Representing Layered Monads

Andrzej Filinski

BRICS*
Department of Computer Science
University of Aarhus
Ny Munkegade, DK-8000 Aarhus C, Denmark

andrzej@brics.dk

Abstract

There has already been considerable research on constructing modular, monad-based specifications of computational effects (state, exceptions, nondeterminism, etc.) in programming languages. We present a simple framework in this tradition, based on a Church-style effect-typing system for an ML-like language. The semantics of this language is formally defined by a series of monadic translations, each one expanding away a layer of effects. Such a layered specification is easy to reason about, but its *direct* implementation (whether by parameterized interpretation or by actual translation) is often prohibitively inefficient.

By exploiting deeper semantic properties of monads, however, it is also possible to derive a vastly more efficient implementation: we show that each layer of effects can be uniformly simulated by continuation-passing, and further that multiple such layers can themselves be simulated by a standard semantics for call/cc and mutable state. Thus, even multi-effect programs can be executed in Scheme or SML/NJ at full native speed, generalizing an earlier single-effect result. As an example, we show how a simple resumption-based semantics of concurrency allows us to directly simulate a shared-state program across all possible dynamic interleavings of execution threads.

1 Introduction

By now, monads are firmly established as an key concept in functional programming, both as a semantic framework for ML-like languages [Mog89], and as a structuring technique for purely functional programs with computational effects [Wad92]. But the situation is less clear for the prospect of using monads to structure multiple, potentially intertwined effects: although a number of frameworks for this have been proposed [Mog90, Ste94, CF94, LHJ95, Esp95], none seem to have gained overwhelming acceptance.

This is perhaps not surprising, since a truly modular characterization of computational effects is probably an ill-specified problem, with no unique solution. Nevertheless, most monad-based formalisms tend to leave at least two areas with definite room for improvement:

- Conceptual overhead. It is usually clear how a single monad in isolation represents a particular notion of computational effect (such as mutable state or exceptions), and how individual effect-operations (such as reading and writing of the state, or raising and handling of exceptions) are expressed in terms of the monadic structure. But with multiple effects, the initial cost is higher: each notion of effect must now be specified in an integrable form, and the specification of the operations must likewise be further parameterized. In some cases, operations for one effect are even fundamentally incompatible with the monadic structure of another, significantly complicating the semantics of the resulting language.
- Practical overhead. A distinct problem with monad-based executable specifications of interacting effects is computational efficiency. Although explicit, purely functional definitions of effects make it easier to reason about programs, in practice key monads (such as state) are usually implemented imperatively by the compiler [LPJ95]. The efficiency problems are compounded for multi-level effects: in a naive implementation, the cost of each computational step is generally directly proportional to the total number of effects being modeled; and "built-in" monads are generally not integrable in a multi-effect framework with the same flexibility as user-specified ones.

Although the present paper is hardly the final word on either subject, it does present a framework for monadic effects addressing both problems in a novel way: conceptually, it offers a simple declarative specification based on nested translations, and practically, an efficient imperative implementation in terms of low-level control and state primitives.

In more detail, each notion of effect is specified independently by a formal monadic translation (state-passing, exception-passing, etc.), which also defines two proto-operations, monadic reflection and reification. These establish a (trivial) bijection between *opaque* and *transparent* representations of an effect-computation, allowing us to define the usual effect-operations in terms of the transparent representation, and then write general programs using only the abstract, opaque form. This specification is purely local, and can be written independently for each effect.

Secondly, we show that each such monadic translation can be simulated by a continuation-passing translation, which re-

---

[*] Basic Research in Computer Science (`http://www.brics.dk`), Centre of the Danish National Research Foundation.

POPL 99, San Antonio, Texas, USA
Copyright ACM 1999 1-58113-095-3/99/01...$5.00

$$\frac{\Gamma(x)=\tau}{\Gamma\vdash x:\tau\,/\,\mathfrak{n}}\qquad\frac{\Sigma(c)=\forall\vec{\alpha}.\,\tau}{\Gamma\vdash c_{\vec{\tau}}:\tau[\vec{\tau}/\vec{\alpha}]\,/\,\mathfrak{n}}\qquad\frac{\Gamma,x{:}\,\tau_1\vdash E:\tau_2\,/\,e}{\Gamma\vdash\lambda x^{\tau_1}.E:(\tau_1\xrightarrow{e}\tau_2)\,/\,\mathfrak{n}}\qquad\frac{\Gamma\vdash E_1:(\tau_1\xrightarrow{e}\tau_2)\,/\,\mathfrak{n}\qquad\Gamma\vdash E_2:\tau_1\,/\,\mathfrak{n}}{\Gamma\vdash E_1\,E_2:\tau_2\,/\,e}$$

$$\frac{\Gamma\vdash E:0\,/\,\mathfrak{n}}{\Gamma\vdash\mathcal{V}E:\tau\,/\,e}\qquad\frac{}{\Gamma\vdash():1\,/\,\mathfrak{n}}\qquad\frac{\Gamma\vdash E_1:\tau_1\,/\,\mathfrak{n}\qquad\Gamma\vdash E_2:\tau_2\,/\,\mathfrak{n}}{\Gamma\vdash(E_1,E_2):\tau_1\times\tau_2\,/\,\mathfrak{n}}\qquad\frac{\Gamma\vdash E:\tau_1\times\tau_2\,/\,\mathfrak{n}}{\Gamma\vdash\pi_1 E:\tau_1\,/\,\mathfrak{n}}\qquad\frac{\Gamma\vdash E:\tau_1\times\tau_2\,/\,\mathfrak{n}}{\Gamma\vdash\pi_2 E:\tau_2\,/\,\mathfrak{n}}$$

$$\frac{\Gamma\vdash E:\tau_1\,/\,\mathfrak{n}}{\Gamma\vdash\mathsf{inl}\,E:\tau_1+\tau_2\,/\,\mathfrak{n}}\qquad\frac{\Gamma\vdash E:\tau_2\,/\,\mathfrak{n}}{\Gamma\vdash\mathsf{inr}\,E:\tau_1+\tau_2\,/\,\mathfrak{n}}\qquad\frac{\Gamma\vdash E:\tau_1+\tau_2\,/\,\mathfrak{n}\qquad\Gamma,x_1{:}\,\tau_1\vdash E_1:\tau\,/\,e\qquad\Gamma,x_2{:}\,\tau_2\vdash E_2:\tau\,/\,e}{\Gamma\vdash\mathsf{case}\,(E,x_1.E_1,x_2.E_2):\tau\,/\,e}$$

$$\frac{\Gamma\vdash E:\tau\,/\,\mathfrak{n}}{\Gamma\vdash\mathbf{val}^e\,E:\tau\,/\,e}\qquad\frac{\Gamma\vdash E_1:\tau_1\,/\,e\qquad\Gamma,x{:}\,\tau_1\vdash E_2:\tau_2\,/\,e}{\Gamma\vdash\mathbf{let}^e\;x\Leftarrow E_1\;\mathbf{in}\;E_2:\tau_2\,/\,e}\qquad\frac{\Gamma\vdash E_1:\tau_1\,/\,e_1\qquad\Gamma,x{:}\,\tau_1\vdash E_2:\tau_2\,/\,e_2\qquad e_1\prec e_2}{\Gamma\vdash\mathbf{let}^{e_1,e_2}\;x\Leftarrow E_1\;\mathbf{in}\;E_2:\tau_2\,/\,e_2}$$

$$\frac{}{e\preceq e}\qquad\frac{e_1\preceq e\qquad e\prec e_2}{e_1\preceq e_2}\qquad\frac{}{\iota\leq\iota}\,{\scriptstyle(\iota=\alpha,b,1,0)}\qquad\frac{\tau_1'\leq\tau_1\qquad e\preceq e'\qquad\tau_2\leq\tau_2}{\tau_1\xrightarrow{e}\tau_2\leq\tau_1'\xrightarrow{e'}\tau_2'}\qquad\frac{\tau_1\leq\tau_1'\qquad\tau_2\leq\tau_2'}{\tau_1\diamond\tau_2\leq\tau_1'\diamond\tau_2'}\,{\scriptstyle(\diamond=\times,+)}$$

$$\frac{\Gamma\vdash E_1:\tau_1\,/\,e_1\qquad\Gamma,x{:}\,\tau_1\vdash E_2:\tau_2\,/\,e_2\qquad e_1\preceq e_2}{\Gamma\vdash\mathbf{let}^{e_1\preceq e_2}\;x\Leftarrow E_1\;\mathbf{in}\;E_2:\tau_2\,/\,e_2}\qquad\frac{\Gamma\vdash E:\tau\,/\,e\qquad\tau\leq\tau'\qquad e\preceq e'}{\Gamma\vdash E:\tau'\,/\,e'}$$

Figure 1: Effect-typing and subtyping rules

tains the original transparent representation of effects, but substitutes a different opaque one. We use this result to show correctness of a direct implementation of the proto-operations in terms of the control operators *shift* and *reset*. As previously shown [Fil94, Fil96], these operators can themselves be implemented by Scheme-style primitives call/cc and state, but this "uses up" the call/cc operation of the host language. Here we show how, in addition to defining shift and reset, we can "regenerate" a call/cc that can be used to implement other effects without interference. Thus, a whole monadic tower can be embedded in a language with Scheme-like primitive effects.

Accordingly, the paper is structured as follows: in Section 2, we present the translation-based specification of monadic effects. In Section 3, we show correctness of the continuation-based implementation using logical relations; we also present a concrete realization of the construction in SML/NJ. Section 4 contains a few practical examples, including a simulation of shared-state concurrency. Finally, Section 5 compares the present results with related work on monad layering and control operators, and Section 6 presents some conclusions and outlines future work.

2 Specifying layered effects

In this section, we introduce a simple functional language with a type system for keeping track of effect-behavior of terms, and show how this language can be systematically extended by new, programmer-defined effects.

2.1 A multi-effect language

For the purpose of the formal results, it will be convenient to work with a syntax in which all computational steps are explicitly sequenced. That is, the results of all non-trivial operations must be explicitly named, as in A-normal or monadic normal forms [FSDF93, HD94]. (Ultimately, however, we will still be able to write concrete programs in ML notation, with implicit call-by-value sequencing.)

We further refine this language with a type system for keeping track of effects, very similar to Tolmach's and (to a lesser degree) Wadler's intermediate languages for ML [Tol98, Wad98]. However, we will use effect-types *prescriptively*, to define a new language, rather than *descriptively*, to analyze an already given one.

The raw syntax is thus as follows:

$$\begin{aligned}\tau &::= \alpha\mid b\mid\tau_1\xrightarrow{e}\tau_2\mid 1\mid\tau_1\times\tau_2\mid 0\mid\tau_1+\tau_2\\ E &::= x\mid c_{\tau_1,\ldots,\tau_n}\mid\lambda x^{\tau}.E\mid E_1\,E_2\mid()\mid\mathcal{V}E\mid(E_1,E_2)\\ &\quad\;\mid\pi_1\,E\mid\pi_2\,E\mid\mathsf{inl}\,E\mid\mathsf{inr}\,E\mid\mathsf{case}\,(E,x_1.E_1,x_2.E_2)\\ &\quad\;\mid\mathbf{val}^e\,E\mid\mathbf{let}^e\;x\Leftarrow E_1\;\mathbf{in}\;E_2\mid\mathbf{let}^{e_1,e_2}\;x\Leftarrow E_1\;\mathbf{in}\;E_2\\ e &::= \mathfrak{n}\mid\mathfrak{p}\mid\cdots\end{aligned}$$

Here e ranges over a set of *effect names*, which classify the range of effects of an expression: the typing judgment $\Gamma\vdash E:\tau\,/\,e$ states that under typing assumptions Γ (mapping variables to types), E has type τ, and possible effects e.

In particular, the effect $\mathfrak{n}$ (*none*) means that evaluation of E has no effects, and that E therefore behaves as a value for the purpose of equational reasoning. Another distinguished effect is $\mathfrak{p}$ (*partiality*), which indicates that evaluation of E may diverge, but has no other effects.

The typing rules are displayed in Figure 1. They are parameterized by an effect-layering relation $\prec$, with $e_1\prec e_2$ expressing that e_2 is layered immediately above e_1, in a sense to be made precise in the next section. (Typically it means that e_2 was defined by a formal translation into a language with e_1-effects.) $\preceq$ is the reflexive, transitive closure of $\prec$.

We sometimes use pattern-matching binding syntax instead of projections; and in particular, we write $\lambda().E$ for $\lambda x^1.E$. We also often write $\tau_1\xrightarrow{\mathfrak{n}}\tau_2$ simply as $\tau_1\to\tau_2$, and $\Gamma\vdash E:\tau\,/\,\mathfrak{n}$ as $\Gamma\vdash E:\tau$.

A *complete program* is a closed term of base type. For simplicity, we may also put restrictions on the potentially "escaping" effects of such programs, in preference to complicating the top-level semantics. (For example, we may require that a complete program handles all exceptions it may raise, using a catch-all exception wrapper.)

The language is also parameterized by a signature Σ assigning (potentially polymorphic) types to the basic constants in the language. These would typically include the standard arithmetic functions, and in particular a family of CBV fixed-point operators,

$$\mathsf{fix}^e_{\alpha_1,\alpha_2}:((\alpha_1\xrightarrow{e}\alpha_2)\to\alpha_1\xrightarrow{e}\alpha_2)\to\alpha_1\xrightarrow{e}\alpha_2\qquad(\mathfrak{p}\preceq e)$$

(Note that all recursively defined functions will thus have at least the effect of partiality.) Although complete programs will be monomorphic, we need the extra generality of type schemas in order to define monad components as polymorphic terms in Section 2.3.

The last two lines of Figure 1 add implicit subeffecting and subtyping to the language. (But note that the only subtyping relations introduced are changes of effect-annotations.) We can expand all occurrences of the general **let**-construct into the two primitive ones as follows:

$$\begin{aligned}
\mathbf{let}^{e \preceq e}\ x \Leftarrow E_1\ \mathbf{in}\ E_2 &= \mathbf{let}^{e}\ x \Leftarrow E_1\ \mathbf{in}\ E_2 \\
\mathbf{let}^{e_1 \preceq e_2}\ x \Leftarrow E_1\ \mathbf{in}\ E_2 &= \\
&\mathbf{let}^{e,e_2}\ x \Leftarrow (\mathbf{let}^{e_1 \preceq e}\ x \Leftarrow E_1\ \mathbf{in}\ \mathbf{val}^{e}\ x)\ \mathbf{in}\ E_2
\end{aligned}$$

Similarly, well-typed terms in the system with subsumption can be expanded into the system without, by replacing all instances of the last rule by appropriate *coercion terms*:

$$\begin{aligned}
\chi^{\tau/e \leq \tau'/e'}(E) &= \mathbf{let}^{e \preceq e'}\ x \Leftarrow E\ \mathbf{in}\ \mathbf{val}^{e'}\ \chi^{\tau \leq \tau'}(x) \\
\chi^{\iota \leq \iota}(a) &= a \\
\chi^{\tau_1 \xrightarrow{e} \tau_2 \leq \tau_1' \xrightarrow{e'} \tau_2'}(f) &= \lambda a.\, \chi^{\tau_2/e \leq \tau_2'/e'}(f\,(\chi^{\tau_1' \leq \tau_1}(a))) \\
\chi^{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}(p) &= (\chi^{\tau_1 \leq \tau_1'}(\pi_1\, p), \chi^{\tau_2 \leq \tau_2'}(\pi_2\, p)) \\
\chi^{\tau_1 + \tau_2 \leq \tau_1' + \tau_2'}(s) &= \mathsf{case}(s, x_1.\ \mathsf{inl}\,(\chi^{\tau_1 \leq \tau_1'}(x_1)) \\
&\qquad\qquad x_2.\ \mathsf{inr}\,(\chi^{\tau_2 \leq \tau_2'}(x_2)))
\end{aligned}$$

As always with subtyping systems, there is a potential coherence problem: the semantics must ensure that different ways of inserting coercions in a program give the same result. We will see that for our application, this is automatically ensured (see the note following Theorem 11).

2.2 Layered monads and effect-semantics

For concreteness, we present a denotational semantics of our language in the setting of CPOs (chain-complete posets, not necessarily containing least elements) and continuous functions. This specific choice is not essential, however.

We start with a standard concept:

Definition 1 *A* monad $\mathcal{T}$ *consists of a triple* $(T, \eta, \star)$, *where* T *maps every CPO* A *to a CPO of* A-computations*;* η *is a family of* value-inclusion *functions* $\eta_A : A \to TA$*; and* $\star$ *(normally written infix) is a family of* binding *functions* $\star_{A,B} : TA \times (A \to TB) \to TB$*; such that for any* $f : A \to TB$ *and* $g : B \to TC$,

$$\eta a \star f = f\, a, \quad t \star \eta = t, \ \ \textit{and} \quad (t \star f) \star g = t \star (\lambda a.\, f\, a \star g)$$

(We generally omit the type subscripts on η *and* $\star$ *when they are clear from the context.) When* $f : A \to B$*, we also define* $Tf = \lambda t.\, t \star (\eta \circ f) : TA \to TB$*; and when* $f : A \to TB$*,* $f^* = \lambda t.\, t \star f : TA \to TB$*.*

Intuitively, elements of TA represent effectful computations yielding values in A. ηa represents the trivial (effect-free) computation of a, while $t \star f$ represents the computation consisting of evaluating t to a value a (possibly with some effects), followed by evaluating $f\, a$ (again possibly with effects). The monad laws ensure that the sequencing of effects is well-behaved.

A very simple monad is the *identity* monad, with $IA = A$, $\eta_A\, a = a$, and $t \star f = f\, t$. An important non-trivial example is the *lifting* monad, where the computation-type constructor is domain-theoretic lifting, $LA = A_\bot$; unit is the inclusion, $\eta a = [a]$; and binding is strict extension, $\bot \star f = \bot$ and $[a] \star f = f a$.

We now introduce a concept useful for stacking monads:

Definition 2 *A* layering *of a monad* $\mathcal{T}$ *over another monad* $\overline{\mathcal{T}} = (\overline{T}, \bar{\eta}, \bar{\star})$ *consists of a function family* $\zeta_A : \overline{T}(TA) \to TA$*, such that each* (TA, ζ_A) *is a* $\overline{\mathcal{T}}$*-algebra [ML71, VI.2], i.e.,*

$$\zeta_A \circ \bar{\eta}_{TA} = \mathsf{id}_{TA} \quad \textit{and} \quad \zeta_A \circ \mathsf{id}^{\bar{*}}_{\overline{T}(TA)} = \zeta_A \circ \overline{T}\zeta_A$$

and such that every f^* *is a* $\overline{T}$*-algebra morphism, i.e.,*

$$f^* \circ \zeta_A = \zeta_B \circ \overline{T} f^* : \overline{T}(TA) \to TB$$

The definition of layering is a bit more technical, but captures the requirement that a $\overline{\mathcal{T}}$-computation can be meaningfully interpreted as a more general $\mathcal{T}$-computation. When T is explicitly constructed in terms of $\overline{T}$, we can generally obtain a suitable ζ directly from the *shape* of T, as shown in Section 2.3. And the additional condition on f^* is usually immediate to verify – informally it expresses that the T-computation represented by $t \star f$ performs any latent $\overline{T}$-effects in t “first”.

Any monad $\mathcal{T}$ can be layered over itself by $\mathsf{id}^*_{TA} : T(TA) \to TA$. And, if $\mathcal{T}$ is layered over $\overline{\mathcal{T}}$ by ζ, and $\overline{\mathcal{T}}$ over $\overline{\overline{\mathcal{T}}}$ by $\bar{\zeta}$, then $\mathcal{T}$ is also layered over $\overline{\overline{\mathcal{T}}}$ by $\zeta_A \circ \bar{\zeta}_{TA} \circ \overline{\overline{T}}\bar{\eta}_{TA} : \overline{\overline{T}}(TA) \to TA$.

When $\mathcal{T}$ is layered over $\overline{\mathcal{T}}$, we can also define a *computation-inclusion* or *lifting* [LHJ95, Tol98] function family $i_A = \zeta_A \circ \overline{T}\eta_A : \overline{T}A \to TA$. This is easily checked to be a monad morphism [Mog90], i.e., to satisfy the equations

$$i_A\,(\bar{\eta}_A\, a) = \eta_A\, a \quad \text{and} \quad i_B\,(t' \mathbin{\bar{\star}} f) = i_A\, t' \star (\lambda a.\, i_B\,(f\, a))$$

Conversely, given a monad morphism $i : \overline{T} \to T$, we can obtain a layering by $\zeta_A = id^*_{TA} \circ i_A$. Both formulations allow us to define a “mixed” binding operation $\tilde{\star}_{A,B} : \overline{T}A \times (A \to TB) \to TB$ by:

$$t' \mathbin{\tilde{\star}_{A,B}} f = (i_A\, t') \star_{A,B} f = \zeta_B\,(t' \mathbin{\bar{\star}_{A,TB}} (\lambda a.\, \bar{\eta}_{TB}\,(f\, a)))$$

Defining layering in terms of inclusions may seem more natural, but it turns out that taking ζ as the primitive notion leads to a more direct implementation of monadic effects in Section 3.

Any monad can be trivially layered over identity by taking $\zeta_A\, t = t$. It can be layered over lifting when each TA is a pointed CPO (i.e., has a least element $\bot_{TA}$, allowing $\zeta\,\bot = \bot_{TA}$ and $\zeta\,[t] = t$) and f^* is always strict (that is, if the original computation t has a divergence-effect, then so will $t \star f$). Conversely, if $\mathcal{T}$ is layered over lifting, TA is necessarily pointed (because for any $t \in TA$, $\zeta\,\bot \sqsubseteq \zeta\,[t] = t$), and f^* is strict (because $f^*\,(\zeta\,\bot) = \zeta\,(Lf^*\,\bot) = \zeta\,\bot$).

We can now define a semantics of our language. This will be done in the style of Church, i.e., we only give meanings to well-typed terms:

Definition 3 (effect-semantics) *A semantics* $\mathcal{L}$ *of an effect-language* L *assigns first to every base type* b*, a CPO* $\mathcal{B}(b)$*; and to every effect* e*, a monad* $\mathcal{E}(e) = (T^e, \eta^e, \star^e)$*, such that if* $e' \prec e$ *then* $\mathcal{E}(e)$ *is layered over* $\mathcal{E}(e')$ *by* ζ^e*. This assignment induces a semantics of general* L*-type phrases as follows. Let* ϱ *be an assignment of CPOs to type variables in* $\vec{\alpha}$*. Then for any type* τ *over* $\vec{\alpha}$*, we define a CPO* $\mathcal{L}[\![\tau]\!]_\varrho$*, as follows:*

$$\begin{aligned}
\mathcal{L}[\![\alpha]\!]_\varrho &= \varrho\,\alpha \\
\mathcal{L}[\![b]\!]_\varrho &= \mathcal{B}(b) \\
\mathcal{L}[\![\tau_1 \times \tau_2]\!]_\varrho &= \mathcal{L}[\![\tau_1]\!]_\varrho \times \mathcal{L}[\![\tau_2]\!]_\varrho \\
\mathcal{L}[\![\tau_1 + \tau_2]\!]_\varrho &= \mathcal{L}[\![\tau_1]\!]_\varrho + \mathcal{L}[\![\tau_2]\!]_\varrho \\
\mathcal{L}[\![\tau_1 \xrightarrow{e} \tau_2]\!]_\varrho &= \mathcal{L}[\![\tau_1]\!]_\varrho \to T^e \mathcal{L}[\![\tau_2]\!]_\varrho
\end{aligned}$$

This also extends straightforwardly to a semantics of type assignments,

$$\mathcal{L}\llbracket\Gamma\rrbracket_\varrho = \prod_{x \in \mathrm{dom}\,\Gamma} \mathcal{L}\llbracket\Gamma(x)\rrbracket_\varrho$$

Further, $\mathcal{L}$ must assign to every constant c of L a family of elements $\mathcal{C}(c)_{\vec{A}} \in \mathcal{L}\llbracket\Sigma(c)\rrbracket_{[\vec{\alpha}\mapsto\vec{A}]}$. Then we define for every well-typed term $\Gamma \vdash_{\vec{\alpha}} E : \tau/e$, its meaning $\mathcal{L}\llbracket E\rrbracket_\varrho : \mathcal{L}\llbracket\Gamma\rrbracket_\varrho \to T^e\mathcal{L}\llbracket\tau\rrbracket_\varrho$ (omitting the subscript ϱ on all semantic brackets):

$$\begin{aligned}
\mathcal{L}\llbracket x\rrbracket\rho &= \rho x \\
\mathcal{L}\llbracket c_{\tau_1,\ldots,\tau_n}\rrbracket\rho &= \mathcal{C}(c)_{\mathcal{L}\llbracket\tau_1\rrbracket,\ldots,\mathcal{L}\llbracket\tau_n\rrbracket} \\
\mathcal{L}\llbracket\lambda x^\tau.\,E\rrbracket\rho &= \lambda a^{\mathcal{L}\llbracket\tau\rrbracket}.\,\mathcal{L}\llbracket E\rrbracket\rho[x\mapsto a] \\
\mathcal{L}\llbracket E_1\,E_2\rrbracket\rho &= \mathcal{L}\llbracket E_1\rrbracket\rho\,(\mathcal{L}\llbracket E_2\rrbracket\rho) \\
\mathcal{L}\llbracket\mathbf{val}^e\,E\rrbracket\rho &= \eta^e\,(\mathcal{L}\llbracket E\rrbracket\rho) \\
\mathcal{L}\llbracket\mathbf{let}^e\ x \Leftarrow E_1\ \mathbf{in}\ E_2\rrbracket\rho &= \mathcal{L}\llbracket E_1\rrbracket\rho \star^e (\lambda a.\,\mathcal{L}\llbracket E_2\rrbracket\rho[x\mapsto a]) \\
\mathcal{L}\llbracket\mathbf{let}^{e_1,e_2}\ x \Leftarrow E_1\ \mathbf{in}\ E_2\rrbracket\rho &= \\
&\zeta^{e_2}(\mathcal{L}\llbracket E_1\rrbracket\rho \star^{e_1} (\lambda a.\,\eta^{e_1}(\mathcal{L}\llbracket E_2\rrbracket\rho[x\mapsto a])))
\end{aligned}$$

(together with straightforward clauses for sums and products).

Finally, $\mathcal{L}$ must include a collection of result interpretation *functions $\xi^e_b : T^e\mathcal{B}(b) \to \mathbf{P}_\bot$ from meanings of complete programs to* observations, *where $\mathbf{P}$ is some countable set of final outputs, such as character strings. For a complete program, $\vdash E : b\ /\ e$, we can then define*

$$\hat{\mathcal{L}}\llbracket E\rrbracket = \xi^e_b\,(\mathcal{L}\llbracket E\rrbracket_\emptyset\,\emptyset)$$

In the standard semantics of the language, we take $\mathcal{E}(\mathfrak{n})$ as the identity monad and $\mathcal{E}(\mathfrak{p})$ as lifting. (An implementation semantics, however, may use a different interpretation: for example, we can give a continuation-based semantics of partiality; the result interpretation function must then be adjusted appropriately.) We will also always use the standard interpretation of fix,

$$\mathcal{C}(\mathsf{fix}^e)_{A_1,A_2} = \lambda f^{(A_1\to T^eA_2)\to A_1\to T^eA_2}.\bigsqcup\nolimits_{i\in\omega} f^i\,(\lambda a.\,\bot_{T^eA_2})$$

where T^eA_2 is pointed because e is layered above partiality.

We can reason about terms in the language by means of a formal equational theory, including in particular the equations

$$\begin{aligned}
(\lambda x.\,E_1)\,E_2 &= E_1[E_2/x] \\
\lambda x.\,E x &= E \quad {\scriptstyle (x\notin FV(E))} \\
\mathbf{val}^{\mathfrak{n}}\,E &= E \\
\mathbf{let}^{e_1\preceq e_2}\ x \Leftarrow \mathbf{val}^{e_1}\,E_1\ \mathbf{in}\ E_2 &= E_2[E_1/x] \\
\mathbf{let}^e\ x \Leftarrow E\ \mathbf{in}\ \mathbf{val}^e\,x &= E \\
\mathbf{let}^{e_2\preceq e_3}\ x_2 \Leftarrow (\mathbf{let}^{e_1\preceq e_2}\ x_1 \Leftarrow E_1\ \mathbf{in}\ E_2)\ \mathbf{in}\ E_3 & \\
&= \\
\mathbf{let}^{e_1\preceq e_3}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{let}^{e_2\preceq e_3}\ x_2 \Leftarrow E_2\ \mathbf{in}\ E_3 & \\
&{\scriptstyle (x_1\notin FV(E_3))}
\end{aligned}$$

(together with the usual ones for products and sums). Note that the strong $\beta\eta$-rules for functions are only valid because we restrict the terms in an application to have no effects.

2.3 Adding a new effect

Of course, we can always enrich the language by adding a new effect at the level of the semantics. But a wide variety of effects can also be defined purely syntactically:

Definition 4 *A* formal monad $\mathbf{T}$ *over effect $\bar{e}$ in L consists of a type constructor and three polymorphic terms,*

$$\begin{aligned}
T- &: \mathit{Type} \to \mathit{Type} \\
\mathit{glue}_\alpha &: (1 \xrightarrow{\bar{e}} T\alpha) \to T\alpha \\
\mathit{unit}_\alpha &: \alpha \to T\alpha \\
\mathit{bind}_{\alpha_1,\alpha_2} &: T\alpha_1 \times (\alpha_1 \to T\alpha_2) \to T\alpha_2
\end{aligned}$$

Such a $\mathbf{T}$ denotes an (actual) monad $\mathcal{L}\llbracket\mathbf{T}\rrbracket^{\mathrm{m}} = (T,\eta,\star,\zeta)$ layered over $\mathcal{E}(\bar{e})$ in a semantics $\mathcal{L}$ of L if $\mathcal{L}\llbracket T\alpha\rrbracket_{[\alpha\mapsto A]} = TA$, $\mathcal{L}\llbracket\mathit{glue}_\alpha\rrbracket_{[\alpha\mapsto A]}\emptyset = \lambda t.\,\zeta_A\,(t\,())$, $\mathcal{L}\llbracket\mathit{unit}_\alpha\rrbracket_{[\alpha\mapsto A]}\emptyset = \eta_A$, and $\mathcal{L}\llbracket\mathit{bind}_{\alpha_1,\alpha_2}\rrbracket_{[\alpha_1\mapsto A_1,\alpha_2\mapsto A_2]}\emptyset = (\star_{A_1,A_2})$.

Note that we do not require that $\mathbf{T}$ denote a monad for all interpretations of the effect $\bar{e}$. For example, the formal list monad (used to give a semantics of nondeterminism in Section 4.2) can only be properly layered over a *commutative* monad [KW93], such as partiality.

The component *glue* can usually be constructed systematically from just the form of T, as follows:

$T\alpha$	$\mathit{glue}^T_\alpha : (1 \xrightarrow{\bar{e}} T\alpha) \to T\alpha$
$F\alpha \xrightarrow{\bar{e}} G\alpha$	$\lambda t.\,\lambda a.\,\mathbf{let}^{\bar{e}}\ f \Leftarrow t\,()\ \mathbf{in}\ f\,a$
$F\alpha \xrightarrow{\mathfrak{n}} T'\alpha$	$\lambda t.\,\lambda a.\,\mathit{glue}^{T'}_\alpha\,(\lambda().\,\mathbf{let}^{\bar{e}}\ f \Leftarrow t\,()\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\,f\,a)$
$T_1\alpha \times T_2\alpha$	$\lambda t.\,(\mathit{glue}^{T_1}_\alpha\,(\lambda().\,\mathbf{let}^{\bar{e}}\ p \Leftarrow t\,()\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\,\pi_1\,p),$ $\mathit{glue}^{T_2}_\alpha\,(\lambda().\,\mathbf{let}^{\bar{e}}\ p \Leftarrow t\,()\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\,\pi_2\,p))$

(In fact, in most cases, the first rule alone suffices.) It is easy to check that such a glue^T satisfies the $\overline{T}$-algebra conditions of Definition 2; and the verification of the additional property of *bind* is usually straightforward.

Given $\mathbf{T}$ we construct an extension L^T of L, with a new effect $\mathfrak{t}$ and new proto-operations, *monadic reflection* and *reification* [Fil94]:

$$\begin{aligned}
\mathsf{reflect}^{\mathfrak{t}}_\alpha &: T\alpha \xrightarrow{\mathfrak{t}} \alpha \\
\mathsf{reify}^{\mathfrak{t}}_\alpha &: (1 \xrightarrow{\mathfrak{t}} \alpha) \to T\alpha
\end{aligned}$$

When $E : T\tau/\mathfrak{n}$, we often write $\mu^{\mathfrak{t}}(E) : \tau/\mathfrak{t}$ for $\mathsf{reflect}^{\mathfrak{t}}_\tau\,E$; and conversely, when $E : \tau/\mathfrak{t}$, $[E]^{\mathfrak{t}} : T\tau/\mathfrak{n}$ for $\mathsf{reify}^{\mathfrak{t}}\,(\lambda().\,E)$. Informally, evaluating $\mu(E)$ performs the action represented by the datum E, while $[E]$ returns a datum representing the action that would result from evaluating E.

Note that L^T is a proper extension of L: any L-program is still a valid L^T program with the same meaning. This means that we can define the semantics of L^T by a formal monadic translation $|{-}|_{\mathbf{T}}$ back into L, expanding out only the new type and term constructors into their L-definitions.

First, the translation of an effect e is an effect-type constructor $|e|_T$:

$$\begin{aligned}
|\mathfrak{t}|\alpha &= T\alpha/\mathfrak{n} \\
|e|\alpha &= \alpha/e \quad (e \neq \mathfrak{t})
\end{aligned}$$

For types, only function spaces have a non-trivial translation:

$$|\tau_1 \xrightarrow{e} \tau_2| = |\tau_1| \xrightarrow{e'} \tau'_2 \text{ where } \tau'_2/e' = |e|\,|\tau_2|$$

And finally, we define a translation $|{-}|_{\mathbf{T}}$ of terms: if $\Gamma \vdash E : \tau\ /\ e$ then $|\Gamma|_T \vdash |E|_{\mathbf{T}} : |e|_T|\tau|_T$. Again, the only non-trivial clauses are:

$$|\mathbf{val}^{\mathfrak{t}}\,E| = \mathit{unit}_{|\tau|}\,|E|$$

$$
\begin{aligned}
|\mathbf{let}^{\mathsf{t}}\ x \Leftarrow E_1\ \mathbf{in}\ E_2| &= \mathit{bind}_{|\tau_1|,|\tau_2|}\,(|E_1|, \lambda x^{|\tau_1|}.\,|E_2|) \\
|\mathbf{let}^{\bar{e},\mathsf{t}}\ x \Leftarrow E_1\ \mathbf{in}\ E_2| &= \\
&\quad \mathit{glue}_{|\tau_2|}\,(\lambda().\,\mathbf{let}^{\bar{e}}\ x \Leftarrow |E_1|\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\ |E_2|) \\
|\mathsf{reflect}^{\mathsf{t}}_{\tau}| &= \lambda t^{T|\tau|}.\,t \\
|\mathsf{reify}^{\mathsf{t}}_{\tau}| &= \lambda t^{1 \to T|\tau|}.\,t\,()
\end{aligned}
$$

This syntactic translation agrees with the denotational semantics:

Lemma 5 $\mathcal{L}[\![|-|_{\mathsf{T}}]\!] = \mathcal{L}^{\mathcal{T}}[\![-]\!]$ *where* $\mathcal{L}^{\mathcal{T}}$ *is the semantics of* L^{T} *determined by extending* $\mathcal{L}$ *with* $\mathcal{E}(\mathsf{t}) = \mathcal{T} = \mathcal{L}[\![\mathbf{T}]\!]^{\mathrm{m}}$, $\mathcal{C}(\mathsf{reflect})_A = \lambda t^{TA}.\,t$, *and* $\mathcal{C}(\mathsf{reify})_A = \lambda t^{1 \to TA}.\,t\,()$.

For example, given some base type *exn* of exception names, the formal monad **Ex** of exceptions is defined by:

$$
\begin{aligned}
T\alpha &= 1 \xrightarrow{\bar{e}} (\alpha + \mathit{exn}) \\
\mathit{glue}_{\alpha} &= \lambda t^{1 \xrightarrow{\bar{e}} T\alpha}.\,\lambda().\,\mathbf{let}^{\bar{e}}\ r \Leftarrow t\,()\ \mathbf{in}\ r\,() \\
\mathit{unit}_{\alpha} &= \lambda a^{\alpha}.\,\lambda().\,\mathsf{inl}\,a \\
\mathit{bind}_{\alpha_1,\alpha_2} &= \lambda(t,f)^{T\alpha_1 \times (\alpha_1 \to T\alpha_2)}.\,\lambda(). \\
&\qquad \mathbf{let}^{\bar{e}}\ r \Leftarrow t\,()\ \mathbf{in}\ \mathsf{case}\,(r, a.f\,a\,(), x.\mathbf{val}^{\bar{e}}\ \mathsf{inr}\,x)
\end{aligned}
$$

This does in fact denote a monad for any monadic interpretation of $\bar{e}$. Calling the new effect $\mathfrak{ex}$, we can define operations in L^{T} with types

$$
\begin{aligned}
\mathit{raise}_{\alpha} &: \mathit{exn} \xrightarrow{\mathfrak{ex}} \alpha \\
\mathit{handle}_{\alpha} &: (1 \xrightarrow{\mathfrak{ex}} \alpha) \to (\mathit{exn} \xrightarrow{\mathfrak{ex}} \alpha) \xrightarrow{\mathfrak{ex}} \alpha
\end{aligned}
$$

in terms of the proto-operations:

$$
\begin{aligned}
\mathit{raise}_{\alpha} &= \lambda x^{\mathit{exn}}.\,\mu^{\mathfrak{ex}}(\lambda().\,\mathbf{val}^{\bar{e}}\ \mathsf{inr}\,x) \\
\mathit{handle}_{\alpha} &= \lambda t^{1 \xrightarrow{\mathfrak{ex}} \alpha}.\,\lambda h^{\mathit{exn} \xrightarrow{\mathfrak{ex}} \alpha}.\,\mathbf{let}^{\bar{e},\mathfrak{ex}}\ r \Leftarrow [t\,()]^{\mathfrak{ex}}\,() \\
&\qquad \mathbf{in}\ \mathsf{case}\,(r, a.\mathbf{val}^{\mathfrak{ex}}\,a, x.h\,x)
\end{aligned}
$$

That is, *raise* constructs an exception-computation that immediately returns with a right-tagged x, and reflects that as an $\mathfrak{ex}$-effect. Conversely *handle* reifies t into a $\bar{e}$-computation of a sum-typed value, performs that computation, and either returns the result or passes the exception name to the handler h. It is easy to check that the semantics of these operations capture the usual behavior of ML-style exceptions, even in the presence of state as an $\bar{e}$-effect.

Note that we can often reason about L^{T}-programs directly, without either expanding them into L-programs, or computing their denotational meanings. In particular, we have the following sound equations for well-typed terms:

$$
\begin{aligned}
\mu([E]) &= E \\
[\mu(E)] &= E \\
[\mathbf{val}^{\mathsf{t}}\,E] &= \mathit{unit}\,E \\
[\mathbf{let}^{\mathsf{t}}\ x \Leftarrow E_1\ \mathbf{in}\ E_2] &= \mathit{bind}\,([E_1], \lambda x.\,[E_2]) \\
[\mathbf{let}^{e \preceq \mathsf{t}}\ x \Leftarrow E_1\ \mathbf{in}\ E_2] &= \\
&\quad \mathit{glue}\,(\lambda().\,\mathbf{let}^{e \preceq \bar{e}}\ x \Leftarrow E_1\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\ [E_2]) \quad (e \preceq \bar{e} \prec \mathsf{t})
\end{aligned}
$$

From these, and the equations at the end of Section 2.2, it is easy to derive laws about the particular operations, such as

$$
\begin{aligned}
\mathbf{let}^{\mathfrak{ex}}\ r \Leftarrow \mathit{raise}\,x\ \mathbf{in}\ E &= \mathit{raise}\,x \\
\mathit{handle}\,(\lambda().\,\mathbf{val}^{\mathfrak{ex}}\,a)\,h &= \mathbf{val}^{\mathfrak{ex}}\,a \\
\mathit{handle}\,(\lambda().\,\mathit{raise}\,x)\,h &= h\,x
\end{aligned}
$$

2.4 Effect-ordering and monadic reflection

The hierarchical organization among the effects is crucial to the translation-based definition. Although being able to integrate effects "flatly" (i.e., with no mutual ordering) might seem a desirable goal, often different orderings correspond to different intended semantics, as illustrated below.

An important consequence of the layering is that some source terms are meaningless, even if their effect-erasures are simply-typable. Specifically, attempting to apply the reification operator of a lower-level effect to a higher-level computation has no counterpart in a program written with explicit effect-passing, and thus cannot be given a well-defined meaning by the translation. Where appropriate, the desired meaning of such a construct must instead be expressed explicitly. As a simple example, let us analyze this issue in the context of mutable state and exceptions.

We saw the definitions of *raise* and *handle* in Section 2.3. Similarly, using the formal state monad **St**, with type constructor

$$
ST\alpha = \mathit{state} \xrightarrow{\bar{e}} (\alpha \times \mathit{state})
$$

for some base type *state*, we can define a new effect $\mathfrak{st}$, with operations

$$
\begin{aligned}
\mathit{get} &: 1 \xrightarrow{\mathfrak{st}} \mathit{state} \\
\mathit{set} &: \mathit{state} \xrightarrow{\mathfrak{st}} 1 \\
\mathit{withst}_{\alpha} &: \mathit{state} \times (1 \xrightarrow{\mathfrak{st}} \alpha) \xrightarrow{\bar{e}} \alpha
\end{aligned}
$$

as follows:

$$
\begin{aligned}
\mathit{get} &= \lambda().\,\mu^{\mathfrak{st}}(\lambda s.\,\mathbf{val}^{\bar{e}}\ (s, s)) \\
\mathit{set} &= \lambda n.\,\mu^{\mathfrak{st}}(\lambda s.\,\mathbf{val}^{\bar{e}}\ ((), n)) \\
\mathit{withst}_{\alpha} &= \lambda(s,t).\,\mathbf{let}^{\bar{e}}\ (a, s') \Leftarrow [t\,()]^{\mathfrak{st}}\,s\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\ a
\end{aligned}
$$

But how do these operations interact with those for exceptions?

First, consider the ML-like layering of exceptions above state above partiality, $\mathfrak{p} \prec \mathfrak{st} \prec \mathfrak{ex}$, with the state persistent across exceptions. The composite translation, defining away first exceptions and then state, corresponds to the following effect type:

$$
\begin{aligned}
||\alpha/\mathfrak{ex}|_{EX}|_{ST} &= |1 \xrightarrow{\mathfrak{st}} \alpha + \mathit{exn}|_{ST} \\
&= 1 \to \mathit{state} \xrightarrow{\mathfrak{p}} (\alpha + \mathit{exn}) \times \mathit{state}
\end{aligned}
$$

That is, when started in some initial state from *state*, a computation may diverge; or it may result in a new state together with either an α-value or raised exception from *exn*. The proto-operators have types:

$$
\begin{aligned}
\mathsf{reify}^{\mathfrak{st}}_{\alpha} &: (1 \xrightarrow{\mathfrak{st}} \alpha) \to \mathit{state} \xrightarrow{\mathfrak{p}} \alpha \times \mathit{state} \\
\mathsf{reflect}^{\mathfrak{st}}_{\alpha} &: (\mathit{state} \xrightarrow{\mathfrak{p}} \alpha \times \mathit{state}) \xrightarrow{\mathfrak{st}} \alpha \\
\mathsf{reify}^{\mathfrak{ex}}_{\alpha} &: (1 \xrightarrow{\mathfrak{ex}} \alpha) \to 1 \xrightarrow{\mathfrak{st}} (\alpha + \mathit{exn}) \\
\mathsf{reflect}^{\mathfrak{ex}}_{\alpha} &: (1 \xrightarrow{\mathfrak{st}} (\alpha + \mathit{exn})) \xrightarrow{\mathfrak{ex}} \alpha
\end{aligned}
$$

In this setting, we can always coerce a $\mathfrak{st}$-computation into an $\mathfrak{ex}$-computation (and with the subtyping system, this coercion can be left implicit). This means that the previous definition of *handle*, based on $\mathfrak{ex}$-reification, works without changes even when the expression being guarded has only state-effects (which includes partiality-effects).

But if we want to extend *withst* to computations which may raise exceptions, we must explicitly account for those.

For example, we can define a more general operation

$$\begin{aligned}
&\mathit{withst'}_\alpha \;:\; \mathit{state} \times (1 \xrightarrow{\mathfrak{er}} \alpha) \xrightarrow{\mathfrak{er}} \alpha \\
&\mathit{withst'}_\alpha \;= \\
&\quad \lambda(s,t).\, \mathbf{let}^{\mathfrak{er}}\ (r,s') \Leftarrow [[t\,()]^{\mathfrak{er}}\,()]^{\mathfrak{st}}\, s\ \mathbf{in}\ \mu^{\mathfrak{er}}(\lambda().\, \mathbf{val}^{\mathfrak{st}}\, r)
\end{aligned}$$

Here we first explicitly $\mathfrak{er}$-reify the original computation of α into a $\mathfrak{st}$-computation of $\alpha + \mathit{exn}$, perform the original *withst*-operation, and finally $\mathfrak{er}$-reflect back the result, which may have the effect of raising an exception.

Note that when *withst'*, is applied to a computation that provably does not have exception-effects, its behavior also provably coincides with the original *withst*. That is, we can show

$$\begin{aligned}
&\mathit{withst'}(s, \lambda().\, \mathbf{let}^{\mathfrak{st},\mathfrak{er}}\ x \Leftarrow E\ \mathbf{in}\ \mathbf{val}^{\mathfrak{er}}\, x) \\
&\quad = \mathbf{let}^{\mathfrak{p} \preceq \mathfrak{er}}\ x \Leftarrow \mathit{withst}(s, \lambda().\, E)\ \mathbf{in}\ \mathbf{val}^{\mathfrak{er}}\, x
\end{aligned}$$

using the equations from the end of Section 2.3. In a general optimizer for higher-order programs, however, it seems preferable to maintain explicit effect-types everywhere, as advocated by Tolmach [Tol98].

Note also that the behavior of *withst'* differs from that of a more naive definition,

$$\begin{aligned}
&\mathit{withst''}_\alpha \;= \\
&\lambda(s,t).\, \mathbf{let}^{\mathfrak{st},\mathfrak{er}}\ o \Leftarrow \mathit{get}\,() \\
&\qquad \mathbf{in}\ \mathbf{let}^{\mathfrak{st},\mathfrak{er}}\ () \Leftarrow \mathit{set}\, s \\
&\qquad\quad \mathbf{in}\ \mathbf{let}^{\mathfrak{er}}\ r \Leftarrow t\,()\ \mathbf{in}\ \mathbf{let}^{\mathfrak{st},\mathfrak{er}}\ () \Leftarrow \mathit{set}\, o\ \mathbf{in}\ \mathbf{val}^{\mathfrak{er}}\, r
\end{aligned}$$

Here, an unhandled exception raised in $t\,()$ does modify the global state, which makes an observable difference if the exception is eventually handled somewhere.

Yet both versions behave the same on programs that never actually raise exceptions. Thus, it is not sufficient to merely require that programs in the original language (partiality and state) should retain their meaning in the extended language (partiality, state, and exceptions); the interactions of effects need to be considered explicitly in each case.

On the other hand, suppose we want a transactional semantics with transient state layered atop exceptions, corresponding to effect-type

$$\begin{aligned}
||\alpha/\mathfrak{st}|_{ST}|_{EX} &= |\mathit{state} \xrightarrow{\mathfrak{er}} \alpha \times \mathit{state}|_{EX} \\
&= \mathit{state} \to 1 \xrightarrow{\mathfrak{p}} (\alpha \times \mathit{state}) + \mathit{exn}
\end{aligned}$$

Here the type already shows that the state must be discarded if a computation aborts with an exception. When $\mathfrak{p} \prec \mathfrak{er} \prec \mathfrak{st}$, the monad reflection operators have types

$$\begin{aligned}
\mathsf{reify}^{\mathfrak{st}}_\alpha &: (1 \xrightarrow{\mathfrak{st}} \alpha) \to \mathit{state} \xrightarrow{\mathfrak{er}} \alpha \times \mathit{state} \\
\mathsf{reflect}^{\mathfrak{st}}_\alpha &: (\mathit{state} \xrightarrow{\mathfrak{er}} \alpha \times \mathit{state}) \xrightarrow{\mathfrak{st}} \alpha \\
\mathsf{reify}^{\mathfrak{er}}_\alpha &: (1 \xrightarrow{\mathfrak{er}} \alpha) \to 1 \xrightarrow{\mathfrak{p}} (\alpha + \mathit{exn}) \\
\mathsf{reflect}^{\mathfrak{er}}_\alpha &: (1 \xrightarrow{\mathfrak{p}} (\alpha + \mathit{exn})) \xrightarrow{\mathfrak{er}} \alpha
\end{aligned}$$

Thus, $\mathfrak{st}$-reification of any computation is well-defined, but we need to explicitly define the meaning of $\mathfrak{er}$-reifying a computation which may have state effects, as when handling exceptions. In this case, a natural revised definition of *handle* is

$$\begin{aligned}
\mathit{handle'}_\alpha \;=\; &\lambda t^{1 \xrightarrow{\mathfrak{st}} \alpha}.\, \lambda h^{\mathit{exn} \xrightarrow{\mathfrak{st}} \alpha}. \\
&\mu^{\mathfrak{st}}(\lambda s.\, \mathbf{let}^{\bar{e},\mathfrak{er}}\ r \Leftarrow [[t\,()]^{\mathfrak{st}}\, s]^{\mathfrak{er}}\,() \\
&\qquad \mathbf{in}\ \mathsf{case}\,(r, (a,s').\mathbf{val}^{\mathfrak{er}}\,(a,s'), x.[h\,x]^{\mathfrak{st}}\, s))
\end{aligned}$$

where the state threading is made explicit before $\mathfrak{er}$-reification.

It is worth reiterating that these considerations typically only arise when we want to assign well-defined meanings to *all* terms in a language without an effect-typing system, but with effect-delimiting operations such as *handle* or *withst*. If we are only using reflection and reification as a more concise notation for programs written with explicit effect-passing, such conflicts have by definition already been resolved in the original program.

3 Implementing layered effects

The previous section describes a framework for adding programmer-defined effects to an ML-like language. However, a direct implementation of this semantics would be problematic for several reasons:

- In the context of a full programming language, it requires us to effectively write a full language processor, including parser, type checker, module system, standard library, etc.; or, at the very least, perform major surgery on an existing implementation.

- Each level of translation imposes a potentially substantial execution-time overhead – especially for programs which only rarely use any particular effect, but must still provide the infrastructure for connecting such scattered uses.

- Perhaps most significantly, the semantics is given by induction on explicitly sequenced, fully effect-annotated terms. Although this verbosity is essentially equivalent to writing a program in explicit monadic style, it imposes an uncomfortably heavy burden on the programmer accustomed to ML's anonymous (and, given a guarantee of left-to-right call-by-value evaluation, often completely implicit) sequencing of computations.

In this section we show how all of these problems can be solved. In doing so, we demonstrate that the monad equations and the layering conditions are not merely arbitrary category-theoretic overhead, or a mere convenience for manual or automated reasoning about programs, but are in fact the key to a vastly more efficient implementation of the specification.

The result falls in two parts: (1) that each individual monadic translation can be uniformly simulated by a layer of continuation-passing, and (2) that any tower of continuation-passing layers can be simulated by a single notion of effect comprising Scheme-style first-class continuations and mutable state.

We phrase these simulations in terms of *realizations* of one language in another, where a realization of an effect-language $L' \supseteq L$ replaces (not necessarily injectively) every new effect of L' with an effect of L, and every new constant of L' by a term of L, such that the meanings of complete L'-programs are preserved.

3.1 Relating monadic effects to continuation-passing

It is a fairly simple observation that continuation-passing can simulate monadic style [PW93], but actually showing that the translations are equivalent is surprisingly complicated.

This was sketched in [Fil94] for a single effect in an otherwise completely pure language; unfortunately the retraction-based approach [MW85] used there does not seem to generalize well to more general settings, such as unrestricted recursion. In [Fil96], the proof was redone with admissible relations in the style of [Rey74], and extended to a base language with arbitrary pre-existing effects; and that approach does generalize to the multi-effect language of the previous section, as sketched in the following.

One major complication is that to obtain a proper simulation, we must pick the answer type for the CPS transform "large enough". In particular, this means that we cannot use a simple base type, but will need a recursively defined type of answers in the implementation language L_μ (the specification language $L \subset L_\mu$ remains simply typed).

Let $\bar{e}$ be some effect of L, $\mathfrak{k}$ (gothic "k") a new effect name, and ω some fixed type with effects from L extended with $\mathfrak{k}$. We then define L^{K} as L extended with $\bar{e} \prec \mathfrak{k}$, and two new constants shift and reset, with types

$$\begin{aligned} \mathsf{shift}_\alpha &: ((\alpha \xrightarrow{\bar{e}} \omega) \xrightarrow{\mathfrak{k}} \omega) \xrightarrow{\mathfrak{k}} \alpha \\ \mathsf{reset} &: (1 \xrightarrow{\mathfrak{k}} \omega) \xrightarrow{\bar{e}} \omega \end{aligned}$$

We write $\mathcal{S}_\alpha k.\, E$ as syntactic sugar for $\mathsf{shift}_\alpha\,(\lambda k.\, E)$, and $\#E$ for $\mathsf{reset}\,(\lambda().\, E)$. (Conversely, we have $\mathsf{shift}_\alpha = \lambda h.\, \mathcal{S}_\alpha k.\, h k$ and $\mathsf{reset} = \lambda t.\, \#(t\,())$.)

Informally, $\mathcal{S}k.\, E$ evaluates E with k bound to a functional representation of the current evaluation context, but with E itself evaluated in an empty context. Conversely, $\#E$ evaluates E in an empty evaluation context, and returns the result to the current context. For example, writing out all the sequencing explicitly,

$$\begin{aligned} &\mathbf{let}^{\bar{e}}\ r \Leftarrow \#(\mathbf{let}^{\mathfrak{k}}\ n \Leftarrow (\mathcal{S}k.\, \mathbf{let}^{\bar{e}}\ x \Leftarrow k\,3\ \mathbf{in}\ k\,x)\ \mathbf{in}\ \mathbf{val}^{\mathfrak{k}}\ 2 \times n) \\ &\mathbf{in}\ \mathbf{val}^{\bar{e}}\ 1 + r \\ &\quad = \mathbf{val}^{\bar{e}}\ 1 + 2 \times (2 \times 3) = \mathbf{val}^{\bar{e}}\ 13 \end{aligned}$$

Much like for general effects, we give the formal semantics of L^{K} by a continuation-passing translation into L_μ. For any type χ, we define first a parameterized translation of the effect $\mathfrak{k}$ as an effect-type constructor:

$$|\mathfrak{k}|_{K(\chi)}\alpha = ((\alpha \xrightarrow{\bar{e}} \chi) \xrightarrow{\bar{e}} \chi)/\mathfrak{n}$$

Let $\hat{\omega} = \mu\chi.\, |\omega|_{K(\chi)}$, with isomorphisms

$$\phi : |\omega|_{K(\hat{\omega})} \to \hat{\omega} \quad \text{and} \quad \psi : \hat{\omega} \to |\omega|_{K(\hat{\omega})}$$

(Note that if ω does not contain any $\mathfrak{k}$-effects, so that $|\omega|_{K(\chi)}$ does not actually depend on χ, we can simply take $\hat{\omega} = \omega$ and the isomorphisms as identities.) We can then define the formal monad K of $\hat{\omega}$-continuations by

$$\begin{aligned} T^{\mathfrak{k}}\alpha = K\alpha &= (\alpha \xrightarrow{\bar{e}} \hat{\omega}) \xrightarrow{\bar{e}} \hat{\omega} \\ \mathit{glue}^{\mathfrak{k}}\, t &= \lambda k.\, \mathbf{let}^{\bar{e}}\ r \Leftarrow t\,()\ \mathbf{in}\ r\,k \\ \mathit{unit}^{\mathfrak{k}}\, a &= \lambda k.\, k\,a \\ \mathit{bind}^{\mathfrak{k}}\,(t, f) &= \lambda k.\, t\,(\lambda a.\, f\,a\,k) \end{aligned}$$

It is easy to check that this determines an actual monad $\mathcal{K} = \mathcal{L}[\![\mathsf{K}]\!]^{\mathrm{m}}$ for any interpretation of $\bar{e}$.

We use the formal monad to define the syntactic translation $|{-}|_{\mathsf{K}}$, extended with the clauses for the specialized control operators:

$$\begin{aligned} |\mathsf{shift}_\tau| = {} & \lambda h^{(|\tau| \xrightarrow{\bar{e}} |\omega|) \to (|\omega| \xrightarrow{\bar{e}} \hat{\omega}) \xrightarrow{\bar{e}} \hat{\omega}}.\, \lambda k^{|\tau| \xrightarrow{\bar{e}} \hat{\omega}}. \\ & h\,(\lambda x^{|\tau|}.\, \mathbf{let}^{\bar{e}}\ a \Leftarrow k\,x\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\ \psi\,a) \\ & \quad (\lambda r^{|\omega|}.\, \mathbf{val}^{\bar{e}}\ \phi\,r) \\ |\mathsf{reset}| = {} & \lambda t^{1 \to (|\omega| \xrightarrow{\bar{e}} \hat{\omega}) \xrightarrow{\bar{e}} \hat{\omega}}. \\ & \mathbf{let}^{\bar{e}}\ a \Leftarrow t\,()\,(\lambda r^{|\omega|}.\, \mathbf{val}^{\bar{e}}\ \phi\,r)\ \mathbf{in}\ \mathbf{val}^{\bar{e}}\ \psi\,a \end{aligned}$$

(We could also have these operators explicitly in terms of $\mu^{\mathfrak{k}}(-)$, $[-]^{\mathfrak{k}}$, ϕ and ψ; the result of the translation would be the same.)

Note that shift and reset are the only operations that actually depend on the choice of $\hat{\omega}$ as the answer type. As usual for a continuation-passing translation, everything else is parametric in the answer type.

Further, let d be a sufficiently large type to embed any L^{T}-type at which we want to reify. (For any particular program, this can always be chosen as a finite sum; and if we only need to reify at outermost level, e.g., for state, it can even be a base type.) More precisely, let $\aleph$ be a (finite or infinite) set of types, with functions

$$\mathit{in}_\tau : \tau \xrightarrow{\mathfrak{n}} d \quad \text{and} \quad \mathit{out}_\tau : d \xrightarrow{\mathfrak{p}} \tau$$

for every τ in $\aleph$, such that $\mathit{out}_\tau\,(\mathit{in}_\tau\, a) = \mathbf{val}^{\mathfrak{p}}\, a$ for any τ-value a.

For the actual simulation, we now take $\omega = Td$, allowing us to define a realization $\Phi^{\mathfrak{t},\mathfrak{k}}_{\mathsf{T}}$ of L^{T} in L^{K} by $\Phi(\mathfrak{t}) = \mathfrak{k}$, and

$$\begin{aligned} &\Phi(\mathsf{reflect}^{\mathfrak{t}}_\alpha) \equiv \\ &\quad \lambda t^{T\alpha}.\, \mathcal{S}_\alpha k^{\alpha \xrightarrow{\bar{e}} Td}.\, \mathbf{val}^{\mathfrak{k}}\ \mathit{bind}_{\alpha,d}\,(t, \lambda a^{\alpha}.\, \mathit{glue}_d\,(\lambda().\, k\,a)) \\ &\quad \Phi(\mathsf{reify}^{\mathfrak{t}}_\alpha) \equiv \\ &\quad \lambda t^{1 \xrightarrow{\mathfrak{k}} \alpha}.\, \mathit{glue}_\alpha\,(\lambda(). \\ &\qquad\qquad \mathbf{let}^{\bar{e}}\ r \Leftarrow \#(\mathbf{let}^{\mathfrak{k}}\ a \Leftarrow t\,()\ \mathbf{in}\ \mathbf{val}^{\mathfrak{k}}\ \mathit{unit}_d\,(\mathit{in}_\alpha a)) \\ &\qquad\qquad \mathbf{in}\ \mathbf{val}^{\bar{e}}\ \mathit{bind}_{d,\alpha}\,(r, \lambda d^{d}.\, \mathbf{let}^{\mathfrak{p}}\ x \Leftarrow \mathit{out}_\alpha\, d\ \mathbf{in}\ \mathit{unit}_\alpha\, x)) \end{aligned}$$

We want to show that for every L^{T}-program $\vdash E : b/\mathfrak{p}$, the two translations give the same result. More precisely, we will show that given a specification semantics $\mathcal{L}_{\mathrm{s}}$ and implementation semantics $\mathcal{L}_{\mathrm{i}}$ of L, such that for complete L-programs, E, $\hat{\mathcal{L}}_{\mathrm{s}}[\![E]\!] = \hat{\mathcal{L}}_{\mathrm{i}}[\![E]\!]$, then also $\hat{\mathcal{L}}_{\mathrm{s}}[\![|E|_{\mathsf{T}}]\!] = \hat{\mathcal{L}}_{\mathrm{i}}[\![|E\{\Phi^{\mathfrak{t},\mathfrak{k}}_{\mathsf{T}}\}|_{\mathsf{K}}]\!]$ for complete L^{T}-programs E. But we cannot show this statement (or any simple variation of it) directly by induction on E: the problem is that at higher types, there is no direct *equational* characterization of the relationship between $|E|_{\mathsf{T}}$ and $|E|_{\mathsf{K}}$. Instead, we use a more general *relational* invariant, that will give us the original equation as a special case.

We say that a relation $R \subseteq A \times A'$ between two CPOs A and A' is *admissible* if it is chain-complete, i.e., if the least upper bounds of pointwise R-related chains are also R-related. We write $\mathrm{ARel}(A, A')$ for the set of all admissible relations between A and A'.

Definition 6 *A* logical relation $\mathcal{R}$ *between* $\mathcal{L}_{\mathrm{s}}$ *and* $\mathcal{L}_{\mathrm{i}}$ *of an effect-language* L *assigns to every base type* b *a relation* $\mathcal{B}^{\mathrm{r}}(b) \in \mathrm{ARel}(\mathcal{B}_{\mathrm{s}}(b), \mathcal{B}_{\mathrm{i}}(b))$ *between their interpretations; and to every effect* e, *a* relational action $\mathcal{E}^{\mathrm{r}}(e)$, *mapping any relation* $R \in \mathrm{ARel}(A, A')$ *to* $\mathcal{E}^{\mathrm{r}}(e)R \in \mathrm{ARel}(T^{e}_{\mathrm{s}} A, T^{e}_{\mathrm{i}} A)$.

Let ϱ *and* ϱ' *map type variables from* $\vec{\alpha}$ *to CPOs, and for each* α *let* $\theta\alpha \in \mathrm{ARel}(\varrho\alpha, \varrho'\alpha)$. *For any type* τ *over* $\vec{\alpha}$, *we then define a relation* $\mathcal{R}[\![\tau]\!]_\theta \in \mathrm{ARel}(\mathcal{L}_{\mathrm{s}}[\![\tau]\!]_\varrho, \mathcal{L}_{\mathrm{i}}[\![\tau]\!]_{\varrho'})$ *by:*

$$\begin{aligned} \mathcal{R}[\![\alpha]\!]_\theta &= \theta\alpha \\ \mathcal{R}[\![b]\!]_\theta &= \mathcal{B}^{\mathrm{r}}(b) \\ \mathcal{R}[\![\tau_1 \times \tau_2]\!]_\theta &= \{((a_1, a_2), (a_1', a_2')) \mid \\ &\qquad (a_1, a_1') \in \mathcal{R}[\![\tau_1]\!]_\theta \wedge (a_2, a_2') \in \mathcal{R}[\![\tau_2]\!]_\theta\} \\ \mathcal{R}[\![\tau_1 + \tau_2]\!]_\theta &= \{((1, a_1), (1, a_1')) \mid (a_1, a_1') \in \mathcal{R}[\![\tau_1]\!]_\theta\} \cup \\ &\quad\ \{((2, a_2), (2, a_2')) \mid (a_2, a_2') \in \mathcal{R}[\![\tau_2]\!]_\theta\} \\ \mathcal{R}[\![\tau_1 \xrightarrow{e} \tau_2]\!]_\theta &= \{(f, f') \mid \forall (a, a') \in \mathcal{R}[\![\tau_1]\!]_\theta. \\ &\qquad\qquad (f\,a, f'\,a') \in \mathcal{E}^{\mathrm{r}}(e)(\mathcal{R}[\![\tau_2]\!]_\theta)\} \end{aligned}$$

It is easy to check that these relations are all admissible.

Moreover, the relations must respect interpretations of effects and constants: for every e of L we require that,

1. *If* $\bar{e} \prec e$ *and* $(t, t') \in \mathcal{E}^{\mathrm{r}}(\bar{e})(\mathcal{E}^{\mathrm{r}}(e)R)$ *then* $(\zeta_{\mathrm{s}}^{e}{}_{A}\, t, \zeta_{\mathrm{i}}^{e}{}_{A'}\, t') \in \mathcal{E}^{\mathrm{r}}(e)R$.
2. *If* $(a, a') \in R$ *then* $(\eta_{\mathrm{s}}^{e}{}_{A}\, a, \eta_{\mathrm{i}}^{e}{}_{A'}\, a') \in \mathcal{E}^{\mathrm{r}}(e)R$.
3. *If* $(t, t') \in \mathcal{E}^{\mathrm{r}}(e)R_1$ *and for every* $(a, a') \in R_1$, $(f\, a, f'\, a') \in \mathcal{E}^{\mathrm{r}}(e)R_2$ *then* $(t \star_{\mathrm{s}}^{e} f, t' \star_{\mathrm{i}}^{e} f') \in \mathcal{E}^{\mathrm{r}}(e)R_2$.

And for $c : \forall\vec{\alpha}.\tau$ *in* Σ_L*, and any relation-environment* θ *with* $\theta\alpha_i \in \mathrm{ARel}(A_i, A_i')$ *for each* α_i*,*

$$(\mathcal{C}_{\mathrm{s}}(c)_{\vec{A}}, \mathcal{C}_{\mathrm{i}}(c)_{\vec{A}'}) \in \mathcal{R}\llbracket\tau\rrbracket_{\theta}$$

Finally, we require that related meanings of complete programs are interpreted as the same observation, i.e., if $(p, p') \in \mathcal{E}^{\mathrm{r}}(e)\mathcal{B}^{\mathrm{r}}(b)$ *then* $\xi_{\mathrm{s}}{}_{b}^{e}\, p = \xi_{\mathrm{i}}{}_{b}^{e}\, p'$.

For our purposes it suffices to take the relation at base types to be simple equality, i.e.,

$$\mathcal{B}^{\mathrm{r}}(b) = \{(n, n) \mid n \in \mathcal{B}(b)\}$$

When $\mathcal{L}_{\mathrm{i}}$ also interprets $\mathfrak{p}$ as the lifting monad, we can define the relational action of $\mathfrak{p}$ by:

$$\mathcal{E}^{\mathrm{r}}(\mathfrak{p})R = \{(\bot, \bot)\} \cup \{([a], [a']) \mid (a, a') \in R\}$$

(Note that then $\mathcal{E}^{\mathrm{r}}(\mathfrak{p})\mathcal{B}^{\mathrm{r}}(b)$ becomes simply the equality relation on $\mathcal{B}(b)_{\bot}$.) In other cases, we must explicitly construct a suitable action, as in the proof of Theorem 11 later.

We can now state the usual logical-relations lemma, straightforwardly extended to effects and polymorphic terms:

Lemma 7 (Logical relations lemma) *Let there be given a logical relation between* $\mathcal{L}_{\mathrm{s}}$ *and* $\mathcal{L}_{\mathrm{i}}$*. Let* $\vec{\alpha}$ *be a list of type variables,* ϱ *and* ϱ' *type environments, and* θ *a relation environment, such that for each* α, $\theta\alpha \in \mathrm{ARel}(\varrho\alpha, \varrho'\alpha)$*. Let* Γ *be a type assignment over* $\vec{\alpha}$*, and let* ρ *and* ρ' *be environments such that for every* $x \in \mathrm{dom}\,\Gamma$, $\rho x \in \mathcal{L}_{\mathrm{s}}\llbracket\tau\rrbracket_{\varrho}$, $\rho' x \in \mathcal{L}_{\mathrm{i}}\llbracket\tau\rrbracket_{\varrho'}$*, and* $(\rho x, \rho' x) \in \mathcal{R}\llbracket\Gamma(x)\rrbracket_{\theta}$*. Then for every well-typed term* $\Gamma \vdash_{\vec{\alpha}} E : \tau / e$*,*

$$(\mathcal{L}_{\mathrm{s}}\llbracket E\rrbracket_{\varrho}\,\rho, \mathcal{L}_{\mathrm{i}}\llbracket E\rrbracket_{\varrho'}\,\rho') \in \mathcal{E}^{\mathrm{r}}(e)(\mathcal{R}\llbracket\tau\rrbracket_{\theta})$$

Proof. Straightforward induction on E. ■

Note that the standard interpretations of fix^{e} are always related. This is easily seen by fixed-point induction, using the fact that $\mathcal{R}\llbracket\tau_1 \xrightarrow{e} \tau_2\rrbracket_{\theta}$ is admissible and contains $(\bot, \bot)$ when $\mathfrak{p} \preceq e$ (follows from Definition 6(1–3)).

Much as we previously interpreted a type constructor as a CPO constructor, we can define for any type constructor $\mathcal{T}$ its relational action, $\mathcal{T}^{\mathrm{r}} : \mathrm{ARel}(A, A') \to \mathrm{ARel}(\mathcal{L}_{\mathrm{s}}\llbracket\mathcal{T}\alpha\rrbracket_{[\alpha\mapsto A]}, \mathcal{L}_{\mathrm{i}}\llbracket\mathcal{T}\alpha\rrbracket_{[\alpha\mapsto A]})$ by $\mathcal{T}^{\mathrm{r}}R = \mathcal{R}\llbracket\mathcal{T}\alpha\rrbracket_{[\alpha\mapsto R]}$.

Suppose now we have a logical relation between semantics $\mathcal{L}_{\mathrm{s}}$ and $\mathcal{L}_{\mathrm{i}}$ of language L, and we want to extend it to $L^{\mathcal{T}}$, where we take $\mathcal{E}_{\mathrm{s}}(\mathfrak{t}) = \mathcal{T}$ and $\mathcal{E}_{\mathrm{i}}(\mathfrak{t}) = \mathcal{K}$ in the extended semantics. We then need to define $\mathcal{E}^{\mathrm{r}}(\mathfrak{t})$. Intuitively, we are representing a T-computation t by the K-computation $u = \lambda k.\, k^{*} t$. So we want something like

$$(t, u) \in \mathcal{E}^{\mathrm{r}}(\mathfrak{t})R \iff (\lambda k.\, t \star k, u) \in K^{\mathrm{r}}R$$

where K^{r} is the natural choice: two K-computations u and u' are related by $K^{\mathrm{r}}R$ if for all continuations k and k' mapping R-related values to $T^{\mathrm{r}}O$-related results, $u\, k$ is $T^{\mathrm{r}}O$-related to $u'\, k'$, for some suitable relation O on u. But how to pick O? A suitable answer is to take O as the *intersection* (always admissible) of all the relational interpretations we will actually need. Formally:

Lemma 8 *The relational action* $\mathcal{E}^{\mathrm{r}}(\mathfrak{t})$ *of* $\mathfrak{t}$ *defined by*

$$\begin{aligned}\mathcal{E}^{\mathrm{r}}(\mathfrak{t})R = \{(t, u) \in TA \times KA' \mid\ & \\ & \forall\tau \in \aleph.\, \forall O \in \mathrm{ARel}(\mathcal{L}_{\mathrm{s}}\llbracket\tau\rrbracket, \mathcal{L}_{\mathrm{i}}\llbracket d\rrbracket). \\ & \quad \forall k : A \to \mathcal{L}_{\mathrm{s}}\llbracket\mathcal{T}\tau\rrbracket, k' : A' \to \mathcal{L}_{\mathrm{i}}\llbracket\mathcal{T} d\rrbracket. \\ & \qquad (\forall(a, a') \in R.\, (k\, a, \zeta\,(\psi\,(k'\, a'))) \in \mathcal{T}^{\mathrm{r}}O) \\ & \qquad\quad \Rightarrow (t \star k, \zeta\,(\psi\,(u\, k'))) \in \mathcal{T}^{\mathrm{r}}O\}\end{aligned}$$

extends the logical relation between semantics $\mathcal{L}_{\mathrm{s}}$ *and* $\mathcal{L}_{\mathrm{i}}$ *of* L *to one between* $\mathcal{L}_{\mathrm{s}}^{\mathcal{T}}$ *and* $\mathcal{L}_{\mathrm{i}}^{\mathcal{K}} \circ \Phi_{\mathbf{T}}^{\mathfrak{t},\mathfrak{k}}$ *of* $L^{\mathcal{T}}$*.*

Proof. Relatively direct verification for both the new effect $\mathfrak{t}$ and the new constants $\mathsf{reflect}^{\mathfrak{t}}$ and $\mathsf{reify}^{\mathfrak{t}}$ [Fil96]. For the latter two, we use the fact that (by Lemma 7) the interpretations of the term components of $\mathbf{T}$ are related, even if $\mathbf{T}$ does not actually define a monad in $\mathcal{L}_{\mathrm{i}}$. ■

Theorem 9 (T-K simulation) *Let* $\mathcal{L}_{\mathrm{s}}$ *and* $\mathcal{L}_{\mathrm{i}}$ *be related semantics of* L*, and let* E *be a complete* $L^{\mathcal{T}}$*-program, i.e.,* $\vdash E : b \,/\, \mathfrak{p}$*. Then* $\hat{\mathcal{L}}_{\mathrm{s}}\llbracket|E|_{\mathbf{T}}\rrbracket = \hat{\mathcal{L}}_{\mathrm{i}}\llbracket|E\{\Phi_{\mathbf{T}}^{\mathfrak{t},\mathfrak{k}}\}|_{\mathbf{K}}\rrbracket$*.*

Proof. By Lemmas 7 and 8, $(\mathcal{L}_{\mathrm{s}}\llbracket|E|_{\mathbf{T}}\rrbracket\emptyset, \mathcal{L}_{\mathrm{i}}\llbracket|E\{\Phi_{\mathbf{T}}^{\mathfrak{t},\mathfrak{k}}\}|_{\mathbf{K}}\rrbracket\emptyset) \in \mathcal{E}^{\mathrm{r}}(\mathfrak{p})\mathcal{B}^{\mathrm{r}}(b)$, from which the result follows by the assumption on ξ_{s} and ξ_{i}. ■

3.2 Relating continuation-passing to primitive effects

Let us now consider an implementation language essentially like Scheme or SML/NJ, i.e., containing first-class continuations and state as primitive effects. To keep things simple, we consider all state to be allocated before program execution proper begins.

That is, for a state-assignment Δ mapping ref-cell names v to types, the language $L_{\mathrm{cs}_{\Delta}}$ contains the basic syntax from Section 2 (products, sums, functions), a single effect $\mathfrak{cs}$, and constants

$$\begin{aligned}\mathsf{escape}_{\alpha} &: ((\alpha \xrightarrow{\mathfrak{cs}} 0) \xrightarrow{\mathfrak{cs}} 0) \xrightarrow{\mathfrak{cs}} \alpha \\ \mathsf{get}^{v} &: 1 \xrightarrow{\mathfrak{cs}} \Delta(v) \\ \mathsf{set}^{v} &: \Delta(v) \xrightarrow{\mathfrak{cs}} 1\end{aligned}$$

escape is a simple variation on *call/cc*, interdefinable with both Scheme's and SML/NJ's operators. We usually write $!v$ for $\mathsf{get}^{v}()$ and $v := E$ for $\mathsf{set}^{v}\, E$.

The formal semantics of this language is also Scheme-like [KCR98]: we interpret $\mathfrak{cs}$ as the continuation-state monad,

$$\begin{aligned}T^{\mathfrak{cs}} A &= A \to S_{\Delta} \to \mathbf{P}_{\bot} \\ \eta^{\mathfrak{cs}}\, a &= \lambda\kappa.\, \lambda\sigma.\, \kappa\, a\, \sigma \\ t \star^{\mathfrak{cs}} f &= \lambda\kappa.\, \lambda\sigma.\, t\,(\lambda a.\, \lambda\sigma'.\, f\, a\, \kappa\, \sigma')\,\sigma\end{aligned}$$

where $S_{\Delta} = \prod_{v \in \mathrm{dom}\,\Delta} \mathcal{L}\llbracket\Delta(v)\rrbracket$. (Note that, since the types in Δ may themselves contain $\mathfrak{cs}$-annotations, S_{Δ} is a recursively defined CPO. We elide the associated isomorphisms for conciseness.)

Then the interpretations of the operations are given by

$$\begin{aligned}
\mathcal{C}(\mathsf{escape})_A &= \lambda f.\lambda\kappa.\lambda\sigma.f\,(\lambda a^{A}.\lambda\kappa'.\lambda\sigma'.\kappa\,a\,\sigma')\,(\lambda z^{0}.\mathcal{V}z)\,\sigma \\
\mathcal{C}(\mathsf{get}^{v}) &= \lambda().\lambda\kappa.\lambda\sigma.\kappa\,(\sigma\,v)\,\sigma \\
\mathcal{C}(\mathsf{set}^{v}) &= \lambda a.\lambda\kappa.\lambda\sigma.\kappa\,()\,(\sigma[v\mapsto a])
\end{aligned}$$

We also define $\xi^{\mathfrak{cs}}_{b}\,t = t\,(\lambda a.\lambda\sigma'.[print_b\,a])\,\sigma_0$, where the initial store σ_0 is some fixed element of S_Δ.

We add a new effect $\mathfrak{k}$ over $\mathfrak{cs}$ and shift/reset to this language as described in Section 3.1, to obtain $L^{K}_{\mathrm{cs}_\Delta}$. Much as before, we now view the continuation-passing transform as the *specification* of the new language, and define a different *implementation.*

To see how to obtain such an implementation, consider the interpretation of a $\mathfrak{k}$-effect:

$$\begin{aligned}
&\mathcal{L}[\![|\tau/\mathfrak{k}|_K]\!] = \mathcal{L}[\![(|\tau|_K \xrightarrow{\mathfrak{cs}} \hat\omega) \xrightarrow{\mathfrak{cs}} \hat\omega]\!] \\
&\quad = \big(\mathcal{L}[\![|\tau|]\!] \to (\mathcal{L}[\![\hat\omega]\!] \to S_\Delta \to \mathbf{P}_\perp) \to S_\Delta \to \mathbf{P}_\perp\big) \\
&\qquad\quad \to (\mathcal{L}[\![\hat\omega]\!] \to S_\Delta \to \mathbf{P}_\perp) \to S_\Delta \to \mathbf{P}_\perp \\
&\quad = \big(\mathcal{L}[\![|\tau|]\!] \to (\mathcal{L}[\![\hat\omega]\!] \to S_\Delta \to \mathbf{P}_\perp) \times S_\Delta \to \mathbf{P}_\perp\big) \\
&\qquad\quad \to (\mathcal{L}[\![\hat\omega]\!] \to S_\Delta \to \mathbf{P}_\perp) \times S_\Delta \to \mathbf{P}_\perp \\
&\quad = \big(\mathcal{L}[\![|\tau|]\!] \to S' \to \mathbf{P}_\perp\big) \to S' \to \mathbf{P}_\perp
\end{aligned}$$

That is, the composite semantics is *also* a continuation-state semantics, only with a larger state. This suggests that we can implement $L^{K}_{\mathrm{cs}_\Delta}$ by a simple embedding into $L_{\mathrm{cs}_{\Delta'}}$, where Δ' extends Δ with a new cell to hold the *metacontinuation* [DF90, Fil96] of the original computation.

Formally, we define the realization $\Phi^{\mathfrak{k},\mathfrak{cs}}_{v:\neg\omega}$ by $\Phi(\mathfrak{k}) = \Phi(\mathfrak{cs}) = \mathfrak{cs}$ (thus conflating two previously distinct layers of effects), and

$$\begin{aligned}
&\Phi(\mathsf{reset}) = \\
&\quad \lambda t^{1 \xrightarrow{\mathfrak{cs}} \omega}.\,\mathsf{escape}_\omega\,(\lambda k^{\omega \xrightarrow{\mathfrak{cs}} 0}. \\
&\qquad \mathbf{let}^{\mathfrak{cs}}\ m \Leftarrow {!v} \\
&\qquad \mathbf{in}\ (v := (\lambda a^{\omega}.v := m; k\,a); \\
&\qquad\quad \mathbf{let}^{\mathfrak{cs}}\ r \Leftarrow t\,()\ \mathbf{in}\ \mathbf{let}^{\mathfrak{cs}}\ m \Leftarrow {!v}\ \mathbf{in}\ m\,r)) \\
&\Phi(\mathsf{shift}_\alpha) = \\
&\quad \lambda h^{(\alpha \xrightarrow{\mathfrak{cs}} \omega) \xrightarrow{\mathfrak{cs}} \omega}.\,\mathsf{escape}_\alpha\,(\lambda k^{\alpha \xrightarrow{\mathfrak{cs}} 0}. \\
&\qquad \mathbf{let}^{\mathfrak{cs}}\ r \Leftarrow h\,(\lambda v^{\alpha}.\,\Phi(\mathsf{reset})\,(\lambda().\mathbf{let}^{\mathfrak{cs}}\ z \Leftarrow k\,v\ \mathbf{in}\ \mathcal{V}\,z)) \\
&\qquad \mathbf{in}\ \mathbf{let}^{\mathfrak{cs}}\ m \Leftarrow {!v}\ \mathbf{in}\ m\,r) \\
&\Phi(\mathsf{escape}_\alpha) = \\
&\quad \lambda h^{(\alpha \xrightarrow{\mathfrak{cs}} 0) \xrightarrow{\mathfrak{cs}} 0}.\,\mathsf{escape}_\alpha\,(\lambda k^{\alpha \xrightarrow{\mathfrak{cs}} 0}. \\
&\qquad \mathbf{let}^{\mathfrak{cs}}\ m \Leftarrow {!v}\ \mathbf{in}\ (v := err; h\,(\lambda\alpha^{a}.(v := m; k\,a))))
\end{aligned}$$

where $(E_1; E_2) \equiv \mathbf{let}^{\mathfrak{cs}}\ () \Leftarrow E_1\ \mathbf{in}\ E_2$, v is a new cell with $\Delta'(v) = \omega \xrightarrow{\mathfrak{cs}} 0$, and $err : \omega \xrightarrow{\mathfrak{cs}} 0$ is an error continuation that will never actually be invoked in an effect-type-correct program.

The realization ensures that the newly exported escape respects the meta-continuation used by shift and reset. [1] But the more significant aspect of the construction is that it eliminates an entire layer of effects (technically, it conflates two layers of state-passing into a single layer with a larger state), making $\mathfrak{k}$- vs. $\mathfrak{cs}$-annotation on **val**s and **let**s unnecessary. Formally, we have the following result:

Theorem 10 (K-CS simulation) *Let E be a complete program of $L^{K}_{\mathrm{cs}_\Delta}$, $\vdash E : b\ /\ \mathfrak{cs}$. Then*

$$\hat{\mathcal{L}}_{\mathrm{cs}_\Delta}[\![|E|_K]\!] = \hat{\mathcal{L}}_{\mathrm{cs}_{\Delta'}}[\![E\{\Phi^{\mathfrak{k},\mathfrak{cs}}_{v:\neg\omega}\}]\!]$$

where $\Delta' = (\Delta, v{:}\omega \xrightarrow{\mathfrak{cs}} 0)$ for $v \notin \mathrm{dom}\,\Delta$.

Proof. (Sketched.) The proof is based on a logical relation between semantics $\mathcal{L}^{K}_{cs_\Delta}$ and $\mathcal{L}_{cs'_\Delta} \circ \Phi$ of $L^{K}_{cs_\Delta}$. For this, we define the relational actions of two effects in the source language,

$$\begin{aligned}
\mathcal{E}^{\mathrm{r}}(\mathfrak{k})R &= \{(t,t') \mid \forall k,\kappa'.\,(\forall(a,a') \in R.\,k\,a \asymp \kappa'\,a') \\
&\qquad\qquad \Rightarrow t\,k \asymp t'\,\kappa'\} \\
\mathcal{E}^{\mathrm{r}}(\mathfrak{cs})R &= \{(t,t') \mid \forall k,\kappa'.\,(\forall(a,a') \in R.\,k\,a \asymp \kappa'\,a') \\
&\qquad\qquad \Rightarrow \lambda\kappa.\,t\,(\lambda a.\,k\,a\,\kappa) \asymp t'\,\kappa'\}
\end{aligned}$$

using auxiliary relations on intermediate answers, $(\asymp) \in \mathrm{ARel}((\mathcal{L}[\![\omega]\!] \to S_\Delta \to \mathbf{P}_\perp) \to S_\Delta \to \mathbf{P}_\perp, S_{\Delta'} \to \mathbf{P}_\perp)$:

$$u \asymp v \iff \forall\kappa,\sigma,\sigma'.\,(\kappa,\sigma) \lhd \sigma' \Rightarrow u\,\kappa\,\sigma = v\,\sigma'$$

on metacontinuation-state pairs, $(\lhd) \in \mathrm{ARel}((\mathcal{L}[\![\omega]\!] \to S_\Delta \to \mathbf{P}_\perp) \times S_\Delta, S_{\Delta'})$:

$$\begin{aligned}
&(\kappa,\sigma) \lhd \sigma' \iff \\
&\quad (\sigma,\sigma') \in \mathcal{R}[\![\Delta]\!] \wedge \\
&\quad \forall(r,r') \in \mathcal{R}[\![\omega]\!], (\sigma_1,\sigma_1') \in \mathcal{R}[\![\Delta]\!], \kappa_0.\,\kappa\,r\,\sigma_1' = \sigma'\,v\,r'\,\kappa_0\sigma_1'
\end{aligned}$$

and on state, $\mathcal{R}[\![\Delta]\!] \in \mathrm{ARel}(S_\Delta, S_{\Delta'},)$:

$$\mathcal{R}[\![\Delta]\!] = \{(\sigma,\sigma') \mid \forall v \in \Delta.\,(\sigma\,v, \sigma'\,v) \in \mathcal{R}[\![\Delta(v)]\!]_\emptyset\}$$

Note that these relations are mutually recursively defined. Thus, their existence is not automatic, but can be established fairly easily using Pitts's techniques [Pit96, Fil96]. We can then check directly that the interpretations of all constants in the two semantics are related by the corresponding relations, so the result follows by Lemma 7. ■

Putting all the pieces together, we finally obtain our main result:

Theorem 11 ($\mathbf{T}^*$-CS* simulation) *For any formal monad T, define the composite realization of L^{T} in L_{cs} by $\Phi^{\mathfrak{t},\mathfrak{cs}}_{\mathsf{T}} = \Phi^{\mathfrak{k},\mathfrak{cs}}_{v_{\mathfrak{t}}:\neg Td} \circ \Phi^{\mathfrak{t},\mathfrak{k}}_{\mathsf{T}}$. Let L_0 be the basic language with no effects other than $\mathfrak{n}$ and $\mathfrak{p}$, with the standard semantics $\mathcal{L}_0$, and let $\Phi^{\mathfrak{p},\mathfrak{cs}}$ be the realization $\Phi(\mathfrak{p}) = \mathfrak{cs}$. Let $\mathsf{T}_1, \ldots \mathsf{T}_n$ denote a sequence of monads, each layered over the previous one (and T_1 over $\mathfrak{p}$). Then there exists a store-typing Δ, such that for any complete $L_0^{T_1,\ldots,T_n}$-program $\vdash E : b/\mathfrak{p}$,*

$$\hat{\mathcal{L}}_0[\![|\cdots|E|_{\mathsf{T}_n}\cdots|_{\mathsf{T}_1}]\!] = \hat{\mathcal{L}}_{\mathrm{cs}_\Delta}[\![E\{\Phi^{\mathfrak{p},\mathfrak{cs}}\}\{\Phi^{\mathfrak{t},\mathfrak{cs}}_{\mathsf{T}_1}\}\cdots\{\Phi^{\mathfrak{t},\mathfrak{cs}}_{\mathsf{T}_n}\}]\!]$$

Proof. For the base case, we need to relate the standard lifting semantics of $\mathfrak{p}$ to the continuation-based one: when $R \in \mathrm{Rel}(A, A')$, we take

$$\begin{aligned}
\mathcal{E}^{\mathrm{r}}(\mathfrak{p})R = \{(t,t') \in A_\perp \times ((A' \to S_\Delta \to \mathbf{P}_\perp) \to S_\Delta \to \mathbf{P}_\perp) \mid \\
\forall\kappa,\kappa',\sigma_0.\,(\forall(a,a') \in R,\sigma.\,\kappa a = \kappa' a'\sigma) \\
\Rightarrow t \star^{\mathfrak{p}} \kappa = t'\,\kappa'\,\sigma_0\}
\end{aligned}$$

The general theorem then follows by induction on n, using Lemma 5 and Theorems 9 and 10 in each step. ■

Writing Φ for the composite realization, we also note that for all effects $e_1 \preceq e_2$, $\mathcal{L}_{cs}[\![(\mathbf{let}^{e_1 \preceq e_2}\ x \Leftarrow E_1\ \mathbf{in}\ E_2)\{\Phi\}]\!] = \mathcal{L}_{cs}[\![\mathbf{let}^{\mathfrak{cs}}\ x \Leftarrow E_1\{\Phi\}\ \mathbf{in}\ E_2\{\Phi\}]\!]$, and thus for all coercions $\mathcal{L}_{cs}[\![\chi(E)\{\Phi\}]\!] = \mathcal{L}_{cs}[\![E]\!]$. That is, the effect-annotations do not actually matter for the purpose of program evaluation, and in particular, we can write our source programs in ML's implicit-sequencing syntax, with the standard elaboration into single-effect monadic normal forms.

---

[1] This redefinition can also be seen as a more principled justification for the practice of redefining the `call/cc` made available to the programmer in order to accommodate an implementation of `dynamic-wind` in Scheme [Ree92, KCR98], and for the implicit adaptation of the `callcc/throw` primitives in SML/NJ to also save and restore exception handlers [BCL+98].

```
abstype void = VOID of void
with fun coerce (VOID v) = coerce v
end;

signature ESCAPE =
sig
   val escape : (('a -> void) -> void) -> 'a
end;

structure Escape0 : ESCAPE =
struct
   fun escape h =
      SMLofNJ.Cont.callcc (fn k =>
         coerce (h (fn x => SMLofNJ.Cont.throw k x)))
end;

signature CTRL =
sig
   type ans
   val shift : (('a -> ans) -> ans) -> 'a
   val reset : (unit -> ans) -> ans
   include ESCAPE
end;

functor RepCPS (type ans
                structure E : ESCAPE) :
   CTRL where type ans = ans  =
struct
   type ans = ans
   fun initmk a = raise Fail "Unexpected control effect"
   val mk = ref (initmk : ans -> void)

   fun abort v = !mk v
   fun reset t =
      E.escape (fn k =>
         let val m = !mk
         in mk := (fn a => (mk := m; k a));
            abort (t ())
         end)
   fun shift h =
      E.escape (fn k =>
         abort (h (fn v =>
                      reset (fn () => coerce (k v)))))

   fun escape h =
      E.escape (fn k =>
        let val m = !mk
        in mk := initmk; h (fn a => (mk := m; k a)) end)
end;
```

Figure 2: Representing continuation-passing with escapes and state

3.3 Representation in SML/NJ

As suggested by the development, the construction applies directly to a language with first-class continuations and state, such as Scheme or SML/NJ. We show it here for the latter (in SML'97 syntax [MTHM97]), using parameterized modules to represent syntactic realizations.

Figure 2 is a straightforward encoding of the control-operator construction from Section 3.2. Figure 3 shows a simple implementation of the universal type required for the answer-embedding in Lemma 8; an alternative implementation in terms of SML's extensible datatype `exn` of exception names is also possible. Finally, Figure 4 shows the implementation of monadic proto-operations using control operators from Section 3.1. (The monad component `show` and the corresponding operation `run` are not formally part of the construction, but are useful for visualization of the effect layering.)

```
signature DYNAMIC =
sig
   exception Dynamic
   type dyn
   val newdyn : unit -> ('a -> dyn) * (dyn -> 'a)
end;

structure Dynamic :> DYNAMIC =
struct
   exception Dynamic
   datatype dyn = DYN of unit -> unit
   fun newdyn () =
      let val r = ref NONE
      in (fn a => DYN (fn () => r := SOME a),
          fn (DYN d) =>
             (r := NONE; d ();
              case !r of SOME a => a
                       | NONE => raise Dynamic))
      end
end;
```

Figure 3: A universal type, with a state-based implementation

4 Examples and applications

In this section, we show two examples of programming with effects in direct style. The first simply explores further the ordering of exceptions and state. The second, more substantial, shows how we to use layered monads to simulate nondeterministic behavior in a shared-state concurrent program.

4.1 Exceptions and state

Using the definitions of the exception and state monads from Figure 5, we can represent a language with exceptions layered over state, as familiar from ML:

```
structure E = Eff0;

structure E = Represent (structure M = Exceptions
                         structure E = E);
structure Rex = E
structure ExcOps = ExceptionOps(structure R = Rex);

structure E = Represent (structure M = State
                         structure E = E)
structure Rst = E
structure StateOps = StateOps(structure R = Rst);

val t1 = E.run (fn () => (StateOps.set 3; "ok"));
(* val t1 = "<st: 3>ok" : string *)
val t2 = E.run (fn () =>
           (StateOps.set 4; ExcOps.fraise "err"; "ok"));
(* val t2 = "<st: 4><exn: err>" : string *)
val t3 = E.run (fn () =>
           (StateOps.set 5;
            ExcOps.fhandle
             (fn () => (StateOps.set 8;
                        ExcOps.fraise "err"; "ok"))
             (fn x => x ^ ", " ^
                Int.toString (StateOps.get ()))))
(* val t3 = "<st: 8>err, 8" : string *)
```

Let us now switch the order of the two effect-definition blocks in the prologue, putting the state-block first. Then running the same three examples gives:

```
val t1' = E.run (fn () => (StateOps.set 3; "ok"));
(* val t1' = "<st: 3>ok" : string *)
val t2' = E.run (fn () =>
```

```
signature MONAD =
sig
   type ’a t
   val unit : ’a -> ’a t
   val bind : ’a t * (’a -> ’b t) -> ’b t
   val glue : (unit -> ’a t) -> ’a t
   val show : string t -> string
end;

signature RMONAD =
sig
   structure M : MONAD
   val reflect : ’a M.t -> ’a
   val reify : (unit -> ’a) -> ’a M.t
end;

signature EFFREP =
sig
   include ESCAPE
   val run : (unit -> string) -> string
end;

structure Eff0 : EFFREP =
struct
   open Escape0
   fun run t = t ()
end;

functor Represent (structure M : MONAD
                   structure E : EFFREP) :
   sig include RMONAD  include EFFREP end =
struct
   structure C = RepCPS (type ans = Dynamic.dyn M.t
                         structure E = E)
   structure M = M
   fun reflect m =
      C.shift (fn k =>
         M.bind (m, fn a => M.glue (fn () => k a)))
   fun reify t =
      let val (in_d, out_d) = Dynamic.newdyn ()
      in M.glue (fn () =>
           M.bind (C.reset (fn () =>
                      M.unit (in_d (t ()))),
                   M.unit o out_d))
      end
   val escape = C.escape
   fun run t = M.show (reify (fn () => E.run t))
end;
```

Figure 4: Representing monadic effects with continuation-passing

```
            (StateOps.set 4; ExcOps.fraise "err"; "ok"));
(* val t2’ = "<exn: err>" : string *)
val t3’ = E.run (fn () =>
            (StateOps.set 5;
             ExcOps.fhandle
             (fn () => (StateOps.set 8;
                        ExcOps.fraise "err"; "ok"))
             (fn x => x ^ ", " ^
               Int.toString (StateOps.get ()))))
(* uncaught exception Fail: Unexpected control effect
     raised at: ctrl.sml:29.25-29.57 *)
```

Here, the computation of `t2’` shows that a raised exception simply discards the current state. `t3’` shows what happens when we attempt to execute an effect-ill-typed program: the state-effect in the first argument to `fhandle` is meaningless in this ordering of effects. Consequently, the translation-based specification says nothing about the meaning of the program, and the simulation theorem does not constrain the behavior of the implementation.

```
structure Exceptions (*: MONAD*) =
struct
   datatype ’a res = OK of ’a | EXN of string
   type ’a t = unit -> ’a res
   fun glue t = fn () => t () ()
   fun unit a = fn () => OK a
   fun bind (t, f) =
      fn () => case t () of OK a => f a () | EXN s => EXN s
   fun show t =
      case t () of OK s => s | EXN x => "<exn: " ^ x ^ ">"
end;

functor ExceptionOps (structure R :
                         RMONAD where M = Exceptions) :
   sig
      val fraise : string -> ’a
      val fhandle : (unit -> ’a) -> (string -> ’a) -> ’a
   end =
struct
   open Exceptions
   fun fraise s = R.reflect (fn () => EXN s)
   fun fhandle t h =
      case R.reify t () of OK a => a | EXN s => h s
end;

structure State : MONAD =
struct
   type state = int
   type ’a t = state -> ’a * state
   fun glue t = fn s => t () s
   fun unit a = fn s => (a,s)
   fun bind (t, f) =
      fn s => let val (a,s’) = t s in f a s’ end
   fun show t =
      let val (a,s) = t 0
      in if s = 0 then a
                  else "<st: " ^ Int.toString s ^ ">" ^ a
      end
end;

functor StateOps (structure R : RMONAD where M = State) :
   sig
      val get : unit -> int
      val set : int -> unit
   end =
struct
   fun get () = R.reflect (fn s => (s, s))
   fun set n = R.reflect (fn s => ((), n))
end;

structure ListMonad : MONAD =
struct
   type ’a t = unit -> ’a list
   fun glue t = fn () => t () ()
   fun unit a = fn () => [a]
   fun mapcan f [] = []
     | mapcan f (h::t) = f h @ mapcan f t
   fun bind (t, f) =
      fn () => mapcan (fn a => f a ()) (t ())

   fun disp [] = "<fail>"
     | disp [x] = x
     | disp (h::t) = h ^ " <or> " ^ disp t
   fun show t = disp (t ())
end;
```

Figure 5: Some simple monads and their operations

4.2 Shared-state concurrency

As a larger example, we will consider the monadic approach to modeling concurrency, as sketched in [Mog90], based on the semantic concept of resumptions [Sch86]. (Strictly speaking, this example goes beyond the language outlined in Section 2.1 by using a (positive) recursively defined type in the monad specifications. The relevant theorem extends easily to this case, however.)

The monad of $\bar{e}$-resumptions is given by

$$T\alpha \;=\; \mu\beta.\, 1 \xrightarrow{\bar{e}} (\alpha + \beta)$$

with straightforward unit and extension operations. That is, a resumption-computation of α is a $\bar{e}$-computation that yields either an α-value (representing the final result), or another resumption-computation (representing the remaining computation). The ML representation of the monad and its associated operations can be found in Figure 6.

As long as all resumption-computations suspend periodically (e.g., by calling `yield`()), this setup can directly simulate the parallel-or operation [Plo77], which returns `true` if either of of its arguments evaluates to `true`, `false` if both evaluate to `false`, and diverges in all other cases. Note that, because `por` constructs another resumption-computation, the branches of a parallel-or can themselves contain parallel subcomputations.

More generally, we can model a concurrent system as a collection of resumption-computations, corresponding to the runnable processes. A *scheduler* flattens this collection into a single computation by repeatedly picking out an element of the collection, running it for a single step, and (if it has not yet terminated) putting it back into the collection. Individual processes can communicate by e.g., a shared store if $\bar{e}$ contains state-effects. Alternatively, we can use a more refined monad,

$$T\alpha = \mu\beta.\, 1 \xrightarrow{\bar{e}} (\alpha + \mathit{req} \times (\mathit{rsp} \to \beta))$$

Here, a computation that suspends now produces a *request* of type req, and must be resumed with *response* of type rsp. With this setup, and a suitable structure on requests and responses, it is simple to write an scheduler matching up senders and receivers in purely functional style. And while similar in efficiency to a traditional call/cc-based thread package [Wan80], such a system also achieves a direct relationship to the usual denotational specification.

We will instead pursue another important aspect of concurrency: instead of making the scheduler pick runnable processes in a strict round-robin fashion, we can make it choose non-deterministically which process to run at each step. That is, we can layer the entire construction atop a nondeterminism monad:

```
structure E = Eff0;

structure E = Represent (structure E = E
                         structure M = Resumptions)
structure Rrs = E
structure ParOps = ParallelOps (structure R = Rrs)

structure E = Represent (structure E = E
                         structure M = State)
structure Rst = E
structure Cell = StateOps (structure R = Rst);

structure E = Represent (structure E = E
                         structure M = ListMonad)
structure Rls = E
structure Conc = ConcurOps (structure RR = Rrs
                            structure RL= Rls);

structure Shared =
struct
   fun store n = (ParOps.yield (); Cell.set n)
   fun fetch () = (ParOps.yield (); Cell.get ())
end;
```

```
structure Resumptions (* : MONAD *) =
struct
   datatype 'a res = DONE of 'a | SUSP of 'a t
   withtype 'a t = unit -> 'a res
   fun glue t = fn () => t () ()

   fun unit a = fn () => DONE a
   fun step (DONE a, f) = f a ()
     | step (SUSP t, f) = SUSP (bind (t, f))
   and bind (t, f) = fn () => step (t (), f)

   fun disp (DONE a) = a
     | disp (SUSP t) = show t
   and show t = disp (t ())
end;

functor ParallelOps (structure R :
                        RMONAD where M = Resumptions) :
   sig
      val yield : unit -> unit
      val por : (unit -> bool) * (unit -> bool) -> bool
   end =
struct
   open Resumptions
   fun yield () = R.reflect (fn () => SUSP (unit ()))
   fun por (t1, t2) =
      let fun step (DONE true, _) = DONE true
            | step (DONE false, p) = p ()
            | step (SUSP t1, t2) = SUSP (rpor (t2, t1))
          and rpor (t1, t2) () = step (t1 (), t2)
      in R.reflect (rpor (R.reify t1, R.reify t2)) end
end;

functor ConcurOps (structure RR :
                      RMONAD where M = Resumptions
                   structure RL :
                      RMONAD where M = ListMonad) :
   sig
      val par : (unit -> 'a) * (unit -> 'b) -> 'a * 'b
      val atomically : (unit -> 'a) -> 'a
   end =
struct
   open Resumptions
   fun atomically t =
      let fun step (DONE a, y) = if y then SUSP (unit a)
                                      else DONE a
            | step (SUSP t, y) = step (t (), true)
          and atom t () = step (t (), false)
      in RR.reflect (atom (RR.reify t)) end
   fun par (t1, t2) =
     let fun step (DONE a, DONE b) = DONE (a,b)
           | step (DONE a, SUSP tb) =
                SUSP (bind (tb, fn b => unit (a,b)))
           | step (SUSP ta, DONE b) =
                SUSP (bind (ta, fn a => unit (a,b)))
           | step (SUSP ta, SUSP tb) = SUSP (rpar (ta, tb))
         and rpar (t1, t2) () =
            if RL.reflect (fn ()=>[true, false])
               then step (t1 (), SUSP t2)
               else step (SUSP t1, t2 ())
     in RR.reflect (rpar (RR.reify t1, RR.reify t2)) end
end;
```

Figure 6: Resumption monad and associated operations

```
val t1 = E.run (fn () =>
           (Conc.par (fn () => Shared.store 3,
                      fn () => Shared.store
                                  (Shared.fetch () + 1));
            Int.toString (Shared.fetch ())))
(* val t1 =
  "<st: 4>4 <or> <st: 4>4 <or> <st: 1>1 <or> <st: 3>3 \
\<or> <st: 4>4 <or> <st: 1>1 <or> <st: 3>3 <or> <st: 1>1 \
\<or> <st: 3>3 <or> <st: 3>3" *)
```

```
val t2 = E.run (fn () =>
           (Conc.par (fn () => Shared.store 3,
                      fn () => Conc.atomically (fn () =>
                                 Shared.store
                                   (Shared.fetch () + 1)));
            Int.toString (Shared.fetch ())))
(* val t2 =
  "<st: 4>4 <or> <st: 3>3 <or> <st: 3>3 <or> <st: 3>3 \
\<or> <st: 3>3 <or> <st: 3>3" *)
```

Here, we see that bracketing a part of the concurrent computation as an atomic section reduces the set of possible changes to the store. While the number of possible interleavings easily gets astronomical in any substantial concurrent program, the simulation is perfectly usable for exhaustively testing individual fragments, such as mutual-exclusion protocols.

Further refinements are possible. For example, by layering a *data* monad ($T\alpha = \mathit{int} \xrightarrow{\bar{e}} \alpha$) above resumptions, we can model thread-specific data, where a running computation can perform a the equivalent of a `getpid()` call to obtain its own thread's unique identifier. We can provide thread-local exceptions, or global ones for aborting the entire concurrent system. We can add an output monad for tracing, but inspect it only for nondeterministic paths in which an exception is raised, and so on.

5 Related work

There are already a large number of proposals for layering effects, both for structuring denotational semantics [Mog90, Esp95] and functional programs [KW93, Ste94, LHJ95]. Generally, however, these approaches pursue modularity in a "flat" multi-effect language, without an explicit effect-typing system. Accordingly, a central problem in such frameworks concerns defining the various effect-operations in such a way that they "lift through" other effects that may be present. For specific effects, more or less ad hoc solutions do exist, but operations that involve reification-like behavior (such as exception handling) do not seem to admit any general solution.

The approach presented in this paper is less ambitious: the basic translation-derived framework exposes the layering explicitly in the effect-types of the proto-operations. Thus, those conflicts that are not automatically resolvable by effect-subsumption must either be exposed to the programmer as typing restrictions that go beyond simple typability (akin to, e.g., keeping track of a function's exception-effects in Java), or the desired semantics must be explicitly encoded in the definitions of the programmer-visible operations. Of course, existing results about lifting specific operations through particular monads can be used for that.

A further difference is that most executable specifications based on monad constructors actually construct and use the full compound monad – explicitly or implicitly – during actual evaluation of programs, usually at a substantial cost. Here, we make a strong distinction between the simple but inefficient specification, and the efficient but (especially for multiple layers) not easily analyzable implementation, taking care only that they agree on the meanings of complete programs.

A second line of related work concerns implementation of various computational paradigms using control operators *directly*, without involving monads at all. Examples include uses of basic call/cc for thread packages [Wan80] and imperative backtracking [HDM93]; simple composable-control [FWFD88, DF92] for nondeterminism and other basic effects, and a number of proposals for hierarchical control [DF90, SF90, GRR95] to represent general layered effects. Most of these are based on operational definitions of the control operators in terms of their actions on evaluation contexts (although many also include sample implementations in terms of Scheme primitives).

Again, we take a more minimalist approach here: we view control operators not as an explicitly exposed programming abstraction in its own right, but only as a means to the end of implementing a monadic specification. (Of course, sometimes – although surprisingly rarely – the most natural description of a computational effect is in fact in terms of continuations; the monadic framework encompasses this as simply another instance.) This frees us from the constraint of defining a general-purpose control mechanism with intuitive operational behavior, and allows us to provide instead a "lean and mean" implementation, which can be formally analyzed without too much work.

6 Conclusions

To be practically compelling, a monad-based framework for effects needs to minimize overhead, both conceptual and computational. We address the former concern by basing the specification on the intuitively familiar concept of definitional translation ("explaining away an effect"), and the latter by an efficient implementation that keeps execution cost at roughly native levels, as long as the effect-invoking and effect-delimiting operations are comparatively rare – as is indeed the case in most functional programs.

In other words, we aim to steer clear of two extremes: on the one hand "the specification is the implementation", resulting from executing a monad-based specification (of, e.g., exceptions) literally; and on the other hand, "the implementation is the specification", resulting from taking a particular imperative implementation (e.g., a thread package) as a guide to specifying interactions with other effects. Instead, we propose a paradigm – monadic reflection – to uniformly relate a layered declarative specification of an effect tower to its ultimate imperative implementation in terms of low-level primitives.

Although the implementation presented here is nominally complete in an operational sense, it should still be viewed as a proof-of-concept prototype, rather than a final solution. That is, the construction establishes that – beyond availability of call/cc and references – no further support from the compiler or runtime system is needed to efficiently implement layered effects. But this does not mean that such support would be undesirable. In particular, a type system for actually enforcing the effect-restrictions statically would be a big help in constructing large programs. To be practical, this would probably need to be largely reconstruction-based, with only minimal explicit annotations; it should also include support for some notion of effect-polymorphism.

On the semantic side, further refinements are also possible. In particular, it should be possible to extend the formal simulation result to "effect-recursive monads", in which the new effect being defined is implicitly used in its own specification (e.g., for a higher-order state, we can store procedures that themselves have state-effects). This would also allow for a more uniform treatment of continuation monads with non-base answer types. It would also be worth investigating whether the present results for simulating a linear $\preceq$-hierarchy can be extended to more general orders.

Acknowledgments

The author is sincerely indebted to Olivier Danvy for a number of perceptive comments, as well as to the POPL reviewers and everyone who influenced earlier versions of this work.

References

[BCL+98] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2), 1998.

[CF94] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In Masami Hagiya and John C. Mitchell, editors, *Symposium on Theoretical Aspects of Computer Software*, number 789 in Lecture Notes in Computer Science, pages 244–272, Sendai, Japan, April 1994.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[Esp95] David A. Espinosa. *Semantic Lego.* PhD thesis, Graduate School of Arts and Sciences, Columbia University, May 1995.

[Fil94] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.

[Fil96] Andrzej Filinski. *Controlling Effects.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1996. Technical Report CMU–CS–96–119.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, 1993.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988.

[GRR95] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Functional Programming and Computer Architecture*, pages 12–23, 1995.

[HD94] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994.

[HDM93] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. (A preliminary version appeared in *Proceedings of the 1991 Symposium on Principles of Programming Languages*).

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(3):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[KW93] David J. King and Philip Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*, pages 134–143, Ayr, Scotland, 1993. Springer-Verlag.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

[LPJ95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, 1971.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.

[Mog90] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML.* The MIT Press, revised edition, 1997.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985.

[Pit96] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, June 1996.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[PW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993.

[Ree92] Jonathan Rees. The Scheme of things: The June 1992 meeting. *Lisp Pointers*, 5(4):40–45, 1992.

[Rey74] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[SF90] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, Portland, Oregon, January 1994.

[Tol98] Andrew Tolmach. Optimizing ML using a hierarchy of monadic types. In Xavier Leroy and Atsushi Ohori, editors, *Types in Compilation, Second International Workshop*, number 1473 in Lecture Notes in Computer Science, pages 97–115, Kyoto, Japan, March 1998.

[Wad92] Philip Wadler. The essence of functional programming (invited talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992.

[Wad98] Philip Wadler. The marriage of effects and monads. In *International Conference on Functional Programming*, 1998.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 LISP Conference*, pages 19–28, Stanford, California, August 1980.