1`, which is passed to `flip` to generate a distribution. This value is in turn lifted into $\mathsf{BehPrb}\ \mathit{bool}$ and bound to boolean `rain` for the rest of the computation, whose result, of type $\mathsf{Prb}\ \mathit{bool}$, is lifted into $\mathsf{BehPrb}\ \mathit{bool}$ by application of `p2bp`.

Properties of the rewriting Importantly, our algorithm infers *principal types*, i.e. most general types. Any of the types accepted by the declarative rules can be converted to the principal type.

As already mentioned, restricting the form and position of monadic types keeps constraint solving during inference *tractable*; our solving procedure chooses least upper bounds of relevant morphisms and runs in linear time. A solution of constraints allows us to instantiate the monadic operators and the morphisms in the elaborated term.

[1] Some trivial simplifications have been made for readability.

types	τ	$::=$	α	
		$\mid$	$T\,\tau_1 ... \tau_n$	($n \geqslant 0$ is the arity of T)
		$\mid$	$\tau_1 \to m\ \tau_2$	
monadic types	m	$::=$	$\mu \mid M$	
type variables	ν	$::=$	$\alpha \mid \mu$	
constraints	Π	$::=$	$\pi_1, ..., \pi_n$	
constraint	π	$::=$	$m_1 \rhd m_2$	
type schemes	σ	$::=$	$\forall \bar{\nu}. \Pi \Rightarrow \tau$	
environment	Γ	$::=$	$\cdot \mid \Gamma, c{:}\sigma \mid \Gamma, x{:}\sigma$	
values	v	$::=$	$x \mid c \mid \lambda x.e$	
expressions	e	$::=$	$v \mid e_1\ e_2 \mid \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2$	

Figure 1. The grammar of types, constraints, environments, and expressions

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

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

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

Figure 2. The constraint entailment relation.

Last but not least, our algorithm enjoys *coherence*: any two rewritings of the same program are semantically equivalent. Said otherwise, choosing a particular solution does not affect the meaning of the program. We eliminate one source of ambiguity by allowing at most one morphism for each pair of monads. Then, we take advantage of the monadic laws to convert different rewritings. For instance, the constraints of the type inferred for the last example admit other valid solutions. One of them directly lifts all the let bindings to the BehPrb monad:

```
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 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" monad as long as possible.

3. Qualified types for monadic programs

This section describes the formal type rules of our system. Figure 1 gives the grammar 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 $\tau_1 \to m\ \tau_2$. In particular, 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.

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

Figure 3. The generic instance relation over type schemes.

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 [11] 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 $\Pi \models \pi$ states that the constraints in Π entail the constraint π. The entailment relation is monotone (where $\Pi' \subseteq \Pi$ implies $\Pi \models \Pi'$), 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 $\Pi \vdash \sigma_1 \geqslant \sigma_2$ 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:

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

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 $\Pi \mid \Gamma \vdash v : \sigma$ states that value expression v is well typed with a type σ, while the rule $\Pi \mid \Gamma \vdash e : m\ \tau$ states that expression e is well-typed with a monadic type $m\ \tau$; in both cases assuming the constraints Π and type environment Γ. The rule (TI-Id) allows one to lift a regular type τ into a monadic type $\mathsf{Id}\ \tau$.

The rules for variables, constants, let-bound values, instantiation, and generalization are all standard. The rule for lambda expressions (TI-Lam) takes a monadic type in the premise to get well-formed function types. An expression like $\lambda x.x$ therefore gets type $\alpha \to \mathsf{Id}\ \alpha$ 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 $\forall i.\ \Pi \models m_i \rhd m$ 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 the previous section, 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 [11]. Since this is entirely standard, we elide the full rules, and only sketch how this 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 where we leave out type parameters for simplicity (since those can be inferred). For the declarative rules, we can define a judgment like $\Pi \mid \Gamma \vdash e : m\ \tau \rightsquigarrow$

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

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

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

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

Figure 4. The basic declarative type rules

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 $\Pi \models m_1 \rhd m_2 \rightsquigarrow f$ returns a morphism witness f with type $\forall \alpha.\, m_1\ \alpha \to m_2\ \alpha$.

As an example, consider the (TI-App) rule. We would get evidence terms for e_1 and e_2 as e_1 and e_2, respectively, and we would get a morphism witness for each constraint $m_i \rhd m$ as f_i. The application $e_1\ e_2$ is now translated into the target language as:

$$\mathit{bind}_m\ (f_1\ \mathsf{e}_1)\ (\lambda \mathsf{x}{:}(\tau_2 \to m_3\ \tau).\, \mathit{bind}_m\ (f_2\ \mathsf{e}_2)\ (\lambda \mathsf{y}{:}\tau_2.\, f_3\ (\mathsf{x}\,\mathsf{y})))$$

The bind_m evidence comes from an implicit *Monad* m constraint that is always satisfied (since it originates from morphism constraints) and therefore it is not present in the type rules. Also note that most of the time the morphisms will be the identity function and the binding operations will be in the Id monad which can all be optimized away. An optimizer can make further use of the morphism 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 extended with the value restriction – we write an ML derivation as $\Gamma \vdash_{\mathrm{ML}} e : \tau$. 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:

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

We can state compatibility with ML formally as:

Theorem 1 (Compatibility with ML)**.** *For any well-typed ML non-value expression e such that $\Gamma \vdash_{\mathrm{ML}} e : \tau$, we also have a valid monadic derivation in the* Id *monad of the form $\emptyset \,|\, \Gamma \vdash e : \mathsf{Id}\ \langle \tau \rangle$. For any well-typed value v where $\Gamma \vdash_{\mathrm{ML}} v : \tau$, 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 Id monad which means we can always reason under an empty constraint set $\emptyset$. Assuming empty constraints, the instance relation and generalization rule coïncide 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 $\emptyset \,|\, \Gamma \vdash e_1 : \mathsf{Id}\ \langle \tau_2 \to \tau \rangle$ and the premise $\emptyset \,|\, \Gamma \vdash e_2 : \mathsf{Id}\ \langle \tau_2 \rangle$. The first premise is equivalent to $\emptyset \,|\, \Gamma \vdash e_1 : \mathsf{Id}\ (\langle \tau_2 \rangle \to \mathsf{Id}\ \langle \tau \rangle)$ by definition. Using the tautology rule of entailment, we can also conclude that $\emptyset \models \mathsf{Id} \rhd \mathsf{Id}$ and therefore we can apply rule (TI-App) to derive $\emptyset \,|\, \Gamma \vdash e_1\ e_2 : \mathsf{Id}\ \langle \tau \rangle$ which is the desired result.

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

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

3.4 Extension with lifting

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

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

The basic rules infer the type of id to be $\forall \alpha.\ \alpha \to \mathsf{Id}\ \alpha$ (we assume the type of iapp is given). With these types both the applications $\mathit{iapp}\ \mathit{id}$ and $\mathit{iapp}\ (\lambda x.x)$ are rejected because id and $\lambda x.x$ have the type $\alpha \to \mathsf{Id}\ \alpha$ where the monadic type Id 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 $\mathit{iapp}\ (\lambda x.\mathit{id}\ x)$ 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 $\forall \alpha \mu_1 \mu_2.\ (\mu_1 \rhd \mu_2) \Rightarrow (\mathit{int} \to \mu_1\ \alpha) \to \mu_2\ \alpha$ where all the given applications are accepted as is without need for η-expansion.

It is possible though 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 $\mathit{iapp}\ \mathit{id}$, $\mathit{iapp}\ (\lambda x.x)$, and $\mathit{iapp}\ (\lambda x.\mathit{id}\ x)$ 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 get slightly more complicated. At the moment our implementation uses the simpler strategy.

$$\boxed{\Pi \,|\, \Gamma \vdash^{\star} v : \tau \quad \Pi \,|\, \Gamma \vdash^{\bullet} e : m\ \tau} \qquad \frac{\Gamma(x) = \sigma \quad \Pi \models \sigma \geqslant \tau}{\Pi \,|\, \Gamma \vdash^{\star} x : \tau}\ \text{(TS-Var)} \qquad \frac{\Gamma(c) = \sigma \quad \Pi \models \sigma \geqslant \tau}{\Pi \,|\, \Gamma \vdash^{\star} c : \tau}\ \text{(TS-Const)}$$

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

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

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

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

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

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

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

Figure 7. The type rules extended with a subsumption rule that allows structural morphisms between monadic types.

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

Figure 8. The syntax directed type rules extended with a lifting rule which replaces the rule (TS-Id).

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 $\Pi \,|\, \Gamma \vdash^{\star} v : \tau$, and one for regular expressions $\Pi \,|\, \Gamma \vdash^{\bullet} e : m\ \tau$. 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* $\Pi \,|\, \Gamma \vdash^{\star} v : \tau$ *there exists a derivation* $\Pi \,|\, \Gamma \vdash v : \tau$*, and similarly, for any* $\Pi \,|\, \Gamma \vdash^{\bullet} e : m\ \tau$ *we have* $\Pi \,|\, \Gamma \vdash e : m\ \tau$*.*
Completeness: *For any derivation on a value expression* $\Pi \,|\, \Gamma \vdash v : \sigma$ *there exists a derivation* $\Pi' \,|\, \Gamma \vdash^{\star} v : \tau$*, such that* $\Gamma \vdash (\Pi' \mid \tau) \geqslant (\Pi \mid \sigma)$*. Similarly, for any derivation* $\Pi \,|\, \Gamma \vdash e : m\ \tau$*, there exists a derivation* $\Pi' \,|\, \Gamma \vdash^{\bullet} e : m'\ \tau'$*, such that* $\Gamma \vdash (\Pi' \mid m'\ \tau') \geqslant (\Pi \mid m\ \tau)$*.*

Both directions are proved by induction on the derivations. Following Jones [11], we use an extension of the instance relation in order to define an ordering of polymorphic types schemes and monadic types under some constraint set. We can define this formally as:

$$\frac{\sigma_1 = \forall \overline{\nu}.\, \overline{\pi} \Rightarrow \tau \quad \Pi_2 \vdash \mathit{Gen}(\Gamma, \forall \overline{\nu}.\, (\Pi_1, \overline{\pi}) \Rightarrow \tau) \geqslant \sigma_2}{\Gamma \vdash (\Pi_1 \mid \sigma_1) \geqslant (\Pi_2 \mid \sigma_2)}$$

$$\frac{\begin{array}{c}\bar{\alpha}, \bar{\mu} = \mathsf{ftv}(m_1, \tau_1, \Pi_1) \setminus \mathsf{ftv}(\Gamma) \quad \theta = [\overline{m}/\bar{\mu}, \bar{\tau}/\bar{\alpha}] \\ \Pi_2 \models \theta\Pi_1 \quad \Pi_2 \models \theta m_1 \rhd m_2 \quad \theta\tau_1 = \tau_2\end{array}}{\Gamma \vdash (\Pi_1 \mid m_1\ \tau_1) \geqslant (\Pi_2 \mid m_2\ \tau_2)}$$

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-Id) with the rule (TS-Lift) given in Figure 8. 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 [11]. 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 [11] 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, while still enjoying principal types, takes advantage of the morphism laws to be both permissive and coherent.

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 derivations in our syntactical system:

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

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

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

We can now state soundness and completeness of our syntactic system with respect to encoded terms in OML, where we write $\Pi \,|\, \Gamma \vdash_{\mathrm{OML}} e : \tau$ for a derivation in the syntax directed inference system of Jones [11].

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

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. $\mathsf{lift} : \forall\alpha\mu.\, \alpha \to \mu\ \alpha$.

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 $iapp : (int \to \mathsf{Beh}\ int) \to \mathsf{Beh}\ int$. The expression $[\![iapp\ (\lambda x.\ id\ (id\ x))]\!]$ has the Haskell type $\forall\mu.\, (\mathsf{Morph}\ \mu\ \mathsf{Beh}) \Rightarrow \mathsf{Beh}\ int$ 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 $show$ [ ] with the type $\mathsf{Show}\ \alpha \Rightarrow string$. In that example, choosing to resolve α as $char$ results in the string "", while any other choice results in [ ]. This is also the essence of why subtyping combined with polymorphism is generally undecidable, where again different choices for free constraint variables can lead to different semantics.

We were initially discouraged by these undecidability results until we realized that our particular morphism constraints confer an advantage: the rich monad morphism laws allow us to show that any valid 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 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 note such constraints could occur with a deep subtyping relation, which we lack). Given a constraint graph we can repeatedly simplify by eagerly substituting free constraint variables μ that only have constant lower bounds, with the least upper bound of their lower bounds. This simple strategy yields a linear-time decision procedure. The next section presents the algorithm in detail and proves coherence of the constraint solutions.

5. Constraint solving and coherence

This section presents an algorithmic formulation (a variation on the Hindley-Milner algorithm W) of the syntax-directed rules in § 3.5. 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 the standard algorithm for qualified types [11], and includes an elaboration into a calculus with first-class polymorphism. Even though we showed in the previous section that we can use the type reconstruction algorithm of Jones' unchanged to do type inference, we are going to formulate a slight modification of that algorithm specifically for our system. The main reason is that we want to prove semantic coherence of the constraint simplification, and improvement for the free constraint variables. This proved very difficult to do on general constraint sets that result from the naïve algorithm.

Compared to [11] our algorithm has two key differences. First, we include (optional) constraint simplification and improvement [12] when generalizing types. Second, the constraints produced by our algorithm are grouped into constraint "bundles"—instead of producing constraints of the form $m \rhd m'$ as in the syntax-

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

Figure 9. The syntax of constraint bundles and a target language e

directed system, we group related constraints together. Constraint bundles come in three different flavors, corresponding to the fragments of the typing derivation (and hence bits of program syntax) that induced the constraints. These bundles allow us to formulate a structured constraint solving algorithm and to reason about its semantic coherence in a more structured way. Despite these two differences, our system follows Jones' type reconstruction algorithm very closely, so we believe that, like his, it produces principal types.

Figure 9 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 bundle $\mathsf{Do}(m_1, m_2, m)$ is induced by the monadic let-binding rule (W-Do); $\mathsf{App}(m_1, m_2, m_3, m)$ by the (W-App) rule; and $\mathsf{Lift}(m)$ by the (W-Lift) rule. Substitutions θ map type variables to types, and $\theta_1\theta_2$ denotes substitution composition.

Figure 10 shows the key rules in our algorithm W, expressed as judgment $\Pi \mid \Gamma \vdash^{\kappa} e : t; \theta \rightsquigarrow \mathsf{e}$ (where t is either τ or $m\ \tau$, as shown in Figure 9). As in the syntax-directed rules κ denotes one of two modes, $\star$ and $\bullet$, and the constraints Π and type t are synthesized. The judgment produces an explicitly typed target term e and a substitution θ that applies to the free type variables in Γ. An invariant of the rules is that $\theta(t) = t$, $\theta(\Pi) = \Pi$, 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 standard and are shown in the appendix.

Rule (W-Lift) switches modes from $\bullet$ to $\star$ in its premise, and generates a fresh monad variable μ, and elaborates the term by inserting the unit for μ.

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 a constraint bundle $\theta\mathsf{App}(m_1, m_2, \mu', \mu)$, which indicates that there be a morphism from each $\theta\ m_1$, $\theta\ m_2$, and $\theta\mu'$ 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 let-generalization. We rewrite the let-bound value v in the first premise, and compute the variable $\bar{\nu}$ over which we can soundly generalize. In the third premise, we compute the variables $\bar{\mu}$ that appear in the constraints Π_1 but are not free in the type τ: these variables are candidates for constraint simplification in the fourth premise. The judgment $\Pi_1 \overset{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi_1'; \theta'$ optionally simplifies constraints; this is a key contribution of our approach and is discussed in the next subsection. The last premise rewrites the body under a context with a generalized type for x. In the conclusion, we translate to System F's application form, where the let-bound value is elaborated to generalize over both its constraints and the type variables $\bar{\nu}$.

5.2 Efficient constraint solving

Intuitively, our algorithm views a constraint set Π 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, with edges from m_1 to and m_2 to m. In the discussion below, we informally use intuitions from this graphical view of Π. For each edge between m and m' in the constraint graph, a solution to Π must compute a specific morphism between m and m'.

We start our description of the algorithm with a standard notion of least-upper bounds, shown below. Our definition is relative to an initial set of constraints P_0 that define the monad constants and primitive morphisms to be used to type a source program.

Definition 5 (Least-upper bounds). *With respect to an initial context Π_0, given a set of monad constants $A = \{M_1, \ldots, M_n\}$, we write* $\mathrm{lub}(A) = M$ *to mean that M is the least upper bound of the monad constants in A, i.e., $\forall i.\Pi_0 \models M_i \rhd M$; and for any M' such that $\forall i.\Pi_0 \models M_i \rhd M'$, we have $\Pi_0 \vdash M \rhd M'$.*

Figure 11 presents a set of inference rules that codify our solving algorithm, implementing the judgment $\Pi \overset{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta$. The algorithm tries to eliminate the free constraint variables $\bar{\mu}$ in Π, replacing as many as possible with monad constants, and returning the residual constraints Π' that cannot be simplified further. This judgment ensures that $\mathit{dom}(\theta) \subseteq \bar{\mu}$ and that $\theta\Pi' = \Pi'$. 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'.

The inference rules make use of three auxiliary functions, defined to the right of Figure 11. First, for a constraint bundle $\mathit{up\text{-}bnd}(\pi)$ 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 Π_μ, the restriction of a set of constraints to a variable μ—intuitively, Π_μ computes the neighborhood of the vertex corresponding to μ in the graph corresponding to Π.

The algorithm starts with a cycle-elimination phase. To keep the rules readable, we omit the definition of cycle-elimination from Figure 11. For each cycle in the constraint graph, we require that every edge in the cycle be solved using the identity morphism. That is, on every cycle, there can be at most one monad constant M, and all variables in the cycle can be substituted for M.

Given a cycle-free constraint set P, the rules in Figure 11 are relatively straightforward. The first three rules state (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 rule (S-M) picks a constraint π where both the lower-bounds and the upper bounds of π are monad constants and solvable (i.e., $\mathit{lub}(\mathit{lo\text{-}bnds}(\pi))$ is defined and is coercible to the upper bound of π) and eliminates this constraint.

The rule (S-μ) is most interesting. It considers a constraint π whose upper bound is a free constraint variable μ which (according to the third premise) has some upper bound in Π. (Note, it is important for the coherence of solutions for μ to have an upper bound in Π in order to solve it—we discuss this point in the next subsection) For such a variable μ, we consider all its lower bounds A, and if we find them all to be monad constants (i.e., $\mathit{lub}(A) = M$ for some M), we assign the variable to M in θ and proceed to solve the rest of the constraints.

The next definition and the following lemma establishes that $P \overset{\text{solve}(\bar{\mu})}{\longrightarrow} P'; \theta$ only produces valid solutions to a constraint set,

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

$$\frac{\begin{array}{c}\Pi_1 \mid \Gamma \vdash^{\bullet} e_1 : m_1\ \tau_1; \theta_1 \rightsquigarrow \mathsf{e}_1 \qquad \Pi_2 \mid \Gamma \vdash^{\bullet} e_2 : m_2\ \tau_2; \theta_2 \rightsquigarrow \mathsf{e}_2 \qquad \mu, \mu', \alpha, \beta \text{ fresh} \\ \theta_2\tau_1 = \theta_3(\alpha \to \mu'\ \beta) \qquad \theta_1\tau_2 = \theta_3\alpha \qquad \Pi = (\theta_3\theta_2\Pi_1), (\theta_3\theta_1\Pi_2), \theta\mathsf{App}(m_1, m_2, \mu', \mu) \qquad \theta = \theta_1\theta_2\theta_3\end{array}}{\Gamma; \Pi \vdash^{\bullet} e_1\ e_2 : \theta_3(\mu\ \beta); \theta \rightsquigarrow \theta(\mathit{bind}_{\mu}(f_{m_1,\mu}\ \mathsf{e}_1)(\lambda x{:}\_.\mathit{bind}_{\mu}\ (f_{m_2,\mu}\mathsf{e}_2)\ \lambda y{:}\_.(f_{\mu',\mu}(x\ y))))}\ \text{(W-App)}$$

$$\frac{\begin{array}{c}\Pi_1 \mid \Gamma \vdash^{\star} v : \tau; \theta_1 \rightsquigarrow \mathsf{e}_1 \qquad \bar{\nu} = \mathsf{ftv}(\Pi_1 \Rightarrow \tau) \setminus \mathsf{ftv}(\Gamma) \qquad \bar{\mu} = (\mathsf{ftv}(\Pi_1) \setminus \mathsf{ftv}(\tau)) \cap \bar{\nu} \\ \Pi_1 \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi_1'; \theta' \qquad \sigma = \forall\bar{\nu}.\Pi_1' \Rightarrow \tau \qquad \Pi_2 \mid \Gamma, x{:}\sigma \vdash^{\bullet} e : m\ \tau; \theta_2 \rightsquigarrow \mathsf{e}_2 \qquad \theta = \theta_1\theta_2\end{array}}{\Gamma; \Pi_2 \vdash^{\bullet} \mathsf{let}\ x{=}v\ \mathsf{in}\ e : \theta_1(m\ \tau); \theta \rightsquigarrow \theta((\lambda x{:}\_.\mathsf{e}_2)\ \Lambda\bar{\nu}.\mathsf{abstractConstraints}(\Pi_1', \theta'\mathsf{e}_1))}\ \text{(W-Let)}$$

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

Figure 10. Selected algorithmic rules for elaboration into System F (with types in elaborated terms omitted for readability)

$$\frac{}{\Pi \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi; \cdot} \qquad \frac{\pi, \Pi \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta}{\pi, \pi, \Pi \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta} \qquad \frac{\Pi_2, \Pi_1 \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta}{\Pi_1, \Pi_2 \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta}$$

$$\text{S-M}\ \frac{\Pi_0 \models \mathit{lub}(\mathit{lo\text{-}bnds}(\pi)) \rhd \mathit{up\text{-}bnd}(\pi) \qquad \Pi \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta}{\pi, \Pi \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta}$$

$$\text{S-}\mu\ \frac{\begin{array}{c}\mathit{up\text{-}bnd}(\pi) = \mu \qquad \bar{\mu} = \mu, \bar{\mu}' \qquad \mu \in \mathit{lo\text{-}bnds}(\Pi) \\ A = \mathit{lo\text{-}bnds}(\pi, \Pi|_{\mu}) \qquad \theta = (\mu \mapsto \mathit{lub}(A)) \qquad \theta\Pi \stackrel{\text{solve}(\bar{\mu}')}{\longrightarrow} \Pi'; \theta'\end{array}}{\pi, \Pi \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta, \theta'}$$

where

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

$$\Pi|_{\mu} = \{\pi \mid \pi \in \Pi\ \wedge\ \mu \in \mathsf{ftv}(\pi)\}$$

Figure 11. $\Pi \stackrel{\text{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta$: Simplifying constraints using the *lub*-strategy

and that these solutions can be found efficiently. The proof is straightforward.

Definition 6 (Valid solutions)**.** *Given an initial context* Π_0*, a constraint set* Π*, a set of variables* $\bar{\mu}$*, and a substitution* θ *with* $dom(\theta) \subseteq \bar{\mu}$*, we say* θ *is valid partial solution for* Π *(denoted* $\Pi_0 \models \theta\ \mathsf{validFor}\ \Pi$*) when for all* $\pi \in \Pi$*, if* $\mu \in \mathit{up\text{-}bnds}(\Pi)$ *and* $\mu \in dom(\theta)$*, then there exists* M *such that* $\mathit{lo\text{-}bnd}(\theta\pi) = M$ *and* $\Pi_0 \models M \rhd \theta\mu$*.*

Lemma 7 (Validity of constraint solving)**.** *For all* $\Pi_0, \Pi, \bar{\mu}, \Pi', \theta$*, if we have* $\Pi_0 \vdash \Pi \stackrel{\mathit{solve}(\bar{\mu})}{\longrightarrow} \Pi'; \theta$*, then* $\Pi_0 \models \theta\ \mathsf{validFor}\ \Pi$*.* □

Theorem 8 (Constraint solving is linear time)**.** *Given a constraint set* Π*, and an initial context* Π_0*, there exists a* $O(|\Pi|)$ *algorithm to decide whether or not* Π *is fully solvable, where a constraint set* Π *is fully solvable if for* $\bar{\mu} = \mathit{ftv}(\Pi)$*,* $\Gamma \vdash \Pi \stackrel{\mathit{solve}(\bar{\mu})}{\longrightarrow} \cdot; \theta$*.*

Proof. (sketch) The algorithm views Π as a graph, with the monad variables and type constructors as nodes, and edges induced by the ordering constraints. A bound on the number of edges (and the number of vertices) is three times the number of W-App constraints, plus twice the number of do constraints, plus the number of lift constraints. Detecting and eliminating cycles in the graph is linear in number of vertices and edges. We then perform a topological sort of the graph ($O(|V| + |E|)$), where in this case the number of vertices and edges is bounded by the number of constraints.

Then start from the leaves and consider 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), where they, by construction have lower bounds; and a lower-bound of (W-Lift) is Id. Computing the lub for elements of a finite lattice in Π_0 is constant time (it can be pre-computed). At each step a variable is eliminated, until we are left either, with a graph that has variables that do not have upper bounds (in which case we answer "no"); or we have a graph with only constants.

Deciding the ordering 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 Π_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 [12]), as a lifting of the solving algorithm.

Definition 9 (Improvement of type schemes)**.** *Given a type scheme* $\sigma' = \forall\bar{\nu}.\Pi_1 \Rightarrow \tau$ *and a set of type variables* $\bar{\mu} = (\mathit{ftv}(\Pi_1) \setminus \mathit{ftv}(\tau)) \cap \bar{\nu}$*. If* $\Pi_1 \stackrel{\mathit{solve}(\bar{\mu})}{\longrightarrow} \Pi_2; \theta$ *then we say* $\sigma' = \forall\bar{\nu}.\Pi_2 \Rightarrow \tau$ *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 $\Pi_0 \models M_1 \rhd M_2$. However, for an arbitrary constant μ, it is not that case that $\Pi_0 \models M_1 \rhd \mu, \mu \rhd M_2$, 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 typeability of the remainder of the program.

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

Intuïtively, we can see this theorem holds because the improvement of a type $\sigma = \forall\bar{\nu}.\Pi_1 \Rightarrow \tau$ to $\forall\bar{\nu}.\Pi_2 \Rightarrow \tau$ only effects the free constraint variables: the actual type τ is unchanged and if $\Pi \models \Pi_1$ we always have $\Pi \models \Pi_2$ too. At any instantiation of σ, we can always substitute the improved type since the type τ is the same and the improved constraints Π_2 are also entailed if the original constraints Π_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 sound in a setting with general purpose type-class 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 sound, 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 Π and a derivation $\Pi_0 \vdash \Pi \xrightarrow{\text{solve}(\bar{\mu})} \cdot; \theta_0$, we call θ_0 the *lub*-solution to Π.
2. We argue that all other solutions to Π can be 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 $\theta'\Pi$ that assigns some other solution M' to μ; i.e., we have some solution $\theta_1 = \theta'(\mu \mapsto M')$. We iterate this process, generating the solution θ_{i+1} from θ_i in this manner.
3. We enumerate the ways in which $\theta_i\Pi$ can differ from $\theta_{i+1}\Pi$, considering interactions between pairs of constraint bundles ($\mathsf{App}/\mathsf{App}$, Do/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 notion of term equality, written $\mathsf{e}_1 \cong \mathsf{e}_2$, is extensional equality on well-typed elaborated terms axiomatized by the morphism laws. The following definition formalizes our notion of well-formed contexts, including the key requirements (3), (4), and (5), that the morphisms form a semi-lattice, and that they satisfy the morphism laws of transitivity and commutation with the bind operations of the relevant monads.

Definition 11 (Well-formedness of a context)**.** *A context* Π_0, Γ *is well-formed if and only if the following are true:*

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* $\Pi_0 \models M_1 \rhd M_2$ *then* Γ *contains a constant* f_{M_1,M_2} *bound at the type* $\forall\alpha.M_1\ \alpha \to M_2\ \alpha$.
3. *For any set of monad constants* A*, there exists* M *such that* $\Pi_0 \vdash \mathsf{lub}(A) = M$.
4. *For all* M_1, M_2, M_3*, if* $\Pi_0 \models M_1 \rhd M_2$ *and* $\Pi_0 \models M_2 \rhd M_3$*, then* $f_{M_2,M_3} \circ f_{M_1,M_2} \cong f_{M_1,M_3}$.
5. *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* $\mathsf{e}_2 : M_1\ \mathsf{t}_2$*, we have* $f_{M_1,M_2}(bind_{M_1}\ \mathsf{e}_1\ \lambda x{:}\mathsf{t}.\mathsf{e}_2) \cong bind_{M_2}\ (f_{M_1,M_2}\ \mathsf{e}_1)\ \lambda x{:}\mathsf{t}.(f_{M_1,M_2}\ \mathsf{e}_2)$

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* Π_0, Γ *is well-formed,* $e, t, \theta, \mathsf{e}, \kappa$*, such that* $\Pi \mid \Gamma \vdash^{\kappa} e : t; \theta \rightsquigarrow \mathsf{e}$*. Then there exists* t *such that* $[\![\theta\Gamma]\!] \vdash_F \mathsf{abstractConstraints}(\Pi, \mathsf{e}) : \mathsf{t}$.

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, 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, Π)*, a local modification to* θ_1 *is a solution* θ_2*, where 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 a constant* M_1*, a lower-bound for* μ*, where* $M_1 = \mathsf{lub}(\textit{lo-bnds}((\theta_1'\Pi)|_\mu) \setminus \{\mu\})$.
3. *There exists a set of constants (upper bounds for* μ*)* $\{M_1', \ldots, M_n'\} = \textit{up-bnds}((\theta_1'\Pi)|_\mu) \setminus \{\mu\}$.
4. *There exists a monad constant* $M' \neq M$ *such that* $\Gamma \vdash M_1 \rhd M$ *and* $\forall i.\Gamma \vdash M \rhd M_i'$*, 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 $\Pi_0, \Gamma, \Pi, e, t, \theta, \theta_1, \theta_2, \mathsf{e}, \kappa, \bar{\mu}$*, such that*

1. Π_0, Γ *is well-formed.*
2. *For all* $\mu \in \bar{\mu}$*, the set* $(\textit{lo-bnds}(\Pi|_\mu) \setminus \{\mu\})$ *is non-empty, i.e.,* μ *has an upper bound.*
3. $\Pi \mid \Gamma \vdash^{\kappa} e : t; \theta \rightsquigarrow \mathsf{e}$.
4. *There exists* θ_1 *such that* $dom(\theta_1) \subseteq \bar{\mu}$ *and* $P_0 \vdash \theta_1\ \mathsf{validFor}\ \Pi$.
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) $\theta_1 = \theta_1'(\mu \mapsto M_1')$, for some μ, M_1', θ_1', and $\Pi = \pi, \Pi'$, 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 $\theta_1'\pi = \mathsf{App}(M_1, M_2, M_3, \mu)$ (since from condition (2) of Definition 13, lower-bounds are only defined on monad constant). To identify the upper bounds of μ, we consider the constraints in $(\theta_1'\Pi')|_\mu$, 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 $\theta'\pi' = \mathsf{App}(m_1, \mu, m_3, M)$, where M 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 $\mathtt{e}, \mathtt{e}_1, \mathtt{e}_2$ are the elaboration of the sub-terms.

1. $\mathit{bind}_M\ (f_{m_1,M}\ \mathtt{e})\ (\lambda x{:}\_.\mathit{bind}_M$
2. $(f_{\mu,M}(\mathit{bind}_\mu\ (f_{M_1,\mu}\ \mathtt{e}_1)\ (\lambda x_1{:}\_.$
3. $\quad \mathit{bind}_\mu\ (f_{M_2,\mu}\ \mathtt{e}_2)\ (\lambda x_2{:}\_.(f_{M_3,\mu}(x_1\ x_2))))))$
4. $(\lambda y{:}\_.f_{m_3,M}\ (x\ y)))$

Under the solutions θ_1 and θ_2, the inner subterm at lines 2 and 3 may differ syntactically. Specifically, the solution θ_1 chooses $\mu \mapsto M_1'$ while θ_2 may choose $\mu \mapsto M_2'$, for some $M_1' \rhd M_2' \rhd M$. However, using two applications of the morphism laws, (condition (5) of Definition 11), we can show that the sub-term in question is extensionally equivalent to the term shown below.

2. $((\mathit{bind}_M\ (f_{\mu,M} \circ f_{M_1,\mu}\ \mathtt{e}_1)\ (\lambda x_1{:}\_.$
3. $\mathit{bind}_M\ (f_{\mu,M} \circ f_{M_2,\mu}\ \mathtt{e}_2)\ (\lambda x_2{:}\_.(f_{\mu,M} \circ f_{M_3,\mu}(x_1\ x_2))))))$

Finally, appealing to condition (4) of Definition 11, we get that the term above is extensionally equivalent to the term below.

2. $((\mathit{bind}_M\ (f_{M_1,M}\ \mathtt{e}_1)\ (\lambda x_1{:}\_.$
3. $\mathit{bind}_M\ (f_{M_2,M}\ \mathtt{e}_2)\ (\lambda x_2{:}\_.(f_{M_3,M}(x_1\ x_2))))))$

The semantics of this term is independent of the choice of μ, and we have our result for this sub-case. The other cases are similar. $\square$

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 effective since it is difficult to construct a term with an unbounded constraint variable, and we conjecture that all such examples consist of 'dead' computations that are never executed. Next we discuss a particular example program with such an ambiguous type. All the other examples in this paper are deemed unambiguous by our algorithm (though many would be rejected by Haskell).

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

```
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 $\forall\mu.\,(\mathsf{ST} \rhd \mu) \Rightarrow \mathit{int} \to \mathsf{Id}\ \mathit{int} \to \mu\ \mathit{char}$. Because of the partial application `f 0`, we must give `g` the type $\forall\mu,\mu'.\,(\mathsf{ST} \rhd \mu, \mathsf{Id} \rhd \mu') \Rightarrow \mathit{unit} \to \mu'\ \mathit{unit}$. 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 solutions for μ, say $\mu = \mathsf{ST}$ or $\mu = \mathsf{IO}$, where $\Pi_0 \models \mathsf{ST} \rhd \mathsf{IO}$, causes the sub-term `w` to be given different types. This is a source of decoherence, 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 $\forall\mu'.\mathsf{Id} \rhd \mu', \Rightarrow \mathit{unit} \to \mu'\ \mathit{unit}$, 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. We implement our basic morphism insertion strategy (i.e., Figure 4 without (TI-Lift) from Figure 5). The rewritings shown in Section 2 were produced with our implementation.

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, Appendix A further considers a source language extended with mutable references, which for tracking information flow requires *parameterized monads*. We can type our rewritten programs using the FlowCaml security type system [20] 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 $\mathsf{Par}\ \alpha$ conveniently hides the input and output strings. Its bind and unit combinators have names `bindp` and `unitp`, respectively. The `token`: $\mathit{char} \to \mathsf{Par}\ \mathit{unit}$ parser parses a particular character, while `choice`: $\mathsf{Par}\ \alpha \to \mathsf{Par}\ \alpha \to \mathsf{Par}\ \alpha$ returns the result of the first parser if it is successful, and otherwise the result of the second argument.

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

The above program would be rejected in ML: the functions `max` and `+` are typed as $\mathit{int} \to \mathit{int} \to \mathit{int}$, which does not match with the type of `n` and `m` of type $\mathsf{Par}\ \ \mathit{int}$.

In our system the example is well-typed where the term gets type $\mathsf{Par}\ \mathit{int}$, and where `n` and `m` can be used with type int (instead of $\mathsf{Par}\ \mathit{int}$). 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
 bindp (unitp (choice (fun () -> nonempty())))
  (fun v3 -> v3 (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 [5, 9], ensures that low-security outputs do not depend on high-security inputs. There is a large body of research in this area (for language-based techniques, see Sabelfeld and Myers' survey [23]). 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 [6, 22, 16, 4]. We focus on an expressive information flow monad for standard ML.

We implement a variant of the `Sec` monad [22] 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 $\{\bot \le L \le H \le \top\}$. 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:

$$\texttt{labup}: \; SecL \, \triangleright \, SecH$$

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 $unit \to SecH\,float$, `rate` returns a public input having type $unit \to SecL\; float$. `add_interest` is a pure function computing the new amount of the account after adding interest, having type $float \to float \to float$. Finally, `current_date` returns the current date, having type $unit \to int$.

```
  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
  (bindh (savings ()) (fun y ->
    unith (add_interest y)))
  (fun f ->
    bindh (labup (rate ())) (fun z ->
      (fun x -> labup (unitl x)) (f z)))
```

Appendix B 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

Filinski previously showed that any monadic effect can be synthesized from first-class (delimited) continuations and a storage cell [8], and thereby can be expressed in direct style without explicit use of bind and unit. Kiselyov and Shan [14] 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. Though we have not explored the ramifications of this difference in depth, our experience implementing the probability and behavior monads suggests that direct implementations may enjoy a performance advantage over the continuation-based approach. Another difference is our support for inferring morphisms between monads; monads in Filinski's system appear to be combined implicitly. As such the programmer has less control over monad interactions (though perhaps morphisms could be inserted manually). Finally, Filinski's representation elides monadic types from terms, complicating program understanding.

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 [18, 17], 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 [24]. 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 appear in positive positions. 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 wide interest.

Acknowledgements The authors would like to thank Gavin Bierman and Matt McCutcheon 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] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
[6] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *TLDI*, pages 59–72, 2011.
[7] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.
[8] Andrzej Filinski. Representing monads. In *POPL*, 1994.
[9] J.A. Goguen and J. Meseguer. Security policy and security models. In *Symposium on Security and Privacy*, pages 11–20, 1982.
[10] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *JFP*, 8(4), 1998.
[11] Mark P. Jones. A theory of qualified types. In *ESOP*, 1992.
[12] Mark P. Jones. Simplifying and Improving Qualified Types. Technical Report YALEU/DCS/RR-1040, Yale University, June 1994.
[13] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, 1993.
[14] Oleg Kiselyov and Chung chieh Shan. Embedded probabilistic programming. In *DSL*, 2009.
[15] Edward Kmett. Parameterized monads. http://comonad.com/haskell/monad-param/dist/doc/html/Control-Monad-Parameterized.html.
[16] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *CSFW*, pages 16–27, 2006.
[17] Z. Luo. Coercions in a polymorphic type system. *MSCS*, 18(4), 2008.
[18] Z. Luo and R. Kießling. Coercions in Hindley-Milner systems. In *Proc. of Types*, 2004.
[19] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, 1989.
[20] F. Pottier and V. Simonet. Information flow inference for ML. *TOPLAS*, 25(1):117–158, 2003.
[21] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.
[22] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Haskell*, 2008.
[23] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *J. SAC*, 21(1), 2003.
[24] Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *ICFP*, 2009.

A. Information flow for computations with side effects

In a language with state like ML, side effects may also reveal information (implicit flows), so we must track the security level of the control flow. Both confidentiality and control flow levels should be parameters of the monad. Let us first consider a simple extension of the language where monad constructors are allowed to have type parameters, morphisms can be polymorphic in those type parameters, and every instantiation of a monad constructor is a monad. Through a sample encoding of the information flow monad, we show the limitations of this approach, and motivate an important extension of our system to handle *parameterized monads*.

An information flow state monad $\mathsf{IST}\ l_1\, l_2\, \tau$ is parameterized with security levels l_1 and l_2. It wraps data of type τ with confidentiality level l_2: data should not be disclosed at level lower than l_2 (explicit flows). The program counter of the program should be at most l_1, so that the effects of the data do not leak information on the program counter (implicit flows). For instance, $\mathsf{IST}\ H\ L$ is a type of a computation that has high-security side effects and low secrecy.

Two morphisms between instances of the IST monad describe the *may-flow* order of the security lattice: (1) it is safe to require higher secrecy; (2) it is safe to assume lower level side effects.

$$\begin{array}{lcl} \texttt{pcdown} & : & \forall l.\, \mathsf{IST}\, H\, l \ \rhd\ \mathsf{IST}\, L\, l \\ \texttt{labup} & : & \forall pc.\, \mathsf{IST}\, pc\, L \ \rhd\ \mathsf{IST}\, pc\, H \end{array}$$

We consider an extension of our language with references, which ML typically provides, though we give them special monadic types. A reference to a value of plain type τ with contents of sensitivity l is given type $l\ \tau\ \mathsf{ref}$. A freshly allocated reference, as well as its contents, are at least as sensitive as the current program counter. Reading an l-secret reference yields a l-secret value. Writing an l-secret value has effect l.

$$\begin{array}{lcl} \texttt{alloc} & : & \forall pc, \alpha.\, \alpha \to \mathsf{IST}\ pc\, pc\, (pc\, \alpha\, \mathsf{ref}) \\ \texttt{read} & : & \forall l, \alpha.\, l\, \alpha\, \mathsf{ref} \to \mathsf{IST}\ H\ l\ \alpha \\ \texttt{write} & : & \forall l, \alpha.\, l\, \alpha\, \mathsf{ref} \to \alpha \to \mathsf{IST}\ l\ L\ \mathit{unit} \end{array}$$

Binding computations In this richer setting we might think to write bind as follows:

$$\forall pc, l, \alpha, \beta.\, \mathsf{IST}\ pc\ l\ \alpha \to (\alpha \to \mathsf{IST}\ pc\ l\ \beta) \to \mathsf{IST}\ pc\ l\ \beta$$

Unfortunately, this would be too permissive: it would allow instantiating *pc* as L and l as H, resulting in a bind for monad $\mathsf{IST}\ L\, H$ of type $\mathsf{IST}\ L\, H\, \alpha \to (\alpha \to \mathsf{IST}\ L\, H\, \beta) \to \mathsf{IST}\ L\, H\, \beta$. Such a bind would allow illegal flows because the latent effects of the second computation should not be visible on level less than the confidentiality level of the first computation.

The simplest fix would be to disallow the $\mathsf{IST}\ L\, H$ monad entirely. We can formalize this solution directly in our current system. Moreover, we show a sound encoding into the FlowCaml security type system in Section A.1.

But doing so could be more restrictive than necessary. Alternatively, we could extend our system to support *parameterized monads* [15], which permit composing computations within different monads. We leave extending our system to future work, but, assuming that we have this extension, we conclude by considering how we would take advantage of it.

Stepping back, the most precise type for `bind` would be: *(Bind_ideal)*

$$\begin{array}{l} \forall pc_1, l_1, pc_2, l_2, pc_3, l_3, \alpha, \beta. \\ \quad \mathsf{IST}\ pc_1\ l_1\ \alpha \to (\alpha \to \mathsf{IST}\ pc_2\ l_2\ \beta) \to \mathsf{IST}\ pc_3\ l_3\ \beta \\ \quad \text{with } l_1 \le pc_2,\ pc_3 = pc_1 \sqcap pc_2,\ l_3 = l_1 \sqcup l_2 \end{array}$$

This type rules out leaks, and ensures that the composition has the lower bound of the effects and the upper bound of the secrecy of the composed computations. However, our system cannot directly support such a type for bind because it operates over several monads, not just one, and because it requires additional constraints on labels, which our system does not support.

In a parameterized monadic bind, the monad may vary:

$$\forall \alpha, \beta.\ M_1\ \alpha \to (\alpha \to M_2\ \beta) \to M_3\ \beta$$

This gets us partway toward expressing *(Bind_ideal)*. The next problem is avoid the additional ordering constraints on labels.

Since we consider a fixed security lattice, we can enumerate all possible legal instantiations of labels that would satisfy these constraints, as expressed by Table 1.

For the sake of completeness, we should support binds (1), (5), (8), (2), and (12); note that only the first three of them are properly monadic. Thanks to the morphisms, the argument types of the other possible binds can always be coerced to match one of these five specifications without any loss of precision. For example, (6) could be converted to (8) by applying the morphism `labup` to its first argument. (5) is needed because we would lose precision if we applied either the `pcdown` morphism to both arguments and used (1), or the `labup` morphism and used (8): the resulting computation would then have type $\mathsf{IST}\ L\, L\, \beta$, or $\mathsf{IST}\ H\, H\, \beta$ instead of $\mathsf{IST}\ H\, L\, \beta$. So we end up with five bind operators for the parameterized monad:

#	pc_1	l_1	pc_2	l_2	pc_3	l_3	
1	L	L	L	L	L	L	Needed, monadic
2	L	L	L	H	L	H	Needed
3	L	H	L	L	$-$	$-$	Violates $l_1 \leq pc_2$
4	L	H	L	H	$-$	$-$	Violates $l_1 \leq pc_2$
5	H	L	H	L	H	L	Needed, monadic
6	H	L	H	H	H	H	(8) via labup-1
7	H	H	H	L	H	H	(8) via labup-2
8	H	H	H	H	H	H	Needed, monadic
9	L	L	H	L	L	L	(1) via pcdown-2
10	L	L	H	H	L	H	(2) via pcdown-2
11	L	H	H	L	L	H	(12) via labup-2
12	L	H	H	H	L	H	Needed
13	H	L	L	L	L	L	(1) via pcdown-1
14	H	L	L	H	L	H	(2) via pcdown-1
15	H	H	L	L	$-$	$-$	Violates $l_1 \leq pc_2$
16	H	H	L	H	$-$	$-$	Violates $l_1 \leq pc_2$

Table 1. Applying *(Bind_ideal)* to the simple $H - L$ lattice

bLL:$\forall\alpha,\beta.\ \mathsf{IST}\ L\ L\ \alpha \to (\alpha \to \mathsf{IST}\ L\ L\ \beta) \to \mathsf{IST}\ L\ L\ \beta$
bHL:$\forall\alpha,\beta.\ \mathsf{IST}\ H\ L\ \alpha \to (\alpha \to \mathsf{IST}\ H\ L\ \beta) \to \mathsf{IST}\ H\ L\ \beta$
bHH:$\forall\alpha,\beta.\ \mathsf{IST}\ H\ H\ \alpha \to (\alpha \to \mathsf{IST}\ H\ H\ \beta) \to \mathsf{IST}\ H\ H\ \beta$
bLLLH:$\forall\alpha,\beta.\ \mathsf{IST}\ L\ L\ \alpha \to (\alpha \to \mathsf{IST}\ L\ H\ \beta) \to \mathsf{IST}\ L\ H\ \beta$
bLHHH:$\forall\alpha,\beta.\ \mathsf{IST}\ L\ H\ \alpha \to (\alpha \to \mathsf{IST}\ H\ H\ \beta) \to \mathsf{IST}\ L\ H\ \beta$

Sequencing The monadic bind implies a computational dependence. In particular, for an arbitrary `bind` e_1 e_2, the computation e_1 influences the computation in e_2. Assuming e_1 always influences e_2 can be too restrictive, though, so we also use a monadic *sequence* operator, `seqi` e_1 e_2 which ensures that the result value of e_1 cannot influence e_2.

$$\texttt{seqi} : \forall pc, l_1, l_2, \alpha, \beta.\quad \begin{array}{l}\mathsf{IST}\ pc\ l_1\ \alpha \to \\ \mathsf{IST}\ pc\ l_2\ \beta \to \mathsf{IST}\ pc\ l_2\ \beta\end{array}$$

To accommodate this special situation, we extended the type directed translation. Programmers must reflect the information flow in their source programs, by using either a normal let-binding (which is rewritten into one of the bind forms above) or a sequencing operator `;` which is rewritten to `seqi`.

Example We can use references to rewrite our example of adding the interest due. First we updates a savings account with the interest due, and we update the public date of the last payment. Variable `savings` is secret (having type H *float* ref), `rate` is a public input (having type L *float* ref), and `lastpayment` is a public output (of type L *int* ref). `add_interest` is a pure function computing the new amount of the account after receiving the interests, having type *float* $\to$ *float* $\to$ *float*. Finally, `current_date` returns the current date, having type $unit \to int$.

```
let current = read savings in
let r = read rate in
write savings (add_interest current r));
write lastpayment (current_date ()
```

The rewriting lifts the access to the low secrecy `rate` to compute the high secrecy update for `savings`. Unlike the first two bindings, sequenced using `let` and rewritten into `bindi`, the last low effect write is independent of the preceding computation and so must be composed using `;` to get rewritten into `seqi`. The type of the resulting computation is $\mathsf{IST}\ L\ L\ unit$.

```
seqi (pcdown
 (bHH (read savings) (fun current->
  labup (bHL (read rate) (fun r ->
     (write savings (add_interest current r)))))))
(write lastpayment (current_date ()))
```

Note that if we reorder our code as shown below:

```
let current =
  (write lastpayment (current_date ());
   read savings) in
let r = read rate in
write savings (add_interest current r)
```

the type inference is still possible, but yields a different type $\mathsf{IST}\ L\ H\ unit$ and a different rewriting:

```
bLHHH (seqi (write lastpayment (current_date ()))
        (read savings)) (fun current->
    labup (bHL(read rate) (fun r ->
    (write savings (add_interest current r)))))
```

Even though the example was simple and natural to write, it is notable that the rewriting is quite sophisticated where multiple monads are automatically combined and lifted to the required types.

A.1 Soundness of the encoding

We prove that our information flow monad is sound by translating rewritten programs into FlowCaml [20], an extension of ML for tracking information flow. Informally, our approach involves defining a type-directed translation from the elaborated terms computed by our inference algorithm (terms that are well-typed against the IST signature) to terms in MLIF, the core calculus of FlowCaml. The type system of MLIF is known to correctly enforce noninterference. So, our approach is to show, first, that our translation preserves types, and then to show that the translation is a (weak) simulation. That is, each reduction step taken by a well-typed target term is matched by one or more reductions in MLIF. Since MLIF programs respect non-interference, programs using our encoding do as well.

We include a concise statement of this result here—the full statement, the definition of the translation, and the proof is in Section B. Note, that we have yet to prove the completeness of our approach, i.e., that every well-typed MLIF program is typeable in our system—we leave this to future work.

Lemma 15 (Soundness of the FlowCaml encoding)**.** *For all* Γ, e_0, τ *if* $\Gamma \vdash^{\bullet}_{s} e_0 : \hat{\mathsf{t}} \rightsquigarrow \mathsf{e}$*, where* $\hat{\mathsf{t}} = IST\ pc\ l\ \tau$*, and* $\Gamma \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{MLIF}{\Longrightarrow} e$ *then*

- *(translation preserves types)* $pc; [\![\Gamma]\!] \vdash_{MLIF} e : [\![\hat{\mathsf{t}}]\!]$
- *(translation is a simulation) For all* M *such that* $\Gamma \models \mathsf{M}$*, and* $\Gamma \vdash \mathsf{M} \overset{MLIF}{\Longrightarrow} M$*;*
  if $(\mathsf{M};\mathsf{e}) \longrightarrow (\mathsf{M}';\mathsf{e}')$
  then $(M;e) \longrightarrow_{MLIF} (M';e')$ *and for* Γ' *such that* $\Gamma' \models \mathsf{M}'$*, we have* $\Gamma' \vdash \mathsf{e}' : \hat{\mathsf{t}} \overset{MLIF}{\Longrightarrow} e'$ *and* $\Gamma' \vdash \mathsf{M} \overset{MLIF}{\Longrightarrow} M'$*.*

B. Soundness of the information flow encoding

We prove our information flow encoding sound (noninterferent) by giving a translation of elaborated terms (well-typed against the IST API) into well-typed FlowCaml programs. We show that the translation is a simulation, and hence, we inherit FlowCaml's noninterference property.

We start by recalling in Figure 12 (a fragment of) the type system MLIF, the core calculus of FlowCaml, as presented by Pottier and Simonet [20]. We restrict our attention to the monomorphic calculus, without fix points, exception handlers, and pairs.

Next, in Figure 13, we give a type-directed translation from the target terms (described first in Section 5.1) to MLIF. Our goal is to show that this translation preserves types, and furthermore, that the transaltion is a (strong) simulation with regard to the reduction relations of each language, i.e., each step of reduction of our elaborated terms is matched by a step in MLIF.

We start with a standard lemma regarding the typing of our target language—a standard result for a polymorphic lambda calculus with references.

Lemma 16 (Target typing (preservation of types))**.** *For all* $\Gamma, \mathsf{M}, \mathsf{e}, \hat{\mathsf{t}}$, *where* $\Gamma \models \mathsf{M}$ *and* $\Gamma \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{\mathit{MLIF}}{\Longrightarrow} \_$. *Then, if* $(\mathsf{M}; \mathsf{e}) \longrightarrow (\mathsf{M}'; \mathsf{e}')$ *then for* $\Gamma' \models M'$*, and extension of* Γ*, we have* $\Gamma' \vdash \mathsf{e}' : \mathsf{t}$.

Proof. Standard induction on the structure of the typing derivation. □

Lemma 17 (Target typing (progress))**.** *For all* $\Gamma, \mathsf{M}, \mathsf{e}, \hat{\mathsf{t}}$, *where* $\Gamma \models \mathsf{M}$ *and* $\Gamma \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{\mathit{MLIF}}{\Longrightarrow} \_$. *Then if* $\mathsf{e} \neq \mathsf{v}$, *there exists* M', e' *such that* $(\mathsf{M}; \mathsf{e}) \longrightarrow (\mathsf{M}'; \mathsf{e}')$.

Proof. Standard induction on the structure of the typing derivation. □

Next, we show that our translation to MLIF preserves types.

Lemma 18 (Translation preserves types (values and expressions))**.**

1. *For all* $\Gamma, \mathsf{v}, \mathsf{t}, v$*; if* $\Gamma \vdash \mathsf{v} : \mathsf{t} \overset{\mathit{MLIF}}{\Longrightarrow} v$*, then* $[\![\Gamma]\!] \vdash_{\mathit{MLIF}} v : [\![\mathsf{t}]\!]^L$.
2. *For all* $\Gamma, \mathsf{e}, \hat{\mathsf{t}}, e$*; if* $\Gamma \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{\mathit{MLIF}}{\Longrightarrow} e$*, where* $\hat{\mathsf{t}} = \mathsf{IST}\ pc\ l\ \mathsf{t}$. *Then* $pc; [\![\Gamma]\!] \vdash_{\mathit{MLIF}} e : [\![\hat{\mathsf{t}}]\!]$.

Proof. Proof by mutual induction over the structure of the value and expression translations.

Case (TV-1), (TV-2), (TV-3): Trivial, noting that from the translation of environments we have $[\![x{:}\mathsf{t}]\!] = x{:}[\![\mathsf{t}]\!]^L$

Case (TV-4):
$$\frac{\Gamma, x{:}\mathsf{t} \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{\text{MLIF}}{\Longrightarrow} e \qquad [\![\mathsf{t} \rightarrow \hat{\mathsf{t}}]\!] = t_1 \xrightarrow{pc} t_2}{\Gamma \vdash \lambda x{:}\mathsf{t}.\mathsf{e} : \mathsf{t} \rightarrow \hat{\mathsf{t}} \overset{\text{MLIF}}{\Longrightarrow} \lambda y{:}t_1.y \ \mathsf{case}\ x \succ e\ \mathsf{else\ raise}} \ \text{(TV-4)}$$

Let $\hat{\mathsf{t}} = \mathsf{IST}\ pc\ l\ \mathsf{t}'$.
From the induction hypothesis, we have (IH) $pc; [\![\Gamma, x{:}\mathsf{t}]\!] \vdash e : [\![\hat{\mathsf{t}}']\!]$, where $\hat{\mathsf{t}'} = t_2$.
From the definition of type translation, we have $t_1 = ([\![\mathsf{t}]\!]^L + unit)^{pc}$.
For the goal, we apply (MLIF-T6), picking the unconstrained l in the conclusion to be L.
For the premise, we must now show $pc; [\![\Gamma]\!], y{:}t_1 \vdash_{\text{MLIF}} y\ \mathsf{case}\ x \succ e\ \mathsf{else\ raise} : t_2$.
This follows from an application of (MLIF-T13), using
the variable rule (MLIF-T1) for the first premise,
the rule for raise, (MLIF-T14), for the last premise, which can be given any type.
and, for the key second premise, we must show $pc \sqcup pc; \Gamma, y{:}t_1, x{:}[\![\mathsf{t}]\!]^L \vdash_{\text{MLIF}} e : t_2$.
This follows from the assumption (IH), and an application of MLIF weakening.

Case (TE-5): Easy from the mutual induction hypothesis and a use of (MLIF-T1), which permits values to be typed at any pc.

Case (TE-6):
$$\frac{\begin{array}{ccc}\Gamma \vdash \mathsf{e} : \mathsf{IST}\ pc_1\ l_1\ \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} e & & \Gamma \vdash \mathsf{v} : (\mathsf{t} \rightarrow \mathsf{IST}\ pc_2\ l_2\ \mathsf{t}') \overset{\text{MLIF}}{\Longrightarrow} v \\ l_1 \leq pc_2 & pc \leq pc_1 \sqcap pc_2 & l_1 \sqcup l_2 \leq l\end{array}}{\Gamma \vdash (\mathsf{e}\ \mathit{bind}[pc, l]\ \mathsf{v}) : \mathsf{IST}\ pc\ l\ \mathsf{t}' \overset{\text{MLIF}}{\Longrightarrow} (\mathsf{bind}\ x = e\ \mathsf{in}\ v\ x)} \ \text{(TE-6)}$$

From the induction hypothesis, we have (IH1) $pc_1; [\![\Gamma]\!] \vdash e : ([\![\mathsf{t}]\!]^L + unit)^{l_1}$
And we have (IH2) $[\![\Gamma]\!] \vdash v : (([\![\mathsf{t}]\!]^L + unit)^{l_1} \xrightarrow{pc_2} [\![\mathsf{IST}\ pc\ l\ \mathsf{t}']\!])^L$
For the conclusion, we apply (MLIF-T9), the rule for $\mathsf{bind} = \mathsf{in}$.
For the first premise, we first apply (MLIF-T16) (corresponding to `pcdown`) with (IH1) in the premise.
The second premise of (MLIF-T16) is satisfied by the constraint $pc \leq pc_1$ in the premise of (TE-6).
For the second premise of (MLIF-T9), we first note that from (IH2), we have (using the app rule),
that $\Gamma, x{:}([\![\mathsf{t}]\!]^L + unit)^{l_1} \vdash_{\text{MLIF}} (vx) : \mathsf{IST}\ pc_2\ l_2\ \mathsf{t}'$.
Then, we apply (MLIF-T16) as for the previous premise to conclude.

Case (TE-7): Easy, from the induction hypothesis and an application of (MLIF-T16).

Case (TE-8): Easy, from the induction hypothesis and an application of (MLIF-T15).

Case (TE-9):
$$\frac{\Gamma \vdash \mathsf{v} : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v}{\Gamma \vdash \mathtt{alloc}\ \mathsf{v} : \mathsf{IST}\ pc\ pc\ ((pc, \mathsf{t})\ \mathsf{ref}) \overset{\text{MLIF}}{\Longrightarrow} \mathsf{ref}\ \mathsf{v}} \ \text{(TE-9)}$$

Our goal is to show that $pc; [\![\Gamma]\!] \vdash_{\text{MLIF}} \mathsf{ref}\,\mathsf{v} : [\![\mathsf{t}]\!]^{pc}\,\mathsf{ref}^{pc}$.
From the induction hypothesis, we have (IH) $[\![\Gamma]\!] \vdash v : [\![\mathsf{t}]\!]^{L}$.
For the conclusion, we apply (MLIF-T11) with (IH) in the premise.
The second premise, $pc \lhd [\![\mathsf{t}]\!]^{pc}$ is satisfied immediately.
And, in the goal, we chose $l = pc$.

Case (TE-10): $$\frac{\Gamma \vdash \mathsf{v} : (l, \mathsf{t})\,\mathsf{ref} \overset{\text{MLIF}}{\Longrightarrow} v}{\Gamma \vdash \mathtt{read}\,\mathsf{v} : \mathsf{IST}\ H\ l\ \mathsf{t} = \ !\mathsf{v}}\ \text{(TE-10)}$$

Our goal is to show $H; [\![\Gamma]\!] \vdash_{\text{MLIF}} !\mathsf{v} : [\![\mathsf{t}]\!]^{l}$.
From the induction hypothesis, we have (IH) $[\![\Gamma]\!] \vdash v : [\![\mathsf{t}]\!]^{l}\,\mathsf{ref}^{L}$.
We use (MLIF-T10), using (IH) for the first premise, and noting that $L \lhd t$ for all t.

Case (TE-11): $$\frac{\Gamma \vdash \mathsf{v}_1 : (pc, \mathsf{t})\,\mathsf{ref} \overset{\text{MLIF}}{\Longrightarrow} v_1 \qquad \Gamma \vdash \mathsf{v}_2 : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v_2}{\Gamma \vdash \mathtt{write}\,\mathsf{v}_1\,\mathsf{v}_2 : \mathsf{IST}\ pc\ L\ unit \overset{\text{MLIF}}{\Longrightarrow} v_1 := v_2}\ \text{(TE-11)}$$

From the induction hypothesis, we have (IH1) $[\![\Gamma]\!] \vdash v_1 : [\![\mathsf{t}]\!]^{pc}\,\mathsf{ref}^{L}$.
From the induction hypothesis, we have (IH2) $[\![\Gamma]\!] \vdash v_2 : \mathsf{t}^{L}$.
For the goal, we require $pc; [\![\Gamma]\!] \vdash_{\text{MLIF}} v_1 := v_2 : unit$.
We apply (MLIF-T10) using,
(IH1) for its first premise;
(IH2) for the second premise, preceded by an application of the value subtyping rule (MLIF-T0).
For the third premise, we need to show $pc \sqcup L \leq pc$, which is valid.

□

Lemma 19 (Translation is substitutive)**.**

1. *For all* $\Gamma, x, \mathsf{t}, \Gamma', \mathsf{v}_1, \mathsf{t}_1, v_1, \mathsf{v}, v$. *If* $\Gamma, x{:}\mathsf{t}, \Gamma' \vdash \mathsf{v}_1 : \hat{\mathsf{t}}_1 \overset{MLIF}{\Longrightarrow} v_1$ *and if* $\Gamma \vdash \mathsf{v} : \mathsf{t} \overset{MLIF}{\Longrightarrow} v$. *Then,* $\Gamma, \Gamma' \vdash \mathsf{v}_1[\mathsf{v}/x] : \hat{\mathsf{t}} \overset{MLIF}{\Longrightarrow} v_1[v/x]$.
2. *For all* $\Gamma, x, \mathsf{t}, \Gamma', \mathsf{e}, \hat{\mathsf{t}}, e, \mathsf{v}, v$. *If* $\Gamma, x{:}\mathsf{t}, \Gamma' \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{MLIF}{\Longrightarrow} e$ *and if* $\Gamma \vdash \mathsf{v} : \mathsf{t} \overset{MLIF}{\Longrightarrow} v$. *Then,* $\Gamma, \Gamma' \vdash \mathsf{e}[\mathsf{v}/x] : \hat{\mathsf{t}} \overset{MLIF}{\Longrightarrow} e[v/x]$.

Proof. Straightforward induction over the structure of the translation judgment. □

Lemma 20 (Translation is a simulation)**.** *For all* $\mathsf{M}, \Gamma, \mathsf{e}, \hat{\mathsf{t}}, e$ *such that* $\Gamma \models \mathsf{M}$. *If* $\Gamma \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{MLIF}{\Longrightarrow} e$*; and* $\Gamma \vdash \mathsf{M} \overset{MLIF}{\Longrightarrow} M$*; and* $(\mathsf{M}; \mathsf{e}) \longrightarrow (\mathsf{M}'; \mathsf{e}')$*; then* $(M; e) \longrightarrow^{+}_{MLIF} (M'; e')$ *and for* $\Gamma' \models \mathsf{M}'$, *we have* $\Gamma' \vdash \mathsf{e}' : \hat{\mathsf{t}} \overset{MLIF}{\Longrightarrow} e'$ *and* $\Gamma' \vdash \mathsf{M}' \overset{MLIF}{\Longrightarrow} M'$.

Proof. By induction over the structure of the translation judgment. A key invariant (INL) is showing that if $\Gamma \vdash \mathsf{v} : \mathsf{IST}\ pc\ l\ \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v$, then $v = \mathsf{inl}\ v'$, which ensures that the translated programs never raise failures.

Case TE-5: $\mathsf{mreturn}[pc, l]\mathsf{v}$ is a value form; and so is $\mathsf{inl}\ v$. Neither steps. Note that the invariant (INL) is true.

Case TE-6: In the case where we have $\mathsf{e}\ \mathit{bind}[pc, l]\ \mathsf{v}$, for e not a value, we simply use the induction hypothesis.
When e is a value, appealing to progress, we have a reduction using (E6).
From the invariant (INL) we have that $\mathsf{e} = \mathsf{mreturn}[pc, l][pc, l]\ \mathsf{v}'$ and that $\mathsf{v}' \overset{\text{MLIF}}{\Longrightarrow} \mathsf{inl}\ v'$
And from inversion, we have $\mathsf{v} = \lambda x{:}\mathsf{t}.\mathsf{e}$
And we have a step to $\mathsf{e}[\mathsf{v}'/x]$.
In MLIF we have $\mathsf{bind}\ y = \mathsf{inl}\ v'\ \mathsf{in}\ \lambda y{:}.y\ \mathsf{case}\ x \succ e\ \mathsf{else}\ \mathsf{raise}$
where $\Gamma, x{:}\mathsf{t} \vdash \mathsf{e}[\![]\!]e$.
We take a steps in MLIFusing (MLIF-E3) followed by (MLIF-E1) to obtain $e[v'/x]$.
Where $\mathsf{e}[\mathsf{v}'/x]$ to $e[v'/x]$ are related using substitutivity.
Case TE-7, TE-8: In each case, the application of the morphism $f\ \mathsf{e}$ is translated to $\mathsf{bind}\ x = e\ \mathsf{in}\ x$, where e is translated to e.
When e is a not a value, we step using (E-7). This is matched in MLIF by a reduction (MLIF-E2).
If e is a value, we have a reduction using (E-8) and this is matched in MLIF using (MLIF-E3). We relate the resulting configurations using substitutivity.

Case TE-9, TE-10, TE-11: These rules correspond to the memory operations and here, the two sematics are in immediate correspondence.

□

$$
\begin{array}{lll}
\text{levels} & l, pc & ::= \; L \mid H \\
\text{values} & v & ::= \; x \mid () \mid c \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \lambda x : t.e \\
\text{expressions} & e & ::= \; v \mid v_1\ v_2 \mid \mathsf{bind}\ x{=}e_1\ \mathsf{in}\ e_2 \mid v\ \mathsf{case}\ x \succ e_1\ \mathsf{else}\ e_2 \mid \mathsf{raise} \mid \mathsf{ref}\ v \mid\ !v \mid v_1 := v_2 \\
\text{pre-types} & \hat{t} & ::= \; \mathit{int} \mid (t_1 + t_2) \mid t\ \mathsf{ref} \mid (t_1 \xrightarrow{pc} t_2) \\
\text{types} & t & ::= \; \mathit{unit} \mid \hat{t}^l \\
\text{memory} & M & ::= \; \{\} \mid M \uplus [\mathit{loc} \mapsto v]
\end{array}
$$

$\boxed{\Gamma \vdash_{\text{MLIF}} v : t}$ $\quad \dfrac{\Gamma \vdash_{\text{MLIF}} v : t' \qquad t \leq t'}{\Gamma \vdash_{\text{MLIF}} v : t'}$ (MLIF-T0)

$\dfrac{v \in \{x, \mathit{loc}\} \qquad \Gamma(v) = t}{\Gamma \vdash_{\text{MLIF}} v : t}$ (MLIF-T1) $\quad \dfrac{}{\Gamma \vdash_{\text{MLIF}} () : \mathit{unit}}$ (MLIF-T2) $\quad \dfrac{}{\Gamma \vdash_{\text{MLIF}} c : \mathit{int}^L}$ (MLIF-T3)

$\dfrac{\Gamma \vdash_{\text{MLIF}} v : t_1}{\Gamma \vdash_{\text{MLIF}} \mathsf{inl}\ v : (t_1 + t_2)^l}$ (MLIF-T4) $\quad \dfrac{\Gamma \vdash_{\text{MLIF}} v : t_2}{\Gamma \vdash_{\text{MLIF}} \mathsf{inr}\ v : (t_1 + t_2)^l}$ (MLIF-T5) $\quad \dfrac{pc; \Gamma, x{:}t \vdash_{\text{MLIF}} e : t'}{\Gamma \vdash_{\text{MLIF}} \lambda x{:}t.e : (t \xrightarrow{pc} t')^l}$ (MLIF-T6)

$\boxed{pc; \Gamma \vdash_{\text{MLIF}} e : t}$ $\quad \dfrac{\Gamma \vdash_{\text{MLIF}} v : t}{pc; \Gamma \vdash_{\text{MLIF}} v : t}$ (MLIF-T7) $\quad \dfrac{\Gamma \vdash_{\text{MLIF}} v_1 : (t \xrightarrow{(pc \sqcup l)} t')^l \qquad \Gamma \vdash_{\text{MLIF}} v_2 : t \qquad l \lhd t}{pc; \Gamma \vdash_{\text{MLIF}} v_1\ v_2 : t'}$ (MLIF-T8)

$\dfrac{pc; \Gamma \vdash_{\text{MLIF}} e : t \qquad pc; \Gamma, x{:}t \vdash_{\text{MLIF}} e' : t'}{pc; \Gamma \vdash_{\text{MLIF}} \mathsf{bind}\ x{=}e\ \mathsf{in}\ e' : t'}$ (MLIF-T9) $\quad \dfrac{\Gamma \vdash_{\text{MLIF}} v : t\ \mathsf{ref}^l \qquad l \lhd t}{pc; \Gamma \vdash_{\text{MLIF}}\ !v : t}$ (MLIF-T10)

$\dfrac{\Gamma \vdash_{\text{MLIF}} v : t \qquad pc \lhd t}{pc; \Gamma \vdash_{\text{MLIF}} \mathsf{ref}\ v : t\ \mathsf{ref}^l}$ (MLIF-T11) $\quad \dfrac{\Gamma \vdash_{\text{MLIF}} v_1 : t\ \mathsf{ref}^l \qquad \Gamma \vdash_{\text{MLIF}} v_2 : t \qquad pc \sqcup l \lhd t}{pc; \Gamma \vdash_{\text{MLIF}} v_1 := v_2 : \mathit{unit}}$ (MLIF-T12)

$\dfrac{\Gamma \vdash_{\text{MLIF}} v : (t_1 + t_2)^l \qquad \forall j \in \{1,2\}.(pc(\sqcup)l); \Gamma, x{:}t_i \vdash e_j : t \qquad l \lhd t}{pc; \Gamma \vdash_{\text{MLIF}} v\ \mathsf{case}\ x \succ e_1\ \mathsf{else}\ e_2 : t}$ (MLIF-T13) $\quad \dfrac{}{pc; \Gamma \vdash_{\text{MLIF}} \mathsf{raise} : t}$ (MLIF-T14)

$\dfrac{pc; \Gamma \vdash_{\text{MLIF}} e : t' \qquad t' \leq t}{pc; \Gamma \vdash_{\text{MLIF}} e : t}$ (MLIF-T15) $\quad \dfrac{pc'; \Gamma \vdash_{\text{MLIF}} e : t \qquad pc \leq pc'}{pc; \Gamma \vdash_{\text{MLIF}} e : t}$ (MLIF-T16)

$\boxed{(M; e) \longrightarrow_{\text{MLIF}} (M'; e')}$

$(M; \lambda x{:}t.e\ v) \longrightarrow_{\text{MLIF}} (M; e[v/x])$ (MLIF-E1)

$(M; \mathsf{bind}\ x{=}e_1\ \mathsf{in}\ e_2) \longrightarrow_{\text{MLIF}} (M'; \mathsf{bind}\ x{=}e_1'\ \mathsf{in}\ e_2)$ $\quad$ when $(M; e_1) \longrightarrow_{\text{MLIF}} (M'; e_1')$ (MLIF-E2)

$(M; \mathsf{bind}\ x{=}v_1\ \mathsf{in}\ e_2) \longrightarrow_{\text{MLIF}} (M'; e_2[v_1/x])$ (MLIF-E3)

$(M; \mathsf{inl}\ v\ \mathsf{case}\ x \succ e_1\ \mathsf{else}\ e_2) \longrightarrow_{\text{MLIF}} (M; e_1[v/x])$ (MLIF-E4)

$(M; \mathsf{inr}\ v\ \mathsf{case}\ x \succ e_1\ \mathsf{else}\ e_2) \longrightarrow_{\text{MLIF}} (M; e_2[v/x])$ (MLIF-E5)

$(M; \mathsf{raise}) \longrightarrow_{\text{MLIF}} (M; \mathsf{raise})$ (MLIF-E6)

$(M; \mathsf{ref}\ v) \longrightarrow_{\text{MLIF}} (M \uplus [\mathit{loc} \mapsto v]; \mathit{loc})$ (MLIF-E7)

$(M \uplus [\mathit{loc} \mapsto v']; \mathit{loc} := v) \longrightarrow_{\text{MLIF}} (M \uplus [\mathit{loc} \mapsto v]; ())$ (MLIF-E8)

$(M \uplus [\mathit{loc} \mapsto v]; !\mathit{loc}) \longrightarrow_{\text{MLIF}} (M \uplus [\mathit{loc} \mapsto v]; v)$ (MLIF-E9)

Figure 12. Syntax, typing and dynamic semantics of a fragment of MLIF, the core calculus of FlowCaml

Syntax of elaborated terms well-typed against IST

$$\begin{array}{llll}
\text{levels} & l, pc & ::= & L \mid H \\
\text{values} & \mathsf{v} & ::= & x \mid loc \mid c \mid \lambda x : \mathsf{t.e} \mid \mathsf{mreturn}[pc, l]\ \mathsf{v} \mid \\
\text{expressions} & \mathsf{e} & ::= & \mathsf{v} \mid \mathsf{e}\ bind[pc, l]\ \mathsf{v} \mid f\ \mathsf{e} \mid \mathtt{alloc}\ \mathsf{v} \mid \mathtt{read}\ \mathsf{v} \mid \mathtt{write}\ \mathsf{v}_1\ \mathsf{v}_2 \\
\text{types} & \mathsf{t} & ::= & unit \mid int \mid (l, \mathsf{t})\ \mathsf{ref} \mid \mathsf{t} \to \hat{\mathsf{t}} \\
\text{mon-types} & \hat{\mathsf{t}} & ::= & \mathsf{IST}\ pc\ l\ \mathsf{t} \\
\text{memory} & \mathsf{M} & ::= & \{\} \mid \mathsf{M} \uplus [loc \mapsto \mathsf{v}]
\end{array}$$

$\boxed{[\![\Gamma]\!]}$ Translation of environments $\quad [\![x{:}\mathsf{t}]\!] = x{:}[\![\mathsf{t}]\!]^L \qquad [\![\Gamma, \Gamma']\!] = [\![\Gamma]\!], [\![\Gamma']\!]$

$\boxed{[\![\mathsf{t}]\!]}$ Translation of types to MLIF pre-types

$$[\![unit]\!] = unit \quad [\![int]\!] = int \quad [\![(l, \mathsf{t}\ \mathsf{ref})]\!] = [\![\mathsf{t}]\!]^l\ \mathsf{ref} \quad [\![\mathsf{t} \to \mathsf{IST}\ pc\ l\ \mathsf{t}']\!] = ([\![\mathsf{t}]\!]^L + unit)^{pc} \xrightarrow{pc} [\![\mathsf{IST}\ pc\ l\ \mathsf{t}']\!]$$

$\boxed{[\![\hat{\mathsf{t}}]\!]}$ Translation of mon-types to MLIF types $\quad [\![\mathsf{IST}\ pc\ l\ \mathsf{t}]\!] = ([\![\mathsf{t}]\!] + unit)^l$

$$\boxed{\Gamma \vdash [\![\mathsf{M}]\!] \overset{\text{MLIF}}{\Longrightarrow} M} \qquad \frac{}{\Gamma \vdash [\![\{\}]\!] \overset{\text{MLIF}}{\Longrightarrow} \{\}} \qquad \frac{\Gamma \vdash [\![\mathsf{M}]\!] \overset{\text{MLIF}}{\Longrightarrow} M \quad \Gamma \vdash \mathsf{v} : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v}{\Gamma \vdash [\![\mathsf{M} \uplus [loc \mapsto \mathsf{v}]]\!] \overset{\text{MLIF}}{\Longrightarrow} M, \{loc \mapsto v\}}$$

$\boxed{\Gamma \vdash \mathsf{v} : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v}$ Translation of well-typed non-monadic values

$$\frac{\Gamma(x) = \mathsf{t}}{\Gamma \vdash x : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} x}\ \text{(TV-1)} \qquad \frac{\Gamma(loc) = \mathsf{t}}{\Gamma \vdash loc : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} loc}\ \text{(TV-2)} \qquad \frac{}{\Gamma \vdash c : int \overset{\text{MLIF}}{\Longrightarrow} c}\ \text{(TV-3)}$$

$$\frac{\Gamma, x{:}\mathsf{t} \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{\text{MLIF}}{\Longrightarrow} e \quad [\![\mathsf{t} \to \hat{\mathsf{t}}]\!] = t_1 \xrightarrow{pc} t_2}{\Gamma \vdash \lambda x{:}\mathsf{t.e} : \mathsf{t} \to \hat{\mathsf{t}} \overset{\text{MLIF}}{\Longrightarrow} \lambda y{:}t_1.y\ \mathsf{case}\ x \succ e\ \mathsf{else\ raise}}\ \text{(TV-4)}$$

$\boxed{\Gamma \vdash \mathsf{e} : \hat{\mathsf{t}} \overset{\text{MLIF}}{\Longrightarrow} e}$ Translation of well-typed expressions

$$\frac{\Gamma \vdash \mathsf{v} : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v}{\Gamma \vdash \mathsf{mreturn}[pc, l]v : \mathsf{IST}\ pc\ l\ v \overset{\text{MLIF}}{\Longrightarrow} \mathsf{inl}\ (v)}\ \text{(TE-5)} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{e} : \mathsf{IST}\ pc_1\ l_1\ \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} e \quad \Gamma \vdash \mathsf{v} : (\mathsf{t} \to \mathsf{IST}\ pc_2\ l_2\ \mathsf{t}') \overset{\text{MLIF}}{\Longrightarrow} v \\ l_1 \leq pc_2 \qquad pc \leq pc_1 \sqcap pc_2 \qquad l_1 \sqcup l_2 \leq l\end{array}}{\Gamma \vdash (\mathsf{e}\ bind[pc, l]\ \mathsf{v}) : \mathsf{IST}\ pc\ l\ \mathsf{t}' \overset{\text{MLIF}}{\Longrightarrow} (\mathsf{bind}\ x = e\ \mathsf{in}\ v\ x)}\ \text{(TE-6)}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{IST}\ H\ l\ \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} e}{\Gamma \vdash \mathtt{pcdown}\ \mathsf{e} : \mathsf{IST}\ L\ l\ \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} \mathsf{bind}\ x = e\ \mathsf{in}\ x}\ \text{(TE-7)} \qquad \frac{\Gamma \vdash \mathsf{e} : \mathsf{IST}\ pc\ L\ \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} e}{\Gamma \vdash \mathtt{labup}\ \mathsf{e} : \mathsf{IST}\ pc\ H\ \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} \mathsf{bind}\ x = e\ \mathsf{in}\ x}\ \text{(TE-8)}$$

$$\frac{\Gamma \vdash \mathsf{v} : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v}{\Gamma \vdash \mathtt{alloc}\ \mathsf{v} : \mathsf{IST}\ pc\ pc\ ((pc, \mathsf{t})\ \mathsf{ref}) \overset{\text{MLIF}}{\Longrightarrow} \mathsf{ref\ v}}\ \text{(TE-9)} \qquad \frac{\Gamma \vdash \mathsf{v} : (l, \mathsf{t})\ \mathsf{ref} \overset{\text{MLIF}}{\Longrightarrow} v}{\Gamma \vdash \mathtt{read}\ \mathsf{v} : \mathsf{IST}\ H\ l\ \mathsf{t} = \ !\mathsf{v}}\ \text{(TE-10)}$$

$$\frac{\Gamma \vdash \mathsf{v}_1 : (pc, \mathsf{t})\ \mathsf{ref} \overset{\text{MLIF}}{\Longrightarrow} v_1 \quad \Gamma \vdash \mathsf{v}_2 : \mathsf{t} \overset{\text{MLIF}}{\Longrightarrow} v_2}{\Gamma \vdash \mathtt{write}\ \mathsf{v}_1\ \mathsf{v}_2 : \mathsf{IST}\ pc\ L\ unit \overset{\text{MLIF}}{\Longrightarrow} v_1 := v_2}\ \text{(TE-11)}$$

$\boxed{(\mathsf{M}; \mathsf{e}) \longrightarrow (\mathsf{M}'; \mathsf{e}')}$ Reduction of elaborated terms

$(\mathsf{M}; \lambda x{:}\mathsf{t.e\ v}) \longrightarrow (\mathsf{M}; \mathsf{e}[\mathsf{v}/x])$ (E1)

$(\mathsf{M}; \mathtt{alloc}\ \mathsf{v}) \longrightarrow (\mathsf{M} \uplus [loc \mapsto \mathsf{v}]; loc)$ (E2)

$(\mathsf{M} \uplus [loc \mapsto \mathsf{v}']; \mathtt{write}\ loc\ \mathsf{v}) \longrightarrow (\mathsf{M} \uplus [loc \mapsto \mathsf{v}]; ())$ (E3)

$(\mathsf{M} \uplus [loc \mapsto \mathsf{v}]; \mathtt{read}\ loc) \longrightarrow (M \uplus [loc \mapsto \mathsf{v}]; \mathsf{v})$ (E4)

$(\mathsf{M}; \mathsf{e}_1\ bind[pc, l]\ \mathsf{v}) \longrightarrow (\mathsf{M}; \mathsf{e}'_1\ bind[\hat{\mathsf{t}}; \mathsf{t}]\ \mathsf{v})\text{when}\ (\mathsf{M}; \mathsf{e}_1) \longrightarrow (\mathsf{M}'; \mathsf{e}'_1)$ (E5)

$(\mathsf{M}; (\mathsf{mreturn}[pc, l]\mathsf{v})\ bind[pc, l]\ \lambda x{:}\_.\mathsf{e}) \longrightarrow (\mathsf{M}; \mathsf{e}[\mathsf{v}/x])$ (E6)

$(\mathsf{M}; f\ \mathsf{e}) \longrightarrow (\mathsf{M}'; f\ \mathsf{e}')\ \ \text{when}(\mathsf{M}; \mathsf{e}) \longrightarrow (\mathsf{M}'; \mathsf{e}')$ (E7)

$(\mathsf{M}; f\ \mathsf{v}) \longrightarrow (\mathsf{M}; \mathsf{v})$ (E8)

Figure 13. Translation of elaborated ML terms, types, and environments into MLIF. NB: We identify $unit^l$ with $unit$, for all l. Additionally, note that, by construction, each of the combinators `bLL, bHH, bHL, bLLLH, bLHHH` satisfy the label-ordering premises of (TV-6) .