The marriage of effects and monads

PHILIP WADLER
Avaya Labs
and
PETER THIEMANN
Universität Freiburg, Germany

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 system into a corresponding monad system.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: monad, effect, type, region, type reconstruction

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; Tal93] among others, and the use of *monads*, proposed by Moggi [Mog89; Mog91], and pursued by Wadler [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 [Has98].

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20TBD ACM 1529-3785/20TBD/0700-0001 $5.00 TBDTBD
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20TBD ACM 1529-3785/20TBD/0700-0001 $5.00

In his pursuit of monads, Wadler 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, we 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. Numerous effect systems have been proposed, carrying more or less type information, and dealing with differing computational effects such as state, continuations, or communication [GL86; Luc87; JG89; TJ92; TJ94; Tal93; NNA97]. Tofte and others propose a system for analysing memory allocation based on effects [TT94; TB98]. 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 a type, region, and effect system based on proposals by Talpin and Jouvelot [TJ92; Tal93; TJ94], where effects indicate which regions of store are initialised, read, or written. 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.

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 monad-based system described here could be applied directly to augment any particular instance of the `ST` monad with effects.

In fact, the `letregion` construct [TT94; TB98] (which was originally named `private` [LG88]) corresponds closely to the `runST` operator in Glasgow Haskell, where the index type plays the role of a region. For the one-region case, this connection has been formalized and proved correct for a simple call-by-value language with `runST` [SS99]. An alternative approach to monadic encapsulation uses a constant `run` with an interesting higher-order type [MP99]. In that approach, the monadic code is abstracted over the monadic operations, which are provided through the `run` constant.

Similarly, in Haskell the monad `IO` is used to represent all computational effects that perform input/output [PW93; PH97]. Dialects of Haskell extend this to call procedures written in other languages [PW93; FLMP99], deal with concurrency [PGF96], or handle exceptions [PRH+99]. 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 expression 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; Tal93; TJ94]. However, we differ in a few particulars.

Talpin and Jouvelot [TJ92] use a simplified treatment of the polymorphic binding `let` $x = v$ `in` e. While their type inference algorithm makes use of type schemes, their correctness proof elides them by assuming that `let` $x = v$ `in` e is expanded to $e[x := v]$. In contrast, here we use standard type schemes throughout.

In subsequent work [TJ94; Tal93], Talpin and Jouvelot also adopt type schemes

throughout. However, they go beyond the present framework in admitting non-values to have polymorphic types. To achieve this goal, they refine effects to also include the type of the effected reference and have the generalization step only abstract those variables that are neither mentioned in the type environment nor in the current effect. In addition, their calculus includes implicit effect masking in the style of `letregion`.

Also, Talpin and Jouvelot follow the classic work of Tofte [Tof87], using an evaluation-style operational semantics ('big step'). In contrast, we follow the 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 did not have to distinguish between pure reductions and those with computational effects, as we do here.

This paper is a revised version of [Wad98]. Changes include the use of type schemes (as in [TJ94; Tal93]); the use of an optimized monad translation (which greatly simplifies the proof that the translation preserves the semantics); the introduction of evaluation contexts in the operational semantics (which follows more closely the development of Wright and Felleisen); and the correction of a some errors in the original (for instance, rule (*get0*) should not have been included in *Monad*).

The results are all obtained by straightforward application of well-known techniques. We do give some proofs to illustrate minor unexpected technical subtleties. However, in general 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 to Tofte [Tof87] and used in the original SML [MTH90], introduces 'imperative' type variables in the presence of computational effects. Numerous other approaches have been suggested, including some based on effects [Wri92; TJ94]. However, by far the simplest is *value polymorphism*. This approach was noted by Tofte [Tof87], promoted by Leroy [Ler93] and Wright [Wri95], and used in the revised SML [MTHM97]. It 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 a potential problem. Moggi's original monad translation was monomorphic. How should it be extended to polymorphism? Several years ago Eugenio Moggi, John Hughes, and Philip Wadler held a discussion where they attempted to add polymorphism to the translation and failed. However, they did not consider value polymorphism, which was less popular back then.

In this paper, we extend the monad translation to include value polymorphism.

The 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. Indeed, similar uses of value polymorphism have been proposed by Harper and Lillibridge [HL93] (for CPS with call/cc) and by Barthe, Hatcliff, and Thiemann [BHT98] (for a configurable monadic metalanguage in the style of Pure Type Systems).

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, it 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 with operations on a store. Its type system includes regions and effects. The second language, *Monad*, is based on Moggi's monadic metalanguage extended 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 three syntactic classes, values, non-values, and expressions. A value is either an identifier, a lambda abstraction, or a recursive function binding. A non-value is either an application, a polymorphic `let` binding for values, a monomorphic `ilet` binding for expressions with imperative effects, or one of three primitive operations on the store, which allocate a new reference, get the value of a reference, or set a reference to a new value. An expression is either a value or a non-value.

A region is either a region variable or a region constant. An effect is either an effect variable, 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 $\sigma \sqsupseteq \sigma'$ when $\sigma = \sigma \cup \sigma'$.

A type is either a type variable, a function type (labelled with the effect that occurs when the function is applied), or a reference type (labelled with the region in which the reference is located). A type scheme $\hat{\tau}$ has the form $\forall \bar{\alpha}, \bar{\gamma}, \bar{\varsigma}.\tau$ where $\bar{\alpha}, \bar{\gamma}, \bar{\varsigma}$ are (possibly empty) sequences of type, region, and effect variables.

A type environment $\mathcal{E}$ maps identifiers to type schemes. We write $\mathcal{E}_x$ for the environment with x removed from its domain, $\{x \mapsto \hat{\tau}\}$ for the environment that

$$
\begin{array}{lll}
x \in \mathit{Id} & & \\
v \in \mathit{Val} & v ::= x \mid \lambda x.\, e \mid \mathtt{rec}\, x.\, \lambda x'.\, e & \\
n \in \mathit{NonVal} & n ::= e\, e' \mid \mathtt{ilet}\, x = e\, \mathtt{in}\, e' \mid \mathtt{let}\, x = v\, \mathtt{in}\, e \mid \mathtt{new}\, e \mid \mathtt{get}\, e \mid \mathtt{set}\, e\, e' & \\
e \in \mathit{Exp} & e ::= v \mid n &
\end{array}
$$

$$
\begin{array}{lll}
r \in \mathit{RegConst} & & \\
\gamma \in \mathit{RegVar} & & \\
\rho \in \mathit{Region} & \rho ::= \gamma \mid r & \\
\varsigma \in \mathit{EffVar} & & \\
\sigma \in \mathit{Effect} & \sigma ::= \varsigma \mid \emptyset \mid \sigma \cup \sigma' \mid \mathtt{init}(\rho) \mid \mathtt{read}(\rho) \mid \mathtt{write}(\rho) &
\end{array}
$$

$$
\begin{array}{lll}
\alpha \in \mathit{TyVar} & & \\
\tau \in \mathit{Type} & \tau ::= \alpha \mid \tau \xrightarrow{\sigma} \tau' \mid \mathtt{ref}_\rho\, \tau & \\
\hat{\tau} \in \mathit{TyScheme} & \hat{\tau} ::= \forall \bar{\alpha}, \bar{\gamma}, \bar{\varsigma}.\tau &
\end{array}
$$

$$
\mathcal{E} \in \mathit{TyEnv} \quad = \quad \mathit{Id} \to \mathit{TyScheme}
$$

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

$$
(\mathit{var})\frac{\hat{\tau} \succeq \tau}{\mathcal{E}_x \cup \{x \mapsto \hat{\tau}\} \vdash_{\text{eff}} x : \tau \,!\, \emptyset} \qquad (\mathit{does})\frac{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma \qquad \sigma' \sqsupseteq \sigma}{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma'}
$$

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

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

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

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

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

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

Fig. 1. The effect calculus, *Effect*

$$
\begin{array}{lll}
e \in \mathit{MonExp} & e ::= x \mid \lambda x.\, e \mid \mathtt{rec}\, x.\, e \mid e\, e' \mid \mathtt{let}\ x = e\ \mathtt{in}\ e' \\
& \quad \mid \texttt{<}e\texttt{>} \mid \mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e' \mid \mathtt{new}\, e \mid \mathtt{get}\, e \mid \mathtt{set}\, e\, e' \\
\tau \in \mathit{MonType} & \tau ::= \alpha \mid \tau \to \tau' \mid \mathtt{T}^{\sigma}\, \tau \mid \mathtt{ref}_{\rho}\, \tau \\
\hat{\tau} \in \mathit{MonTyScheme} & \hat{\tau} ::= \forall \bar{\alpha}, \bar{\gamma}, \bar{\varsigma}.\tau \\
\mathcal{E} \in \mathit{MonTyEnv} & = \quad \mathit{Id} \to \mathit{MonTyScheme} \\
\theta \in \mathit{Subst} & = \quad (\mathit{TyVar} \to \mathit{MonType}) \times (\mathit{RegVar} \to \mathit{Region}) \times (\mathit{EffVar} \to \mathit{Effect})
\end{array}
$$

$$(\mathit{var})\frac{\hat{\tau} \succeq \tau}{\mathcal{E}_x \cup \{x \mapsto \hat{\tau}\} \vdash_{\mathrm{mon}} x : \tau} \qquad (\mathit{does})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{T}^{\sigma}\, \tau \qquad \sigma' \sqsupseteq \sigma}{\mathcal{E} \vdash_{\mathrm{mon}} e : \mathtt{T}^{\sigma'}\, \tau}$$

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

$$(\mathit{let})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \tau \qquad \mathcal{E}_x \cup \{x \mapsto \mathrm{gen}(\mathcal{E}, \tau)\} \vdash_{\mathrm{mon}} e' : \tau'}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{let}\ x = e\ \mathtt{in}\ e' : \tau'}$$

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

$$(\mathit{rec})\frac{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\mathrm{mon}} e : \tau}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{rec}\, x.\, e : \tau} \qquad (\mathit{new})\frac{\mathcal{E} \vdash_{\mathrm{mon}} e : \tau}{\mathcal{E} \vdash_{\mathrm{mon}} \mathtt{new}\, e : \mathtt{T}^{\mathtt{init}(\rho)}\, \mathtt{ref}_{\rho}\, \tau}$$

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

Fig. 2. The monad language, *Monad*

maps identifier x to type scheme $\hat{\tau}$, $\mathcal{E} \cup \mathcal{E}'$ for the union of two maps with disjoint domains, and $\mathcal{E} \supseteq \mathcal{E}'$ when the first map contains the second. Similar notation will be used later for substitutions and stores.

We write $\mathrm{free}(\tau)$, $\mathrm{free}(\hat{\tau})$ and $\mathrm{free}(\mathcal{E})$ for the set of free type, region, and effect variables in a type, type scheme, or type environment. We write $e[x := v]$ for the capture-avoiding substitution of value v for variable x in expression e, and we write $\mathrm{free}(e)$ for the free identifiers of an expression.

We define generic instances and generalization in the usual way. A substitution θ maps type variables to types, region variables to regions, and effect variables to effects. We say that type scheme $\hat{\tau}'$ is a *generic instance* of the type scheme $\hat{\tau}$, written $\hat{\tau} \succeq \hat{\tau}'$, if $\hat{\tau} = \forall \bar{\alpha}, \bar{\gamma}, \bar{\varsigma}.\tau$ and $\hat{\tau}' = \forall \bar{\alpha}', \bar{\gamma}', \bar{\varsigma}'.\tau'$ and there is a substitution

θ such that $\tau' = \theta\tau$, where the domain of θ is restricted to the free variables of τ (that is, $\mathrm{dom}(\theta) = \{\bar{\alpha}, \bar{\gamma}, \bar{\varsigma}\}$) and the bound variables of $\hat{\tau}'$ are not free in $\hat{\tau}$ (that is, $\{\bar{\alpha}', \bar{\gamma}', \bar{\varsigma}'\} \cap \mathrm{free}(\hat{\tau}) = \emptyset$). We say that type scheme $\hat{\tau}$ is the *generalization* of type τ with respect to environment $\mathcal{E}$, written $\hat{\tau} = \mathrm{gen}(\mathcal{E}, \tau)$, if $\hat{\tau} = \forall\bar{\alpha}, \bar{\gamma}, \bar{\varsigma}.\tau$ where $\{\bar{\alpha}, \bar{\gamma}, \bar{\varsigma}\} = \mathrm{free}(\tau) \setminus \mathrm{free}(\mathcal{E})$.

A typing $\mathcal{E} \vdash_{\mathrm{eff}} e : \tau \,!\, \sigma$ 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. 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.

Each `let` binding construct comes with its own typing rule. Rule (*let*) handles polymorphic `let` binding of values, and rule (*ilet*) handles monomorphic `ilet` binding of expressions with imperative effects. Rules (*let*) and (*var*) use type schemes in the usual way.

Rule (*does*) permits a form of subeffecting. Effects indicate an upper bound on the side effects an expression 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 following lemmas are standard results for type and effect systems. They state that syntactic values have no effects and that substitution of values preserves typing. Their proofs are straightforward.

LEMMA 2.1. *(Values are pure) If* $\mathcal{E} \vdash_{\mathrm{eff}} v : \tau \,!\, \sigma$ *then* $\mathcal{E} \vdash_{\mathrm{eff}} v : \tau \,!\, \emptyset$.

LEMMA 2.2. *(Value substitution) Let* $\mathcal{E} = \mathcal{E}_x \cup \{x \mapsto \tau\}$. *If* $\mathcal{E} \vdash_{\mathrm{eff}} e : \tau' \,!\, \sigma$ *and* $\mathcal{E} \vdash_{\mathrm{eff}} v : \tau \,!\, \emptyset$ *then* $\mathcal{E} \vdash_{\mathrm{eff}} e[x := v] : \tau' \,!\, \sigma$.

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 language, with side effects occuring only at top-level. All computations with side effects are represented by the new monad type.

We use call-by-name for monads to stress the relation to Haskell. Like Plotkin's CPS translation, the image of Moggi's monad translation is indifferent: it delivers identical results regardless 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 for polymorphism, since evaluation has no side effects. However, there are monad values which serve as results of a computation in *Monad*. Expressions are extended with two new forms for manipulating monads (we 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 `<`e`>` denotes the computation that immediately returns the value of e, with no effects. Hence in (*unit*) the effect is empty. The monad bind `let` $x \Leftarrow e$ `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 `<`e`>` and `let` $x \Leftarrow e$ `in` e' are written in Haskell as `return` e and e `>>=` $\lambda x.\, e'$, respectively.)

Polymorphic binding `let` $x = e$ `in` e' is distinct from monad bind. 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

Figure 3 shows the translation from *Effect* to *Monad*. It is a typed call-by-value monad translation, similar to the standard translation given by Sabry and Wadler [SW97]. The translation given here is optimized so as not to introduce certain 'administrative' redexes. Although this makes the translation more complex, it simplifies the proof (to be given in the next section) that the translation preserves the semantics.

We write $\tau^\dagger$ for the translation of a type, $v^\dagger$ for the translation of a value, e^* for the translation of an expression, and $\mathcal{E}^\dagger$ for the translation of a type environment.

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

PROPOSITION 2.3. *(Translation preserves types)*

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

The proof is by induction on the structure of type derivations.

The translation of `let` works out neatly thanks to value polymorphism. Whereas the translation of an expression is in a monad, and so must be bound with the non-polymorphic monad bind, the translation of a value is not in 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 (after closure under reduction), and types in *Effect*. In the image, application always has some translated value for function and argument, ordinary `let` always binds to a translated value, and monad unit always contains a translated value.

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* appears 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

$$
\begin{array}{rcl}
\alpha^\dagger & \equiv & \alpha \\
(\tau \xrightarrow{\sigma} \tau')^\dagger & \equiv & \tau^\dagger \to \mathtt{T}^\sigma\, \tau'^\dagger \\
(\mathtt{ref}_\rho\, \tau)^\dagger & \equiv & \mathtt{ref}_\rho\, \tau^\dagger \\
(\forall \bar\alpha, \bar\gamma, \bar\varsigma . \tau)^\dagger & \equiv & \forall \bar\alpha, \bar\gamma, \bar\varsigma . \tau^\dagger
\end{array}
$$

$$
\begin{array}{rcl}
x^\dagger & \equiv & x \\
(\lambda x.\, e)^\dagger & \equiv & \lambda x.\, e^* \\
(\mathtt{rec}\, x.\, \lambda x'.\, e)^\dagger & \equiv & \mathtt{rec}\, x.\, \lambda x'.\, e^*
\end{array}
$$

$$
\begin{array}{rcl}
v^* & \equiv & \mathtt{<}v^\dagger\mathtt{>} \\
(n\, e)^* & \equiv & \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (x\, e)^* \\
(v\, n)^* & \equiv & \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (v\, x)^* \\
(v\, v')^* & \equiv & v^\dagger\, v'^\dagger \\
(\mathtt{let}\ x = v\ \mathtt{in}\ e)^* & \equiv & \mathtt{let}\ x = v^\dagger\ \mathtt{in}\ e^* \\
(\mathtt{ilet}\ x = e\ \mathtt{in}\ e')^* & \equiv & \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ e'^* \\
(\mathtt{new}\, n)^* & \equiv & \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (\mathtt{new}\, x)^* \\
(\mathtt{new}\, v)^* & \equiv & \mathtt{new}\, v^\dagger \\
(\mathtt{get}\, n)^* & \equiv & \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (\mathtt{get}\, x)^* \\
(\mathtt{get}\, v)^* & \equiv & \mathtt{get}\, v^\dagger \\
(\mathtt{set}\, n\, e)^* & \equiv & \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (\mathtt{set}\, x\, e)^* \\
(\mathtt{set}\, v\, n)^* & \equiv & \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (\mathtt{set}\, v\, x)^* \\
(\mathtt{set}\, v\, v')^* & \equiv & \mathtt{set}\, v^\dagger\, v'^\dagger
\end{array}
$$

$$
(x_1 : \tau_1, \ldots, x_n : \tau_n)^\dagger \equiv x_1 : \tau_1^\dagger, \ldots, x_n : \tau_n^\dagger
$$

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

Fig. 3. Translation from *Effect* to *Monad*

results of Wright and Felleisen [WF94] to take effects and monads into account. We also show that the translation preserves semantics, in that it preserves instrumented reduction.

3.1 Semantics for *Effect*

The operational semantics for *Effect* is 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.

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

$$
\begin{array}{ll}
\mathit{EvaluationContext} & E ::= [\,] \mid E\,e \mid v\,E \mid \mathtt{ilet}\ x = E\ \mathtt{in}\ e \\
& \qquad \mid \mathtt{new}\,E \mid \mathtt{get}\,E \mid \mathtt{set}\,E\,e \mid \mathtt{set}\,v\,E
\end{array}
$$

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

$$
(\mathit{context})\frac{s, e \xrightarrow{f}_{\mathrm{eff}} s', e'}{s, E[e] \xrightarrow{f}_{\mathrm{eff}} s', E[e']} \qquad (\mathit{step})\frac{s, e \xrightarrow{f}_{\mathrm{eff}} s', e'}{s, e \xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{eff}} s', e'}
$$

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

Fig. 4. Semantics for *Effect*

If $l \notin \mathrm{dom}(s)$, we write $s \cup \{l \mapsto v\}$ for the store that maps location l to value v and otherwise behaves like s. We let s_l range over stores that do not bind l, that is, $l \notin \mathrm{dom}(s_l)$.

An *evaluation state* is a pair s, e where all free variables of e are locations in s ($\mathrm{free}(e) \subseteq \mathrm{dom}(s)$) and the same holds for all stored values ($\forall l \in \mathrm{dom}(s)$, $\mathrm{free}(s(l)) \subseteq \mathrm{dom}(s)$). 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 effect 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 (*context*) forms the contextual closure of reduction with respect to evaluation contexts. An evaluation context E is an expression with a hole in place of

$$\begin{array}{lcl} l \in \mathit{Location} & \subseteq & \mathit{Id} \\ s \in \mathit{Store} & = & \mathit{Location} \to \mathit{MonExp} \end{array}$$

$$\begin{array}{ll} \text{Monad context} & M ::= [\,] \mid \mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ e \\ \text{Operator context} & O ::= [\,] \mid \mathtt{get}\,[\,] \mid \mathtt{set}\,[\,]\,e \\ \text{Pure context} & P ::= [\,] \mid P\,e \end{array}$$

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

$$(\mathit{monad})\frac{s, e \xrightarrow{f}_{\mathrm{mon}} s', e'}{s, M[e] \xrightarrow{f}_{\mathrm{mon}} s', M[e']} \qquad (\mathit{operator})\frac{e \longrightarrow_{\mathrm{mon}} e'}{s, O[e] \xrightarrow{\emptyset}_{\mathrm{mon}} s, O[e']}$$

$$(\mathit{pure})\frac{e \longrightarrow_{\mathrm{mon}} e'}{P[e] \longrightarrow_{\mathrm{mon}} P[e']} \qquad (\mathit{step})\frac{s, e \xrightarrow{f}_{\mathrm{mon}} s', e'}{s, e \xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{mon}} s', e'}$$

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

Fig. 5. Semantics for *Monad*

the next sub-expression to be evaluated. Defining $E ::= [\,] \mid E\,e \mid v\,E \mid \ldots$ specifies the order of evaluation, since the function in an application must be reduced to a value before the argument is elegible for reduction. The handling of the operations on the store is similar. Finally, rules (*step*), (*refl*), and (*tran*) specify $\xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{eff}}$ as the reflexive and transitive closure of $\xrightarrow{f}_{\mathrm{eff}}$.

There are additional judgements 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, for each $l \in \mathrm{dom}(s)$, if $\mathcal{E}(l) = \mathtt{ref}_\rho\, \tau$ then $\mathcal{E} \vdash_{\mathrm{eff}} s(l) : \tau\,!\,\emptyset$. Write $\mathcal{E} \models_{\mathrm{eff}} f\,!\,\sigma$ if

for each $\mathtt{init}(l)$ in f we have $\mathcal{E}(l) = \mathtt{ref}_\rho\, \tau$ and $\mathtt{init}(\rho) \sqsubseteq \sigma$,

for each $\mathtt{read}(l)$ in f we have $\mathcal{E}(l) = \mathtt{ref}_\rho\,\tau$ and $\mathtt{read}(\rho) \sqsubseteq \sigma$,
for each $\mathtt{write}(l)$ in f we have $\mathcal{E}(l) = \mathtt{ref}_\rho\,\tau$ and $\mathtt{write}(\rho) \sqsubseteq \sigma$.

Write $\mathcal{E} \vdash_{\text{eff}} s, e : \tau\,!\,\sigma$ if s, e is an evaluation state and $\mathcal{E} \vdash_{\text{eff}} s$ and $\mathcal{E} \vdash_{\text{eff}} e : \tau\,!\,\sigma$.
Reduction preserves types and is consistent with effects.

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

The proof is by case analysis on the definition of $s, e \xrightarrow{f}_{\text{eff}} s', e'$.
The form of a value is determined by its type.

LEMMA 3.2. *(Canonical forms)*
Let s, v be an evaluation state and $\mathcal{E} \vdash_{\text{eff}} s, v : \tau\,!\,\emptyset$.

(1) If $\tau = \tau' \xrightarrow{\sigma} \tau''$ then v is either $\lambda x.\, e$ or $\mathtt{rec}\, x.\, \lambda x'.\, e$.
(2) If $\tau = \mathtt{ref}_\rho\,\tau$ then v is a location $l \in \mathrm{dom}(s)$.

The proof is by case analysis on τ.
A well-typed evaluation state is never stuck.

PROPOSITION 3.3. *(Progress)*
Suppose $\mathcal{E} \vdash_{\text{eff}} s, e : \tau\,!\,\sigma$. Either e is a value or there exists s', e' and f such that $s, e \xrightarrow{f}_{\text{eff}} s', e'$.

The proof is by induction on the derivation of $\mathcal{E} \vdash_{\text{eff}} s, e : \tau\,!\,\sigma$. The complete proof is given in Appendix A.1.
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 $f = \bigcup f_i$. Then we have the following.

PROPOSITION 3.4. *(Type soundness)*
If $\mathcal{E} \vdash_{\text{eff}} s, e : \tau\,!\,\sigma$ then there exists $\mathcal{E}' \supseteq \mathcal{E}$ so that either

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

Type soundness is an immediate corrollary of progress.

It is interesting to consider which expressions would give problems during evaluation. An evaluation state s, e is *stuck* if e is not a value and there is no f, s', e' such that $s, e \xrightarrow{f}_{\text{eff}} s', e'$. A simple case analysis shows that a state is stuck if and only if it has one of the following forms:

$e = E[v\, v']$, where v is a location,
$e = E[\mathtt{get}\, v]$, where v is not a location or $v \notin \mathrm{dom}(s)$, or
$e = E[\mathtt{set}\, v\, v']$, where v is not a location or $v \notin \mathrm{dom}(s)$.

Stuck expressions are not typable.

3.2 Semantics for *Monad*

Figure 5 shows the operational semantics for *Monad*. Locations and traces are as before, but a store now maps locations to expressions. There are two notions of reduction.

Pure reductions do not access the store and have no effect. They are written $e \longrightarrow_{\mathrm{mon}} e'$. Rule (*beta*) specifies function application; the language *Monad* is call-by-name as the argument need not be a value for the rule to apply. The rule is pure and makes no reference to the store. Rules (*rec*) and (*let*) are similar.

Monadic reductions perform sequencing and execute the operations on the store. They may have an effect and are written $s, e \xrightarrow{f}_{\mathrm{mon}} s, e'$. Rule (*bind*) simplifies a monadic bind of 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 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.

There are three sorts of contexts, monad contexts M, operator contexts O, and pure contexts P, and three corresponding context rules. Rule (*monad*) forms the contextual closure of monadic reductions over monadic reductions through a monad context M; rule (*operator*) forms the contextual closure of monadic reductions over pure reductions through an operator context O; and rule (*pure*) forms the contextual closure of pure reductions over pure reductions through a pure context P. (These rules permit 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.) Since the monad language is call-by-name, pure contexts do not reduce the argument of a function. Since expressions, not values, are placed in the store, operator contexts neither reduce the argument of `new` nor the second argument of `set`.

As before, rules (*step*), (*refl*), and (*tran*) specify $\xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{mon}}$ as the reflexive and transitive closure of $\xrightarrow{f}_{\mathrm{mon}}$. The relations $\mathcal{E} \vdash_{\mathrm{mon}} s$, $\mathcal{E} \vdash_{\mathrm{mon}} s, e : \tau$, and $\mathcal{E} \models_{\mathrm{mon}} f\,!\,\sigma$ are defined, *mutatis mutandis*, as for *Effect*.

As before, reduction preserves types and is consistent with effects.

PROPOSITION 3.5. *(Subject reduction)*

—*If* $\mathcal{E} \vdash_{\mathrm{mon}} e : \tau$ *and* $e \longrightarrow_{\mathrm{mon}} e'$ *then* $\mathcal{E} \vdash_{\mathrm{mon}} e' : tau$.

—*If* $\mathcal{E} \vdash_{\mathrm{mon}} s, e : \mathtt{T}^{\sigma}\,\tau$ *and* $s, e \xrightarrow{f}_{\mathrm{mon}} s', e'$ *then there exists some* $\mathcal{E}' \supseteq \mathcal{E}$ *such that* $\mathcal{E}' \vdash_{\mathrm{mon}} s', e' : \mathtt{T}^{\sigma}\,\tau$ *and* $\mathcal{E}' \models_{\mathrm{mon}} f\,!\,\sigma$.

The proof is by induction on the definitions of $e \longrightarrow_{\mathrm{mon}} e'$ and $s, e \xrightarrow{f}_{\mathrm{mon}} s', e'$.

We define values for the calculus *Monad* as follows.

$$v \in \mathit{MonVal} \quad v ::= l \mid \lambda x.\, e \mid \mathtt{<}e\mathtt{>}.$$

As before, the form of a value is determined by its type.

LEMMA 3.6. *(Canonical forms) Let* s, v *be an evaluation state and* $\mathcal{E} \vdash_{\mathrm{mon}} s, v : \tau$.

(1) If $\tau = \tau' \to \tau''$ *then* $v = \lambda x.\, e$.

(2) *If* $\tau = \mathtt{T}^{\sigma}\,\tau'$ *then* $v = \mathtt{<}e\mathtt{>}$.
(3) *If* $\tau = \mathtt{ref}_{\rho}\,\tau$ *then* $v = l \in \mathit{Location}$ *and* $l \in \mathrm{dom}(s)$.

The proof is by case analysis on τ.

As before, a well-typed evaluation is never stuck.

PROPOSITION 3.7. *(Progress) Suppose that* $\mathcal{E} \vdash_{\mathrm{mon}} s, e : \tau$. *Then either* $e \in \mathit{MonVal}$ *or there exist* f, s', e' *such that* $s, e \xrightarrow{f}_{\mathrm{mon}} s', e'$.

The proof is by induction on the derivation of $\mathcal{E} \vdash_{\mathrm{eff}} s, e : \tau \,!\, \sigma$.

As before, type soundness is an immediate corrollary of progress.

PROPOSITION 3.8. *(Type soundness) If* $\mathcal{E} \vdash_{\mathrm{mon}} s, e : \tau$ *then there exists* $\mathcal{E}' \supseteq \mathcal{E}$ *so that either*

—$s, e \Uparrow^{f}_{\mathrm{mon}}$ *and* $\mathcal{E}' \models_{\mathrm{mon}} f \,!\, \sigma$, *or*
—$s, e \xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{mon}} s', v$ *and* $\mathcal{E}' \vdash_{\mathrm{mon}} s', v : \tau$ *and* $\mathcal{E}' \models_{\mathrm{mon}} f \,!\, \sigma$.

In both cases, if $\tau = \mathtt{T}^{\sigma'}\,\tau'$ *then* $\sigma = \sigma'$, *otherwise* $\sigma = \emptyset$.

As before, it is interesting to consider which expressions would give problems during evaluation. Now an evaluation state s, e is stuck if one of the following conditions holds:

$e = M[O[P[v\,e]]]$, where v is not a lambda,
$e = M[\mathtt{let}\ x \Leftarrow v\ \mathtt{in}\ e]$, where v is not a monad unit,
$e = M[\mathtt{get}\,v]$, where v is not a location or $v \notin \mathrm{dom}(s)$,
$e = M[\mathtt{set}\,v\,e]$, where v is not a location or $v \notin \mathrm{dom}(s)$.

Again, stuck expressions are not typable.

3.3 Translation

It is well known that the monad translation preserves semantics, and this property continues to hold for the 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 $e'\,e$ or $\mathtt{ilet}\ x = e\ \mathtt{in}\ e'$ or $\mathtt{<}e\mathtt{>}$ only when e is the image of an *Effect*-value, that is, only when e has the form $\dot{v}$.

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

Translation commutes with substitution of *Effect*-values.

LEMMA 3.9. *For all* $e \in \mathit{Exp}$ *and* $v \in \mathit{Val}$, $e^{*}[x := v^{\dagger}] = (e[x := v])^{*}$.

Formally, preservation of semantics corresponds to a simulation result between *Effect* and *Monad*. The reduction of a term in *Effect* runs almost in lock-step with the reduction of its image in *Monad*. Sometimes an additional administrative reduction is required. For instance, consider a reduction in *Effect*,

$$n\,v' \xrightarrow{f}_{\mathrm{eff}} v\,v'$$

where $n^{*} \xrightarrow{f}_{\mathrm{mon}} v^{\dagger}$ (disregarding the store component for simplicity). The image of this reduction in *Monad* is given by

$$(n\,v')^{*} = \mathtt{let}\ x \Leftarrow n^{*}\ \mathtt{in}\ x\,v'^{\dagger} \xrightarrow{f}_{\mathrm{mon}} \mathtt{let}\ x \Leftarrow \mathtt{<}v^{\dagger}\mathtt{>}\ \mathtt{in}\ x\,v'^{\dagger} \longrightarrow_{\mathrm{mon}} v^{\dagger}\,v'^{\dagger} = (v\,v')^{*}$$

which requires one extra administrative reduction. Hence, each reduction in *Effect* gives rise to one or two corresponding reductions in *Monad*.

Another peculiarity arises in the translation of stuck terms. Given a stuck term in *Effect*, its image in *Monad* can sometimes perform one reduction step before it gets stuck, too. This is a consequence of translating the call-by-value recursion operator in *Effect* to a call-by-name recursion operator in *Monad*. If the original term is stuck because of some $\mathtt{rec}\, x.\, \lambda x'.\, e$, then the translated term must unwind the recursion once before it becomes stuck, too.

PROPOSITION 3.10. *(Translation preserves semantics)*

(1) *If* $s, e \xrightarrow{f}_{\mathrm{eff}} s', e'$ *then either* $s^\dagger, e^* \xrightarrow{f}_{\mathrm{mon}} s'^\dagger, e'^*$, *or* $s^\dagger, e^* \xrightarrow{f}_{\mathrm{mon}} s'^\dagger, e_0$ *and* $e_0 \longrightarrow_{\mathrm{mon}} e'^*$.
(2) *If* $s, e \xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{eff}} s', e'$ *then* $s^\dagger, e^* \xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{mon}} s'^\dagger, e'^*$.
(3) *If* $s, e \Uparrow^f_{\mathrm{eff}}$ *then* $s^\dagger, e^* \Uparrow^f_{\mathrm{mon}}$.
(4) *If* s, e *is stuck then either* $s^\dagger, e^*$ *is stuck or* $e^* \longrightarrow_{\mathrm{mon}} e_0$ *where* $s^\dagger, e_0$ *is stuck.*

The proofs of 1 and 4 are by induction on the definitions of $\xrightarrow{f}_{\mathrm{eff}}$, and the proofs of 2 and 3 are by induction on $\xrightarrow{f}\!\!\!\!\rightarrow_{\mathrm{eff}}$. The complete proofs are given in Appendix A.2.

As mentioned in the previous section, the proof is considerably simplified by the use of an optimizing translation. The non-optimized translation introduces many additional administrative redexes, which obscure the correspondence between the effect and monad systems.

Another choice which simplifies the proof is the use of two syntactically distinct forms for `let`, a polymorphic `let` binding of values and a monomorphic `let` binding of expressions with imperative effects. Without this distinction, it is still possible to obtain a simulation result, but at the price of a contrived monad translation.

Note that the image of the translation does not include all possible stuck expressions in *Monad*. In particular, the P context is always trivial because the monad translation moves computations out of the argument positions. The monad translation also guarantees that the reduction (*bind*) never gets stuck.

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

A substitution maps type variables to types, region variables to regions, and effect variables to effects. The substitution id is the identity substitution. Substitutions and the unification algorithms for *Effect* are shown in Figure 6, and the modifications for *Monad* are shown in Figure 7.

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

$$\begin{aligned}
\mathcal{U}_{\mathrm{eff}}(\alpha, \alpha') &= \{\alpha \mapsto \alpha'\} \\
\mathcal{U}_{\mathrm{eff}}(\alpha, \tau) &= \text{if } \alpha \in \mathrm{free}(\tau) \text{ then fail else } \{\alpha \mapsto \tau\} \\
\mathcal{U}_{\mathrm{eff}}(\tau, \alpha) &= \mathcal{U}_{\mathrm{eff}}(\alpha, \tau) \\
\mathcal{U}_{\mathrm{eff}}(\tau_0 \xrightarrow{\varsigma} \tau_1, \tau_0' \xrightarrow{\varsigma'} \tau_1') &= \text{let } \theta = \{\varsigma \mapsto \varsigma'\} \\
&\qquad\quad \theta' = \mathcal{U}_{\mathrm{eff}}(\theta\tau_0, \theta\tau_0') \\
&\qquad\quad \theta'' = \mathcal{U}_{\mathrm{eff}}(\theta'\theta\tau_1, \theta'\theta\tau_1') \\
&\qquad \text{in } \theta''\theta'\theta \\
\mathcal{U}_{\mathrm{eff}}(\mathtt{ref}_{\gamma}\,\tau, \mathtt{ref}_{\gamma'}\,\tau') &= \text{let } \theta = \{\gamma \mapsto \gamma'\} \\
&\qquad\quad \theta' = \mathcal{U}_{\mathrm{eff}}(\theta\tau, \theta\tau') \\
&\qquad \text{in } \theta'\theta \\
\mathcal{U}_{\mathrm{eff}}(-, -) &= \text{fail}
\end{aligned}$$

Fig. 6. Unification of *Effect* Types

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

$$\begin{aligned}
\mathcal{U}_{\mathrm{mon}}(\tau_0 \to \tau_1, \tau_0' \to \tau_1') &= \text{let } \theta = \mathcal{U}_{\mathrm{mon}}(\tau_0, \tau_0') \\
&\qquad\quad \theta' = \mathcal{U}_{\mathrm{mon}}(\theta\tau_1, \theta\tau_1') \\
&\qquad \text{in } \theta'\theta \\
\mathcal{U}_{\mathrm{mon}}(\mathtt{T}^{\varsigma}\,\tau, \mathtt{T}^{\varsigma'}\,\tau') &= \text{let } \theta = \{\varsigma \mapsto \varsigma'\} \\
&\qquad\quad \theta' = \mathcal{U}_{\mathrm{mon}}(\theta\tau, \theta\tau') \\
&\qquad \text{in } \theta'\theta
\end{aligned}$$

Fig. 7. Changes to the unification algorithm for *Monad*

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

$$\begin{aligned}
\mathcal{K}(\emptyset) &= \mathrm{id} \\
\mathcal{K}(\{\varsigma \sqsupseteq \sigma\} \cup \kappa) &= \text{let } \mu = \mathcal{K}(\kappa) \text{ in } \{\varsigma \mapsto \varsigma \cup \mu(\sigma)\} \circ \mu
\end{aligned}$$

Fig. 8. Constraints

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

$$
\begin{array}{rcl}
\mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, x) & = & \text{let new } \bar{\omega}' \\
& & \quad \forall \bar{\omega}.\, (\tau, \kappa') = \mathcal{E}(x) \\
& & \quad \theta = \{\bar{\omega} \mapsto \bar{\omega}'\} \\
& & \text{in } \langle \mathrm{id}, \theta\tau, \emptyset, \kappa \cup \theta\kappa' \rangle \\
\mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, \lambda x.\, e) & = & \text{let new } \alpha, \varsigma \\
& & \quad \langle \theta, \tau, \sigma, \kappa' \rangle = \mathcal{I}_{\mathrm{eff}}(\mathcal{E}_x \cup \{x \mapsto \alpha\}, \kappa, e) \\
& & \text{in } \langle \theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, \kappa' \cup \{\varsigma \sqsupseteq \sigma\} \rangle \\
\mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, \mathtt{rec}\, x.\, \lambda x'.\, e) & = & \text{let new } \alpha, \alpha', \varsigma \\
& & \quad \langle \theta, \tau, \sigma, \kappa' \rangle = \mathcal{I}_{\mathrm{eff}}(\mathcal{E}_{x,x'} \cup \{x \mapsto \alpha \xrightarrow{\varsigma} \alpha', x' \mapsto \alpha\}, \kappa, e) \\
& & \quad \theta' = \mathcal{U}_{\mathrm{eff}}(\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}_{\mathrm{eff}}(\mathcal{E}, \kappa, e\, e') & = & \text{let new } \alpha, \varsigma \\
& & \quad \langle \theta, \tau, \sigma, \kappa' \rangle = \mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, e) \\
& & \quad \langle \theta', \tau', \sigma', \kappa'' \rangle = \mathcal{I}_{\mathrm{eff}}(\theta\mathcal{E}, \kappa', e') \\
& & \quad \theta'' = \mathcal{U}_{\mathrm{eff}}(\theta'\tau, \tau' \xrightarrow{\varsigma} \alpha) \\
& & \text{in } \langle \theta''\theta'\theta, \theta''\alpha, \theta''(\theta'\sigma \cup \sigma' \cup \varsigma), \theta''\kappa'' \rangle \\
\mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, \mathtt{let}\ x = v\ \mathtt{in}\ e) & = & \text{let } \langle \theta, \tau, \emptyset, \kappa' \rangle = \mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, v) \\
& & \quad \bar{\omega} = (\mathrm{free}(\mathcal{K}(\kappa')\tau)) \setminus \mathrm{free}(\mathcal{K}(\kappa')\theta\mathcal{E}) \\
& & \quad \kappa'' = \{\varsigma \sqsupseteq \sigma \in \kappa' \mid \varsigma \in \bar{\omega}\} \\
& & \quad \kappa''' = \kappa' \setminus \kappa'' \\
& & \quad \langle \theta', \tau', \sigma, \kappa'''' \rangle = \mathcal{I}_{\mathrm{eff}}(\theta\mathcal{E}_x \cup \{x \mapsto \forall \bar{\omega}.\, (\tau, \kappa'')\}, \kappa''', e) \\
& & \text{in } \langle \theta'\theta, \tau', \sigma, \kappa'''' \rangle \\
\mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, \mathtt{ilet}\ x = e\ \mathtt{in}\ e') & = & \text{let } \langle \theta, \tau, \sigma, \kappa' \rangle = \mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, e) \\
& & \quad \langle \theta', \tau', \sigma', \kappa'' \rangle = \mathcal{I}_{\mathrm{eff}}(\theta\mathcal{E}_x \cup \{x \mapsto \tau\}, \kappa', e) \\
& & \text{in } \langle \theta'\theta, \tau', \sigma \cup \sigma', \kappa'' \rangle
\end{array}
$$

Initial type environment

$$
\begin{array}{rcl}
\mathcal{E}(\mathtt{new}) & = & \forall \alpha, \gamma, \varsigma.\, (\alpha \xrightarrow{\varsigma} \mathtt{ref}_\gamma\, \alpha, \{\varsigma \sqsupseteq \mathtt{init}(\gamma)\}) \\
\mathcal{E}(\mathtt{get}) & = & \forall \alpha, \gamma, \varsigma.\, (\mathtt{ref}_\gamma\, \alpha \xrightarrow{\varsigma} \alpha, \{\varsigma \sqsupseteq \mathtt{read}(\gamma)\}) \\
\mathcal{E}(\mathtt{set}) & = & \forall \alpha, \gamma, \varsigma, \varsigma'.\, (\mathtt{ref}_\gamma\, \alpha \xrightarrow{\varsigma} \alpha \xrightarrow{\varsigma'} \alpha, \{\varsigma' \sqsupseteq \mathtt{write}(\gamma)\})
\end{array}
$$

Fig. 9. Type reconstruction for *Effect*

$$
\begin{array}{rcl}
\mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,x) & = & \text{let new } \bar{\omega}' \\
& & \quad \forall\bar{\omega}.\,(\tau,\kappa') = \mathcal{E}(x) \\
& & \quad \theta = \{\bar{\omega} \mapsto \bar{\omega}'\} \\
& & \text{in } \langle \mathrm{id}, \theta\tau, \kappa \cup \theta\kappa' \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,\lambda x.\,e) & = & \text{let new } \alpha \\
& & \quad \langle \theta,\tau,\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}_x \cup \{x \mapsto \alpha\},\kappa,e) \\
& & \text{in } \langle \theta, \theta\alpha \to \tau, \kappa' \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,\mathtt{rec}\,x.\,e) & = & \text{let new } \alpha \\
& & \quad \langle \theta,\tau,\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E}_x \cup \{x \mapsto \alpha\},\kappa,e) \\
& & \quad \theta' = \mathcal{U}_{\mathrm{mon}}(\theta\alpha,\tau) \\
& & \text{in } \langle \theta'\theta, \theta'\tau, \theta'\kappa' \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,e\,e') & = & \text{let new } \alpha \\
& & \quad \langle \theta,\tau,\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,e) \\
& & \quad \langle \theta',\tau',\kappa'' \rangle = \mathcal{I}_{\mathrm{mon}}(\theta\mathcal{E},\kappa',e') \\
& & \quad \theta'' = \mathcal{U}_{\mathrm{mon}}(\theta'\tau,\tau' \to \alpha) \\
& & \text{in } \langle \theta''\theta'\theta, \theta''\alpha, \theta''\kappa'' \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,\mathtt{let}\ x = e\ \mathtt{in}\ e') & = & \text{let } \langle \theta,\tau,\kappa \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,e) \\
& & \quad \bar{\omega} = (\mathrm{free}(\mathcal{K}(\kappa')\tau)) \setminus \mathrm{free}(\mathcal{K}(\kappa')\theta\mathcal{E}) \\
& & \quad \kappa'' = \{\varsigma \sqsupseteq \sigma \in \kappa' \mid \varsigma \in \bar{\omega}\} \\
& & \quad \kappa''' = \kappa' \setminus \kappa'' \\
& & \quad \langle \theta',\tau',\kappa'''' \rangle = \mathcal{I}_{\mathrm{mon}}(\theta\mathcal{E}_x \cup \{x \mapsto \forall\bar{\omega}.\,(\tau,\kappa)\},\kappa''',e') \\
& & \text{in } \langle \theta'\theta, \tau', \kappa'''' \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,\texttt{<}e\texttt{>}) & = & \text{let new } \varsigma \\
& & \quad \langle \theta,\tau,\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,e) \\
& & \text{in } \langle \theta, \mathtt{T}^{\varsigma}\,\tau, \kappa' \rangle \\
\mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,\mathtt{let}\ x \Leftarrow e\ \mathtt{in}\ e') & = & \text{let new } \alpha,\alpha',\varsigma,\varsigma',\varsigma'' \\
& & \quad \langle \theta,\tau,\kappa' \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E},\kappa,e) \\
& & \quad \theta' = \mathcal{U}_{\mathrm{mon}}(\tau,\mathtt{T}^{\varsigma}\,\alpha) \\
& & \quad \langle \theta'',\tau',\kappa'' \rangle = \mathcal{I}_{\mathrm{mon}}(\mathcal{E} \cup \{x \mapsto \theta'\alpha\},\theta'\kappa',e') \\
& & \quad \theta''' = \mathcal{U}_{\mathrm{mon}}(\tau',\mathtt{T}^{\varsigma'}\,\alpha') \\
& & \text{in } \langle \theta'''\theta''\theta'\theta, \mathtt{T}^{\varsigma''}\,\theta'''\alpha', \theta'''(\kappa'' \cup \{\varsigma'' \sqsupseteq \theta''\theta'\varsigma \cup \varsigma'\}) \rangle
\end{array}
$$

Initial type environment

$$
\begin{array}{rcl}
\mathcal{E}(\mathtt{new}) & = & \forall\alpha,\gamma,\varsigma.\,(\alpha \to \mathtt{T}^{\varsigma}\,\mathtt{ref}_{\gamma}\,\alpha, \{\varsigma \sqsupseteq \mathtt{init}(\gamma)\}) \\
\mathcal{E}(\mathtt{get}) & = & \forall\alpha,\gamma,\varsigma.\,(\mathtt{ref}_{\gamma}\,\alpha \to \mathtt{T}^{\varsigma}\,\alpha, \{\varsigma \sqsupseteq \mathtt{read}(\gamma)\}) \\
\mathcal{E}(\mathtt{set}) & = & \forall\alpha,\gamma,\varsigma'.\,(\mathtt{ref}_{\gamma}\,\alpha \to \alpha \to \mathtt{T}^{\varsigma'}\,\alpha, \{\varsigma' \sqsupseteq \mathtt{write}(\gamma)\})
\end{array}
$$

Fig. 10. Type reconstruction for *Monad*

A key trick in the reconstruction algorithm is to ensure that all effects and regions are represented by variables, to simplify unification. A type, type scheme, type environment, or substitution is *normalised* if the only regions and effects it contains are variables. (This notion is also present in the work of Talpin and Jouvelot, but only implicitly.)

The unification algorithms $\mathcal{U}_{\text{eff}}(\tau, \tau')$ and $\mathcal{U}_{\text{mon}}(\tau, \tau')$ take two normalised types and return a normalised substitution θ.

PROPOSITION 4.1. *(Unification) Let $\mathcal{U}$ be one of $\mathcal{U}_{\text{eff}}$ or $\mathcal{U}_{\text{mon}}$.*

—*(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.

4.2 Constraints

Constraints and the constraint solution algorithm are shown in Figure 8. A set of constraints κ is a set of inequations of the form $\varsigma \sqsupseteq \sigma$, asserting that ς encompasses at least the effect σ.

A substitution μ is a solution of κ, written $\mu \models \kappa$, if $\mu\varsigma \sqsupseteq \mu\sigma$ for each inequation $\varsigma \sqsupseteq \sigma$ in κ. Such a solution always exists.

The constraint solution algorithm $\mathcal{K}(\kappa)$ takes a constraint set and returns a substitution μ which solves κ. It assumes that effect variables on the left hand side of constraints in κ are distinct, which can be achieved by repeatedly merging two constraints $\varsigma \sqsupseteq \sigma$ and $\varsigma \sqsupseteq \sigma'$ into one constraint $\varsigma \sqsupseteq \sigma \cup \sigma'$.

PROPOSITION 4.2. *(Constraint solution)*

—*(Sound) $\mathcal{K}(\kappa) \models \kappa$.*

—*(Complete) If $\mu \models \kappa$ then $\mu = \mu \circ \mathcal{K}(\kappa)$.*

The proof is given in Appendix A.3.

The algorithm is identical to the one in Talpin's thesis [TJ94; Tal93]. It computes a principal solution of the constraint set κ, independently of the order in which the constraints are visited. (The algorithm is subtly different to their earlier algorithm *Min* [TJ92]. Algorithm *Min* yields a solution, which is minimal with respect to the ordering $\sqsupseteq$ defined by $\mu \sqsupseteq \mu'$ iff, for all $\varsigma \in \text{dom}(\mu')$, $\mu(s) \sqsupseteq \mu'(s)$. Unfortunately, this ordering is not defined in their paper.)

4.3 Reconstruction for *Effect*

Type schemes and the reconstruction algorithm for *Effect* are shown in Figure 9. A different flavor of type schemes is introduced which fits better with the type inference. Following [TJ94; Tal93], such a reconstruction scheme has the form $\forall\bar{\omega}.\,(\tau, \kappa)$ where $\bar{\omega}$ is a sequence of type, region, or effect variables; 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 $\bar{\omega}$. Reconstruction environments are taken to map identifiers to type schemes; the environment is normalised if all types in it are normalised. We write $\{\bar{\omega} \mapsto \bar{\omega}'\}$ for a substitution, when $\bar{\omega}$ and $\bar{\omega}'$

have the same length, and each has type, region, and effect variables in the same positions as the other.

The reconstruction algorithm $\mathcal{I}_{\text{eff}}(\mathcal{E}, \kappa, e)$ takes a normalised reconstruction environment $\mathcal{E}$, an initial constraint set κ, 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. The algorithm is essentially drawn from Talpin and Jouvelot's later work [TJ94].

To relate a reconstruction scheme to an equivalent type scheme, we exploit the algorithm for solving constraints in Figure 8. If $\hat{\tau}$ is a type reconstruction scheme, we define $\overline{\hat{\tau}} = \hat{\tau}'$ where $\hat{\tau} = \forall \bar{\alpha}, \bar{\gamma}, \bar{\varsigma}.\, (\tau, \kappa)$ and $\hat{\tau}' = \forall \bar{\alpha}, \bar{\gamma}, \bar{\varsigma}.\, \mathcal{K}(\kappa)(\tau)$. We define $\overline{\mathcal{E}}$ by pointwise extension: $\overline{\mathcal{E}}(x) = \overline{\mathcal{E}(x)}$ for each $x \in \text{dom}(\mathcal{E})$.

To state completeness, we use the generic instance relation $\hat{\tau} \succeq \hat{\tau}'$ defined in Section 2. We define $\mathcal{E} \succeq \mathcal{E}'$ by pointwise extension: it holds if $\mathcal{E}(x) \succeq \mathcal{E}'(x)$ for each $x \in \text{dom}(\mathcal{E})$.

The reconstruction algorithm is sound and complete.

PROPOSITION 4.3. *(Type reconstruction)*

—*(Sound) If* $\mathcal{I}_{\text{eff}}(\mathcal{E}, \kappa, e) = \langle \theta, \tau, \sigma, \kappa' \rangle$ *and* $\mu = \mathcal{K}(\kappa')$ *then* $\mu\theta\overline{\mathcal{E}} \vdash_{\text{eff}} e : \mu\tau\,!\,\mu\sigma$*, with* $\mathcal{E}$*,* θ*, and* τ *normalised.*

—*(Complete) Let* $\mathcal{E}$ *be a normalised reconstruction environment, let* $\mathcal{E}'$ *be a type environment such that* $\overline{\mathcal{E}} \succeq \mathcal{E}'$*, and let* $\theta \models \kappa$*. If* $\theta\mathcal{E}' \vdash_{\text{eff}} e : \tau\,!\,\sigma$ *then* $\mathcal{I}_{\text{eff}}(\mathcal{E}, \kappa, e) = \langle \theta', \tau', \sigma', \kappa' \rangle$ *and there exists a substitution* θ'' *such that* $\theta''\theta'\overline{\mathcal{E}} \succeq \theta\mathcal{E}'$ *and* $\tau = \theta''\tau'$ *and* $\sigma \supseteq \theta''\sigma'$ *and* $\theta'' \models \kappa'$*, with* θ' *and* τ' *normalised.*

The proof for the first part is by induction on the structure of expressions, and for the second by induction on the structure of type derivations. (A similar proof is given by Talpin and Jouvelot [TJ94]. In their earlier work [TJ92], the proof skips the case of polymorphic 'let' binding, assuming such bindings have been expanded out.)

Another way to obtain a reconstruction result would be to introduce *arrow effects* as proposed by Tofte and others [TT94]. In fact, type schemes with arrow effects are equivalent to the reconstruction type schemes introduced in the present section: simply replace each arrow effect $\varsigma.\sigma$ by the effect variable ς and add the constraint $\varsigma \sqsupseteq \sigma$.

4.4 Reconstruction for *Monad*

The reconstruction algorithm for *Monad* is shown in Figure 9. The unification algorithm, type schemes, and type environments are as before, with types for *Monad* replacing types for *Effect*, *mutatis mutandis*. Constraints carry over without change.

The reconstruction algorithm $\mathcal{I}_{\text{mon}}(\mathcal{E}, \kappa, e)$ takes a type environment $\mathcal{E}$, an initial constraint set κ, 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 as 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 4.4. *(Type reconstruction)*

—*(Sound) If* $\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \kappa, e) = \langle \theta, \tau, \kappa' \rangle$ *and* $\mu = \mathcal{K}(\kappa')$ *then* $\mu\theta\overline{\mathcal{E}} \vdash_{\mathrm{mon}} e : \mu\tau$*, with* $\mathcal{E}$*,* θ*, and* τ *normalised.*

—*(Complete) Let* $\mathcal{E}$ *be a normalised reconstruction environment,* $\mathcal{E}'$ *a type environment such that* $\overline{\mathcal{E}} \succeq \mathcal{E}'$*, and let* $\theta \models \kappa$*. If* $\theta\mathcal{E}' \vdash_{\mathrm{mon}} e : \tau$ *then* $\mathcal{I}_{\mathrm{mon}}(\mathcal{E}, \kappa, e) = \langle \theta', \tau', \kappa' \rangle$ *and there exists a substitution* θ'' *such that* $\theta''\theta'\overline{\mathcal{E}} \succeq \theta\mathcal{E}$ *and* $\tau = \theta''\tau'$ *and* $\theta'' \models \kappa'$*, with* 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 $\mu \models \kappa'$. The translation is extended to apply to reconstruction schemes by taking $(\forall\bar{\omega}.\,(\tau, \kappa))^\dagger = \forall\bar{\omega}.\,(\tau^\dagger, \kappa)$.

PROPOSITION 4.5. *(Translation preserves type reconstruction) If* $\mathcal{I}_{\mathrm{eff}}(\mathcal{E}, \kappa, e) = \langle \theta', \tau', \sigma', \kappa' \rangle$ *and* $\mathcal{I}_{\mathrm{mon}}(\mathcal{E}^\dagger, \kappa, e^*) = \langle \theta'', \tau'', \kappa'' \rangle$ *then there exist* ς *and* μ *such that* $\mathsf{T}^\varsigma\,\tau^\dagger = \tau'$ *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 might 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 $\mathtt{read}(\rho)$ or $\mathtt{init}(\rho)$ is in σ, and S^σ is a store restricted to contain only locations in regions ρ such that $\mathtt{init}(\rho)$ or $\mathtt{write}(\rho)$ 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 Breazu-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. Thanks to Mads Tofte, Jon Riecke, Matthias Felleisen, and J.-P. Talpin for comments on earlier drafts of this paper.

REFERENCES

Gilles Barthe, John Hatcliff, and Peter Thiemann. Monadic type systems: Pure type systems for impure settings (preliminary report). In *Proceedings of HOOTS'97*, volume 10 of *ENTCS*. Elsevier, 1998.

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

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.

Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simpon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. *4'th ACM International Conference on Functional Programming*, Paris, September 1999.

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

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.

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

Haskell98, a non-strict, purely functional language. `http://www.haskell.org`, December 1998.

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

Robert Harper and Mark Lillibridge. Polymorphic type assignment and cps conversion. *LISP and Symbolic Computation*, 6(4):361–380, 1993.

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.

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.

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

Xavier Leroy. Polymorphism by name for references and continuations. In *Conference Record of POPL '93: The* 20$^{\text{st}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231, Charleston, South Carolina, USA, January 1993. ACM Press.

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of POPL '88: The* 15$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, 1988. ACM Press.

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.

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

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

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

E. Moggi and F. Palumbo, Monadic Encapsulation of Effects: A Revised Approach, *HOOTS '99 Higher Order Operational Techniques in Semantics*, Electronic Notes in Theoretical Computer Science 26, Elsevier Science, 1999.

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

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

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

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

Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In Mads Dam, editor, *Proceedings of the Fifth LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

Simon Peyton Jones, Alastair Reid, Tony Hoare, Simon Marlow, and Fergus Henderson. A semantics for imprecise exceptions. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices, Atlanta, Georgia, USA, May 1999. SIGPLAN Notices 34(5).

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

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

Simon Peyton Jones, Andrew Gordon, and Sigbjrn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, USA, 21–24 January 1996. ACM Press.

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

Miley Semmelroth and Amr Sabry. Monadic Encapsulation in ML. *4'th ACM International Conference on Functional Programming*, pages 8–17, Paris, September 1999.

Amr Sabry and Philip Wadler, A reflection on call-by-value, *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.)

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

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

J.-P. Talpin, Theoretical and Practical Aspects of Type and Effect Inference, PhD Thesis, Ecole des Mines de Paris and University Paris VI, 1993.

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

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

Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

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

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

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

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.

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

P. Wadler, The marriage of effects and monads, *ACM 3'rd International Conference on Functional Programming*, Baltimore, September 1998.

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

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

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

A. PROOFS

A.1 Progress

Proposition 3.3 *(Progress)*
Suppose $\mathcal{E} \vdash_{\text{eff}} s, e : \tau \,!\, \sigma$. *Either* e *is a value or there exists* s', e' *and* f *such that* $s, e \xrightarrow{f}_{\text{eff}} s', e'$.

PROOF. By induction on the derivation of $\mathcal{E} \vdash_{\text{eff}} s, e : \tau \,!\, \sigma$. In each case, we have that $\mathcal{E} \vdash_{\text{eff}} s$ and $\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma$.

Case $(var)\dfrac{\hat{\tau} \succeq \tau}{\mathcal{E}_x \cup \{x \mapsto \hat{\tau}\} \vdash_{\text{eff}} x : \tau \,!\, \emptyset}$. A value.

Case $(does)\dfrac{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma \qquad \sigma' \sqsupseteq \sigma}{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma'}$. The claim is immediate by the inductive hypothesis.

Case $(abs)\dfrac{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\text{eff}} e : \tau' \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \lambda x.\, e : \tau \xrightarrow{\sigma} \tau' \,!\, \emptyset}$. A value.

Case $(app)\dfrac{\mathcal{E} \vdash_{\text{eff}} e_1 : \tau \xrightarrow{\sigma''} \tau' \,!\, \sigma \qquad \mathcal{E} \vdash_{\text{eff}} e_2 : \tau \,!\, \sigma'}{\mathcal{E} \vdash_{\text{eff}} e_1\, e_2 : \tau' \,!\, \sigma \cup \sigma' \cup \sigma''}$. By induction, either $s, e_1 \xrightarrow{f}_{\text{eff}} s', e_1'$ or e_1 is a value v_1.

In the first case, rule $(context)$ yields that $s, e_1\, e_2 \xrightarrow{f}_{\text{eff}} s', e_1'\, e_2$.

If $e_1 = v_1$, a value, then, by induction, either $s, e_2 \xrightarrow{f}_{\text{eff}} s', e_2'$ or e_2 is a value.

In the first case, rule $(context)$ yields that $s, v_1\, e_2 \xrightarrow{f}_{\text{eff}} s', v_1\, e_2'$.

If $e_2 = v_2$, a value, then the canonical forms lemma 3.2 applied to v_1 yields that v_1 is either $\lambda x.\, e$ or $\mathtt{rec}\, x.\, \lambda x'.\, e$. In both cases, the expression is a redex using either $(beta)$ or (rec).

$$\mathcal{E} \vdash_{\text{eff}} v : \tau \,!\, \emptyset$$

Case $(let)\dfrac{\mathcal{E}_x \cup \{x \mapsto \text{gen}(\mathcal{E}, \tau)\} \vdash_{\text{eff}} e : \tau' \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{let}\ x = v\ \mathtt{in}\ e : \tau' \,!\, \sigma}$. This expression is a redex.

$$\mathcal{E} \vdash_{\text{eff}} e_1 : \tau \,!\, \sigma$$

Case $(ilet)\dfrac{\mathcal{E}_x \cup \{x \mapsto \tau\} \vdash_{\text{eff}} e_2 : \tau' \,!\, \sigma'}{\mathcal{E} \vdash_{\text{eff}} \mathtt{ilet}\ x = e_1\ \mathtt{in}\ e_2 : \tau' \,!\, \sigma \cup \sigma'}$. By induction, either e_1 is a value, in which case the whole expression is a redex, or $s, e_1 \xrightarrow{f}_{\text{eff}} s', e_1'$. In the latter case, the whole expression reduces due to $(context)$.

Case $(rec)\dfrac{\mathcal{E}_{x,x'} \cup \{x \mapsto \tau \xrightarrow{\sigma} \tau', x' \mapsto \tau\} \vdash_{\text{eff}} e : \tau' \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{rec}\, x.\, \lambda x'.\, e : \tau \xrightarrow{\sigma} \tau' \,!\, \emptyset}$. A value.

Case $(new)\dfrac{\mathcal{E} \vdash_{\text{eff}} e : \tau \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{new}\, e : \mathtt{ref}_\rho\, \tau \,!\, \sigma \cup \mathtt{init}(\rho)}$. By induction, either e is a value, in which case the whole expression is a redex, or $s, e \xrightarrow{f}_{\text{eff}} s', e'$. In the latter case, the whole expression reduces, too, by rule $(context)$.

Case $(get)\dfrac{\mathcal{E} \vdash_{\text{eff}} e : \mathtt{ref}_\rho\, \tau \,!\, \sigma}{\mathcal{E} \vdash_{\text{eff}} \mathtt{get}\, e : \tau \,!\, \sigma \cup \mathtt{read}(\rho)}$. By induction, either e is a value, in which case the canonical forms lemma 3.2 shows that $e = l \in \mathit{Location}$, $l \in \text{dom}(s)$, and hence the whole expression is a redex, or $s, e \xrightarrow{f}_{\text{eff}} s', e'$. In the latter case, the whole expression reduces by rules $(context)$.

Case $(set)\frac{\mathcal{E} \vdash_{\text{eff}} e_1 : \mathtt{ref}_\rho\, \tau \,!\, \sigma \qquad \mathcal{E} \vdash_{\text{eff}} e_2 : \tau \,!\, \sigma'}{\mathcal{E} \vdash_{\text{eff}} \mathtt{set}\, e_1\, e_2 : \tau \,!\, \sigma \cup \sigma' \cup \mathtt{write}(\rho)}$. By induction, either e_1 is a value, in which case the canonical forms lemma 3.2 shows that $e_1 = l \in \mathit{Location}$ and $l \in \mathrm{dom}(s)$, or $s, e_1 \xrightarrow{f}_{\text{eff}} s', e_1'$. In the latter case, the whole expression reduces by rules (*context*).

If $e_1 = l$ then, by induction, either e_2 is a value, in which case the whole expression is a redex, or $s, e_2 \xrightarrow{s'}_{\text{eff}}, e_2'$. In the latter case, the whole expression reduces by (*context*). □

A.2 Translation preserves semantics

Proposition 3.10 *(Translation preserves semantics)*

(1) *If* $s, e \xrightarrow{f}_{\text{eff}} s', e'$ *then either* $s^\dagger, e^* \xrightarrow{f}_{\text{mon}} s'^\dagger, e'^*$, *or* $s^\dagger, e^* \xrightarrow{f}_{\text{mon}} s'^\dagger, e_0$ *and* $e_0 \longrightarrow_{\text{mon}} e'^*$.

(2) *If* $s, e \xrightarrow{f}\!\!\!\!\rightarrow_{\text{eff}} s', e'$ *then* $s^\dagger, e^* \xrightarrow{f}\!\!\!\!\rightarrow_{\text{mon}} s'^\dagger, e'^*$.

(3) *If* $s, e \Uparrow^f_{\text{eff}}$ *then* $s^\dagger, e^* \Uparrow^f_{\text{mon}}$.

(4) *If* s, e *is stuck then either* $s^\dagger, e^*$ *is stuck or* $e^* \longrightarrow_{\text{mon}} e_0$ *where* $s^\dagger, e_0$ *is stuck.*

Proof. **Item 1** by induction on the definition of $\xrightarrow{f}_{\text{eff}}$.

Case $(beta)$: $s, (\lambda x.\, e)v \xrightarrow{\emptyset}_{\text{eff}} s, e[x := v]$.

$$\begin{array}{ll} & s^\dagger, ((\lambda x.\, e)v)^* \\ = & s^\dagger, (\lambda x.\, e)^\dagger\, v^\dagger \\ = & s^\dagger, (\lambda x.\, e^*)\, v^\dagger \\ \xrightarrow{\emptyset}_{\text{mon}} & s^\dagger, e^*[x := v^\dagger] \\ = & s^\dagger, (e[x := v])^* \end{array}$$

Case (rec): $s, (\mathtt{rec}\, x.\, \lambda x'.\, e)v \xrightarrow{\emptyset}_{\text{eff}} s, (\lambda x'.\, e[x := \mathtt{rec}\, x.\, \lambda x'.\, e])v$.

$$\begin{array}{ll} & s^\dagger, (\mathtt{rec}\, x.\, \lambda x'.\, e)^\dagger\, v^\dagger \\ = & s^\dagger, (\mathtt{rec}\, x.\, \lambda x'.\, e^*)\, v^\dagger \\ \xrightarrow{\emptyset}_{\text{mon}} & s^\dagger, (\lambda x'.\, e^*[x := \mathtt{rec}\, x.\, \lambda x'.\, e^*])\, v^\dagger \\ = & s^\dagger, (\lambda x'.\, e^*[x := (\mathtt{rec}\, x.\, \lambda x'.\, e)^\dagger])\, v^\dagger \\ = & s^\dagger, ((\lambda x'.\, e[x := \mathtt{rec}\, x.\, \lambda x'.\, e])v)^* \end{array}$$

Case $(letv)$: $s, \mathtt{let}\ x = v\ \mathtt{in}\ e \xrightarrow{\emptyset}_{\text{eff}} s, e[x := v]$.

$$\begin{array}{ll} & s^\dagger, (\mathtt{let}\ x = v\ \mathtt{in}\ e)^* \\ = & s^\dagger, \mathtt{let}\ x = v^\dagger\ \mathtt{in}\ e^* \\ \xrightarrow{\emptyset}_{\text{mon}} & s^\dagger, e^*[x := v^\dagger] \\ = & s^\dagger, (e[x := v])^* \end{array}$$

Case (*let*): $s, \texttt{ilet}\ x = v\ \texttt{in}\ e \xrightarrow{\emptyset}_{\text{eff}} s, e[x := v]$.

$$\begin{array}{ll} & s^\dagger, (\texttt{ilet}\ x = v\ \texttt{in}\ e)^* \\ = & s^\dagger, \texttt{let}\ x \Leftarrow v^*\ \texttt{in}\ e^* \\ = & s^\dagger, \texttt{let}\ x \Leftarrow \texttt{<}v^\dagger\texttt{>}\ \texttt{in}\ e^* \\ \xrightarrow{\emptyset}_{\text{mon}} & s^\dagger, e^*[x := v^\dagger] \\ = & s^\dagger, (e[x := v])^* \end{array}$$

Case (*new*): $s, \texttt{new}\, v \xrightarrow{\texttt{init}(l)}_{\text{eff}} s \cup \{l \mapsto v\}, l$ where $l \notin \text{dom}(s)$.

$$\begin{array}{ll} & s^\dagger, (\texttt{new}\, v)^* \\ = & s^\dagger, \texttt{new}\, v^\dagger \\ \xrightarrow{\texttt{init}(l)}_{\text{mon}} & s^\dagger \cup \{l \mapsto v^\dagger\}, \texttt{<}l\texttt{>} \\ = & s \cup \{l \mapsto v\}^\dagger, l^* \end{array}$$

Case (*get*): $s_l \cup \{l \mapsto v\}, \texttt{get}\, l \xrightarrow{\texttt{read}(l)}_{\text{eff}} s_l \cup \{l \mapsto v\}, v$.

$$\begin{array}{ll} & s_l \cup \{l \mapsto v\}^\dagger, (\texttt{get}\, l)^* \\ = & s_l^\dagger \cup \{l \mapsto v^\dagger\}, \texttt{get}\, l \\ \xrightarrow{\texttt{read}(l)}_{\text{mon}} & s_l^\dagger \cup \{l \mapsto v^\dagger\}, \texttt{<}v^\dagger\texttt{>} \\ = & s_l \cup \{l \mapsto v\}^\dagger, v^* \end{array}$$

Case (*set*): $s_l \cup \{l \mapsto v\}, \texttt{set}\, l\, v' \xrightarrow{\texttt{write}(l)}_{\text{eff}} s_l \cup \{l \mapsto v'\}, v'$.

$$\begin{array}{ll} & (s_l \cup \{l \mapsto v\})^\dagger, (\texttt{set}\, l\, v')^* \\ = & s_l^\dagger \cup \{l \mapsto v^\dagger\}, \texttt{set}\, l\, v'^\dagger \\ \xrightarrow{\texttt{write}(l)}_{\text{mon}} & s_l^\dagger \cup \{l \mapsto v'^\dagger\}, \texttt{<}v'^\dagger\texttt{>} \\ = & (s_l \cup \{l \mapsto v'\})^\dagger, v'^* \end{array}$$

Case (*context*) $\dfrac{s, n \xrightarrow{f}_{\text{eff}} s', e'}{s, E[n] \xrightarrow{f}_{\text{eff}} s', E[e']}$. The expression, n, must be a non-value because otherwise the reduction would be impossible.

Subcase $E = [\,]\, e$.

$$\begin{array}{ll} & s^\dagger, (n\, e)^* \\ = & s^\dagger, \texttt{let}\ x \Leftarrow n^*\ \texttt{in}\ (x\, e)^* \\ \xrightarrow{f}_{\text{mon}} & \text{by induction, and context } M \\ & s^\dagger, \texttt{let}\ x \Leftarrow e'^*\ \texttt{in}\ (x\, e)^* \end{array}$$

If $e' \in \mathit{NonVal}$ then

$$= \quad s^\dagger, (e'\, e)^*$$

If $e' = v' \in \mathit{Val}$ then

$$\begin{array}{ll} = & s^\dagger, \texttt{let}\ x \Leftarrow v'^*\ \texttt{in}\ (x\, e)^* \\ = & s^\dagger, \texttt{let}\ x \Leftarrow \texttt{<}v'^\dagger\texttt{>}\ \texttt{in}\ (x\, e)^* \\ \xrightarrow{\emptyset}_{\text{mon}} & s^\dagger, (x\, e)^*[x := v'^\dagger] \\ = & s^\dagger, (v'\, e)^* \end{array}$$

Subcase $E = v\,[\,]$.

$$\begin{array}{ll} & s^\dagger, (v\,n)^* \\ = & s^\dagger, \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (v\,x)^* \\ \xrightarrow{f}_{\mathrm{mon}} & \text{by induction, and context } M \\ & s^\dagger, \mathtt{let}\ x \Leftarrow e'^*\ \mathtt{in}\ (v\,x)^* \end{array}$$

If $e' \in \mathit{NonVal}$ then

$$= \quad s^\dagger, (v\,e')^*$$

If $e' = v' \in \mathit{Val}$ then

$$\begin{array}{ll} = & s^\dagger, \mathtt{let}\ x \Leftarrow v'^*\ \mathtt{in}\ (v\,x)^* \\ = & s^\dagger, \mathtt{let}\ x \Leftarrow \texttt{<}v'^\dagger\texttt{>}\ \mathtt{in}\ (v\,x)^* \\ \xrightarrow{\emptyset}_{\mathrm{mon}} & s^\dagger, (v\,x)^*[x := v'^\dagger] \\ = & s^\dagger, (v\,v')^* \end{array}$$

Subcase $E = \mathtt{ilet}\ x = [\,]\ \mathtt{in}\ e$.

$$\begin{array}{ll} & s^\dagger, (\mathtt{ilet}\ x = n\ \mathtt{in}\ e)^* \\ = & s^\dagger, \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ e^* \\ \xrightarrow{f}_{\mathrm{mon}} & \text{by induction, and context } M \\ & s^\dagger, \mathtt{let}\ x \Leftarrow e'^*\ \mathtt{in}\ e^* \\ = & s^\dagger, (\mathtt{ilet}\ x = e'\ \mathtt{in}\ e)^* \end{array}$$

Subcase $E = \mathtt{new}\,[\,]$.

$$\begin{array}{ll} & s^\dagger, (\mathtt{new}\,n)^* \\ = & s^\dagger, \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (\mathtt{new}\,x)^* \\ \xrightarrow{f}_{\mathrm{mon}} & \text{by induction, and context } M \\ & s^\dagger, \mathtt{let}\ x \Leftarrow e'^*\ \mathtt{in}\ (\mathtt{new}\,x)^* \end{array}$$

If $e' \in \mathit{NonVal}$ then

$$= \quad s^\dagger, (\mathtt{new}\,e')^*$$

If $e' = v' \in \mathit{Val}$ then

$$\begin{array}{ll} = & s^\dagger, \mathtt{let}\ x \Leftarrow v'^*\ \mathtt{in}\ (\mathtt{new}\,x)^* \\ = & s^\dagger, \mathtt{let}\ x \Leftarrow \texttt{<}v'^\dagger\texttt{>}\ \mathtt{in}\ (\mathtt{new}\,x)^* \\ \xrightarrow{\emptyset}_{\mathrm{mon}} & s^\dagger, (\mathtt{new}\,x)^*[x := v'^\dagger] \\ = & s^\dagger, (\mathtt{new}\,v')^* \end{array}$$

Subcase $E = \mathtt{get}\,[\,]$.

$$\begin{array}{ll} & s^\dagger, (\mathtt{get}\,n)^* \\ = & s^\dagger, \mathtt{let}\ x \Leftarrow n^*\ \mathtt{in}\ (\mathtt{get}\,x)^* \\ \xrightarrow{f}_{\mathrm{mon}} & \text{by induction, and context } M \\ & s^\dagger, \mathtt{let}\ x \Leftarrow e'^*\ \mathtt{in}\ (\mathtt{get}\,x)^* \end{array}$$

If $e' \in \mathit{NonVal}$ then

$$= \quad s^\dagger, (\mathtt{get}\, e')^*$$

If $e' = v' \in \mathit{Val}$ then

$$\begin{array}{ll} = & s^\dagger, \mathtt{let}\ x \Leftarrow v'^*\ \mathtt{in}\ (\mathtt{get}\, x)^* \\ = & s^\dagger, \mathtt{let}\ x \Leftarrow \mathtt{<}v'^\dagger\mathtt{>}\ \mathtt{in}\ (\mathtt{get}\, x)^* \\ \xrightarrow{\emptyset}_{\mathrm{mon}} & s^\dagger, (\mathtt{get}\, x)^*[x := v'^\dagger] \\ = & s^\dagger, (\mathtt{get}\, v')^* \end{array}$$

Subcase $E = \mathtt{set}\,[\,]\,e$. Analogous to $E = [\,]\,e$.
Subcase $E = \mathtt{set}\,v\,[\,]$. Analogous to $E = v\,[\,]$.
Item 2 by induction on the definition of $\xrightarrow{f}_{\mathrm{eff}}$.
Item 3 is immediate from item 2.
Item 4 requires an inductive proof, again:
Case $l\,v'$, where $l \in \mathit{Location}$.
$(l\,v')^* = l^\dagger\, v'^\dagger = l\, v'^\dagger$ is stuck because l is not a lambda.
Case $\mathtt{get}\,v$, where v is not a location or $v \notin \mathrm{dom}(s)$.
$(\mathtt{get}\,v)^* = \mathtt{get}\,v^\dagger$. There are three cases for v.
Subcase $v = l \in \mathit{Location}$ and $l \notin \mathrm{dom}(s)$.
Then $\mathtt{get}\,l^\dagger = \mathtt{get}\,l$ is stuck because $l \notin \mathrm{dom}(s^\dagger) = \mathrm{dom}(s)$.
Subcase $v = \lambda x.\, e$.
Then $\mathtt{get}\,(\lambda x.\, e)^\dagger = \mathtt{get}\,\lambda x.\, e^*$ is stuck because $v^\dagger$ is not a location.
Subcase $v = \mathtt{rec}\, x.\, \lambda x'.\, e$.
Then

$$\begin{array}{ll} & \mathtt{get}\,(\mathtt{rec}\, x.\, \lambda x'.\, e)^\dagger \\ = & \mathtt{get}\,(\mathtt{rec}\, x.\, \lambda x'.\, e^*) \\ \longrightarrow_{\mathrm{mon}} & \mathtt{get}\,(\lambda x'.\, e^*[x := \mathtt{rec}\, x.\, \lambda x'.\, e^*]) \end{array}$$

which is stuck because $\lambda x'.\ \ldots$ is not a location.
Case $\mathtt{set}\,v\,v'$, where v is not a location or $v \notin \mathrm{dom}(s)$.
$(\mathtt{set}\,v\,v')^* = \mathtt{set}\,v^\dagger\, v'^\dagger$. Analogous to subcase $\mathtt{get}\,v$.
Case if s, e is stuck, then $s, E[e]$ is stuck. By definition of stuck, e must be a non-value.
Subcase $E = [\,]\,e'$.
$(e\,e')^* = \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ (x\,e')^*$. By induction, $s^\dagger, e^*$ is stuck so that e^* has the form $M[e_0]$, where e_0 is one of the cases in the definition of stuck. Hence, for some monad context M',

$$\begin{array}{ll} & (e\,e')^* \\ = & \mathtt{let}\ x \Leftarrow M[e_0]\ \mathtt{in}\ (x\,e')^* \\ = & M'[e_0] \end{array}$$

This proves the claim.
Subcase $E = v\,[\,]$.
$(v\,e)^* = \mathtt{let}\ x \Leftarrow e^*\ \mathtt{in}\ (v\,x)^*$. Stuck by analogous reasoning as in the previous subcase.
Subcase $E = \mathtt{ilet}\ x = [\,]\ \mathtt{in}\ e'$.

$(\texttt{ilet}\ x = e\ \texttt{in}\ e')^* = \texttt{let}\ x \Leftarrow e^*\ \texttt{in}\ e'^*$. Stuck by analogous reasoning as in the previous subcase.

Subcase $E = \texttt{new}\,[\,]$.

$(\texttt{new}\,e)^* = \texttt{let}\ x \Leftarrow e^*\ \texttt{in}\ (\texttt{new}\,x)^*$. Stuck by analogous reasoning as in the previous subcase.

Subcase $E = \texttt{get}\,[\,]$.

$(\texttt{get}\,e)^* = \texttt{let}\ x \Leftarrow e^*\ \texttt{in}\ (\texttt{get}\,x)^*$. Analogous.

Subcase $E = \texttt{set}\,[\,]\,e'$.

$(\texttt{set}\,e\,e')^* = \texttt{let}\ x \Leftarrow e^*\ \texttt{in}\ (\texttt{set}\,x\,e')^*$. Analogous to subcase $e\,e'$.

Subcase $E = \texttt{set}\,v\,[\,]$.

$(\texttt{set}\,v\,e)^* = \texttt{let}\ x \Leftarrow e^*\ \texttt{in}\ (\texttt{set}\,v\,x)^*$. Analogous to subcase $v\,e$. □

A.3 Constraint solution

Proposition 4.2 *(Constraint solution)*

—*(Sound)* $\mathcal{K}(\kappa) \models \kappa$.

—*(Complete) If* $\mu \models \kappa$ *then* $\mu = \mu \circ \mathcal{K}(\kappa)$.

Proof. To show soundness, suppose that the constraint set has the form $\kappa \cup \{\varsigma \sqsupseteq \sigma\}$. Now,

$$\begin{array}{ll} & \mathcal{K}(\kappa \cup \{\varsigma \sqsupseteq \sigma\})(\varsigma) \\ = & (\{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\} \circ \mathcal{K}(\kappa))(\varsigma) \\ = & \{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\}(\varsigma) \\ = & \varsigma \cup \mathcal{K}(\kappa)(\sigma) \\ \sqsupseteq & (\{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\} \circ \mathcal{K}(\kappa))(\sigma) \\ = & \mathcal{K}(\kappa \cup \{\varsigma \sqsupseteq \sigma\})(\sigma) \end{array}$$

Completeness is shown by induction on κ.

Case $\emptyset$ is immediate.

Case $\kappa \cup \{\varsigma \sqsupseteq \sigma\}$.

Let $\mu' = \mu \circ \mathcal{K}(\kappa \cup \{\varsigma \sqsupseteq \sigma\}) = \mu \circ \{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\} \circ \mathcal{K}(\kappa)$. Show that, for each ς', $\mu'(\varsigma') = \mu(\varsigma')$.

First, observe that $\mu \circ \{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\} = \mu$ by considering this substitution on ς.

$$\begin{array}{ll} & \mu(\{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\}(\varsigma)) \\ = & \mu(\varsigma \cup \mathcal{K}(\kappa)(\sigma)) \\ = & \mu(\varsigma) \cup \mu(\mathcal{K}(\kappa)(\sigma)) \\ & \text{[by the inductive hypothesis]} \\ = & \mu(\varsigma) \cup \mu(\sigma) \\ & \text{[since } \mu \text{ solution]} \\ = & \mu(\varsigma) \end{array}$$

There are three cases to consider.

Subcase $\varsigma' \notin \mathrm{dom}(\mathcal{K}(\kappa))$ and $\varsigma' \neq \varsigma$.

$\mu'(\varsigma') = \mu(\{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\}(\mathcal{K}(\kappa)(\varsigma'))) = \mu(\varsigma')$.

Subcase $\varsigma' = \varsigma$.

$$\begin{array}{ll} & \mu'(\varsigma) \\ = & \mu(\{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\}(\mathcal{K}(\kappa)(\varsigma))) \\ & [\text{since } \varsigma \notin \mathrm{dom}(\mathcal{K}(\kappa))] \\ = & \mu(\{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\}(\varsigma)) \\ & [\text{by the preceding observation}] \\ = & \mu(\varsigma) \end{array}$$

Subcase $\varsigma' \in \mathrm{dom}(\mathcal{K}(\kappa))$.

$$\begin{array}{ll} & \mu'(\varsigma') \\ = & \mu(\{\varsigma \mapsto \varsigma \cup \mathcal{K}(\kappa)(\sigma)\}(\mathcal{K}(\kappa)(\varsigma'))) \\ & [\text{by the preceding observation}] \\ = & \mu(\mathcal{K}(\kappa)(\varsigma')) \\ = & \mu(\varsigma') \end{array}$$

□

Received November 1999; revised February 2001; accepted March 2001.