An Effectful Object Calculus

Francesco Dagnino
DIBRIS, Università di Genova, Italy

Paola Giannini
DiSSTE, Università del Piemonte Orientale, Italy

Elena Zucca
DIBRIS, Università di Genova, Italy

Abstract

We show how to smoothly incorporate in the object-oriented paradigm constructs to raise, compose, and handle effects in an arbitrary monad. The underlying pure calculus is meant to be a representative of the last generation of OO languages, and the effectful extension is manageable enough for ordinary programmers; notably, constructs to raise effects are just special methods. We equip the calculus with an expressive type-and-effect system, which, again by relying on standard features such as inheritance and generic types, allows a simple form of effect polymorphism. The soundness of the type-and-effect system is expressed and proved by a recently introduced technique, where the semantics is formalized by a one-step reduction relation from language expressions into monadic ones, so that it is enough to prove progress and subject reduction properties on this relation.

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

Keywords and phrases Object calculi, handlers, type-and-effect systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.8

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2504.15936` [11]

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

Every modern programming language needs to support a wide range of computational effects, such as exceptions, input/output, interaction with a memory and classical or probabilistic non-determinism. Several approaches have been proposed for designing language constructs dealing with different computational effects in a uniform and principled way. The monad pattern [31, 32, 46] and algebraic effects and handlers [34, 35, 36, 38, 39, 6, 41] stand out for their impact on real world programming, and are now incorporated into many mainstream languages such as Haskell, OCaml 5 and Scala. However, this area of research focuses on the functional paradigm. The aim of this paper, instead, is to show that constructs to raise, compose, and handle effects can be smoothly incorporated into object-oriented languages; in fact, the specific features of the paradigm help in allowing a simple effectful extension.

To illustrate the effectful extension, we choose a pure object calculus designed for the purpose, rather than Featherweight Java (FJ)[1] [21], for more than twenty years the paradigmatic calculus for Java-like languages. As FJ, our calculus only includes distinctive ingredients, such as nominal types, inheritance, and objects, and no imperative features. However, following the current trend in OO languages, and inspired by recent work [49, 43, 50], object values are, rather than class instances with fields, *object literals*, extending a sequence

[1] In its version with generic types.

of nominal types with additional methods. In other words, we select, as subset of Java features, interfaces with default methods and anonymous classes, rather than classes with fields and constructors. This choice leads to generality, in the sense that the language design encompasses both Java-like and functional calculi. More precisely, FJ can be easily encoded by representing fields as constant methods, and, at the same time, as in seminal object calculi [1], the simply-typed lambda calculus can be seen as a language subset, since functions are a special case of stateless objects.

The effectful extension builds on generic effects [36] and handlers [38], carefully adapted to be more familiar to OO programmers. Constructs for raising effects are just method calls of a special kind, called *magic*.[2] They are made available through predefined interfaces, and do not specify an implementation, like abstract ones. However, rather than being deferred to subtypes, their implementation is provided "by the system". For instance, throwing an exception is a magic call `Exception.throw()`. This fits naturally in the OO paradigm, where computations happen in the context of a program. Again to be familiar, the construct to handle effects is a generalized try-block with catch clauses. A raised effect (a magic call) is caught by a clause when the receiver's type is (a subtype of) that specified in the clause, as for Java exceptions, generalized to arbitrary magic calls. To this end, besides those interrupting the normal flow of execution, we allow catch clauses replacing an effect with an alternative behaviour in a continuous manner. Altogether, the construct provides a good compromise between simplicity and expressivity: notably, it does not explicitly handle continuations, as in handlers of algebraic effects [38, 39, 24, 19], yet expressing a variety of handling mechanisms.

We endow our calculus with a type-and-effect system where effect types[3] again generalize sets of exceptions, as in `throws` clauses, to sets of *call-effects*, of shape $T.\mathsf{m}[\overline{T}]$, approximating the computational effects of a call of m in type T, with types $\overline{T}$ instantiating the type variables of the method. This approach, inspired by [15], provides more expressivity than the effect systems for algebraic effects [5, 6, 41], which only track the *names* of operations raising effects, without any type information. Moreover, T can be a type variable X, expressing a call-effect not yet specified, which will become, when X is instantiated to a specific type, the effect type declared there for the method. This allows a rather sophisticated approximation of effects, encompassing a simple form of *effect polymorphism* [26, 20, 29, 7, 9, 8].

Last but not least, the soundness of the type-and-effect system is expressed and proved by a newly introduced approach [12], building on a recent line of research on *monadic operational semantics* for effectful languages [17, 18]. That is, the semantics is formalized by a one-step reduction from language expressions into monadic ones, parameterized by a monad modeling the capabilities provided by the system. In this way, we can uniformly deal with a wide range of effects, by just providing an interpretation of magic methods in the monad. Then, applying a technique from [12], to ensure type-and-effect soundness it is enough to prove progress and subject reduction on the monadic one-step reduction. Accordingly with its shape, subject reduction roughly means that reducing an expression to a monadic one preserves its type-and-effect; hence, to express and prove this property, we need *monadic typing judgments*, which can be derived by *lifting* the non-monadic ones.

We describe the language in Section 2, and its semantics in Section 3. The type-and-effect system is illustrated in Section 4, and its soundness is shown in Section 5. Section 6 discusses related work and Section 7 summarizes our contribution and outlines future work. Auxiliary definitions and omitted proofs can be found in the extended version [11].

[2] This terminology is taken from [50], and also informally used in Java reflection and Python.
[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.

2 Language

To have a smoother presentation, we first provide the syntax, discuss the key features, and show some examples, for the effect-free subset of the language; then we illustrate the constructs to raise, compose, and handle computational effects.

Effect-free language. As anticipated, in our calculus objects are not created by invoking constructors, but directly obtained by extending a sequence of nominal types with additional methods. In this way, the notions of class (more in general, nominal type) and object almost coincide. The only difference is that nominal types are top-level entities in a program, possibly generic (declaring type variables), and abstract (including non-implemented methods). On the other hand, objects can be seen as anonymous types declared on-the-fly in the code.

Syntax and types are given in Figure 1. We assume *type names* N, *method names* m, *type variables* X, Y, and *variables* x, y. The metavariable $\overline{td}$ stands for a sequence $td_1 \ldots td_n$, and analogously for other overlined metavariables. An empty sequence will be denoted by ϵ.

P	$::=$	$\overline{td}$	program
td	$::=$	$\mathsf{N}[\overline{Y} \triangleleft \overline{T}] \triangleleft \overline{N}\{\overline{md}\}$	(nominal) type declaration
md	$::=$	$\mathsf{m} : \mathsf{abs}\, MT \mid \mathsf{m} : \mathsf{def}\, MT\, \langle x\, \overline{x}, e\rangle$	method declaration
e	$::=$	$x \mid \overline{N}\{\overline{md}\} \mid e.\mathsf{m}[\overline{T}](\overline{e})$	expression: variable, object, method call
T, U	$::=$	$X \mid \overline{N}\{s\}$	type
N	$::=$	$\mathsf{N}[\overline{T}]$	nominal type
s	$::=$	$\overline{\mathsf{m}} : \mathsf{k}\ \overline{MT}$	signature (structural type)
k	$::=$	$\mathsf{abs} \mid \mathsf{def}$	(method) kind
MT	$::=$	$[\overline{X} \triangleleft \overline{U}]\overline{T} \to T$	method type

Figure 1 Effect-free syntax and types.

A program P is a sequence of declarations of *(nominal) types*[4]. This sequence is assumed to be a map, that is, type names are distinct. Hence, we can safely use the notations $P(\mathsf{N})$ and $\mathsf{dom}(P)$, as we will do for other (sequences representing) maps.

A type declaration $\mathsf{N}[\overline{Y} \triangleleft \overline{T}] \triangleleft \overline{N}\{\overline{md}\}$ introduces a generic nominal type $\mathsf{N}[\overline{Y} \triangleleft \overline{T}]$, inheriting from all those in $\overline{N}$. We assume that the notation $\overline{Y} \triangleleft \overline{T}$ represents a map from type variables to types (their bounds), that is, type variables are distinct and the two sequences have the same length; moreover, $\overline{N}$ represents a set, that is, order and repetitions are immaterial, and $\overline{md}$ represents a map, that is, method names are distinct. Analogous assumptions hold in an object $\overline{N}\{\overline{md}\}$ and a method type $[\overline{X} \triangleleft \overline{U}]\overline{T} \to T$.

Method declarations can be either abstract (abs), that is, only specifying a method type, or defined (def), providing the (variables to be used as) parameters and the method body. The programmer can choose an arbitrary variable x for the first parameter in defined methods to denote the current object, rather than a fixed name as, e.g., this in Java.

In the calculus, we adopt a uniform syntax for abstract and defined methods, to simplify the technical treatment; a more realistic language would likely use a different concrete syntax, for instance declaring parameters together with their types.

▶ Remark 1. Lambda-expressions can be encoded in the calculus, similarly to what is done in Java; that is, the object $\{\mathsf{apply} : \mathsf{def}\ T \to U\ \langle \_\ x, e\rangle\}$ can be abbreviated $\lambda x{:}T.e$. Note that, in this way, encoded functions are possibly recursive and higher-order.

[4] We adopt this terminology to avoid more connoted terms such as class, interface, trait.

Types are either type variables X or *object types*, of shape $\overline{N}\{s\}$. Nominal types are (instantiations of) those declared in the program. Object types are a form of intersection types, present also in Scala 3, denoting, without introducing a name, a type which is a subtype of all types in $\overline{N}$, additionally providing methods in the signature s. Nominal types can be seen as special object types. We assume that a signature s represents a map from method names to method kinds and types, that is, method names are distinct and the three sequences have the same length. Hence, we can safely use the notations $s(\mathsf{m})$ and $\mathsf{dom}(s)$.

Note that type annotations written by the programmer are allowed to be arbitrary types, rather than only nominal. In this respect, our calculus offers more generality than Java.

In the following, we will write $\overline{N}$ for $\overline{N}\{\}$; note that such a syntactic form can be seen both as an object and the corresponding object type. Moreover, we will write Object for $\{\}$, which, as a type, is the top of the subtyping relation.

Examples. In the code examples, we will use a number of other obvious conventions and abbreviations, such as omitting square brackets around an empty sequence in type/method declarations and method calls, and writing the pair of sequences $\overline{X} \triangleleft \overline{T}$ as a sequence of pairs, omitting bounds which are Object. Finally, in both code and formalism, we will often use the wildcard _ to indicate that some variable or meta-variable does not matter.

▶ **Example 2.** As an example of a program, we show the classic encoding of booleans and natural numbers in the object-oriented paradigm, through an inheritance hierarchy, using the visitor pattern for defining functions over them.

```
Bool {
  if: abs [X Y◁ThenElse[X]] Y -> X
  not: def -> Bool <b, b.if[Bool ThenElse[Bool]](
    ThenElse[Bool]{
      then: def -> Bool <_, False>
      else: def -> Bool <_, True>
    }
  )
}

True ◁ Bool { if: def [X Y◁ThenElse[X]] Y -> X <_ te, te.then()> }
False ◁ Bool { if: def [X Y◁ThenElse[X]] Y -> X <_ te, te.else()> }

ThenElse[X] {
  then: abs -> X
  else: abs -> X
}
```

Let us focus first on the method `if`, which takes a parameter of (a subtype of) `ThenElse[X]`, expected to provide two alternative results of type `X`. The method, abstract in `Bool`, is implemented in `True` and `False` by selecting the `then` and `else` alternative provided by the argument, respectively. As said above, we use the wildcard as variable for the current object, since it does not occur in the body.

The type `ThenElse[X]` declares methods `then` and `else` to be implemented in its subtypes. To illustrate the language features, we define `not` in `Bool` by invoking `if` on the current object with, as argument, an object which implements `ThenElse[Bool]`. Clearly `not` could declared abstract in `Bool` and defined in the obvious way in the two subtypes.

Turning now to naturals, a similar encoding through an inheritance hierarchy could be:

```
Nat {
  succ: def -> Nat <n, Succ{pred: def -> Nat <_, n>}
  match: abs [X] NatMatch[X] -> X
}
```

```
Zero ◁ Nat { match: def [X] NatMatch[X] -> X <_ nm, nm.zero()> }
Succ ◁ Nat {
  pred: abs -> Nat
  match: def [X] NatMatch[X] -> X <n nm, nm.succ(n.pred())>
}

NatMatch[X] { zero: abs -> X   succ: abs Nat -> X }
```

Since in the calculus data are uniformly implemented by stateless objects, numbers greater than 0 are encoded as objects which extend `Succ` by implementing the predecessor method; that is, if $\texttt{Nat}_n$ is the object encoding number n, then $n+1$ is encoded by $\texttt{Nat}_n\texttt{.succ()}$, which evaluates to `Succ{pred: def -> Nat <_, Nat_n>}`.

The type `NatMatch[X]` declares methods `zero` and `succ` to be implemented in subtypes, offering a programming schema for definitions given by arithmetic induction. For instance:

```
Even ◁ NatMatch[Bool] {
  zero: def -> Bool <_,True>
  succ: def Nat -> Bool <even n, n.match(even).not()>
}
```

A different encoding of natural numbers could be provided with no inheritance hierarchy:

```
Nat {
  match: def [X] NatMatch[X] -> X <_ nm, nm.zero()>
  succ: def -> Nat
    <n, Nat{match: def [X] NatMatch[X] -> X <_ nm, nm.succ(n)>}}
}
```

In this version, 0 is encoded by `Nat`, and, if $\texttt{Nat}_n$ is the object encoding number n, then $n+1$ is encoded by $\texttt{Nat.succ(Nat}_n\texttt{)}$, which evaluates to

```
Nat{ match: def [X] NatMatch[X] -> X <_ nm, nm.succ(Nat_n)>} }
```

Effectful language. In Figure 2 we extend the language by adding constructs to raise, compose, and handle effects. As customary, we adopt a fine-grain syntax [27], where *values*

P	$::=$	$\overline{td}$	program
td	$::=$	$\mathsf{N}[\overline{Y \triangleleft T}] \triangleleft \overline{N}\{\overline{md}\}$	type declaration
md	$::=$	$\mathsf{m} : \mathsf{abs}\, MT \mid \mathsf{m} : \mathsf{def}\, MT\ \langle x\, \overline{x}, e\rangle \mid \mathsf{m} : \mathsf{mgc}\, MT$	method declaration
v	$::=$	$x \mid \overline{N}\{\overline{md}\}$	value
e	$::=$	$v.\mathsf{m}[\overline{T}](\overline{v}) \mid \texttt{return}\ v \mid \texttt{do}\ x = e_1\texttt{;}\ e_2 \mid \texttt{try}\ e\ \texttt{with}\ h$	expression
h	$::=$	$\overline{c}, \langle x, e\rangle$	handler
c	$::=$	$N.\mathsf{m} : [\overline{X}]\langle x\, \overline{x}, e\rangle_{\mu}$	(catch) clause
μ	$::=$	$\mathsf{c} \mid \mathsf{s}$	mode
k	$::=$	$\mathsf{abs} \mid \mathsf{def} \mid \mathsf{mgc}$	(method) kind

Figure 2 Fine-grain syntax for the effectful language.

are effect-free, whereas expressions may raise effects, and are also called *computations*. Methods can be, besides abstract and defined, *magic* (abbreviated mgc). Magic methods are made available to the programmer in the type declarations composing the program, and do not specify an implementation, like abstract methods. However, rather than being deferred to subtypes, their implementation is provided "by the system". We add the standard operators `return` for embedding values into computations, and `do` for composing computations sequentially passing the result of the former to the latter. Finally, we provide a try-block enclosing a computation with a *handler*, consisting of a sequence of *(catch) clauses*, and a *final expression*, parametric on a variable. A clause specifies a type and method name, expected to identify a magic method declaration; the clause may catch calls of such method, by executing the *clause expression* instead, parametric on type variables and variables, analogously to a method body. After that, the final expression is either executed or not depending on the *mode*, either c or s, for "continue" and "stop", respectively, of the clause. 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.

3 Monadic Operational Semantics

In this section, following the approach in [12], we define a *monadic operational semantics* for the language, parametric on an underlying monad.

Monads. First of all we recall basic notions about monads, referring to standard textbooks [42] for a detailed presentation.

A *monad* $\mathbb{M} = \langle M, \eta, \mu\rangle$ (on $\mathcal{S}et$) consists of a functor $M : \mathcal{S}et \to \mathcal{S}et$ 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:

$$\begin{tikzcd}
MX \arrow[r, "\eta_{MX}"] \arrow[dr, "\mathsf{id}_{MX}"'] & M^2X \arrow[d, "\mu_X"] & MX \arrow[l, "M\eta_X"'] \arrow[dl, "\mathsf{id}_{MX}"] \\
& MX &
\end{tikzcd}
\qquad
\begin{tikzcd}
M^3X \arrow[r, "M\mu_X"] \arrow[d, "\mu_{MX}"'] & M^2X \arrow[d, "\mu_X"] \\
M^2X \arrow[r, "\mu_X"] & MX
\end{tikzcd}$$

The functor M specifies, for every set X, a set MX of monadic elements built over X, in a way 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.

Functions of type $X \to MY$ are called *Kleisli functions* and play a special role: they can be seen as "effectful functions" from X to Y, raising effects described by the monad $\mathbb{M}$. Given a Kleisli function, by *Kleisli extension* we can define:

$$f^\dagger : MX \to MY \qquad f^\dagger = \mu_Y \circ Mf$$

that is, first we lift f through M to apply it to monadic elements and then we flatten the result using μ_Y. Moreover, given $\alpha \in MX$, $f : X \to MY$ and $g : X \to Y$, we set

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

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.

$$\textsc{(invk)}\ \frac{}{v.\mathsf{m}[\overline{T}](\overline{v}) \to_p e[\overline{T}/\overline{X}][v/x][\overline{v}/\overline{x}]}\quad \mathsf{lookup}(v,\mathsf{m}) = \langle \overline{X}, x, \overline{x}, e\rangle$$

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

$$\textsc{(try-ret)}\ \frac{}{\texttt{try return } v \texttt{ with } h \to_p \texttt{do } x = \texttt{return } v\texttt{; } e'}$$

$$\textsc{(try-do)}\ \frac{}{\texttt{try do } y = e_1\texttt{; } e_2 \texttt{ with } h \to_p \texttt{try } e_1 \texttt{ with } \overline{c}, \langle y, \texttt{try } e_2 \texttt{ with } h\rangle}$$

$$\textsc{(catch-continue)}\ \frac{}{\begin{array}{l}\texttt{try } v.\mathsf{m}[\overline{T}](\overline{v}) \texttt{ with } h \to_p \\ \qquad \texttt{do } x = e[\overline{T}/\overline{X}][v/x][\overline{v}/\overline{x}]\texttt{; } e'\end{array}}\quad \begin{array}{l}\mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, \_\rangle \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) = [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{c}}\end{array}$$

$$\textsc{(catch-stop)}\ \frac{}{\texttt{try } v.\mathsf{m}[\overline{T}](\overline{v}) \texttt{ with } h \to_p e[\overline{T}/\overline{X}][v/x][\overline{v}/\overline{x}]}\quad \begin{array}{l}\mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, \_\rangle \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) = [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{s}}\end{array}$$

$$\textsc{(fwd)}\ \frac{}{\texttt{try } v.\mathsf{m}[\overline{T}](\overline{v}) \texttt{ with } h \to_p \texttt{do } x = v.\mathsf{m}[\overline{T}](\overline{v})\texttt{; } e'}\quad \begin{array}{l}\mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, \_\rangle \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) \text{ undefined}\end{array}$$

$$\textsc{(try-ctx)}\ \frac{e_1 \to_p e_2}{\texttt{try } e_1 \texttt{ with } h \to_p \texttt{try } e_2 \texttt{ with } h}$$

Figure 3 Pure reduction.

Monadic reduction. Let Val and Exp be the sets of closed values and expressions, respectively. In the following, we define a monadic (one-step) reduction for the language, being a relation $\to$ on $\mathsf{Exp} \times M\mathsf{Exp}$, parametric on a monad $\mathbb{M} = \langle M, \eta, \mu\rangle$. More in detail, its definition depends on the following ingredients:

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

Moreover we assume:

- For every magic method m declared in N, a partial function $\mathsf{run}_{\mathsf{N},\mathsf{m}} : \mathsf{Val} \times \mathsf{Val}^\star \rightharpoonup M\mathsf{Val}$, returning a monadic value expressing the effects raised by a call. The function could be undefined, for instance, when arguments do not have the expected types.

The monadic reduction is modularly defined on top of a "pure" reduction $\to_p$ on $\mathsf{Exp} \times \mathsf{Exp}$, which transforms calls of non-magic methods into the corresponding bodies, as usual, and try-blocks into either magic calls or `do` expressions, by distributing and possibly applying catch clauses; magic calls and `do` expressions are, then, normal forms for the pure reduction.

Rules defining the pure reduction are given in Figure 3. The abbreviation h for $\overline{c}, \langle x, e'\rangle$ is intended to be used in all the rules below. The lookup function models method look-up, searching for the method declaration corresponding to a call, with two different successful outcomes: either a defined method, and the result are its type variables, variables, and body, or a magic method, necessarily in a type declaration, and the result is an mgc tag, and the type name. The cmatch function is defined, on a pair consisting of a magic call and a sequence of clauses, when there is a (first) clause catching the call; in this case, the corresponding clause expression is returned. The formal definitions are given in Figure 4.

Rule (INVK) is the standard rule for method invocation. That is, method look-up finds a defined method, and the call reduces to the method's body where type variables and variables are replaced by actual types and arguments. The other rules model the behaviour of a computation enclosed in a try-block.

$$\mathsf{lookup}(\overline{N}\{\overline{md}\},\mathsf{m}) = \begin{cases} [\overline{X}]\langle x\,\overline{x}, e\rangle & \text{if } \overline{md}(\mathsf{m}) = \mathsf{m} : \mathsf{def}\,[\overline{X} \triangleleft \_]\_ \to \_\,\langle x\,\overline{x}, e\rangle \\ \mathsf{lookup}(\overline{N},\mathsf{m}) & \text{otherwise} \end{cases}$$

$$\mathsf{lookup}(N_1 \ldots N_n,\mathsf{m}) = \begin{cases} \mathsf{lookup}(N_i,\mathsf{m}) & \text{if } \mathsf{lookup}(N_i,\mathsf{m}) \text{ defined for a unique } i \in 1..n \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{lookup}(\mathsf{N}[\overline{T}],\mathsf{m}) = \begin{cases} [\overline{X}]\langle x\,\overline{x}, e[\overline{T}/\overline{Y}]\rangle & \text{if } \overline{md}(\mathsf{m}) = \mathsf{m} : \mathsf{def}\,[\overline{X} \triangleleft \_]\_ \to \_\,\langle x\,\overline{x}, e\rangle \\ \langle \mathsf{mgc}, \mathsf{N}\rangle & \text{if } \overline{md}(\mathsf{m}) = \mathsf{m} : \mathsf{mgc}\,[\_ \triangleleft \_]\_ \to \_ \\ \mathsf{lookup}(\overline{N}[\overline{T}/\overline{Y}],\mathsf{m}) & \text{otherwise} \end{cases}$$
$$\text{where } P(\mathsf{N}) = \mathsf{N}[\overline{Y} \triangleleft \_] \triangleleft \overline{N}\{\overline{md}\}$$

$$\mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c) = \begin{cases} [\overline{X}]\langle x\,\overline{x}, e\rangle_\mu & \text{if } c = N.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_\mu \text{ and } v \text{ } \mathsf{instof} \text{ } N \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c\,\overline{c}) = \begin{cases} \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \overline{c}) & \text{if } \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c) \text{ undefined} \\ \mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), c) & \text{otherwise} \end{cases}$$

$$\mathsf{cmatch}(v.\mathsf{m}[\overline{T}](\overline{v}), \epsilon) = \text{undefined}$$

Figure 4 Method look-up and matching.

When the enclosed computation has no effects, rule (TRY-RET), the try-block is reduced to the sequential composition of this computation with the final expression. In case of a `do` composition of two computations, the `do` is eliminated by reducing it to a try-block enclosing the first computation, with as final expression another try-block enclosing the second one; clauses are propagated to both.

When the computation calls a magic method, the behaviour depends on whether a matching clause is found. If it is found, then the clause expression is executed, after replacing parameters by arguments, see rules (CATCH-CONTINUE) and (CATCH-STOP). In a c-clause, the final expression is then executed. If no matching clause is found, then in rule (FWD) the try-block is reduced to the sequential composition of the magic call with the final expression.

Finally, rule (TRY-CTX) is the standard contextual rule. Note that, as already mentioned, in the pure reduction there are no rules for magic calls and `do` expressions. Indeed, the pure reduction only handles expressions when they are enclosed in a try-block, so that effects which would be raised by magic calls can be possibly caught.

The lookup function is defined in Figure 4. When a method is invoked on an object receiver, it is first searched among the method declarations in the object itself (first clause). Objects cannot declare magic methods, since they are made available to the programmer through predefined interfaces, as illustrated by examples later. If not found, then look-up is propagated to the parent types. In this case, there should be exactly one parent type where method look-up is successful. In a nominal type, analogously, the method is first searched among those defined in the corresponding declaration. If a defined method is found, then the type parameters of the declaration are replaced by the corresponding type arguments in the nominal type. If a magic method is found, necessarily in a type declaration, then the result is an mgc tag, and the type name. If the method is not found, then look-up is propagated to the parents, again replacing type parameters with the corresponding type arguments.

The cmatch function is defined in Figure 4 as well. We write v instof N meaning that the dynamic type of v, which can be extracted from v by just erasing method bodies, defined in the obvious way, is a subtype of N. Note that the extraction of the dynamic type and the subtyping check, being part of the runtime semantics, are purely syntactic. Notably, the extracted type could violate constraints on conflicting methods and overriding (which will be

$$\text{(PURE)}\ \frac{e \rightarrow_p e'}{e \rightarrow \eta(e')} \qquad \text{(MGC)}\ \frac{}{v.\mathsf{m}[\overline{T}](\overline{v}) \rightarrow \mathsf{map}\,(\mathtt{return}\ [\,])\,\mathsf{run}_{\mathsf{N},\mathsf{m}}(v,\overline{v})}\ \mathsf{lookup}(v,\mathsf{m}) = \langle \mathsf{mgc}, \mathsf{N}\rangle$$

$$\text{(RET)}\ \frac{}{\mathtt{do}\ x = \mathtt{return}\ v;\ e \rightarrow \eta(e[v/x])} \qquad \text{(DO)}\ \frac{e_1 \rightarrow \mathrm{E}}{\mathtt{do}\ x = e_1;\ e_2 \rightarrow \mathsf{map}\,(\mathtt{do}\ x = [\,];\ e_2)\,\mathrm{E}}$$

Figure 5 Monadic (one-step) reduction.

checked by the type system) or even be ill-formed. If a (first) matching clause is found, then such clause provides an alternative body with its type parameters and parameters, which are instantiated with the corresponding arguments in the magic call. This resembles very much what happens for the call of a defined method, except that look-up is performed in the catch clauses, following the syntactic order.

In Figure 5 we give the rules for the monadic reduction. An expression is reduced to a monadic expression either by propagating a pure step, embedding its result in the monad, as shown in rule (PURE), or by interpreting in the monad magic calls and `do` expressions, as shown in the following rules.

In rule (MGC), method look-up finds a magic method in a type declaration named N, and the call reduces to the corresponding monadic expression. In other words, the effect is actually raised. To this end, we apply the function of type $M\mathsf{Val} \rightarrow M\mathsf{Exp}$ 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 $\mathsf{Val} \rightarrow \mathsf{Exp}$.

Rules (RET) and (DO) are the monadic version of the standard ones for these constructs. More in detail, given a `do` expression, when the first subterm does not raise effects, just returning a value, the expression can be reduced to the monadic embedding of the second subterm, after replacing the variable with the returned value, as shown in rule (RET). Rule (DO), instead, propagates the reduction of the first subterm, taking into account possibly raised effects. To this end, we apply the function of type $M\mathsf{Exp} \rightarrow M\mathsf{Exp}$ obtained by lifting, through map, the context $\mathtt{do}\ x = [\,];\ e_2$, to the monadic expression obtained from e_1. Analogously to above, we identify the context $\mathtt{do}\ x = [\,];\ e_2$, which is an expression with a hole, with the function $e \mapsto \mathtt{do}\ x = [e];\ e_2$ of type $\mathsf{Exp} \rightarrow \mathsf{Exp}$.

Monadic small-step reduction and semantics. Following the approach in [12], we can define, on top of the monadic one-step reduction $\rightarrow$ on $\mathsf{Exp} \times M\mathsf{Exp}$:

- a small-step reduction on *monadic configurations*
- a *finitary semantics* of expressions
- assuming an appropriate structure on the monad, an *infinitary semantics* of expressions.

To apply the construction in [12], the monadic one-step reduction is required to be deterministic, and this is the case indeed.

▶ **Proposition 3** (Determinism). *If* $e \rightarrow \mathrm{E}_1$ *and* $e \rightarrow \mathrm{E}_2$ *then* $\mathrm{E}_1 = \mathrm{E}_2$.

We outline the construction, referring to [12] for technical details and proofs.

First of all, we set $\mathsf{Conf} = \mathsf{Exp} + \mathsf{Res}$, where $\mathsf{Res} = \mathsf{Val} + \mathsf{Wr}$, $\mathsf{Wr} = \{\mathsf{wrong}\}$. That is, a *configuration* c is either an expression or a result r, which, in turn, is either a value, modelling successful termination, or wrong, modelling a stuck computation. Then, we extend the monadic reduction $\rightarrow$ to configurations, getting the relation $\xrightarrow[\mathsf{step}]{} \subseteq \mathsf{Conf} \times M\mathsf{Conf}$ shown in Figure 6. Expressions which represent terminated computations reduce to the monadic embedding of the corresponding value or wrong, respectively; moreover, results (either values or wrong) conventionally reduce to their monadic embedding.

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

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

Figure 6 Monadic (one-step) reduction on configurations.

Now, we can define a relation $\Rightarrow$ on $M\mathsf{Conf}$ as follows:

$\mathrm{C} \Rightarrow \mathrm{C}'$ iff $\mathsf{step}^\dagger(\mathrm{C}) = \mathrm{C}'$.

In this way, computations on monadic configurations are described, as usual in small-step style, by sequences of $\Rightarrow$ steps.

Then, the finitary semantics is the function $[\![-]\!]_\star : \mathsf{Exp} \to M\mathsf{Res} + \{\infty\}$ defined as follows[5]:

$$[\![e]\!]_\star = \begin{cases} \mathrm{R} & \text{if } \eta_{\mathsf{Conf}}(e) \Rightarrow^\star \mathrm{R} \\ \infty & \text{otherwise} \end{cases}$$

where $\Rightarrow^\star$ is the reflexive and transitive closure of $\Rightarrow$. This semantics 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.

To overcome this limitation, we introduce an *infinitary semantics*. As formally detailed in [12], we assume, for each set X, a partial order on MX with a least element $\bot_X$ and suprema $\bigsqcup$ of ω-chains. Let $\mathsf{res}_0 : \mathsf{Conf} \to M\mathsf{Res}$ be the function given by

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

For every $e \in \mathsf{Exp}$ and $n \in \mathbb{N}$, we define $[\![e]\!]_n = \mathsf{res}_0{}^\dagger(\mathrm{C})$ iff $\eta_{\mathsf{Conf}}(e) \Rightarrow^n \mathrm{C}$. Note that $[\![e]\!]_n$ is well-defined for all $n \in \mathbb{N}$ since $\Rightarrow$ is (the graph of) a total function and so is its n-th iteration. The sequence $([\![e]\!]_n)_{n\in\mathbb{N}}$ turns out to be an ω-chain, 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, thus describing also the observable behaviour of possibly diverging computations.

Examples. We show now examples of type declarations providing the user interface to raise effects in an underlying monad, together with some reductions and semantics. Booleans, natural numbers, conditional, and checking that a number is even, are encoded as shown in Example 2, except that, for brevity, we assume that `not` in `True` directly returns `False`. We recall the (inductive) definition of $\mathtt{Nat}_n$, the object encoding the natural number n:

$\mathtt{Nat}_0 = \mathtt{Zero} \qquad \mathtt{Nat}_{n+1} = \mathtt{Succ\{pred:\ def\ ->\ Nat\ <\_,\ Nat}_n\mathtt{>\}}$

Finally, a handler of shape $\overline{c}, \langle x, \mathtt{return}\ x\rangle$ is abbreviated by $\overline{c}$.

[5] Here and in rule (EXP) we omit the injection into monadic configurations.

▶ **Example 4** (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 $E_{\mathsf{Exc}}X = X + \mathsf{Exc}$, and

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

where $+$ denotes disjoint union (coproduct) and, for simplicity, we omit the injections.

We define a nominal type `Exception` of the (objects representing) exceptions. More formally, we assume that, for each v such that v instof `Exception`, that is, the dynamic type of v is a subtype of `Exception`, there is an associated exception in Exc, denoted $\mathsf{exc}(v)$. As a minimal example, we define, besides `Exception`, a subtype `MyException`, and assume[6] $\mathsf{exc}(\texttt{Exception}) = \mathsf{E}$, and $\mathsf{exc}(\texttt{MyException}) = \mathsf{MyE}$, with E, MyE elements of Exc.

```
Exception { throw : mgc[X] -> X }

MyException◁Exception { throw : mgc[X] -> X }
```

Both types declare a magic method `throw` that can be called to raise the exception corresponding to the receiver object. Note that the method `throw` declares a type variable as result type, since it could be called in an arbitrary context. We have:

- $\mathsf{run}_{\texttt{Exception},\texttt{throw}} : \mathsf{Val} \times \mathsf{Val}^\star \rightharpoonup M\mathsf{Val}$ only defined on $\langle v, \epsilon\rangle$ with v instof `Exception`
  $\mathsf{run}_{\texttt{Exception},\texttt{throw}}(v, \epsilon) = \mathsf{exc}(v)$
- $\mathsf{run}_{\texttt{MyException},\texttt{throw}} : \mathsf{Val} \times \mathsf{Val}^\star \rightharpoonup M\mathsf{Val}$ only defined on $\langle v, \epsilon\rangle$ with v instof `MyException`
  $\mathsf{run}_{\texttt{MyException},\texttt{throw}}(v, \epsilon) = \mathsf{exc}(v)$

The `throw` magic method in `MyException` has the same behaviour of that in the parent type; however, redefining the method will be significant in the type-and-effect system (Section 4), to precisely track the possibly raised exceptions.

The type `My` below defines a method that, depending on whether its argument (a natural number) is either even or odd, returns a natural or calls the method `throw` of `MyException`.

```
My {
  m : def Nat -> Nat<x y, y.match(Even).if[Nat ThenElse[Nat]](MyTE)}>
}

MyTE = ThenElse[Nat]{
    then : def [Nat] ->Nat <_, return One>
    else : def [Nat] ->Nat <_, Exception.throw[Nat]()>
}

One = Nat₁ = Succ{pred : def -> Nat <_,Zero>}
```

In the following reductions, we use the abbreviations `MyTE` and `One` above and write `if` for `if[Nat ThenElse[Nat]]`.

Consider the expression:

$$e_1 = \texttt{try}\,\texttt{My.m(One)}\,\texttt{with}\,c_1 \text{ where } c_1 = \texttt{Exception.throw} : [X]\langle x, \texttt{return One}\rangle_{\mathsf{s}}$$

[6] In this simple example exc only depends on the exception type; generally, it could depend on a specific instance, for instance if exceptions provide error messages or other information.

We have the following small-step reduction sequences on monadic configurations[7]:

$$
\begin{array}{ll}
e_1 & \Rightarrow\ \texttt{try One.match(Even).if(MyTE) with } c_1 \\
& \Rightarrow\ \texttt{try Even.succ(One.pred()).if(MyTE) with } c_1 \\
& \Rightarrow\ \texttt{try One.pred().match(Even).not().if(MyTE) with } c_1 \\
& \Rightarrow\ \texttt{try Zero.match(Even).not().if(MyTE) with } c_1 \\
& \Rightarrow\ \texttt{try Even.zero().not().if(MyTE) with } c_1 \\
& \Rightarrow\ \texttt{try True.not().if(MyTE) with } c_1 \\
& \Rightarrow\ \texttt{try False.if(MyTE) with } c_1 \\
& \Rightarrow\ \texttt{try Exception.throw[Nat]() with } c_1 \\
& \Rightarrow\ \texttt{return One} \Rightarrow \texttt{One}
\end{array}
$$

where all steps are derived by rules (PURE) in Figure 5 and (EXP) in Figure 6, except the last one, which is derived by rule (RET) in Figure 6.

Considering now:

$$e_2 = \texttt{try My.m(One) with } c_2 \text{ where } c_2 = \texttt{MyException.throw} : [X]\langle x\,, \texttt{return One}\rangle$$

we have

$$
\begin{array}{ll}
e_2 & \Rightarrow^{\star}\ \texttt{try False.if(MyTE) with } c_2 \text{ first 8 steps as for } e_1 \\
& \Rightarrow\ \texttt{do } x = \texttt{Exception.throw(); return Zero} \\
& \Rightarrow\ \mathsf{E} \text{ since} \\
& \quad\ \texttt{do } x = \texttt{Exception.throw(); return Zero} \to \mathsf{E} \text{ by rule (DO)}
\end{array}
$$

As the reader may have noted, the s-clause models the expected behaviour of exceptions, which, even when caught, interrupt the normal flow of execution. In the following example we show a different interface, still using the exception monad, paired with c-clauses, which replace an effect with an alternative behaviour in a continuous manner.

▶ **Example 5** (Failure).[8] Consider the exception monad with Fail ∈ Exc, and the following type declarations, with $\mathsf{run}_{\texttt{Failure,fail}}$ returning Fail.

```
Failure[X]{ fail : mgc -> X }

String{ ...  toNat: def -> Nat < ... > }

Test {
 sumAsNat: def String String -> Nat
   <_ s1 s2,do n1=s1.toNat();do n2=s2.toNat();do n=n1.sum(n2);return n>
}
```

Method `toNat`, whose implementation is omitted, is expected to return the natural number represented by a string, if any; otherwise, Fail is raised, through the magic call `Failure[Nat].fail()`. We assume a method `sum` in `Nat` returning the sum of two numbers.

The expression `Test.sumAsNat(s1,s2)` clearly raises Fail if one of the two strings does not represent a natural number. However, we can catch such effect returning a default value:

```
try Test.sumAsNat(s1,s2) with Failure[Nat].fail : <_ , return Zero>c
```

[7] We omit the injections from monadic expressions and values.
[8] This example is a minimal version of one in [9].

In this way, we always get a result; in particular, if `s1` does not represent a natural number, and `s2` represents the natural number `n`, we get `n`, as expected. Instead, with a `s`-clause, we would get `Zero`, without performing the sum. An example of reduction sequence is below.[9]

$$e = \texttt{try}\,\texttt{Test.sumAsNat("1","a")}\,\texttt{with}\ c \text{ where } c = \texttt{Failure[Nat].fail} : \langle \_\,, \texttt{return Zero}\rangle_{\texttt{c}}$$

$$\begin{array}{rll}
e & \Rightarrow & \texttt{try do n1="1".toNat();do n2="a".toNat();do n=n1.sum(n2);ret n with}\ c \\
 & \Rightarrow & \texttt{try "1".toNat() with}\ c, \langle \texttt{n1}, \texttt{try do n2="a".toNat();do n=n1.sum(n2);ret n with}\ c\rangle \\
 & \Rightarrow^\star & \texttt{try ret One with}\ c, \langle \texttt{n1}, \texttt{try do n2="a".toNat();do n=n1.sum(n2);ret n with}\ c\rangle \\
 & \Rightarrow & \texttt{do n1 = return One; try do n2 = "a".toNat(); return n1.sum(n2) with}\ c \\
 & \Rightarrow & \texttt{try do n2="a".toNat();do n=One.sum(n2);ret n with}\ c \\
 & \Rightarrow & \texttt{try "a".toNat() with}\ c, \langle \texttt{n2}, \texttt{try do n=One.sum(n2);ret n with}\ c\rangle \\
 & \Rightarrow^\star & \texttt{try Failure[Nat].fail with}\ c, \langle \texttt{n2}, \texttt{try do n=One.sum(n2);ret n with}\ c\rangle (*) \\
 & \Rightarrow & \texttt{do n2 = return Zero; try do n=One.sum(n2);ret n with}\ c \\
 & \Rightarrow & \texttt{try do n=One.sum(Zero);ret n with}\ c \\
 & \Rightarrow & \texttt{try One.sum(Zero) with}\ c, \langle \texttt{n}, \texttt{ret n}\rangle \\
 & \Rightarrow^\star & \texttt{try return One with}\ c, \langle \texttt{n}, \texttt{ret n}\rangle \\
 & \Rightarrow & \texttt{do n = return One; return n} \\
 & \Rightarrow & \texttt{return One} \Rightarrow \texttt{One}
\end{array}$$

If the catch `c`-clause were a `s`-clause, then the expression at line $(*)$ would reduce to `return Zero` and therefore the reduction would produce `Zero`.

It is worthwhile to compare the interfaces:

```
Exception { throw : mgc[X] -> X }

Failure[X]{ fail : mgc String -> X }
```

`Exception` offers a generic method `throw`, so that a magic call `Exception.throw()` can occur in any context. On the other hand, `Failure` is a parametric type, so that a magic call `Failure[T].fail()` can only occur where a `T` is expected.

▶ **Example 6** (Non-determinism). The monad $\mathbb{L} = \langle L, \eta^{\mathbb{L}}, \mu^{\mathbb{L}}\rangle$ is given by LX the set of (possibly infinite) lists over X, 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 $\eta^{\mathbb{L}}_X(x) = [x]$, and the bind is corecursively defined as follows: $\epsilon \gg\!= f = \epsilon$ and $(x\colon l) \gg\!= f = f(x)(l \gg\!= f)$, where juxtaposition denotes the concatenation of possibly infinite lists. We define the nominal type `Chooser` declaring the magic method `choose`, with $\mathsf{run}_{\texttt{Chooser},\texttt{choose}}$ returning the list (monadic value) consisting of the values `True` and `False` and we use the abbreviations `MyTE1` and `MyTE`^{y} defined below.

```
Chooser { choose : mgc -> Bool }

My {
  m1 : def -> Nat  <_, do z = Chooser.choose();
                          z.if[Nat ThenElse[Nat]](MyTE1)>
  m2 : def Nat -> Nat <_ y, do z = Chooser.choose();
                               z.if[Nat ThenElse[Nat]](MyTE^y)>
}

MyTE1 = ThenElse[Nat]{
  then : def [Nat] -> Nat <_, return One>
  else : def [Nat] -> Nat <_, return Zero>
}
```

[9] We use `ret` for `return` to save space.

```
MyTE^y = ThenElse[Nat]{
  then : def [Nat] ->Nat <_, return y>
  else : def [Nat] ->Nat <_, My.m2(y.succ())>
}

Two = Nat_2 = Succ{ pred: def ->Nat <_,Succ{pred: def ->Nat <_,Zero>}> }
```

We have the following small-step reduction sequences:

$$\begin{array}{lll}
[\,\texttt{My.m1()}\,] & \Rightarrow & [\,\texttt{do z=Chooser.choose();z.if(MyTE1)}\,] \\
 & \Rightarrow & [\,\texttt{do z=ret True;z.if(MyTE1), do z=ret False;z.if(MyTE1)}\,] \\
 & \Rightarrow & [\,\texttt{True.if(MyTE1), False.if(MyTE1)}\,] \\
 & \Rightarrow & [\,\texttt{ret One, ret Zero}\,] \\
 & \Rightarrow & [\,\texttt{One, Zero}\,] \\
[\,\texttt{My.m2(Zero)}\,] & \Rightarrow & [\,\texttt{do z=Chooser.choose();z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
 & \Rightarrow & [\,\texttt{do z=ret True;z.if(MyTE}^{\texttt{y}}\texttt{)}, \texttt{do z=ret False;z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
 & \Rightarrow & [\,\texttt{True.if(MyTE}^{\texttt{y}}\texttt{)}, \texttt{False.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
 & \Rightarrow & [\,\texttt{ret Zero}, \texttt{My.m2(Zero.succ())}\,] \\
 & \Rightarrow & [\,\texttt{Zero}, \texttt{My.m2(One)}\,] \\
 & \Rightarrow & [\,\texttt{Zero}, \texttt{do z=Chooser.choose();z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
 & \Rightarrow & [\,\texttt{Zero}, \texttt{do z=ret True;z.if(MyTE}^{\texttt{y}}\texttt{)}, \texttt{do z=ret False;z.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
 & \Rightarrow & [\,\texttt{Zero}, \texttt{True.if(MyTE}^{\texttt{y}}\texttt{)}, \texttt{False.if(MyTE}^{\texttt{y}}\texttt{)}\,] \\
 & \Rightarrow & [\,\texttt{Zero}, \texttt{ret One}, \texttt{My.m2(One.succ())}\,] \\
 & \Rightarrow & [\,\texttt{Zero}, \texttt{One}, \texttt{My.m2(Two)}\,] \\
 & \Rightarrow^\star & [\,\texttt{Zero}, \texttt{One}, \texttt{Two}, \texttt{My.m2(Two.succ())}\,] \\
 & \dots &
\end{array}$$

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 $[\![\,[\,\texttt{My.m2(Zero)}\,]\,]\!]_\star = \infty$. With the infinitary semantics, instead, we get the following ω-chain:

$$\texttt{[]}, \dots, \texttt{[Zero]}, \dots, \texttt{[Zero, One]}, \dots, \texttt{[Zero, One, Two]}, \dots, [\texttt{Nat}_0, \dots, \texttt{Nat}_n], \dots,$$

whose supremum is the infinite list of the (objects representing the) natural numbers.

▶ **Example 7.** Denote by DX the set of probability subdistributions α over X with countable support, i.e., $\alpha : X \to [0..1]$ 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

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

We use the same nominal type `Chooser` with the magic method `choose`, now returning the distribution consisting of the values `True` and `False` with probability $\frac{1}{2}$, that we denote by $[\,\frac{1}{2} : \texttt{return True}, \frac{1}{2} : \texttt{return False}\,]$.

The expressions $[\,1 : \texttt{My.m1()}\,]$ and $[\,1 : \texttt{My.m2(Zero)}\,]$ can be reduced analogously to the previous example:

$[\,1 : \texttt{My.m1()}\,] \Rightarrow^\star [\,\frac{1}{2} : \texttt{One}, \frac{1}{2} : \texttt{Zero}\,]$
$[\,1 : \texttt{My.m2(Zero)}\,] \Rightarrow^\star [\,\frac{1}{2} : \texttt{Zero}, \frac{1}{4} : \texttt{One}, \frac{1}{8} : \texttt{Two}, \frac{1}{16} : \texttt{My.m2(Two.succ())}\,] \ \dots$

Again, the second reduction is non-terminating, hence, with the finitary semantics, we get $[\![\,[\,1 : \texttt{My.m2(Zero)}\,]\,]\!]_\star = \infty$. With the infinitary semantics we get the following ω-chain:

$$\texttt{[]}, \dots, [\tfrac{1}{2}\texttt{:Zero}], \dots, [\tfrac{1}{2}\texttt{:Zero}, \tfrac{1}{4}\texttt{:One}], \dots, [\tfrac{1}{2}\texttt{:Nat}_0, \dots, \tfrac{1}{2^{n+1}}\texttt{:Nat}_n], \dots,$$

whose supremum is the infinite list where each (object representing the) number n has probability $\frac{1}{2^{n+1}}$.

4 Type-and-effect System

In order to equip the language with a type-and-effect system, first of all we extend the syntax and the signatures, as shown in Figure 7. In method declarations, method types are replaced by *method type-and-effects*, where an *effect* component is added; we maintain the same meta-variable for simplicity. They are considered equal up-to α-renaming. That is, $[\overline{X} \triangleleft \overline{U}]\overline{T} \rightarrow T!E = ([\overline{Y} \triangleleft \overline{U'}]\overline{T'} \rightarrow T'!E')[\overline{X}/\overline{Y}]$.

MT	$::=$	$[\overline{X} \triangleleft \overline{U}]\overline{T} \rightarrow T!E$	method type-and-effect
md	$::=$	$\mathsf{m} : \mathsf{def}\ MT\ \langle x\, \overline{x}, e\rangle \mid \mathsf{m} : \mathsf{abs}\ MT \mid \mathsf{m} : \mathsf{mgc}\ MT$	method declaration
E	$::=$	$\bullet \mid \top \mid E{\vee}E' \mid T.\mathsf{m}[\overline{T}]$	effect
s	$::=$	$\overline{\mathsf{m}} : \overline{\mathsf{k}}\ \overline{MT}$	signature
k	$::=$	$\mathsf{abs} \mid \mathsf{def} \mid \mathsf{mgc}$	(method) kind

Figure 7 Adding effects.

Effects are the empty effect, the top effect, union of effects, and *call-effects*. For magic methods, this component is assumed to have a canonical form, hence can be omitted in the concrete syntax. Notably, for a method $\mathsf{m} : \mathsf{mgc}\, [\overline{X} \triangleleft \_]\_ \rightarrow \_$ in the declaration of the nominal type $\mathsf{N}[\overline{Y} \triangleleft \_]$, the effect is $\mathsf{N}[\overline{Y}].\mathsf{m}[\overline{X}]$. In signatures, the information associated to method names is analogously extended; moreover, the additional kind mgc is considered.

Φ	$::=$	$\overline{X} \leq \overline{T}$	type environment
Γ	$::=$	$\overline{x} : \overline{T}$	environment

Figure 8 Syntax of (type) environments.

Type environments and environments, defined in Figure 8, are assumed to be maps, from type variables to types (their bounds), and from variables to types, respectively. Hence (type) variables are distinct, and the two sequences have the same length.

Before the formal details, we illustrate the most distinctive feature of our type system.

Variable call-effects and simplification. As shown in Figure 7, except for $\top$, effects are essentially (representations of) sets of call-effects, with $\bullet$ the empty set and $\vee$ the union. Sets of "atomic" effects are a rather natural idea, generalizing what happens, e.g., in Java `throws` clauses. What is interesting here is the nature of such atomic effects, and the role of $\top$. A call-effect $T.\mathsf{m}[\overline{T}]$ is a static approximation of the computational effects of a call to m with receiver of type T and type arguments $\overline{T}$. More precisely, we only allow call-effects which are:

magic $T.\mathsf{m}[\overline{T}]$ with T object type (that is, not of shape X), and m magic in T. As expected, this means that this magic method could be possibily invoked, raising the corresponding computational effect; for instance, as will be shown in Example 8, a call-effect `Exception.throw` denotes that an exception could be possibly thrown.

variable $X.\mathsf{m}[\overline{T}]$. This call-effect can be assigned to code parametric on the type variable X; the meaning is that, for each instantiation of X with an object type T, this becomes, through a non-trivial process called *simplification*, the effect of m in T. In other words, $X.\mathsf{m}[\overline{T}]$ is a parametric effect, which can be made concrete in the types which replace X.

In addition to sets of call-effects, we include the *top* effect, which plays the role of default for (typically abstract) methods which do not pose constraints on the effects in implementations. In a sense, this generalizes the meaning of a `throws Exception` clause in Java.

We illustrate now the above features on an example, notably showing how (simplification of) variable call-effects allows a very precise approximation.

▶ **Example 8.** Consider again the encoding of booleans in Example 2 (Section 2), where we added effect annotations.

```
Bool { if : abs [X Y◁ThenElse[X]] Y -> X ! Y.then ∨ Y.else }

True ◁ Bool {
  if : def [X Y◁ThenElse[X]] Y -> X ! Y.then <_ te, return te.then()>
}

False ◁ Bool {
  if : def [X Y◁ThenElse[X]] Y -> X ! Y.else <_ te, return te.else()>
}

ThenElse[X] {
  then: abs -> X ! ⊤
  else: abs -> X ! ⊤
}
```

The abstract method if in `Bool` has, as parameter type, the type variable `Y`, expected to be instantiated with subtypes of `ThenElse[X]`, hence providing methods `then` and `else`. As effect, the method declares the union of two variable call-effects, `Y.then` ∨ `Y.else`. This means that the computational effects of this method can only be those propagated from calling either `then` or `else` on the argument. Subtypes `True` and `False` specializes the effect of `if` as expected, since they select the `then` and the `else` alternative provided by the argument, respectively.

In the type `ThenElse[X]` the methods `then` and `else` are abstract, and their implementation in subtypes is allowed to raise arbitrary effects, as denoted by ⊤. This makes sense, since we would like to instantiate `ThenElse[X]` on types implementing these methods in arbitrary ways.

For instance, consider the following code, where `b` is an expression of type `Bool`, and `Exception`, `MyTE` are the type and object introduced in Example 4, where the latter has been annotated with effects, and `MyTEType` is the corresponding object type.

```
b.if[Nat, MyTEType](MyTE)

MyTE = ThenElse[Nat]{
  then: def  -> Nat ! • <_, return Zero>
  else: def  -> Nat ! MyException.throw[Nat] <_, MyException.throw()>
}

MyTEType =
ThenElse[Nat]{
  then: def -> Nat ! •,
  else: def -> Nat ! MyException.throw[Nat]
}
```

The argument passed to `if` is an object implementing `then` by returning `Zero`, hence declaring no effects, whereas `else` calls a magic method, and declares the corresponding effect.

The effect computed for the call is, as intuitively expected, `MyException.throw[Nat]`. This happens thanks to effect simplification, which will be formally specified in Figure 10. Indeed, looking for method `if` in the receiver's type `Bool` gives the effect `X.then` ∨ `X.else`, which is instantiated to the argument type, giving `MyTEType.then` ∨ `MyTEType.else`. This effect would be highly inaccurate. However, `MyTEType.then` and `MyTEType.else` are simplified to the effects of the corresponding methods, that is, • and `MyException.throw[Nat]`, respectively.

In conclusion, our type-and-effect system supports a form of effect polymorphism which does not need additional ingredients, such as, e.g., explicit effect variables. Indeed, the effect of a method can be parametric on those of methods called on type variables in the context (variable call-effects): for instance, the effect of `if` is parametric on `Y.then` and `Y.else`. This only relies on existing language features, notably on the OO paradigm. The key point is that a method is identified by, besides its name, the type where it is declared. Hence, to be parametric on the effect of a method, it is enough to be parametric on the type where it is declared, and this is for free since we have type variables. In other words, whereas effect variables stand for arbitrary effects, variable call-effects stand for effects declared by a method, and standard instantiation is complemented here by a non-trivial step of simplification.

Key formal definitions. The typing rules for values and expressions are given in Figure 9. The typing judgment for values has the shape $\Phi;\Gamma \vdash v : T$, since values have no effects; the one for expressions, instead, has the shape $\Phi;\Gamma \vdash e : T!E$.

$$\textsc{(t-var)}\ \frac{}{\Phi;\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \textsc{(t-obj)}\ \frac{\Phi;\Gamma;\overline{N}\{s\} \vdash \overline{md}\,\diamond}{\Phi;\Gamma \vdash \overline{N}\{\overline{md}\} : \overline{N}\{s\}}\ \begin{array}{l}\Phi \vdash \overline{md} \rightsquigarrow s\\ \Phi \vdash \overline{N}\{s\} \rightsquigarrow s'\\ \mathsf{NoMgc}(s)\\ \mathsf{NoAbs}(s')\end{array}$$

$$\textsc{(t-invk)}\ \frac{\Phi;\Gamma \vdash v_0 : T_0 \qquad \Phi;\Gamma \vdash v_i : T'_i\ \forall i \in 1..n}{\Phi;\Gamma \vdash v_0.\mathsf{m}[\overline{T}](v_1,\ldots,v_n) : T[\overline{T}/\overline{X}]!E'}\ \begin{array}{l}\mathsf{mtype}_\Phi(T_0,\mathsf{m}){=}\_\,[\overline{X} \lhd \overline{U}]\,T_1\ldots T_n \to T!E\\ \Phi \vdash \overline{T} \leq \overline{U}[\overline{T}/\overline{X}]\\ \Phi \vdash T'_i \leq T_i[\overline{T}/\overline{X}]\ \forall i \in 1..n\\ \Phi \vdash E[\overline{T}/\overline{X}]{\Downarrow}E'\end{array}$$

$$\textsc{(t-ret)}\ \frac{\Phi;\Gamma \vdash v : T}{\Phi;\Gamma \vdash \texttt{return}\ v : T!\bullet} \qquad \textsc{(t-do)}\ \frac{\Phi;\Gamma \vdash e : T!E \qquad \Phi;\Gamma, x : T \vdash e' : T'!E'}{\Phi;\Gamma \vdash \texttt{do}\ x = e;\ e' : T'!E{\vee}E'}$$

$$\textsc{(t-try)}\ \frac{\Phi;\Gamma \vdash e : T!E \qquad \Phi;\Gamma;T \vdash h : T'!H}{\Phi;\Gamma \vdash \texttt{try}\ e\ \texttt{with}\ h : T'!\mathcal{F}_H(E)}$$

$$\textsc{(t-handler)}\ \frac{\Phi;\Gamma, x : T \vdash e' : T'!E' \qquad \Phi;\Gamma;T'' \vdash c_i \Rightarrow C_i\ \forall i \in 1..n}{\Phi;\Gamma;T \vdash c_1\ldots c_n, \langle x, e'\rangle : T''!C_1\ldots C_n, E'}\ \Phi \vdash T' \leq T''$$

$$\textsc{(t-continue)}\ \frac{\Phi;\Gamma, x : N_x, \overline{x}{:}\overline{T} \vdash e : T''!E}{\Phi;\Gamma;T'{\vdash}N_x.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{c}}{\Rightarrow}T_x.\mathsf{m} : [\overline{X}]\langle E\rangle}\ \begin{array}{l}\mathsf{mtype}_\Phi(N_x,\mathsf{m}){=}\mathsf{mgc}\,[\overline{X} \lhd \overline{U}]\overline{T} \to T!\_\\ \Phi \vdash T'' \leq T\end{array}$$

$$\textsc{(t-stop)}\ \frac{\Phi;\Gamma, x : N_x, \overline{x}{:}\overline{T} \vdash e : T''!E}{\Phi;\Gamma;T'{\vdash}N_x.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mathsf{s}}{\Rightarrow}T_x.\mathsf{m} : [\overline{X}]\langle E\rangle}\ \begin{array}{l}\mathsf{mtype}_\Phi(N_x,\mathsf{m}){=}\mathsf{mgc}\,[\overline{X} \lhd \overline{U}]\overline{T} \to \_!\_\\ \Phi \vdash T'' \leq T'\end{array}$$

Figure 9 Typing rules for values and expressions.

They rely on the following auxiliary notations:

1. $\Phi \vdash E{\Downarrow}E'$ meaning that the effect E can be simplified to E'
2. $\mathcal{F}_h$ the *filter function* associated to the handler h
3. $\Phi \vdash T \rightsquigarrow s$ meaning that we can safely extract a signature s from a type; the notation $\mathsf{mtype}_\Phi(T,\mathsf{m}){=}\mathsf{k}\,MT$ is an abbreviation for $\Phi \vdash T \rightsquigarrow s$ and $s(\mathsf{m}) = \mathsf{k}\,MT$
4. $\Phi \vdash T \leq T'$ the subtyping relation

We illustrate the typing rules and notations (1) and (2), whereas the formal definitions of (3) and (4), which are almost standard, are given in the extended version [11]. The reader should only know that (3) models extracting the structural type information; in this phase, constraints about no conflicting method definitions and safe overriding are checked. Then, typing rules model another phase where code (method bodies) is typechecked against this type information.[10]

[10] Differently from other type systems for Java-like languages [21], here the two phases cannot be sequenced, since we still need signature extraction for the types introduced "on the fly" by objects.

$$(\textsc{empty})\ \frac{}{\Phi \vdash \bullet\Downarrow\bullet} \qquad (\textsc{top})\ \frac{}{\Phi \vdash \top\Downarrow\top} \qquad (\textsc{var})\ \frac{}{\Phi \vdash X.\mathsf{m}[\overline{T}]\Downarrow X.\mathsf{m}[\overline{T}]}\ X \in \mathsf{dom}(\Phi)$$

$$(\textsc{mgc})\ \frac{}{\Phi \vdash T.\mathsf{m}[\overline{T}]\Downarrow T.\mathsf{m}[\overline{T}]}\ \mathsf{mtype}_{\Phi}(T,\mathsf{m}){=}\mathsf{mgc}\,[\overline{X}\lhd\_]\_ \to \_!E$$

$$(\textsc{simplify-non-mgc})\ \frac{\Phi \vdash E[\overline{T}/\overline{X}]\Downarrow E'}{\Phi \vdash T.\mathsf{m}[\overline{T}]\Downarrow E'}\ \begin{array}{l}\mathsf{mtype}_{\Phi}(T,\mathsf{m}){=}\mathsf{k}\,[\overline{X}\lhd\_]\_ \to \_!E\\ \mathsf{k}\neq\mathsf{mgc}\end{array}$$

$$(\textsc{simplify-union})\ \frac{\Phi \vdash E_i\Downarrow E'_i\ \ i \in 1..2}{\Phi \vdash E_1{\vee}E_2\Downarrow E'_1{\vee}E'_2}$$

Figure 10 Simplification of effects.

Rule (T-VAR) is straightforward. An object is well-typed, rule (T-OBJ), if a signature can be safely extracted from its type. This essentially means that there are no conflicts among the parent types, and they are safely overriden by the method declarations in the object (first and second side conditions). Moreover, an object cannot declare magic methods (third side condition), and should provide an implementation for all its (either inherited or declared) methods (last side condition). Finally, in the premise, (bodies of) declared methods should be well-typed with respect to an enclosing type which is the object type.

In rule (T-INVK), the type-and-effect of the invoked method is found in the (signature extracted from) the receiver's type, as expressed by the first side condition. In the second side condition, the type annotations in the call should (recursively) satisfy the bounds for the corresponding type variables, and, in the third side condition, the argument types should be subtypes of the corresponding parameter types where type variables have been replaced by the corresponding type annotations. The type assigned to the call is the return type of the method, with the same replacement. The method effect is instantiated by replacing the type variables with the type annotations. However, the effect assigned to the call is obtained by a further *simplification step* (last side condition).

The formal definition of simplification is given in Figure 10, where $\Phi \vdash E\Downarrow E'$ means that the (not necessarily simplified) effect E is simplified to E', which only contains call-effects which are either magic or variable, as described before.

We assume that effect annotations are written by the programmer (or preliminarily reduced) in a simplified shape. Non-simplified effects can, however, appear during typechecking, when a method declared with an effect E which is generic, that is, depending on the method's type variables $\overline{X}$, is invoked with actual type arguments[11] Indeed, in this case, instantiating the type variables, in some variable call-effect $X.\mathsf{m}[\overline{T}]$ in E, X can become an object type providing a (non-magic) method m, allowing to simplify to the corresponding effect of m in the object type. This allows us to give a more refined approximations of the effects raised at runtime, as illustrated by Example 8 above, where the non-simplified effect `MyTEType.then` $\vee$ `MyTEType.else` is simplified to $\bullet$ $\vee$ `MyException.throw[Nat]`. To ensure decidability of typechecking, an issue we do not deal with in this paper, termination of effect simplification should be enforced by some standard technique, essentially by forbidding (mutual) recursion in effect annotations.

[11] This happens in rule (T-INVK) in Figure 9.

$$\begin{array}{llll}
h & ::= & \overline{c}, \langle x, e\rangle & \text{handler}\\
H & ::= & \overline{C}, E & \text{filter}\\
c & ::= & N.\mathsf{m} : [\overline{X}]\langle x\,\overline{x}, e\rangle_{\mu} & \text{catch clause}\\
C & ::= & N.\mathsf{m} : [\overline{X}]\langle E\rangle & \text{clause filter}
\end{array}$$

$\mathcal{F}_H$ defined by:
$\mathcal{F}_H(E) = \mathcal{F}_{\overline{C}}(E) \vee E'$ if $H = \overline{C}, E'$

$\mathcal{F}_{\overline{C}}$ defined by:
$\mathcal{F}_{\overline{C}}(\bullet) = \bullet \quad \mathcal{F}_{\overline{C}}(\top) = \top \quad \mathcal{F}_{\overline{C}}(E_1 \vee E_2) = \mathcal{F}_{\overline{C}}(E_1) \vee \mathcal{F}_{\overline{C}}(E_2)$

$$E = T.\mathsf{m}[\overline{T}] \qquad \mathcal{F}_{C\,\overline{C}}(E) = \begin{cases} \mathcal{F}_C(E) & \text{if } \mathcal{F}_C(E) \text{ defined}\\ \mathcal{F}_{\overline{C}}(E) & \text{otherwise}\end{cases} \qquad \mathcal{F}_{\epsilon}(E) = E$$

$$C = N.\mathsf{m} : [\overline{X}]\langle E\rangle \qquad \mathcal{F}_C(T.\mathsf{m}[\overline{T}]) = \begin{cases} E[\overline{T}/\overline{X}] & \text{if } \Phi \vdash T \leq N\\ \text{undefined} & \text{otherwise}\end{cases}$$

Figure 11 Filters.

In rule (T-RET), a computation which is the embedding of a value has no effects, and, in (T-DO), a sequential composition of two computations has the union of the two effects.

Typing rules for try-blocks rely on *filter functions* associated to handlers, which describe how they transform effects, by essentially replacing calls of magic methods matching some clause with the effect of the clause expression, as formally defined in Figure 11. Filter functions allow to typecheck try-blocks in a very precise way, since effects of the expression in a clause are only added if the clause could be possibly applied. This generalizes to arbitrary effects what happens for Java catch clauses; however, in Java this is part of the analysis of unreachable code, whereas here it is a feature of the type system. Filters are not a novelty of this type-and-effect system, since they were firstly used in [12] for a functional calculus; however, it is worthwhile to describe them in detail since they are a new feature, and the general idea is applied here to rather different effects, leading to another formal definition.

As shown in Figure 11, filters are the type information which can be extracted from a handler, consisting of a sequence of clause filters and a final effect. The filter function $\mathcal{F}_H$ associated to H transforms an effect by first applying the sequence of clause filters, and then adding the final effect. The transformation applying a sequence of clause filters is defined inductively. The significant case is a call-effect, which is either transformed by a (first) matching clause filter, or remains unaffected. A clause filter matches a call-effect with the same method name and a subtype of the nominal type; the call-effect is replaced by the effect of the clause filter, where type variables have been substituted by the types in the call-effect.

In rule (T-TRY), in order to typecheck a try-block, first we get the type and effect of the enclosed expression. This type is then used to typecheck the handler, as type of the parameter of the final expression, see rule (T-HANDLER). By typechecking the handler we get a type, being a supertype 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 of the enclosed expression, getting the resulting effect of the whole expression.

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, required to be a subtype of that of the handler. This latter 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.

The filter extracted from a clause keeps the first three components (nominal type, method name, and type variables), and adds the effect obtained typechecking the clause expression, as shown in rules (T-CONTINUE) and (T-STOP). A c-clause is meant to provide alternative code

$$\text{(T-METHS)}\ \frac{\Phi;\Gamma;T\vdash md_i\ \diamond\ \ \forall i\in 1..n}{\Phi;\Gamma;T\vdash md_1\ldots md_n\diamond}$$

$$\text{(T-METH)}\ \frac{\Phi,\overline{X}\lhd\overline{U};\Gamma,x:T_x,\overline{x}{:}\overline{T}\vdash e:T'!E'}{\Phi;\Gamma;T_x\vdash \mathsf{m}:\mathsf{def}\,[\overline{X}\lhd\overline{U}]\overline{T}\to T!E\,\langle x\,\overline{x},e\rangle\diamond}\ \ \Phi,\overline{X}\lhd\overline{U}\vdash T'!E'\leq T!E$$

$$\text{(T-PROG)}\ \frac{\vdash td_i\ \diamond\ \ \forall i\in 1..n}{\vdash td_1\ldots td_n\diamond}$$

$$\text{(T-NTYPE)}\ \frac{\overline{Y}\lhd\overline{T};\emptyset;\mathsf{N}[\overline{Y}]\vdash \mathsf{GetDef}(\overline{md})\diamond}{\vdash \mathsf{N}[\overline{Y}\lhd\overline{T}]\lhd\overline{N}\{\overline{md}\}\diamond}\ \ \vdash \mathsf{N}[\overline{Y}\lhd\overline{T}]\lhd\overline{N}\{\overline{md}\}\rightsquigarrow \_$$

Figure 12 Typing rules for method and type declarations.

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.

Referring to Example 4, note that catching an exception with a c-clause would be ill-typed. Indeed, the type of the clause expression should be (a subtype of) the return type of `throw`, which is a type variable X. Since no value has type X, no value could be returned[12], as already noted in [36].

In Figure 12 we show the typing rules for method and type declarations, which are mainly straightforward. The typing judgment for method definitions has shape $\Phi;\Gamma;T\vdash\overline{md}\diamond$. In rule (T-METH), a method definition is well-typed if the body is well-typed with respect to a type environment enriched by the type variables with their bounds, and an environment enriched by the variable denoting the current object with the enclosing type, and parameters with the corresponding parameter types. The type-and-effect of the body should be a sub-type-and-effect of that declared for the method. The typing judgment for type declarations has shape $\vdash\overline{td}\diamond$. In rule (T-NTYPE), a type declaration, assumed to have passed the extraction phase (side condition), is well-typed if its defined method declarations, denoted by $\mathsf{GetDef}(\overline{md})$, are well-typed with respect to the type environment consisting of the type variables with their bounds, the empty environment, and the declared type as enclosing type. Here the environment is empty since these method declarations are top-level, whereas in those inside objects, handled in rule (T-OBJ), there can be variables declared at an outer level.

5 Type-and-effect Soundness

In this section, we express and prove soundness of our type-and-effect system, by applying definitions and results in [12]. We focus on explaining the concepts, referring to [12] for detailed formal definitions and proofs.

Informal introduction. In Section 3 we defined, on top of the monadic one-step reduction:

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

In standard soundness we expect the result, if any, to be in agreement with the expression type. Here, since the expression has also an effect, approximating the computational effects raised by its execution, we expect the monadic result, if any, to be in agreement with the expression type and effect.

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

Let us write $\vdash e : T!E$ for $\emptyset;\emptyset \vdash e : T!E$, and analogously for $\vdash v : T$, and other ground judgments. Moreover, $\vdash r : T$ only holds if $r = v$ and $\vdash v : T$ holds. Note that the result wrong is never well-typed, that is, $\vdash \mathsf{wrong} : T$ does not hold for any type T.

To formally express the above soundness requirement, we need analogous typing judgments[13] $\vdash_E \mathrm{R} : T$, one for each type and effect, on monadic results. Assuming to have such judgments, we can express the soundness results as follows.

▶ **Theorem 9** (Finitary type-and-effect soundness). *$\vdash e : T!E$ and $[\![e]\!]_\star = \mathrm{R}$ imply $\vdash_E \mathrm{R} : T$.*

▶ **Theorem 10** (Infinitary type-and-effect soundness). *$\vdash e : T!E$ implies $\vdash_E [\![e]\!]_\infty : T$.*

Note that finitary soundness is vacuous when the finitary semantics of an expression is ∞. On the other hand, the infinitary semantics is always a monadic result.

Before explaining how the monadic typing judgments can be derived from the non-monadic ones, we show some examples.

▶ **Example 11.** Consider the exception monad as in Example 4. We have $E\mathsf{Res} = \mathsf{Res} + \mathsf{Exc}$, that is, monadic results are either values, or wrong, or exceptions. We expect the typing judgment $\vdash_E \mathrm{R} : T$ to be defined as follows:

$$\text{(T-VAL)}\ \frac{\vdash v : T}{\vdash_E v : T} \qquad \text{(T-EXC)}\ \frac{}{\vdash_E \mathsf{e} : T}\ \mathsf{e} \in [\![E]\!]$$

That is, a monadic result is well-typed with a given type-and-effect if it is either a well-typed value with the type, or an exception belonging to (the set denoted by) the effect. Note that wrong is ill-typed. The set of exceptions $[\![E]\!]$ represented by E is defined by:

$$\begin{array}{lll} [\![\bullet]\!] = \emptyset & [\![\top]\!] = \mathsf{Exc} & [\![E \vee E']\!] = [\![E]\!] \cup [\![E']\!] \\ [\![\mathsf{N}.\mathtt{throw}]\!] = \{\mathsf{e} \mid \mathsf{e} = \mathsf{exc}(v), v\ \mathsf{instof}\ \mathsf{N}\} & & \end{array}$$

That is, call-effects of the `throw` method in N denote the set of the exceptions (represented by objects) of subtypes of N, and the other operators have the obvious set-theoretic meaning.

▶ **Example 12.** Consider the monad of non-determinism as in Example 6. We have $L\mathsf{Res}$ the set of (possibly infinite) lists of results (values or wrong). We show two different ways to define the typing judgment for monadic results, denoted $\vdash^{\mathsf{q}}_E \mathrm{R} : T$ for $\mathsf{q} ::= \forall \mid \exists$.

$$\text{(T-NO-RES)}\ \frac{}{\vdash^{\mathsf{q}}_E \epsilon : T} \qquad \text{(T-DET)}\ \frac{\vdash v : T}{\vdash^{\mathsf{q}}_E [v] : T}\ [\![E]\!] = 0$$

$$\text{(T-}\forall\text{)}\ \frac{\vdash r : T\ \forall r \in \mathrm{R}}{\vdash^{\forall}_E \mathrm{R} : T}\ [\![E]\!] = 1 \qquad \text{(T-}\exists\text{)}\ \frac{\vdash r : T}{\vdash^{\exists}_E \mathrm{R} : T}\ \begin{array}{l}[\![E]\!] = 1 \\ r \in \mathrm{R}\end{array}$$

where $[\![E]\!]$ is 1 if non-determinism is allowed, 0 otherwise, as defined below:

$$\begin{array}{lll} [\![\bullet]\!] = 0 & [\![\top]\!] = 1 & [\![E \vee E']\!] = [\![E]\!] \vee [\![E']\!] \\ [\![\mathtt{Chooser.choose}[\ ]]\!] = 1 & & \end{array}$$

That is, monadic results (representing possible results of a computation) are well-typed with a type effect (denoting) 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 (denoting) 1 if all the results in the list are well-typed

[13] The effect is written under the turnstile to emphasize that this typing judgment is obtained by lifting the non-monadic one, as will be described in the following.

values, or there is either no result, or at least one well-typed value, respectively. The two interpretations express "must" and "may" soundness of a non-deterministic computation: with $\forall$, we require every possible result to be well-typed, whereas, with $\exists$, it is enough to have a well-typed result, and the others could be even wrong.

Interpretation of effect types. Monadic typing judgments can be derived from non-monadic ones by providing an *interpretation of effect types* into the considered monad.

Let us denote by Eff the set of effects, assumed to be simplified (Figure 10), ground, and well-formed, considered modulo the equivalence induced by the subtyping preorder[14]. As customary, we write E for the element of Eff it represents, i.e., its equivalence class. Then, it is easy to see that Eff is the carrier of an ordered monoid $\mathcal{E}$, where the unit is $\bullet$, the multiplication, which turns out to be idempotent and commutative, is $\vee$, and the order is $\leq$.

For a set X, we denote by $\mathcal{P}(X)$ the poset of all subsets (a.k.a. predicates or properties) on X, ordered by subset inclusion. For a function $f : X \to Y$, we have a monotone function $\mathcal{P}_f : \mathcal{P}(Y) \to \mathcal{P}(X)$, given by the inverse image: for $A \subseteq Y$, $\mathcal{P}_f(A) = \{x \in X \mid f(x) \in A\}$.

▶ **Definition 13** (Interpretation of effect types). *Let $\mathbb{M} = \langle M, \mu, \eta \rangle$ be a monad. Then, an* interpretation *of $\mathcal{E}$ in $\mathbb{M}$ consists of a family λ of monotone functions $\lambda^E_X : \mathcal{P}(X) \to \mathcal{P}(MX)$, for every $E \in \mathsf{Eff}$ and set X, such that*

1. $\lambda^E_X(\mathcal{P}_f(A)) = \mathcal{P}_{Mf}(\lambda^E_Y(A))$*, for every $A \subseteq Y$ and function $f : X \to Y$*
2. $E \leq E'$ *implies* $\lambda^E_X(A) \subseteq \lambda^{E'}_X(A)$*, for every $A \subseteq X$*
3. $A \subseteq \mathcal{P}_{\eta_X}(\lambda^\bullet_X(A))$*, for every $A \subseteq X$*
4. $\lambda^E_{MX}(\lambda^{E'}_X(A)) \subseteq \mathcal{P}_{\mu_X}(\lambda^{E \vee E'}_X(A))$*, for every $A \subseteq X$.*

The family $\lambda = (\lambda^E)_{E \in \mathsf{Eff}}$ is a family of predicate liftings [22] for the monad $\mathbb{M}$, indexed by effect types. That is, for each effect type E, λ^E transforms predicates on X into predicates on MX, and can be regarded as the semantics of the effect type E.

Item 1 states that λ^E_X is natural in X. This ensures that the semantics of each effect type is independent from the specific set X. Item 2 states that λ^E_X is monotone with respect to the order on effects, that is, computational effects described by E are also described by E'. Item 3 states that monadic elements in the image of η_X contain computational effects described by $\bullet$, 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 E', and then by lifting through E. By flattening such elements through $\mu_X : M^2X \to MX$ we obtain elements whose computational effects are described by the composition $E \vee E'$.

In the proof of soundness for our calculus, we only need Item 3 and Item 1, instantiated as will be detailed in Lemma 16. Item 2 and Item 4 are only required in the general framework in [12] to derive soundness from progress and subject reduction.

We show now how the previous examples can be obtained through an appropriate interpretation of effect types.

In Example 11, we can take $\lambda^E_X(A) = A + [\![E]\!]$. That is, the interpretation of an effect type E transforms a predicate A on X into a predicate on MX which holds either on elements which satisfy the original predicate, or exceptions in the set denoted by E.

In Example 12, we can take the following two interpretations $\forall$ and $\exists$:

if $[\![E]\!] = 1$, then

[14] That is, we consider effects as sets of call-effects, plus the top effect. For the formal definition of well-formedness and subtyping, which are standard, see [11].

$\forall^{E}_{X}(A) = \{B \in PX \mid B \subseteq A\}$ and
$\exists^{E}_{X}(A) = \{B \in PX \mid B = \emptyset \text{ or } B \cap A \neq \emptyset\}$
if $[\![E]\!] = 1$, then $\forall^{0}_{X}(A) = \exists^{0}_{X}(A) = \{B \in PX \mid B \subseteq A \text{ and } \sharp B \leq 1\}$

That is, in both cases, the interpretation of 0 forbids 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 $\exists$.

Proof of type-and-effect soundness. Thanks to a general result proved in [12], to prove Theorem 9 and Theorem 10 we can use a technique similar to that widely used to prove soundness of a type system with respect to a small-step semantics, that is, it is enough to prove the progress and subject reduction properties, stated below. For the infinitary soundness, the interpretation of effect types has to respect the additional structure of the monad, that is, the least monadic element should be always well-typed, and the monadic typing judgment should be closed with respect to suprema of ω-chains. We refer to [12] for the formal definition of such requirement, which trivially holds in our case.

Let us denote by $\vdash_{\hat{E}} \mathrm{V} : T$ and $\vdash_{\hat{E}} \mathrm{E} : T!E$ the typing judgments on monadic values and expressions obtained by lifting the non-monadic ones through $\hat{E}$.

▶ **Theorem 14** (Monadic Progress). *$\vdash e : T!E$ implies either $e = \texttt{return}\ v$ for some $v \in$ Val, or $e \to \mathrm{E}$ for some $\mathrm{E} \in M$Exp.*

▶ **Theorem 15** (Monadic Subject Reduction). *$\vdash e : T!E$ and $e \to \mathrm{E}$ imply $\vdash_{\hat{E}} \mathrm{E} : T!E'$ for some $T', \hat{E}, E'$ such that $T'!\hat{E}\vee E' \leq T!E$.*

Monadic progress is standard: a well-typed expression either is (the embedding of) a value or can reduce. For monadic subject reduction, if a well-typed expression reduces to a monadic one, then the monadic expression has a more specific type, and the expression effect is an upper bound of the computational effects produced by the current reduction step, described by $\hat{E}$, composed with those produced by future reductions, described by E'.

Remarkably enough, the proofs of monadic progress and subject reduction can be driven by the usual rule-based reasoning, without any need to know about Definition 13, thanks to Lemma 16 below. Recall that the interpretation of effects *lifts* predicates to monadic predicates, in our case typing judgments to monadic typing judgments. Lemma 16 shows that it is even possible to lift, in a sense, the type system itself. That is, we can derive a typing rule for each operator on monadic elements used in the semantics. The proofs in the following only rely on such monadic type system.

▶ **Lemma 16.** *The following rules can be derived.*

$$\text{(T-UNIT)}\ \frac{\vdash e : T!E}{\vdash_{\bullet} \eta(e) : T!E} \qquad \text{(T-RET-LIFT)}\ \frac{\vdash_{\hat{E}} \mathrm{V} : T}{\vdash_{\hat{E}} \mathsf{map}\,(\texttt{return}\ [\,])\ \mathrm{V} : T!\bullet}$$

$$\text{(T-DO-LIFT)}\ \frac{\vdash_{\hat{E}} \mathrm{E} : T!E \qquad \emptyset; x : T \vdash e' : T'!E'}{\vdash_{\hat{E}} \mathsf{map}\,(\texttt{do}\ x = [\,];\ e')\mathrm{E} : T'!E\vee E'}$$

Proof. By instantiating Definition 13(3) with $X =$ Exp and $A = \{e \mid \vdash e : T!E\}$, we get $A \subseteq \{e \mid \vdash_{\bullet} \eta(e) : T!E\}$, that is, $\vdash e : T!E$ implies $\vdash_{\bullet} \eta(e) : T!E$, as expressed by rule (T-UNIT).

We show how to derive (T-RET-LIFT), (T-DO-LIFT) can be derived analogously. Since, by rule (T-RET), $\vdash v{:}T$ implies $\vdash \texttt{return}\ v{:}T!\bullet$, by the monotonicity of $\lambda^{\hat{E}}$ we have that, set $Pre = \{v \mid \vdash v : T\}$, $Cons = \{v \mid \vdash \texttt{return}\ v : T!\bullet\}$, $\lambda^{\hat{E}}_{\mathsf{Val}}(Pre) \subseteq \lambda^{\hat{E}}_{\mathsf{Val}}(Cons)$. Since $\lambda^{\hat{E}}_{\mathsf{Val}}(Pre) = \{\mathrm{V} \mid$

$\vdash_{\hat{E}} \textsc{v} : T\}$, and, by Definition 13(1), instantiated with function (context) $\texttt{return} : \mathsf{Val} \to \mathsf{Exp}$ and $A = \{e \mid \vdash e{:}T!\bullet\}$, $\lambda^{\hat{E}}_{\mathsf{Val}}(\mathit{Cons}) = \{\textsc{v} \mid \vdash_{\hat{E}} \mathsf{map}\,(\texttt{return}\,[\,])\ \textsc{v} : T!\bullet\}$, we get that $\vdash_{\hat{E}} \textsc{v} : T$ implies $\vdash_{\hat{E}} \mathsf{map}\,(\texttt{return}\,[\,])\ \textsc{v} : T!\bullet$, as expressed by rule (T-RET-LIFT). ◀

Lemma 16 derives a typing rule for each monadic operator used in the semantics. In addition, the semantics uses monadic constants, which are the results of some $\mathsf{run}_{\mathsf{N},\mathsf{m}}$ function, depending on (the magic methods declared in) the specific program. Hence, like standard constants, we have to assume that these are well-typed. To this end, we require a typing rule for such constants, as given below. Essentially, since the (canonical) effect of a magic method m declared in N should be an approximation of the computational effects raised by a call, formalized by the monadic value obtained as result of $\mathsf{run}_{\mathsf{N},\mathsf{m}}$, then such monadic value should be well-typed with respect to the type-and-effect of the method.

$$\text{(T-RUN)}\ \frac{\vdash v_0 : T_0 \qquad \vdash \overline{v} : \overline{T'}}{\vdash_{\mathsf{N}[\overline{T_Y}].\mathsf{m}[\overline{T_X}]} \mathsf{run}_{\mathsf{N},\mathsf{m}}(\overline{v}) : T[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]} \quad \begin{array}{l} \mathsf{mtype}(\mathsf{N}[\overline{Y \triangleleft U_Y}] \triangleleft \_\{\_\}, \mathsf{m}) {=} \mathsf{mgc}\, MT \\ MT = [\overline{X \triangleleft U_X}]\overline{T} \to T!\mathsf{N}[\overline{Y}].\mathsf{m}[\overline{X}] \\ T_0 \leq \mathsf{N}[\overline{T_Y}] \\ \overline{T_Y \leq U_Y[\overline{T_Y}/\overline{Y}]} \\ \overline{T' \leq T[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]} \\ \overline{T_X \leq U_X[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]} \end{array}$$

More in detail, given a magic method m declared in N (first and second side conditions, where the effect is the canonical one), the result of a $\mathsf{run}_{\mathsf{N},\mathsf{m}}$ function is well-typed if the first argument has a subtype of one obtained by instantiating the type variables in N, and each other argument has a subtype of that obtained by instantiating the type variables in N and those in m with the corresponding parameter type. The type and the effect of the result are obtained by an analogous instantiation of variables of the return type-and-effect.

We provide below the proof of monadic subject reduction (Theorem 15), noteworthy since it relies on the monadic typing rules, that is, those derived in Lemma 16 and rule (T-RUN). The proofs of monadic progress and subject reduction for the pure relation (Lemma 17) can be found in [11], together with the standard inversion and substitution lemmas.

▶ **Lemma 17** (Subject Reduction). *If* $\vdash e{:}T!E$ *and* $e \to_p e'$, *then* $\vdash e'{:}T'!E'$ *and* $T'!E' {\leq} T!E$.

Proof of Theorem 15. Assuming $\vdash e{:}T!E$ and $e \to \textsc{e}$, we have to prove that $\vdash_{\hat{E}} \textsc{e} : T!E'$ for some $T', \hat{E}, E'$ such that $T'!\hat{E}{\vee}E' \leq T!E$. By induction on the reduction rules of Figure 5.

(pure) $\textsc{e}$ is $\eta(e')$ and $e \to_p e'$. From $\vdash e{:}T!E$ and Lemma 17 we get $\vdash e'{:}T'!E'$ with $T'!E' \leq T!E$. From Lemma 16 by rule (T-UNIT) we derive $\vdash_{\bullet} \eta(e') : T'!E'$, and $T'!\bullet \vee E \leq T!E$.

(mgc) e is $v_0.\mathsf{m}[\overline{T_X}](\overline{v})$ and $\textsc{e} = \mathsf{map}\,(\texttt{return}\,[\,])\,\mathsf{run}_{\mathsf{N},\mathsf{m}}(v, \overline{v})$ and $\mathsf{lookup}(v_0, \mathsf{m}) = \langle \mathsf{mgc}, \mathsf{N} \rangle$. By inversion we get $\vdash v_0{:}T_0$ and $\vdash \overline{v}{:}\overline{T}$, and, from the definition of lookup, $\mathsf{mkind}(T_0, \mathsf{m}) = \mathsf{mgc}$. Hence, again by inversion,

- $\mathsf{mtype}(td, \mathsf{m}) {=} \mathsf{mgc}\, [\overline{X \triangleleft U_X}]\overline{T''} \to T''!\mathsf{N}[\overline{Y}].\mathsf{m}[\overline{X}]$ for some $td = \mathsf{N}[\overline{Y \triangleleft U_Y}] \triangleleft \_\{\_\}$
- there is $\overline{T_Y}$ such that $T_0 \leq \mathsf{N}[\overline{T_Y}]$ and $\overline{T_Y \leq U_Y[\overline{T_Y}/\overline{Y}]}$
- $\overline{T_X \leq U_X[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]}$ and $\overline{T' \leq T''[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]}$ and $T = T''[\overline{T_Y}/\overline{Y}][\overline{T_X}/\overline{X}]$ and $E = \mathsf{N}[\overline{T_Y}].\mathsf{m}[\overline{T_X}]$.

Since the premises of rule (T-RUN) are satisfied, we get $\vdash_{E} \mathsf{run}_{\mathsf{N},\mathsf{m}}(v, \overline{v}) : T$. From Lemma 16 by rule (T-RET-LIFT) we get $\vdash_{E} \mathsf{map}\,(\texttt{return}\,[\,])\ \mathsf{run}_{\mathsf{N},\mathsf{m}}(v, \overline{v}) : T!\bullet$, and $T!E{\vee}\bullet \leq T!E$.

(ret) e is $\texttt{do}\ x = \texttt{return}\ v;\ e'$ and $\textsc{e} = \eta(e'[v/x])$. By inversion $\vdash v{:}T_1$ and $\emptyset; x{:}T_1 {\vdash} e'{:}T!E$. Therefore, by substitution, we get $\vdash e'[v/x]{:}T!E$. From Lemma 16 by (T-UNIT) we derive $\vdash_{\bullet} \eta(e'[v/x]) : T!E$, and $T!\bullet \vee E \leq T!E$.

(do) $e = \texttt{do}\ x = e_1\texttt{;}\ e_2$ and $\mathrm{E} = \texttt{map}\,(\texttt{do}\ x = [\,]\texttt{;}\ e_2)\,\mathrm{E}_1$ and $e_1 \rightarrow \mathrm{E}_1$. By inversion $E' = E_1 \vee E_2$ and $\vdash e_1{:}T_1!E_1$ and $\emptyset; x : T_1 \vdash e_2 : T!E_2$. By induction hypothesis $\vdash_{\hat{E}_1} \mathrm{E}_1 : T_1'!E_1'$ for some $\hat{E}_1$ and E_1' with $T_1'!\hat{E}_1 \vee E_1' \leq T_1!E_1$. From Lemma 16 by rule (T-DO-LIFT) we get $\vdash_{\hat{E}_1} \texttt{map}\,(\texttt{do}\ x = [\,]\texttt{;}\ e')\mathrm{E} : T!E_1' \vee E_2\mathrm{U}$. Therefore, we get the thesis noting that $T!\hat{E}_1 \vee E_1' \vee E_2 \leq T!E_1 \vee E_2 \leq T!E$. ◀

6 Related Work

Design of the pure language. As mentioned, the pure calculus we choose as basis for the effectful extension is inspired by recent work [49, 43, 50] proposing an object-oriented paradigm based on interfaces with default methods and anonymous classes, rather than class instances with fields. In particular, classless objects were also adopted in *interface-based programming* [49], enabling multiple inheritance by decoupling state and behaviour. Then, in *λ-based object-oriented programming* [43], objects are classless and stateless, but limited to be lambdas implementing a single abstract method. In the Fearless language [50], as in our calculus, this approach is generalized to arbitrary Java anonymous classes with no fields.

Design of the effectful language. In our effectful language, constructs to raise effects, similarly to generic effects [36], are just method calls of a special kind, called *magic*. As mentioned, this terminology is taken from Fearless [50], and also informally used in Java reflection and Python. Concerning constructs for handling effects, handlers of algebraic effects [38, 39, 24, 19] are an extremely powerful programming abstraction, describing the semantics of algebraic operations in the language itself, thus enabling the simulation of several effectful programs, such as stream redirection or cooperative concurrency [39, 24, 6, 41]. The code executed when a call is caught can resume the original computation, using a form of continuation-passing style. Differently from exception handling, all occurrences of an operation within the scope of a handler are handled. Other forms of handlers have been considered, e.g., shallow handlers [24, 19], where only the first call is handled. The handling mechanism in our language is meant to provide a good compromise between simplicity and expressivity: notably, it does not explicitly handle continuations, as in handlers of algebraic effects, hence we cannot resume the original computation more than once; yet we can express a variety of handling mechanisms.

Type-and-effect system. Type-and-effect systems, or simply effect systems [47, 33, 48, 30, 25], are a popular way of statically controlling computational effects. Many of them have been designed for specific notions of computational effect and also implemented in mainstream programming languages, the most well-known being the mechanism of Java 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.

Our effects are essentially sets, like in formal models of Java exceptions [3], tracking which magic calls an expression is allowed to do. Differently from effect systems for algebraic effects [5, 6, 41], also implemented in Eff [4], and inspired by [15], we enrich the effects with type information, not using just the name of magic methods. By using this type information, which is polymorphic, we also allow effects to depend on the effect of other methods, smoothly enabling a form of effect polymorphism. Effect systems supporting effect polymorphism have been widely studied for algebraic effects [26, 20, 29, 7, 9, 8], especially in the form of row polymorphism. A precise comparison with these systems is left for future work.

Semantics for effectful languages. Algebraic effects are typically treated as uninterpreted operations, that is, the evaluation process just builds a tree of operation calls [23, 44, 41]. Monadic operational semantics for λ-calculi with algebraic effects are also considered, mainly in the form of a monadic definitional interpreter (see, e.g., [28, 14, 16, 13, 10]). 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 [17, 18].

An alternative way of interpreting algebraic operations is by means of runners, a.k.a. comodels [40, 37, 45, 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.

The motivation to have methods interpreted in a monad, rather than implemented in code, is the same nicely illustrated for runners by [2]. They observe that some kinds of effects, such as input/output, even though represented in the language as algebraic operations (as in Eff) or a monad (as in Haskell), are handled by an operating system functionality. Hence, semantics lacks modularity, which can be recovered by introducing runners. The same modularity is achieved in our calculus by providing the interpretations of magic methods as an independent ingredient of the semantics, which can be regarded as the operational counterpart of (effectful) runners. Differently from runners of [2], our magic methods can be interpreted in an arbitrary monad. Moreover, our interpretation is a parameter of the operational semantics, while runners of [2] are described in the language itself through a mechanism similar to effect handlers, with a limited use of continuations.

7 Conclusion

This paper provides a positive answer to the question:

Is it possible to smoothly incorporate in the object-oriented paradigm constructs to raise, compose, and handle effects in an arbitrary monad?

In particular, we consider a pure OO paradigm based on interfaces with default methods and anonymous classes, exemplified by a higher-order calculus subsuming both FJ and λ-calculus. The effectful extension is designed to be familiar for OO programmers, with magic methods to raise effects, and an expressive, yet simple, handling mechanism. Effect types are designed to be familiar, being sets of call-effects generalizing exception names; the key novel feature is that type effects containing type variables can be refined in concrete calls, through a non-trivial process called *simplification.* The semantics and the soundness proof are given by a novel approach [12], providing a significant application for a language with a complex type system (both nominal and structural types, subtyping and generic types). Instantiating such an approach, we derive from the type system a *monadic type system*, allowing to carry the proof of subject reduction by the usual rule-based reasoning.

The integration of effectful features turned out to be, in fact, even smoother than in the functional case, fitting naturally in the OO paradigm, where computations happen in the context of a given program (class table in Java). That is, magic methods are, from the point of view of the programmer, as other methods offered by the environment (system, libraries), with the only difference that their calls can be handled. Another advantage is that, since methods "belong to a type", atomic effects (call-effects in our paper) naturally include a type, rather than being just operation names as in previous literature handling the functional case. Thus, effect polymorphism is obtained for free from the fact that types can be type variables, without any need of ad-hoc effect variables.

Concerning the specific calculus, we could have likely used FJ (with generic types) as well. However we have, besides nominal, structural and intersection types, hence a paradigm partly object-based, as in seminal object calculi [1], rather than class-based. This leads to more flexibility, hence the tracking of effects is more flexible as well (can be done on an object's basis). In particular, FJ is not higher-order, meaning that functions are not first-class values as customary in functional languages. Formally, in the translation of λ-calculus shown in Remark 1, any arbitrary function is encoded by an object literal; without object literals, only a finite collection of functions could be encoded, each one by a class, in a program.

This is a foundational paper, hence we focus on the introduction of design features, formalisation, and results through a toy language. Of course, any step towards an implementation and/or the application to a realistic language would be an important contribution, as it would be to mechanize some proofs. Other directions for future work are outlined below.

We plan to analyze better the handling mechanism proposed in the paper, and its relation with other approaches. The type-and-effect system we design is, as said above, novel and expressive; however, it is very simple in the sense that effect types are essentially sets. Hence, it is not adequate to approximate computational effects where the order matters, such as, e.g., sequences of write operations. A type-and-effect system with a non-commutative operator on effects is shown in [12]; however, effects there are purely semantics, and a syntactic representation should be investigated in order to apply the approach to real languages.

Finally, the interpretation of magic methods could return monadic expressions, rather than values. This would enable a more interactive behaviour with the system; for instance, the semantics of a magic method, instead of returning an unrecoverable error, could return a call to the method `throw` of an exception, which then could be handled by the program.

References

1 Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996. doi:10.1006/INCO.1996.0024.

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 Davide Ancona, Giovanni Lagorio, and Elena Zucca. A core calculus for Java exceptions. In Linda M. Northrop and John M. Vlissides, editors, *Proceedings of the ACM International Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA 2001*, pages 16–30. ACM Press, 2001. doi:10.1145/504282.504284.

4 Andrej Bauer and Matija Pretnar. Eff. URL: https://www.eff-lang.org/.

5 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014. doi:10.2168/LMCS-10(4:9)2014.

6 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.

7 Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Abstracting algebraic effects. *Proceedings of the ACM on Programming Languages*, 3(POPL):6:1–6:28, 2019. doi:10.1145/3290319.

8 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):126:1–126:30, 2020. doi:10.1145/3428194.

9 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. *Journal of Functional Programming*, 30:e8, 2020. doi:10.1017/S0956796820000027.

10 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.

11 Francesco Dagnino, Paola Giannini, and Elena Zucca. An effectful object calculus (extended version). *CoRR*, 2025. doi:10.48550/arXiv.2504.15936.

12 Francesco Dagnino, Paola Giannini, and Elena Zucca. Monadic type-and-effect soundness. In Jonathan Aldrich and Alexandra Silva, editors, *39th European Conference on Object-Oriented Programming, ECOOP 2025*, volume 333 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ECOOP.2025.1.

13 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.

14 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.

15 Isaac Oscar Gariano, James Noble, and Marco Servetto. Calle: an effect system for method calls. In Hidehiko Masuhara and Tomas Petricek, editors, *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019*, pages 32–45. ACM, 2019. doi:10.1145/3359591.3359731.

16 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.

17 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.

18 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.

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 Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. Continuation passing style for effect handlers. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017*, volume 84 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.FSCD.2017.18.

21 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the ACM International Conference on Object-Oriented Programming: Systems, Languages and Applications, OOPSLA 1999*, pages 132–146. ACM Press, 1999. doi:10.1145/320384.320395.

22 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.

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 Daan Leijen. Type directed compilation of row-typed algebraic effects. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 486–499. ACM, 2017. `doi:10.1145/3009837.3009872`.

27 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`.

28 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`.

29 Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM/SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 500–514. ACM, 2017. `doi:10.1145/3009837.3009897`.

30 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`.

31 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`.

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

33 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`.

34 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`.

35 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`.

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

37 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`.

38 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`.

39 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`.

40 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.

41 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.

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

43 Marco Servetto and Elena Zucca. λ-based object-oriented programming (pearl). In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021*, volume 194 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ECOOP.2021.21.

44 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.

45 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.

46 Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995. doi:10.1007/3-540-59451-5_2.

47 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.

48 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.

49 Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. Classless Java. In Bernd Fischer and Ina Schaefer, editors, *Generative Programming: Concepts and Experiences, GPCE 2016*, pages 14–24. ACM Press, 2016. doi:10.1145/2993236.2993238.

50 Nick Webster, Marco Servetto, and Michael Homer. The Fearless journey [Draft]. *CoRR*, abs/2405.06233, 2014. doi:10.48550/arXiv.2405.06233.