Monadic Type-And-Effect Soundness

Francesco Dagnino
DIBRIS, Università di Genova, Italy

Paola Giannini
DiSSTE, Università del Piemonte Orientale, Vercelli, Italy

Elena Zucca
DIBRIS, Università di Genova, Italy

Abstract

We introduce the abstract notions of *monadic operational semantics*, a small-step semantics where computational effects are modularly modeled by a monad, and *type-and-effect system*, including *effect types* whose interpretation lifts well-typedness to its monadic version. In this meta-theory, as usual in the non-monadic case, we can express progress and subject reduction, and provide a proof, given once and for all, that they imply soundness. The approach is illustrated on a lambda calculus with generic effects, equipped with an expressive type-and-effect system We provide proofs of progress and subject reduction, parametric on the interpretation of effect types. In this way, we obtain as instances many significant examples, such as checking exceptions, preventing/limiting non-determinism, constraining order/fairness of outputs. We also provide an extension with constructs to raise and handle computational effects, which can be instantiated to model different policies.

2012 ACM Subject Classification Theory of computation → Operational semantics; Theory of computation → Type structures

Keywords and phrases Effects, monads, type soundness

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.7

Related Version *Full Version*: `https://arxiv.org/abs/2504.10159` [9]

Funding This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the Università del Piemonte Orientale.

1 Introduction

It would be hard to overstate the impact on foundations of programming languages of, on one hand, the idea that computational effects can be modeled by monads [30, 31], and, on the other hand, the technique based on progress and subject reduction to prove the soundness of a type system with respect to a small-step operational semantics [47].

Moggi's seminal work [30, 31] recognized monads as the suitable structure to modularly describe the denotational semantics of effectful languages. The key idea was the distinction between *pure* (effect-free) and *monadic* (effectful) expressions, also called *computations*, the latter getting semantics in a monad. Haskell has firstly[1] shown that such an approach can be fruitfully adopted in a mainstream language, through a monad type constructor allowing to encapsulate effectful code. However, the structure of a monad[2] does not include operations for *raising* effects, which need to be defined ad-hoc in instances. *Algebraic and generic effects* [33, 34, 35], instead, explicitly consider operations to raise effects, interpreted by additional structure on the monad. Such an approach, combined with handlers [37, 38, 4, 40], has been exploited in fully-fledged programming languages, e.g., in Scala and OCaml 5.

[1] Many other languages have then supported a monad pattern, e.g., Scheme, Python, Racket, Scala, F#.
[2] In Haskell, methods of the `Monad` typeclass.


© Francesco Dagnino, Paola Giannini, and Elena Zucca;
licensed under Creative Commons License CC-BY 4.0

To provide guarantees on, besides the result, the computational effects possibly raised by a computation, type systems are generalized to *type-and-effect systems*. A great many of these have been designed for specific calculi, modelling effects by relying on auxiliary structures, e.g., memory in imperative calculi, and providing ad-hoc soundness proofs; Katsumata [25] has provided a unified view of such systems, however based on denotational semantics. In this paper, instead, we provide an *operational meta-theory of monadic type-and-effect soundness*, analogous to the one mentioned above for usual type soundness based on small-step semantics, progress and subject reduction [47]. To this end, we provide abstract notions of small-step monadic semantics, type-and-effect system, and soundness, as detailed below.[3]

Operational semantics. We design a language semantics which is *monadic*, since effects are, as customary, expressed by a monad, and simultaneously *small-step*, since we define sequences of reduction steps. To this end, we start from a reduction from language expressions to monadic expressions (in a given monad) required to be deterministic, and extend such a relation to a *total* function, so to be able to combine steps by Kleisli composition, similarly to the approach in [17]. In this way, reduction sequences are always infinite, so termination is conventionally represented by monadic elements called *results*, which always reduce to themselves without raising any effect. On top of the reduction, we define the *finitary* semantics of an expression, which is either the monadic result reached in many steps, if any, or divergence. This semantics does not describe the computational effects raised by infinite computations. Hence, we define an *infinitary* semantics, obtained, as customary, as the supremum of a chain of approximants, provided that the monad has the necessary structure.

Type-and-effect system. As done in [7, 6] for standard type systems, we abstractly model a *type-and-effect system* as a family of predicates over expressions, indexed by types and *effect types*, statically approximating the computational effects that may be raised during evaluation. Effect types are required to form an ordered monoid, as typically assumed in effect systems [32, 29] and proposed as algebraic structure by [25]. The relation between an effect type and the allowed computational effects is specified by a family of *predicate liftings* [21]. In this way the transformation from a predicate to a monadic one associated to a given effect type is independent from the predicate and its universe.

Soundness. We provide abstract definitions of monadic progress and monadic subject reduction, and a proof, given once and for all, that they imply soundness. The latter means that, if a monadic element is the result of a well-typed expression, then it should be well-typed, that is, satisfy the lifting through the effect type of well-typedness of values.

We illustrate the approach on Λ_Σ, a lambda calculus with generic effects, equipped with an expressive type-and-effect system. We provide proofs of progress and subject reduction parametrically on the interpretation of effect types. In this way, we obtain as instances many significant examples, such as checking exceptions, preventing/limiting non-determinism, constraining order/fairness of outputs on different locations. We also provide an extension with constructs to handle effects, which can be instantiated as well to model different policies.

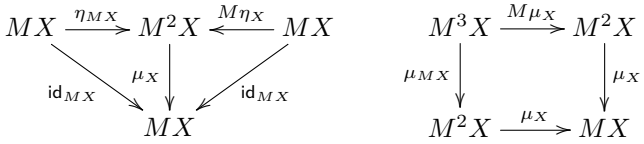
[3] The term "effect" is used in literature both as synonym of computational effect, and in the context of type-and-effect systems, as a static approximation of the former. We will use "effect" when there is no ambiguity, otherwise "computational effect" and "effect type", respectively.

Outline. Section 2 reports the background on monads. Section 3 introduces monadic operational semantics, exemplified through Λ_Σ in Section 4, where we also design a type-and-effect system, discussing its soundness. The approach is formalized by the abstract framework in Section 5; the proof technique introduced there is applied in Section 6 to Λ_Σ. Finally, in Section 7 we enhance the example by handlers, and in Section 8 we discuss related and future work, and summarize the contributions. Omitted proofs can be found in the extended version [9].

2 Preliminaries on monads

Monads [13, 43] are a fundamental notion in category theory, enabling an abstract and unified study of algebraic structures. Since Moggi's seminal papers [30, 31], they have also become a major tool in computer science, especially for describing the semantics of computational effects, and integrating them in programming languages in a structured and principled way. In this section, we recall basic notions about monads, and provide some examples. We will focus on monads on the category of sets and functions, denoted by $\mathcal{Set}$, referring the reader to standard textbooks [41] for a detailed introduction in full generality.

A *monad* $\mathbb{M} = \langle M, \eta, \mu \rangle$ (on $\mathcal{Set}$) consists of a functor $M \colon \mathcal{Set} \to \mathcal{Set}$ and two natural transformations $\eta : \mathsf{Id} \Rightarrow M$ and $\mu : M^2 \Rightarrow M$ such that, for every set X, the following diagrams commute:



The functor M specifies, for every set X, a set MX of monadic elements built over X, in a way that is compatible with functions. The map η_X, named *unit*, embeds elements of X into monadic elements in MX, and the map μ_X, named *multiplication*, flattens monadic elements built on top of other monadic elements into plain monadic elements.

From these data, one can derive an operation on functions of type $X \to MY$, dubbed *Kleisli extension*, which is crucial for modelling computational effects using monads. For all sets X, Y, we have a function $(-)^\dagger \colon (X \to MY) \to (MX \to MY)$, defined by $f^\dagger = \mu_Y \circ Mf$, that is, first we lift f through M to apply it to monadic elements and then we flatten the result using μ_Y. It is easy to see that the operation $(-)^\dagger$ satisfies the following equations for all $f \colon X \to MY$ and $g \colon Y \to MZ$:

$$\eta_X^\dagger = \mathsf{id}_{MX} \qquad f^\dagger \circ \eta_X = f \qquad g^\dagger \circ f^\dagger = (g^\dagger \circ f)^\dagger$$

Actually, a monad can be equivalently specified in the form of a *Kleisli triple* $\langle M, \eta, (-)^\dagger \rangle$ [28], where M is a mapping on sets, η is a family of functions $\eta_X \colon X \to MX$, for every set X, and $(-)^\dagger$ is a family of functions $(-)^\dagger \colon (X \to MY) \to (MX \to MY)$, for all sets X, Y, satisfying the three equations above. In particular we have $\mu_X = \mathsf{id}^\dagger_{MX}$.

Functions of type $X \to MY$ are called *Kleisli functions* and play a special role: they can be regarded as "effectful functions" from X to Y, raising effects described by the monad $\mathbb{M}$. Indeed, from the Kleisli extension, we can define a composition on Kleisli functions, known as Kleisli composition: given $f \colon X \to MY$ and $g \colon Y \to MZ$ we set

$$g * f = g^\dagger \circ f = \mu_Z \circ Mg \circ f$$

Intuitively, $g * f$ applies f followed by g, sequentially composing the effects they may raise. It is immediate to see that Kleisli composition is associative and η_X is the identity Kleisli function on the set X, that is, η_X is the function raising no effects.

We introduce some useful notation, corresponding to standard operations of monadic types in languages, where such types are assigned to expressions with effects. Given $\alpha \in MX$, $f\colon X \to MY$ and $g\colon X \to Y$, we set

$$\begin{array}{ll} - \gg\!= -\colon MX \to (X \to MY) \to MY & \alpha \gg\!= f = f^\dagger(\alpha) \\ \mathsf{map}\colon (X \to Y) \to MX \to MY & \mathsf{map}\, g\, \alpha = Mg(\alpha) \end{array}$$

The operator $\gg\!=$ is also called bind. As its definition shows, it can be seen as an alternative description of the Kleisli extension, where the parameters are taken in inverse order. This view corresponds, intuitively, to the sequential composition of two expressions with effects, where the latter depends on a parameter *bound* to the result of the former. The operator map describes the effect of the functor M on functions. That is, the lifting of function g through M is applied to a monadic value α. Note that bind and map are interdefinable:

$$\alpha \gg\!= f = \mu_Y(\mathsf{map}\, f\, \alpha) \qquad\qquad \mathsf{map}\, g\, \alpha = \alpha \gg\!= (\eta_Y \circ g)$$

Furthermore, we can express Kleisli composition using bind: $(g * f)(x) = f(x) \gg\!= g$.

In the following examples we characterize the monads by defining bind rather than multiplication μ since this is often more insightful, and customary in programming languages.

▶ **Example 1** (Exceptions)**.** Let us fix a set Exc. The monad $\mathbb{E}_{\mathsf{Exc}} = \langle E_{\mathsf{Exc}}, \eta^{\mathbb{E}_{\mathsf{Exc}}}, \mu^{\mathbb{E}_{\mathsf{Exc}}} \rangle$ is given by $E_{\mathsf{Exc}}X = \mathsf{Exc} + X$, and

$$\eta^{\mathbb{E}_{\mathsf{Exc}}}(x) = \iota_2(x) \qquad\qquad \alpha \gg\!= f = \begin{cases} f(x) \text{ if } \alpha = \iota_1(x) \\ \alpha \text{ otherwise } (\alpha = \iota_2(\mathsf{e}) \text{ for some } \mathsf{e} \in \mathsf{Exc}) \end{cases}$$

where $+$ denotes disjoint union (coproduct) and ι_1, ι_2 the left and right injections, respectively. We will omit the reference to the set Exc when it is clear from the context.

▶ **Example 2** (Classical Non-Determinism)**.** The monad $\mathbb{P} = \langle P, \eta^{\mathbb{P}}, \mu^{\mathbb{P}} \rangle$ is given by $PX = \wp(X)$, that is, PX is the set of all subsets of X, and

$$\eta^{\mathbb{P}}(x) = \{x\} \qquad\qquad \alpha \gg\!= f = \bigcup_{x \in \alpha} f(x)$$

A variant of this monad is the list monad $\mathbb{L} = \langle L, \eta^{\mathbb{L}}, \mu^{\mathbb{L}} \rangle$, where the set LX of (possibly infinite) lists over X is coinductively defined by the following rules: $\epsilon \in L(X)$ and, if $x \in X$ and $l \in L(X)$, then $x\colon l \in L(X)$. We use the notation $[x_1, \ldots, x_n]$ to denote the finite list $x_1\colon \ldots\colon x_n\colon \epsilon$. Then, the unit is given by $\eta^{\mathbb{L}}_X(x) = [x]$ and the monadic bind is corecursively defined by the following clauses: $\epsilon \gg\!= f = \epsilon$ and $(x\colon l) \gg\!= f = f(x)(l \gg\!= f)$, where juxtaposition denotes the concatenation of possibly infinite lists.

▶ **Example 3** (Probabilistic Non-Determinism)**.** Denote by DX the set of probability subdistributions α over X with countable support, i.e., $\alpha\colon X \to [0..1]$ with $\sum_{x \in X} \alpha(x) \leq 1$ and $\mathsf{supp}(\alpha) = \{x \in X \mid \alpha(x) \neq 0\}$ countable set. We write $r \cdot \alpha$ for the pointwise multiplication of a subdistribution α with a number $r \in [0, 1]$. The monad $\mathbb{D} = \langle D, \eta^{\mathbb{D}}, \mu^{\mathbb{D}} \rangle$ is given by

$$\eta^{\mathbb{D}}(x) = y \mapsto \begin{cases} 1 & y = x \\ 0 & \text{otherwise} \end{cases} \qquad\qquad \alpha \gg\!= f = \sum_{x \in X} \alpha(x) \cdot f(x)$$

▶ **Example 4** (Output/Writer). Let $\langle \mathsf{Out}, \cdot, \varepsilon \rangle$ be a monoid, e.g., the monoid of strings over a fixed alphabet. The monad $\mathbb{O} = \langle O, \eta^{\mathbb{O}}, \mu^{\mathbb{O}} \rangle$ is given by $OX = \mathsf{Out} \times X$ and

$$\eta^{\mathbb{O}}(x) = \langle \varepsilon, x \rangle \qquad \langle o, x \rangle \gg\!= f = \langle o \cdot \pi_1(f(x)), \pi_2(f(x)) \rangle$$

Combining this monad with the exception monad of Example 1, we obtain the pointed output monad, whose underlying functor is given by $O'X = \mathsf{Out} \times (X + \{\bot\})$.

▶ **Example 5** (Global State). Let S be a set of states. The monad $\mathbb{S} = \langle S, \eta^{\mathbb{S}}, \mu^{\mathbb{S}} \rangle$ is given by $SX = \mathsf{S} \to \mathsf{S} \times X$ and

$$\eta^{\mathbb{S}}(x) = s \mapsto \langle s, x \rangle \qquad \alpha \gg\!= f = s \mapsto f(\pi_2(\alpha(s)))(\pi_1(\alpha(s)))$$

We can combine this monad with the exception monad of Example 1 obtaining $\mathbb{S}_{\mathsf{Exc}} = \langle S_{\mathsf{Exc}}, \eta^{\mathbb{S}_{\mathsf{Exc}}}, \mu^{\mathbb{S}_{\mathsf{Exc}}} \rangle$ where $S_{\mathsf{Exc}} = \mathsf{S} \to (\mathsf{S} \times X) + \mathsf{Exc}$ and $\eta^{\mathbb{S}_{\mathsf{Exc}}}_X(x) = s \mapsto \eta^{\mathbb{E}_{\mathsf{Exc}}}_{\mathsf{S} \times X}(\langle s, x \rangle)$ and $\alpha \gg\!= f = s \mapsto (\alpha(s)) \gg\!=_{\mathbb{E}_{\mathsf{Exc}}} (x \mapsto f(x)(s))$. This combination yields a monad thanks to the fact that $\mathbb{S}$ determines a monad transformer [27, 22].

3 Monadic operational semantics

In this section we abstractly describe a framework for (deterministic) monadic operational semantics, adapting from [16, 17].

▶ **Definition 6.** *Let $\mathcal{L}$ be a triple $\langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret} \rangle$, called a* language*, with Exp the set of* expressions*, Val the set of* values*, and $\mathsf{ret} \colon \mathsf{Val} \to \mathsf{Exp}$ an injective function. A* monadic operational semantics *for $\mathcal{L}$ consists of:*

- *a monad $\mathbb{M} = \langle M, \eta, \mu \rangle$*
- *a relation $\to \subseteq \mathsf{Exp} \times M\mathsf{Exp}$, called* monadic (one-step) reduction*, such that*
  - *$\to$ is a partial function and*
  - *for all $v \in \mathsf{Val}$, $\mathsf{ret}(v) \not\to$.*

The set Exp contains expressions that can be executed, while Val contains values produced by the computation. The inclusion ret identifies the expressions representing successful termination with a given value. The elements of $M\mathsf{Exp}$, called *monadic expressions*, are the counterpart of expressions in the monad $\mathbb{M}$. The relation $\to$ models single computation steps, which transform expressions into monadic ones, thus possibly raising computational effects. Finally, the first requirement on $\to$ ensures that it is deterministic, while the latter one that expressions representing values cannot be reduced.

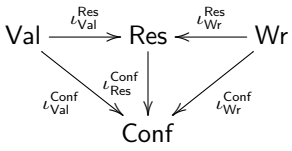
Assume a monadic operational semantics $\langle \mathbb{M}, \to \rangle$ for a language $\langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret} \rangle$. In standard (small-step) operational semantics, starting from the one-step reduction we can model computations as (either finite of infinite) sequences of reduction steps. In particular, finite computations are obtained by the reflexive and transitive closure $\to^\star$ of the one-step reduction. Starting from the *monadic* one-step reduction, which is a relation from a set to a different one, there is no transitive closure in the usual sense.

In the solution proposed in [16], the monadic reduction can be an arbitrary relation; however, this requires a relational extension of the monad [3]. On the other hand, given a relation $\to \subseteq \mathsf{Exp} \times M\mathsf{Exp}$ which is a *total* function, we can define, by iterating Kleisli composition, a relation $\to^\star \subseteq \mathsf{Exp} \times M\mathsf{Exp}$ which plays the role of transitive closure, as in [17].

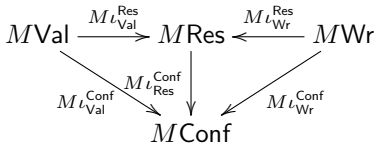
Our aim here is to define $\to^\star$ taking the second approach, which does not require relational extensions. Unfortunately, the monadic reduction, exactly as the standard one, is by its own nature a partial function, where some expressions, representing terminated computations,

cannot be reduced. Notably, those representing successful termination with a value, and others, intuitively corresponding to stuck computations. To obtain a total function, we extend the monadic reduction to *configurations* (expressions, values, or a special result wrong). In particular, expressions representing terminating computations reduce to (the monadic embedding of) a value, and wrong, respectively. In this way, we can define the transitive closure by Kleisli composition, as formally detailed below.

Set $\mathsf{Res} = \mathsf{Val} + \mathsf{Wr}$, where $\mathsf{Wr} = \{\mathsf{wrong}\}$, that is, a result r is either a value, modelling successful termination, or wrong, modelling a stuck computation. Then, we consider the set $\mathsf{Conf} = \mathsf{Exp} + \mathsf{Res}$ of *configurations*, ranged over by c. We have the following commutative diagram of coproduct injections:



As customary, with a slight abuse of notation, we identify elements with their images along such injections. We use E, V, R and C to range over monadic expressions, monadic values, monadic results and monadic configurations, respectively, that is, elements of $M\mathsf{Exp}$, $M\mathsf{Val}$, $M\mathsf{Res}$ and $M\mathsf{Conf}$. Since monads on $\mathcal{S}et$ preserve injections [1], by applying M to the diagram above we get another diagram of injections:



In the following, we use some shortcuts for the application of such injections: notably, we write $\hat{\mathrm{E}}$ for $M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}(\mathrm{E})$, $\hat{\mathrm{R}}$ for $M\iota^{\mathsf{Conf}}_{\mathsf{Res}}(\mathrm{R})$ and $\hat{\mathrm{V}}$ for $M\iota^{\mathsf{Res}}_{\mathsf{Val}}(\mathrm{V})$.

We can now extend the monadic reduction $\to$ to configurations, getting the relation $\xrightarrow[\mathsf{step}]{} \subseteq \mathsf{Conf} \times M\mathsf{Conf}$ shown in Figure 1. As said above, reduction is extended to expressions

$$(\textsc{exp})\ \frac{e \to \mathrm{E}}{e \xrightarrow[\mathsf{step}]{} \hat{\mathrm{E}}} \qquad (\textsc{ret})\ \frac{}{\mathsf{ret}(v) \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(v)}$$

$$(\textsc{wrong})\ \frac{e \not\to \quad e \neq \mathsf{ret}(v) \text{ for all } v \in \mathsf{Val}}{e \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(\mathsf{wrong})} \qquad (\textsc{res})\ \frac{}{r \xrightarrow[\mathsf{step}]{} \eta_{\mathsf{Conf}}(r)}$$

Figure 1 Monadic (one-step) reduction on configurations.

which represent terminated computations, which reduce to the monadic embedding of the corresponding value or wrong, respectively; moreover, it is extended to results (either values or wrong) as well, which conventionally reduce to their monadic embedding.

It is immediate to see that $\xrightarrow[\mathsf{step}]{}$ is (the graph of) a total function from Conf to $M\mathsf{Conf}$, which we simply write step. Clearly step is a Kleisli function for $\mathbb{M}$, hence we can define the "monadic reflexive and transitive closure" $\xrightarrow[\mathsf{step}]{}^{\star} \subseteq \mathsf{Conf} \times M\mathsf{Conf}$ of $\xrightarrow[\mathsf{step}]{}$ as follows:

$$(\textsc{refl})\ \frac{}{c \xrightarrow[\mathsf{step}]{}^{\star} \eta_{\mathsf{Conf}}(c)} \qquad (\textsc{step})\ \frac{c \xrightarrow[\mathsf{step}]{}^{\star} \mathrm{C}}{c \xrightarrow[\mathsf{step}]{}^{\star} \mathrm{C} \gg\!= \mathsf{step}}$$

These rules are analogous to those defining the reflexive and transitive closure of a standard one-step relation. In (REFL) a configuration reduces, rather than to itself, to its monadic counterpart. In rule (STEP), $\xrightarrow[\mathsf{step}]{}^{\star}$ is combined with $\xrightarrow[\mathsf{step}]{}$, rather than by standard composition, through the $\gg=$ operator. That is, a computation is extended by one step through a monadic binding of the previously computed monadic configuration C to the step function.

Equivalently, we can define the Kleisli n-th iteration step^n of the step function by setting $\mathsf{step}^0 = \eta_{\mathsf{Conf}}$ and $\mathsf{step}^{n+1} = \mathsf{step} * \mathsf{step}^n$. Then, the following holds:

▶ **Proposition 7.** *$c \xrightarrow[\mathsf{step}]{}^{\star} \mathrm{C}$ if and only if $\mathsf{step}^n(c) = \mathrm{C}$ for some $n \in \mathbb{N}$.*

In a similar way, we can define a small-step reduction on monadic configurations. Recall that the Kleisli extension of step gives the function $\mathsf{step}^{\dagger} \colon M\mathsf{Conf} \to M\mathsf{Conf}$.

▶ **Definition 8.** *The* small-step reduction induced by $\xrightarrow[\mathsf{step}]{}$ *is the relation $\Rightarrow$ on $M\mathsf{Conf}$ defined by: $\mathrm{C} \Rightarrow \mathrm{C}'$ iff $\mathsf{step}^{\dagger}(\mathrm{C}) = \mathrm{C}'$.*

Then, since $\Rightarrow$ is a relation on $M\mathsf{Conf}$, we can consider its (standard) reflexive and transitive closure $\Rightarrow^{\star} \subseteq M\mathsf{Conf} \times M\mathsf{Conf}$, which describes computations on monadic configurations.

▶ **Proposition 9.** *$\mathrm{C} \Rightarrow^{\star} \mathrm{C}'$ if and only if $\mathrm{C} \gg= \mathsf{step}^n = \mathrm{C}'$ for some $n \in \mathbb{N}$.*

Combining Propositions 7 and 9, we get the following corollary which relates $\xrightarrow[\mathsf{step}]{}^{\star}$ and $\Rightarrow^{\star}$.

▶ **Corollary 10.** *The following are equivalent:*

1. $c \xrightarrow[\mathsf{step}]{}^{\star} \mathrm{C}$
2. $\eta_{\mathsf{Conf}}(c) \Rightarrow^{\star} \mathrm{C}$
3. $c \xrightarrow[\mathsf{step}]{}^{\star} \mathrm{C}' \Rightarrow^{\star} \mathrm{C}$, *for some* $\mathrm{C}' \in M\mathsf{Exp}$

To use the above machinery for describing the semantics of expressions, we essentially follow the approach in [17], with minor adjustments to fit our context.

First of all note that, being defined on top of a total function, $\Rightarrow^{\star}$ has no normal forms. However, monadic results should intuitively correspond to termination. Formally, this is a consequence of the proposition below, stating that a monadic configuration which is a result only reduces to itself; hence, when a monadic result is reached, its reduction continues with an infinite sequence of trivial reduction steps, which can be seen as a representation of termination. Hence, the outcome of a (terminating) computation is a monadic result.

▶ **Proposition 11.** *$\hat{\mathrm{R}} \Rightarrow \mathrm{C}$ if and only if $\mathrm{C} = \hat{\mathrm{R}}$.*

Thanks to the above proposition, we can prove the following, stating that the monadic result of a computation, if any, is unique.

▶ **Proposition 12.** *If $c \xrightarrow[\mathsf{step}]{}^{\star} \hat{\mathrm{R}}_1$ and $c \xrightarrow[\mathsf{step}]{}^{\star} \hat{\mathrm{R}}_2$, then $\mathrm{R}_1 = \mathrm{R}_2$.*

Hence, we can define a function $[\![-]\!]_{\star} \colon \mathsf{Exp} \to M\mathsf{Res} + \{\infty\}$ describing the semantics of expressions as follows:

$$[\![e]\!]_{\star} = \begin{cases} \mathrm{R} & \text{if } e \xrightarrow[\mathsf{step}]{}^{\star} \hat{\mathrm{R}} \\ \infty & \text{otherwise} \end{cases}$$

This is called *finitary semantics*, as it describes only monadic results that can be reached in finitely many steps. In other words, all diverging computations are identified and no information on computational effects they may produce is available. Even worse, when the monad supports some form of non-determinism, we may have computations that terminate in some cases and diverge in others, but the finitary semantics considers them as diverging, as they never reach a result after finitely many steps.

To overcome this limitation, again following [17], we introduce an *infinitary semantics*, which is able to provide more information on diverging computations. To achieve this, we need to assume more structure on the monad $\mathbb{M}$. Recall that, given a partially ordered set $\langle P, \sqsubseteq \rangle$, an ω-chain is an increasing sequence $(x_n)_{n\in\mathbb{N}}$ of points in P. We say that $\langle P, \sqsubseteq \rangle$ is an ω-CPO if it has a least element $\bot$ and every ω-chain in $\langle P, \sqsubseteq \rangle$ has a supremum $\bigsqcup_{n\in\mathbb{N}} x_n$. A function $f\colon \langle P, \sqsubseteq \rangle \to \langle P', \sqsubseteq' \rangle$ between ω-CPO is said to be ω-continuous if it preserves the least element, and suprema of ω-chains. Note that an ω-continuous function is necessarily monotone. Then, we have the following definition:

▶ **Definition 13.** *An ω-CPO-ordered monad $\mathbb{M} = \langle M, \sqsubseteq, \eta, \mu \rangle$ is a monad $\langle M, \eta, \mu \rangle$ together with a partial order $\sqsubseteq_X$ on MX, for every set X, such that*

1. *for every set X, the poset $\langle MX, \sqsubseteq_X \rangle$ is an ω-CPO and*
2. *for all sets X, Y, the Kleisli extension $(-)^\dagger\colon (X \to MY) \to (MX \to MY)$ is ω-continuous with respect to the pointwise extension of $\sqsubseteq_Y$ to function spaces $X \to MY$ and $MX \to MY$.*

▶ **Example 14.** The powerset and list monads of Example 2 are ω-CPO-ordered with the subset and prefix ordering, respectively. The subdistribution monad of Example 3 is ω-CPO-ordered with the pointwise ordering on subdistributions. The other monads of Section 2 can also be turned into ω-CPO-ordered monads, but require adjustements, typically a combination with the exception monad. For instance, in Example 4, the output monad is not ω-CPO-ordered in general, but its pointed version is ω-CPO-ordered when the underlying monoid is an ω-CPO and the multiplication is ω-continuous in the second argument.

From now on, we assume the monad $\mathbb{M}$ to have an ω-CPO-ordered structure. Our goal is to define a function $[\![-]\!]_\infty\colon \mathsf{Exp} \to M\mathsf{Res}$ modelling the infinitary semantics of expressions. To this end, we first define a function $\mathsf{res}\colon M\mathsf{Conf} \to M\mathsf{Res}$ extracting monadic results from monadic configurations. Let $\mathsf{res}_0\colon \mathsf{Conf} \to M\mathsf{Res}$ be the function given by

$$\mathsf{res}_0(c) = \begin{cases} \bot_{\mathsf{Res}} & c = e \\ \eta_{\mathsf{Res}}(r) & c = r \end{cases}$$

and set $\mathsf{res} = \mathsf{res}_0^\dagger$. The key point is that the total relation $\Rightarrow$ on monadic configurations is compatible with the order $\sqsubseteq_{\mathsf{Res}}$ under the application of res, as the following proposition shows.

▶ **Proposition 15.** *If $c \Rightarrow c'$ then $\mathsf{res}(c) \sqsubseteq_{\mathsf{Res}} \mathsf{res}(c')$.*

For every $e \in \mathsf{Exp}$ and $n \in \mathbb{N}$, we define $[\![e]\!]_n = \mathsf{res}(\mathsf{step}^n(e))$. From Propositions 7 and 9, we easily derive $\mathsf{step}^n(e) \Rightarrow \mathsf{step}^{n+1}(e)$, and, by Proposition 15, $[\![e]\!]_n \sqsubseteq_{\mathsf{Res}} [\![e]\!]_{n+1}$. Hence, the sequence $([\![e]\!]_n)_{n\in\mathbb{N}}$ is an ω-chain in $\langle M\mathsf{Res}, \sqsubseteq_{\mathsf{Res}} \rangle$ and so we define the infinitary semantics as

$$[\![e]\!]_\infty = \bigsqcup_{n\in\mathbb{N}} [\![e]\!]_n$$

Intuitively, $[\![e]\!]_n$ is the portion of the result that is reached after n reduction steps. Hence, the actual result is obtained as the supremum of all such approximations and it may be never reached, thus describing also the observable behaviour of possibly diverging computations.

We conclude this section by stating that infinitary and finitary semantics agree on terminating computations.

▶ **Proposition 16.** *If $[\![e]\!]_\star = r$, then $[\![e]\!]_\infty = r$.*

4 Example: a lambda calculus with generic effects

The aim of this section is twofold:

- to ilustrate the monadic operational semantics in Section 3 through a simple example
- to equip such example with a type-and-effect system, and to discuss how to express and prove type soundness with respect to finitary/infinitary semantics

To this end, we introduce Λ_Σ, a call-by-value λ-calculus with *generic effects*. Here Σ is a family of sets $\{\Sigma_k\}_{k\in\mathbb{N}}$ of k-ary operations raising effects. We choose generic rather than algebraic effects, thus avoiding explicit continuations, to have a style more convenient for a programmer [40], and a more significant monadic reduction.[4]

The syntax is shown in Figure 2. We use $\overline{v}$ as metavariable for sequences $v_1, \ldots, v_n$, and analogously for other sequences. We assume variables x, y, f, ..., using the last for variables

$$\begin{array}{lcll} v & ::= & x \mid \mathtt{rec}\, f.\lambda x.e \mid \ldots & \text{value} \\ e & ::= & v\, v' \mid op(\overline{v}) \mid \mathtt{return}\ v \mid \mathtt{do}\ x = e_1\,;\ e_2 \mid \ldots & \text{expression} \end{array}$$

Figure 2 Λ_Σ: fine-grain syntax.

denoting functions. We adopt, as customary, the fine-grain approach [26], where *values* are effect-free, whereas *expressions*, also called *computations*, may raise effects. Dots stand for additional, unspecified, constructs, such as operators of primitive types, conditional, etc.

To illustrate type soundness with respect to the infinitary semantics as well, the calculus includes recursive functions; notably, $\mathtt{rec}\, f.\lambda x.e$ is a function with parameter x and body e which can recursively call itself through the variable f. Standard lambda expressions can be recovered as those where f does not occur free in e, that is, when the function is non-recursive, and we will use the abbreviation $\lambda x.e$ for such expressions.

In this section, Exp and Val denote the sets of closed expressions and values of Λ_Σ, respectively. In the following, we define a monadic (one-step) reduction for the language, parametric on a monad $\mathbb{M} = \langle M, \eta, \mu\rangle$, being a relation $\rightarrow$ on $\mathsf{Exp} \times M\mathsf{Exp}$. As in Section 3, we use V and E to range over $M\mathsf{Val}$ and $M\mathsf{Exp}$, respectively.

This relation is modularly defined on top of a "pure" reduction $\rightarrow_p$ on $\mathsf{Exp} \times \mathsf{Exp}$. In our example, such relation only reduces function calls into the corresponding bodies, as shown in Figure 3; other rules should be added to deal with additional language constructs, as we will do for handlers in Section 7. `Do` expressions are, then, normal forms for the pure reduction, and will be handled by the rules of the monadic reduction.

$$(\textsc{app})\ \frac{}{v\, v' \rightarrow_p e[v/f][v'/x]}\ \ v = \mathtt{rec}\, f.\lambda x.e$$

Figure 3 Pure reduction.

Rules defining the monadic reduction are given in Figure 4. As mentioned, they are parametric on the underlying monad; more in detail, they depend on the following ingredients:

- The function $\eta_{\mathsf{Exp}}\colon \mathsf{Exp} \rightarrow M\mathsf{Exp}$ embedding language expressions into their counterpart in the monad, written simply η in this section.
- The function $\mathsf{map}\colon (\mathsf{Exp} \rightarrow \mathsf{Exp}) \rightarrow M\mathsf{Exp} \rightarrow M\mathsf{Exp}$ lifting functions from expressions to expressions to their counterpart in the monad.

[4] In the case of algebraic effects there would be no monadic reduction inside a context, as in rule (DO).

Moreover we assume, for each operation op with arity k, a partial function $\mathsf{run}_{op}{:}\mathsf{Val}^k \rightharpoonup M\mathsf{Val}$, returning a monadic value expressing the effects raised by a call of the operation. The function could be undefined, for instance when arguments do not have the expected types.

$$(\textsc{pure})\ \frac{e \to_p e'}{e \to \eta(e')} \qquad (\textsc{effect})\ \frac{}{op(\overline{v}) \to \mathsf{map}\,(\mathtt{return}\ [\,])\,\mathsf{run}_{op}(\overline{v})}$$

$$(\textsc{ret})\ \frac{}{\mathtt{do}\ x = \mathtt{return}\ v;\ e \to \eta(e[v/x])} \qquad (\textsc{do})\ \frac{e_1 \to \mathrm{E}}{\mathtt{do}\ x = e_1;\ e_2 \to \mathsf{map}\,(\mathtt{do}\ x = [\,];\ e_2)\,\mathrm{E}}$$

Figure 4 Monadic (one-step) reduction.

Rule (PURE) propagates a pure step, embedding its result in the monad. In rule (EFFECT), the effect is actually raised. To this end, we apply the function of type $M\mathsf{Val} \to M\mathsf{Exp}$ obtained by lifting, through `map`, the context `return` $[\,]$ to the monadic value obtained from the call. Here we identify the context `return` $[\,]$, which is an expression with a hole, with the function $v \mapsto \mathtt{return}\ [v]$ of type $\mathsf{Val} \to \mathsf{Exp}$. In rule (RET), when the first subterm of a `do` expression returns a value, the expression is reduced to the monadic embedding of the second subterm, after replacing the variable with the value. Rule (DO), instead, propagates the reduction of the first subterm. To take into account raised effects, we apply the function of type $M\mathsf{Exp} \to M\mathsf{Exp}$ obtained by lifting, through `map`, the context `do` $x = [\,];\ e_2$ to the monadic expression obtained from e_1. Analogously to above, we identify the context `do` $x = [\,];\ e_2$ with the function $e \mapsto \mathtt{do}\ x = [e];\ e_2$ of type $\mathsf{Exp} \to \mathsf{Exp}$.

The following property is needed to have an instance of the framework in Section 3.

▶ **Proposition 17** (Determinism). *If $e \to E_1$ and $e \to E_2$ then $E_1 = E_2$.*

We show now some examples of expressions and their monadic operational semantics. We assume the calculus to be extended with standard constructs, such `unit`, 0, `succ`, `true` and `false` constructors, `pred` selector, `iszero` test, and conditional. We write $e;e'$ for `do` $x = e;\ e'$ when x does not occur free in e', and sometimes, to save space, n for $\mathtt{succ}^n\,0$.

▶ **Example 18.** Set, as underlying monad, the monad of exceptions introduced in Example 1, where $EX = X + \mathsf{Exc}$. For each $\mathsf{e} \in \mathsf{Exc}$, we assume an operation $\mathtt{raise}\langle\mathsf{e}\rangle$, with

$$\mathsf{run}_{\mathtt{raise}\langle\mathsf{e}\rangle} : \mathbf{1} \to M\mathsf{Val} \qquad \mathsf{run}_{\mathtt{raise}\langle\mathsf{e}\rangle} = \iota_2(\mathsf{e})$$

The function $\mathsf{predfun} = \lambda x.\mathtt{if}\ \mathtt{iszero}\ x\ \mathtt{then}\ \mathtt{raise}\langle\mathsf{PredZero}\rangle\ \mathtt{else}\ \mathtt{return}\ \mathtt{pred}\,x$ raises the exception PredZero when the argument is 0. The following are examples of small-step reduction sequences on monadic configurations[5]:

$$\begin{aligned}
\mathsf{predfun}\,\mathtt{succ}\,0 &\Rightarrow \mathtt{if}\ \mathtt{iszero}\ \mathtt{succ}\,0\ \mathtt{then}\ \mathtt{raise}\langle\mathsf{PredZero}\rangle\ \mathtt{else}\ \mathtt{return}\ 0\\
&\Rightarrow \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ \mathtt{raise}\langle\mathsf{PredZero}\rangle\ \mathtt{else}\ \mathtt{return}\ 0\\
&\Rightarrow \mathtt{return}\ 0\\
&\Rightarrow 0\\
\mathsf{predfun}\,0 &\Rightarrow \mathtt{if}\ \mathtt{iszero}\ 0\ \mathtt{then}\ \mathtt{raise}\langle\mathsf{PredZero}\rangle\ \mathtt{else}\ \mathtt{return}\ 0\\
&\Rightarrow \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{raise}\langle\mathsf{PredZero}\rangle\ \mathtt{else}\ \mathtt{return}\ 0\\
&\Rightarrow \mathtt{raise}\langle\mathsf{PredZero}\rangle\\
&\Rightarrow \mathsf{PredZero}
\end{aligned}$$

[5] Where we omit the injections from monadic expressions and values.

In the first reduction sequence, all steps are derived by rules (PURE) in Figure 4 and (EXP) in Figure 1, except for the last one, which is derived by rule (RET) in Figure 1. Analogously in the second reduction sequence, where the last step is derived by rule (EFFECT) in Figure 4 and (EXP) in Figure 1. Note that, here and in the following examples, after reaching a monadic result the sequence of steps continues with an infinite sequence of steps, in the case above $0 \Rightarrow 0$ and $\mathsf{PredZero} \Rightarrow \mathsf{PredZero}$ steps.

▶ **Example 19.** Set, as underlying monad, the monad of non-determinism of Example 2, in the variant of the possibly infinite lists. We assume a constant operation `choose`, with

$$\mathsf{run}_{\mathtt{choose}} \colon \mathbf{1} \to M\mathsf{Val} \qquad \mathsf{run}_{\mathtt{choose}} = [\mathtt{true}, \mathtt{false}]$$

Then, $e = \mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathtt{succ}\,0$ reduces as follows.[6]

$$\begin{array}{rcl}
[\,e\,] & \Rightarrow & [\,\mathtt{do}\ y = \mathtt{true};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathtt{succ}\,0, \\
& & \ \mathtt{do}\ y = \mathtt{false};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathtt{succ}\,0\,] \\
& \Rightarrow & [\,\mathtt{if}\ \mathtt{t}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathtt{ret}\ 1, \mathtt{if}\ \mathtt{f}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathtt{ret}\ 1\,] \\
& \Rightarrow & [\,\mathtt{return}\ 0, \mathtt{return}\ \mathtt{succ}\,0\,] \\
& \Rightarrow & [\,0, \mathtt{succ}\,0\,]
\end{array}$$

Given $\mathsf{chfun}^{\uparrow} = \mathtt{rec}\, f.\lambda x.\mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ x\ \mathtt{else}\ f\,\mathtt{succ}\,x$, the expression $\mathsf{chfun}^{\uparrow}\,0$ reduces as follows:

$$\begin{array}{rcl}
[\,\mathsf{chfun}^{\uparrow}\,0\,] & \Rightarrow & [\,\mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,\mathtt{do}\ y = \mathtt{true};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathtt{return}\ \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0, \\
& & \ \mathtt{do}\ y = \mathtt{false};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{return}\ 0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,\mathtt{if}\ \mathtt{t}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathtt{ret}\ 1, \mathtt{if}\ \mathtt{f}\ \mathtt{then}\ \mathtt{ret}\ 0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\ 1\,] \\
& \Rightarrow & [\,\mathtt{return}\ 0, \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,0, \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,0\,] \\
& \Rightarrow & [\,0, \mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{ret}\ \mathtt{succ}\,0\ \mathtt{else}\ \mathsf{chfun}^{\uparrow}\,\mathtt{s}\,\mathtt{s}\,0\,] \\
& \Rightarrow^{\star} & [\,0, \mathtt{succ}\,0, \mathsf{chfun}^{\uparrow}\,\mathtt{succ}\,\mathtt{succ}\,0\,] \\
& \cdots &
\end{array}$$

Note that the second reduction is non-terminating, in the sense that a monadic result (a list of values) is never reached. Hence, with the finitary semantics, we get $[\![\mathsf{chfun}^{\uparrow}\,0]\!]_{\star} = \infty$. With the infinitary semantics, instead, we get the following ω-chain:

$$[\ ], \ldots, [\,0\,], \ldots, [\,0, \mathtt{succ}\,0\,], \ldots, [\,0, \mathtt{succ}\,0, \ldots, \mathtt{succ}^n\,0\,], \ldots,$$

whose supremum is, as expected, the infinite list of the (values representing the) natural numbers. On the other end, given the function

$$\mathsf{chfun}^{\downarrow} = \mathtt{rec}\, f.\lambda x.\mathtt{if}\ \mathtt{iszero}\ x\ \mathtt{then}\ \mathtt{ret}\ x\ \mathtt{else}\ \mathtt{do}\ y = \mathtt{choose};\ \mathtt{if}\ y\ \mathtt{then}\ \mathtt{ret}\ x\ \mathtt{else}\ f\,\mathtt{pred}\,x$$

we get $[\![\mathsf{chfun}^{\downarrow}\,\mathtt{succ}^n\,0]\!]_{\star} = [\![\mathsf{chfun}^{\downarrow}\,\mathtt{succ}^n\,0]\!]_{\infty} = [\,\mathtt{succ}^n\,0, \ldots, 0\,]$.

▶ **Example 20.** Set, as underlying monad, the monad of probabilistic non-determinism of Example 3. We consider a discrete uniform distribution over a set of two elements and use the same function `choose`, now returning the list consisting of the values `true` and `false` with probability $\frac{1}{2}$, that we denote by $[\,\frac{1}{2} : \mathtt{true}, \frac{1}{2} : \mathtt{false}\,]$.

[6] We use `t`, `f`, `ret`, and `s`, for `true`, `false`, `return`, and `succ`, to save space.

Then, $[\,1 : e\,]$ and $[\,1 : \mathsf{chfun}^{\uparrow}\,0\,]$ reduce analogously to the previous example:

$$[\,1 : e\,] \Rightarrow^{\star} [\,\frac{1}{2} : 0, \frac{1}{2} : \mathtt{succ}\,0\,]$$

$$[\,[\,1 : \mathsf{chfun}^{\uparrow}\,0\,]\,] \Rightarrow^{\star} [\,\frac{1}{2} : 0, \frac{1}{4} : \mathtt{succ}\,0, \frac{1}{8} : \mathtt{succ}^2\,0, \frac{1}{16} : \mathtt{succ}^3\,0\,] \Rightarrow \dots$$

Again, the second reduction is non-terminating, hence, with the finitary semantics, we get ∞, whereas, with the infinitary semantics, we get an ω-chain whose supremum is the infinite list where each (value representing the) number n has probability $\frac{1}{2^{n+1}}$.

▶ **Example 21.** Set, as underlying monad, the output monad of Example 4, in its pointed version. As a simple concrete choice, we take as elements of Out sequences of pairs $\langle \ell, \mathtt{succ}^n\,0\rangle$ where ℓ ranges over a fixed set Loc of *output locations* modeling, e.g., file names or output channels. We assume, for each ℓ, an operation $\mathtt{write}\langle\ell\rangle\colon \mathtt{Nat} \to \mathtt{Unit}$, with

$$\mathsf{run}_{\mathtt{write}\langle\ell\rangle}\colon \mathsf{Val} \to M\mathsf{Val} \qquad \mathsf{run}_{\mathtt{write}\langle\ell\rangle}(v) = \begin{cases} \langle\langle \ell, v\rangle, \mathtt{unit}\rangle & \text{if } v = \mathtt{succ}^n\,0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Given two distinct output locations ℓ, ℓ', and the functions

$$\mathsf{wfun}^{\uparrow} = \mathtt{rec}\,f.\lambda x.\mathtt{write}\langle\ell\rangle(x)\mathtt{;write}\langle\ell'\rangle(x)\mathtt{;}f\,\mathtt{succ}\,x$$
$$\mathsf{wfun}^{\downarrow} = \mathtt{rec}\,f.\lambda x.\mathtt{write}\langle\ell\rangle(x)\mathtt{;write}\langle\ell'\rangle(x)\mathtt{;if\ iszero}\ x\ \mathtt{then\ unit\ else}\ f\,\mathtt{pred}\,x$$

we get, as in the previous examples, the following semantics:

$$[\![\mathsf{wfun}^{\uparrow}\,0]\!]_{\star} = \infty$$
$$[\![\mathsf{wfun}^{\uparrow}\,0]\!]_{\infty} = \langle\langle \ell, 0\rangle \cdot \langle \ell', 0\rangle \cdot \langle \ell, \mathtt{succ}\,0\rangle \cdot \langle \ell', \mathtt{succ}\,0\rangle \cdot \dots, \bot\rangle$$
$$[\![\mathsf{wfun}^{\downarrow}\,\mathtt{s}^n\,0]\!]_{\star} = [\![\mathsf{wfun}^{\downarrow}\,\mathtt{s}^n\,0]\!]_{\infty} = \langle\langle \ell, \mathtt{s}^n\,0\rangle \cdot \langle \ell', \mathtt{s}^n\,0\rangle \cdot \dots \cdot \langle \ell, 0\rangle \cdot \langle \ell', 0\rangle, \mathtt{unit}\rangle$$

In the first two cases the reduction does not terminate, so no value is returned. With the finitary semantics also no effect is produced, whereas with the infinitary semantics the effect is the infinite sequence of outputs.

In order to equip the calculus (Figure 2) with a type-and-effect system, we need the ingredients shown in Figure 5. Besides types, which are functional types and additional unspecified

$$\begin{array}{lcll} T & ::= & \dots \mid T \to_E T' & \text{type} \\ \Gamma & ::= & \overline{x : T} & \text{context} \end{array}$$

Figure 5 Types and contexts.

types, we consider *effect types* (*effects* when there is no ambiguity), ranged over by E, meant to be static approximations of the computational effects raised by an expression. As formally detailed below, effects are sets of (possibly infinite) sequences of operations. In this way, they are expressive enough to approximate computational effects in many different monads, as we will describe in Section 6, and we abstract away from details of a syntactic representation, which of course would be needed in a real language. Functional types are annotated with an effect, approximating the computational effects of calling the function. Finally, we assume operations to be typed; formally, for each op, we write $op\colon T_1 \dots T_n \to T$.

Set Σ^{∞} the set of either finite or infinite sequences of operations. We use α, β to range over elements of Σ^{∞}, denote by ϵ the empty sequence, by $op{:}\alpha$ the sequence consisting of op followed by α, and by $\cdot$ sequence concatenation, coinductively defined by:

$$\epsilon\cdot\beta = \beta \qquad (op{:}\alpha)\cdot\beta = op{:}(\alpha\cdot\beta)$$

As customary, we write op for the sequence $op{:}\epsilon$.

An *effect* is a non-empty subset of Σ^∞. We denote by $\cdot$ composition of effects, defined by:

$$E{\cdot}E' = \{\alpha{\cdot}\beta \mid \alpha \in E, \beta \in E'\}$$

Absence of effects is modeled by the set $\{\epsilon\}$; the empty effect, if allowed, could be assigned to non-terminating computations which never call operations; however, since $E{\cdot}\emptyset = \emptyset$, effects assigned to a previous terminating computation would be lost.

The type-and-effect system is shown in Figure 6. The subtyping judgment has shape

$$\text{(SUB-FUN)}\ \frac{T_1' \le T_1 \quad T_2 \le T_2'}{T_1 \to_E T_2 \le T_1' \to_{E'} T_2'}\ E \subseteq E' \qquad \text{(SUB-REFL)}\ \frac{}{T \le T}$$

$$\text{(SUB-TRANS)}\ \frac{T \le T' \quad T' \le T''}{T \le T''} \qquad \text{(SUB-TE)}\ \frac{T \le T' \quad E \subseteq E'}{T!E \le T'!E'}$$

$$\text{(T-VAR)}\ \frac{}{\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(T-ABS)}\ \frac{\Gamma, f : T \to_E T', x : T \vdash e : T''!E'}{\Gamma \vdash \mathtt{rec}\, f.\lambda x.e : T \to_E T'}\ T''!E' \le T'!E$$

$$\text{(T-APP)}\ \frac{\Gamma \vdash v_1 : T_1 \to_E T \quad \Gamma \vdash v_2 : T_2}{\Gamma \vdash v_1\, v_2 : T!E}\ T_2 \le T_1 \qquad \text{(T-OP)}\ \frac{\Gamma \vdash v_i : T_i' \ \ \forall i \in 1..n}{\Gamma \vdash op(\overline{v}) : T!\{op\}}\ \begin{array}{l}\overline{v} = v_1, \ldots, v_n \\ op{:}\, T_1 \ldots T_n \to T \\ T_i' \le T_i\ \forall i \in 1..n\end{array}$$

$$\text{(T-RET)}\ \frac{\Gamma \vdash v : T}{\Gamma \vdash \mathtt{return}\ v : T!\{\epsilon\}} \qquad \text{(T-DO)}\ \frac{\Gamma \vdash e_1 : T_1!E_1 \quad \Gamma, x : T_2 \vdash e_2 : T!E_2}{\Gamma \vdash \mathtt{do}\ x = e_1\mathtt{;}\ e_2 : T!E_1{\cdot}E_2}\ T_1 \le T_2$$

Figure 6 Type-and-effect system.

$T \le T'$. In (SUB-FUN) inclusion of effect types is propagated to functional types. Moreover subtyping is, as expected, covariant/contravariant on the result/parameter of functions. The other rules are standard.

The typing judgment for values has shape $\Gamma \vdash v : T$, since they have no effects. The judgment for expressions, instead, has shape $\Gamma \vdash e : T!E$.

Rule (T-VAR) is standard. In rule (T-ABS), a (possibly recursive) function gets a functional type, consisting of parameter/result types and effect, if the body, in a context where parameter and function are added with their types, gets a subtype and a subeffect of the result type and effect of the function. In rule (T-APP), an application gets the result type and the effect of the applied function, provided that the argument type is subtype of the expected one. In rule (T-OP), calling an operation raises the corresponding singleton effect, provided that the argument types are subtypes of the expected ones. In rule (T-RET), an expression representing a value has the trivial effect, and, in (T-DO), a sequential composition of two computations has the composition of the two effects.

▶ **Example 22.** We show some typing judgments which can be derived for the previous examples. We assume primitive types `Nat` and `Bool`, an empty type `Bot` subtype of any type, the singleton type `Unit` for the constant `unit`, and the obvious typing rule for conditional which takes the union of the effects of the two branches. Finally, we denote by α^n and α^ω a finite and infinite concatenation of αs, respectively.

1. In Example 18, with, for each $\mathsf{e} \in \mathsf{Exc}$, $\mathtt{raise}\langle \mathsf{e} \rangle{:}\, \mathbf{1} \to \mathtt{Bot}$,

$$\emptyset \vdash \mathsf{predfun} : \mathtt{Nat} \to_{\{\epsilon, \mathtt{raise}\langle \mathsf{PredZero} \rangle\}} \mathtt{Nat}$$

$$\emptyset \vdash \mathsf{predfun}\, v : \mathtt{Nat}!\{\epsilon, \mathtt{raise}\langle \mathsf{PredZero} \rangle\}\mathtt{if}\emptyset \vdash \mathtt{v} : \mathtt{Nat}$$

Note a significant feature of our type effects: differently from, e.g., Java checked exceptions, we can distinguish code which *may* raise an exception, as expressed by the effect $\{\epsilon, \texttt{raise}\langle\mathsf{PredZero}\rangle\}$, from code which *necessarily* raises an exception, as expressed by the effect $\{\texttt{raise}\langle\mathsf{PredZero}\rangle\}$, which is assigned, e.g., to the function $\lambda x.\texttt{raise}\langle\mathsf{PredZero}\rangle$. More in general, our type effects can *force* computational effects to be raised.

2. In Example 19, with $\texttt{choose}\colon \to \texttt{Bool}$,

$$\emptyset \vdash \mathsf{chfun}^{\uparrow} : \texttt{Nat} \to_{\{\texttt{choose}^n \mid n \geq 1\}} \texttt{Nat}$$
$$\emptyset \vdash \mathsf{chfun}^{\downarrow} : \texttt{Nat} \to_{\{\texttt{choose}^n \mid n \geq 0\}} \texttt{Nat}$$

Again, the effect of the first function forces non-determinism, differently from that of the second one. Apart from that, the two effects are very similar, even though calls of the first and second function always diverge and terminate, respectively. Indeed, as usual, effect types only provide a static approximation of the computational effects.

3. In Example 21, with, for each output location ℓ, $\texttt{write}\langle\ell\rangle\colon \texttt{Nat} \to \texttt{Unit}$,

$$\emptyset \vdash \mathsf{wfun}^{\uparrow} : \texttt{Nat} \to_{\{(\texttt{write}\langle\ell\rangle \cdot \texttt{write}\langle\ell'\rangle)^{\omega}\}} \texttt{Unit}$$
$$\emptyset \vdash \mathsf{wfun}^{\downarrow} : \texttt{Nat} \to_{\{(\texttt{write}\langle\ell\rangle \cdot \texttt{write}\langle\ell'\rangle)^{n} \mid n \geq 0\}} \texttt{Unit}$$

Here the difference between the effects of the two functions is even more significant: in the former, the sequence of two write calls is necessarily done infinitely many times, in the latter it can be done any arbitrary, yet finite, positive number of times. Moreover, in this case effects also provide an information on the the order among different write calls; for instance, here a $\texttt{write}\langle\ell\rangle$ call should be always followed by a $\texttt{write}\langle\ell'\rangle$ call.

We discuss now how to express and prove type soundness. Recall that the monadic operational semantics defined in Section 3 constructs, on top of the one-step reduction:

- a finitary semantics $[\![-]\!]_{\star}\colon \mathsf{Exp} \to M\mathsf{Res} + \{\infty\}$
- an infinitary semantics $[\![-]\!]_{\infty}\colon \mathsf{Exp} \to M\mathsf{Res}$

where $\mathsf{Res} = \mathsf{Val} + \mathsf{Wr}$, with the latter modelling a stuck computation. Hence, we expect a sound type-and-effect system to guarantee, first of all, that

(1) the (monadic) result of a well-typed expression is never wrong

analogously to what we expect for a standard type system. In the standard case, we also expect the result, if any, to be in agreement with the expression type. Here, since the expression has an effect as well, approximating the computational effects raised by its execution, we expect that

(2) the monadic result, if any, is in agreement with the expression type and effect

In finitary semantics, (2) imposes nothing on diverging expressions, since they have no monadic result, whereas, in infinitary semantics, (2) is significant for diverging expressions as well.

To formally express (2), we need to derive, from the well-typedness predicates (one for each type and effect), analogous predicates on monadic results. In the following section, this is achieved through a *predicate lifting* [21] λ, that is, a way to lift, for every set X, predicates over X to predicates over MX. Intuitively, λ adds requirements on the computational effects,expressed by an effect type, that is, lifting provides an *interpretation of effect types*.

5 Monadic type-and-effect soundness

The standard technique for proving type soundness with respect to a small-step operational semantics is as a consequence, by a simple inductive argument, of progress and subject reduction properties [47]. In this section, we introduce an analogous technique for monadic

operational semantics. Notably, we express progress and subject reduction for the monadic one-step reduction, and prove that they imply soundness. We develop our technique for *type-and-effect systems* [45, 32, 46, 29, 25], that is, formal systems providing an (over)approximation not only of the result of a computation, but also of its computational effects.

Following [7, 6], a type system can be abstractly seen as a family of predicates over expressions and values indexed by types. In a type-and-effect system, predicates over expressions will be indexed not only by types but also by *effect types*, describing the computational effects that expressions can produce during their evaluation, as defined below.

▶ **Definition 23.** *A* type-and-effect system $\Theta = \langle \mathsf{Ty}, \mathcal{E}, \mathsf{WT}^{\mathsf{E}}, \mathsf{WT}^{\mathsf{V}} \rangle$ *for a language* $\mathcal{L} = \langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret} \rangle$ *consists of the following data:*

- *a set* Ty *of* types
- *an ordered monoid* $\mathcal{E} = \langle \mathsf{Eff}, \preceq, \cdot, 1 \rangle$ *of* effect types
- *for every* $\tau \in \mathsf{Ty}$ *and* $\varepsilon \in \mathsf{Eff}$*, predicates* $\mathsf{WT}^{\mathsf{V}}_{\tau} \subseteq \mathsf{Val}$ *and* $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} \subseteq \mathsf{Exp}$ *such that*
  - $\varepsilon \preceq \varepsilon'$ *implies* $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} \subseteq \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon'}$ *and*
  - $\mathsf{ret}(v) \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *iff* $v \in \mathsf{WT}^{\mathsf{V}}_{\tau}$ *and* $1 \preceq \varepsilon$

The ordered monoid is a typical structure for effect systems [32, 29, 25]: 1 represents the absence of computational effects, $\varepsilon_1 \cdot \varepsilon_2$ represents the composition of computational effects described by ε_1 and ε_2, and $\varepsilon_1 \preceq \varepsilon_2$ states that the effect type ε_1 is more specific than ε_2.

The two families WT^{V} and WT^{E} are, for each index, predicates over values and expressions, respectively: $\mathsf{WT}^{\mathsf{V}}_{\tau}$ is the set of values of type τ, and $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ is the set of expressions of type τ which may raise effects described by ε. The first requirement, that is, monotonicity with respect to the order, states that the latter actually models if $\varepsilon_1 \preceq \varepsilon_2$, then ε_1 is really more specific than ε_2. The second requirement states that an expression which is the embedding of a value has the same type, and an effect type which is not forcing any effect.[7]

Consider now an operational semantics $\langle \mathbb{M}, \rightarrow \rangle$, with $\mathbb{M} = \langle M, \mu, \eta \rangle$, and focus, e.g., on reduction from expressions to monadic expressions. To express type preservation, we should define, for each τ and ε, the monadic counterpart of $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$, being a predicate on $M\mathsf{Exp}$. The key idea is to obtain such predicate by applying a *predicate lifting* [21], that is, a way to lift, for every set X, predicates over X to predicates over MX, adding requirements on the computational effects modeled by the monad. In our case, for each effect type ε, the predicate lifting modularly models the meaning of ε, that is, the computational effects approximated by ε, independently from the set X and the predicate A, as formally detailed below.

For a set X, we denote by $\mathcal{P}(X)$ the poset of all subsets (a.k.a. predicates) on X, ordered by subset inclusion. For a function $f\colon X \to Y$, we have a monotone function $\mathcal{P}_f\colon \mathcal{P}(Y) \to \mathcal{P}(X)$, given by the inverse image: for $A \subseteq Y$, $\mathcal{P}_f(A) = \{x \in X \mid f(x) \in A\}$. That is, $\mathcal{P}_f$ is a predicate transformer, giving, for each predicate A on Y, the weakest condition elements of X should satisfy to be mapped by f in elements satisfying A. These data determine a functor $\mathcal{P}\colon \mathcal{S}et^{\mathrm{op}} \to \mathcal{P}os$, where $\mathcal{P}os$ denotes the category of posets and monotone functions.

▶ **Definition 24** (Interpretation of effect types)**.** *Let* $\mathbb{M} = \langle M, \mu, \eta \rangle$ *be a monad, and* $\mathcal{E} = \langle \mathsf{Eff}, \preceq, \cdot, 1 \rangle$ *an ordered monoid of effect types. Then, an* interpretation *of* $\mathcal{E}$ *in* $\mathbb{M}$ *consists of a family* λ *of monotone functions* $\lambda^{\varepsilon}_X\colon \mathcal{P}(X) \to \mathcal{P}(MX)$*, for every* $\varepsilon \in \mathsf{Eff}$ *and set* X*, such that*

1. $\lambda^{\varepsilon}_X(\mathcal{P}_f(A)) = \mathcal{P}_{Mf}(\lambda^{\varepsilon}_Y(A))$*, for every* $A \subseteq Y$ *and function* $f\colon X \to Y$
2. $\varepsilon \preceq \varepsilon'$ *implies* $\lambda^{\varepsilon}_X(A) \subseteq \lambda^{\varepsilon'}_X(A)$*, for every* $A \subseteq X$*,*
3. $A \subseteq \mathcal{P}_{\eta X}(\lambda^{1}_X(A))$*, for every* $A \subseteq X$*,*
4. $\lambda^{\varepsilon}_{MX}(\lambda^{\varepsilon'}_X(A)) \subseteq \mathcal{P}_{\mu X}(\lambda^{\varepsilon \cdot \varepsilon'}_X(A))$*, for every* $A \subseteq X$*.*

[7] For instance, in Example 22(1), we have $\{\epsilon\} \preceq \{\epsilon, \mathtt{raise}\langle \mathsf{PredZero} \rangle\}$, whereas $\{\epsilon\} \not\preceq \{\mathtt{raise}\langle \mathsf{PredZero} \rangle\}$.

The family $\lambda = (\lambda^{\varepsilon})_{\varepsilon \in \mathsf{Eff}}$ is a family of predicate liftings for the monad $\mathbb{M}$, indexed by effect types. For a subset $A \subseteq X$, the subset $\lambda^{\varepsilon}_X(A) \subseteq MX$ contains monadic elements which agree with A and whose computational effects are described by ε.

Item 1 states that λ^{ε}_X is natural in X, that is, for every $\varepsilon \in \mathsf{Eff}$, we have a natural transformation $\lambda^{\varepsilon} : \mathcal{P} \Rightarrow \mathcal{P} \circ M^{\mathrm{op}}$.[8] The naturality on X ensures that the semantics of each effect type is independent from the specific set X, thus depending only on the functor M.

Item 2 states that λ^{ε}_X is monotone with respect to the order on effects, that is, computational effects described by ε are also described by ε'.

Item 3 states that monadic elements in the image of η_X contain computational effects described by 1, that is, no computational effect.

Finally, in Item 4 we consider elements of M^2X whose computational effects are described by lifting predicates to MX through ε', and then by lifting through ε. By flattening such elements through $\mu_X \colon M^2X \to MX$ we obtain elements whose computational effects are described by the composition $\varepsilon \cdot \varepsilon'$.

▶ Remark 25. The monad $\mathbb{M}$ with an interpretation λ determine a structure on the functor $\mathcal{P}$, which can be described as a graded/parametric monad [14] on $\mathcal{P}$ in an appropriate 2-category (see e.g., [10]). Equivalently, λ determines a graded/parametric monad above $\mathbb{M}$ [25, Def. 2.6] along the fibration obtained from $\mathcal{P}$ by the Grothendieck construction [18].

▶ **Example 26.** Consider the exception monad $\mathbb{E}_{\mathsf{Exc}}$ of Example 1 and the ordered monoid $\langle \wp(\mathsf{Exc} + \{\mathsf{none}\}), \subseteq, \cdot, \{\mathsf{none}\}\rangle$ where $\mathsf{E}_1 \cdot \mathsf{E}_2 = (\mathsf{E}_1 \setminus \{\mathsf{none}\}) \cup \mathsf{E}_2$, if $\mathsf{none} \in \mathsf{E}_1$, and $\mathsf{E}_1 \cdot \mathsf{E}_2 = \mathsf{E}_1$, otherwise. For every $\mathsf{E} \in \wp(\mathsf{Exc} + \{\mathsf{none}\})$, set X, and $A \subseteq X$, the assigment

$$\lambda^{\mathsf{E}}_X(A) = \begin{cases} A + (\mathsf{E} \setminus \{\mathsf{none}\}) & \text{if } \mathsf{none} \in \mathsf{E} \\ \mathsf{E} & \text{otherwise} \end{cases}$$

determines an interpretation of effect types into $\mathbb{E}_{\mathsf{Exc}}$. Intuitively, the interpretation of E requires exceptions possibly raised to be in E, and, if it is allowed that no exception be raised ($\mathsf{none} \in \mathsf{E}$), requires the predicate A to be satisfied.

▶ **Example 27.** Consider the powerset monad $\mathbb{P}$ of Example 2.

1. Taking the ordered monoid $\langle \{0,1\}, \leq, \vee, 0\rangle$, for every set X and $A \subseteq X$, the following assignments determine two interpretations of effect types into $\mathbb{P}$:

$$\forall^1_X(A) = \{B \in PX \mid B \subseteq A\},$$
$$\exists^1_X(A) = \{B \in PX \mid B = \emptyset \text{ or } B \cap A \neq \emptyset\} \textit{and}$$
$$\forall^0_X(A) = \exists^0_X(A) = \{B \in PX \mid B \subseteq A \text{ and } \sharp B \leq 1\}$$

   where $\sharp B$ is the cardinality of B. Intuitively, in both cases, the interpretation of 0 disallows non-determinism, while the interpretation of 1 requires the predicate A to be always satisfied, according to $\forall$, and satisfied in at least one case, according to $\exists$.
2. Taking instead the ordered monoid $\langle \mathbb{N} \cup \{\infty\}, \leq, \cdot, 1\rangle$, we can give a finer version of $\forall$:

$$\forall^n_X(A) = \{B \in PX \mid B \subseteq A \text{ and } \sharp B \leq n\}$$
$$\forall^\infty_X(A) = \{B \in PX \mid B \subseteq A\}$$

   In this way, we can quantify the level of non-determinism in terms of the maximum number of possible outcomes.

[8] Here $M^{\mathrm{op}} \colon \mathcal{S}et^{\mathrm{op}} \to \mathcal{S}et^{\mathrm{op}}$ denotes the functor defined exactly as M but on the opposite category.

Similar interpretations can be defined for the list and subdistribution monads of Example 3.

▶ **Example 28.** Consider the output monad $\mathbb{O}$ of Example 4 for the monoid $\langle A^\infty, \cdot, \epsilon\rangle$ of possibly infinite words over A and the ordered monoid $\langle \mathbb{N}\cup\{\infty\}, \leq, +, 0\rangle$ of effect types. For a word $\sigma \in A^\infty$, we write $|\sigma|$ for its length, which is an element of $\mathbb{N}\cup\{\infty\}$. For every $n \in \mathbb{N}\cup\{\infty\}$, set X and $A \subseteq X$, the assignment $\lambda^n_X(A) = \{\langle\sigma, x\rangle \in OX \mid x \in A, |\sigma| \leq n\}$ determines an interpretation of effect types into $\mathbb{O}$. Intuitively, such interpretation imposes an upper bound (or none) to the length of the outputs.

▶ **Example 29.** Let $\mathcal{E}$ be an ordered monoid of effect types and λ an interpretation of $\mathcal{E}$ into a monad $\mathbb{M}$. Let $\mathcal{E}'$ be another ordered monoid. To give an interpretation of $\mathcal{E}'$ into $\mathbb{M}$, it suffices to give a *lax monoid homomorphism* $f\colon \mathcal{E}' \to \mathcal{E}$, that is, a monotone function $f\colon \langle \mathsf{Eff}', \preceq'\rangle \to \langle \mathsf{Eff}, \preceq\rangle$ such that $1 \preceq f(1')$ and $f(\varepsilon'_1)\cdot f(\varepsilon'_2) \preceq f(\varepsilon'_1\cdot\varepsilon'_2)$. Then, we can define an interpretation ρ of $\mathcal{E}'$ into $\mathbb{M}$ by setting $\rho^{\varepsilon'} = \lambda^{f(\varepsilon')}$ for all $\varepsilon' \in \mathsf{Eff}'$.

Let us fix a monadic operational semantics $\langle \mathbb{M}, \to\rangle$ for a language $\mathcal{L} = \langle \mathsf{Exp}, \mathsf{Val}, \mathsf{ret}\rangle$, a type-and-effect system $\Theta = \langle \mathsf{Ty}, \mathcal{E}, \mathsf{WT}^{\mathsf{E}}, \mathsf{WT}^{\mathsf{V}}\rangle$ for $\mathcal{L}$, and an interpretation λ of $\mathcal{E}$ into $\mathbb{M}$.

Then, we can formally state monadic progress and monadic subject reduction.

▶ **Definition 30** (Monadic Progress). *The type-and-effect system Θ has* monadic progress *if $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ implies either $e = \mathsf{ret}(v)$ for some $v \in \mathsf{Val}$, or $e \to \mathrm{E}$ for some $\mathrm{E} \in M\mathsf{Exp}$.*

▶ **Definition 31** (Monadic Subject Reduction). *The type-and-effect system Θ has* monadic subject reduction *if $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ and $e \to \mathrm{E}$ imply $\mathrm{E} \in \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2})$ for some $\varepsilon_1\cdot\varepsilon_2 \preceq \varepsilon$.*

Monadic progress is standard: a well-typed expression either represents a value or can reduce. Monadic subject reduction, instead, takes into account effects: if an expression of type τ and effect ε reduces to a monadic expression E, then E "has type τ and effect ε" as well, meaning that: ε can be decomposed as $\varepsilon_1\cdot\varepsilon_2$ and E contains computational effects described by ε_1 and expressions of type τ and effect ε_2. In other words, the type τ is preserved and the effect ε is an upper bound of the computational effects produced by the current reduction step, described by ε_1, composed with those produced by future reductions, described by ε_2.

Our next step is expressing type-and-effect soundness. In standard small-step semantics, soundness means that, starting from a well-typed expression, if termination, that is, an expression which cannot be reduced, is reached, then such expression should be a well-typed value. In our monadic operational semantics, termination is conventionally represented by monadic results. Hence, an analogous statement is that, starting from a well-typed expression, if termination, that is, a monadic result, is reached, then this should be a well-typed result, meaning that is satisfies the lifting through the effect type of well-typedness of values.

Again slightly abusing the notation, we will consider predicates $\mathsf{WT}^{\mathsf{V}}_\tau$ on values also as predicates on results. Note that in particular $\mathsf{wrong} \notin \mathsf{WT}^{\mathsf{V}}_\tau$ for all $\tau \in \mathsf{Ty}$.

▶ **Definition 32** (Finitary type-and-effect soundness). *The type-and-effect system Θ is* finitarily sound *if $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ and $[\![e]\!]_\star = R$ imply $R \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_\tau)$.*

This notion of soundness is very general: whenever an expression of type τ and effect ε evaluates to a monadic result, this belongs to the interpretation of ε applied to (the image of) values of type τ. Hence, the nature of the soundness property heavily depends on the interpretation λ of effect types. For instance, considering the interpretations for the powerset monad of Example 27, $\forall$ and $\exists$ induce induce a notion of must-soundness, and may-soundness, respectively: the former ensures that the evaluation of a well-typed expression never reaches wrong, while the latter only that it either diverges or reaches at least a well-typed value.

More specifically, it is not guaranteed that the monadic result is actually a monadic value. Formally, viewing $M\mathsf{Val}$ as a subset of $M\mathsf{Res}$, the inclusion $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau}) \subseteq M\mathsf{Val}$ does not hold in general, as happens for instance with the $\exists$ interpretation. However, we can recover this property when the interpretation λ enjoys an additional condition, as detailed below.

Given $f\colon X \to Y$, the mapping $\mathcal{P}_f\colon \mathcal{P}(Y) \to \mathcal{P}(X)$ has a left adjoint $\mathcal{P}^f\colon \mathcal{P}(X) \to \mathcal{P}(Y)$, that is, a monotone function such that, for every $A \subseteq X$ and $B \subseteq Y$, $\mathcal{P}^f(A) \subseteq B$ if and only if $A \subseteq \mathcal{P}_f(B)$. The function $\mathcal{P}^f$ is the direct image along f, that is, for $A \subseteq X$, $\mathcal{P}^f(A) = \{f(x) \mid x \in A\}$. Then, the following is an easy observation.

▶ **Proposition 33.** *If λ satisfies*
5. $\lambda^{\varepsilon}_{Y}(\mathcal{P}^f(A)) \subseteq \mathcal{P}^{Mf}(\lambda^{\varepsilon}_{X}(A))$ *for $f\colon X \to Y$ and $A \subseteq X$*
then $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau}) \subseteq M\mathsf{Val}$.

Proof. Recall that we are implicitly using an inclusion $\iota^{\mathsf{Res}}_{\mathsf{Val}}\colon \mathsf{Val} \to \mathsf{Res}$. Making it explicit, the thesis becomes $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathcal{P}^{\iota^{\mathsf{Res}}_{\mathsf{Val}}}(\mathsf{WT}^{\mathsf{V}}_{\tau})) \subseteq \mathcal{P}^{\iota^{\mathsf{Res}}_{\mathsf{Val}}}(M\mathsf{Val})$. This follows from $\lambda^{\varepsilon}_{\mathsf{Res}}(\mathcal{P}^{\iota^{\mathsf{Res}}_{\mathsf{Val}}}(\mathsf{WT}^{\mathsf{V}}_{\tau})) \subseteq \mathcal{P}^{M\iota^{\mathsf{Res}}_{\mathsf{Val}}}(\lambda^{\varepsilon}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{\tau})) \subseteq \mathcal{P}^{M\iota^{\mathsf{Res}}_{\mathsf{Val}}}(M\mathsf{Val})$. ◀

Note that the inclusion in Item 5 is actually an equality, since the converse always holds thanks to Item 1 of Definition 24, as $\mathcal{P}^f$ is the left adjoint of $\mathcal{P}_f$. This ensures that a monadic result $\mathrm{R} \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$ contains only values of type τ, hence, in particular, cannot contain wrong. In fact, the $\exists$ interpretation of Example 27 does not satisfy Item 5 of Proposition 33.

From now on, we assume that Θ has monadic progress and monadic subject reduction, and our goal is to prove that they imply type-and-effect soundness.

We first extend the type-and-effect system to configurations, defining $\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon} \subseteq \mathsf{Conf}$ as

$$\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon} = \begin{cases} \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} + \mathsf{WT}^{\mathsf{V}}_{\tau} & \text{if } 1 \preceq \varepsilon \\ \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon} & \text{otherwise} \end{cases}$$

Note that wrong is never a well-typed configuration, while configurations which are values of type τ are well-typed with type τ and effect ε only when ε is larger than 1, that is, the type effect does not force raising effects.

Then, we should extend monadic progress and monadic subject reduction to the reduction relation $\xrightarrow[\mathsf{step}]{}$. However, since it is a total function, it trivially enjoys progress, hence, we only have to deal with subject reduction. In the proof, we also use monadic progress of the monadic reduction $\to$ on expressions to ensure that wrong, which is ill-typed, is not produced.

▶ **Lemma 34.** *If $c \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ and $c \xrightarrow[\mathsf{step}]{} \mathrm{C}$, then $\mathrm{C} \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\varepsilon_2})$ with $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$.*

Proof. We split cases on the shape of c.

- $c = e$. From $e \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ we derive $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$. By monadic progress, either $e = \mathsf{ret}(v)$ or $e \to \mathrm{E}$. In the former case, $\mathrm{C} = \eta_{\mathsf{Conf}}(v)$ and, by Definition 23, $1 \preceq \varepsilon$ and $v \in \mathsf{WT}^{\mathsf{V}}_{\tau}$. Hence, the thesis follows by Definition 24(3), taking $\varepsilon_1 = 1$ and $\varepsilon_2 = \varepsilon$. In the latter case, $\mathrm{C} = \hat{\mathrm{E}} = M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}(\mathrm{E})$ and, by monadic subject reduction, $\mathrm{E} \in \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2})$, with $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$. From $\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2} = \mathcal{P}_{\iota^{\mathsf{Conf}}_{\mathsf{Exp}}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$, by Definition 24(1),

  $$\mathrm{E} \in \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon_2}) = \lambda^{\varepsilon_1}_{\mathsf{Exp}}(\mathcal{P}_{\iota^{\mathsf{Conf}}_{\mathsf{Exp}}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})) = \mathcal{P}_{M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}}(\lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2}))$$

  which implies that $\hat{\mathrm{E}} = M\iota^{\mathsf{Conf}}_{\mathsf{Exp}}(\mathrm{E}) \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$, as needed.
- $c = r$. Since $r \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$, we have $r \neq \mathsf{wrong}$, hence $r = v$ and this implies that $v \in \mathsf{WT}^{\mathsf{V}}_{\tau}$ and $1 \preceq \varepsilon$. By definition of $\xrightarrow[\mathsf{step}]{}$, we also know that $\mathrm{C} = \eta_{\mathsf{Conf}}(v)$, hence, the thesis follows from Definition 24(3) taking $\varepsilon_1 = 1$ and $\varepsilon_2 = \varepsilon$. ◀

Then, we obtain the following result, showing a form of soundness for the multistep reduction on configurations.

▶ **Theorem 35.** *If* $c \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ *and* $c \xrightarrow[\mathsf{step}]{}{}^{\star} \mathrm{C}$*, then* $\mathrm{C} \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$ *with* $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$*.*

Proof. By induction on rules defining $\xrightarrow[\mathsf{step}]{}{}^{\star}$.

(REFL). We have $\mathrm{C} = \eta_{\mathsf{Conf}}(c)$. By Definition 24(3), $c \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon} \subseteq \mathcal{P}_{\eta_{\mathsf{Conf}}}(\lambda^{1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}))$, which implies $\mathrm{C} = \eta_{\mathsf{Conf}}(c) \in \lambda^{1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon})$. This proves the thesis since $1 \cdot \varepsilon \preceq \varepsilon$.

(STEP). We know that $c \xrightarrow[\mathsf{step}]{}{}^{\star} \mathrm{C}_1$ and $\mathrm{C} = \mathrm{C}_1 \gg\!= \mathsf{step}$. By induction hypothesis, $\mathrm{C}_1 \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2})$ with $\varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$. By Lemma 34, we derive $\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2} \subseteq \mathcal{P}_{\mathsf{step}}(\lambda^{\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))$ with $\varepsilon'_1 \cdot \varepsilon'_2 \preceq \varepsilon_2$. Then, using Items 1 and 4 of Definition 24, we have

$$\begin{aligned}
\mathrm{C}_1 \in \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon_2}) &\subseteq \lambda^{\varepsilon_1}_{\mathsf{Conf}}(\mathcal{P}_{\mathsf{step}}(\lambda^{\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))) = \mathcal{P}_{M\mathsf{step}}(\lambda^{\varepsilon_1}_{M\mathsf{Conf}}(\lambda^{\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2})))\\
&\subseteq \mathcal{P}_{M\mathsf{step}}(\mathcal{P}_{\mu_{\mathsf{Conf}}}(\lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))) = \mathcal{P}_{\mu_{\mathsf{Conf}}\circ M\mathsf{step}}(\lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))\\
&= \mathcal{P}_{(\mathsf{step})^{\dagger}}(\lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2}))
\end{aligned}$$

This implies that $\mathrm{C} = (\mathsf{step})^{\dagger}(\mathrm{C}_1) \in \lambda^{\varepsilon_1\cdot\varepsilon'_1}_{\mathsf{Conf}}(\mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon'_2})$, hence the thesis follows observing that $(\varepsilon_1 \cdot \varepsilon'_1) \cdot \varepsilon'_2 \preceq \varepsilon_1 \cdot \varepsilon_2 \preceq \varepsilon$. ◀

▶ **Theorem 36** (Finitary type-and-effect soundness)**.** *If* $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *and* $[\![e]\!]_{\star} = R$ *then* $R \in \lambda^{\varepsilon}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

Theorem 36 states that monadic progress and monadic subject reduction imply soundness with respect to the finitary semantics. To state an analogous result for infinitary semantics, the interpretation of effect types has to take into account the additional structure of the monad.

▶ **Definition 37.** *Let* $\mathbb{M} = \langle M, \sqsubseteq, \mu, \eta \rangle$ *be an* ω*-CPO-ordered monad. An interpretation* λ *of* $\mathcal{E}$ *in* $\mathbb{M}$ *is* ω-CPO-ordered *if, for every effect type* $\varepsilon \in \mathsf{Eff}$*, set* X*, and* $A \subseteq X$*, we have*

1. $\bot_X \in \lambda^{\varepsilon}_X(A)$ *and*
2. *for every* ω*-chain* $(\alpha_n)_{n\in\mathbb{N}}$ *in* MX*,* $\alpha_n \in \lambda^{\varepsilon}_X(A)$ *for all* $n \in \mathbb{N}$ *implies* $\bigsqcup_{n\in\mathbb{N}} \alpha_n \in \lambda^{\varepsilon}_X(A)$*.*

For example, the interpretations in Example 27 are ω-CPO-ordered and also that in Example 28 can be turned into an ω-CPO-ordered one if applied to the pointed output monad. Finally, the construction of Example 29 applies to ω-CPO-ordered interpretations as well.

From now on, we assume that the monad $\mathbb{M}$ has an ω-CPO-ordered structure and the interpretation λ of effect types is ω-CPO-ordered as well.

▶ **Definition 38** (Infinitary type-and-effect soundness)**.** *The type-and-effect system* Θ *is* infinitarily sound *if* $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *implies* $[\![e]\!]_{\infty} \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

Infinitary soundness states that the limit behaviour of an expression of type τ and effect type ε is a monadic result belonging to the interpretation of ε applied to values of type τ. Observations in Proposition 33 applies to infinitary soundness as well.

In order to prove that monadic progress and monadic subject reduction imply infinitary soundness, we first need a simple property of the function $\mathsf{res} = \mathsf{res}^{\dagger}_0$, introduced at page 8, which is at the basis of the definition of the infinitary semantics.

▶ **Lemma 39.** *If* $\mathrm{C} \in \mathsf{WT}^{\mathsf{C}}_{\tau,\varepsilon}$ *then* $\mathsf{res}_0(c) \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

▶ **Theorem 40** (Infinitary type-and-effect soundness)**.** *If* $e \in \mathsf{WT}^{\mathsf{E}}_{\tau,\varepsilon}$ *then* $[\![e]\!]_{\infty} \in \lambda^{\varepsilon}_{\mathsf{Res}}(\mathsf{WT}^{\mathsf{V}}_{\tau})$*.*

6 Example of soundness proof

We show an instance of the technique introduced in the previous section, by proving monadic progress (Theorem 42) and monadic subject reduction (Theorem 45), hence, type-and-effect soundness, for our example. Recall that monadic reduction in Section 4 is parametric on a monad $\mathbb{M}$, and, for each operation op with arity k, a partial function $\mathsf{run}_{op}{:}\mathsf{Val}^k \rightharpoonup M\mathsf{Val}$.

The type-and-effect system defined in Section 4 is an example of Definition 23, where, omitting empty environments and environments in judgments for simplicity:

- Ty is the set of types T as in Figure 5
- $\mathcal{E} = \langle \mathsf{Eff}, \subseteq, \cdot, \{\epsilon\}\rangle$ where Eff is the set of non-empty subsets of Σ^∞
- $\mathsf{WT}^{\mathsf{E}}_{T,E}(e)$ iff $\vdash e : T'!E'$ for some T', E' such that $T'!E' \leq T!E$.
- $\mathsf{WT}^{\mathsf{V}}_{T}(v)$ iff $\vdash v : T'$ for some T' such that $T' \leq T$

In order to prove progress and subject reduction properties, we need a last parameter, that is, an interpretation λ of effect types. Since the proof is parametric on the computational effects raised by operations, these two parameters should agree, as described below.

for each $op\colon T_1 \dots T_n \to T$, and $\overline{v}$ such that $\vdash \overline{v} : \overline{T}'$ and $\overline{T}' \leq \overline{T}$

(RUN) $\quad \mathsf{run}_{op}(\overline{v}) \in \lambda^{\{op\}}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T})$

▶ **Example 41.** We describe interpretations of the effect types suitable for the examples in Section 4. Such interpretations are defined by first mapping[9] the effect types into one of the ordered monoids in Section 5, and then taking the interpretation of the latter into the monad; in this way, as described in Example 29, we get an interpretation of the original effect types. In other words, for an instantiation of the calculus on specific monad and operations, sets of possibly infinite sequences could be reduced to simpler effect types, as exemplified below.

1. In Example 18, we reduce effect types to sets whose elements are either exceptions or none:

$$[\![\epsilon]\!] = \{\mathsf{none}\} \qquad [\![\mathtt{raise}\langle \mathsf{e}\rangle{:}\alpha]\!] = \{\mathsf{e}\} \qquad [\![E]\!] = \bigcup_{\alpha\in E} [\![\alpha]\!]$$

That is, effect types are mapped into those of Example 26, so that, if $[\![E]\!] = \mathsf{E}$, then

$$\lambda^{E}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T}) = \begin{cases} \mathsf{WT}^{\mathsf{V}}_{T} + \mathsf{E} \text{ if } \epsilon \in E \\ \mathsf{E} \text{ otherwise} \end{cases}$$

In this way, monadic values[10] (either values or exceptions) are well-typed if they are either exceptions in E, or, if it is allowed that no exception be raised ($\mathsf{none} \in \mathsf{E}$), well-typed values. Note that an expression such as, e.g., $\mathtt{raise}\langle \mathsf{e}\rangle\mathtt{;raise}\langle \mathsf{e}'\rangle$, gets the effect (reducing to) $\{\mathsf{e}\}$, highlighting the fact that $\mathtt{raise}\langle \mathsf{e}'\rangle$ cannot be reached.

2. In Example 19, the simplest interpretation is to reduce effect types to either 0 or 1:

$[\![\epsilon]\!] = 0$
$[\![\mathtt{choose}{:}\alpha]\!] = 1$
$[\![E]\!] = 1$ if $[\![\alpha]\!] = 1$ for some $\alpha \in E, 0$ otherwise

[9] In all the examples it is easy to see that the mapping is a lax monoid homomorphism.
[10] We explain how the lifting works on values; of course the same applies to expressions and configurations.

That is, effect types are mapped into those of Example 27(1), so that, if $[\![E]\!] = 0$, then

$$\lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \lambda^0_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\mathsf{V} \in P\mathsf{Val} \mid \mathsf{V} \subseteq \mathsf{WT}^{\mathsf{V}}_T \text{ and } \sharp\mathsf{V} \leq 1\}$$

If, instead, $[\![E]\!] = 1$, then we can choose

$$\text{either } \lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \forall^1_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\mathsf{V} \in P\mathsf{Val} \mid \mathsf{V} \subseteq \mathsf{WT}^{\mathsf{V}}_T\}$$
$$\text{or } \lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \exists^1_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\mathsf{V} \in P\mathsf{Val} \mid \mathsf{V} = \emptyset \text{ or } \mathsf{V} \cap \mathsf{WT}^{\mathsf{V}}_T \neq \emptyset\}$$

In this way, monadic values (sets of values, representing possibile results of a computation) are well-typed with a type effect (reducing to) 0 if they have at most one element, and this element, if any, is well-typed; in other words, the computation is deterministic. On the other hand, they are well-typed with a type effect (reducing to) 1 if all the values in the set are well-typed, or there is at least one well-typed value, respectively.

3. A finer interpretation for Example 19 is to reduce effect types to the ordered monoid $\langle \mathbb{N} \cup \{\infty\}, \leq, \cdot, 1\rangle$ of Example 27(2), thus controlling the level of non-determinism:

$$[\![\mathtt{choose}^n]\!] = 2^n \text{ for } n \in \mathbb{N}, [\![\mathtt{choose}^\omega]\!] = \infty$$
$$[\![E]\!] = \sup\{[\![\alpha]\!] \mid \alpha \in E\}$$

Indeed, each call can be seen as a node in a binary tree of choices. In this way, if $[\![E]\!] = k \in \mathbb{N} \cup \{\infty\}$, then monadic values (sets of values, representing possibile results of a computation) are well-typed with E if there are at most 2^k values, hence possible results, in the set and these are all well typed.

4. In Example 21, a possible interpretation of a sequence of $\mathtt{write}\langle\ell\rangle$ is its length:

$$[\![\mathtt{write}\langle\ell\rangle^{\mathtt{n}}]\!] = \mathtt{n} \text{ for } \mathtt{n} \in \mathbb{N} \cup \{\infty\}$$
$$[\![E]\!] = \sup\{[\![\alpha]\!] \mid \alpha \in E\}$$

That is, effect types are mapped into $\mathbb{N} \cup \{\infty\}$ as done in Example 28, so that, if $[\![E]\!] = n$, then $\lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \lambda^n_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\langle\sigma, v\rangle \in O\mathsf{Val} \mid v \in \mathsf{WT}^{\mathsf{V}}_T \text{ and } \mid \sigma \mid \leq n\}$. In this way, an upper bound (or none) is imposed on the length of the produced outputs.

5. A finer interpretation for Example 21 is obtained by taking effect types as they are, and $\lambda^E_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_T) = \{\langle\sigma, v\rangle \in O\mathsf{Val} \mid v \in \mathsf{WT}^{\mathsf{V}}_T, \mathsf{extract}(\sigma) \in E\}$ where, if $\sigma = \langle\ell_1, n_1\rangle \ldots \langle\ell_k, n_k\rangle$, then $\mathsf{extract}(\sigma) = \mathtt{write}\langle\ell_1\rangle \ldots \mathtt{write}\langle\ell_{\mathtt{k}}\rangle$.In this way, effect types can express properties about the order, or the fairness, in which write operations to different output locations can be performed. Similar sophisticated properties can be expressed in cases where different operations can be performed, e.g., reading and updating in the global state monad.

We state now monadic progress and monadic subject reduction for the type-and-effect system in Section 4; as shown in Section 5, they imply monadic soundness. We report only the proof of monadic subject reduction; other proofs and lemmas they depend on are given in [9].

▶ **Theorem 42** (Monadic Progress). *If $e \in \mathsf{WT}^{\mathsf{E}}_{T,E}$ then either $e = \mathtt{return}\ v$ or $e \to_E$.*

The proof of monadic subject reduction uses the standard substitution lemma, and subject reduction for the pure relation $\to_p$ defined in Figure 3. Both properties do not involve any monadic ingredient, and are proved by standard techniques.

▶ **Lemma 43** (Substitution). *If $\Gamma, \overline{x} : \overline{T} \vdash e : T!E$ and $\overline{T'} \leq \overline{T}$, then $\vdash \overline{v} : \overline{T'}$ implies $\Gamma \vdash e[\overline{v}/\overline{x}] : T'!E'$ with $T'!E' \leq T!E$.*

▶ **Lemma 44** (Subject Reduction). *If $\vdash e : T!E$ and $e \to_p e'$ then $\vdash e' : T'!E'$ with $T'!E' \leq T!E$.*

▶ **Theorem 45** (Monadic Subject Reduction). *If* $e \in \mathsf{WT}^{\mathsf{E}}_{T,E}$ *and* $e \to \mathrm{E}$ *then* $\mathrm{E} \in \lambda^{E_1}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E_2})$ *for some* E_1 *and* E_2 *such that* $E_1 \cdot E_2 \subseteq E$.

Proof. From $e \in \mathsf{WT}^{\mathsf{E}}_{T,E}$ we get $\vdash e : T'!E'$ and $T'!E' \leq T!E$. By induction on the reduction rules of Figure 4.

(pure) In this case $e \to_p e'$ and $\mathrm{E} = \eta(e')$. From $\vdash e : T'!E'$ and Lemma 44 we get $\vdash e' : T_1!E_1$ and $T_1!E_1 \leq T'!E'$ and, by transitivity of $\leq$, $e' \in \mathsf{WT}^{\mathsf{E}}_{T,E}$. From Definition 24(3) we derive $\eta(e') \in \lambda^{\{\epsilon\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E})$ with $\{\epsilon\} \cdot E \subseteq E$.

(effect) In this case $e = op(\overline{v})$ and $\mathrm{E} = \mathsf{map}\,(\mathtt{return}\ [\ ])\,\mathrm{V}$, with $\mathrm{V} = \mathsf{run}_{op}(\overline{v})$. From rule (T-OP), $op\colon T_1 \dots T_n \to T'$ and $\vdash \overline{v} : \overline{T}'$ and $\overline{T}' \leq \overline{T}$ and $E' = \{op\}$. Hence, by rule (RUN), $\mathrm{V} \in \lambda^{\{op\}}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T'})$. Let $f\colon \mathsf{Val} \to \mathsf{Exp}$ be defined by $f(v) = \mathtt{return}\ v$, then $\mathrm{E} = \mathsf{map}\,(\mathtt{return}\ [\ ])\,\mathrm{V} = Mf(\mathrm{V})$. From Definition 24(1) and $\mathsf{WT}^{\mathsf{V}}_{T'} = \mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}})$

$$\mathrm{V} \in \lambda^{\{op\}}_{\mathsf{Val}}(\mathsf{WT}^{\mathsf{V}}_{T'}) = \lambda^{\{op\}}_{\mathsf{Val}}(\mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}}) = \mathcal{P}_{Mf}(\lambda^{\{op\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}}))$$

This implies $\mathrm{E} = Mf(v) \in \lambda^{\{op\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}})$. Since $T' \leq T$, $\mathsf{WT}^{\mathsf{E}}_{T',\{\epsilon\}} \subseteq \mathsf{WT}^{\mathsf{E}}_{T,\{\epsilon\}}$, hence by monotonicity of $\lambda^{\{op\}}_{\mathsf{Exp}}$, we get $\mathrm{E} \in \lambda^{\{op\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,\{\epsilon\}})$, with $\{op\}\cdot\{\epsilon\} = \{op\} \subseteq E$.

(ret) In this case $e = \mathtt{do}\ x = \mathtt{return}\ v\mathtt{;}\ e'$ and $\mathrm{E} = \eta(e'[v/x])$. From rules (T-DO) and (T-RET), $\vdash v : T_1$ and $x : T_1' \vdash e' : T'!E'$, with $T_1 \leq T_1'$. By Lemma 43, we get $\vdash e'[v/x] : T''!E''$ with $T''!E'' \leq T'!E'$. Hence, $T''!E'' \leq T!E$ and so $e'[v/x] \in \mathsf{WT}^{\mathsf{E}}_{T,E}$. Finally, from Definition 24(3), $\eta(e'[v/x]) \in \lambda^{\{\epsilon\}}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E})$ with $\{\epsilon\}\cdot E = E$.

(do) In this case $e = \mathtt{do}\ x = e_1\mathtt{;}\ e_2$ and $\mathrm{E} = \mathsf{map}\,(\mathtt{do}\ x = [\ ]\mathtt{;}\ e_2)\,\mathrm{E}_1$ and $e_1 \to \mathrm{E}_1$. From rule (T-DO), $\vdash e_1 : T_1!E_1$ and $x : T_1' \vdash e_2 : T'!E_2$ with $E' = E_1 \cdot E_2$ and $T_1 \leq T_1'$. Hence, from $e_1 \in \mathsf{WT}^{\mathsf{E}}_{T_1,E_1}$, by induction hypothesis we get that $\mathrm{E}_1 \in \lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T_1,E_2'})$ with $E_1' \cdot E_2' \subseteq E_1$. Let $f\colon \mathsf{Exp} \to \mathsf{Exp}$ be defined by $f(\hat{e}) = \mathtt{do}\ x = \hat{e}\mathtt{;}\ e_2$, hence $\mathrm{E} = Mf(\mathrm{E}_1)$. By rule (T-DO), we know that $\hat{e} \in \mathsf{WT}^{\mathsf{E}}_{T_1,\hat{E}}$ implies $f(\hat{e}) \in \mathsf{WT}^{\mathsf{E}}_{T',\hat{E}\cdot E_2} \subseteq \mathsf{WT}^{\mathsf{E}}_{T,\hat{E}\cdot E_2}$, that is, $\mathsf{WT}^{\mathsf{E}}_{T_1,\hat{E}} \subseteq \mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T,\hat{E}\cdot E_2})$. From Definition 24(1) and monotonicity of $\lambda^{E_1'}_{\mathsf{Exp}}$ we get

$$\mathrm{E}_1 \in \lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T_1,E_2'}) \subseteq \lambda^{E_1'}_{\mathsf{Exp}}(\mathcal{P}_f(\mathsf{WT}^{\mathsf{E}}_{T,E_2'\cdot E_2})) = \mathcal{P}_{Mf}(\lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E_2'\cdot E_2}))$$

that is, $\mathrm{E} = Mf(\mathrm{E}_1) \in \lambda^{E_1'}_{\mathsf{Exp}}(\mathsf{WT}^{\mathsf{E}}_{T,E_2'\cdot E_2})$, and we get the thesis since $E_1'\cdot E_2'\cdot E_2 \subseteq E_1\cdot E_2 \subseteq E'$. ◀

The results hold for the core calculus in Figure 2, for an arbitrary family Σ of operations. The calculus, the type system and the proofs can be modularly extended by just considering cases for additional constructs, as we will do in Section 7 for handlers. Extending the subtyping relation, instead, requires some care to preserve the needed properties.[11]

7 Handlers

We extend Λ_Σ with *handlers*, showing how our framework can deal with more sophisticated language features and, at the same time, how proofs can be modularly extended. In particular, it is important to illustrate that monadic semantics can incorporate handlers. Constructs and terminology are inspired by those for algebraic effects, see, e.g., [40]; however, the approach is different since our calculus, being based on generic effects, has no explicit continuations.

[11] For instance, adding $\mathtt{Bot} \leq T$ for all T as in Example 18 is sound since $\mathtt{Bot}$ is an empty type.

$$
\begin{array}{lcll}
e & ::= & \ldots \mid \texttt{handle}\ e\ \texttt{with}\ h & \text{expression with handler} \\
h & ::= & \overline{c}, x \mapsto e & \text{handler} \\
c & ::= & op(\overline{x}) \mapsto_{\mu} e & \text{clause} \\
\mu & ::= & \texttt{c} \mid \texttt{s} & \text{mode}
\end{array}
$$

Figure 7 Syntax of handlers.

The syntax is reported in Figure 7. A handler specifies a *final expression*, and a sequence of *clauses*, assumed to be a map, that is, there can be at most one clause for an operation. Such a clause, if any, handles a call of the operation by executing the clause expression. After that, the final expression is either executed or not depending on the *mode*, either `c` or `s`, for "continue" and "stop", respectively. As illustrated in the following examples, a `c`-clause replaces an effect with an alternative behaviour in a continuous manner, whereas in `s`-clauses handling the computational effect interrupts the normal flow of execution.

$$h = \overline{c}, x \mapsto e'$$

$$\text{(WITH-DO)}\ \frac{}{\texttt{handle do}\ y = e_1\texttt{;}\ e_2\ \texttt{with}\ h \to_p \texttt{handle}\ e_1\ \texttt{with}\ \overline{c}, y \mapsto (\texttt{handle}\ e_2\ \texttt{with}\ h)}$$

$$\text{(WITH-RET)}\ \frac{}{\texttt{handle return}\ v\ \texttt{with}\ h \to_p \texttt{do}\ x = \texttt{return}\ v\texttt{;}\ e'}$$

$$\text{(WITH-CONTINUE)}\ \frac{}{\texttt{handle}\ op(\overline{v})\ \texttt{with}\ h \to_p \texttt{do}\ x = e[\overline{v}/\overline{x}]\texttt{;}\ e'}\ op(\overline{x}) \mapsto_{\texttt{c}} e \in \overline{c}$$

$$\text{(WITH-STOP)}\ \frac{}{\texttt{handle}\ op(\overline{v})\ \texttt{with}\ h \to_p e[\overline{v}/\overline{x}]}\ op(\overline{x}) \mapsto_{\texttt{s}} e \in \overline{c}$$

$$\text{(WITH-FWD)}\ \frac{}{\texttt{handle}\ op(\overline{v})\ \texttt{with}\ h \to_p \texttt{do}\ x = op(\overline{v})\texttt{;}\ e'}\ op \notin \overline{c}$$

$$\text{(WITH-CTX)}\ \frac{e \to_p e'}{\texttt{handle}\ e\ \texttt{with}\ h \to_p \texttt{handle}\ e'\ \texttt{with}\ h}$$

Figure 8 Pure reduction with handlers.

The pure reduction extended with handlers is shown in Figure 8. The behaviour of an expression with handler depends on the shape of the handled expression.

In case of a `do` composition of two subexpressions, the `do` is eliminated by reducing to the first subexpression with as final expression the second one; clauses are propagated to both the subexpressions. In case of a `return`, the handler is eliminated by reducing to the `do` composition of the handled expression and the final expression.

In case of an operation call, the behaviour depends on whether a matching clause is found or not. If it is found, then the clause expression is executed, after replacing parameters by arguments, as shown in rules (WITH-CONTINUE) and (WITH-STOP). In a `c`-clause, the final expression is executed as well. If there is no matching clause, instead, the handler is eliminated, by reducing to the `do` composition of the operation call and the final expression. The outcome is that the operation call is forwarded to be possibly handled by an outer level. gFinally, the contextual rule is as expected.

To extend the type-and-effect system, we rely on *filter functions* associated to handlers, which describe how they transform effects, by essentially replacing operations matching some clause with the effect of the clause expression.

To this end, first we define a *(handler) filter* H to be the information about transforming effects which can be extracted from a handler, as shown in the top section of Figure 9. Then, given a filter H, we define the associated function $\widehat{\mathcal{F}}_H\colon \mathsf{Eff} \to \mathsf{Eff}$. This function, as shown in the bottom section of Figure 9, is obtained on top of the function $\mathcal{F}_H\colon \Sigma^\infty \to \mathsf{Eff}^\infty$ which transforms a single possibly infinite sequence of operations into a possibly infinite sequence of effects. The latter is transformed into a unique effect by taking the possibly infinite concatenation of its elements, denoted $\bullet^\infty\colon \mathsf{Eff}^\infty \to \mathsf{Eff}$. Finally, the function is extended to effects (sets of sequences) in the obvious way.

$$\begin{array}{lcll} H & ::= & \overline{C}, E & \text{filter} \\ C & ::= & op \mapsto_\mu E & \text{clause filter} \end{array}$$

$H = op_1 \mapsto_{\mu_1} E_1 \dots op_n \mapsto_{\mu_n} E_n, E$

$\mathcal{F}_H\colon \Sigma^\infty \to \mathsf{Eff}^\infty$ coinductively defined by:

$\mathcal{F}_H(\epsilon) = E$

$\mathcal{F}_H(op_i{:}\alpha) = E_i{:}\mathcal{F}_H(\alpha) \qquad i \in 1..n, \mu_i = \mathsf{c}$

$\mathcal{F}_H(op_i{:}\alpha) = E_i \qquad i \in 1..n, \mu_i = \mathsf{s}$

$\mathcal{F}_H(op{:}\alpha) = \{op\}{:}\mathcal{F}_H(\alpha) \qquad op \neq op_i \text{ for all } i \in 1..n$

$\widehat{\mathcal{F}}_H\colon \mathsf{Eff} \to \mathsf{Eff}$

$\widehat{\mathcal{F}}_H(E) = \bigcup_{\alpha \in E} \{\bullet^\infty \mathcal{F}_H(\alpha)\}$

Figure 9 Filters.

$$\text{(T-WITH)}\ \frac{\Gamma \vdash e : T!E \quad \Gamma;\, T' \vdash h : T''!H}{\Gamma \vdash \texttt{handle}\ e\ \texttt{with}\ h : T''!\mathcal{F}_H(E)}\ T \leq T'$$

$$\text{(T-HANDLER)}\ \frac{\Gamma, x : T \vdash e' : T'!E' \quad \Gamma;\, T'' \vdash c_i : C_i}{\Gamma;\, T \vdash c_1 \dots c_n, x \mapsto e' : T''!C_1 \dots C_n, E'}\ T' \leq T''$$

$$\text{(T-CONTINUE)}\ \frac{\Gamma, \overline{x}{:}\overline{T} \vdash e : T''!E'}{\Gamma;\, T' \vdash op(\overline{x}) \mapsto_{\mathsf{c}} e : op \mapsto_{\mathsf{c}} E'}\ \begin{array}{l} op{:}\,\overline{T} \to T \\ T'' \leq T \end{array}$$

$$\text{(T-STOP)}\ \frac{\Gamma, \overline{x}{:}\overline{T} \vdash e : T''!E'}{\Gamma;\, T' \vdash op(\overline{x}) \mapsto_{\mathsf{s}} e : op \mapsto_{\mathsf{s}} E'}\ \begin{array}{l} op{:}\,\overline{T} \to T \\ T'' \leq T' \end{array}$$

Figure 10 Typing rules for handlers.

In Figure 10 we show the typing rules for expressions with handlers. In rule (T-WITH), in order to typecheck an expression with handler, first we get the type and effect of the handled expression. The type is used to typecheck the handler, as (subtype of the) type of the parameter of the final expression, see rule (T-HANDLER). Typechecking the handler we get a type, being that of the final expression, which will be the type of the whole expression. Moreover, we extract from the handler a filter, which is used to transform the effect E of the handled expression, getting the resulting effect of the whole expression. In detail, as formally described in Figure 9, the filter transforms any sequence of operations in E by replacing the first operation matching some clause, if any, with the effect of the clause expression; then, the remaining sequence is disregarded if the clause is s, otherwise filtered in turn. If the sequence to be filtered is finite, and no matching s-clause is found, then the final effect is appended in the end.

In rule (T-HANDLER), as said above, the type on the left of the judgment is used as type of the parameter of the final expression, whose type will be returned by the handler. This type is also needed to typecheck s-clauses, see below. The filter extracted from the handler consists in a clause filter for each clause, and the effect of the final expression.

For each clause, the extracted filter consists of the operation name, mode, and effect of the expression, as shown in rules (T-CONTINUE) and (T-STOP). A c-clause is meant to provide alternative code to be executed before the final expression, hence the type of the clause expression should be (a subtype of) the return type of the operation. In a s-clause, instead, the result of the clause expression becomes that of the whole expression with handler, hence the type of the former should be (a subtype of) the latter.

▶ **Example 46.** We show handlers for some of the previous examples. A handler of shape $\overline{c}, x \mapsto \texttt{return}\ x$ is abbreviated by $\overline{c}$.

1. Set $h = \texttt{raise}\langle\mathsf{PredZero}\rangle() \mapsto_{\mathsf{s}} \texttt{return}\ 0$. Then

$$\begin{array}{lcl} \texttt{handle}\ \mathsf{predfun}\,0\ \texttt{with}\ h & \Rightarrow^{\star} & \texttt{handle}\ \texttt{raise}\langle\mathsf{PredZero}\rangle\ \texttt{with}\ h \\ & \Rightarrow & \texttt{return}\ 0 \\ & \Rightarrow & 0 \end{array}$$

   As shown in Example 22(1), we get the judgment $\emptyset \vdash \mathsf{predfun}\,0 : \texttt{Nat}!\{\epsilon, \texttt{raise}\langle\mathsf{PredZero}\rangle\}$. On the other hand, with the handler we get

$$\emptyset \vdash \texttt{handle}\ \mathsf{predfun}\,0\ \texttt{with}\ h : \texttt{Nat}!\{\epsilon\}$$

   since $\widehat{\mathcal{F}}_H(\{\epsilon, \texttt{raise}\langle\mathsf{PredZero}\rangle\}) = \{\epsilon\}$ where $H = \texttt{raise}\langle\mathsf{PredZero}\rangle \mapsto_{\mathsf{s}} \{\epsilon\}, \{\epsilon\}$ is the filter extracted from h. As the reader could expect, an s-clause is appropriate in this case. With a c-clause, see rule (T-CONTINUE), the type of the clause expression should be (a subtype of) the return type of the operation, which is `Bot`. Since no value has type `Bot`, no value could be returned, as already noted in [35].[12]

2. Assuming the function even: `Nat` $\to$ `Bool` checking the parity of a number, set

$$h_1 = \texttt{write}\langle\ell'\rangle(x) \mapsto_{\mathsf{c}} \texttt{write}\langle\ell\rangle(x)$$
$$h_2 = \texttt{write}\langle\ell'\rangle(x) \mapsto_{\mathsf{c}} \texttt{if}\ \mathsf{even}(x)\ \texttt{then return}\ x\ \texttt{else write}\langle\ell\rangle(x)$$

   Then

$$[\![\texttt{handle}\ \mathsf{wfun}^{\uparrow}\ 0\ \texttt{with}\ h_1]\!]_{\infty} = \langle\langle\ell,0\rangle\cdot\langle\ell,0\rangle\cdot\langle\ell,1\rangle\cdot\langle\ell,1\rangle\cdot\ldots\cdot\langle\ell,n\rangle\cdot\langle\ell,n\rangle\cdot\ldots,\bot\rangle$$
$$[\![\texttt{handle}\ \mathsf{wfun}^{\uparrow}\ 0\ \texttt{with}\ h_2]\!]_{\infty} = \langle\langle\ell,0\rangle\cdot\langle\ell,1\rangle\cdot\langle\ell,1\rangle\cdot\ldots\langle\ell,2k\rangle\cdot\langle\ell,2k{+}1\rangle\cdot\langle\ell,2k{+}1\rangle\cdot\ldots,\bot\rangle$$
$$[\![\texttt{handle}\ \mathsf{wfun}^{\downarrow}\ n\ \texttt{with}\ h_1]\!]_{\star} = \langle\langle\ell,n\rangle\cdot\langle\ell,n\rangle\cdot\ldots\cdot\langle\ell,0\rangle\cdot\langle\ell,0\rangle,\texttt{unit}\rangle$$
$$[\![\texttt{handle}\ \mathsf{wfun}^{\downarrow}\ 2k\ \texttt{with}\ h_2]\!]_{\star} = \langle\langle\ell,2k\rangle\cdot\langle\ell,2k{-}1\rangle\cdot\langle\ell,2k{-}1\rangle\ldots\cdot\langle\ell,1\rangle\cdot\langle\ell,1\rangle\cdot\langle\ell,0\rangle,\texttt{unit}\rangle$$

   In this case, a c-clause is appropriate, since the aim is to continuously handle the $\texttt{write}\langle\ell'\rangle$ operation. By the typing judgments shown in Example 22(3), we get

$$\emptyset \vdash \mathsf{wfun}^{\uparrow}\,0 : \texttt{Unit}!\{(\texttt{write}\langle\ell\rangle\cdot\texttt{write}\langle\ell'\rangle)^{\omega}\}$$
$$\emptyset \vdash \mathsf{wfun}^{\downarrow} : \texttt{Unit}!\{(\texttt{write}\langle\ell\rangle\cdot\texttt{write}\langle\ell'\rangle)^{\mathtt{n}} \mid \mathtt{n} \geq 1\}$$

[12] Hence, the clause expression could only be another raise or a diverging expression.

On the other hand, with the handler we get, with $\alpha ::= \epsilon \mid \texttt{write}\langle\ell\rangle$

$$\emptyset \vdash \texttt{handle wfun}^{\uparrow}\, 0 \texttt{ with } h_1 : \texttt{Unit!}\{(\texttt{write}\langle\ell\rangle \cdot \texttt{write}\langle\ell\rangle)^{\omega}\}$$
$$\emptyset \vdash \texttt{handle wfun}^{\uparrow}\, 0 \texttt{ with } h_2 : \texttt{Unit!}\{\alpha \cdot \texttt{write}\langle\ell\rangle)^{\omega}\}$$
$$\emptyset \vdash \texttt{wfun}^{\downarrow}\, n : \texttt{Unit!}\{\alpha \cdot \texttt{write}\langle\ell'\rangle)^{\texttt{n}} \mid \texttt{n} \geq 1\}$$

As already noted, effect types only provide a static approximation of the computational effects; notably, in the last two judgments, the effect type contains other sequences besides the two which can be actually performed, depending on the argument.

Soundness for handlers. The results of Section 6 can be extended to handlers. For monadic subject reduction we only need to show subject reduction for the newly introduced rules, since they are pure. The proofs of the results are in the extended version [9].

▶ **Lemma 47** (Monadic Progress for handlers). *Set e of shape* `handle _ with _`.
If $\vdash e : T!E$ then $e \to \mathrm{E}$ for some $\mathrm{E} \in M\mathsf{Exp}$.

▶ **Lemma 48** (Subject Reduction for handlers). *Set e of shape* `handle _ with _`.
If $\vdash e : T!E$ and $e \to_p e'$, then $\vdash e' : T'!E'$ such that $T'!E' \leq T!E$.

8 Related work and conclusion

Monadic semantics. The idea that monads can model computational effects in programming languages goes back to the pioneering Moggi's work [30, 31]. He showed that one can use (strong) monads to organise the denotational semantics of effectful languages, interpreting impure expressions as functions (actually arrows of an arbitrary category) returning monadic values, which can be sequenced by Kleisli composition. However, the structure of a monad does not include any operation for actually raising computational effects, which thus need to be defined ad-hoc in specific instances. Moreover, monads are difficult to combine, requiring non trivial notions like monad transformers [27, 22].

To overcome these difficulties and make the model closer to the syntax, Plotkin and Power [33, 34, 35] introduced algebraic effects which, instead, explicitly consider operations to raise computational effects. These can be interpreted by additional structure on the monad and, moreover, when equipped with an equational theory, they actually determine a monad, which provides a syntactic model for the language. Thus, one reduces the problem of combining monads to the much easier problem of combining theories [20], greatly increasing modularity.

An alternative, essentially equivalent, way of interpreting algebraic operations is by means of runners, a.k.a. comodels [39, 36, 44, 2]. Roughly, runners describe how operations are executed by the system, that is, how they transform the environment where they are run. This essentially amounts to giving an interpretation of operations in the state monad. More general runners, where the system is modelled in a more expressive way, are considered by [2], where the state monad is combined with errors and system primitive operations.

On the operational side, algebraic effects are typically treated as uninterpreted operations, that is, the evaluation process just builds a tree of operation calls [23, 42, 48, 40]. Monadic operational semantics for λ-calculi with algebraic effects are also considered, mainly in the form of a monadic definitional interpreter (see, e.g., [27, 12, 15, 11, 8]). That is, they directly define a function from expressions to monadic values, which essentially corresponds to our infinitary semantics. Small-step approaches are also considered by [16, 17]. The former tackles a different problem, that is, studying monadic rewriting systems, which require the

use of sophisticated relational techniques, and thus restricts the class of available monads. We can avoid these difficulties since we focus on deterministic rewriting, which can be addressed using just sets and functions. The latter, instead, studies a specific calculus where, as already noticed, the way sequences of steps are constructed is very close to ours; however, they do not need to introduce wrong in configurations, as type errors are prevented syntactically.

Type-and-effect systems. Type-and-effect systems, or simply effect systems [45, 32, 46, 29, 25], are the most popular way of statically controlling computational effects. Many have been designed for specific notions of computational effect and implemented in mainstream programming languages, the most well-known being the mechanism of Java checked exceptions. Katsumata [25] recognized that effect systems share a common algebraic structure, notably they form an ordered monoid, and gave them denotational semantics through parametric monads, using a structure equivalent to our notion of interpretation (see Remark 25).

Effect handlers. Plotkin and Pretnar [37, 38] introduced effect handlers as a generalisation of exception handling mechanisms. They are an extremely powerful programming abstraction, allowing to describe the semantics of algebraic operations in the language itself, thus enabling the simulation of several effectful programs, such as stream redirection or cooperative concurrency [38, 24, 4, 40]. When a call to an algebraic operation is caught, the alternative code can resume the original computation using a form of continuation-passing style. Other forms of handlers have been considered, notably, shallow handlers [24, 19], where only the first call to an operation is handled. Our handlers are inspired by those for algebraic effects, see, e.g., [40]; however, the approach is different since our calculus has no explicit continuations.

Summary. In the research on foundations of programming languages, it is a routine task to describe execution through a small-step reduction, and prove progress and subject reduction for the type system. Can this be smoothly combined with the long-established approach where computational effects are modularly modeled by a monad, so to enjoy all the advantages of separation of concerns? The answer provided in this paper is yes. Notably, we provide a meta-theory defining abstract notions of monadic small-step semantics and type-and-effect system, and prove that type-and-effect soundness is implied by progress and subject reduction properties, with an inductive argument similar to the standard one.

This overall achievement relies on two key specific contributions. On one hand, we provide a canonical way to construct, on top of a monadic reduction, a small-step operational semantics where computations, even though always represented by infinite sequences, can be distinguished as either non-terminating, or successfully terminating, or stuck. On the other hand, we provide a formal model of the "meaning" of effect types, independent in principle from the underlying language and type system.

Discussion and future work. The way we define the "transitive closure" of a monadic reduction, which is a relation from a set to a different one, is similar, as said, to that proposed by [17]. Notably, such reduction is assumed to be deterministic, since starting from an arbitrary relation would require a relational extension of the monad [3]. Confluence as well would require strong assumptions on the monad, notably some form of commutativity, ruling out most of the relevant examples. Moreover, the aim here is to prove soundness for programming languages, which typically adopt a deterministic evaluation strategy. Differently from [17], we provide a language independent definition; moreover, whereas they consider an intrinsically total reduction, in this paper, as mentioned above, we address the additional problem to characterize stuck computations, as needed to express soundness.

In our framework, non-termination is always possibile, rather than be considered as an effect. This is essentially a choice we made, possibly influenced by the fact that in standard soundness we have three possible outcomes: non-termination, termination with a value, and stuck. The coinductive Delay monad [5] could be an alternative approach to define the infinitary semantics, assuming a way to be combined with the monad modeling computational effects, that is, a distributive law. The relationship between these two approaches, as far as we know, is not clear, and is an interesting direction to be investigated.

Our definition of ω-CPO-ordered monad is given for monads on $\mathcal{S}et$. A challenging and relevant problem is to consider a category different from $\mathcal{S}et$; our feeling is that the notion could be generalized by considering a monad $\mathbb{M}$ on a category $\mathcal{C}$ such that the Kleisli category $\mathcal{C}^{\mathbb{M}}$ is CPO-enriched.

In this paper, where the focus is different, we did not study decidability of the type-and-effect system; we did not even provide a syntactic representation of effects, which are considered semantic entities, notably possibly infinite sets of possibly infinite sequences. Of course decidability is a very important issue to be investigated; the first step should then be to choose a finite representation, e.g., by means of a system of guarded equations.

We illustrated our approach by a lambda-calculus with generic effects. Clearly, it would be important to investigate how other calculi can be formalized as to take advantage of the meta-theory. Notably, we plan to apply the approach to an object-oriented calculus.

Finally, here we considered non-standard handlers, as our calculus is based on generic effects and so it does not use explicit continuations. Hence, it would be nice to investigate the precise relationship between them and handlers for algebraic effects used in the literature.

References

1 Jirí Adámek, Nathan J. Bowler, Paul Blain Levy, and Stefan Milius. Coproducts of monads on set. *CoRR*, abs/1409.3804, 2014. arXiv:1409.3804.

2 Danel Ahman and Andrej Bauer. Runners in action. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 29–55. Springer, 2020. doi:10.1007/978-3-030-44914-8_2.

3 Michael Barr. Relational algebras. In S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney, and S. Swierczkowski, editors, *Reports of the Midwest Category Seminar IV*, number 137 in Lecture Notes in Mathematics, pages 39–55, Berlin, Heidelberg, 1970. Springer Berlin Heidelberg.

4 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. doi:10.1016/J.JLAMP.2014.02.001.

5 Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005. doi:10.2168/LMCS-1(2:1)2005.

6 Francesco Dagnino. A meta-theory for big-step semantics. *ACM Transactions on Computational Logic*, 23(3):20:1–20:50, 2022. doi:10.1145/3522729.

7 Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 169–196. Springer, 2020. doi:10.1007/978-3-030-44914-8_7.

8 Francesco Dagnino and Francesco Gavazzo. A fibrational tale of operational logical relations: Pure, effectful and differential. *Logical Methods in Computer Science*, 20(2), 2024. doi:10.46298/LMCS-20(2:1)2024.

9 Francesco Dagnino, Paola Giannini, and Elena Zucca. Monadic type-and-effect soundness (extended version). *CoRR*, 2025. URL: http://arxiv.org/abs/2504.10159.

10 Francesco Dagnino and Giuseppe Rosolini. Doctrines, modalities and comonads. *Mathematical Structures in Computer Science*, pages 1–30, 2021. doi:10.1017/S0960129521000207.

11 Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022. doi:10.1145/3498692.

12 David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):12:1–12:25, 2017. doi:10.1145/3110256.

13 Samuel Eilenberg and John C. Moore. Adjoint functors and triples. *Illinois Journal of Mathematics*, 9(3):381–398, 1965. doi:10.1215/ijm/1256068141.

14 Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a formal theory of graded monads. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures, 19th International Conference, FoSSaCS 2016*, volume 9634 of *Lecture Notes in Computer Science*, pages 513–530. Springer, 2016. doi:10.1007/978-3-662-49630-5_30.

15 Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 452–461. ACM, 2018. doi:10.1145/3209108.3209149.

16 Francesco Gavazzo and Claudia Faggian. A relational theory of monadic rewriting systems, part I. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470633.

17 Francesco Gavazzo, Riccardo Treglia, and Gabriele Vanoni. Monadic intersection types, relationally. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024*, volume 14576 of *Lecture Notes in Computer Science*, pages 22–51. Springer, 2024. doi:10.1007/978-3-031-57262-3_2.

18 Alexander Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental*, pages 145–194. Springer, 1971.

19 Daniel Hillerström and Sam Lindley. Shallow effect handlers. In Sukyoung Ryu, editor, *Proceedings of the 16th Asian Symposium on Programming Languages and Systems, APLAS 2018*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435. Springer, 2018. doi:10.1007/978-3-030-02768-1_22.

20 Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006. doi:10.1016/J.TCS.2006.03.013.

21 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

22 Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51-52):4441–4466, 2010. doi:10.1016/J.TCS.2010.09.011.

23 Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *Proceedings of the 25th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2010*, pages 209–218. IEEE Computer Society, 2010. doi:10.1109/LICS.2010.29.

24 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

25 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In Suresh Jagannathan and Peter Sewell, editors, *Proceedings of the 41st ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 2014*, pages 633–646. ACM, 2014. doi:10.1145/2535838.2535846.

26 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

27 Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In Ron K. Cytron and Peter Lee, editors, *Proceedings of the 22nd ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 1995*, pages 333–343. ACM Press, 1995. doi:10.1145/199448.199528.

28 Ernest G. Manes. *Algebraic Theories*. Graduate Texts in Mathematics. Springer, 1976. doi:10.1007/978-1-4612-9860-1.

29 Daniel Marino and Todd D. Millstein. A generic type-and-effect system. In Andrew Kennedy and Amal Ahmed, editors, *TLDI'09: Types in Languages Design and Implementatio*, pages 39–50. ACM Press, 2009. doi:10.1145/1481861.1481868.

30 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 1989*, pages 14–23. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39155.

31 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

32 Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer, 1999. doi:10.1007/3-540-48092-7_6.

33 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FoSSaCS 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. doi:10.1007/3-540-45315-6_1.

34 Gordon D. Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. doi:10.1007/3-540-45931-6_24.

35 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

36 Gordon D. Plotkin and John Power. Tensors of comodels and models for operational semantics. In Andrej Bauer and Michael W. Mislove, editors, *The 24th Conference on Mathematical Foundations of Programming Semantics, MFPS 2008*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 295–311. Elsevier, 2008. doi:10.1016/J.ENTCS.2008.10.018.

37 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems - 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.

38 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.

39 A. John Power and Olha Shkaravska. From comodels to coalgebras: State and arrays. In Jirí Adámek and Stefan Milius, editors, *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, CMCS 2004*, volume 106 of *Electronic Notes in Theoretical Computer Science*, pages 297–314. Elsevier, 2004. doi:10.1016/J.ENTCS.2004.02.041.

40 Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In Dan R. Ghica, editor, *The 31st Conference on Mathematical Foundations of Programming Semantics, MFPS 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, 2015. doi:10.1016/J.ENTCS.2015.12.003.

41 Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

42 Alex Simpson and Niels F. W. Voorneveld. Behavioural equivalence via modalities for algebraic effects. *ACM Transactions on Programming Languages and Systems*, 42(1):4:1–4:45, 2020. doi:10.1145/3363518.

43 Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972. doi:10.1016/0022-4049(72)90019-9.

44 Tarmo Uustalu. Stateful runners of effectful computations. In Dan R. Ghica, editor, *The 31st Conference on Mathematical Foundations of Programming Semantics, MFPS 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 403–421. Elsevier, 2015. doi:10.1016/J.ENTCS.2015.12.024.

45 Philip Wadler. The marriage of effects and monads. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *3rd ACM SIGPLAN International Conference on Functional Programming, ICFP 1998*, pages 63–74. ACM, 1998. doi:10.1145/289423.289429.

46 Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic*, 4(1):1–32, 2003. doi:10.1145/601775.601776.

47 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

48 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.