The marriage of effects and monads

Philip Wadler
Bell Laboratories, Lucent Technologies
wadler@research.bell-labs.com

Abstract

Gifford and others proposed an *effect* typing discipline to delimit the scope of computational effects within a program, while Moggi and others proposed *monads* for much the same purpose. Here we marry effects to monads, uniting two previously separate lines of research. In particular, we show that the type, region, and effect system of Talpin and Jouvelot carries over directly to an analogous system for monads, including a type and effect reconstruction algorithm. The same technique should allow one to transpose any effect systems into a corresponding monad system.

1 Introduction

Computational effects, such as state or continuations, are powerful medicine. If taken as directed they may cure a nasty bug, but one must be wary of the side effects.

For this reason, many researchers in computing seek to exploit the benefits of computational effects while delimiting their scope. Two such lines of research are the *effect* typing discipline, proposed by Gifford and Lucassen [GL86, Luc87], and pursued by Talpin and Jouvelot [TJ92, TJ94] among others, and the use of *monads*, proposed by Moggi [Mog89, Mog91], and pursued by myself [Wad90, Wad92, Wad93, Wad95] among others. Effect systems are typically found in strict languages, such as FX [GJLS87] (a variant of Lisp), while monads are typically found in lazy languages, such as Haskell [PH97].

In my pursuit of monads, I wrote the following:

> ... the use of monads is similar to the use of *effect systems* .... An intriguing question is whether a similar form of type inference could apply to a language based on monads. [Wad92]

Half a decade later, I can answer that question in the affirmative. Goodness knows why it took so long, because the correspondence between effects and monads turns out to be surprisingly close.

The marriage of effects and monads Recall that a monad language introduces a type $\mathtt{T}\,\tau$ to represent a computation that yields a value of type τ and may have side effects. If the call-by-value translation of τ is $\tau^\dagger$, then we have that $(\tau \to \tau')^\dagger$, where $\to$ represents a function that may have side effects, is equal to $\tau^\dagger \to \mathtt{T}\,\tau'^\dagger$, where $\to$ represents a pure function with no side effects.

Recall also that an effect system labels each function with its possible effects, so a function type is now written $\tau \xrightarrow{\sigma} \tau'$, indicating a function that may have effects delimited by σ.

The innovation of this paper is to marry effects to monads, writing $\mathtt{T}^\sigma\,\tau$ for a computation that yields a value in τ and may have effects delimited by σ. Now we have that $(\tau \xrightarrow{\sigma} \tau')^\dagger$ is $\tau^\dagger \to \mathtt{T}^\sigma\,\tau'^\dagger$.

The monad translation offers insight into the structure of the original effect system. In the original system, variables and lambda abstractions are labelled with the empty effect, and applications are labeled with the union of three effects (the effects of evaluating the function, the argument, and the function body). In the monad system, effects appear in just two places: the 'unit' of the monad, which is labeled with the empty effect; and the 'bind' of the monad, which is labeled with the union of two effects. The translation of variables and lambda abstractions introduces 'unit', hence they are labeled with an empty effect; and the translation of application introduces two occurrences of 'bind', hence it is labeled with a union of three effects (each $\cup$ symbol in $\sigma \cup \sigma' \cup \sigma''$ coming from one 'bind').

Transposing effects to monads Several effect systems have been proposed, carrying more or less type information, and dealing with differing computational effects such as state or continuations [GL86, Luc87, JG89, TJ92, TJ94]. Java contains a simple effect system, without effect variables, where each method is labeled with the exceptions it might raise [GJS96].

For concreteness, this paper works with the type, region, and effect system proposed by Talpin and Jouvelot [TJ92], where effects indicate which regions of store are initialised, read, or written. All of Talpin and Jouvelot's results transpose in a straightforward way to a monad formulation. It seems clear that other effect systems can be transposed to monads in a similar way. For instance, Talpin and Jouvelot later proposed a variant system [TJ94], and Tofte and Bikedal [TB98] propose a system for analysing memory allocation, and it appear either of these might work equally well as a basis for a monad formulation.

The system used in [TJ92] allows many effect variables to appear in a union and maintains sets of constraints on effects, while the systems used in [TJ94] and [TB98] requires exactly one effect variable to appear in each union and requires no constraints other than those imposed by unification. Either form of bookkeeping appears to transpose readily to the monad setting.

Applications In Glasgow Haskell, the monad `ST` is used to represent computational effects on state [PW93, LP94]. All effects on state are lumped into a single monad. There is no way to distinguish an operation that reads the store from one that writes the store, or to distinguish operations that write two distinct regions of the store (and hence cannot interfere with each other). The type, region, and effect system of Talpin and Jouvelot addresses precisely this problem, and the system described here could be applied directly to augment the `ST` monad with effects.

Similarly, in Haskell the monad `IO` is used to represent all computational effects that perform input/output [PW93, PH97]. In the Glasgow and Chalmers dialects of Haskell, this includes calls of procedures written in other languages [PW93]. Again, all effects are lumped into a single monad, and again a variant of the system described here could be used to augment the `IO` monad with effects.

Monads labeled with effects can also be applied to optimizing strict languages such as Standard ML. Whereas Haskell requires the user to explicitly introduce monads, Standard ML can be regarded as implicitly introducing a monad everywhere, via Moggi's translation from call-by-value lambda calculus into a monadic metalanguage. The implicit monad of Standard ML incorporates all side effects, including operations on references and input-output, much like a combination of Haskell's `ST` and `IO` monads. As before, labeling the monad with effects can be used to delimit the scope of effects. In particular, where the monad is labeled with the empty effect, the corresponding term is pure and additional optimizations may be applied. Or when the monad reads but does not write the store, certain operations may be commuted. This technique has been applied to intermediate languages for Standard ML compilers by Tolmach [Tol98] and by Benton, Kennedy, and Russel [BKR98]. Our work can be regarded as complementary to theirs: we provide the theory and they provide the practice.

Summary of results Talpin and Jouvelot present (i) a type system with effects, (ii) a semantics, with a proof that types and effects are consistent with the semantics (iii) a type and effect reconstruction algorithm, with a proof that it is sound and complete. We review each of these results, following it by the corresponding result for the monad system. We also recall the call-by-value translation from lambda calculus into a monad language, and show that this translation preserves (i) types, (ii) semantics, and (iii) the principal types derived by the reconstruction algorithms.

By and large, we stick to the notation and formulation of Talpin and Jouvelot [TJ92]. (Along the way, we correct a few infelicities in their paper.) One difference from Talpin and Jouvelot is that they follow the classic work of Tofte [Tof87] and use an operational semantics based on normalistation ('big step'), while we follow the updated approach of Wright and Felleisen [WF94] and use an operational semantics based on reduction ('small step'). As noted by Wright and Felleisen, this leads to a simpler proof: instead of a complex relation between values and types (specified as a greatest fixpoint), we can use the existing type relation (specified by structural induction).

The monad translation we use is standard. It was introduced by Moggi [Mog89, Mog91], and has been further studied by Hatcliff and Danvy [HD94] and Sabry and Wadler [SW97]. Our reduction semantics for the monad is new. It most closely resembles the work of Hatcliff and Danvy, but they did not deal with state and therefore failed to distinguish between pure reductions and those with computational effects, as we do here.

The results are all obtained by straightforward application of well-known techniques, and so we don't bother to give the proofs in detail. The lack of interest in the proofs is part of our point: results for effect systems transpose to monads without much effort.

Value polymorphism Some care is required when mixing computational effects with polymorphic types, lest soundness be forfeit. One approach, due Tofte [Tof87] and used in the original SML [MTH90], introduces 'imperative' type variables in the presence of computational effects. Numerous other approaches have been broached, including some based on effects [Wri92, TJ94]. However, by far the simplest is *value polymorphism.* This approach, noted by Tofte [Tof87], promoted by Wright [Wri95], and used in the revised SML [MTHM97], restricts polymorphism to values, a subclass of expressions that can have no computational effects. Talpin and Jouvelot [TJ92] used value polymorphism, and we do so here.

There is potentially a problem here. Moggi's original monad translation was monomorphic, and it was not entirely obvious how to extend it to polymorphism. I recall a conversation several years ago between Moggi, John Hughes, and myself where we attempted to add polymorphism to the translation and failed. However, we did not consider value polymorphism, which was less popular back then.

One contribution of this paper is to extend the monad translation to include value polymorphism. This extension is presented for the monad system with effects, but applies equally well when effects are absent. In retrospect, the extension seems obvious, since the monad translation handles values specially. One might say that value polymorphism fits monads to a 'T'.

Outline The remainder of this paper is organised as follows. Section 2 introduces the effect type system and the corresponding type system for monads, and introduces the monad translation and shows that it preserves types. Section 3 presents an operational semantics for effects and a corresponding semantics for monads, shows each semantics sound with respect to types, and shows that the monad translation preserves semantics. Section 4 presents a type, region, and effect reconstruction algorithm for effects and a corresponding algorithm for monad, shows each algorithm is sound and complete, and shows that the monad translation relates the two algorithms. Section 5 concludes.

2 Types

This section introduces two languages and their type systems, and the translation between them. The first language, *Effect*, is a call-by-value lambda calculus together with operations on a store, with a type system that includes regions and effects. The second language, *Monad*, is based on Moggi's monad metalanguage together with the same store operations, and with a type system augmented by the same regions and effects. We extend the usual monad translation to include effects, and show that it preserves typings.

2.1 Types for *Effect*

The language *Effect* and its type system is shown in Figure 1. There are two syntactic classes, values and expressions. A value is either an identifier, a lambda abstraction, or a recursive function binding. An expression is either a value, an application, a `let` binding, or one of three primitives operations on the store, which allocate a new reference, get the value of a reference, and set a reference to a new value.

$$
\begin{array}{ll}
x \in \mathit{Id} & \\
v \in \mathit{Val} & v ::= x \mid \lambda x.\, e \mid \mathtt{rec}\, x.\, \lambda x'.\, e \\
e \in \mathit{Exp} & e ::= v \mid e\, e' \mid \mathtt{let}\ x = e\ \mathtt{in}\ e' \mid \mathtt{new}\, e \mid \mathtt{get}\, e \mid \mathtt{set}\, e\, e' \\
\\
r \in \mathit{RegConst} & \\
\gamma \in \mathit{RegVar} & \\
\rho \in \mathit{Region} & \rho ::= \gamma \mid r \\
\varsigma \in \mathit{EffVar} & \\
\sigma \in \mathit{Effect} & \sigma ::= \varsigma \mid \emptyset \mid \sigma \cup \sigma' \mid \mathtt{init}(\rho) \mid \mathtt{read}(\rho) \mid \mathtt{write}(\rho) \\
\\
\alpha \in \mathit{TyVar} & \\
\iota \in \mathit{BaseType} & \\
\tau \in \mathit{Type} & \tau ::= \alpha \mid \iota \mid \tau \xrightarrow{\sigma} \tau' \mid \mathtt{ref}_\rho\, \tau \\
\mathcal{E} \in \mathit{TyEnv} & = \mathit{Id} \to \mathit{Type}
\end{array}
$$

$$
(\mathit{var})\frac{}{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\text{eff}} x : \tau \,!\, \emptyset}
\qquad
(\mathit{rec})\frac{\mathcal{E}_{x,x'} \cup \{x \mapsto \tau \xrightarrow{\sigma} \tau', x' \mapsto \tau\} \vdash_{\text{eff}} e : \tau \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{rec}\, x.\, \lambda x'.\, e : \tau \xrightarrow{\sigma} \tau' \,!\, \emptyset}
$$

$$
(\mathit{abs})\frac{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\text{eff}} e : \tau' \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \lambda x.\, e : \tau \xrightarrow{\sigma} \tau' \,!\, \emptyset}
\qquad
(\mathit{app})\frac{\mathcal{E} \vdash_{\text{eff}} e : \tau \xrightarrow{\sigma''} \tau' \,!\, \sigma \qquad \mathcal{E} \vdash_{\text{eff}} e' : \tau \,!\, \sigma'}{\mathcal{E} \vdash_{\text{eff}} e\, e' : \tau' \,!\, \sigma \cup \sigma' \cup \sigma''}
$$

$$
(\mathit{let})\frac{\mathcal{E} \vdash_{\text{eff}} v : \tau \,!\, \emptyset \qquad \mathcal{E} \vdash_{\text{eff}} e[v/x] : \tau' \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{let}\ x = v\ \mathtt{in}\ e : \tau' \,!\, \sigma}
\qquad
(\mathit{ilet})\frac{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma \qquad \mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\text{eff}} e' : \tau' \,!\, \sigma'}{\mathcal{E} \vdash_{\text{eff}} \mathtt{let}\ x = e\ \mathtt{in}\ e' : \tau' \,!\, \sigma \cup \sigma'}
$$

$$
(\mathit{does})\frac{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma \qquad \sigma' \supseteq \sigma}{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma'}
\qquad
(\mathit{new})\frac{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{new}\, e : \mathtt{ref}_\rho\, \tau \,!\, \sigma \cup \mathtt{init}(\rho)}
$$

$$
(\mathit{get})\frac{\mathcal{E} \vdash_{\text{eff}} e : \mathtt{ref}_\rho\, \tau \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{get}\, e : \tau \,!\, \sigma \cup \mathtt{read}(\rho)}
\qquad
(\mathit{set})\frac{\mathcal{E} \vdash_{\text{eff}} e : \mathtt{ref}_\rho\, \tau \,!\, \sigma \qquad \mathcal{E} \vdash_{\text{eff}} e' : \tau \,!\, \sigma'}{\mathcal{E} \vdash_{\text{eff}} \mathtt{set}\, e\, e' : \tau \,!\, \sigma \cup \sigma' \cup \mathtt{write}(\rho)}
$$

Figure 1: The effect calculus, *Effect*

A region is either a region variable or a region constant. An effect is either the empty effect, the union of two effects, or one of three effects corresponding to the three operations on the store, each of which is labelled with the region of store affected. Equality on effects is modulo the assumption that $\cup$ is associative, commutative, idempotent, and has $\emptyset$ as a unit. We write $\sigma \supseteq \sigma'$ when $\sigma = \sigma \cup \sigma'$.

A type is either a type variable, a base type, a function type (labelled with the effect that occurs when the function is applied), a reference type (labelled with the region in which the reference is located).

A type environment maps identifiers to types. We write $\mathcal{E}_x$ for the environment with x removed from its domain, $\{x \mapsto \tau\}$ for the environment that maps x to τ, and $\mathcal{E} \cup \mathcal{E}'$ for the union of two maps with disjoint domains. (Similar notation will be used later for stores and substitutions.)

A typing $\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma$ indicates that expression e yields a value of type τ and has effect delimited by σ, where the type environment $\mathcal{E}$ maps the free identifiers of e to types.

In the rule for abstraction, (*abs*), the effect is empty because evaluation immediately returns the function, with no side effects; while the effect on the function arrow is the same as the effect for the function body, because applying the function will have the same side effects as evaluating the body. In the rule for application, (*app*), the effect is the union of the effects for evaluating the function, evaluating the argument, and applying the function.

There are two rules for `let` binding, a polymorphic rule for binding values (*let*), and a non-polymorphic rule otherwise (*ilet*). Following Talpin and Jouvelot, we use substitution rather than type schemes to indicate polymorphism. The equivalence of the two forms of specification is well known (e.g., see Mitchell's text [Mit96]). The notation $e[v/x]$ stands for the substitution of value v for identifier x in expression e, with renaming to avoid capture of bound identifiers. Of course, actually performing the substitution is far too expensive when it comes to type reconstruction, and Talpin and Jouvelot's algorithm uses a form of type scheme, as one would expect. Note that if $\mathcal{E} \vdash v : \tau \,!\, \sigma$ then σ must be $\emptyset$.

Rule (*does*) permits a form of subeffecting. Effects indicate an upper bound on the side effects a term may have, and so may always be made larger. The rules for the three primitive operations, (*new*), (*get*), and (*set*), add the corresponding effect to the effects for their arguments. The region in the effect matches the region in the reference type. The (*new*) rule may allocate a new reference in any region.

2.2 Types for *Monad*

Whereas *Effect* is a call-by-value language, with side effects occuring when any expression is evaluated, *Monad* is a call-by-name (or call-by-need) language, with side effects occuring only at top-level. All computations with side effects are

$$
\begin{array}{lll}
e \in \mathit{MonExp} & e ::= x \mid \lambda x.\, e \mid \mathtt{rec}\, x.\, e \mid e\, e' \mid \mathtt{let}\ x = e\ \mathtt{in}\ e' \\
& \quad \mid \mathtt{<}e\mathtt{>} \mid \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e' \mid \mathtt{new}\, e \mid \mathtt{get}\, e \mid \mathtt{set}\, e\, e' \\
\\
\tau \in \mathit{MonType} & \tau ::= \alpha \mid \iota \mid \tau \to \tau' \mid \mathtt{T}^{\sigma}\, \tau \mid \mathtt{ref}_{\rho}\, \tau \\
\mathcal{E} \in \mathit{MonTyEnv} & = \mathit{Id} \to \mathit{MonType}
\end{array}
$$

$$
(\mathit{var})\frac{}{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\mathrm{mon}} x : \tau}
\qquad
(\mathit{abs})\frac{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\mathrm{mon}} e : \tau'}{\mathcal{E} \vdash_{\mathrm{mon}} \lambda x.\, e : \tau \to \tau'}
\qquad
(\mathit{rec})\frac{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\mathrm{mon}} e : \tau}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{rec}\, x.\, e : \tau}
$$

$$
(\mathit{app})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \tau \to \tau' \qquad \mathcal{E} \vdash_{\mathrm{mon}} e' : \tau}{\mathcal{E} \vdash_{\mathrm{mon}} e\, e' : \tau'}
\qquad
(\mathit{let})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \tau \qquad \mathcal{E} \vdash_{\mathrm{mon}} e'[e/x] : \tau'}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{let}\ x = e\ \mathtt{in}\ e' : \tau'}
$$

$$
(\mathit{unit})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \tau}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{<}e\mathtt{>} : \mathtt{T}^{\emptyset}\, \tau}
\qquad
(\mathit{bind})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{T}^{\sigma}\, \tau \qquad \mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\mathrm{mon}} e' : \mathtt{T}^{\sigma'}\, \tau'}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e' : \mathtt{T}^{\sigma \cup \sigma'}\, \tau'}
$$

$$
(\mathit{does})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{T}^{\sigma}\, \tau \qquad \sigma' \sqsupseteq \sigma}{\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{T}^{\sigma'}\, \tau}
\qquad
(\mathit{new})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \tau}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{new}\, e : \mathtt{T}^{\mathtt{init}(\rho)}\, \mathtt{ref}_{\rho}\, \tau}
$$

$$
(\mathit{get})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{ref}_{\rho}\, \tau}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{get}\, e : \mathtt{T}^{\mathtt{read}(\rho)}\, \mathtt{ref}_{\rho}\, \tau}
\qquad
(\mathit{set})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{ref}_{\rho}\, \tau \qquad \mathcal{E} \vdash_{\mathrm{mon}} e' : \tau}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{set}\, e\, e' : \mathtt{T}^{\mathtt{write}(\rho)}\, \tau}
$$

Figure 2: The monad language, *Monad*

represented by the new monad type.

We use call-by-name for monads to stress the relation to Haskell. Like Plotkin's CPS translation, Moggi's monad translation is indifferent: it remains valid whether the monad language uses call-by-value or call-by-name [Plo75, HD94, SW97].

The language *Monad* and its type system is shown in Figure 2. The distinction between values and expressions is no longer relevant, since evaluation has no side effects. Expressions are extended with two new forms for manipulating monads (we will describe these shortly). Regions and effects are as before. The function type $\tau \xrightarrow{\sigma} \tau'$ of before is here broken into the pure function type $\tau \to \tau'$, and the monad type $\mathtt{T}^{\sigma}\, \tau$, representing a computation that yields a value of type τ and has effects delimited by σ.

The monad unit $\mathtt{<}e\mathtt{>}$ denotes the computation that immediately returns the value of e, with no effects. Hence in (*unit*) the effect is empty. The monad bind $\mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e'$ denotes the computation that first performs computation e, binds x to the result, and then performs computation e'. Hence in (*bind*) the effect is the union of the effects of its two subcomputations. (The forms $\mathtt{<}e\mathtt{>}$ and $\mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e'$ are written in Haskell as $\mathtt{return}\, e$ and $e \mathbin{\mathtt{>>=}} \lambda x.\, e'$, respectively.)

Ordinary binding $\mathtt{let}\ x = e\ \mathtt{in}\ e'$ is distinct from monad bind. As shown in rule (*let*), it corresponds to polymorphism. Since expressions have no side effects, there is no need to restrict polymorphism to values. The remaining rules are straightforward adjustments of the previous forms. The three primitive operations, since they involve computational effects, have monad types.

2.3 The translation

The translation from *Effect* to *Monad* is shown in Figure 3. This is just the usual typed call-by-value monad translation. We write $\tau^{\dagger}$ for the translation on types, $v^{\dagger}$ for the translation on values, e^{*} for the translation on expressions, and $\mathcal{E}^{\dagger}$ for the translation on type environments.

As is well known, the monad translation preserves typing, a property that continues hold for our systems with effects.

Proposition 2.1 *(Translation preserves types)*

- *If* $\mathcal{E} \vdash_{\mathrm{eff}} v : \tau \,!\, \emptyset$ *then* $\mathcal{E}^{\dagger} \vdash_{\mathrm{mon}} v^{\dagger} : \tau^{\dagger}$.
- *If* $\mathcal{E} \vdash_{\mathrm{eff}} e : \tau \,!\, \sigma$ *then* $\mathcal{E}^{\dagger} \vdash_{\mathrm{mon}} e^{*} : \mathtt{T}^{\sigma}\, \tau^{\dagger}$.

The proof is by induction on the structure of type derivations. For example, the translation of variables and lambda abstractions introduces 'unit', hence they are labeled with an empty effect; and the translation of application introduces two occurrences of 'bind', hence it is labeled with a union of three effects (each $\cup$ symbol in $\sigma \cup \sigma' \cup \sigma''$ coming from one 'bind').

The translation of `let` works out neatly thanks to the value polymorphism restriction. Whereas the translation of an expression is a monad, and so must be bound with the non-polymorphic monad bind, the translation of a value is not a monad, and can safely be bound with the polymorphic `let`.

The figure also shows the grammar of expressions and types in *Monad* that are in the image of the translation from values, expressions, and types in *Effect*. In the image, application always has values for function and the argument, ordinary `let` always binds to a value, and monad unit always contains a value. This explains the indifference property alluded to earlier: since functions are applied only to values, call-by-value and call-by-name agree in the image of the translation.

$$
\begin{array}{rcl}
\alpha^\dagger & = & \alpha \\
(\iota)^\dagger & = & \iota \\
(\tau \xrightarrow{\sigma} \tau')^\dagger & = & \tau^\dagger \to \mathtt{T}^\sigma\, \tau'^\dagger \\
(\mathtt{ref}_\rho\, \tau)^\dagger & = & \mathtt{ref}_\rho\, \tau^\dagger \\
\\
x^\dagger & = & x \\
(\lambda x.\, e)^\dagger & = & \lambda x.\, e^* \\
(\mathtt{rec}\, x.\, \lambda x'.\, e)^\dagger & = & \mathtt{rec}\, x.\, \lambda x'.\, e^* \\
\\
v^* & = & \mathtt{<}v^\dagger\mathtt{>} \\
(e\, e')^* & = & \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ \mathtt{let}\ x' \Leftarrow e'^*\ \mathtt{in}\ x\, x' \\
(\mathtt{let}\ x = v\ \mathtt{in}\ e)^* & = & \mathtt{let}\ x = v^\dagger\ \mathtt{in}\ e^* \\
(\mathtt{let}\ x = e\ \mathtt{in}\ e')^* & = & \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ e'^* \\
(\mathtt{new}\, e)^* & = & \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ \mathtt{new}\, x \\
(\mathtt{get}\, e)^* & = & \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ \mathtt{get}\, x \\
(\mathtt{set}\, e\, e')^* & = & \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ \mathtt{let}\ x' \Leftarrow e'^*\ \mathtt{in}\ \mathtt{set}\, x\, x' \\
\\
(x_1 : \tau_1, \ldots, x_n : \tau_n)^\dagger & = & x_1 : \tau_1^\dagger, \ldots, x_n : \tau_n^\dagger
\end{array}
$$

$$
\begin{array}{ll}
v \in \mathit{TranVal} & v ::= x \mid \lambda x.\, e \\
e \in \mathit{TranExp} & e ::= v \mid \mathtt{rec}\, x.\, e \mid v\, v' \mid \mathtt{let}\ x = v\ \mathtt{in}\ e' \\
 & \quad \mid \mathtt{<}v\mathtt{>} \mid \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e' \mid \mathtt{new}\, v \mid \mathtt{get}\, v \mid \mathtt{set}\, v\, v' \\
\\
\tau \in \mathit{TranType} & \tau ::= \alpha \mid \iota \mid \tau \to \mathtt{T}^\sigma\, \tau' \mid \mathtt{ref}_\rho\, \tau
\end{array}
$$

Figure 3: Translation from *Effect* to *Monad*

3 Semantics

This section presents operational semantics of the two languages. The reduction system for *Effect* is standard, save for instrumentation to trace operations on the store, which is used to demonstrate consistency between semantics and effects. The reduction system for *Monad* appear to be new, even without the instrumentation. It resembles that of Hatcliff and Danvy [HD94], but differs in distinguishing two sorts of reductions, those that may have side effects and those that do not. For both effects and monads, we show that the type and effect system is sound, modifying the results of Wright and Felleisen [WF94] to take take effects and monads into account. We also show that the translation preserves semantics, in that it preserves instrumented reductions.

3.1 Semantics for *Effect*

The operational semantics for *Effect* are shown in Figure 4. Locations l are a designated subset of the variables. By convention, a location is never used as the bound variable in a lambda or `let` expression. A store s maps locations to values. A trace f is the semantic equivalent of an effect, where regions are replaced by locations. The notation $s_l \cup \{l \mapsto v\}$ stands for the store that maps location l to value v and otherwise behaves like store s (by convention, s does not have l in its domain). An expression e is closed with respect to a store s if the only free variables in e are locations in s, that is, if $\mathrm{fv}(e) \subseteq \mathrm{dom}(s)$. We restrict our attention to reduction states s, e where e is closed with respect to s. A single reduction step is written $s, e \xrightarrow{f}_{\mathrm{eff}} s', e'$, where s, e is the state before the step, f is a trace of the effects of the step, and s', e' is the state after the step.

Rule (*beta*) specifies function application; the language *Effect* is call-by-value as the argument must be a value for the rule to apply. The rule leaves the store unchanged and is labeled with an empty effect. Rules (*rec*) and (*let*) are similar. Rules (*new*), (*get*), and (*set*) perform actions on the store and have corresponding effects. Rule (*app0*) allows reduction of the first part of an application; and once it is reduced to a value, rule (*app1*) allows reduction of the second part; eventually either rule (*beta*) or (*rec*) will apply. The other numbered rules are similar. Finally, rules (*step*), (*refl*), and (*tran*) specify $\twoheadrightarrow$ as the reflexive and transitive closure of $\longrightarrow$.

(Wright and Felleisen, among others, use evaluation contexts as a concise notation that achieves the same effect as the numbered rules here. We'll see why we don't use contexts here in the next subsection.)

We need to relate stores to type environments, and traces to effects. Write $\mathcal{E} \vdash_{\mathrm{eff}} s$ if $\mathrm{dom}(s) = \mathrm{dom}(\mathcal{E})$ and $\mathcal{E} \vdash_{\mathrm{eff}} s(l) : \mathcal{E}(l)$ for each $l \in \mathrm{dom}(s)$. Write $\mathcal{E} \vdash_{\mathrm{eff}} f\,!\,\sigma$ if

- when $\mathtt{init}(l) \subseteq f$ then $\mathcal{E}(l) = \mathtt{ref}_\rho\, \tau$ and $\mathtt{init}(\rho) \subseteq \sigma$,
- when $\mathtt{read}(l) \subseteq f$ then $\mathcal{E}(l) = \mathtt{ref}_\rho\, \tau$ and $\mathtt{read}(\rho) \subseteq \sigma$,
- when $\mathtt{write}(l) \subseteq f$ then $\mathcal{E}(l) = \mathtt{ref}_\rho\, \tau$ and $\mathtt{write}(\rho) \subseteq \sigma$.

Reductions preserve types and are consistent with effects.

Proposition 3.1 *(Subject reduction) If $\mathcal{E} \vdash_{\mathrm{eff}} s$ and $\mathcal{E} \vdash_{\mathrm{eff}} e : \tau\,!\,\sigma$ and $s, e \xrightarrow{f}_{\mathrm{eff}} s', e'$ then there exists a $\mathcal{E}' \supseteq \mathcal{E}$ such that $\mathcal{E}' \vdash_{\mathrm{eff}} s'$ and $\mathcal{E}' \vdash_{\mathrm{eff}} e' : \tau\,!\,\sigma$ and $\mathcal{E}' \vdash_{\mathrm{eff}} f\,!\,\sigma$.*

$$
\begin{array}{lll}
l \in \mathit{Ref} & \subseteq \mathit{Var} \\
s \in \mathit{Store} & = \mathit{Ref} \to \mathit{Value} \\
f \in \mathit{Trace} & f ::= \emptyset \mid f \cup f' \mid \mathtt{init}(l) \mid \mathtt{read}(l) \mid \mathtt{write}(l)
\end{array}
$$

$$
\begin{array}{llcll}
(\mathit{beta}) & s, (\lambda x.\, e)v & \xrightarrow{\emptyset}_{\text{eff}} & s, e[v/x] \\
(\mathit{rec}) & s, (\mathtt{rec}\, x.\, \lambda x'.\, e)v & \xrightarrow{\emptyset}_{\text{eff}} & s, (\lambda x'.\, e[\mathtt{rec}\, x.\, \lambda x'.\, e/x])v \\
(\mathit{let}) & s, \mathtt{let}\ x = v\ \mathtt{in}\ e & \xrightarrow{\emptyset}_{\text{eff}} & s, e[v/x] \\
(\mathit{new}) & s, \mathtt{new}\, v & \xrightarrow{\mathtt{init}(l)}_{\text{eff}} & s \cup \{l \mapsto v\}, x & \text{fresh } l \notin s \\
(\mathit{get}) & s_l \cup \{l \mapsto v\}, \mathtt{get}\, l & \xrightarrow{\mathtt{read}(l)}_{\text{eff}} & s_l \cup \{l \mapsto v\}, v \\
(\mathit{set}) & s_l \cup \{l \mapsto v\}, \mathtt{set}\, l\, v' & \xrightarrow{\mathtt{write}(l)}_{\text{eff}} & s_l \cup \{l \mapsto v'\}, v'
\end{array}
$$

$$
(\mathit{app0})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, e\, e'' \xrightarrow{f}_{\text{eff}} s', e'\, e''}
\qquad
(\mathit{app1})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, v\, e \xrightarrow{f}_{\text{eff}} s, v\, e'}
\qquad
(\mathit{let0})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, \mathtt{let}\ x = e\ \mathtt{in}\ e'' \xrightarrow{f}_{\text{eff}} s', \mathtt{let}\ x = e'\ \mathtt{in}\ e''}
$$

$$
(\mathit{new0})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, \mathtt{new}\, e \xrightarrow{f}_{\text{eff}} s', \mathtt{new}\, e'}
\qquad
(\mathit{get0})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, \mathtt{get}\ e \xrightarrow{f}_{\text{eff}} s', \mathtt{get}\ e'}
$$

$$
(\mathit{set0})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, \mathtt{set}\, e\, e'' \xrightarrow{f}_{\text{eff}} s', \mathtt{set}\, e'\, e''}
\qquad
(\mathit{set1})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, \mathtt{set}\, v\, e \xrightarrow{f}_{\text{eff}} s', \mathtt{set}\, v\, e'}
$$

$$
(\mathit{step})\frac{s, e \xrightarrow{f}_{\text{eff}} s', e'}{s, e \xrightarrow{f}\!\!\!\rightarrow_{\text{eff}} s', e'}
\qquad
(\mathit{refl})\frac{}{s, e \xrightarrow{\emptyset}\!\!\!\rightarrow_{\text{eff}} s, e}
\qquad
(\mathit{tran})\frac{s, e \xrightarrow{f}\!\!\!\rightarrow_{\text{eff}} s', e' \qquad s', e' \xrightarrow{f'}\!\!\!\rightarrow_{\text{eff}} s'', e''}{s, e \xrightarrow{f \cup f'}\!\!\!\rightarrow_{\text{eff}} s'', e''}
$$

Figure 4: Semantics for *Effect*

The proof is by case analysis on the reduction step.

As noted by Wright and Felleisen [WF94], for type soundness one also wants to prove a syntactic equivalent of Milner's slogan 'well typed expressions cannot go wrong'. An expression e is *faulty* if it contains a subexpression in one of the following forms:

$v\, e$, where v is a location,
$\mathtt{get}\, v$, where v is not a location,
$\mathtt{set}\, v\, v'$, where v is not a location.

An evaluation state s, e is *stuck* if there is no f, s', e' such that $s, e \xrightarrow{f}_{\text{eff}} s', e'$, and if e is not a value. Evaluation becomes stuck only for faulty expressions, while well-typed expressions are never faulty.

Proposition 3.2 *(Uniform evaluation) If e is closed over s and s, e is stuck, then e is faulty.*

Proposition 3.3 *(Well-typed expressions are not faulty) If $\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma$, then e is not faulty.*

The first proof is by induction over the structure of e, and the second by case analysis of the definition of faulty.

It follows that evaluation of well-typed terms never gets stuck. Write $s, e \Uparrow^f_{\text{eff}}$ if there is an infinite reduction

$$
s, e \xrightarrow{f_0}_{\text{eff}} s_1, e_1 \xrightarrow{f_1}_{\text{eff}} s_2, e_2 \xrightarrow{f_2}_{\text{eff}} \cdots
$$

with $f = \bigcup f_i$. We have the following corollary.

Proposition 3.4 *(Type soundness) If $\mathcal{E} \vdash_{\text{eff}} s$ and $\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma$ then either*

- *$s, e \Uparrow^f_{\text{eff}}$ and $\mathcal{E} \vdash_{\text{eff}} f \,!\, \sigma$, or*
- *$s, e \xrightarrow{f}_{\text{eff}} s', v$ and $\mathcal{E} \vdash_{\text{eff}} s'$ and $\mathcal{E} \vdash_{\text{eff}} v : \tau \,!\, \emptyset$ and $\mathcal{E} \vdash_{\text{eff}} f \,!\, \sigma$.*

3.2 Semantics for *Monad*

Our reduction system is specialised to the case where the top-level expression has a monad type. Evaluation is call-by-name, and proceeds only to the point where the top-level expression has reduced to a monad unit, forcing all operations on the store to occur. This corresponds to Haskell, where the top-level expression is a monad over the trivial type, `IO ()`, and is executed for its side effects rather than the value retured.

The operational semantics for *Monad* are shown in Figure 5. Locations and traces are as before, but a store now maps locations to expressions. Reductions divide into two sorts. *Pure* reductions do not access the store and have no effect, and are written $e \longrightarrow_{\text{mon}} e'$. *Monadic* reductions are executed at top-level, may access the store and have an effect, and are written $s, e \xrightarrow{f}_{\text{mon}} s, e'$.

Rule (*beta*) specifies function application; the language *Monad* is call-by-name as the argument need not be a

$$
\begin{array}{ll}
l \in \mathit{Ref} & \subseteq \mathit{Var} \\
s \in \mathit{Store} & = \mathit{Ref} \rightarrow \mathit{MonExp}
\end{array}
$$

$$
\begin{array}{llll}
(\mathit{beta}) & (\lambda x.\, e')e & \longrightarrow_{\mathrm{mon}} & e'[e/x] \\
(\mathit{rec}) & \mathtt{rec}\, x.\, e & \longrightarrow_{\mathrm{mon}} & e[\mathtt{rec}\, x.\, e/x] \\
(\mathit{let}) & \mathtt{let}\ x = e\ \mathtt{in}\ e' & \longrightarrow_{\mathrm{mon}} & e'[e/x] \\
(\mathit{bind}) & s, \mathtt{let}\ x \Leftarrow \mathtt{<}e\mathtt{>}\ \mathtt{in}\ e' & \xrightarrow{\emptyset}_{\mathrm{mon}} & s, e'[e/x] \\
(\mathit{new}) & s, \mathtt{new}\, e & \xrightarrow{\mathtt{init}(l)}_{\mathrm{mon}} & s \cup \{l \mapsto e\}, \mathtt{<}l\mathtt{>} \quad \text{fresh } l \notin s \\
(\mathit{get}) & s_l \cup \{l \mapsto e\}, \mathtt{get}\, l & \xrightarrow{\mathtt{read}(l)}_{\mathrm{mon}} & s_l \cup \{l \mapsto e\}, \mathtt{<}e\mathtt{>} \\
(\mathit{set}) & s_l \cup \{l \mapsto e\}, \mathtt{set}\, l\, e' & \xrightarrow{\mathtt{write}(l)}_{\mathrm{mon}} & s_l \cup \{l \mapsto e'\}, \mathtt{<}e'\mathtt{>}
\end{array}
$$

$$
(\mathit{app0})\frac{e \longrightarrow e'}{e\, e'' \longrightarrow e'\, e''} \qquad (\mathit{pure\text{-}step})\frac{e \longrightarrow e'}{e \twoheadrightarrow e'} \qquad (\mathit{pure\text{-}refl})\frac{}{e \twoheadrightarrow e} \qquad (\mathit{pure\text{-}tran})\frac{e \twoheadrightarrow e' \qquad e' \twoheadrightarrow e''}{e \twoheadrightarrow e''}
$$

$$
(\mathit{pure})\frac{e \longrightarrow e'}{s, e \xrightarrow{\emptyset}_{\mathrm{mon}} s, e'} \qquad (\mathit{bind0})\frac{s, e \xrightarrow{f}_{\mathrm{mon}} s', e'}{s, \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e'' \xrightarrow{f}_{\mathrm{mon}} s', \mathtt{let}\ x \Leftarrow e'\ \mathtt{in}\ e''}
$$

$$
(\mathit{new0})\frac{s, e \xrightarrow{f}_{\mathrm{mon}} s', e'}{s, \mathtt{new}\, e \xrightarrow{f}_{\mathrm{mon}} s', \mathtt{new}\, e'} \qquad (\mathit{get0})\frac{e \longrightarrow e'}{s, \mathtt{get}\, e \xrightarrow{\emptyset}_{\mathrm{mon}} s, \mathtt{get}\, e'} \qquad (\mathit{set0})\frac{e \longrightarrow e'}{s, \mathtt{set}\, e\, e'' \xrightarrow{\emptyset}_{\mathrm{mon}} s, \mathtt{set}\, e'\, e''}
$$

$$
(\mathit{step})\frac{s, e \xrightarrow{f}_{\mathrm{mon}} s', e'}{s, e \xrightarrow{f}\!\!\!\twoheadrightarrow_{\mathrm{mon}} s', e'} \qquad (\mathit{refl})\frac{}{s, e \xrightarrow{\emptyset}\!\!\!\twoheadrightarrow_{\mathrm{mon}} s, e} \qquad (\mathit{tran})\frac{s, e \xrightarrow{f}\!\!\!\twoheadrightarrow_{\mathrm{mon}} s', e' \qquad s', e' \xrightarrow{f'}\!\!\!\twoheadrightarrow_{\mathrm{mon}} s'', e''}{s, e \xrightarrow{f \cup f'}\!\!\!\twoheadrightarrow_{\mathrm{mon}} s'', e''}
$$

Figure 5: Semantics for *Monad*

value for the rule to apply. The rule is pure and makes no reference to the store. Rules (*rec*) and (*let*) are similar. Rule (*bind*) simplifies a monadic bind to a monadic unit; it leaves the store unchanged and is labeled with an empty effect. (But it is not a pure operation: this prevents reduction of ill-typed and nonsensical expressions such as $(\mathtt{let}\ x \Leftarrow \mathtt{<}(\lambda y.\, y)\mathtt{>}\ \mathtt{in}\ x)z$, where the monadic expression is not at top-level.) Rules (*new*), (*get*), and (*set*) perform actions on the store and have corresponding effects.

Rule (*app0*) allows reduction of the function part of an application; eventually it will reduce to a lambda and rule (*beta*) will apply. Since *Monad* is call-by-name, the argument of an application is not reduced. Rule (*pure*) allows pure reductions at top-level. (This permits reduction of sensible expressions such as $(\lambda y.\, \mathtt{let}\ x \Leftarrow \mathtt{<}y\mathtt{>}\ \mathtt{in}\ x)z$ where an application yields a monadic expression at top-level.) Rule (*bind0*) allows reduction of the first part of a monad bind; eventually it will simplify to monad unit and rule (*bind*) will apply. Rules (*get0*) and (*put0*) reduce the location argument to an operation on the store. Locations are not monads, so their reductions are pure. Since expressions, not values, are placed in the store there is no need to reduce the argument of `new` or the second argument of `put`. Finally, rules specify $\twoheadrightarrow$ as the reflexive and transitive closure of *red*, for both pure and monadic versions.

(One may formulate the above in terms of evaluation contexts, but it gets messy. It seems to require three sorts of contexts: pure context with pure hole for (*app0*), monad context with monad hole for (*bind0*), monad context with pure hole for (*pure*), (*get0*), and (*set0*). Hence our eschewal of evaluation contexts.)

The relations $\mathcal{E} \vdash_{\mathrm{mon}} s$ and $\mathcal{E} \vdash_{\mathrm{mon}} f\,!\,\sigma$ are defined, *mutatus mutandem*, as for the effect system. Again, reductions preserve types and are consistent with effects.

Proposition 3.5 *(Subject reduction) If $\mathcal{E} \vdash_{\mathrm{mon}} s$ and $\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{T}^{\sigma}\,\tau$ and $s, e \xrightarrow{f}_{\mathrm{mon}} s', e'$ then there exists a $\mathcal{E}' \supseteq \mathcal{E}$ such that $\mathcal{E}' \vdash_{\mathrm{mon}} s'$ and $\mathcal{E}' \vdash_{\mathrm{mon}} e' : \mathtt{T}^{\sigma}\,\tau$ and $\mathcal{E}' \vdash_{\mathrm{mon}} f\,!\,\sigma$.*

Now an expression e is *faulty* if it contains a subexpression in one of the following forms:

$e\,e'$, where e is a location,
$\mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e'$, where e is a lambda or location,
$\mathtt{get}\, e$, where e is a lambda or monad unit,
$\mathtt{set}\, e\, e'$, where e is a lambda or monad unit.

The other definitions and results carry through *mutatus mutandem*.

Proposition 3.6 *(Uniform evaluation) If e is closed over s and s, e is stuck, then e is faulty.*

Proposition 3.7 *(Well-typed expressions are not faulty) If $\mathcal{E} \vdash_{\mathrm{mon}} e : \tau$, then e is not faulty.*

Proposition 3.8 *(Type soundness) If $\mathcal{E} \vdash_{\mathrm{mon}} s$ and $\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{T}^{\sigma}\,\tau$ then either*

$$\theta \in \mathit{Subst} \quad = (\mathit{TyVar} \to \mathit{Type}) \times (\mathit{RegVar} \to \mathit{Region}) \times (\mathit{EffVar} \to \mathit{Effect})$$

$$\begin{array}{rcl}
\mathcal{U}(\alpha, \alpha') & = & \{\alpha \mapsto \alpha'\} \\
\mathcal{U}(\iota, \iota) & = & \mathrm{id} \\
\mathcal{U}(\alpha, \tau) & = & \text{if } \alpha \in \mathrm{fv}(\tau) \text{ then fail else } \{\alpha \mapsto \tau\} \\
\mathcal{U}(\tau, \alpha) & = & \mathcal{U}(\alpha, \tau) \\
\mathcal{U}(\tau_0 \xrightarrow{\varsigma} \tau_1, \tau_0' \xrightarrow{\varsigma'} \tau_1') & = & \text{let } \theta = \{\varsigma \mapsto \varsigma'\}; \theta' = \mathcal{U}(\theta\tau_0, \theta\tau_0'); \theta'' = \mathcal{U}(\theta'\theta\tau_1, \theta'\theta\tau_1') \text{ in } \theta''\theta'\theta \\
\mathcal{U}(\mathtt{ref}_\gamma\, \tau, \mathtt{ref}_{\gamma'}\, \tau') & = & \text{let } \theta = \{\gamma \mapsto \gamma'\}; \theta' = \mathcal{U}(\theta\tau, \theta\tau') \text{ in } \theta'\theta \\
\mathcal{U}(-, -) & = & \text{fail}
\end{array}$$

Figure 6: Unification

$$\begin{array}{ll}
\kappa \in \mathit{Constraint} & = \wp(\mathit{EffVar} \times \mathit{Effect}) \\
\mu \in \mathit{EffModel} & = \mathit{EffVar} \to \mathit{Effect}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{K}(\emptyset) & = & \mathrm{id} \\
\mathcal{K}(\{\varsigma \sqsubseteq \sigma\} \cup \kappa) & = & \text{let } \mu = \mathcal{K}(\kappa) \text{ in } \{\varsigma \mapsto \mu\sigma \setminus \varsigma\}\mu
\end{array}$$

Figure 7: Constraints

- $s, e \Uparrow^{f}_{\mathrm{eff}}$ *and* $\mathcal{E} \vdash_{\mathrm{mon}} f \,!\, \sigma$, *or*
- $s, e \xrightarrow{f}_{\mathrm{eff}} s', \texttt{<}e'\texttt{>}$ *and* $\mathcal{E} \vdash_{\mathrm{mon}} s'$ *and* $\mathcal{E} \vdash_{\mathrm{mon}} e : \tau$ *and* $\mathcal{E} \vdash_{\mathrm{mon}} f \,!\, \sigma$.

3.3 The translation

As is well known, the monad translation preserves semantics, a property that continues to hold for our instrumented semantics. A key to the correspondence is that if a term in *Effect* is translated to *Monad* then the resulting term has subterms of the form $e'\,e$, $\mathtt{let}\ x = e\ \mathtt{in}\ e'$ or $\texttt{<}e\texttt{>}$ only where e is a value.

If s is a store in *Effect*, we write $s^\dagger$ for the corresponding store in *Monad*, with $s^\dagger(l) = (s(l))^\dagger$ for each $l \in \mathrm{dom}(s)$.

Proposition 3.9 *(The translation preserves semantics) If* $s, e \xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{eff}} s', e'$ *then* $s^\dagger, e^* \xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{mon}} s'^\dagger, e'^*$.

The proof is by induction over reduction sequences.

4 Type reconstruction

This section presents type, region, and effect reconstruction algorithms for the two languages. The reconstruction algorithm for *Effect*, due to Talpin and Jouvelot, closely resembles Milner's original type reconstruction algorithm [Mil78]. Effects are handled by accumulating a set of constraints, similar to the handling of subtypes in Mitchell's inference algorithm [Mit91]. It is straightforward to transpose the reconstruction algorithm from *Effect* to *Monad*. Both algorithms are sound and complete, and typings yielded by the two algorithms are related by the translation between the two languages.

4.1 Unification

Substitutions and the unification algorithm are shown in Figure 6. A substitution maps type variables to types, region variables to regions, and effect variables to effects. We write id for the identity substitution.

A central trick in the reconstruction algorithm is to ensure that all effects and regions are represented by variables, to simplify unification. We call a type or substitution *normalised* if the only regions and effects it contains are variables. (One infelicity of Talpin and Jouvelot is that they neglect to mention which types and substitutions are normalised in the statement of their theorems.)

The unification algorithm $\mathcal{U}(\tau, \tau')$ takes two normalised types and returns a normalised substitution θ.

Proposition 4.1 *(Unification)*

- *(Sound) If* $\theta = \mathcal{U}(\tau, \tau')$ *then* $\theta\tau = \theta\tau'$ *(with* θ, τ, τ' *normalised).*
- *(Complete) If* $\theta\tau = \theta\tau'$ *then there exist* θ' *and* θ'' *such that* $\theta' = \mathcal{U}(\tau, \tau')$ *and* $\theta = \theta''\theta'$ *(with* τ, τ', θ' *normalised).*

The proof is standard, as normalisation eliminates any potentially tricky cases.

$$
\begin{array}{lll}
\omega \in \mathit{Var} & = \mathit{TyVar} + \mathit{RegVar} + \mathit{EffVar} \\
\hat{\tau} \in \mathit{TyScheme} & \hat{\tau} ::= \forall\omega_1,\ldots,\omega_n.\,(\tau,\kappa) \\
\mathcal{E} \in \mathit{TyEnv} & = \mathit{Id} \to \mathit{TyScheme}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{I}_{\text{eff}}(\mathcal{E}, x) & = & \text{let } \forall\omega_1,\ldots,\omega_k.\,(\tau,\kappa) = \mathcal{E}(x) \\
& & \quad \text{new } \omega_1',\ldots,\omega_k' \\
& & \quad \theta = \{\omega_1 \mapsto \omega_1', \ldots, \omega_n \mapsto \omega_n'\} \\
& & \text{in } \langle \text{id}, \theta\tau, \emptyset, \theta\kappa \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, \lambda x.\, e) & = & \text{let new } \alpha, \varsigma \\
& & \quad \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}_x \cup \{x \mapsto \alpha\}, e) \\
& & \text{in } \langle \theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa \cup \{\varsigma \sqsupseteq \sigma\} \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, \texttt{rec}\, x.\, \lambda x'.\, e) & = & \text{let new } \alpha, \alpha', \varsigma \\
& & \quad \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}_{x,x'} \cup \{x \mapsto \alpha \xrightarrow{\varsigma} \alpha', x' \mapsto \alpha\}, e) \\
& & \quad \theta' = \mathcal{U}(\theta\alpha', \tau) \\
& & \text{in } \langle \theta'\theta, \theta'\theta(\alpha \xrightarrow{\varsigma} \alpha'), \emptyset, \theta'(\kappa \cup \{\theta\varsigma \sqsupseteq \sigma\}) \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, e\, e') & = & \text{let } \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}, e) \\
& & \quad \langle \theta', \tau', \sigma', \kappa' \rangle = \mathcal{I}_{\text{eff}}(\theta\mathcal{E}, e') \\
& & \quad \text{new } \alpha, \varsigma \\
& & \quad \theta'' = \mathcal{U}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha) \\
& & \text{in } \langle \theta''\theta'\theta, \theta''\alpha, \theta''(\theta'\sigma \cup \sigma' \cup \varsigma), \theta''(\theta'\kappa \cup \kappa') \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, \texttt{let}\ x = v\ \texttt{in}\ e) & = & \text{let } \langle \theta, \tau, \emptyset, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}, v) \\
& & \quad \omega_1,\ldots,\omega_n = \text{fv}(\tau) \cup \text{fv}(\kappa) \setminus \text{fv}(\theta\mathcal{E}) \\
& & \quad \langle \theta', \tau', \sigma, \kappa' \rangle = \mathcal{I}_{\text{eff}}(\theta\mathcal{E}_x \cup \{x \mapsto \forall\omega_1,\ldots,\omega_n.\,(\tau,\kappa)\}, e) \\
& & \text{in } \langle \theta'\theta, \tau', \sigma, \kappa' \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, \texttt{let}\ x = e\ \texttt{in}\ e') & = & \text{let } \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}, e) \\
& & \quad \langle \theta', \tau', \sigma', \kappa' \rangle = \mathcal{I}_{\text{eff}}(\theta\mathcal{E}_x \cup \{x \mapsto \tau\}, e) \\
& & \text{in } \langle \theta'\theta, \tau', \sigma \cup \sigma', \kappa \cup \kappa' \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, \texttt{new}\, e) & = & \text{let } \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}, e) \\
& & \quad \text{new } \gamma \\
& & \text{in } \langle \theta, \texttt{ref}_\gamma\, \tau, \sigma \cup \texttt{init}(\gamma), \kappa \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, \texttt{get}\, e) & = & \text{let } \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}, e) \\
& & \quad \text{new } \alpha, \gamma \\
& & \quad \theta' = \mathcal{U}(\tau, \texttt{ref}_\gamma\, \alpha) \\
& & \text{in } \langle \theta'\theta, \theta'\alpha, \sigma \cup \texttt{read}(\theta'\gamma), \theta'\kappa \rangle \\
\mathcal{I}_{\text{eff}}(\mathcal{E}, \texttt{set}\, e\, e') & = & \text{let } \langle \theta, \tau, \sigma, \kappa \rangle = \mathcal{I}_{\text{eff}}(\mathcal{E}, e) \\
& & \quad \langle \theta', \tau', \sigma', \kappa' \rangle = \mathcal{I}_{\text{eff}}(\theta\mathcal{E}, e) \\
& & \quad \text{new } \gamma \\
& & \quad \theta'' = \mathcal{U}(\theta\tau, \texttt{ref}_\gamma\, \tau') \\
& & \text{in } \langle \theta''\theta'\theta, \theta''\tau', \theta''(\theta'\sigma \cup \sigma' \cup \texttt{write}(\gamma)), \theta''(\theta'\kappa \cup \kappa') \rangle
\end{array}
$$

Figure 8: Type reconstruction for *Effect*

4.2 Constraints

Constraints and the constraint solution algorithm are shown in Figure 7. A set of constraints κ is a set of inequations of the form $\varsigma \sqsupseteq \sigma$, asserting that effect variable ς is bounded below by effect σ.

Constraints always have solutions. A substitution μ taking effect variables to effects models a constraint set κ, written $\mu \models \kappa$, if $\mu\varsigma \sqsupseteq \mu\sigma$ for each inequation $\varsigma \sqsupseteq \sigma$ in κ. (Another infelicity of Talpin and Jouvelot is that they assert the solutions are minimal. The solutions are not minimal in general, though they may be minimal over the domain of κ. However, minimality is irrelevant to the remainder of their results, or to ours.)

The constraint solution algorithm $\mathcal{K}(\kappa)$ takes a constraint set and returns a model μ. It assumes that effect variables on the left hand of constraints in κ are distinct, which can be achieved by merging the two constraints $\varsigma \sqsupseteq \sigma$ and $\varsigma \sqsupseteq \sigma'$ into the equivalent constraint $\varsigma \sqsupseteq \sigma \cup \sigma'$. The notation $\sigma \setminus \varsigma$ stands for the effect σ' such that $\sigma = \sigma' \cup \varsigma$ and ς does not appear in σ'. The result of the algorithm is independent of the order in which the constraints are visited.

$$
\begin{array}{rcl}
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, x) & = & \text{let } \forall\omega_1,\ldots,\omega_k.\,(\tau,\kappa) = \mathcal{E}(x) \\
& & \qquad \text{new } \omega_1',\ldots,\omega_k' \\
& & \qquad \theta = \{\omega_1 \mapsto \omega_1',\ldots,\omega_n \mapsto \omega_n'\} \\
& & \text{in } \langle \mathrm{id}, \theta\tau, \theta\kappa \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \lambda x.\,e) & = & \text{let new } \alpha \\
& & \qquad \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}_x \cup \{x \mapsto \alpha\}, e) \\
& & \text{in } \langle \theta, \theta\alpha \to \tau, \kappa \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \mathtt{rec}\, x.\,e) & = & \text{let new } \alpha \\
& & \qquad \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}_x \cup \{x \mapsto \alpha\}, e) \\
& & \qquad \theta' = \mathcal{U}(\theta\alpha, \tau) \\
& & \text{in } \langle \theta'\theta, \theta'\tau, \theta'\kappa \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e\,e') & = & \text{let } \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e) \\
& & \qquad \langle \theta',\tau',\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\theta\mathcal{E}, e') \\
& & \qquad \text{new } \alpha \\
& & \qquad \theta'' = \mathcal{U}(\theta'\tau, \tau' \to \alpha) \\
& & \text{in } \langle \theta''\theta'\theta, \theta''\alpha, \theta''(\theta'\kappa \cup \kappa') \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \mathtt{let}\ x = e\ \mathtt{in}\ e') & = & \text{let } \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e) \\
& & \qquad \omega_1,\ldots,\omega_n = \mathrm{fv}(\tau) \cup \mathrm{fv}(\kappa) \setminus \mathrm{fv}(\theta\mathcal{E}) \\
& & \qquad \langle \theta',\tau',\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\theta\mathcal{E}_x \cup \{x \mapsto \forall\omega_1,\ldots,\omega_n.\,(\tau,\kappa)\}, e') \\
& & \text{in } \langle \theta'\theta, \tau', \kappa' \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \mathtt{<}e\mathtt{>}) & = & \text{let } \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e) \\
& & \qquad \text{new } \varsigma \\
& & \text{in } \langle \theta, \mathtt{T}^{\varsigma}\,\tau, \kappa \cup \{\varsigma \sqsupseteq \emptyset\} \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e') & = & \text{let } \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e) \\
& & \qquad \text{new } \alpha, \varsigma \\
& & \qquad \theta' = \mathcal{U}(\tau, \mathtt{T}^{\varsigma}\,\alpha) \\
& & \qquad \langle \theta'',\tau',\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E} \cup \{x \mapsto \theta'\alpha\}, e') \\
& & \qquad \text{new } \alpha', \varsigma' \\
& & \qquad \theta''' = \mathcal{U}(\tau', \mathtt{T}^{\varsigma'}\,\alpha') \\
& & \qquad \text{new } \varsigma'' \\
& & \text{in } \langle \theta'''\theta''\theta'\theta, \mathtt{T}^{\varsigma''}\,\theta'''\alpha', \theta'''(\theta''\theta'\kappa \cup \kappa' \cup \{\varsigma'' \sqsubseteq \theta''\theta'\varsigma \cup \varsigma'\}) \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \mathtt{new}\, e) & = & \text{let } \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e) \\
& & \qquad \text{new } \gamma, \varsigma \\
& & \text{in } \langle \theta, \mathtt{T}^{\varsigma}\,\mathtt{ref}_{\gamma}\,\tau, \kappa \cup \{\varsigma \sqsupseteq \mathtt{init}(\gamma)\} \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \mathtt{get}\, e) & = & \text{let } \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e) \\
& & \qquad \text{new } \alpha, \gamma \\
& & \qquad \theta' = \mathcal{U}(\tau, \mathtt{ref}_{\gamma}\,\alpha) \\
& & \qquad \text{new } \varsigma \\
& & \text{in } \langle \theta'\theta, \mathtt{T}^{\varsigma}\,\theta'\alpha, \theta'\kappa \cup \{\varsigma \sqsupseteq \mathtt{read}(\theta'\gamma)\} \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \mathtt{set}\, e\,e') & = & \text{let } \langle \theta,\tau,\sigma,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}, e) \\
& & \qquad \langle \theta',\tau',\sigma',\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\theta\mathcal{E}, e) \\
& & \qquad \text{new } \gamma \\
& & \qquad \theta'' = \mathcal{U}(\theta\tau, \mathtt{ref}_{\gamma}\,\tau') \\
& & \qquad \text{new } \varsigma \\
& & \text{in } \langle \theta''\theta'\theta, \mathtt{T}^{\varsigma}\,(\theta''\tau'), \theta''(\theta'\kappa \cup \kappa') \cup \{\varsigma \sqsupseteq \mathtt{write}(\theta''\gamma)\} \rangle
\end{array}
$$

Figure 9: Type reconstruction for *Monad*

4.3 Reconstruction for *Effect*

Type schemes and the reconstruction algorithm for *Effect* are shown in Figure 8. Type schemes are introduced to represent all possible types associated with a polymorphically bound variable, thus avoiding the computationally infeasible use of substitution suggested by a naive reading of the type rule for `let`. A type scheme has the form $\forall\omega_1,\ldots,\omega_n.\,(\tau,\kappa)$ where each ω is a type, region, or effect variable; the scheme is normalised if τ is normalised. Such a scheme represents all types of the form $\theta\tau$ where $\theta \models \kappa$ and the domain of θ is contained in $\omega_1,\ldots,\omega_n$. Type environments are now taken to map identifiers to type schemes; the environment is normalised if all types in it are normalised.

The reconstruction algorithm $\mathcal{I}_{\mathrm{eff}}(\mathcal{E}, e)$ takes a normalised type environment $\mathcal{E}$ and an expression e, and returns a quadruple $\langle\theta, \tau, \sigma, \kappa\rangle$, with θ and τ normalised. It fails if some unification within it fails. The substitution θ is idempotent, and τ, σ, and κ are invariant under θ.

As shown by Talpin and Jouvelot, the reconstruction algorithm is sound and complete.

Proposition 4.2 *(Type reconstruction, Talpin and Jouvelot)*

- *(Sound) If* $\mathcal{I}_{\text{eff}}(\mathcal{E}, e) = \langle\theta, \tau, \sigma, \kappa\rangle$ *and* $\mu \models \kappa$ *then* $\mu\theta\mathcal{E} \vdash_{\text{eff}} e : \mu\tau \,!\, \mu\sigma$, *with* $\mathcal{E}$, θ, *and* τ *normalised.*
- *(Complete) If* $\theta\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma$ *then* $\mathcal{I}_{\text{eff}}(\mathcal{E}, e) = \langle\theta', \tau', \sigma', \kappa'\rangle$ *and there exists a substitution* θ'' *such that* $\theta\mathcal{E} = \theta''\theta'\mathcal{E}$ *and* $\tau = \theta''\tau'$ *and* $\sigma \supseteq \theta''\sigma'$ *and* $\theta \models \kappa'$, *with* $\mathcal{E}$, θ' *and* τ' *normalised.*

The proof for the first part by induction on the structure of expressions, and for the second by induction on the structure of type derivations. (A final infelicity of Talpin and Jouvelot is that they skip the case of polymorphic 'let' binding, assuming such bindings have been expanded out. Fortunately, it is easy to give a proof for these cases, along the lines of the standard proof in Mitchell's text [Mit96] or a later proof of Talpin and Jouvelot [TJ94]. Alternatively, it is easy to prove a lemma showing that $\mathcal{I}_{\text{eff}}(\mathcal{E}, \texttt{let}\ x = e\ \texttt{in}\ e')$ and $\mathcal{I}_{\text{eff}}(\mathcal{E}, e'[e/x])$ yield the same results, justifying the expansion.)

4.4 Reconstruction for *Monad*

The reconstruction algorithm for *Monad* is shown in Figure 8. The unification algorithm, type schemes, and type environments are as before, with types for *Monad* replacing types for *Effect*, *mutatus mutandem.* Constraints carry over without change.

The reconstruction algorithm $\mathcal{I}_{\text{mon}}(\mathcal{E}, e)$ takes a type environment $\mathcal{E}$ and an expression e, and returns a triple $\langle\theta, \tau, \kappa\rangle$, or fails if some unification within it fails. The reconstruction algorithm is easily transposed to the new setting. It has much the same structure before, the largest difference being that effects are mentioned only in monad types, and effects in types are always represented by variables, so a few extra constraints are required.

It is also easy to transpose the results regarding the algorithm.

Proposition 4.3 *(Type reconstruction)*

- *(Sound) If* $\mathcal{I}_{\text{mon}}(\mathcal{E}, e) = \langle\theta, \tau, \kappa\rangle$ *and* $\mu \models \kappa$ *then* $\mu\theta\mathcal{E} \vdash_{\text{mon}} e : \mu\tau$, *with* $\mathcal{E}$, θ, *and* τ *normalised.*
- *(Complete) If* $\theta\mathcal{E} \vdash_{\text{mon}} e : \tau$ *then* $\mathcal{I}_{\text{mon}}(\mathcal{E}, e) = \langle\theta', \tau', \kappa'\rangle$ *and there exists a substitution* θ'' *such that* $\theta\mathcal{E} = \theta''\theta'\mathcal{E}$ *and* $\tau = \theta''\tau'$ *and* $\theta \models \kappa'$, *with* $\mathcal{E}$, θ', *and* τ' *normalised.*

4.5 Translation

The two reconstruction algorithms yield results that are related by the translation. Write $\kappa \simeq \kappa'$ if for all μ we have $\mu \models \kappa$ if and only if $\mu \models \kappa'$.

Proposition 4.4 *(Translation preserves type reconstruction) If* $\mathcal{I}_{\text{eff}}(\mathcal{E}, e) = \langle\theta, \tau, \sigma, \kappa\rangle$ *and* $\mathcal{I}_{\text{mon}}(\mathcal{E}^{\dagger}, e^{*}) = \langle\theta', \tau', \kappa'\rangle$ *then there exist* ς *and* μ *such that* $\mathtt{T}^{\varsigma}\,\tau^{\dagger} = \tau'$ *and* $\theta = \mu\theta'$ *and* $\sigma = \mu\varsigma$ *and* $\kappa \simeq \mu\kappa'$.

The proof is by induction on the structure of expressions.

5 Conclusions

We have verified the conjecture, first broached half a decade past, that effect systems can be adapted to monads. We have demonstrated this for the specific case of the type, region, and effect system of Talpin and Jouvelot, but it seems clear that any effect system can be adapted to monads in a similar way.

Here are points for future work.

Denotational semantics It is straightforward to provide semantics for effects and monads in a denotational style. In this semantics, the instrumentation can be factored out as a separate monad transformer. The factoring uses the well known result that if $T\,X$ is a monad, then so is $T_A\,X = T\,X \times A$, where A is a monoid. In this case, A is taken to be the monoid of traces, with identity $\emptyset$ and operator $\cup$.

Coherent semantics An alternative approach to denotational semantics might be to eliminate the instrumentation, and associate with each effect σ a different monad T^{σ}. For state, one traditionally defines $T\,X = S \to X \times S$ where the store S is a mapping from locations to values. Here one would define $T^{\sigma}\,\tau = S_{\sigma} \to X \times S^{\sigma}$ where S_{σ} is a store restricted to contain only locations in regions ρ such that $\texttt{read}(\rho)$ is in σ, and S^{σ} is a store restricted to contain only locations in regions ρ such that $\texttt{init}(\rho)$ or $\texttt{write}(\rho)$ is in σ. Corresponding to each effect inclusion $\sigma \subseteq \sigma'$ there should be a monad morphism $T^{\sigma} \to T^{\sigma'}$. In order to ensure coherence in the style of Breazau-Tannen *et al.* [BCGS91], we should expect transitivity of inclusions to correspond to composition of the corresponding morphisms.

A general theory of effects and monads As hypothesised by Moggi and as born out by practice, most computational effects can be viewed as a monad. Does this provide the possibility to formulate a general theory of effects and monads, avoiding the need to create a new effect system for each new effect?

Acknowledgements I thank Mads Tofte, Jon Riecke, Matthias Felleisen, and J.-P. Talpin for comments on earlier drafts of this paper.

References

[BKR98] N. Benton, A. Kennedy, and G. Russell, Compiling Standard ML to Java Bytecodes, *ACM 3'rd International Conference on Functional Programming*, Baltimore, September 1998.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov, Inheritance as explicit coercion, *Information and Computation*, 93(1):172–221, 1991. Reprinted in C. A. Gunter and J. C. Mitchell, editors, *Theoretical aspects of object-oriented programming*, MIT Press, 1994.

[GL86] D. K. Gifford and J. M. Lucassen, Integrating functional and imperative programming, *ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, August 1986.

[GJLS87] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon, FX-87 Reference Manual, Technical report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[GJS96] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Java Series, Sun Microsystems, 1996.

[HD94] J. Hatcliff and O. Danvy, A generic account of continuation-passing styles, *ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

[JG89] P. Jouvelot and D. K. Gifford, Reasoning about continuations with control effects, Technical report MIT/LCS/TM-378, MIT Laboratory for Computer Science, January 1989.

[Jon95] M. P. Jones, Functional programming with overloading and higher-order polymorphism, in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[LP94] J. Launchbury and S. L. Peyton Jones, Lazy functional state threads, *ACM Conference on Programming Language Design and Implementation*, Orlando, Florida, 1994.

[Luc87] J. M. Lucassen, Types and effects, towards the integration of functional and imperative programming, Ph.D. Thesis, Technical report MIT/LCS/TR-408, MIT Laboratory for Computer Science, August 1987.

[Mil78] R. Milner, A theory for type polymorphism in programming, *Journal of Computer and Systems Science*, 17:348–375, 1978.

[Mit91] J. C. Mitchell, Type inference with simple subtypes, *Journal of Functional Programming*, 1(3):245–286, 1991.

[Mit96] J. C. Mitchell, *Foundations for programming languages*, MIT Press, 1996.

[MTH90] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, 1990.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[Mog89] E. Moggi, Computational lambda calculus and monads, *IEEE Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

[Mog91] E. Moggi, Notions of computation and monads, *Information and Computation*, 93(1), 1991.

[PH97] J. Peterson and K. Hammond, editors, Haskell 1.4, a non-strict, purely functional language, Technical report, Yale University, April 1997.

[Plo75] G. Plotkin, Call-by-name, call-by-value, and the λ-calculus, *Theoretical Computer Science*, 1:125–159, 1975.

[PW93] S. L. Peyton Jones and P. Wadler, Imperative functional programming, *ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[SW97] A reflection on call-by-value. Amr Sabry and Philip Wadler. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, November 1997. (An earlier version appeared in *1'st ACM International Conference on Functional Programming*, Philadelphia, May 1996.)

[TJ92] J.-P. Talpin and P. Jouvelot, Polymorphic type, region, and effect inference, *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TJ94] J.-P. Talpin and P. Jouvelot, The type and effect discipline, *Information and Computation*, 111(2):245–296, 1994.

[Tof87] M. Tofte, Operational semantics and polymorphic type inference, PhD Thesis, University of Edinburgh, 1987.

[TB98] M. Tofte and L. Birkedal, A region inference algorithm, *Transactions on Programming Languages and Systems*, November 1998 (to appear).

[Tol98] A. Tolmach, Optimizing ML using a hierarchy of monadic yypes *Workshop on Types in Compilation*, Kyoto, March 1998.

[Wad90] P. Wadler, Comprehending monads. *ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[Wad92] P. Wadler, The essence of functional programming (Invited talk), *ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[Wad93] P. Wadler, Monads for functional programming, in M. Broy, editor, *Program Design Calculi*, NATO ASI Series, Springer Verlag, 1993. Also in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

[Wad95] P. Wadler, How to declare an imperative (Invited talk), *International Logic Programming Symposium*, Portland, Oregon, MIT Press, December 1995.

[WF94] A. Wright and M. Felleisen, A syntactic approach to type soundness, *Information and Computation*, 115(1):38–94, November 1994.

[Wri92] A. Wright, Typing references by effect inference, *4th European Symposium on Programming*, Rennes, France, February 1992, Springer-Verlang LNCS 582.

[Wri95] A. Wright, Simple imperative polymorphism, *Lisp and Symbolic Computation*, 8:343–355, 1995.