Monads in Action

Andrzej Filinski
DIKU, University of Copenhagen
andrzej@diku.dk

Abstract

In functional programming, monadic characterizations of computational effects are normally understood denotationally: they describe how an effectful program can be systematically expanded or translated into a larger, pure program, which can then be evaluated according to an effect-free semantics. Any effect-specific operations expressible in the monad are also given purely functional definitions, but these definitions are only directly executable in the context of an already translated program. This approach thus takes an inherently Church-style view of effects: the nominal meaning of every effectful term in the program depends crucially on its type.

We present here a complementary, operational view of monadic effects, in which an effect definition directly induces an imperative behavior of the new operations expressible in the monad. This behavior is formalized as additional operational rules for only the new constructs; it does not require any structural changes to the evaluation judgment. Specifically, we give a small-step operational semantics of a prototypical functional language supporting programmer-definable, layered effects, and show how this semantics naturally supports reasoning by familiar syntactic techniques, such as showing soundness of a Curry-style effect-type system by the progress+preservation method.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives, Type structure

General Terms Languages, Theory

Keywords Monads, Computational Effects, Modular Semantics

1. Introduction

Since their introduction by Moggi [12] and popularization by Wadler [20], monads have become an important tool for structuring pure functional programs. Conceptually imperative computations, using features such as exceptions or mutable state, can be uniformly expressed by splitting the program into a main program written in "monadic style", and a prelude of effect definitions, each given by a monad and some associated operations.

Conceptually, the effect definitions are then inlined into the main program, giving a purely functional program that can be evaluated using a standard, pure semantics. This approach takes an inherently Church-style view of effects: the meaning of a term depends fundamentally on its monadic type, since that type (possibly reconstructed using overloading resolution or similar) is used to guide the expansion.

In ML-like languages, one can of course use such an approach as well, but it seems less natural. Rather, programmers tend to think of a fixed collection of computational effects as being built into the language, but possibly being used in restricted ways. That is, rather than building up new behaviors, effect-type systems aim to classify patterns of existing behavior, such as whether a subterm may raise a particular exception, or access part of the store. In particular, the view is Curry-style: types form a (necessarily conservative) approximation of the intrinsic operational behavior of untyped terms.

Monadic reflection [4] attempts to bridge these two views: from a translational specification of a collection of effects in terms of layered monads, it constructs an efficient imperative implementation based on native effects in the language, namely low-level primitives for control and state manipulation. That is, the program can be thought of and analyzed in terms of its definitional expansion into a pure one, but the actual execution happens by an embedding into a language with a fixed collection of effects and associated operational semantics.

For example, consider the prototypical effect of exceptions. Given a type exn of exception names, even in the absence of dedicated syntax, we can make available the essential exception-behaviors in the form of two procedures:

$$\begin{aligned} \mathsf{raise} &: \mathsf{exn} \xrightarrow{ex} \alpha \\ \mathsf{try} &: (1 \xrightarrow{ex} \alpha) \to (\mathsf{exn} \xrightarrow{\epsilon} \alpha) \xrightarrow{\epsilon} \alpha \quad \epsilon \in \{\bot, ex\} \end{aligned}$$

(Ignore for the moment the annotations on the arrows.)

Operationally, $\mathsf{raise}\ e$ signals the exception e, while $\mathsf{try}\ t\ h$ evaluates the protected thunk t with handler h: if $t\,()$ returns a normal, α-typed result a, that a will also be the result of the try; but if evaluation of the thunk causes an exception e to be signaled, then evaluation of $t\,()$ is abandoned, $h\,e$ is evaluated instead, and its result (possibly another raised exception) becomes the final result of the try.

The effect annotations summarize this behavior. An annotated arrow type $\tau \xrightarrow{\epsilon} \tau'$ represents a function from τ to τ' with potential ϵ-effects. Here $\bot$ indicates a *pure* computation, i.e., with at most divergence as an effect, while *ex* indicates a computation that may raise exceptions in addition to diverging. Unannotated function arrows represent manifestly total functions, with no effects at all. In the example, raise evidently has *ex*-effects, whereas the type of try says that, even though the protected expression may signal exceptions, the whole try-expression will not, unless they are signaled by the handler.

How can we formalize this intuitive description of the semantics of exceptions? In the monadic approach, we model a computation with exception-effects using a monad type $\mathsf{T}\alpha$, defined by

$$\mathsf{T}\alpha \;\equiv\; 1 \xrightarrow{\bot} (\alpha + \mathsf{exn})$$

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01...$10.00

That is, an *ex*-computation of a value of type α is a $\bot$-computation returning either α or exn. The associated term components can be defined as follows:

$$\begin{aligned}
\mathsf{unit} &: \alpha \to \mathsf{T}\alpha \\
\mathsf{unit} &\equiv \lambda a.\, \lambda().\, {'}\mathbf{inl}\, a \\[4pt]
\mathsf{bind} &: \mathsf{T}\alpha \to (\alpha \to \mathsf{T}\beta) \to \mathsf{T}\beta \\
\mathsf{bind} &\equiv \lambda t.\, \lambda f.\, \lambda().\, \mathbf{let}\ s = t\,()\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{case}\ s\ \mathbf{of} \\
&\qquad\qquad\qquad\quad \mathbf{inl}\, a.\, f\, a\, () \\
&\qquad\qquad\qquad\quad \mathbf{inr}\, e.\, {'}\mathbf{inr}\, e
\end{aligned}$$

(We write ${'}M$ to emphasize the normally implicit coercion of a trivial computation M into a potentially effectful one.)

A *monadic reflection* is then a pair of functions witnessing the isomorphism between the original opaque, "imperative" view of exceptions, and the above transparent, "declarative" one:

$$\begin{aligned}
\mathsf{reflect}^{ex} &: \mathsf{T}\alpha \to (1 \overset{ex}{\to} \alpha) \\
\mathsf{reify}^{ex} &: (1 \overset{ex}{\to} \alpha) \to \mathsf{T}\alpha
\end{aligned}$$

(We write $\mathsf{reflect}$ as a total function returning a thunk, to emphasize the symmetry with reify.) Using those, we can define the actual exception operations as follows:

$$\begin{aligned}
\mathsf{raise} &\equiv \lambda e.\, \mathsf{reflect}^{ex}(\lambda().\, {'}\mathbf{inr}\, e)\ () \\
\mathsf{try} &\equiv \lambda t.\, \lambda h.\, \mathbf{let}\ s = (\mathsf{reify}^{ex}\, t)\ ()\ \mathbf{in} \\
&\qquad\quad \mathbf{case}\ s\ \mathbf{of} \\
&\qquad\quad\ \ \mathbf{inl}\, a.\, {'}a \\
&\qquad\quad\ \ \mathbf{inr}\, e.\, h\, e
\end{aligned}$$

That is, $\mathsf{raise}\, e$ constructs a transparent representation of a raised exception, and reflects it as an *ex*-computation, while try reifies the protected *ex*-computation into a sum-returning pure one, runs it, and splits into cases on the result.

For example, if $\mathsf{exn} = \mathsf{string}$, we can then write a program like

$$\begin{aligned}
&\mathsf{try}\ (\lambda().\, 3 + \mathsf{raise}\ \texttt{"foo"}) \\
&\quad (\lambda e.\, \mathbf{if}\ e = \texttt{"foo"}\ \mathbf{then}\ 4\ \mathbf{else}\ 5)
\end{aligned}$$

or, with explicit computation-sequencing:

$$\begin{aligned}
&\mathsf{try}\ (\lambda().\, \mathbf{let}\ r = \mathsf{raise}\ \texttt{"foo"}\ \mathbf{in}\ {'}(3 + r)) \\
&\quad (\lambda e.\, \mathbf{if}\ e = \texttt{"foo"}\ \mathbf{then}\ {'}4\ \mathbf{else}\ {'}5)
\end{aligned}$$

To compute the meaning of such a program according to the defined semantics of exceptions, we can expand the definitions of raise/try from above, monad-translate the entire program (i.e.. transform relevant **let**s into binds and ${'}$-s into units), and finally replace $\mathsf{reflect}$ and reify by identity functions. Then the resulting program would be well typed, and evaluate to 4, using only $\bot$-computations.

Note, however, that the monad translation needs the type information: the **let** and ${'}$ in the first line of the main program should be transformed, but the ones in unit, bind, and raise should not, because they only sequence $\bot$-computations. Even more subtly, whether the ${'}$ in try should be transformed or not depends on which type instance we are using try at: in the sample program, the handler h itself was pure, so the result from try should not be tagged; but if we had replaced the ${'}5$ with $\mathsf{raise}\ e$ (i.e., taking $\epsilon = ex$ in the type of try), the successful result a from try would need to be explicitly tagged as non-exceptional.

Monadic reflection and reification are thus easy to understand from the Church perspective: as part of the conceptual expansion of the effectful program into a pure, monadic-style one, they actually disappear entirely; their function is merely to mark the boundaries between parts of the program being translated using different monads.

A natural question arises, however: could we instead give the reflection operators a direct, effectful operational semantics, so that the raise and try defined in terms of them would behave in the expected way? More generally can we derive such a semantics purely mechanically, using only the definitions of the monad components, for an arbitrary monadic effect? And finally, can we assign meanings even to terms without effect-annotations, rather than requiring a potentially complex type inference/reconstruction phase before execution?

In the following, we answer these questions affirmatively. We show how a simple, small-step operational semantics of a prototypical functional language (conceptually situated somewhere in the middle between Haskell and ML on the purity spectrum) can be incrementally extended with programmer-defined monadic effects, in the form of additional reduction rules for the new operations, but not requiring modifications or extensions of existing ones. Moreover, for several familiar examples, those automatically constructed rules correspond very directly to a Felleisen-style evaluation-context semantics of the effectful operations.

Finally, the construction can be carried out entirely in Curry style: all syntactically correct programs are assigned inherent operational meanings, independent of any type information. However, a descriptive type system can subsequently be imposed on them, with soundness (absence of stuck states) shown by the usual syntactic methods. We consider a particular such type system here, though more advanced ones are certainly possible – either allowing more programs to be typed, or giving more informative types to already typable ones.

2. Metalanguage

Though our account could be given in the context of an ML-like CBV language, the development runs significantly smoother in the slightly more abstract setting of a computational metalanguage with explicit sequencing of computations using **let** and value-inclusion, as illustrated in the Introduction. It is hoped that this presentation style will also provide a neutral ground, familiar to both ML programmers (who can think of it as merely a systematic convention of naming the results of all subexpressions, akin to A-normal forms), and to Haskell programmers (who can think of it as a variant of **do**-notation, although with an unusual operational semantics).

The metalanguage is essentially the MultiMonadic Metalanguage (M^{3}L) of [5], with a slightly restricted term syntax, better suited for operational interpretations. The language is stratified into a *term level*, in which all the computations are expressed, and a *program level*, used to also express effect definitions. In this section, we introduce the syntax and typing of both levels; in the next one, we will look at the operational semantics. Throughout the paper, we will use the convention that shaded constructs, judgments, and rules represent the extensions specifically needed to work with programmer-defined effects and monadic reflection, while the unshaded ones are essentially independent of how effects are added to the metalanguage.

2.1 Effects

The fundamental concept of the metalanguage is that of an *effect* e, representing a collection of possible computational behaviors. The simplest effect is that of potential nontermination, written $\bot$. However, it is often more appropriate to think of $\bot$ as simply the underlying computational fabric of the language. In other words, possible divergence of programs is merely the price we must pay for allowing general recursive computations, rather than something we explicitly set out to encompass. (Explicit partiality of a function is often better represented using some variant of the exception monad.) All effects in the language will be built on top of $\bot$, to

Syntax

$$
\begin{array}{rcl}
e & ::= & \bot \mid \varepsilon \\
\tau & ::= & \alpha \mid 1 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_0 \mid \mu\alpha.\tau \mid \sigma \\
\sigma & ::= & \langle e\rangle\tau \mid \tau \to \sigma \mid \top \mid \sigma_1 \mathbin{\&} \sigma_2 \\
\Sigma & ::= & \cdot \mid \Sigma, \varepsilon \sim \alpha.\sigma, e \prec \varepsilon
\end{array}
$$

Subeffecting $\boxed{\vdash_\Sigma e \le e'}$

$$
\frac{}{\vdash_\Sigma e \le e}\ (\text{SE-Refl}) \qquad \frac{\vdash_\Sigma e \le e'' \quad \vdash_\Sigma e'' \le e'}{\vdash_\Sigma e \le e'}\ (\text{SE-Trans})
$$

$$
\frac{}{\vdash_\Sigma \bot \le e}\ (\text{SE-Bot}) \qquad \frac{(e \prec \varepsilon) \in \Sigma}{\vdash_\Sigma e \le \varepsilon}\ (\text{SE-Def})
$$

Effect basing $\boxed{\vdash_\Sigma e \preceq \sigma}$

$$
\frac{}{\vdash_\Sigma e \preceq \langle e\rangle\tau}\ (\text{EB-Eff}) \qquad \frac{\vdash_\Sigma e \preceq \sigma}{\vdash_\Sigma e \preceq \tau \to \sigma}\ (\text{EB-Fun})
$$

$$
\frac{}{\vdash_\Sigma e \preceq \top}\ (\text{EB-Unit}) \qquad \frac{\vdash_\Sigma e \preceq \sigma_1 \qquad \vdash_\Sigma e \preceq \sigma_2}{\vdash_\Sigma e \preceq \sigma_1 \mathbin{\&} \sigma_2}\ (\text{EB-Prod})
$$

$$
\frac{\vdash_\Sigma e \le e' \quad \vdash_\Sigma e' \preceq \sigma}{\vdash_\Sigma e \preceq \sigma}\ (\text{EB-Sub})
$$

Effect signature $\boxed{\vdash \Sigma}$

$$
\frac{}{\vdash \cdot}\ (\text{D-Empty}) \qquad \frac{\vdash \Sigma \quad \vdash_\Sigma e \preceq \sigma}{\vdash \Sigma, \varepsilon \sim \alpha.\sigma, e \prec \varepsilon}\ (\text{D-Eff})
$$

Figure 1. Effects and types

allow a simple treatment of especially type-level recursion. Indeed, we can think of $\bot$ as representing the effect of performing any non-trivial computations at all, even ones that could relatively easily be seen to be terminating.

We could have parameterized the language by a larger collection of base effects, but to keep the operational semantics concise, we require all other effects to be explicitly defined at the program level. Since the language for expressing effects includes the syntactic counterparts of the main domain-theoretic primitives (though not powerdomains), most conventional monadic effects can be indeed defined in this way. An effect is thus either the single primitive effect $\bot$, or a defined effect ε.

The second fundamental notion is that of *subeffecting*. Informally, an effect e is a subeffect of e', written $e \le e'$, if all behaviors modeled by e are also valid behaviors of e'. In general, this covers qualitative notions of behavior inclusion (such as exceptions being a subeffect of exceptions-and-state), but also quantitative ones, such as the exact set of exceptions that a computation may signal. Subeffecting is naturally reflexive and transitive, with $\bot$ as the least element. (Denotationally, a subeffecting $e \le e'$ corresponds to a monad morphism between the monads representing e and e', but we will not pursue the category-theoretic ramifications here.)

For our purposes, the only source of proper subeffecting will be when a new effect ε is explicitly defined as an immediate supereffect of some existing e, written $e \prec \varepsilon$. The effect structure is summarized in Figure 1.

2.2 Types

Although the proposed operational semantics will not depend on type information, we present the language here together with a simple type system, since most of the terms can be better understood in such a setting.

In the syntax of types, we distinguish between general, or *value*, types τ, and a subclass of *computation types* σ. Only terms of computation type will be given an operational interpretation; value types represent inert data. (When appropriate, suspended computations can also be treated as data, though.)

The grammar of types is also given in Figure 1. Again, since most conventional base types can be built up from the general constructs, there are no type constants. For example, we will later define the natural numbers as $\mathsf{nat} \equiv \mu\alpha.1 + \alpha$. (Throughout the paper, we use a $\mathsf{sans\text{-}serif}$ font for notational abbreviations.)

The general types include products, sums, and recursion, as well as computation types. The latter include function, product, and effect types. To avoid confusion between the two notion of products, we use & and $\top$ for the computation variant (inspired by the linear-logic notation, though with no formal connection implied). The effect type $\langle e\rangle\tau$ classifies computations producing τ-typed results, with potential e-behaviors. Note that the codomain of a function type must always be computational (though possibly itself another function space). Thus, $\mathsf{nat} \to \mathsf{nat} \to \langle\bot\rangle\mathsf{nat}$ is a well-formed type (isomorphic to $\mathsf{nat} \times \mathsf{nat} \to \langle\bot\rangle\mathsf{nat}$), but $\mathsf{nat} \to \mathsf{nat}$ is not. This is a slight relaxation of Moggi's metalanguage, in which function codomains can only be effect-types directly. The extension is mainly for syntactic convenience: as suggested above, it allows curried notation for functions that would otherwise have to be written in uncurried style.

The intuition behind general computation-types is that they classify *parameterized* effectful computations. In the case of effect-types, there is no parametrization; function-typed computations are parameterized by the argument value, and product-typed computations are parameterized by whether their first or second argument is needed. (In particular, $\sigma \mathbin{\&} \sigma$ is thus isomorphic to $(1 + 1) \to \sigma$.)

Formalizing this intuition, we say that a computation type σ is *based on* an effect e, written $\vdash_\Sigma e \preceq \sigma$, when σ can be seen as a parameterized e-computation. We refer to computations of type $\langle e\rangle\tau$ as *simple* e-computations; others (including $\langle e'\rangle\tau$ when $e < e'$) are called *generalized*. (Denotationally, when $\vdash_\Sigma e \preceq \sigma$, σ can be interpreted as an *algebra* for the monad representing effect e [11, VI.2]. In a domain-theoretic setting, computation types denote pointed cpos, i.e., algebras for the lifting monad that models $\bot$.)

We finally define the notion of an effect signature Σ, containing *declarations* of all effects introduced in a program, each with their immediate subeffect, and a conceptual realization as an already well-formed computation type (i.e., only the type-constructor part of the formal monad defining the effect). For example, the effect of exceptions might be declared as $\mathit{ex} \sim \alpha.\langle\bot\rangle(\alpha + \mathsf{nat})$, saying that a computation with exception-effects can be understood as a pure computation returning either the expected result, or an error code. If a notion of state-effects st had been declared earlier in the signature, we could also have declared exceptions as $\mathit{ex} \sim \langle \mathit{st}\rangle(\alpha + \mathsf{nat})$, signifying that this notion of an ex-computation may also perform state access, in addition to potentially raising an exception.

For an immediate-subeffect declaration $e \prec \varepsilon$ in a signature to be valid, the computation-type realizing ε must be based (possibly indirectly) on e; this is checked by rule D-Eff.

2.3 Terms

Let us look now at the level of terms. We consider first the generic parts, and then the reflection/reification operators specifically.

2.3.1 Generic fragment

The syntax and typing rules are given in Figure 2. We generally use N for terms of computation type, and M for unrestricted ones, but there is no formal distinction between value- and computation-terms in the syntax. (In particular, variables can denote either kind.) The rules of the operational semantics will specify exactly when a

Syntax

$$M, N ::= x \mid \mathbf{val}\, M \mid \mathbf{glet}\, x \Leftarrow N_1.\, N_2 \mid \lambda x.\, N \mid N\, M \mid \langle\rangle \mid \langle N_1, N_2\rangle \mid \mathbf{prj}_i\, N \mid () \mid (M_1, M_2) \mid \mathbf{split}(M, x_1.x_2.N)$$
$$\mid \mathbf{inj}_i\, M \mid \mathbf{case}(M, x_1.N_1, x_2.N_2) \mid \mathbf{roll}\, M \mid \mathbf{unroll}\,(M, x.N) \mid \mathbf{fix}\, x.N \mid [N]^{\varepsilon} \mid \mathbf{glet}\, x \Leftarrow \mu^{\varepsilon}(N_1).\, N_2$$
$$P ::= \mathbf{run}\, N \mid \mathbf{leteffect}\, \varepsilon \succ e\, \mathbf{be}\, (\alpha.\sigma_\varepsilon, N_{\mathrm{u}}, N_{\mathrm{b}})\, \mathbf{in}\, P$$

Typing environment $\boxed{\vdash_\Sigma \Gamma}$

$$\Gamma ::= \cdot \mid \Gamma, x{:}\,\tau \text{ (where } \vdash_\Sigma \tau)$$

Term typing $\boxed{\Gamma \vdash_\Sigma M : \tau}$

$$\frac{(x{:}\,\tau) \in \Gamma}{\Gamma \vdash_\Sigma x : \tau}\ (\text{Tv-Var}) \qquad \frac{\Gamma \vdash_\Sigma M : \tau}{\Gamma \vdash_\Sigma \mathbf{val}\, M : \langle e\rangle\tau}\ (\text{Tc-Val}) \qquad \frac{\Gamma \vdash_\Sigma N_1 : \langle e\rangle\tau \quad \Gamma, x{:}\,\tau \vdash_\Sigma N_2 : \sigma \quad \vdash_\Sigma e \preceq \sigma}{\Gamma \vdash_\Sigma \mathbf{glet}\, x \Leftarrow N_1.\, N_2 : \sigma}\ (\text{Tc-Glet})$$

$$\frac{\Gamma, x{:}\,\tau \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma \lambda x.\, N : \tau \to \sigma}\ (\text{Tc-Lam}) \qquad \frac{\Gamma \vdash_\Sigma N : \tau \to \sigma \quad \Gamma \vdash_\Sigma M : \tau}{\Gamma \vdash_\Sigma N\, M : \sigma}\ (\text{Tc-App})$$

$$\frac{}{\Gamma \vdash_\Sigma \langle\rangle : \top}\ (\text{Tc-Unit}) \qquad \frac{\Gamma \vdash_\Sigma N_1 : \sigma_1 \quad \Gamma \vdash_\Sigma N_2 : \sigma_2}{\Gamma \vdash_\Sigma \langle N_1, N_2\rangle : \sigma_1 \,\&\, \sigma_2}\ (\text{Tc-Pair}) \qquad \frac{\Gamma \vdash_\Sigma N : \sigma_1 \,\&\, \sigma_2}{\Gamma \vdash_\Sigma \mathbf{prj}_i\, N : \sigma_i}\ (\text{Tc-Prj})$$

$$\frac{}{\Gamma \vdash_\Sigma () : 1}\ (\text{Tv-Unit}) \qquad \frac{\Gamma \vdash_\Sigma M_1 : \tau_1 \quad \Gamma \vdash_\Sigma M_2 : \tau_2}{\Gamma \vdash_\Sigma (M_1, M_2) : \tau_1 \times \tau_2}\ (\text{Tv-Pair}) \qquad \frac{\Gamma \vdash_\Sigma M : \tau_1 \times \tau_2 \quad \Gamma, x_1{:}\,\tau_1, x_2{:}\,\tau_2 \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma \mathbf{split}(M, x_1.x_2.N) : \sigma}\ (\text{Tc-Split})$$

$$\frac{\Gamma \vdash_\Sigma M : \tau_i}{\Gamma \vdash_\Sigma \mathbf{inj}_i\, M : \tau_1 + \tau_2}\ (\text{Tv-Inj}) \qquad \frac{\Gamma \vdash_\Sigma M : \tau_1 + \tau_2 \quad \Gamma, x{:}\,\tau_1 \vdash_\Sigma N_1 : \sigma \quad \Gamma, x{:}\,\tau_2 \vdash_\Sigma N_2 : \sigma}{\Gamma \vdash_\Sigma \mathbf{case}(M, x.N_1, x.N_2) : \sigma}\ (\text{Tc-Case})$$

$$\frac{\Gamma \vdash_\Sigma M : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash_\Sigma \mathbf{roll}\, M : \mu\alpha.\tau}\ (\text{Tv-Roll}) \qquad \frac{\Gamma \vdash_\Sigma \mathbf{roll}\, M : \mu\alpha.\tau \quad \Gamma, x{:}\,\tau[\mu\alpha.\tau/\alpha] \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma \mathbf{unroll}\,(M, x.N) : \sigma}\ (\text{Tc-Unroll}) \qquad \frac{\Gamma, x{:}\,\sigma \vdash_\Sigma N : \sigma}{\Gamma \vdash_\Sigma \mathbf{fix}\, x.N : \sigma}\ (\text{Tc-Rec})$$

$$\frac{\Gamma \vdash_\Sigma N : \langle\varepsilon\rangle\tau \quad (\varepsilon \sim \alpha.\sigma_\varepsilon) \in \Sigma}{\Gamma \vdash_\Sigma [N]^{\varepsilon} : \sigma_\varepsilon[\tau/\alpha]}\ (\text{Tc-Reif}) \qquad \frac{\Gamma \vdash_\Sigma N_1 : \sigma_\varepsilon[\tau/\alpha] \quad \Gamma, x{:}\,\tau \vdash_\Sigma N_2 : \sigma \quad \vdash_\Sigma \varepsilon \preceq \sigma \quad (\varepsilon \sim \alpha.\sigma_\varepsilon) \in \Sigma}{\Gamma \vdash_\Sigma \mathbf{glet}\, x \Leftarrow \mu^{\varepsilon}(N_1).\, N_2 : \sigma}\ (\text{Tc-Glet-Refl})$$

Program typing $\boxed{\vdash_\Sigma P}$:

$$\frac{\cdot \vdash_\Sigma N : \langle\bot\rangle\mathsf{nat}}{\vdash_\Sigma \mathbf{run}\, N}\ (\text{PT-Run})$$

$$\frac{\vdash_\Sigma e \preceq \sigma_\varepsilon \quad \cdot \vdash_\Sigma N_{\mathrm{u}} : \alpha_1 \to \sigma_\varepsilon[\alpha_1/\alpha] \quad \cdot \vdash_\Sigma N_{\mathrm{b}} : \sigma_\varepsilon[\alpha_1/\alpha] \to (\alpha_1 \to \sigma_\varepsilon[\alpha_2/\alpha]) \to \sigma_\varepsilon[\alpha_2/\alpha] \quad \vdash_{\Sigma, \varepsilon\sim\alpha.\sigma_\varepsilon, e \prec \varepsilon} P}{\vdash_\Sigma \mathbf{leteffect}\, \varepsilon \succ e\, \mathbf{be}\, (\alpha.\sigma_\varepsilon, N_{\mathrm{u}}, N_{\mathrm{b}})\, \mathbf{in}\, P}\ (\text{PT-Leteff})$$

Figure 2. Term and program syntax and typing

term of computation type should be evaluated, and when it should be treated as a passive datum.

The term syntax is largely conventional, corresponding to the usual introduction and elimination principles for the available type constructors. We note that all elimination has to happen in the context of a computation, i.e., the type of the result in **split**, **case**, or **unroll** must be computational, and thus ultimately based on $\bot$. This restriction significantly simplifies the reduction semantics, while not presenting any major problems in elaborating common object languages into the metalanguage. With that proviso, all the unshaded typing rules except TC-VAL and TC-GLET should be self-explanatory.

The rule for **val** is also straightforward: **val** M represents an effect-free computation, that merely returns M; it corresponds to the $'M$ notation of the Introduction, or *return* M in Haskell. **Glet** represents computation sequencing; informally, **glet** $x \Leftarrow N_1.\, N_2$ first evaluates N_1 (performing its effects), and then evaluates N_2 with x bound to the result of N_1. The rule TC-GLET is a generalization of the usual monadic **let**-typing rule (or Haskell's **do**-notation), however: as usual, N_1 must have an effect-type $\langle e\rangle\tau$, but the body N_2 can be of any computation-type based on e, not necessarily exactly of the form $\langle e\rangle\tau'$. This includes function spaces into e-computations, and notably also computations with supereffects (immediate or otherwise) of e. In particular, the idiom

$$\mathbf{glet}\, x \Leftarrow N.\, \mathbf{val}\, x$$

can lift a term $N : \langle e\rangle\tau$ to the type $\langle e'\rangle\tau$, when $e \leq e'$. (Later we shall introduce a subtyping system, allowing us to write N directly, without such an explicit effect-inclusion.) Note that, when N_2 is a non-trivially parameterized computation, then so is **glet** $x \Leftarrow N_1.\, N_2$; that is, the computation in N_1 is not performed until the entire computation has been supplied with the parameter value.

Summarizing, the language defined so far is thus essentially FPC [7], with explicit sequencing of $\bot$-effects. As previously mentioned, we can define natural numbers with the usual operations:

$$\begin{aligned}
\mathsf{nat} &\equiv \mu\alpha.1 + \alpha \\
\mathsf{z} &\equiv \mathbf{roll}\, \mathbf{inj}_1\, () \\
\mathsf{s}\, M &\equiv \mathbf{roll}\, \mathbf{inj}_2\, M \\
\mathsf{ncase}(M, N_{\mathrm{z}}, x.N_{\mathrm{s}}) &\equiv \mathbf{unroll}\,(M, m.\mathbf{case}(m, \_.N_{\mathrm{z}}, x.N_{\mathrm{s}})) \\
\mathsf{plus} &\equiv \\
\mathbf{fix}\, f.\lambda n.\, \lambda m.\, \mathsf{ncase}&(n, \mathbf{val}\, m, n'.\mathbf{glet}\, r \Leftarrow f\, n'\, m.\, \mathbf{val}\,(\mathsf{s}\, r))
\end{aligned}$$

(The function plus could also have been coded tail recursively, but the above variant is more representative of the sequentialized style.) We can derive the following typing rules for the above abbreviations:

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{z} : \mathsf{nat}} \qquad \frac{\Gamma \vdash_\Sigma M : \mathsf{nat}}{\Gamma \vdash_\Sigma \mathsf{s}\, M : \mathsf{nat}}$$

$$\frac{\Gamma \vdash_\Sigma M : \mathsf{nat} \quad \Gamma \vdash_\Sigma N_\mathrm{z} : \sigma \quad \Gamma, x{:}\,\mathsf{nat} \vdash_\Sigma N_\mathrm{s} : \sigma}{\Gamma \vdash_\Sigma \mathsf{ncase}(M, N_\mathrm{z}, x.N_\mathrm{s}) : \sigma}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \langle e \rangle \mathsf{nat}}$$

(We write derived rules with dashed lines.) The type of plus can be derived for any e, not only $\bot$. Again, it does not mean that plus is intended to be partial, only that it involves non-trivial computation.

With general μ-types, we can define other inductive types such as lists, but also datatypes with embedded computations, such as infinite streams or search trees, or even properly reflexive ones such as $\mu\alpha.\mathsf{nat} + (\alpha \to \langle\bot\rangle\alpha)$.

2.3.2 Monadic reflection

The last two term constructs enable actual programming with defined computational effects. The first, *monadic reification* $[N]^\varepsilon$, represents an unsealing of an effect ε in terms of its declared computational realization. Continuing the exception example from before, a term $N : \langle ex \rangle\mathsf{bool}$ is an opaque boolean-returning computation, which can only be sequenced before or after another *ex*-computation using **glet**. But its reification, $[N]^{ex} : \langle\bot\rangle(\mathsf{bool}+\mathsf{nat})$ can be explicitly evaluated, assuming it terminates, to a sum-typed value, which can then be the subject of a **case**. In particular, the reification can be used to halt the propagation of a raised exception: by reifying a computation, we get to exercise non-standard control over its behavior. We could, of course, work with the reified representation all the time, but this is likely to be both less efficient and more verbose.

The other operation, lending its name to the whole pair, is *monadic reflection*. It seals an explicit computation in the realization type into an opaque computation in the defined effect-type. For technical convenience in the operational semantics later, we have integrated it as a variant of the **glet** construct, but it can be understood independently. In particular, we can introduce an "isolated" reflect as an abbreviation,

$$\hat{\mu}^\varepsilon(N) \;\equiv\; \mathbf{glet}\, x \Leftarrow \mu^\varepsilon(N).\, \mathbf{val}\, x$$

with a derived typing rule exactly opposite to TC-REIF:

$$\frac{\Gamma \vdash_\Sigma N : \sigma_\varepsilon[\tau/\alpha] \quad (\varepsilon \sim \alpha.\sigma_\varepsilon) \in \Sigma}{\Gamma \vdash_\Sigma \hat{\mu}^\varepsilon(N) : \langle\varepsilon\rangle\tau}$$

Reflection provides the only means for introducing non-trivial behaviors of an effect-type; otherwise, only pure computations (including divergence) would be expressible. In the exception example, reflection can be used to create an $\langle ex \rangle\mathsf{bool}$-computation from the explicit error-returning computation $\mathbf{val}\,\mathbf{inj}_2\,(\mathsf{s}\,\mathsf{z}) : \langle\bot\rangle(\mathsf{bool}+\mathsf{nat})$. As long as the generic rules for **glet** properly propagate such an exceptional result until it is reified again, reification and reflection together allow us to define a conventional exception facility, as sketched in the Introduction. We will return to this example in Section 4.1, as well as other examples of defined effects.

2.4 Subtyping

The type system as presented so far requires effects in types to match up exactly: if a term with a given effect is needed somewhere (e.g., as a function argument), we cannot simply supply a term with a lesser effect. Instead, we must explicitly include an effect into its supereffect, using the **glet**–**val** idiom from above.

However, since the operational semantics of effects does not use the type information, the extra **glet** and **val** play no material role at runtime, other than adding an extra reduction step. To free the producer (human or automated) of metalanguage code from inserting such eta-like redexes explicitly, just in order to make the program typecheck, we extend the type system with a simple notion of inclusive subtyping, as detailed in Figure 3.

Subtyping $\boxed{\vdash_\Sigma \tau \le \tau'}$

$$\frac{}{\vdash_\Sigma \tau \le \tau}\ (\text{STV-REFL}) \qquad \frac{\vdash_\Sigma e \le e' \quad \vdash_\Sigma \tau \le \tau'}{\vdash_\Sigma \langle e\rangle\tau \le \langle e'\rangle\tau'}\ (\text{STC-EFF})$$

$$\frac{\vdash_\Sigma \tau' \le \tau \quad \vdash_\Sigma \sigma \le \sigma'}{\vdash_\Sigma \tau \to \sigma \le \tau' \to \sigma'}\ (\text{STC-FUN})$$

$$\frac{}{\vdash_\Sigma \top \le \top}\ (\text{STC-UNIT}) \qquad \frac{\vdash_\Sigma \sigma_1 \le \sigma_1' \quad \vdash_\Sigma \sigma_2 \le \sigma_2'}{\vdash_\Sigma \sigma_1 \,\&\, \sigma_2 \le \sigma_1' \,\&\, \sigma_2'}\ (\text{STC-PROD})$$

Additional rule for $\boxed{\Gamma \vdash_\Sigma M : \tau}$:

$$\frac{\Gamma \vdash_\Sigma M : \tau \quad \vdash_\Sigma \tau \le \tau'}{\Gamma \vdash_\Sigma M : \tau'}\ (\text{T-SUB})$$

Figure 3. Subtyping

We define notions of subtyping for general and computation types (where only the latter is non-trivial), as well as a subsumption rule allowing a term with a given type to be used directly where a supertype is needed. Note that for two types to be related by subtyping, they must have exactly the same shape, differing only in the effects occurring inside $\langle\cdot\rangle$s.

Although the subtyping relation does not include a transitivity rule, such a rule is evidently admissible:

LEMMA 2.1 (Transitivity of subtyping). *If $\tau \le \tau'$ and $\tau' \le \tau''$ then $\tau \le \tau''$*

Proof. Easy induction on the total size of the two derivations (since they must be swapped in the rule STC-FUN), ultimately relying on transitivity of the subeffect relation $\vdash_\Sigma e \le e'$. ■

2.5 Programs

The grammar and typing of programs are shown in the last part of Figure 2. The level of programs contains the actual *definitions* of any effects used in a computation. We refer to the definitions as the program's *prelude*, and the final effectful computation as its *body*. Since general effects will not be uniformly observable, we require the program body $\mathbf{run}\ N$ to not have any top-level behaviors other than potential divergence. All behaviors introduced by reflection thus have to be eliminated by reification, before the final program result can be observed. The type system straightforwardly enforces this requirement. Also, to allow final results to be observed atomically, we require the result (if any) to contain no embedded computations, not even with only $\bot$-behavior; for concreteness, we specify that it must be exactly of type $\langle\bot\rangle\mathsf{nat}$.

The **leteffect** construct introduces a new effect in terms of its monadic specification. That is, in addition to the realization of the effect as a computation type based on a simpler one (and ultimately just on $\bot$), we must supply a *unit* function, expressing how values are turned into computations with **val**, and a *bind* function for sequencing computations in the effect with **let**. Note that we need only define sequencing of simple computations; the framework will induce the appropriate behavior for the generalized instances of **glet**. At the time an effect is defined, we must also declare which effect it is to be considered an immediate supereffect of, for the purpose of creating the effect signature. The unit and bind functions must be typable with the given top-level polymorphic type schemas; this ensures that they can be soundly instantiated at arbitrary types during execution.

To be precise, we call the triple of type constructor and unit/bind terms a *syntactic monad*. We expect them to obey the monad laws and layering condition (see [5]), but of course cannot check those in the type system. For the purpose of the operational semantics, it doesn't actually matter whether the laws hold; but the programs may behave in very counterintuitive, and hard-to-predict, ways if they do not.

In principle, the definitions of the unit and bind functions may use reification and reflection for previously defined effects. In practice, it seems that this generality is rarely useful, and it might be reasonable to require N_{u} and N_{b} to be typable in an empty effect-signature.

Effects with subeffecting evidently form a tree, with $\bot$ at the root. This is more general than what was considered for layered monadic reflection in earlier work [4], where effects were required to be linearly ordered, but we expect the continuation/state-based implementation methodology to generalize relatively straightforwardly to this more general setting.

3. Operational semantics

We now consider how to assign an executable semantics to programs in the language. Like for typing, we start with the straightforward constructs in the base language.

3.1 Generic fragment

The semantics is expressed in conventional small-step style, and is shown in Figure 4. The effect-free fragment is inspired by Levy's Call-By-Push-Value formalism [10], though without a syntactic distinction between values and computations; instead, whether computations are performed or treated as data depends on where they occur. The judgment for term reduction, $\vdash_\Phi N \to N'$, specifies when a closed term reduces to another; it is parameterized by an effect-definition environment Φ, which keeps track of the unit/bind functions of all defined effects, and will only come into play in the next section.

The unshaded rules fall into two classes. The RC-rules enumerate the contexts in which reductions are permitted, while RR-axioms represent proper reductions. Note that computations eliminating value-type terms involve only proper reductions. This is because a *closed* term of type $\tau_1 \times \tau_2$ must necessarily already be a syntactic pair (M_1, M_2); a sum-typed term must be an $\mathbf{inj}_i\, M$; and a μ-typed one must be of the shape $\mathbf{roll}\, M$, so there are no context-reduction variants of **split**, **case**, or **unroll**.

For the computation types, the context rules specify CBN evaluation in the case where a function argument is of a computation type (and hence potentially effectful). CBV evaluation can be forced either at the call site, writing $\mathbf{glet}\; x \Leftarrow N_2.\, N_1\,(\mathbf{val}\; x)$ instead of $N_1\, N_2$; or as part of the function definition, e.g., $\lambda t.\, \mathbf{glet}\; x \Leftarrow t. \cdots$. If the function's domain type is a proper value type, the argument will already have been evaluated by the time the function is called.

We note that the semantics specifies a PCF-style CBN variant: there is no way to force evaluation of a function-typed term, without actually applying the function. In particular, for $\Omega = \mathbf{fix}\; x.x$, there is no observable difference between $\Omega : \tau \to \sigma$ and $\lambda x.\, \Omega : \tau \to \sigma$: both are functions that diverge when applied to anything.

The rule RR-GLET-VAL handles the case where the first part of a sequential composition has no effects. Note that no unit/bind functions are involved here: defining a potential effect does not impose any overhead on the parts of the program that happen not to use it, even if their type allows them to; for example, a successfully returning function of declared type $\mathsf{nat} \to \langle ex\rangle\mathsf{bool}$ goes through exactly the same steps as one of type $\mathsf{nat} \to \langle\bot\rangle\mathsf{bool}$.

3.2 Monadic reflection

Consider now the rules involving reflection and reification, exemplified by the particular case of exceptions. Note first that the interior of a reification is also a reduction context (rule RC-REIF): to reduce a term $[N]^{ex}$ of type $\langle\bot\rangle(\mathsf{bool} + \mathsf{nat})$, we reduce its body $N : \langle ex\rangle\mathsf{bool}$, for as long as this is possible. N may reduce forever, in which case so will $[N]^{ex}$. Or N may eventually reduce to $\mathbf{val}\; M$, where M is either true or false), in which case RR-REIF-VAL will return the transparent representation of a trivial computation, as specified by the N_{u} for exceptions, i.e., $\mathbf{val}\; \mathbf{inj}_1\, M$.

The rule RR-REIF-GLET-REFL covers the case when the body of a reification is a **glet** whose binder is a reflection. In this case, N_1 is a transparent representation of the ε-computation that is just about to happen, and thus we can directly use the bind function of the monad to sequence N_1 before the still-to-be-reified N_2. Note that, in a well-typed program, the body N_2 will be exactly of type $\langle e\rangle\tau'$, rather than a general e-computation, thus requiring only a non-generalized bind function.

The role of RP-GLET is to *propagate* an active reflection to meet a matching reflection. It does so by using the general flattening rule for monadic **glet**s,

$$\begin{aligned} &\mathbf{glet}\; y \Leftarrow (\mathbf{glet}\; x \Leftarrow N_1.\, N_2).\, N_3 \\ &\quad \to \mathbf{glet}\; x \Leftarrow N_1.\, \mathbf{glet}\; y \Leftarrow N_2.\, N_3 \end{aligned}$$

to rotate a reflect in position N_1 to the head of the term. While this reduction is meaning-preserving even when N_1 is not a reflection, using it only when truly needed, keeps the reduction relation deterministic.

The three other RP-rules propagate reflections out of the other kinds of evaluation contexts: RP-APP and RP-PRJ do so from applications and projections respectively. (Again, in a denotational semantics of the metalanguage [5], these are valid equivalences even with an arbitrary term in place of $\mu^\varepsilon(N_1)$, essentially because application and projections are morphisms of the monad algebras interpreting the computation types in question.)

Finally, RP-REIF propagates ε-reflections destined for some matching reification out of a reification-context for a proper super-effect ε'. For example, assuming $ex = \alpha.\langle st\rangle(\alpha + \mathsf{nat})$ where *st* is a state-effect, if a *st*-reflect (essentially a read/write-request; see Section 4.2) meets an *ex*-reification (typically an exception handler), the reflect should propagate through the reification, but remember it when resuming the computation with the result of the reflection.

The condition $\vdash_\Phi \varepsilon < \varepsilon'$ in the RP-REIFY rule is a bit subtle. Note first that it uses a separate operational judgment to check at runtime that ε is a proper ancestor of ε' in the effect tree. But in a well-typed program, the typing rules actually ensure that $\vdash_\Sigma \varepsilon \leq \varepsilon'$, so it would in principle suffice to merely check that $\varepsilon \neq \varepsilon'$, to prevent an overlap with RR-REIF-GLET-REFL. (Such a check for inequality would allow us to omit the immediate-subeffect assumptions in Φ entirely.) The only role of the stronger condition is thus to make some terms get stuck, that would otherwise have kept on reducing, at least for a while.

We keep the explicit check to emphasize that, in the absence of static type checking (or checking with only an effect-oblivious type system), encountering an reflection from a part of the effect tree not below the reification-effect is considered a runtime type error that should be caught and reported, even if the computation is not stuck hard at this stage. (We will return to this issue in Section 4.3.)

The reduction rules for programs are straightforward: The rule PRC-LETEFF merely makes the monad definition available to its body by extending Φ, while PRR-LETEFF-VAL says that, once a program body has terminated, the effect declaration around it can be removed.

Effect definitions

$$\Phi ::= \cdot \mid \Phi, \varepsilon = (N_{\mathrm{u}}, N_{\mathrm{b}}), e \prec \varepsilon$$

Effect layering $\boxed{\vdash_\Phi e < \varepsilon}$, $\boxed{\vdash_\Phi e \leq e'}$

$$\frac{\vdash_\Phi e \leq e' \qquad (e' \prec \varepsilon) \in \Phi}{\vdash_\Phi e < \varepsilon}\ (\text{RLT-DECL}) \qquad \frac{}{\vdash_\Phi e \leq e}\ (\text{RLE-EQ}) \qquad \frac{\vdash_\Phi e < \varepsilon}{\vdash_\Phi e \leq \varepsilon}\ (\text{RLE-LESS})$$

Term reduction $\boxed{\vdash_\Phi N \to N'}$

$$\frac{\vdash_\Phi N_1 \to N_1'}{\vdash_\Phi \mathbf{glet}\ x \Leftarrow N_1.\, N_2 \to \mathbf{glet}\ x \Leftarrow N_1'.\, N_2}\ (\text{RC-GLET}) \qquad \frac{}{\vdash_\Phi \mathbf{glet}\ x \Leftarrow \mathbf{val}\ M.\, N \to N[M/x]}\ (\text{RR-GLET-VAL})$$

$$\frac{}{\vdash_\Phi \mathbf{glet}\ y \Leftarrow (\mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, N_2).\, N_3 \to \mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, \mathbf{glet}\ y \Leftarrow N_2.\, N_3}\ (\text{RP-GLET})$$

$$\frac{\vdash_\Phi N \to N'}{\vdash_\Phi N\, M \to N'\, M}\ (\text{RC-APP}) \qquad \frac{}{\vdash_\Phi (\lambda x.\, N)\, M \to N[M/x]}\ (\text{RR-APP-LAM})$$

$$\frac{}{\vdash_\Phi (\mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, N_2)\, M \to \mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, N_2\, M}\ (\text{RP-APP})$$

$$\frac{\vdash_\Phi N \to N'}{\vdash_\Phi \mathbf{prj}_i\, N \to \mathbf{prj}_i\, N'}\ (\text{RC-PRJ}) \qquad \frac{}{\vdash_\Phi \mathbf{prj}_i\, \langle N_1, N_2\rangle \to N_i}\ (\text{RR-PRJ-PAIR})$$

$$\frac{}{\vdash_\Phi \mathbf{prj}_i\, (\mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, N_2) \to \mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, \mathbf{prj}_i\, N_2}\ (\text{RP-PRJ})$$

$$\frac{\vdash_\Phi N \to N'}{\vdash_\Phi [N]^\varepsilon \to [N']^\varepsilon}\ (\text{RC-REIF}) \qquad \frac{(\varepsilon = (N_{\mathrm{u}}, N_{\mathrm{b}})) \in \Phi}{\vdash_\Phi [\mathbf{val}\ M]^\varepsilon \to N_{\mathrm{u}}\, M}\ (\text{RR-REIF-VAL})$$

$$\frac{(\varepsilon = (N_{\mathrm{u}}, N_{\mathrm{b}})) \in \Phi}{\vdash_\Phi [\mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, N_2]^\varepsilon \to N_{\mathrm{b}}\, N_1\, (\lambda x.\, [N_2]^\varepsilon)}\ (\text{RR-REIF-GLET-REFL})$$

$$\frac{\vdash_\Phi \varepsilon < \varepsilon'}{\vdash_\Phi [\mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, N_2]^{\varepsilon'} \to \mathbf{glet}\ x \Leftarrow \mu^\varepsilon(N_1).\, [N_2]^{\varepsilon'}}\ (\text{RP-REIF})$$

$$\frac{}{\vdash_\Phi \mathbf{split}((M_1, M_2), x_1.x_2.N) \to N[M_1/x_1, M_2/x_2]}\ (\text{RR-SPLIT-PAIR}) \qquad \frac{}{\vdash_\Phi \mathbf{case}(\mathbf{inj}_i\, M, x_1.N_1, x_2.N_2) \to N_i[M/x_i]}\ (\text{RR-CASE-INJ})$$

$$\frac{}{\vdash_\Phi \mathbf{unroll}\,(\mathbf{roll}\, M, x.N) \to N[M/x]}\ (\text{RR-UNROLL-ROLL}) \qquad \frac{}{\vdash_\Phi \mathbf{fix}\ x.N \to N[(\mathbf{fix}\ x.N)/x]}\ (\text{RR-FIX})$$

Program reduction $\boxed{\vdash_\Phi P \to P'}$

$$\frac{\vdash_\Phi N \to N'}{\vdash_\Phi \mathbf{run}\ N \to \mathbf{run}\ N'}\ (\text{PRC-RUN})$$

$$\frac{\vdash_{\Phi, \varepsilon = (N_{\mathrm{u}}, N_{\mathrm{b}}), e \prec \varepsilon} P \to P'}{\vdash_\Phi \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.\sigma_\varepsilon, N_{\mathrm{u}}, N_{\mathrm{b}})\ \mathbf{in}\ P \to \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.\sigma_\varepsilon, N_{\mathrm{u}}, N_{\mathrm{b}})\ \mathbf{in}\ P'}\ (\text{PRC-LETEFF})$$

$$\frac{}{\vdash_\Phi \mathbf{leteffect}\ \varepsilon \succ e\ \mathbf{be}\ (\alpha.\sigma_\varepsilon, N_{\mathrm{u}}, N_{\mathrm{b}})\ \mathbf{in}\ \mathbf{run}\ \mathbf{val}\ M \to \mathbf{run}\ \mathbf{val}\ M}\ (\text{PRR-LETEFF-VAL})$$

Figure 4. Reduction rules (explicit formulation)

3.3 Properties of the reduction system

We can show a number of desirable results of the reduction system. All of these have been fully formalized and verified using the Twelf proof assistant [16]. (The formalization is available from `http://www.diku.dk/~andrzej/papers/`.)

DEFINITION 3.1. *A closed, computation-typed term is said to be* canonical *if it is of one of the forms* **val** M, $\lambda x.\, N$, $\langle\rangle$, $\langle N_1, N_2\rangle$, *or* **glet** $x \Leftarrow \mu^\varepsilon(N_1).\, N_2$. *A complete program is* finished *when of the form* **run val** M.

THEOREM 3.2 (Determinacy). *Term and programs reduce uniquely:*

1. *If* $\vdash_\Phi N \to N'$ *and* $\vdash_\Phi N \to N''$ *then* $N' = N''$
2. *If* $\vdash_\Phi P \to P'$ *and* $\vdash_\Phi P \to P''$ *then* $P' = P''$

Proof. Straightforward induction on derivations, using that canonical terms as defined above are irreducible, and that strict subeffecting $\vdash_\Phi e < e'$ is irreflexive (thus preventing an overlap between RR-REIF-GLET-REFL and RP-REIF). ∎

We can also show soundness of the type system, using the well-established method of progress and preservation lemmas [17, 21]. We first define the judgment $\boxed{\vdash \Phi : \Sigma}$, expressing that a set of monad definitions in the operational semantics matches the corresponding declarations in the type system:

$$\frac{}{\vdash \cdot : \cdot}\ (\text{DT-EMPTY})$$

$$\frac{\vdash \Phi : \Sigma \qquad \vdash_\Sigma N_{\mathrm{u}} : \alpha_1 \to \sigma_\varepsilon[\alpha_1/\alpha] \qquad \vdash_\Sigma N_{\mathrm{b}} : \cdots}{\vdash (\Phi, \varepsilon = (N_{\mathrm{u}}, N_{\mathrm{b}}), e \prec \varepsilon) : (\Sigma, \varepsilon \sim \alpha.\sigma_\varepsilon, e \prec \varepsilon)}\ (\text{DT-EFF})$$

A closed term (or program) is said to be *stuck* if it is not canonical (or finished), but cannot be further reduced by any rule. The progress theorem says that well-typed terms and programs are never stuck:

THEOREM 3.3 (Progress). *Suppose* $\vdash \Phi : \Sigma$. *Then*

1. *If* $\cdot \vdash_\Sigma N : \sigma$, *then either* N *is canonical or there exists an* N' *such that* $\vdash_\Phi N \to N'$.
2. *If* $\vdash_\Sigma P$, *then either* P *is finished or there exists a* P' *such that* $\vdash_\Phi P \to P'$.

Proof. Part 1 follows by induction on the typing derivation. The reasoning is somewhat complicated by subtyping: we need canonical-forms lemmas to the effect that, even in the presence of subtyping, an irreducible term of functional type is indeed a lambda-abstraction, etc. Otherwise, the proof is uneventful. We note that the typing rule TC-GLET-REFL ensures that one of the reduction rules RR-REIF-GLET-REFL or RP-REIFY will apply.

Part 2 follows from part 1 by a simple induction on the derivation of $\vdash_\Sigma P$. ■

We can also show that typability is preserved by reduction:

THEOREM 3.4 (Preservation). *Suppose* $\vdash \Phi : \Sigma$. *Then*

1. *If* $\cdot \vdash_\Sigma N : \sigma$ *and* $\vdash_\Phi N \to N'$ *then* $\cdot \vdash_\Sigma N' : \sigma$
2. *If* $\vdash_\Sigma P$ *and* $\vdash_\Phi P \to P'$ *then* $\vdash_\Sigma P'$.

(The theorem also holds for the system without subtyping, i.e., if the typing of N *does not need* T-SUB, *then neither does* N'.*)*

Proof. Part 1 is by induction on the derivation of the reduction relation, with an inner induction on the typing relation. (The other way around is also possible, and involves exactly the same arguments in a different order.) Again most cases are straightforward; the only complications arise from subtyping: for each canonical-form typing rule (TC-LAM, TC-VAL, etc.), we need an inversion lemma to the effect that if the conclusion of the rule is derivable (possibly using subsumption), then so are the premises.

For program-typing preservation we note that PR-LETEFF-VAL preserves typability, because an M of type nat cannot contain any occurrences of the effect ε. ■

Preservation and progress together give us:

COROLLARY 3.5 (Type soundness). *Any well-typed complete program either runs forever, or eventually reduces to a (unique) number.*

3.4 A context formulation

The reduction system in Figure 4 is convenient for formal reasoning, but often does not quite reflect the operational intuition behind the effects. We thus introduce an alternative presentation, which by collapsing certain sequences of reduction steps, captures the connection between syntactically distant effect-invoking and effect-delimiting operations in a single rule.

The reformulated system is shown in Figure 5. The main change is that we have introduced an explicit syntactic class of evaluation contexts E (using curly braces for holes and filling, purely to avoid yet another overloading of square brackets); this allows us to merge all the individual RR-X and RP-X rules into a few more general ones. Note also that the central new reduction rule CR-REIF-CTX-REFL is formulated only in terms of the isolated-reflection operator from Section 2.3.2.

As suggested by the notation, each reduction by $\twoheadrightarrow$ corresponds to one or more reduction steps in the original system:

LEMMA 3.6. *If* $\vdash_\Phi N \twoheadrightarrow N'$ *then* $\vdash_\Phi N \xrightarrow{+} N'$

Proof. For all of the CR-X rules without premises, we use the analogous RR-X rule in the original formulation. The new CR-REIF-VAL also corresponds to RR-REIF-VAL, and the general context rule CR-CTX corresponds to nested uses of the RC-X rules according to the structure of E.

For CR-REIF-CTX-REFL, we use a straightforward induction on the derivation of $\vdash_\Phi \varepsilon < E$ to show first that

$$\vdash_\Phi E\{\hat{\mu}^\varepsilon(N)\} \xrightarrow{*} \mathbf{glet}\; x \Leftarrow \mu^\varepsilon(N).\, E\{\mathbf{val}\, x\}$$

(In the base case, where $E = \{\}$, the two sides are already equal by definition.) From this, the result follows immediately using the RC-REIF and RR-REIF-GLET-REFL rules. ■

4. Examples

In this section we consider a number of familiar examples of monadic effects, to see how they fit into the framework.

4.1 Exceptions

Let us return to the exception example from the Introduction, now in a more formal setting. Let exn be some arbitrarily defined type of exception names (e.g., simply exn = nat). We then construct metalanguage definitions for the syntactic monad components:

$$\begin{aligned} \mathsf{T}^{\mathsf{ex}}(e,\tau) &\equiv \langle e\rangle(\tau + \mathsf{exn}) \\ \mathsf{unit}^{\mathsf{ex}} &\equiv \lambda a.\, \mathbf{val}\; \mathbf{inj}_1\, a \\ \mathsf{bind}^{\mathsf{ex}} &\equiv \lambda t.\, \lambda f.\, \mathbf{glet}\; s \Leftarrow t.\, \mathbf{case}(s, a.f\, a, x.\mathbf{val}\; \mathbf{inj}_2\, x) \end{aligned}$$

Note that this is really defining a *monad transformer*, not just a single monad, because the base effect e can be chosen arbitrarily. Fixing this base effect to some e_0, possibly just $\bot$, we can then introduce the effect *ex* as follows:

$$\mathbf{leteffect}\; ex \succ e_0\; \mathbf{be}\; (\alpha.\mathsf{T}^{\mathsf{ex}}(e_0,\alpha), \mathsf{unit}^{\mathsf{ex}}, \mathsf{bind}^{\mathsf{ex}})\; \mathbf{in}\; \ldots$$

The effect definition sets up the infrastructure for working with exceptions. The next step is to define the effect operations themselves:

$$\begin{aligned} \mathsf{raise}\; M &\equiv \hat{\mu}^{ex}(\mathbf{val}\; \mathbf{inj}_2\, M) \\ \mathsf{try}\; N_1\; \mathsf{with}\; N_2 &\equiv \mathbf{glet}\; s \Leftarrow [N_1]^{ex}.\, \mathbf{case}(s, a.\mathbf{val}\; a, x.N_2\, x) \end{aligned}$$

(The M in raise is a value, not a computation; to determine the exception name itself as the result of a computation, we would write $\mathbf{glet}\; n \Leftarrow N.\, \mathsf{raise}\; n$.)

Again, raise constructs the monadic representation of a raised exception as the trivial computation of a right-tagged exception name, and reflects it. Conversely, try reifies the opaque representation of the protected expression, evaluates it (thus performing any e_0-effects it might have), and inspects the result: if it is left-tagged, the value is just returned, and N_2 is ignored; otherwise, we apply the handler function N_2 to the exception name.

For these abbreviations, the system of Figure 2 lets us derive the following sound typing rules:

$$\frac{\Gamma \vdash_\Sigma M : \mathsf{exn}}{\Gamma \vdash_\Sigma \mathsf{raise}\; M : \langle ex\rangle\tau}$$

$$\frac{\Gamma \vdash_\Sigma N_1 : \langle ex\rangle\tau \quad \Gamma \vdash_\Sigma N_2 : \mathsf{exn} \to \langle e\rangle\tau \quad \vdash_\Sigma e_0 \leq e}{\Gamma \vdash_\Sigma \mathsf{try}\; N_1\; \mathsf{with}\; N_2 : \langle e\rangle\tau}$$

The underlying value type τ of the protected expression and the handler must be the same (possibly achieved through subsumption), but they may have different effects. In particular, if the handler only has e_0-effects, so will the whole try-expression.

Looking at the dynamic semantics, we see that try $\{\}$ with N_2 is an evaluation context, so we have the derived rule:

$$\frac{\vdash_\Phi N_1 \to N_1'}{\vdash_\Phi \mathsf{try}\; N_1\; \mathsf{with}\; N_2 \to \mathsf{try}\; N_1'\; \mathsf{with}\; N_2}$$

Evaluation contexts

$$E ::= \{\} \mid \mathbf{glet}\; x \Leftarrow E.\, N \mid E\, M \mid \mathbf{prj}_i\, E \mid [E]^{\varepsilon}$$

e-evaluation contexts $\boxed{\vdash_\Phi e < E}$

$$\frac{}{\vdash_\Phi e < \{\}}\;(\text{E-HOLE}) \qquad \frac{\vdash_\Phi e < E}{\vdash_\Phi e < \mathbf{glet}\; x \Leftarrow E.\, N}\;(\text{E-GLET}) \qquad \frac{\vdash_\Phi e < E}{\vdash_\Phi e < E\, M}\;(\text{E-APP})$$

$$\frac{\vdash_\Phi e < E}{\vdash_\Phi e < \mathbf{prj}_i\, E}\;(\text{E-PRJ}) \qquad \frac{\vdash_\Phi e < E \qquad \vdash_\Phi e < \varepsilon}{\vdash_\Phi e < [E]^{\varepsilon}}\;(\text{E-REIF})$$

Contextual reduction $\boxed{\vdash_\Phi N \twoheadrightarrow N'}$

$$\frac{}{\vdash_\Phi \mathbf{glet}\; x \Leftarrow \mathbf{val}\, M.\, N \twoheadrightarrow N[M/x]}\;(\text{CR-GLET-VAL}) \qquad \frac{}{\vdash_\Phi (\lambda x.\, N)\, M \twoheadrightarrow N[M/x]}\;(\text{CR-APP-LAM})$$

$$\frac{}{\vdash_\Phi \mathbf{prj}_i\, \langle N_1, N_2\rangle \twoheadrightarrow N_i}\;(\text{CR-PRJ-PAIR}) \qquad \frac{}{\vdash_\Phi \mathbf{split}((M_1, M_2), x_1.x_2.N) \twoheadrightarrow N[M_1/x_1, M_2/x_2]}\;(\text{CR-SPLIT-PAIR})$$

$$\frac{}{\vdash_\Phi \mathbf{case}(\mathbf{inj}_i\, M, x_1.N_1, x_2.N_2) \twoheadrightarrow N_i[M/x_i]}\;(\text{CR-CASE-INJ}) \qquad \frac{}{\vdash_\Phi \mathbf{unroll}\, (\mathbf{roll}\, M, x.N) \twoheadrightarrow N[M/x]}\;(\text{CR-UNROLL-ROLL})$$

$$\frac{}{\vdash_\Phi \mathbf{fix}\, x.N \twoheadrightarrow N[(\mathbf{fix}\, x.N)/x]}\;(\text{CR-FIX})$$

$$\frac{(\varepsilon = (N_\mathrm{u}, N_\mathrm{b})) \in \Phi}{\vdash_\Phi [\mathbf{val}\, M]^{\varepsilon} \twoheadrightarrow N_\mathrm{u}\, M}\;(\text{CR-REIF-VAL}) \qquad \frac{(\varepsilon = (N_\mathrm{u}, N_\mathrm{b})) \in \Phi \qquad \vdash_\Phi \varepsilon < E}{\vdash_\Phi [E\{\hat\mu^{\varepsilon}(N)\}]^{\varepsilon} \twoheadrightarrow N_\mathrm{b}\, N\, (\lambda x.\, [E\{\mathbf{val}\, x\}]^{\varepsilon})}\;(\text{CR-REIF-CTX-REFL})$$

$$\frac{\vdash_\Phi N \twoheadrightarrow N'}{\vdash_\Phi E\{N\} \twoheadrightarrow E\{N'\}}\;(\text{CR-CTX})$$

Figure 5. Reduction rules (context formulation)

Further,

$$\begin{aligned}
&\mathsf{try}\; \mathbf{val}\, M\; \mathsf{with}\; N_2\\
&\quad = \mathbf{glet}\; s \Leftarrow [\mathbf{val}\, M]^{ex}.\, \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to \mathbf{glet}\; s \Leftarrow \mathsf{unit}^{ex}\, M.\, \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to \mathbf{glet}\; s \Leftarrow \mathbf{val}\, \mathbf{inj}_1\, M.\, \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to \mathbf{case}(\mathbf{inj}_1\, M, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to \mathbf{val}\, M
\end{aligned}$$

That is, we have a derived rule:

$$\frac{}{\vdash_\Phi \mathsf{try}\; \mathbf{val}\, M\; \mathsf{with}\; N_2 \xrightarrow{+} \mathbf{val}\, M}$$

Now, suppose $\vdash_\Phi\ ex < E$, so that in particular E does not itself include an inner try as part of the evaluation context. Then,

$$\begin{aligned}
&\mathsf{try}\; E\{\mathsf{raise}\; M\}\; \mathsf{with}\; N_2\\
&\quad = \mathbf{glet}\; s \Leftarrow [E\{\hat\mu^{ex}(\mathbf{val}\, \mathbf{inj}_2\, M)\}]^{ex}.\\
&\qquad\quad \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \twoheadrightarrow \mathbf{glet}\; s \Leftarrow \mathsf{bind}^{ex}\, (\mathbf{val}\, \mathbf{inj}_2\, M)\, (\lambda r.\, [E\{\mathbf{val}\, r\}]^{ex}).\\
&\qquad\quad \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \xrightarrow{2} \mathbf{glet}\; s \Leftarrow (\mathbf{glet}\; s \Leftarrow \mathbf{val}\, \mathbf{inj}_2\, M.\\
&\qquad\qquad\qquad \mathbf{case}(s, a.(\lambda r.\, [E\{\mathbf{val}\, r\}]^{ex})\, a, x.\mathbf{val}\, \mathbf{inj}_2\, x)).\\
&\qquad\quad \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to \mathbf{glet}\; s \Leftarrow \mathbf{case}(\mathbf{inj}_2\, M, a.(\lambda r.\, [E\{\mathbf{val}\, r\}]^{ex})\, a, x.\mathbf{val}\, \mathbf{inj}_2\, x).\\
&\qquad\quad \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to \mathbf{glet}\; s \Leftarrow \mathbf{val}\, \mathbf{inj}_2\, M.\, \mathbf{case}(s, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to \mathbf{case}(\mathbf{inj}_2\, M, a.\mathbf{val}\, a, x.N_2\, x)\\
&\quad \to N_2\, M
\end{aligned}$$

Here E represents the local evaluation context of the raise. Note that it gets discarded already by the bind that occurs as part of the semantics of reification, not by the definition of try. Indeed, with the chosen definition of the monad, there is no way to express an exception-handling construct that can access the local context of the raise (for example to resume it). This is exactly what we would expect from the realization type: if the reified meaning of a computation is just a right-tagged exception name, there is no context available to be inspected or resumed. In summary,

$$\frac{\vdash_\Phi ex < E}{\vdash_\Phi \mathsf{try}\; E\{\mathsf{raise}\; M\}\; \mathsf{with}\; N_2 \xrightarrow{+} N_2\, M}$$

Note that $\mathsf{raise}\; M$ only reduces meaningfully in the context of a try. And in fact the typing rules enforce this: the top-level effect must be $\bot$, so in a well-typed program, there can be no unguarded exception-computations.

Although we do not typically expose the general reflect and reify operations for exceptions directly to the programmer, it is instructive to consider one particular instance: what if we apply $\hat\mu^{ex}(\cdot)$ to the transparent representation of an effect-free computation returning M, i.e., $\mathbf{val}\, \mathbf{inj}_1\, M$? We calculate:

$$\begin{aligned}
&[E\{\hat\mu^{ex}(\mathbf{val}\, \mathbf{inj}_1\, M)\}]^{ex}\\
&\quad \twoheadrightarrow \mathsf{bind}^{ex}\, (\mathbf{val}\, \mathbf{inj}_1\, M)\, (\lambda x.\, [E\{\mathbf{val}\, x\}]^{ex})\\
&\quad \xrightarrow{2} \mathbf{glet}\; s \Leftarrow \mathbf{val}\, \mathbf{inj}_1\, M.\\
&\qquad\quad \mathbf{case}(s, a.(\lambda x.\, [E\{\mathbf{val}\, x\}]^{ex})\, a, e.\mathbf{val}\, \mathbf{inj}_2\, e)\\
&\quad \to \mathbf{case}(\mathbf{inj}_1\, M, a.(\lambda x.\, [E\{\mathbf{val}\, x\}]^{ex})\, a, e.\mathbf{val}\, \mathbf{inj}_2\, e)\\
&\quad \xrightarrow{2} [E\{\mathbf{val}\, M\}]^{ex}
\end{aligned}$$

That is, in any evaluation context anchored by $[\cdot]^{ex}$, an occurrence of $\hat\mu^{ex}(\mathsf{unit}^{ex}\, M)$ behaves same as just $\mathbf{val}\, M$. This is no accident, of course; it follows from the fact that the exception monad satisfies the monad laws (specifically, $\mathsf{bind}\; (\mathsf{unit}\; M)N = N\, M$. In other words, though the operational semantics is well defined for "effects" determined by any pair of terms N_u and N_b of the right types, we must require them to satisfy the monad laws as at least observational equivalences, if we want an agreement between the translational and the operational views of the effect.

4.2 State

In the exception monad, the local context at the point of reflection played no role. For most monadic effects, though, that context is meant to be resumed with the result of the effectful operation. The prime example of such an effect is state.

Consider the components of the state monad: Let state be some type and take

$$\begin{aligned}
\mathsf{T}^{\mathsf{st}}(e,\tau) &\equiv \mathsf{state} \to \langle e\rangle(\tau \times \mathsf{state})\\
\mathsf{unit}^{\mathsf{st}} &\equiv \lambda a.\,\lambda s.\,\mathbf{val}\,(a,s)\\
\mathsf{bind}^{\mathsf{st}} &\equiv \lambda t.\,\lambda f.\,\lambda s.\,\mathbf{glet}\,(a,s') \Leftarrow t\,s.\,f\,a\,s'
\end{aligned}$$

We write $\mathbf{glet}\,(x,y) \Leftarrow N.\,N'$ as a straightforward abbreviation for $\mathbf{glet}\,p \Leftarrow N.\,\mathbf{split}(p,x.y.N')$, where $p \notin FV(N')$.

Again, we introduce the effect,

$$\mathbf{leteffect}\ \mathit{st} \succ e_0\ \mathbf{be}\ (\alpha.\mathsf{T}^{\mathsf{st}}(e_0,\alpha), \mathsf{unit}^{\mathsf{st}}, \mathsf{bind}^{\mathsf{st}})\ \mathbf{in}\ \ldots$$

with companion abbreviations:

$$\begin{aligned}
\mathsf{withst}\ M\ \mathsf{do}\ N &\equiv \mathbf{glet}\,(a,s) \Leftarrow [N]^{\mathit{st}}\,M.\,\mathbf{val}\,a\\
\mathsf{getst} &\equiv \hat{\mu}^{\mathit{st}}(\lambda s.\,\mathbf{val}\,(s,s))\\
\mathsf{setst}\ M &\equiv \hat{\mu}^{\mathit{st}}(\lambda s.\,\mathbf{val}\,((),M)) \quad {\scriptstyle (s \notin FV(M))}
\end{aligned}$$

We derive the following type rules for the constructs:

$$\frac{\Gamma \vdash_\Sigma M : \mathsf{state} \quad \Gamma \vdash_\Sigma N : \langle \mathit{st}\rangle\tau}{\mathsf{withst}\ M\ \mathsf{do}\ N : \langle e_0\rangle\tau}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathsf{getst} : \langle \mathit{st}\rangle\mathsf{state}} \qquad \frac{\Gamma \vdash_\Sigma M : \mathsf{state}}{\Gamma \vdash_\Sigma \mathsf{setst}\ M : \langle \mathit{st}\rangle 1}$$

Operationally, we see that $\mathsf{withst}\ M\ \mathsf{do}\ \{\}$ is an evaluation context, so we obtain the following derived reduction rule:

$$\frac{\vdash_\Phi N \to N'}{\vdash_\Phi \mathsf{withst}\ M\ \mathsf{do}\ N \to \mathsf{withst}\ M\ \mathsf{do}\ N'}$$

We also have,

$$\begin{aligned}
&\mathsf{withst}\ M_1\ \mathsf{do}\ \mathbf{val}\ M_2\\
&\ = \mathbf{glet}\,(a,s) \Leftarrow [\mathbf{val}\ M_2]^{\mathit{st}}\,M_1.\,\mathbf{val}\,a\\
&\ \to \mathbf{glet}\,(a,s) \Leftarrow \mathsf{unit}^{\mathsf{st}}\,(\mathbf{val}\ M_2)\,M_1.\,\mathbf{val}\,a\\
&\ \xrightarrow{2} \mathbf{glet}\,(a,s) \Leftarrow \mathbf{val}\,(M_2,M_1).\,\mathbf{val}\,a\\
&\ \xrightarrow{2} \mathbf{val}\ M_2
\end{aligned}$$

which we summarize as:

$$\frac{}{\vdash_\Phi \mathsf{withst}\ M_1\ \mathsf{do}\ \mathbf{val}\ M_2 \xrightarrow{+} \mathbf{val}\ M_2}$$

Consider now what happens when getst occurs dynamically within a withst. Again, let $\vdash_\Phi \mathit{st} < E$, so that E does not contain an inner withst; then,

$$\begin{aligned}
&\mathsf{withst}\ M\ \mathsf{do}\ E\{\mathsf{getst}\}\\
&\ = \mathbf{glet}\,(a,s) \Leftarrow [E\{\hat{\mu}^{\mathit{st}}(\lambda s.\,\mathbf{val}\,(s,s))\}]^{\mathit{st}}\,M.\,\mathbf{val}\,a\\
&\ \twoheadrightarrow \mathbf{glet}\,(a,s) \Leftarrow \mathsf{bind}^{\mathsf{st}}\,(\lambda s.\,\mathbf{val}\,(s,s))\,(\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,M.\\
&\qquad \mathbf{val}\,a\\
&\ \xrightarrow{3} \mathbf{glet}\,(a,s) \Leftarrow (\mathbf{glet}\,(a,s') \Leftarrow (\lambda s.\,\mathbf{val}\,(s,s))\,M.\\
&\qquad\qquad\qquad (\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,a\,s').\\
&\qquad \mathbf{val}\,a\\
&\ \to \mathbf{glet}\,(a,s) \Leftarrow (\mathbf{glet}\,(a,s') \Leftarrow \mathbf{val}\,(M,M).\\
&\qquad\qquad\qquad (\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,a\,s').\\
&\qquad \mathbf{val}\,a\\
&\ \xrightarrow{2} \mathbf{glet}\,(a,s) \Leftarrow (\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,M\,M.\,\mathbf{val}\,a\\
&\ \to \mathbf{glet}\,(a,s) \Leftarrow [E\{\mathbf{val}\,M\}]^{\mathit{st}}\,M.\,\mathbf{val}\,a\\
&\ = \mathsf{withst}\ M\ \mathsf{do}\ E\{\mathbf{val}\ M\}
\end{aligned}$$

That is, the call to getst gets replaced by the value representing the current state, but the evaluation context remains unchanged.

Completely analogously, for setst we calculate:

$$\begin{aligned}
&\mathsf{withst}\ M\ \mathsf{do}\ E\{\mathsf{setst}\ M'\}\\
&\ = \mathbf{glet}\,(a,s) \Leftarrow [E\{\hat{\mu}^{\mathit{st}}(\lambda s.\,\mathbf{val}\,((),M'))\}]^{\mathit{st}}\,M.\,\mathbf{val}\,a\\
&\ \twoheadrightarrow \mathbf{glet}\,(a,s) \Leftarrow \mathsf{bind}^{\mathsf{st}}\,(\lambda s.\,\mathbf{val}\,((),M'))\,(\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,M.\\
&\qquad \mathbf{val}\,a\\
&\ \xrightarrow{3} \mathbf{glet}\,(a,s) \Leftarrow (\mathbf{glet}\,(a,s') \Leftarrow (\lambda s.\,\mathbf{val}\,((),M'))\,M.\\
&\qquad\qquad\qquad (\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,a\,s').\\
&\qquad \mathbf{val}\,a\\
&\ \to \mathbf{glet}\,(a,s) \Leftarrow (\mathbf{glet}\,(a,s') \Leftarrow \mathbf{val}\,((),M').\\
&\qquad\qquad\qquad (\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,a\,s').\\
&\qquad \mathbf{val}\,a\\
&\ \xrightarrow{2} \mathbf{glet}\,(a,s) \Leftarrow (\lambda x.\,[E\{\mathbf{val}\,x\}]^{\mathit{st}})\,()\,M'.\,\mathbf{val}\,a\\
&\ \to \mathbf{glet}\,(a,s) \Leftarrow [E\{\mathbf{val}\,()\}]^{\mathit{st}}\,M'.\,\mathbf{val}\,a\\
&\ = \mathsf{withst}\ M'\ \mathsf{do}\ E\{\mathbf{val}\,()\}
\end{aligned}$$

Here, the call to $\mathsf{setst}\ M'$ gets replaced by simply $\mathbf{val}\,()$, but the surrounding context gets modified to now report the state as M' for any future calls to getst, very much like in a specialized theory for state [3]. Summarizing, we can complete our derived reduction rules with

$$\frac{\vdash_\Phi \mathit{st} < E}{\vdash_\Phi \mathsf{withst}\ M\ \mathsf{do}\ E\{\mathsf{getst}\} \xrightarrow{+} \mathsf{withst}\ M\ \mathsf{do}\ E\{\mathbf{val}\ M\}}$$

$$\frac{\vdash_\Phi \mathit{st} < E}{\vdash_\Phi \mathsf{withst}\ M\ \mathsf{do}\ E\{\mathsf{setst}\ M'\} \xrightarrow{+} \mathsf{withst}\ M'\ \mathsf{do}\ E\{\mathbf{val}\,()\}}$$

4.3 Exceptions and state

If we want to write programs with both exceptions and state, we can simply use the above two effect-definitions together; in particular, the derived reduction rules remain valid. We must still make a choice, however, about how to order the effects.

We specify an ML-like semantics, with state persisting across raised exceptions, as follows:

$$\begin{aligned}
&\mathbf{leteffect}\ \mathit{st} \succ \bot\ \mathbf{be}\ (\alpha.\mathsf{T}^{\mathsf{st}}(\bot,\alpha), \mathsf{unit}^{\mathsf{st}}, \mathsf{bind}^{\mathsf{st}})\ \mathbf{in}\\
&\quad \mathbf{leteffect}\ \mathit{ex} \succ \mathit{st}\ \mathbf{be}\ (\alpha.\mathsf{T}^{\mathsf{ex}}(\mathit{st},\alpha), \mathsf{unit}^{\mathsf{ex}}, \mathsf{bind}^{\mathsf{ex}})\ \mathbf{in}\ \ldots
\end{aligned}$$

In this ordering, the program will typically contain a single, outermost withst, representing the global state, and serving as an anchor for all *st*-reflections. Such reflections propagate freely through try-terms (*ex*-reifications), as we would expect. However, in a type-correct program, any raised exceptions must be caught before they can reach the withst, if only by a catch-all handler that reports a top-level uncaught exception. Expanding the effect definitions, we see that the constructor associated with exceptions-and-state computations is given by $\mathsf{T}\alpha = \mathsf{state} \to \langle\bot\rangle((\alpha + \mathsf{exn}) \times \mathsf{state})$.

Had we instead introduced the effects in the opposite order, with *st* layered above *ex*, we would get a "transactional" semantics, where the current state is lost when an exception is signaled. That is, the combined type constructor modeling computations would become $\mathsf{T}\alpha = \mathsf{state} \to \langle\bot\rangle((\alpha \times \mathsf{state}) + \mathsf{exn})$. In this variant, raised exceptions can freely propagate out of withst, but the semantics no longer prescribes a behavior for when a state access occurs directly within a try.

It may be tempting to relax the condition $\varepsilon < \varepsilon'$ in RP-REIF to $\varepsilon \neq \varepsilon'$, allowing all reflections to commute with other reifications. This often happens to give a sensible operational behavior. However, the price we pay for such an "anarchic" combination of individually specified monadic effects is that we can no longer explain the combination of exceptions and state translationally using either of the two above versions of T, and so we lose the equivalence principle between opaque and transparent representations, making sound reasoning about the system significantly harder (cf. [13]).

4.4 Continuations

As a final example, let us consider first-class continuations. Let ans be a suitable type of final answers. The monad components can then be taken as

$$\begin{aligned} \mathsf{T}^{\mathsf{ct}}(e,\alpha) &\equiv (\alpha \to \langle e\rangle \mathsf{ans}) \to \langle e\rangle\mathsf{ans} \\ \mathsf{unit}^{\mathsf{ct}} &\equiv \lambda a.\, \lambda k.\, k\, a \\ \mathsf{bind}^{\mathsf{ct}} &\equiv \lambda t.\, \lambda f.\, \lambda k.\, t\,(\lambda a.\, f\, a\, k) \end{aligned}$$

For any base effect e_0, we then introduce the effect of ans-continuations with e_0-effects:

$$\mathbf{leteffect}\ \mathit{ct} \succ e_0\ \mathbf{be}\ (\alpha.\mathsf{T}^{\mathsf{ct}}(e_0,\alpha), \mathsf{unit}^{\mathsf{ct}}, \mathsf{bind}^{\mathsf{ct}})\ \mathbf{in}\ \ldots$$

In the scope of this declaration, we can introduce abbreviations

$$\begin{aligned} \mathsf{cont}(\tau) &\equiv \tau \to \langle e_0\rangle\mathsf{ans} \\ \#N &\equiv [N]^{ct}\,(\lambda r.\, \mathbf{val}\, r) \\ \mathsf{letcc}\ k.N &\equiv \hat{\mu}^{ct}(\lambda k.\, [N]^{ct}\, k) \\ \mathsf{throw}\ M\ \mathsf{to}\ K &\equiv \hat{\mu}^{ct}(\lambda k'.\, K\, M) \quad {\scriptstyle (k' \notin FV(K) \cup FV(M))} \end{aligned}$$

Note that, erasing the reify/reflect operations, the above is exactly how we could define prompts, letcc, and throw in a continuation-passing translation. Their derivable typing rules are:

$$\frac{\Gamma \vdash_\Sigma N : \langle \mathit{ct}\rangle\mathsf{ans}}{\Gamma \vdash_\Sigma \#N : \langle e_0\rangle\mathsf{ans}}$$

$$\frac{\Gamma, k{:}\,\mathsf{cont}(\tau) \vdash_\Sigma N : \langle \mathit{ct}\rangle\tau}{\Gamma \vdash_\Sigma \mathsf{letcc}\ k.N : \langle \mathit{ct}\rangle\tau} \qquad \frac{\Gamma \vdash_\Sigma K : \mathsf{cont}(\tau) \quad \Gamma \vdash_\Sigma M : \tau}{\Gamma \vdash_\Sigma \mathsf{throw}\ M\ \mathsf{to}\ K : \langle \mathit{ct}\rangle\tau'}$$

We immediately see that body of a prompt is an evaluation context, and easily derive the usual cleanup rule:

$$\frac{}{\vdash_\Phi \#\mathbf{val}\, M \xrightarrow{+} \mathbf{val}\, M}$$

We also have:

$$\begin{aligned} &\#E\{\mathsf{throw}\ M\ \mathsf{to}\ K\} \\ &\quad = [E\{\hat{\mu}^{ct}(\lambda k'.\, K\, M)\}]^{ct}\,(\lambda r.\, \mathbf{val}\, r) \\ &\quad \twoheadrightarrow \mathsf{bind}^{\mathsf{ct}}\,(\lambda k.\, K\, M)\,(\lambda x.\, [E\{\mathbf{val}\, x\}]^{ct})\,(\lambda r.\, \mathbf{val}\, r) \\ &\quad \xrightarrow{3} (\lambda k'.\, K\, M)\,(\lambda a.\, (\lambda x.\, [E\{\mathbf{val}\, x\}]^{ct})\, a\,(\lambda r.\, \mathbf{val}\, r)) \\ &\quad \to K\, M \end{aligned}$$

That is, a throw replaces the entire current evaluation context (including the anchoring prompt) with the one obtained from K.

Conversely, for letcc we get:

$$\begin{aligned} &\#E\{\mathsf{letcc}\ k.N\} \\ &\quad = [E\{\hat{\mu}^{ct}(\lambda k.\, [N]^{ct}\, k)\}]^{ct}\,(\lambda r.\, \mathbf{val}\, r) \\ &\quad \twoheadrightarrow \mathsf{bind}^{\mathsf{ct}}\,(\lambda k.\, [N]^{ct}\, k)\,(\lambda x.\, [E\{\mathbf{val}\, x\}]^{ct})\,(\lambda r.\, \mathbf{val}\, r) \\ &\quad \xrightarrow{3} (\lambda k'.\, [N]^{ct}\, k)\,(\lambda a.\, (\lambda x.\, [E\{\mathbf{val}\, x\}]^{ct})\, a\,(\lambda r.\, \mathbf{val}\, r)) \\ &\quad \xrightarrow{\sim} (\lambda k.\, [N]^{ct}\, k)\,(\lambda a.\, [E\{\mathbf{val}\, a\}]^{ct}\,(\lambda r.\, \mathbf{val}\, r)) \\ &\quad = (\lambda k.\, [N]^{ct}\, k)\,(\lambda a.\, \#E\{\mathbf{val}\, a\}) \\ &\quad \to [N[(\lambda a.\, \#E\{\mathbf{val}\, a\})/k]]^{ct}\,(\lambda a.\, \#E\{\mathbf{val}\, a\}) \end{aligned}$$

In the transition marked with a $\sim$, we have cheated slightly, by eagerly contracting a trivial beta-redex, which in reality would only happen once the continuation was actually applied. Even so, the reduced term is evidently no longer in the fragment spanned by our specialized continuation primitives, because it uses reification with a non-empty continuation. We can certainly continue reducing it further using the general rules for $\hat{\mu}^{ct}()$ and $[]^{ct}$, so the semantics remains correct. However, in this particular case, we could instead have defined the abbreviation letcc slightly more verbosely:

$$\mathsf{letcc}\ k.N \equiv \hat{\mu}^{ct}(\lambda k.\, [\mathbf{glet}\ x \Leftarrow N.\, k\, x]^{ct}\,(\lambda r.\, \mathbf{val}\, r)) \quad {\scriptstyle (x \neq k)}$$

This is denotationally equivalent to the original one (in the sense that their monadic translations are $\beta\eta$-equivalent in the base language), but has a shape that is closed under reduction. We get

$$\frac{}{\begin{aligned}&\#E\{\mathsf{letcc}\ k.N\} \xrightarrow{+} \\ &\quad \#\mathbf{glet}\ x \Leftarrow N[((\lambda x.\, \#E\{\mathbf{val}\, x\}))/k].\, \#E\{\mathbf{val}\, x\}\end{aligned}}$$

Incidentally, using another denotationally valid equation,

$$\#\mathbf{glet}\ x \Leftarrow N'.\, \#E\{\mathbf{val}\, x\} \;=\; \#E\{N'\}$$

the result of the reduction is equivalent to

$$\#E\{N[(\lambda x.\, \#E\{\mathbf{val}\, x\})/k]\}$$

which one would probably have written as the intended reduct of $\#E\{\mathsf{letcc}\ k.N\}$, if defining the operational semantics directly.

5. Related work

The metalanguage used to host the monadic reflection operators was first introduced in a purely denotational setting for relating monadic semantics [5], although its roots go back to at least [4]. The single-effect fragment is very close to Levy's Call-By-Push-Value metalanguage [10], at least with respect to the operational and denotational semantics (the syntaxes are less similar). However, CBPV primarily uses the notion of computation types to faithfully encode PCF- or Algol-like call-by-name semantics, whereas our focus is on incremental definition of effects. In particular, the syntax and semantics of CBPV include nothing like the generic monadic reflection operators; instead, the semantics of a language with any fixed notion of effects is defined in terms of a customized reduction relation, with specialized rules for the effect-operations.

There has been a fair amount of work on generic operational semantics for effects and related constructs. Some, like ours, is based on assigning operational interpretations to denotational or categorical constructions. In particular, there is long line of research by Plotkin and Power (e.g., [18]) on relating the operational and categorical formulations of so-called algebraic effect operations – roughly, the reflect-like ones in our terminology. This work has recently been extended to also account for some effect-delimiting operations – i.e., reify-like ones – such as exception handlers [19], but it does not yet cover the full range of monadic effects familiar from functional programming. On the other hand, it seems well suited for modeling several important effects that are not even conceptually definable within the language, such as true nondeterminism, as interpreted using a powerdomain.

Other investigations start directly with an operational view, and propose systems apparently general enough to model most effects of common interest. This includes in particular Mosses's Modular SOS [14], based on labelled transition systems. Another interesting approach is Pfenning's destination-passing style [15]. Both give several examples of how common effects can be represented, but it seems hard to delimit the exact spectrum of computations expressible in those frameworks, or to relate the encodings to the otherwise very successful denotational/monadic models of the same effects.

There is also a successful tradition of modeling a variety of computational effects in primarily functional languages, in terms of evaluation-context manipulation [3], either by custom reduction rules for particular effect-operations, or by exposing the context-manipulating power to the programmer through sufficiently powerful general control operators (e.g., [8, 9]). For several individual effects, the resulting operational rules end up quite similar to our monad-based ones, but a significant conceptual difference is that our model is ultimately based on a hierarchical composition of effects, whereas most continuation-based ones take a more liberal, "flat" approach, imposing no particular ordering on the components being combined, as outlined in Section 4.3.

A particularly intriguing example of such a flat composition of effects is the early work by Cartwright and Felleisen on extensible denotational semantics [1]. Though the pattern of splitting the semantics of effect-like language features into "handlers" and "requests" is very similar to ours, their presentation is ultimately about a domain-theoretic presentation of an operational idea – whereas we give an operational account of language constructs originally motivated by a denotational/categorical semantics.

While notions of effects based directly on continuations or evaluation contexts are evidently at least as expressive as the one proposed here (indeed, the representation of layered monads in terms of delimited continuations [4] uses a particularly restricted continuation-manipulating operator), this expressivity comes at a price: there is in general no simple way of characterizing the extent to which a term does *not* exploit the full power of context manipulation, and therefore can be reasoned about in ways that are unsound for terms with unrestricted effects. We propose the monadic-reflection abstraction as precisely such a characterization.

6. Conclusions and future work

Though monadic effects are usually thought of in denotational terms, and in an inherently typed, Church-style setting, our results show that, somewhat surprisingly, they can also be given a purely Curry-style semantics. That is, the generic operations for invoking and delimiting effects have an intrinsic operational meaning, given by an simple, effect-independent notion of reduction, and defined by a small, fixed collection of rules. Yet, in concrete examples, these fixed rules closely agree with how one would define the effect "by hand". It is hoped that the results, and natural extensions of them, can be used to strengthen the interplay of operational and denotational techniques for reasoning about programs with effects.

The framework presented here allows us to understand a monadic effect in two complementary ways: either we fix the meanings of the effectful operations to be their monad-based definitions, and translate the program around them to fit those meanings; or we fix the semantics of programs, and transform the monad-based definitions of the operations into specialized operational rules that can be merged directly into the fixed evaluation judgment. Work certainly remains to be done in formally relating the two views, but already the results and examples presented here should illustrate the viability and utility of the basic principle.

Further topics to explore include practical effect-type inference for the system, as well as extensions with various notions of polymorphism. In particular, one might consider finer notions of subeffecting, such as letting the exception monad be parametrized by an explicit set of exception names, to keep precise track of which exceptions are raised or handled where. Another useful addition would be a notion of typed dynamic allocation as a primitive base effect, to allow sound modeling of dynamically created ref-cells of arbitrary types. The operational semantics, being type-free, should not need any significant changes or extensions, but the translational/denotational account would need a refinement to possible-world or functor-category semantics.

Finally, it is a little disconcerting that, though the operational semantics itself is reduction-based, its ultimate justification is still denotational, in that the monad laws (and their extensions to monad morphisms, algebras, etc.) all talk about equality, not reduction. This dependence may not be easy to eliminate: for more interesting monads, involving type-level recursion (such as streams or search trees), the easiest way to prove the monad laws is apparently by domain-theoretic methods, such as rigid induction [6], which ultimately reduce to reasoning about minimal invariants. Ultimately, however, such principles could probably also be ported to an operational setting [2], making the whole treatment self-contained.

References

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

[2] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic Notes in Theoretical Computer Science*, 172, 2007.

[3] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, September 1992.

[4] Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999.

[5] Andrzej Filinski. On the relations between monadic semantics. *Theoretical Computer Science*, 375(1-3):41–75, 2007.

[6] Andrzej Filinski and Kristian Støvring. Inductive reasoning about effectful data types. In *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 97–110. ACM Press, October 2007.

[7] Marcelo Fiore and Gordon D. Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In *Proceedings of the Ninth Symposium on Logic in Computer Science*, pages 92–102, Paris, France, 1994.

[8] 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.

[9] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. In *ICFP'06: Proceedings of the 11th International Conference on Functional Programming*, pages 26–37, 2006.

[10] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.

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

[12] 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.

[13] Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

[14] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[15] Frank Pfenning. Substructural operational semantics and linear destination-passing style (abstract). In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, volume 3302 of *Lecture Notes in Computer Science*, page 196, 2004.

[16] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, 1999.

[17] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[18] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[19] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *18th European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94, March 2009.

[20] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, June 1990.

[21] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.