Effectful Normal Form Bisimulation

Ugo Dal Lago, Francesco Gavazzo

Effectful Normal Form Bisimulation *

Ugo Dal Lago and Francesco Gavazzo

University of Bologna and INRIA Sophia Antipolis

Abstract. Normal form bisimulation, also known as *open* bisimulation, is a coinductive technique for higher-order program equivalence in which programs are compared by looking at their essentially infinitary tree-like normal forms, i.e. at their Böhm or Lévy-Longo trees. The technique has been shown to be useful not only when proving metatheorems about λ-calculi and their semantics, but also when looking at concrete examples of terms. In this paper, we show that there is a way to generalise normal form bisimulation to calculi with algebraic effects, *à la* Plotkin and Power. We show that some mild conditions on monads and relators, which have already been shown to guarantee effectful applicative bisimilarity to be a congruence relation, are enough to prove that the obtained notion of bisimilarity, which we call *effectful normal form bisimilarity*, is a congruence relation, and thus sound for contextual equivalence. Additionally, contrary to applicative bisimilarity, normal form bisimilarity allows for enhancements of the bisimulation proof method, hence proving a powerful reasoning principle for effectful programming languages.

1 Introduction

The study of program equivalence has always been one of the central tasks of programming language theory: giving satisfactory definitions and methodologies for it can be fruitful in contexts like program verification and compiler optimisation design, but also helps in understanding the *nature* of the programming language at hand. This is particularly true when dealing with higher-order languages, in which giving satisfactory notions of program equivalence is well-known to be hard. Indeed, the problem has been approached in many different ways. One can define program equivalence through denotational semantics, thus relying on a model. One could also proceed following the route traced by Morris [51], and define programs to be *contextually* equivalent when they behave the same in every context, this way taking program equivalence as the *largest* adequate congruence.

Both these approaches have their drawbacks, the first one relying on the existence of a (not too coarse) denotational model, the latter quantifying over all contexts, and thus making concrete proofs of equivalence hard. Among the many alternative techniques the research community has been proposing along the years, one can cite logical relations and applicative bisimilarity [4,8,1], both based on the idea that equivalent higher-order terms should behave the same

* Thanks to the ANR projects 14CE250005 ELICA and 16CE250011 REPAS.

when fed with any (pair of related) inputs. This way, terms are compared mimicking any possible action a discriminating context could possibly perform on the tested terms. In other words, the universal quantification on all possible contexts, although not *explicitly* present, is anyway *implicitly* captured by the bisimulation or logical game.

Starting from the pioneering work by Böhm, another way of defining program equivalence has been proved extremely useful not only when giving metatheorems about λ-calculi and programming languages, but also when proving concrete programs to be (contextually) equivalent. What we are referring to, of course, is the notion of a *Böhm tree* of a λ-term e (see [5] for a formal definition), which is a possibly infinite tree representing the *head normal* h form of e, if e has one, but also analyzing the arguments to the head variable of h in a coinductive way. The celebrated Böhm Theorem, also known as Separation Theorem [11], stipulates that two terms are contextually equivalent *if and only if* their respective (appropriately η-equated) Böhm trees are the same.

The notion of equivalence induced by Böhm trees can be characterised without any reference to trees, by means of a suitable bisimilarity relation [37,65]. Additionally, Böhm trees can also be defined when λ-terms are *not* evaluated to their *head* normal form, like in the classical theory of λ-calculus, but to their *weak head* normal form (like in the call-by-name [37,65]), or to their *eager* normal form (like in the call-by-value λ-calculus [38]). In both cases, the notion of program equivalence one obtains by comparing the syntactic structure of trees, admits an elegant coinductive characterisation as a suitable bisimilarity relation. The family of bisimilarity relations thus obtained goes under the generic name of *normal form bisimilarity*.

Real world functional programming languages, however, come equipped not only with higher-order functions, but also with *computational effects*, turning them into *impure* languages in which functions cannot be seen merely as turning an input to an output. This requires switching to a new model, which cannot be the usual, pure, λ-calculus. Indeed, program equivalence in effectful λ-calculi [56,49] have been studied by way of denotational semantics [31,20,18], logical relations [10,14], applicative bisimilarity [36,16,13], and normal form bisimilarity [20,41]. While the denotational semantics, logical relation semantics, and applicative bisimilarity of effectful calculi have been studied in the abstract [25,30,15], the same cannot be said about normal form bisimilarity. Particularly relevant for our purposes is [15], where a notion of applicative bisimilarity for generic algebraic effects, called *effectful applicative bisimilarity*, based on the (standard) notion of a monad, and on the (less standard) notion of a *relator* [71] or *lax extension* [6,26], is introduced.

Intuitively, a relator is an abstraction axiomatising the structural properties of relation lifting operations. This way, relators allow for an abstract description of the possible ways a relation between programs can be lifted to a relation between (the results of) effectful computations, the latter being described throughout monads and algebraic operations. Several concrete notions of program equivalence, such as pure, nondeterministic and probabilistic applicative bisimilarity

[1,36,52,16] can be analysed using relators. Additionally, besides their prime role in the study of effectful applicative bisimilarity, relators have also been used to study logic-based equivalences [67] and applicative distances [23] for languages with generic algebraic effects.

The main contribution of [15] consists in devising a set of axioms on monads and relators (summarised in the notions of a *Σ-continuous monad* and a *Σ-continuous relator*) which are both satisfied by many concrete examples, and that abstractly guarantee that the associated notion of applicative bisimilarity is a congruence.

In this paper, we show that an abstract notion of normal form (bi)simulation can indeed be given for calculi with algebraic effects, thus defining a theory analogous to [15]. Remarkably, we show that the defining axioms of Σ-continuous monads and Σ-continuous relators guarantee the resulting notion of normal form (bi)similarity to be a (pre)congruence relation, thus enabling compositional reasoning about program equivalence and refinement. Given that these axioms have already been shown to hold in many relevant examples of calculi with effects, our work shows that there is a way to "cook up" notions of *effectful* normal form bisimulation *without* having to reprove congruence of the obtained notion of program equivalence: this comes somehow for free. Moreover, this holds both when call-by-name and call-by-value program evaluation is considered, although in this paper we will mostly focus on the latter, since the call-by-value reduction strategy is more natural in presence of computational effects[1].

Compared to (effectful) applicative bisimilarity, as well as to other standard operational techniques — such as contextual and CIU equivalence [51,47], or logical relations [61,55] — (effectful) normal form bisimilarity has the major advantage of being an *intensional* program equivalence, equating programs according to the syntactic structure of their (possibly infinitary) normal forms. As a consequence, in order to deem two programs as normal form bisimilar, it is sufficient to test them in isolation, i.e. independently of their interaction with the environment. This way, we obtain easier proofs of equivalence between (effectful) programs. Additionally, normal form bisimilarity allows for enhancements of the bisimulation proof method [60], hence qualifying as a powerful and effective tool for program equivalence.

Intensionality represents a major difference between normal form bisimilarity and applicative bisimilarity, where the environment interacts with the tested programs by passing them arbitrary input arguments (thus making applicative bisimilarity an *extensional* notion of program equivalence). Testing programs in isolation has, however, its drawbacks. In fact, although we prove effectful normal form bisimilarity to be a sound proof technique for (effectful) applicative bisimilarity (and thus for contextual equivalence), full abstraction fails, as already observed in the case of the pure λ-calculus [3,38] (nonetheless, it is worth mentioning that full abstraction results are known to hold for calculi with a rich

[1] Besides, as we will discuss in Section 6.4, the formal analysis of call-by-name normal form bisimilarity strictly follows the corresponding (more challenging) analysis of call-by-value normal form bisimilarity.

expressive power [65,68]).

In light of these observations, we devote some energy to studying some concrete examples which highlight the weaknesses of applicative bisimilarity, on the one hand, and the strengths of normal form bisimilarity, on the other hand.

This paper is structured as follows. In Section 2 we informally discuss examples of (pairs of) programs which are operational equivalent, but whose equivalence cannot be readily established using standard operational methods. Throughout this paper, we will show how effectful normal form bisimilarity allows for handy proofs of such equivalences. Section 3 is dedicated to mathematical preliminaries, with a special focus on (selected) examples of monads and algebraic operations. In Section 4 we define our vehicle calculus Λ_{Σ}, an untyped λ-calculus enriched with algebraic operations, to which we give call-by-value monadic operational semantics. Section 5 introduces relators and their main properties. In Section 6 we introduce *effectful eager normal form (bi)similarity*, the call-by-value instantiation of effectful normal form (bi)similarity, and its main metatheoretical properties. In particular, we prove effectful eager normal form (bi)similarity to be a (pre)congruence relation (Theorem 2) included in effectful applicative (bi)similarity (Proposition 5). Additionally, we prove soundness of eager normal bisimulation up-to context (Theorem 3), a powerful enhancement of the bisimulation proof method that allows for handy proof of program equivalence. Finally, in Section 6.4 we briefly discuss how to modify our theory to deal with call-by-name calculi.

2 From Applicative to Normal Form Bisimilarity

In this section, some examples of (pairs of) programs which can be shown equivalent by effectful normal form bisimilarity will be provided, giving evidence on the flexibility and strength of the proposed technique. We will focus on examples drawn from fixed point theory, simply because these, being infinitary in nature, are quite hard to be dealt with "finitary" techniques like contextual equivalence or applicative bisimilarity.

Example 1. Our first example comes from the ordinary theory of pure, untyped λ-calculus. Let us consider Curry's and Turing's call-by-value fixed point combinators Y and Z:

$$Y \triangleq \lambda y.\Delta\Delta, \quad Z \triangleq \Theta\Theta, \quad \Delta \triangleq \lambda x.y(\lambda z.xxz), \quad \Theta \triangleq \lambda x.\lambda y.y(\lambda z.xxyz).$$

It is well known that Y and Z are contextually equivalent, although proving such an equivalence from first principles is doomed to be hard. For that reason, one usually looks at proof techniques for contextual equivalence. Here we consider applicative bisimilarity [1]. As in the pure λ-calculus applicative bisimilarity coincides with the intersection of applicative similarity and its converse, for the sake of the argument we discuss which difficulties one faces when trying to prove Z to be applicatively similar to Y.

Let us try to construct an applicative simulation $\mathcal{R}$ relating Y and Z. Clearly we need to have $(Y, Z) \in \mathcal{R}$. Since Y evaluates to $\lambda y.\Delta\Delta$, and Z evaluates to $\lambda y.y(\lambda z.\Theta\Theta yz)$, in order for $\mathcal{R}$ to be an applicative simulation, we need to show that for any value v, $(\Delta[v/y]\Delta[v/y], v(\lambda z.\Theta\Theta vz)) \in \mathcal{R}$. Since the result of the evaluation of $\Delta[v/y]\Delta[v/y]$ is the same of $v(\lambda z.\Delta[v/y]\Delta[v/y]z)$, we have reached a point in which we are stuck: in order to ensure $(Y, Z) \in \mathcal{R}$, we need to show that $(v(\lambda z.\Delta[v/y]\Delta[v/y]z), v(\lambda z.\Theta\Theta vz)) \in \mathcal{R}$. However, the value v being provided by the environment, no information on it is available. That is, we have no information on how v tests its input program. In particular, given any context $\mathcal{C}[-]$, we can consider the value $\lambda x.\mathcal{C}[x]$, meaning that proving Y and Z to be applicatively bisimilar is almost as hard as proving them to be contextually equivalent from first principles.

As we will see, proving Z to be normal form similar to Y is straightforward, since in order to test $\lambda y.\Delta\Delta$ and $\lambda y.y(\lambda z.\Theta\Theta yz)$, we simply test their subterms $\Delta\Delta$ and $y(\lambda z.\Theta\Theta yz)$, thus not allowing the environment to influence computations.

Example 2. Our next example is a refinement of Example 1 to a probabilistic setting, as proposed in [66] (but in a call-by-name setting). We consider a variation of Turing's call-by-value fixed point combinator which, at any iteration, can probabilistically decide whether to start another iteration (following the pattern of the standard Turing's fixed point combinator) or to turn for good into Y, where Y and Δ are defined as in Example 1:

$$Z \triangleq \Theta\Theta, \qquad \Theta \triangleq \lambda x.\lambda y.(y(\lambda z.\Delta\Delta z) \mathbf{\ or\ } y(\lambda z.xxyz)).$$

Notice that the constructor **or** behaves as a (fair) probabilistic choice operator, hence acting as an effect producer. It is natural to ask whether these new versions of Y and Z are still equivalent. However, following insights from previous example, it is not hard to see the equivalence between Y and Z cannot be readily proved by means of standard operational methods such as probabilistic contextual equivalence [16], probabilistic CIU equivalence and logical relations [10], and probabilistic applicative bisimilarity [16,13]. All the aforementioned techniques require to test programs in a given environment (such as a whole context or an input argument), and are thus ineffective in handling fixed point combinators such as Y and Z.

We will give an elementary proof of the equivalence between Y and Z in Example 17, and a more elegant proof relying on a suitable *up-to context* technique in Example 18. In [66], the call-by-name counterparts of Y and Z are proved to be equivalent using probabilistic environmental bisimilarity. The notion of an environmental bisimulation [63] involves both an environment storing pairs of terms played during the bisimulation game, and a clause universally quantifying over pairs of terms in the evaluation context closure of such an environment[2], thus making environmental bisimilarity a rather heavy technique to use. Our

[2] Meaning that two terms e_1, e_2 are tested for their applicative behaviour against all terms of the form $E[e], E[e']$, for any pair of terms (e, e') stored in the environment.

proof of the equivalence of Y and Z is simpler: in fact, our notion of effectful normal form bisimulation does not involve any universal quantification over all possible closed function arguments (like applicative bisimilarity), or their evaluation context closure (like environmental bisimilarity), or closed instantiation of uses (like CIU equivalence).

Example 3. Our third example concerns call-by-name calculi and shows how our notion of normal form bisimilarity can handle even intricate recursion schemes. We consider the following argument-switching probabilistic fixed point combinators:

$$P \triangleq AA, \qquad A \triangleq \lambda x.\lambda y.\lambda z.(y(xxyz) \mathbf{\ or\ } z(xxzy)),$$
$$Q \triangleq BB, \qquad B \triangleq \lambda x.\lambda y.\lambda z.(y(xxzy) \mathbf{\ or\ } z(xxyz)).$$

We easily see that P and Q satisfy the following (informal) program equations:

$$Pef = e(Pef) \mathbf{\ or\ } f(Pfe), \qquad Qef = e(Qfe) \mathbf{\ or\ } f(Qef).$$

Again, proving the equivalence between P and Q using applicative bisimilarity is problematic. In fact, testing the applicative behaviour of P and Q requires to reason about the behaviour of e.g. $e(Pef)$, which in turn requires to reason about the (arbitrary) term e, on which no information is provided. The (essentially infinitary) normal forms of P and Q, however, can be proved to be essentially the same by reasoning about the syntactical structure of P and Q. Moreover, our *up-to context* technique enables an elegant and concise proof of the equivalence between P and Q (Section 6.4).

Example 4. Our last example discusses the use of the cost monad as an *instrument* to facilitate a more intensional analysis of programs. In fact, we can use the ticking operation **tick** to perform cost analysis. For instance, we can consider the following variation of Curry's and Turing's fixed point combinator of Example 1, obtained by adding the operation symbol **tick** after every λ-abstraction.

$$Y \triangleq \lambda y.\mathbf{tick}(\Delta\Delta), \qquad \Delta \triangleq \lambda x.\mathbf{tick}(y(\lambda z.\mathbf{tick}(xxz))),$$
$$Z \triangleq \Theta\Theta, \qquad \Theta \triangleq \lambda x.\mathbf{tick}(\lambda y.\mathbf{tick}(y(\lambda z.\mathbf{tick}(xxyz)))).$$

Every time a β-redex $(\lambda x.\mathbf{tick}(e))v$ is reduced, the ticking operation **tick** increases an imaginary cost counter of a unit. Using ticking, we can provide a more intensional analysis of the relationship between Y and Z, along the lines of Sands' improvement theory [62].

3 Preliminaries: Monads and Algebraic Operations

In this section we recall some basic definitions and results needed in the rest of the paper. Unfortunately, there is no hope to be comprehensive, and thus we assume the reader to be familiar with basic domain theory [2] (in particular

with the notions of ω-complete (pointed) partial order — ω-cppo, for short — monotone, and continuous functions), basic order theory [19], and basic category theory [46]. Additionally, we assume the reader to be acquainted with the notion of a Kleisli triple [46] $\mathbb{T} = \langle T, \eta, -^\dagger \rangle$. As it is customary, we use the notation $f^\dagger : TX \to TY$ for the Kleisli extension of $f : X \to TY$, and reserve the letter η to denote the unit of $\mathbb{T}$. Due to their equivalence, oftentimes we refer to Kleisli triples as monads.

Concerning notation, we try to follow [46] and [2], with the only exception that we use the notation $(x_n)_n$ to denote an ω-chain $x_0 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ in a domain $(X, \sqsubseteq, \bot)$. The notation $\mathbb{T} = \langle T, \eta, -^\dagger \rangle$ for an arbitrary Kleisli triple is standard, but it is not very handy when dealing with multiple monads at the same time. To fix this issue, we sometimes use the notation $\mathbb{T} = \langle T, \mathrm{T}, -^{\mathbb{T}} \rangle$ to denote a Kleisli triple. Additionally, when unambiguous we omit subscripts. Finally, we denote by Set the category of sets and functions, and by Rel the category of sets and relations. We reserve the symbol 1 to denote the identity function. Unless explicitly stated, we assume functors (and monads) to be functors (and monads) on Set. As a consequence, we write *functors* to refer to endofunctors on Set.

We use monads to give operational semantics to our calculi. Following Moggi [49,50], we model notions of computation as monads, meaning that we use monads as mathematical models of the kind of (side) effects computations may produce. The following are examples of monads modelling relevant notions of computation. Due to space constraints, we omit several interesting examples such as the output, the exception, and the nondeterministic/powerset monad, for which the reader is referred to e.g. [50,73].

Example 5 (Partiality). Partial computations are modelled by the partiality (also called maybe) monad $\mathbb{M} = \langle M, \mathrm{M}, -^{\mathbb{M}} \rangle$. The carrier MX of $\mathbb{M}$ is defined as $\{just\ x \mid x \in X\} \cup \{\bot\}$, where $\bot$ is a special symbol denoting divergence. The unit and Kleisli extension of $\mathbb{M}$ are defined as follows:

$$\mathrm{M}(x) \triangleq just\ x, \qquad f^{\mathbb{M}}(just\ x) \triangleq f(x), \qquad f^{\mathbb{M}}(\bot) \triangleq \bot.$$

Example 6 (Probabilistic Nondeterminism). In this example we assume sets to be countable[3]. The (discrete) distribution monad $\mathbb{D} = \langle D, \mathrm{D}, -^{\mathbb{D}} \rangle$ has carrier $\mathbb{D}X \triangleq \{\mu : X \to [0,1] \mid \sum_x \mu(x) = 1\}$, whereas the maps D and $-^{\mathbb{D}}$ are defined as follows (where $y \neq x$):

$$\mathrm{D}(x)(x) \triangleq 1, \qquad \mathrm{D}(x)(y) \triangleq 0, \qquad f^{\mathbb{D}}(\mu)(y) \triangleq \textstyle\sum_{x \in X} \mu(x) \cdot f(x)(y).$$

Oftentimes, we write a distribution μ as a weighted formal sum. That is, we write μ as the sum[4] $\sum_{i \in I} p_i \cdot x_i$ such that $\mu(x) = \sum_{x_i = x} p_i$. $\mathbb{D}$ models probabilistic total computations, according to the rationale that a (total) probabilistic program

[3] Although this is not strictly necessary, for simplicity we work with distributions over countable sets only, as the sets of values and normal forms are countable.

[4] For simplicity, we write only those p_is such that $p_i > 0$.

evaluates to a distribution over values, the latter describing the possible results of the evaluation. Finally, we model probabilistic partial computations using the monad $\mathbb{DM} = \langle DM, \mathrm{DM}, -^{\mathbb{DM}} \rangle$. The carrier of $\mathbb{DM}$ is defined as $DMX \triangleq D(MX)$, whereas the unit DM is defined in the obvious way. For $f : X \to DMY$, define:

$$f^{\mathbb{DM}}(\mu)(y) \triangleq \sum_{x \in X} \mu(just\ x) \cdot f(x)(y) + \mu(\bot) \cdot \mathrm{D}(\bot)(y).$$

It is easy to see that $\mathbb{DM}$ is isomorphic to the subdistribution monad.

Example 7 (Cost). The cost (also known as ticking or improvement [62]) monad $\mathbb{C} = \langle C, \mathrm{C}, -^{\mathbb{C}} \rangle$ has carrier $CX \triangleq M(\mathbb{N} \times X)$. The unit of $\mathbb{C}$ is defined as $\mathrm{C}(x) \triangleq just\ (0, x)$, whereas Kleisli extension is defined as follows:

$$f^{\mathbb{C}}(\chi) \triangleq \begin{cases} \bot & \text{if } \chi = \bot, \text{ or } \chi = just\ (n, x) \text{ and } f(x) = \bot \\ just\ (n + m, y) & \text{if } \chi = just\ (n, x) \text{ and } f(x) = just\ (m, y). \end{cases}$$

The cost monad is used to model the cost of (partial) computations. An element of the form $just\ (n, x)$ models the result of a computation outputting the value x with cost n (the latter being an abstract notion that can be instantiated to e.g. the number of reduction steps performed). Partiality is modelled as the element $\bot$, according to the rationale that we can assume all divergent computations to have the same cost, so that such information need not be explicitly written (for instance, measuring the number of reduction steps performed, we would have that divergent computations all have cost ∞).

Example 8 (Global states). Let $\mathcal{L}$ be a set of public location names. We assume the content of locations to be encoded as families of values (such as numerals or booleans) and denote the collection of such values as $\mathcal{V}$. A store (or state) is a function $\sigma : \mathcal{L} \to \mathcal{V}$. We write S for the set of stores $\mathcal{V}^{\mathcal{L}}$. The global state monad $\mathbb{G} = \langle G, \mathrm{G}, -^{\mathbb{G}} \rangle$ has carrier $GX \triangleq (X \times S)^S$, whereas G and $-^{\mathbb{G}}$ are defined by:

$$\mathrm{G}(x)(\sigma) \triangleq (x, \sigma), \qquad f^{\mathbb{G}}(\alpha)(\sigma) \triangleq f(x')(\sigma'),$$

where $\alpha(\sigma) = (x', \sigma')$. It is straightforward to see that we can combine the global state monad with the partiality monad, obtaining the monad $\mathbb{M} \otimes \mathbb{G}$ whose carrier is $(M \otimes G)X \triangleq M(X \times S)^S$. In a similar fashion, we see that we can combine the global state monad with $\mathbb{DM}$ and $\mathbb{C}$, as we are going to see in Remark 1.

Remark 1. The monads $\mathbb{DM}$ and $\mathbb{M} \otimes \mathbb{G}$ of Example 6 and Example 8, respectively, are instances of two general constructions, namely the *sum* and *tensor* of effects [28]. Although these operations are defined on Lawvere theories [40,29], here we can rephrase them in terms of monads as follows.

Proposition 1. *Given a monad $\mathbb{T} = \langle T, \mathrm{T}, -^{\mathbb{T}} \rangle$, define the* sum $\mathbb{TM}$ *of $\mathbb{T}$ and $\mathbb{M}$ and the* tensor $\mathbb{T} \otimes \mathbb{G}$ *of $\mathbb{T}$ and $\mathbb{G}$, as the triples $\langle TM, \mathrm{TM}, -^{\mathbb{TM}} \rangle$ and $\langle T \otimes G, \mathrm{T} \otimes \mathrm{G}, -^{\mathbb{T} \otimes \mathbb{G}} \rangle$, respectively. The carriers of the triples are defined as $TMX \triangleq T(MX)$ and $(T \otimes G)X \triangleq T(S \times X)^S$, whereas the maps TM and $\mathrm{T} \otimes \mathrm{G}$ are defined as $\mathrm{TM}_X \triangleq \mathrm{T}_{MX} \circ \mathrm{M}_X$ and $(\mathrm{T} \otimes \mathrm{G})_X \triangleq \mathsf{curry}\ \mathrm{T}_{S \times X}$, respectively. Finally, define:*

$$f^{\mathbb{TM}} \triangleq (f_M)^{\mathbb{T}}, \qquad f^{\mathbb{T} \otimes \mathbb{G}}(\alpha)(\sigma) \triangleq (\mathsf{uncurry}\ f)^{\mathbb{T}}(\alpha)(\sigma),$$

where, for a function $f : X \to TMY$ *we define* $f_M : MX \to TMY$ *as* $f_M(\bot) \triangleq \mathrm{T}_{MX}(\bot)$, $f_M(just\ x) \triangleq f(x)$, *and* curry *and* uncurry *are defined as usual. Then* $\mathbb{T}\mathbb{M}$ *and* $\mathbb{T} \otimes \mathbb{G}$ *are monads.*

Proving Proposition 1 is a straightforward exercise (the reader can also consult [28]). We notice that tensoring $\mathbb{G}$ with $\mathbb{D}\mathbb{M}$ we obtain a monad for probabilistic imperative computations, whereas tensoring $\mathbb{G}$ with $\mathbb{C}$ we obtain a monad for imperative computations with cost.

3.1 Algebraic Operations

Monads provide an elegant way to structure effectful computations. However, they do not offer any actual effect constructor. Following Plotkin and Power [56,57,58], we use *algebraic operations* as effect producers. From an operational perspective, algebraic operations are those operations whose behaviour is independent of their continuations or, equivalently, of the environment in which they are evaluated. Intuitively, that means that e.g. $E[e_1\ \mathbf{or}\ e_2]$ is operationally equivalent to $E[e_1]\,\mathbf{or}\,E[e_2]$, for any evaluation context E. Examples of algebraic operations are given by (binary) nondeterministic and probabilistic choices as well as primitives for rising exceptions and output operations.

Syntactically, algebraic operations are given via a signature Σ consisting of a set of operation symbols (uninterpreted operations) together with their arity (i.e. their number of operands). Semantically, operation symbols are interpreted as algebraic operations on monads. To any n-ary operation symbol[5] $(\mathbf{op} : n) \in \Sigma$ and any set X we associate a map $[\![\mathbf{op}]\!]_X : (TX)^n \to TX$ (so that we equip TX with a Σ-algebra structure [12]) such that $f^\dagger$ is Σ-algebra morphism, meaning that for any $f : X \to TY$, and elements $x_1, \ldots, x_n \in TX$ we have $[\![\mathbf{op}]\!]_Y(f^\dagger(x_1), \ldots, f^\dagger(x_n)) = f^\dagger([\![\mathbf{op}]\!]_X(x_1, \ldots, x_n))$.

Example 9. The partiality monad $\mathbb{M}$ usually comes with no operation, as the possibility of divergence is an implicit feature of any Turing complete language. However, it is sometimes useful to add an explicit divergence operation (for instance, in strongly normalising calculi). For that, we consider the signature $\Sigma_{\mathbb{M}} \triangleq \{\underline{\Omega} : 0\}$. Having arity zero, the operation $\underline{\Omega}$ acts as a constant, and has semantics $[\![\underline{\Omega}]\!] = \bot$. Since $f^{\mathbb{M}}(\bot) = \bot$, we see that $\underline{\Omega}$ in indeed an algebraic operation on $\mathbb{M}$.

For the distribution monad $\mathbb{D}$ we define the signature $\Sigma_{\mathbb{D}} \triangleq \{\mathbf{or} : 2\}$. The intended semantics of a program $e_1\ \mathbf{or}\ e_2$ is to evaluate to e_i $(i \in \{1, 2\})$ with probability 0.5. The interpretation of **or** is defined by $[\![\mathbf{or}]\!](\mu, \nu)(x) \triangleq 0.5 \cdot \mu(x) + 0.5 \cdot \nu(x)$. It is easy to see that **or** is an algebraic operation on $\mathbb{D}$, and that it trivially extends to $\mathbb{D}\mathbb{M}$.

Finally, for the cost monad $\mathbb{C}$ we define the signature $\Sigma_{\mathbb{C}} \triangleq \{\mathbf{tick} : 1\}$. The intended semantics of **tick** is to add a unit to the cost counter:

$$[\![\mathbf{tick}]\!](\bot) \triangleq \bot, \qquad [\![\mathbf{tick}]\!](just\ (n, x)) \triangleq just\ (n + 1, x).$$

[5] Here **op** denotes the operation symbol, whereas $n \geq 0$ denotes its arity.

The framework we have just described works fine for modelling operations with finite arity, but does not allow to handle operations with infinitary arity. This is witnessed, for instance, by imperative calculi with global stores, where it is natural to have operations of the form $\mathbf{get}_\ell(x.k)$ with the following intended semantics: $\mathbf{get}_\ell(x.k)$ reads the content of the location ℓ, say it is a value v, and continue as $k[v/x]$. In order to take such operations into account, we follow [58] and work with generalised operations.

A *generalised operation* (operation, for short) on a set X is a function $\omega : P \times X^I \to X$. The set P is called the *parameter set* of the operation, whereas the (index) set I is called the *arity* of the operation. A generalised operation $\omega : P \times X^I \to X$ thus takes as arguments a parameter p (such as a location name) and a map $\kappa : I \to X$ giving for each index $i \in I$ the argument $\kappa(i)$ to pass to ω. Syntactically, generalised operations are given via a signature Σ consisting of a set of elements of the form $\mathbf{op} : P \rightsquigarrow I$ (the latter being nothing but a notation denoting that the operation symbols $\mathbf{op}$ has parameter set P and index set I). Semantically, an interpretation of an operation symbol $\mathbf{op} : P \rightsquigarrow I$ on a monad $\mathbb{T}$ associates to any set X a map $[\![\mathbf{op}]\!]_X : P \times (TX)^I \to TX$ such that that for any $f : X \to TY$, $p \in P$, and $\kappa : I \to TX$:

$$f^\dagger([\![\mathbf{op}]\!]_X(p, \kappa)) = [\![\mathbf{op}]\!]_Y(p, f^\dagger \circ \kappa).$$

If $\mathbb{T}$ comes with an interpretation for operation symbols in Σ, we say that $\mathbb{T}$ is *Σ-algebraic*.

It is easy to see by taking the one-element set $1 = \{*\}$ as parameter set and a finite set as arity set, generalised operations subsume finitary operations. For simplicity, we use the notation $\mathbf{op} : n$ in place of $\mathbf{op} : 1 \rightsquigarrow n$, and write $\mathbf{op}(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ in place of $\mathbf{op}(*, n \mapsto \mathcal{X}_n)$.

Example 10. For the global state monad we consider the signature $\Sigma_{\mathbb{G}} \triangleq \{\mathbf{set}_\ell : \mathcal{V} \rightsquigarrow 1, \mathbf{get}_\ell : 1 \rightsquigarrow \mathcal{V} \mid \ell \in \mathcal{L}\}$. From a computational perspective, such operations are used to build programs of the form $\mathbf{set}_\ell(v, e)$ and $\mathbf{get}_\ell(x.e)$. The former stores the value v in the location ℓ and continues as e, whereas the latter reads the content of the location ℓ, say it is v, and continue as $e[v/x]$. Here e is used as the description of a function κ_e from values to terms defined by $\kappa_e(v) \triangleq e[v/x]$. The interpretation of the new operations on $\mathbb{G}$ is standard:

$$[\![\mathbf{set}_\ell]\!](v, \alpha)(\sigma) = \alpha(\sigma[\ell := v]), \qquad [\![\mathbf{get}_\ell]\!](\kappa)(\sigma) = \kappa(\sigma(\ell))(\sigma).$$

Straightforward calculations show that indeed $\mathbf{set}_\ell$ and $\mathbf{get}_\ell$ are algebraic operations on $\mathbb{G}$. Moreover, such operations can be easily extended to the partial global state monad $\mathbb{M} \otimes \mathbb{G}$ as well as to the probabilistic (partial) global store monad $\mathbb{DM} \otimes \mathbb{G}$. These extensions share a common pattern, which is nothing but an instance of the tensor of effects. In fact, given a $\Sigma_{\mathbb{T}}$-algebraic monad $\mathbb{T}$ we can define the signature $\Sigma_{\mathbb{T}\otimes\mathbb{G}}$ as $\Sigma_{\mathbb{T}} \cup \Sigma_{\mathbb{G}}$, and observe that the $\mathbb{T} \otimes \mathbb{G}$ is $\Sigma_{\mathbb{T}\otimes\mathbb{G}}$-algebraic. We refer the reader to [28] for details. Here we simply notice that we can define the interpretation $[\![\mathbf{op}]\!]^{\mathbb{T}\otimes\mathbb{G}}$ of $\mathbf{op} : P \rightsquigarrow \mathcal{V}$ on $\mathbb{T} \otimes \mathbb{G}$ as $[\![\mathbf{op}]\!]^{\mathbb{T}\otimes\mathbb{G}}_X(p, \kappa)(\sigma) \triangleq [\![\mathbf{op}]\!]^{\mathbb{T}}_{S\times X}(p, v \mapsto \kappa(v)(\sigma))$, where $[\![\mathbf{op}]\!]^{\mathbb{T}}$ is the interpretation of $\mathbf{op}$ on $\mathbb{T}$ (the interpretations of $\mathbf{set}_\ell$ and $\mathbf{get}_\ell$ are straightforward).

Monads and algebraic operations provide mathematical abstractions to structure and produce effectful computations. However, in order to give operational semantics to, e.g., probabilistic calculi [17] we need monads to account for infinitary computational behaviours. We thus look at *Σ-continuous monads*.

Definition 1. *A Σ-algebraic monad $\mathbb{T} = \langle T, \eta, -^\dagger \rangle$ is* Σ-continuous *(cf. [24]) if to any set X is associated an order $\sqsubseteq_X$ and an element $\bot_X \in TX$ such that $\langle TX, \sqsubseteq_X, \bot_X \rangle$ is an ω-cppo, and for all $(\mathbf{op} : P \rightsquigarrow I) \in \Sigma$, $f, f_n, g : X \to TY$, $\kappa, \kappa_n, \nu : I \to TX$, $\chi, \chi_n, y \in TX$, we have $f^\dagger(\bot) = \bot$ and:*

$$\kappa \sqsubseteq \nu \implies [\![\mathbf{op}]\!](p,\kappa) \sqsubseteq [\![\mathbf{op}]\!](p,\nu) \qquad [\![\mathbf{op}]\!](p, \bigsqcup_n \kappa_n) = \bigsqcup_n [\![\mathbf{op}]\!](p,\kappa_n)$$
$$f \sqsubseteq g \implies f^\dagger \sqsubseteq g^\dagger \qquad (\bigsqcup_n f_n)^\dagger = \bigsqcup_n f_n^\dagger$$
$$\chi \sqsubseteq y \implies f^\dagger(\chi) \sqsubseteq f^\dagger(y) \qquad f^\dagger(\bigsqcup_n \chi_n) = \bigsqcup_n f^\dagger(\chi_n).$$

When clear from the context, we will omit subscripts in $\bot_X$ and $\sqsubseteq_X$.

Example 11. The monads $\mathbb{M}$, $\mathbb{DM}$, $\mathbb{GM}$, and $\mathbb{C}$ are Σ-continuous. The order on MX and $\mathbb{C}$ is the flat ordering $\sqsubseteq$ defined by $\chi \sqsubseteq y \overset{\triangle}{\iff} \chi = \bot$ or $\chi = y$, whereas the order on DMX is defined by $\mu \sqsubseteq \nu \overset{\triangle}{\iff} \forall x \in X.\ \mu(just\ x) \leq \nu(just\ x)$. Finally, the order on GMX is defined pointwise from the flat ordering on $M(X \times S)$.

Having introduced the notion of a Σ-continuous monad, we can now define our vehicle calculus Λ_Σ and its monadic operational semantics.

4 A Computational Call-by-value Calculus with Algebraic Operations

In this section we define the calculus Λ_Σ. Λ_Σ is an untyped λ-calculus parametrised by a signature of operation symbols, and corresponds to the coarse-grain [44] version of the calculus studied in [15]. Formally, terms of Λ_Σ are defined by the following grammar, where x ranges over a countably infinite set of variables and $\mathbf{op}$ is a generalised operation symbol in Σ.

$$e \ ::= x \ \mid \ \lambda x.e \ \mid \ ee \ \mid \ \mathbf{op}(p, x.e).$$

A value is either a variable or a λ-abstraction. We denote by Λ the collection of terms and by $\mathcal{V}$ the collection of values of Λ_Σ. For an operation symbol $\mathbf{op} : P \rightsquigarrow I$, we assume that set I to be encoded by some subset of $\mathcal{V}$ (using e.g. Church's encoding). In particular, in a term of the form $\mathbf{op}(p, x.e)$, e acts as a function in the variable x that takes as input a value. Notice also how parameters $p \in P$ are part of the syntax. For simplicity, we ignore the specific subset of values used to encode elements of I, and simply write $\mathbf{op} : P \rightsquigarrow \mathcal{V}$ for operation symbols in Σ.

We adopt standard syntactical conventions as in [5] (notably the so-called variable convention). The notion of a free (resp. bound) variable is defined as

usual (notice that the variable x is bound in $\mathbf{op}(p, x.e)$). As it is customary, we identify terms up to renaming of bound variables and say that a term is closed if it has no free variables (and that it is open, otherwise). Finally, we write $f[e/x]$ for the capture-free substitution of the term e for all free occurrences of x in f. In particular, $\mathbf{op}(p, x'.f)[e/x]$ is defined as $\mathbf{op}(p, x'.f[e/x])$.

Before giving Λ_Σ call-by-value operational semantics, it is useful to remark a couple of points. First of all, testing terms according to their (possibly infinitary) normal forms obviously requires to work with open terms. Indeed, in order to inspect the *intensional* behaviour of a value $\lambda x.e$, one has to inspect the intensional behaviour of e, which is an open term. As a consequence, contrary to the usual practice, we give operational semantics to both *open* and *closed* terms. Actually, the very distinction between open and closed terms is not that meaningful in this context, and thus we simply speak of terms. Second, we notice that *values* constitute a syntactic category defined independently of the operational semantics of the calculus: values are just variables and λ-abstractions. However, giving operational semantics to arbitrary terms we are interested in richer collections of irreducible expressions, i.e. expressions that cannot be simplified any further. Such collections will be different accordingly to the operational semantics adopted. For instance, in a call-by-name setting it is natural to regard the term $x((\lambda x.x)v)$ as a terminal expression (being it a head normal form), whereas in a call-by-value setting $x((\lambda x.x)v)$ can be further simplified to xv, which in turn should be regarded as a terminal expression.

We now give Λ_Σ a monadic *call-by-value* operational semantics [15], postponing the definition of monadic *call-by-name* operational semantics to Section 6.4. Recall that a (call-by-value) evaluation context [22] is a term with a single hole $[-]$ defined by the following grammar, where $e \in \Lambda$ and $v \in \mathcal{V}$:

$$E \ ::= \ [-] \ \mid \ Ee \ \mid \ vE.$$

We write $E[e]$ for the term obtained by substituting the term e for the hole $[-]$ in E.

Following [38], we define a *stuck term* as a term of the form $E[xv]$. Intuitively, a stuck term is an expression whose evaluation is stuck. For instance, the term $e \triangleq y(\lambda x.x)$ is stuck. Obviously, e is not a value, but at the same time it cannot be simplified any further, as y is a variable, and not a λ-abstraction. Following this intuition, we define the collection $\mathcal{E}$ of *eager normal forms* (enfs hereafter) as the collection of values and stuck terms. We let letters $s, t, \ldots$ range over elements in $\mathcal{E}$.

Lemma 1. *Any term e is either a value v, or can be uniquely decomposed as either $E[vw]$ or $E[\mathbf{op}(p, x.f)]$.*

Operational semantics of Λ_Σ is defined with respect to a Σ-continuous monad $\mathbb{T} = \langle T, \eta, -^\dagger \rangle$ relying on Lemma 1. More precisely, we define a *call-by-value* evaluation function $[\![-]\!]$ mapping each term to an element in $T\mathcal{E}$. For instance, evaluating a probabilistic term e we obtain a distribution over eager normal forms (plus bottom), the latter being either values (meaning that the evaluation

of e terminates) or stuck terms (meaning that the evaluation of e went stuck at some point).

Definition 2. *Define the* $\mathbb{N}$*-indexed family of maps* $[\![-]\!]_n : \Lambda \to T\mathcal{E}$ *as follows:*

$$\begin{aligned}
[\![e]\!]_0 &\triangleq \bot,\\
[\![v]\!]_{n+1} &\triangleq \eta(v),\\
[\![E[xv]]\!]_{n+1} &\triangleq \eta(E[xv]),\\
[\![E[(\lambda x.e)v]]\!]_{n+1} &\triangleq [\![E[e[v/x]]]\!]_n,\\
[\![E[\mathbf{op}(p, x.e)]]\!]_{n+1} &\triangleq [\![\mathbf{op}]\!]_{\mathcal{E}}(p, v \mapsto [\![E[e[v/x]]]\!]_n).
\end{aligned}$$

The monad $\mathbb{T}$ being Σ-continuous, we see that the sequence $([\![e]\!]_n)_n$ forms an ω-chain in $T\mathcal{E}$, so that we can define $[\![e]\!]$ as $\bigsqcup_n [\![e]\!]_n$. Moreover, exploiting Σ-continuity of $\mathbb{T}$ we see that $[\![-]\!]$ is continuous.

We compare the behaviour of terms of Λ_Σ relying on the notion of an *effectful eager normal form (bi)simulation*, the extension of eager normal form (bi)simulation [38] to calculi with algebraic effects. In order to account for effectful behaviours, we follow [15] and parametrise our notions of equivalence and refinement by *relators* [6,71].

5 Relators

The notion of a *relator* for a functor T (on Set) [71] (also called *lax extension* of T [6]) is a construction lifting a relation $\mathcal{R}$ between two sets X and Y to a relation $\Gamma\mathcal{R}$ between TX and TY. Besides their applications in categorical topology [6] and coalgebra [71], relators have been recently used to study notions of applicative bisimulation [15], logic-based equivalence [67], and bisimulation-based distances [23] for λ-calculi extended with algebraic effects. Moreover, several forms of monadic lifting [32,25] resembling relators have been used to study abstract notions of logical relations [55,61].

Before defining relators formally, it is useful to recall some background notions on (binary) relations. The reader is referred to [26] for further details. We denote by Rel the category of sets and relations, and use the notation $\mathcal{R} : X \nrightarrow Y$ for a relation $\mathcal{R}$ between sets X and Y. Given relations $\mathcal{R} : X \nrightarrow Y$ and $\mathcal{S} : Y \nrightarrow Z$, we write $\mathcal{S} \circ \mathcal{R} : X \nrightarrow Z$ for their composition, and $\mathsf{I}_X : X \nrightarrow X$ for the identity relation on X. Finally, we recall that for all sets X, Y, the hom-set $\mathsf{Rel}(X, Y)$ has a complete lattice structure, meaning that we can define relations both inductively and coinductively.

Given a relation $\mathcal{R} : X \nrightarrow Y$, we denote by $\mathcal{R}^\circ : Y \nrightarrow X$ its dual (or opposite) relations and by $-_\circ : \mathsf{Set} \to \mathsf{Rel}$ the graph functor mapping each function $f : X \to Y$ to its graph $f_\circ : X \nrightarrow Y$. The functor $-_\circ$ being faithful, we will often write $f : X \to Y$ in place of $f_\circ : X \nrightarrow Y$. It is useful to keep in mind the pointwise reading of relations of the form $g^\circ \circ \mathcal{S} \circ f$, for a relation $\mathcal{S} : Z \nrightarrow W$ and functions $f : X \to Z$, $g : Y \to W$:

$$(g^\circ \circ \mathcal{S} \circ f)(x, y) = \mathcal{S}(f(x), g(y)).$$

Given $\mathcal{R} : X \nrightarrow Y$, we can thus express a generalised monotonicity condition in a pointfree fashion using the inclusion $\mathcal{R} \subseteq g^\circ \circ \mathcal{S} \circ f$. Finally, since we are interested in preorder and equivalence relations, we recall that a relation $\mathcal{R} : X \nrightarrow X$ is reflexive if $\mathsf{I}_X \subseteq \mathcal{R}$, transitive if $\mathcal{R} \circ \mathcal{R} \subseteq \mathcal{R}$, and symmetric if $\mathcal{R} \subseteq \mathcal{R}^\circ$. We can now define relators formally.

Definition 3. *A* relator *for a functor T (on Set) is a set-indexed family of maps $(\mathcal{R} : X \nrightarrow Y) \mapsto (\Gamma\mathcal{R} : TX \nrightarrow TY)$ satisfying conditions* (rel 1)-(rel 4)*. We say that Γ is* conversive *if it additionally satisfies condition* (rel 5)*.*

$$\mathsf{I}_{TX} \subseteq \Gamma(\mathsf{I}_X), \tag{rel 1}$$

$$\Gamma\mathcal{S} \circ \Gamma\mathcal{R} \subseteq \Gamma(\mathcal{S} \circ \mathcal{R}), \tag{rel 2}$$

$$Tf \subseteq \Gamma f, \quad (Tf)^\circ \subseteq \Gamma f^\circ, \tag{rel 3}$$

$$\mathcal{R} \subseteq \mathcal{S} \implies \Gamma\mathcal{R} \subseteq \Gamma\mathcal{S}, \tag{rel 4}$$

$$\Gamma(\mathcal{R}^\circ) = (\Gamma\mathcal{R})^\circ. \tag{rel 5}$$

Conditions (rel 1), (rel 2), and (rel 4) are rather standard[6]. As we will see, condition (rel 4) makes the defining functional of (bi)simulation relations monotone, whereas conditions (rel 1) and (rel 2) make notions of (bi)similarity reflexive and transitive. Similarly, condition (rel 5) makes notions of bisimilarity symmetric. Condition (rel 3), which actually consists of two conditions, states that relators behave as expected when acting on (graphs of) functions. In [43,15] a kernel preservation condition is required in place of (rel 3). Such a condition is also known as *stability* in [27]. Stability requires the equality $\Gamma(g^\circ \circ \mathcal{R} \circ f) = (Tg)^\circ \circ \Gamma\mathcal{R} \circ Tf$ to hold. It is easy to see that a relator always satisfies stability (see Corollary III.1.4.4 in [26]).

Relators provide a powerful abstraction of notions of 'relation lifting', as witnessed by the numerous examples of relators we are going to discuss. However, before discussing such examples, we introduce the notion of a *relator for a monad* or *lax extension of a monad*. In fact, since we modelled computational effects as monads, it seems natural to define the notion of a relator for a *monad* (and not just for a functor).

Definition 4. *Let $\mathbb{T} = \langle T, \eta, -^\dagger \rangle$ be a monad, and Γ be a relator for T. We say that Γ is a relator for $\mathbb{T}$ if it satisfies the following conditions:*

$$\mathcal{R} \subseteq \eta_Y^\circ \circ \Gamma\mathcal{R} \circ \eta_X, \tag{rel 7}$$

$$\mathcal{R} \subseteq g^\circ \circ \Gamma\mathcal{S} \circ f \implies \Gamma\mathcal{R} \subseteq (g^\dagger)^\circ \circ \Gamma\mathcal{S} \circ f^\dagger. \tag{rel 8}$$

Finally, we observe that the collection of relators is closed under specific operations (see [43]).

Proposition 2. *Let T, U be functors, and let UT denote their composition. Moreover, let Γ, Δ be relators for T and U, respectively, and $\{\Gamma_i\}_{i \in I}$ be a family of relators for T. Then:*

[6] Notice that since $\mathsf{I} = (1)_\circ$ we can derive condition (rel 1) from condition (rel 3).

1. *The map $\Delta\Gamma$ defined by $\Delta\Gamma\mathcal{R} \triangleq \Delta(\Gamma\mathcal{R})$ is a relator for UT.*
2. *The maps $\bigwedge_{i\in I}\Gamma_i$ and Γ° defined by $(\bigwedge_{i\in I}\Gamma_i)\mathcal{R} \triangleq \bigcap_{i\in I}\Gamma_i\mathcal{R}$ and $\Gamma^\circ\mathcal{R} \triangleq (\Gamma\mathcal{R}^\circ)^\circ$, respectively, are relators for T.*
3. *Additionally, if Γ is a relator for a monad $\mathbb{T}$, then so are $\bigwedge_{i\in I}\Gamma_i$ and Γ°.*

Example 12. For the partiality monad $\mathbb{M}$ we define the set-indexed family of maps $\hat{\mathbb{M}} : \mathsf{Rel}(X,Y) \to \mathsf{Rel}(MX, MY)$ as:

$$\chi\ \hat{\mathbb{M}}\mathcal{R}\ y \overset{\triangle}{\iff} (\chi = \bot) \vee (\exists x \in X.\ \exists y \in Y.\ \chi = \mathit{just}\ x \wedge y = \mathit{just}\ y \wedge x\ \mathcal{R}\ y).$$

The mapping $\hat{\mathbb{M}}$ describes the structure of the usual *simulation* clause for partial computations, whereas $\mathbb{M}^\circ$ describes the corresponding *co-simulation* clause. It is easy to see that $\hat{\mathbb{M}}$ is a relator for $\mathbb{M}$. By Proposition 2, the map $\hat{\mathbb{M}} \wedge \hat{\mathbb{M}}^\circ$ is a conversive relator for $\mathbb{M}$. It is immediate to see that the latter relator describes the structure of the usual *bisimulation* clause for partial computations.

Example 13. For the distribution monad we define the relator $\hat{\mathbb{D}}$ relying on the notion of a *coupling* and results from optimal transport [72]. Recall that a *coupling* for $\mu \in D(X)$ and $\nu \in D(Y)$ a is a joint distribution $\omega \in D(X \times Y)$ such that: $\mu = \sum_{y\in Y}\omega(-, y)$ and $\nu = \sum_{x\in X}\omega(x, -)$. We denote the set of couplings of μ and ν by $\Omega(\mu,\nu)$. Define the (set-indexed) map $\hat{\mathbb{D}} : \mathsf{Rel}(X,Y) \to \mathsf{Rel}(DX, DY)$ as follows:

$$\mu\ \hat{\mathbb{D}}\mathcal{R}\ \nu \overset{\triangle}{\iff} (\exists\omega \in \Omega(\mu,\nu).\ \forall x, y.\ \omega(x,y) > 0 \implies x\ \mathcal{R}\ y).$$

We can show that $\hat{\mathbb{D}}$ is a relator for $\mathbb{D}$ relying on *Strassen's Theorem* [69], which shows that $\hat{\mathbb{D}}$ can be characterised universally (i.e. using an universal quantification).

Theorem 1 (Strassen's Theorem [69]). *For all $\mu \in DX$, $\nu \in DY$, and $\mathcal{R} : X \nrightarrow Y$, we have: $\mu\ \hat{\mathbb{D}}\mathcal{R}\ \nu \iff \forall \mathcal{X} \subseteq X.\ \mu(\mathcal{X}) \leq \nu(\mathcal{R}[\mathcal{X}])$.*

As a corollary of Theorem 1, we see that $\hat{\mathbb{D}}$ describes the defining clause of Larsen-Skou bisimulation for Markov chains (based on full distributions) [34]. Finally, we observe that $\mathbb{D}\hat{\mathbb{M}} \triangleq \hat{\mathbb{D}}\hat{\mathbb{M}}$ is a relator for $\mathbb{D}\mathbb{M}$.

Example 14. For relations $\mathcal{R} : X \nrightarrow Y$, $\mathcal{S} : X' \nrightarrow Y'$, let $\mathcal{R}\times\mathcal{S} : X\times X' \nrightarrow Y\times Y'$ be defined as $(\mathcal{R}\times\mathcal{S})((x,x'),(y,y')) \overset{\triangle}{\iff} \mathcal{R}(x,y)\wedge\mathcal{S}(x',y')$. We define the relator $\hat{\mathbb{C}} : \mathsf{Rel}(X,Y) \to \mathsf{Rel}(CX, CY)$ for the cost monad $\mathbb{C}$ as $\hat{\mathbb{C}}\mathcal{R} \triangleq \hat{\mathbb{M}}(\geq \times \mathcal{R})$, where $\geq$ denotes the opposite of the natural ordering on $\mathbb{N}$. It is straightforward to see that $\hat{\mathbb{C}}$ is indeed a relator for $\mathbb{C}$. The use of the opposite of the natural order in the definition of $\hat{\mathbb{C}}$ captures the idea that we use $\hat{\mathbb{C}}$ to measure complexity. Notice that $\hat{\mathbb{C}}$ describes Sands' simulation clause for program improvement [62].

Example 15. For the global state monad $\mathbb{G}$ we define the map $\hat{\mathbb{G}} : \mathsf{Rel}(X,Y) \to \mathsf{Rel}(GX, GY)$ as $\alpha\ \hat{\mathbb{G}}\mathcal{R}\ \beta \overset{\triangle}{\iff} \forall\sigma \in S.\ \alpha(\sigma)\ (\mathsf{I}_S \times \mathcal{R})\ \beta(\sigma)$. It is straightforward to see that $\hat{\mathbb{G}}$ is a relator for $\mathbb{G}$.

It is not hard to see that we can extend $\hat{\mathbb{G}}$ to relators for $\mathbb{M} \otimes \mathbb{G}$, $\mathbb{DM} \otimes \mathbb{G}$, and $\mathbb{C} \otimes \mathbb{G}$. In fact, Proposition 1 extends to relators.

Proposition 3. *Given a monad $\mathbb{T} = \langle T, \mathrm{T}, -^{\mathrm{T}} \rangle$ and a relator $\hat{\mathbb{T}}$ for $\mathbb{T}$, define the sum $\mathbb{T}\hat{\mathbb{M}}$ of $\hat{\mathbb{T}}$ and $\hat{\mathbb{M}}$ as $\hat{\mathbb{T}}\hat{\mathbb{M}}$. Additionally, define the tensor $\widehat{\mathbb{T} \otimes \mathbb{G}}$ of $\hat{\mathbb{T}}$ and $\hat{\mathbb{G}}$ by $\alpha \, \widehat{(\mathbb{T} \otimes \mathbb{G})} \mathcal{R} \, \beta$ if an only if $\forall \sigma.\ \alpha(\sigma) \, \hat{\mathbb{T}}(\mathsf{I}_S \times \mathcal{R}) \, \beta(\sigma)$. Then $\mathbb{T}\hat{\mathbb{M}}$ is a relator for $\mathbb{TM}$, and $\widehat{(\mathbb{T} \otimes \mathbb{G})}$ is a relator for $\mathbb{T} \otimes \mathbb{G}$.*

Finally, we require relators to properly interact with the Σ-continuous structure of monads.

Definition 5. *Let $\mathbb{T} = \langle T, \eta, -^{\dagger} \rangle$ be a Σ-continuous monad and Γ be relator for $\mathbb{T}$. We say that Γ is* Σ-continuous *if it satisfies the following clauses — called the* inductive conditions *— for any ω-chain $(\chi_n)_n$ in TX, element $y \in TY$, elements $\chi, \chi' \in TX$, and relation $\mathcal{R} : X \nrightarrow Y$.*

$$\bot \, \Gamma\mathcal{R} \, y, \quad \chi \sqsubseteq \chi', \chi' \, \Gamma\mathcal{R} \, y \implies \chi \, \Gamma\mathcal{R} \, y, \quad \forall n.\ \chi_n \, \Gamma\mathcal{R} \, y \implies \bigsqcup_n \chi_n \, \Gamma\mathcal{R} \, y.$$

The relators $\hat{\mathbb{M}}$, $\hat{\mathbb{DM}}$, $\hat{\mathbb{C}}$, $\widehat{\mathbb{M} \otimes \mathbb{G}}$, $\widehat{\mathbb{DM} \otimes \mathbb{G}}$, $\widehat{\mathbb{C} \otimes \mathbb{G}}$ are all Σ-continuous. The reader might have noticed that we have not imposed any condition on how relators should interact with algebraic operations. Nonetheless, it would be quite natural to require a relator Γ to satisfy condition (rel 9) below, for all operation symbol $\mathbf{op} : P \rightsquigarrow I \in \Sigma$, maps $\kappa, \nu : I \to TX$, parameter $p \in P$, and relation $\mathcal{R}$.

$$\forall i \in I.\ \kappa(i) \, \Gamma\mathcal{R} \, \nu(i) \implies [\![\mathbf{op}]\!](p, \kappa) \, \Gamma\mathcal{R} \, [\![\mathbf{op}]\!](p, \nu) \qquad \text{(rel 9)}$$

Remarkably, if $\mathbb{T}$ is Σ-algebraic, then any relator for $\mathbb{T}$ satisfies (rel 9) (cf. [15]).

Proposition 4. *Let $\mathbb{T} = \langle T, \eta, -^{\dagger} \rangle$ be a Σ-algebraic monad, and let Γ be a relator for $\mathbb{T}$. Then Γ satisfies condition (rel 9).*

Having defined relators and their basic properties, we now introduce the notion of an effectful eager normal form (bi)simulation.

6 Effecful Eager Normal Form (Bi)simulation

In this section we tacitly assume a Σ-continuous monad $\mathbb{T} = \langle T, \eta, -^{\dagger} \rangle$ and a Σ-continuous relator Γ for it be fixed. Σ-continuity of Γ is not required for defining effectful eager normal form (bi)simulation, but it is crucial to prove that the induced notion of similarity and bisimilarity are precongruence and congruence relations, respectively.

Working with effectful calculi, it is important to distinguish between relations over *terms* and relations over *eager normal forms*. For that reason we we will work with pairs of relations of the form $(\mathcal{R}_\Lambda : \Lambda \nrightarrow \Lambda, \mathcal{R}_\mathcal{E} : \mathcal{E} \nrightarrow \mathcal{E})$, which we call λ-term relations (or term relations, for short). We use letters $\mathcal{R}, \mathcal{S}, \ldots$ to denote term relations. The collection of λ-term relations (i.e. $\mathsf{Rel}(\Lambda, \Lambda) \times \mathsf{Rel}(\mathcal{E}, \mathcal{E})$) inherits a complete lattice structure from $\mathsf{Rel}(\Lambda, \Lambda)$ and $\mathsf{Rel}(\mathcal{E}, \mathcal{E})$ pointwise, hence allowing λ-term relations to be defined both inductively and coinductively. We use these properties to define our notion of effectful eager normal form similarity.

Definition 6. *A term relation* $\mathcal{R} = (\mathcal{R}_\Lambda : \Lambda \nrightarrow \Lambda, \mathcal{R}_\mathcal{E} : \mathcal{E} \nrightarrow \mathcal{E})$ *is an* effectful eager normal form simulation *with respect to* Γ *(hereafter enf-simulation, as* Γ *will be clear from the context) if the following conditions hold, where in condition* (enf 4) $z \notin FV(E) \cup FV(E')$.

$$e\,\mathcal{R}_\Lambda\,f \implies [\![e]\!]\,\Gamma\mathcal{R}_\mathcal{E}\,[\![f]\!], \tag{enf 1}$$

$$x\,\mathcal{R}_\mathcal{E}\,s \implies s = x, \tag{enf 2}$$

$$\lambda x.e\,\mathcal{R}_\mathcal{E}\,s \implies \exists f.\ s = \lambda x.f \wedge e\,\mathcal{R}_\Lambda\,f, \tag{enf 3}$$

$$E[xv]\,\mathcal{R}_\mathcal{E}\,s \implies \exists E', v'.\ s = E'[xv'] \wedge v\,\mathcal{R}_\mathcal{E}\,v' \wedge \exists z.\ E[z]\,\mathcal{R}_\Lambda\,E'[z]. \tag{enf 4}$$

We say that relation $\mathcal{R}$ respects enfs *if it satisfies conditions* (enf 2)-(enf 4).

Definition 6 is quite standard. Clause (enf 1) is morally the same clause on terms used to define effectful applicative similarity in [15]. Clauses (enf 2) and (enf 3) state that whenever two enfs are related by $\mathcal{R}_\mathcal{E}$, then they must have the same outermost syntactic structure, and their subterms must be pairwise related. For instance, if $\lambda x.e\,\mathcal{R}_\mathcal{E}\,s$ holds, then s must the a λ-abstraction, i.e. an expression of the form $\lambda x.f$, and e and f must be related by $\mathcal{R}_\Lambda$.

Clause (enf 4) is the most interesting one. It states that whenever $E[xv]\,\mathcal{R}_\mathcal{E}\,s$, then s must be a stuck term $E'[xv']$, for some evaluation context E' and value v'. Notice that $E[xv]$ and s must have the same 'stuck variable' x. Additionally, v and v' must be related by $\mathcal{R}_\mathcal{E}$, and E and E' must be properly related too. The idea is that to see whether E and E' are related, we replace the stuck expressions xv, xv' with a fresh variable z, and test $E[z]$ and $E'[z]$ (thus resuming the evaluation process). We require $E[z]\,\mathcal{R}_\mathcal{E}\,E'[z]$ to hold, for *some* fresh variable z. The choice of the variable does not really matter, provided it is fresh. In fact, as we will see, effectful eager normal form similarity $\preceq^{\mathsf{E}}$ is substitutive and reflexive. In particular, if $E[z] \preceq^{\mathsf{E}}_\mathcal{E} E'[z]$ holds, then $E[y] \preceq^{\mathsf{E}}_\mathcal{E} E'[y]$ holds as well, for any variable $y \notin FV(E) \cup FV(E')$.

Notice that Definition 6 does not involve any universal quantification. In particular, enfs are tested by inspecting their syntactic structure, thus making the definition of an enf-simulation somehow 'local': terms are tested in isolation and not via their interaction with the environment. This is a major difference with e.g. applicative (bi)simulation, where the environment interacts with λ-abstractions by passing them arbitrary (closed) values as arguments.

Definition 6 induces a functional $\mathcal{R} \mapsto [\mathcal{R}]$ on the complete lattice $\mathsf{Rel}(\Lambda, \Lambda) \times \mathsf{Rel}(\mathcal{E}, \mathcal{E})$, where $[\mathcal{R}] = ([\mathcal{R}]_\Lambda, [\mathcal{R}]_\mathcal{E})$ is defined as follows (here $\mathsf{I}_\mathcal{X}$ denotes the identity relation on variables, i.e. the set of pairs of the form (x, x)):

$$[\mathcal{R}]_\Lambda \triangleq \{(e, f) \mid [\![e]\!]\,\Gamma\mathcal{R}_\mathcal{E}\,[\![f]\!]\}$$

$$[\mathcal{R}]_\mathcal{E} \triangleq \mathsf{I}_\mathcal{X} \cup \{(\lambda x.e, \lambda x.f) \mid e\,\mathcal{R}_\Lambda\,f\},$$
$$\cup\, \{(E[xv], E'[xv']) \mid v\,\mathcal{R}_\mathcal{E}\,v' \wedge \exists z \notin FV(E) \cup FV(E').\ E[z]\,\mathcal{R}_\Lambda\,E'[z]\}.$$

It is easy to see that a term relation $\mathcal{R}$ is an enf-simulation if and only if $\mathcal{R} \subseteq [\mathcal{R}]$. Notice also that although $[\mathcal{R}]_\mathcal{E}$ always contains the identity relation on variables, $\mathcal{R}_\mathcal{E}$ does not have to: the empty relation $(\emptyset, \emptyset)$ is an enf-simulation.

Finally, since relators are monotone (condition (rel 4)), $\mathcal{R} \mapsto [\mathcal{R}]$ is monotone too. As a consequence, by Knaster-Tarski Theorem [70], it has a greatest fixed point which we call *effectful eager normal form similarity* with respect to Γ (hereafter enf-similarity) and denote by $\preceq^{\mathsf{E}} = (\preceq^{\mathsf{E}}_{\Lambda}, \preceq^{\mathsf{E}}_{\mathcal{E}})$. Enf-similarity is thus the largest enf-simulation with respect to Γ. Moreover, $\preceq^{\mathsf{E}}$ being defined coinductively, it comes with an associated coinduction proof principle stating that if a term relation $\mathcal{R}$ is an enf-simulation, then it is contained in $\preceq^{\mathsf{E}}$. Symbolically: $\mathcal{R} \subseteq [\mathcal{R}] \implies \mathcal{R} \subseteq \preceq^{\mathsf{E}}$.

Example 16. We use the coinduction proof principle to show that $\preceq^{\mathsf{E}}$ contains the β-rule, viz. $(\lambda x.e)v \preceq^{\mathsf{E}}_{\Lambda} e[v/x]$. For that, we simply observe that the term relation $(\{((\lambda x.e)v, e[v/x])\}, \mathsf{I}_{\mathcal{E}})$ is an enf-simulation. Indeed, $[\![(\lambda x.e)v]\!] = [\![e[v/x]]\!]$, so that by (rel 1) we have $[\![(\lambda x.e)v]\!]\, \Gamma \mathsf{I}_{\mathcal{E}}\, [\![e[v/x]]\!]$.

Finally, we define effectful eager normal form *bisimilarity*.

Definition 7. *A term relation $\mathcal{R}$ is an* effectful eager normal form bisimulation with respect to Γ *(enf-bisimulation, for short) if it is a* symmetric *enf-simulation.* Eager normal bisimilarity with respect to Γ *(enf-bisimilarity, for short) $\simeq^{\mathsf{E}}$ is the largest symmetric enf-simulation. In particular, enf-bisimilarity (with respect to Γ) coincides with enf-similarity with respect to $\Gamma \wedge \Gamma^{\circ}$.*

Example 17. We show that the probabilistic call-by-value fixed point combinators Y and Z of Example 2 are enf-bisimilar. In light of Proposition 5, this allows us to conclude that Y and Z are applicatively bisimilar, and thus contextually equivalent [15]. Let us consider the relator $\mathbb{D}\hat{\mathbb{M}}$ for probabilistic partial computations. We show $Y \simeq^{\mathsf{E}}_{\Lambda} Z$ by coinduction, proving that the symmetric closure of the term relation $\mathcal{R} = (\mathcal{R}_{\Lambda}, \mathcal{R}_{\mathcal{E}})$ defined as follows is an enf-simulation:

$$\mathcal{R}_{\Lambda} \triangleq \{(Y, Z), (\Delta\Delta z, Zyz), (\Delta\Delta, y(\lambda z.\Delta\Delta z)\ \mathbf{or}\ y(\lambda z.Zyz))\} \cup \mathsf{I}_{\Lambda}$$

$$\mathcal{R}_{\mathcal{E}} \triangleq \{(y(\lambda z.\Delta\Delta z), y(\lambda z.Zyz)), (\lambda z.\Delta\Delta z, \lambda z.Zyz),$$
$$(\lambda y.\Delta\Delta, \lambda y.(y(\lambda z.\Delta\Delta z)\ \mathbf{or}\ y(\lambda z.Zyz))), (y(\lambda z.\Delta\Delta z)z, y(\lambda z.Zyz)z)\} \cup \mathsf{I}_{\mathcal{E}}.$$

The term relation $\mathcal{R}$ is obtained from the relation $\{(Y, Z)\}$ by progressively adding terms and enfs according to clauses (enf 1)-(enf 4) in Definition 6. Checking that $\mathcal{R}$ is an enf-simulation is straightforward. As an illustrative example, we prove that $\Delta\Delta z\, \mathcal{R}_{\Lambda}\, Zyz$ implies $[\![\Delta\Delta z]\!]\, \mathbb{D}\hat{\mathbb{M}}(\mathcal{R}_{\mathcal{E}})\, [\![Zyz]\!]$. The latter amounts to show:

$$\big(1 \cdot just\ y(\lambda z.\Delta\Delta z)z\big)\ \mathbb{D}\hat{\mathbb{M}}(\mathcal{R}_{\mathcal{E}})\ \big(\frac{1}{2} \cdot just\ y(\lambda z.\Delta\Delta z)z + \frac{1}{2} \cdot just\ y(\lambda z.Zyz)z\big),$$

where, as usual, we write distributions as weighted formal sums. To prove the latter, it is sufficient to find a suitable coupling of $[\![\Delta\Delta z]\!]$ and $[\![Zyz]\!]$. Define the distribution $\omega \in D(M\mathcal{E} \times M\mathcal{E})$ as follows:

$$\omega(just\ y(\lambda z.\Delta\Delta z)z, just\ y(\lambda z.\Delta\Delta z)z) = \frac{1}{2},$$
$$\omega(just\ y(\lambda z.\Delta\Delta z)z, just\ y(\lambda z.Zyz)z) = \frac{1}{2},$$

and assigning zero to all other pairs in $M\mathcal{E} \times M\mathcal{E}$. Obviously ω is a coupling of $[\![\Delta\Delta z]\!]$ and $[\![Zyz]\!]$. Additionally, we see that $\omega(\chi, y)$ implies $\chi\, \hat{\mathbb{M}}\mathcal{R}_\varepsilon\, y$, since both $y(\lambda z.\Delta\Delta z)z\, \mathcal{R}_\varepsilon\, y(\lambda z.\Delta\Delta z)z$, and $y(\lambda z.\Delta\Delta z)z\, \mathcal{R}_\varepsilon\, y(\lambda z.Zyz)z$ hold.

As already discussed in Example 2, the operational equivalence between Y and Z is an example of an equivalence that cannot be readily established using standard operational methods — such as CIU equivalence or applicative bisimilarity — but whose proof is straightforward using enf-bisimilarity. Additionally, Theorem 3 will allow us to reduce the size of $\mathcal{R}$, thus minimising the task of checking that our relation is indeed an enf-bisimulation. To the best of the authors' knowledge, the probabilistic instance of enf-(bi)similarity is the first example of a *probabilistic eager normal form (bi)similarity* in the literature.

6.1 Congruence and Precongruence Theorems

In order for $\preceq^{\mathsf{E}}$ and $\simeq^{\mathsf{E}}$ to qualify as good notions of program refinement and equivalence, respectively, they have to allow for compositional reasoning. Roughly speaking, a term relation $\mathcal{R}$ is compositional if the validity of the relationship $\mathcal{C}[e]\, \mathcal{R}\, \mathcal{C}[e']$ between compound terms $\mathcal{C}[e], \mathcal{C}[e']$ follows from the validity of the relationship $e\, \mathcal{R}\, e'$ between the subterms e, e'. Mathematically, the notion of compositionality is formalised throughout the notion of *compatibility*, which directly leads to the notions of a precongruence and congruence relation. In this section we prove that $\preceq^{\mathsf{E}}$ and $\simeq^{\mathsf{E}}$ are substitutive precongruence and congruence relations, that is preorder and equivalence relations closed under term constructors of Λ_Σ and substitution, respectively. To prove such results, we generalise Lassen's relational construction for the pure call-by-name λ-calculus [37]. Such a construction has been previously adapted to the *pure* call-by-value λ-calculus (and its extension with delimited and abortive control operators) in [9], whereas Lassen has proved compatibility of pure eager normal form bisimilarity via a CPS translation [38]. Both those proofs rely on syntactical properties of the calculus (mostly expressed using suitable small-step semantics), and thus seem to be hardly adaptable to effectful calculi. On the contrary, our proofs rely on the properties of relators, thereby making our results and techniques more modular and thus valid for a large class of effects.

We begin proving precongruence of enf-similarity. The central tool we use to prove the wished precongruence theorem is the so-called *(substitutive) context closure* [37] $\mathcal{R}^{\mathsf{sc}}$ of a term relation $\mathcal{R}$, which is inductively defined by the rules in Figure 1, where $\mathsf{x} \in \{\Lambda, \mathcal{E}\}$, $i \in \{1, 2\}$, and $z \notin FV(E) \cup FV(E')$.

We easily see that $\mathcal{R}^{\mathsf{sc}}$ is the smallest term relation that contains $\mathcal{R}$, it is closed under language constructors of Λ_Σ (a property known as *compatibility* [5]), and it is closed under the substitution operation (a property known as *substitutivity* [5]). As a consequence, we say that a term relation $\mathcal{R}$ is a *substitutive compatible* relation if $\mathcal{R}^{\mathsf{sc}} \subseteq \mathcal{R}$ (and thus $\mathcal{R} = \mathcal{R}^{\mathsf{sc}}$). If, additionally, $\mathcal{R}$ is a preorder (resp. equivalence) relation, then we say that $\mathcal{R}$ is a *substitutive precongruence* (resp. *substitutive congruence*) relation.

$$\frac{}{x\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ x}\ (\mathsf{sc\text{-}var}) \quad \frac{e\ \mathcal{R}_{\times}\ e'}{e\ \mathcal{R}_{\times}^{\mathsf{SC}}\ e'}\ (\mathsf{sc\text{-}}\downarrow) \quad \frac{s\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ s'}{s\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ s'}\ (\mathsf{sc\text{-}to\text{-}}\lambda)$$

$$\frac{e\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ f}{\lambda x.e\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ \lambda x.f}\ (\mathsf{sc\text{-}abs}) \quad \frac{e_i\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ e_i'}{e_1e_2\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ e_1'e_2'}\ (\mathsf{sc\text{-}app}) \quad \frac{e\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ f}{\mathbf{op}(p,x.e)\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ \mathbf{op}(p,x.f)}\ (\mathsf{sc\text{-}op})$$

$$\frac{v\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ v' \quad w\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ w'}{v[w/x]\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ v'[w'/x]}\ (\mathsf{sc\text{-}subst\text{-}val}) \quad \frac{e\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ e' \quad v\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ v'}{e[v/x]\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ e'[v'/x]}\ (\mathsf{sc\text{-}subst})$$

$$\frac{E[z]\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ E'[z] \quad v\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ v'}{E[xv]\ \mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ E[xv']}\ (\mathsf{sc\text{-}stuck}) \quad \frac{E[z]\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ E'[z] \quad e\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ e'}{E[e]\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ E'[e']}\ (\mathsf{sc\text{-}ectx})$$

Fig. 1. Compatible and substitutive closure construction.

We are now going to prove that if $\mathcal{R}$ is an enf-simulation, then so is $\mathcal{R}^{\mathsf{SC}}$. In particular, we will infer that $(\preceq^{\mathsf{E}})^{\mathsf{SC}}$ is a enf-simulation, and thus it is contained in $\preceq^{\mathsf{E}}$, by coinduction.

Lemma 2 (Main Lemma). *If $\mathcal{R}$ be an enf-simulation, then so is $\mathcal{R}^{\mathsf{SC}}$.*

Proof (sketch). The proof is long and non-trivial. Due to space constraints here we simply give some intuitions behind it. First, a routine proof by induction shows that since $\mathcal{R}$ respects enfs, then so does $\mathcal{R}^{\mathsf{SC}}$. Next, we wish to prove that $e\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ f$ implies $[\![e]\!]\ \Gamma\mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ [\![f]\!]$. Since Γ is inductive, the latter follows if for any $n \geq 0$, $e\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ f$ implies $[\![e]\!]_n\ \Gamma\mathcal{R}_{\mathcal{E}}^{\mathsf{SC}}\ [\![f]\!]$. We prove the latter implication by lexicographic induction on (1) the natural number n and (2) the derivation $e\ \mathcal{R}_{\Lambda}^{\mathsf{SC}}\ f$. The case for $n = 0$ is trivial (since Γ is inductive). The remaining cases are nontrivial, and are handled observing that $[\![E[e]]\!] = (s \mapsto [\![E[s]]\!])^{\dagger}[\![e]\!]$ and $[\![e[v/x]]\!]_n \sqsubseteq [\![-[v/x]]\!]_n^{\dagger}[\![e]\!]_n$. Both these identities allow us to apply condition (rel 8) to simplify proof obligations (usually relying on part (2) of the induction hypothesis as well). This scheme is iterated until we reach either an enf (in which case we are done by condition (rel 7)) or a pair of expressions on which we can apply part (1) of the induction hypothesis.

Theorem 2. *Enf-similarity (resp. bisimilarity) is a substitutive precongruence (resp. congruence) relation.*

Proof. We show that enf-similarity is a substitutive precongruence relation. By Lemma 2, it is sufficient to show that $\preceq^{\mathsf{E}}$ is a preorder. This follows by coinduction, since the term relations I and $\preceq^{\mathsf{E}} \circ \preceq^{\mathsf{E}}$ are enf-simulations (the proofs make use of conditions (rel 1) and (rel 2), as well as of substitutivity of $\preceq^{\mathsf{E}}$).

Finally, we show that enf-bisimilarity is a substitutive congruence relation. Obviously $\simeq^{\mathsf{E}}$ is an equivalence relation, so that it is sufficient to prove $(\simeq^{\mathsf{E}})^{\mathsf{SC}} \subseteq \simeq^{\mathsf{E}}$. That directly follows by coinduction relying on Lemma 2, provided that $(\simeq^{\mathsf{E}})^{\mathsf{SC}}$ is symmetric. An easy inspection of the rules in Figure 1 reveals that $\mathcal{R}^{\mathsf{SC}}$ is symmetric, whenever $\mathcal{R}$ is.

6.2 Soundness for Effectful Applicative (Bi)similarity

Theorem 2 qualifies enf-bisimilarity and enf-similarity as good candidate notions of program equivalence and refinement for Λ_Σ, at least from a structural perspective. However, we gave motivations for such notions looking at specific examples where effectful applicative (bi)similarity is ineffective. It is then natural to ask whether enf-(bi)similarity can be used as a proof technique for effectful applicative (bi)similarity.

Here we give a formal comparison between enf-(bi)similarity and effectful applicative (bi)similarity, as defined in [15]. First of all, we rephrase the notion of an effectful applicative (bi)simulation of [15] to our calculus Λ_Σ. For that, we use the following notational convention. Let $\Lambda_0, \mathcal{V}_0$ denote the collections of closed terms and closed values, respectively. We notice that if $e \in \Lambda_0$, then $[\![e]\!] \in T\mathcal{V}_0$. As a consequence, $[\![-]\!]$ induces a closed evaluation function $|-| : \Lambda_0 \to T\mathcal{V}_0$ characterised by the identity $[\![-]\!] \circ \iota = T\iota \circ |-|$, where $\iota : \mathcal{V}_0 \hookrightarrow \mathcal{E}$ is the obvious inclusion map. We can thus phrase the definition of effectful applicative similarity (with respect to a relator Γ) as follows.

Definition 8. *A term relation $\mathcal{R} = (\mathcal{R}_{\Lambda_0} : \Lambda_0 \nrightarrow \Lambda_0, \mathcal{R}_{\mathcal{V}_0} : \mathcal{V}_0 \nrightarrow \mathcal{V}_0)$ is an effectful applicative simulation with respect to Γ (applicative simulation, for short) if:*

$$e\, \mathcal{R}_{\Lambda_0}\, f \implies |e|\, \Gamma \mathcal{R}_{\mathcal{V}_0}\, |f|, \tag{app 1}$$

$$\lambda x.e\, \mathcal{R}_{\mathcal{V}_0}\, \lambda x.f \implies \forall v \in \mathcal{V}_0.\ e[v/x]\, \mathcal{R}_{\Lambda_0}\, f[v/x]. \tag{app 2}$$

As usual, we can define effectful applicative similarity with respect to Γ (applicative similarity, for short), denoted by $\preceq_0^{\mathsf{A}} = (\preceq_{\Lambda_0}^{\mathsf{A}}, \preceq_{\mathcal{V}_0}^{\mathsf{A}})$, coinductively as the largest applicative simulation. Its associated coinduction proof principle states that if a relation is an applicative simulation, then it is contained in applicative similarity. Finally, we extend $\preceq_0^{\mathsf{A}}$ to arbitrary terms by defining the relation $\preceq^{\mathsf{A}} = (\preceq_{\Lambda}^{\mathsf{A}}, \preceq_{\mathcal{V}}^{\mathsf{A}})$ as follows: let e, f, w, u be terms and values with free variables among $\bar{x} = x_1, \ldots, x_n$. We let $\bar{v}$ range over n-ary sequences of closed values $v_1, \ldots, v_n$. Define:

$$e \preceq_{\Lambda}^{\mathsf{A}} f \overset{\triangle}{\iff} \forall \bar{v}.\ e[\bar{v}/\bar{x}] \preceq_{\Lambda_0}^{\mathsf{A}} f[\bar{v}/\bar{x}], \qquad w \preceq_{\Lambda}^{\mathsf{A}} u \overset{\triangle}{\iff} \forall \bar{v}.\ w[\bar{v}/\bar{x}] \preceq_{\Lambda_0}^{\mathsf{A}} u[\bar{v}/\bar{x}].$$

The following result states that enf-similarity is a sound proof technique for applicative similarity.

Proposition 5. *Enf-similarity $\preceq^{\mathsf{E}}$ is included in applicative similarity $\preceq^{\mathsf{A}}$.*

Proof. Let $\preceq^{\mathsf{C}} = (\preceq_{\Lambda}^{\mathsf{C}}, \preceq_{\mathcal{V}}^{\mathsf{C}})$ denote enf-similarity restricted to closed terms and values. We first show that $\preceq^{\mathsf{C}}$ is an applicative simulation, from which follows, by coinduction, that it is included in $\preceq_0^{\mathsf{A}}$. It is easy to see that $\preceq^{\mathsf{C}}$ satisfies condition (app 2). In order to prove that it also satisfies condition (app 1), we have to show that for all $e, f \in \Lambda_\circ$, $e \preceq_{\Lambda}^{\mathsf{C}} f$ implies $|e|\, \Gamma \preceq_{\mathcal{V}}^{\mathsf{C}} |f|$. Since $e \preceq_{\Lambda}^{\mathsf{C}} f$ obviously implies $\iota(e) \preceq_{\Lambda}^{\mathsf{E}} \iota(f)$, by (enf 1) we infer $[\![\iota(e)]\!]\, \Gamma \preceq_{\mathcal{V}}^{\mathsf{E}} [\![\iota(f)]\!]$, and thus $T\iota|e|\, \Gamma \preceq_{\mathcal{V}}^{\mathsf{E}} T\iota|f|$.

By stability of Γ, the latter implies $|e|\ \Gamma(\iota^\circ \circ \preceq_\mathcal{E} \circ \iota)\ |f|$, and thus the wished thesis, since $\iota^\circ \circ \preceq_\mathcal{E} \circ \iota$ is nothing but $\preceq^{\mathsf{C}}_{\mathcal{V}}$. Finally, we show that for all terms e, f, if $e \preceq^{\mathsf{E}}_{\Lambda} f$, then $e \preceq^{\mathsf{A}}_{\Lambda} f$ (a similar result holds *mutatis mutandis* for values, so that we can conclude $\preceq^{\mathsf{E}} \subseteq \preceq^{\mathsf{A}}$). Indeed, suppose $FV(e) \cup FV(f) \subseteq \bar{x}$, then by substitutivity of $\preceq^{\mathsf{E}}$ we have that $e \preceq^{\mathsf{E}}_{\Lambda} f$ implies $e[\bar{v}/\bar{x}] \preceq^{\mathsf{E}}_{\Lambda} f[\bar{v}/\bar{x}]$, for all closed values $\bar{v}$ (notice that since we are substituting *closed* values, sequential and simultaneous substitution coincide). That essentially means $e[\bar{v}/\bar{x}] \preceq^{\mathsf{C}}_{\Lambda} f[\bar{v}/\bar{x}]$, and thus $e[\bar{v}/\bar{x}] \preceq^{\mathsf{A}}_{\Lambda_0} f[\bar{v}/\bar{x}]$. We thus conclude $e \preceq^{\mathsf{A}}_{\Lambda} f$.

Since in [15] it is shown that effectful applicative similarity (resp. bisimilarity) is contained in effectful contextual approximation (resp. equivalence), Proposition 5 gives the following result.

Corollary 1. *Enf-similarity and enf-bisimilarity are sound proof techniques for contextual approximation and equivalence, respectively.*

Although sound, enf-bisimilarity is *not* fully abstract for applicative bisimilarity. In fact, as already observed in [38], in the pure λ-calculus enf-bisimilarity is strictly finer than applicative bisimilarity (and thus strictly finer than contextual equivalence too). For instance, the terms xv and $(\lambda y.xv)(xv)$ are obviously applicatively bisimilar but not enf-bisimilar.

6.3 Eager Normal Form (Bi)simulation Up-to Context

The up-to context technique [60,37,64] is a refinement of the coinduction proof principle of enf-(bi)similarity that allows for handier proofs of equivalence and refinement between terms. When exhibiting a candidate enf-(bi)simulation relation $\mathcal{R}$, it is desirable for $\mathcal{R}$ to be as small as possible, so to minimise the task of verifying that $\mathcal{R}$ is indeed an enf-(bi)simulation.

The motivation behind such a technique can be easily seen looking at Example 17, where we showed the equivalence between the probabilistic fixed point combinators Y and Z working with relations containing several administrative pairs of terms. The presence of such pairs was forced by Definition 7, although they appear somehow unnecessary in order to convince that Y and Z exhibit the same operational behaviour.

Enf-(bi)simulation up-to context is a refinement of enf-(bi)simulation that allows to check that a relation $\mathcal{R}$ behaves as an enf-(bi)simulation relation up to its substitutive and compatible closure.

Definition 9. *A term relation $\mathcal{R} = (\mathcal{R}_\Lambda : \Lambda \nrightarrow \Lambda, \mathcal{R}_\mathcal{E} : \mathcal{E} \nrightarrow \mathcal{E})$ is an* effectful eager normal form simulation up-to context with respect to Γ *(enf-simulation up-to context, hereafter) if satisfies the following conditions, where in condition* (up-to 4) *$z \notin FV(E) \cup FV(E')$.*

$$e\,\mathcal{R}_\Lambda\, f \implies [\![e]\!]\ \Gamma\mathcal{R}^{\mathsf{SC}}_\mathcal{E}\ [\![f]\!], \tag{up-to 1}$$

$$x\,\mathcal{R}_\mathcal{E}\, s \implies s = x, \tag{up-to 2}$$

$$\lambda x.e\,\mathcal{R}_\mathcal{E}\, s \implies \exists f.\ s = \lambda x.f \wedge e\,\mathcal{R}^{\mathsf{SC}}_\Lambda\, f, \tag{up-to 3}$$

$$E[xv]\,\mathcal{R}_\mathcal{E}\, s \implies \exists E', v'.\ s = E'[xv'] \wedge v\,\mathcal{R}^{\mathsf{SC}}_\mathcal{E}\, v' \wedge \exists z.\ E[z]\,\mathcal{R}^{\mathsf{SC}}_\Lambda\, E'[z]. \tag{up-to 4}$$

In order for the up-to context technique to be sound, we need to show that every enf-simulation up-to context is contained in enf-similarity. This is a direct consequence of the following variation of Lemma 2.

Lemma 3. *If $\mathcal{R}$ is a enf-simulation up-to context, then $\mathcal{R}^{\mathsf{sc}}$ is a enf-simulation.*

Proof. The proof is structurally identical to the one of Lemma 2, where we simply observe that wherever we use the assumption that $\mathcal{R}$ is an enf-simulation, we can use the weaker assumption that $\mathcal{R}$ is an enf-simulation up-to context.

In particular, since by Lemma 2 we have that $\preceq^{\mathsf{E}} = (\preceq^{\mathsf{E}})^{\mathsf{sc}}$, we see that enf-similarity is an enf-simulation up-to context. Additionally, by Lemma 3 it is the largest such. Since the same result holds for enf-bisimilarity and enf-bisimilarity up-to context, we have the following theorem.

Theorem 3. *Enf-similarity is the largest enf-simulation up-to context, and enf-bisimilarity is the largest enf-bisimulation up-to context.*

Example 18. We apply Theorem 3 to simplify the proof of the equivalence between Y and Z given in Example 17. In fact, it is sufficient to show that the symmetric closure of term relation $\mathcal{R}$ defined below is an enf-bisimulation up-to context.

$$\mathcal{R}_\Lambda \triangleq \{(Y, Z), (\Delta\Delta z, Zyz), (\Delta\Delta, y(\lambda z.\Delta\Delta z) \mathbf{\ or\ } y(\lambda z.Zyz))\}, \qquad \mathcal{R}_\varepsilon \triangleq \mathsf{I}_\varepsilon.$$

Example 19. Recall the fixed point combinators with ticking operations Y and Z of Example 4. Let us consider the relator $\hat{\mathbb{C}}$. It is not hard to see that Y and Z are not enf-bisimilar (that is because the ticking operation is evaluated at different moments, so to speak). Nonetheless, once we pass them a variable x_0 as argument, we have $Zx_0 \preceq^{\mathsf{E}}_\Lambda Yx_0$. For, observe that the the term relation $\mathcal{R}$ defined below is an enf-simulation up-context.

$$\mathcal{R}_\Lambda \triangleq \{(Yx_0, Zx_0), (\mathbf{tick}(\Delta[x_0/y]\Delta[x_0/y]z), \mathbf{tick}(\Theta\Theta x_0 z))\}, \qquad \mathcal{R}_\varepsilon = \emptyset.$$

Intuitively, Y executes a tick first, and then proceeds iterating the evaluation of $\Delta[x_0/y]\Delta[x_0/y]$, the latter involving two tickings only. On the contrary, Z proceeds by recursively call itself, hence involving three tickings at any iteration, so to speak. Since $\preceq^{\mathsf{E}}$ is substitutive, for any value v we have $Zv \preceq^{\mathsf{E}} Yv$.

Theorem 3 makes enf-(bi)similarity an extremely powerful proof technique for program equivalence/refinement, especially because it is yet unknown whether there exist *sound* up-to context techniques for applicative (bi)similarity [35].

6.4 Weak Head Normal Form (Bi)simulation

So far we have focused on call-by-value calculi, since in presence of effects the call-by-value evaluation strategy seems the more natural one. Nonetheless, our framework can be easily adapted to deal with call-by-name calculi too. In this last

section we spend some words on *effectful weak head normal form (bi)similarity* (whnf-(bi)similarity, for short). The latter is nothing but the call-by-name counterpart of enf-(bi)similarity. The main difference between enf-(bi)similarity and whnf-(bi)similarity relies on the notion of an evaluation context (and thus of a stuck term). In fact, in a call-by-name setting, Λ_Σ evaluation contexts are expressions of the form $[-]e_1 \cdots e_n$, which are somehow simpler than their call-by-value counterparts. Such a simplicity is reflected in the definition of whnf-(bi)similarity, which allows to prove *mutatis mutandis* all results proved for enf-(bi)similarity (such results are, without much of a surprise, actually easier to prove).

We briefly expand on that. The collection of weak head normal forms (whnfs, for short) $\mathcal{W}$ is defined as the union of $\mathcal{V}$ and the collection of stuck terms, the latter being expressions of the form $xe_1 \cdots e_n$. The evaluation function of Definition 2 now maps terms to elements in $T\mathcal{W}$, and it is essentially obtained modifying Definition 2 defining $[\![E[xe]]\!]_{n+1} \triangleq \eta(E[xe])$ and $[\![E[(\lambda x.f)e]]\!]_{n+1} \triangleq [\![E[f[e/x]]]\!]_n$. The notion of a whnf-(bi)simulation (and thus the notions of whnf-(bi)similarity) is obtained modifying Definition 6 accordingly. In particular, clauses (enf 2) and (enf 4) are replaced by the following clause, where we use the notation $\mathcal{R} = (\mathcal{R}_\Lambda : \Lambda \nrightarrow \Lambda, \mathcal{R}_\mathcal{W} : \mathcal{W} \nrightarrow \mathcal{W})$ to denote a (call-by-name) λ-term relation.

$$xe_0 \cdots e_k \, \mathcal{R}_\mathcal{W} \, s \implies \exists f_0, \ldots, f_k.\ s = x f_0 \cdots f_k \wedge \forall i.\ e_i \, \mathcal{R}_\Lambda \, f_i.$$

A straightforward modifications of the rules in Figure 1 allows to prove an analogous of Lemma 2 for whnf-simulations, and thus to conclude (pre)congruence properties of whnf-(bi)similarity. Additionally, such results generalise to whnf-(bi)simulation up to-context, the latter being defined according to Definition 9, so that we have an analogous of Theorem 3 as well. The latter allows to infer the equivalence of the argument-switching fixed point combinators of Example 3, simply by noticing that the symmetric closure of the term relation $\mathcal{R} = (\{(P, Q), (Pyz, Qzy), (Pzy, Qyz)\}, \emptyset)$ is a whnf-bisimulation up-to context.

Finally, it is straightforward to observe that whnf-(bi)similarity is included in the call-by-name counterpart of effectful applicative (bi)similarity, but that the inclusion is strict. In fact, the (pure λ-calculus) terms xx and $x(\lambda y.xy)$ are applicatively bisimilar, but not whnf-bisimilar.

7 Related Work

Normal form (bi)similarity has been originally introduced for the call-by-name λ-calculus in [65], where it was called *open bisimilarity*. Open bisimilarity provides a coinductive characterisation of Lévy-Longo tree equivalence [45,53,42], and has been shown to coincide with the equivalence (notably weak bisimilarity) induced by Milner's encoding of the λ-calculus into the π-calculus [48].

In [37] normal form bisimilarity relations characterising both Böhm and Lévy-Longo tree equivalences have been studied by purely operational means, providing new congruence proofs of the aforementioned tree equivalences based on

suitable relational constructions. Such results have been extended to the call-by-value λ-calculus in [38], where the so-called *eager normal form bisimilarity* is introduced. The latter is shown to coincide with the Lévy-Longo tree equivalence induced by a suitable CPS translation [54], and thus to be a congruence relation. An elementary proof of congruence properties of eager normal form bisimilarity is given in [9], where Lassen's relational construction [37] is extended to the call-by-value λ-calculus, as well as its extensions with delimited and abortive control operators. Finally, following [65], eager normal form bisimilarity has been recently characterised as the equivalence induced by a suitable encoding of the (call-by-value) λ-calculus in the π-calculus [21].

Concerning effectful extensions of normal form bisimilarity, our work seems to be rather new. In fact, normal form bisimilarity has been studied for *deterministic* extensions of the λ-calculus with specific *non*-algebraic effects, notably control operators [9], as well as control and state [68] (where full abstraction of the obtained notion of normal form bisimilarity is proved). The only extension of normal form bisimilarity to an algebraic effect the authors are aware of, is given in [39], where normal form bisimilarity is studied for a *nondeterministic call-by-name* λ-calculus. However, we should mention that contrary to normal form bisimilarity, both nondeterministic [20] and probabilistic [41] extensions of Böhm tree equivalence have been investigated (although none of them employ, to the best of the authors' knowledge, coinductive techniques).

8 Conclusion

This paper shows that effectful normal form bisimulation is indeed a powerful methodology for program equivalence. Interestingly, the proof of congruence for normal form bisimilarity can be given just once, without the necessity of redoing it for every distinct notion of algebraic effect considered. This relies on the fact that the underlying monad and relator are Σ-continuous, something which has already been proved for many distinct notions of effects [15].

Topics for further work are plentiful. First of all, a natural question is whether the obtained notion of bisimilarity coincides with contextual equivalence. This is known *not* to hold in the deterministic case [37,38], but to hold in presence of control and state [68], which offer the environment the necessary discriminating power. Is there any (sufficient) condition on effects guaranteeing full abstraction of normal form bisimilarity? This is an intriguing question we are currently investigating. In fact, contrary to applicative bisimilarity (which is known to be unsound in presence of non-algebraic effects [33], such as local states), the syntactic nature of normal form bisimilarity seems to be well-suited for languages combining both algebraic and non-algebraic effects.

Another interesting topic for future research, is investigating whether normal form bisimilarity can be extended to languages having both algebraic operations and effect handlers [59,7].

References

1. Abramsky, S.: The lazy lambda calculus. In: Turner, D. (ed.) Research Topics in Functional Programming. pp. 65–117. Addison Wesley (1990)
2. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science. pp. 1–168. Clarendon Press (1994)
3. Abramsky, S., Ong, C.L.: Full abstraction in the lazy lambda calculus. Inf. Comput. **105**(2), 159–267 (1993)
4. Appel, A., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. **23**(5), 657–683 (2001)
5. Barendregt, H.: The lambda calculus: its syntax and semantics. Studies in logic and the foundations of mathematics, North-Holland (1984)
6. Barr, M.: Relational algebras. Lect. Notes Math. **137**, 39–55 (1970)
7. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebr. Meth. Program. **84**(1), 108–123 (2015)
8. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations: higher-order store. In: Proc. of PPDP 2009. pp. 301–312 (2009)
9. Biernacki, D., Lenglet, S., Polesiuk, P.: Proving soundness of extensional normal-form bisimilarities. Electr. Notes Theor. Comput. Sci. **336**, 41–56 (2018)
10. Bizjak, A., Birkedal, L.: Step-indexed logical relations for probability. In: Proc. of FOSSACS 2015. pp. 279–294 (2015)
11. Böhm, C.: Alcune proprietà delle forme $\beta\eta$-normali del λk-calcolo. Pubblicazioni dell'Istituto per le Applicazioni del Calcolo **696** (1968)
12. Burris, S., Sankappanavar, H.: A course in universal algebra. Graduate texts in mathematics, Springer-Verlag (1981)
13. Crubillé, R., Dal Lago, U.: On probabilistic applicative bisimulation and call-by-value lambda-calculi. In: Proc. of ESOP 2014. pp. 209–228 (2014)
14. Culpepper, R., Cobb, A.: Contextual equivalence for probabilistic programs with continuous random variables and scoring. In: Proceedings of ESOP 2017. pp. 368–392 (2017)
15. Dal Lago, U., Gavazzo, F., Levy, P.: Effectful applicative bisimilarity: Monads, relators, and howe's method. In: Proc. of LICS 2017. pp. 1–12 (2017)
16. Dal Lago, U., Sangiorgi, D., Alberti, M.: On coinductive equivalences for higher-order probabilistic functional programs. In: Proc. of POPL 2014. pp. 297–308 (2014)
17. Dal Lago, U., Zorzi, M.: Probabilistic operational semantics for the lambda calculus. RAIRO - Theor. Inf. and Applic. **46**(3), 413–450 (2012)
18. Danos, V., Harmer, R.: Probabilistic game semantics. ACM Transactions on Computational Logic **3**(3), 359–382 (2002)
19. Davey, B., Priestley, H.: Introduction to lattices and order. Cambridge University Press (1990)
20. De Liguoro, U., Piperno, A.: Non deterministic extensions of untyped lambda-calculus. Inf. Comput. **122**(2), 149–177 (1995)
21. Durier, A., Hirschkoff, D., Sangiorgi, D.: Eager functions as processes. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018. pp. 364–373 (2018)
22. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2), 235–271 (1992)

23. Gavazzo, F.: Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 452–461 (2018)
24. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. J. ACM **24**(1), 68–95 (1977)
25. Goubault-Larrecq, J., Lasota, S., Nowak, D.: Logical relations for monadic types. Mathematical Structures in Computer Science **18**(6), 1169–1217 (2008)
26. Hofmann, D., Seal, G., Tholen, W. (eds.): Monoidal Topology. A Categorical Approach to Order, Metric, and Topology. No. 153 in Encyclopedia of Mathematics and its Applications, Cambridge University Press (2014)
27. Hughes, J., Jacobs, B.: Simulations in coalgebra. Theor. Comput. Sci. **327**(1-2), 71–108 (2004)
28. Hyland, M., Plotkin, G.D., Power, J.: Combining effects: Sum and tensor. Theor. Comput. Sci. **357**(1-3), 70–99 (2006)
29. Hyland, M., Power, J.: The category theoretic understanding of universal algebra: Lawvere theories and monads. Electr. Notes Theor. Comput. Sci. **172**, 437–458 (2007)
30. Johann, P., Simpson, A., Voigtländer, J.: A generic operational metatheory for algebraic effects. In: Proc. of LICS 2010. pp. 209–218. IEEE Computer Society (2010)
31. Jones, C.: Probabilistic non-determinism. Ph.D. thesis, University of Edinburgh, UK (1990)
32. Katsumata, S., Sato, T.: Preorders on monads and coalgebraic simulations. In: Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. pp. 145–160 (2013)
33. Koutavas, V., Levy, P.B., Sumii, E.: From applicative to environmental bisimulation. Electr. Notes Theor. Comput. Sci. **276**, 215–235 (2011)
34. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. In: Proceedings of POPL 1989. pp. 344–352 (1989)
35. Lassen, S.: Relational reasoning about contexts. In: Gordon, A.D., Pitts, A.M. (eds.) Higher Order Operational Techniques in Semantics, pp. 91–136 (1998)
36. Lassen, S.: Relational Reasoning about Functions and Nondeterminism. Ph.D. thesis, Dept. of Computer Science, University of Aarhus (May 1998)
37. Lassen, S.B.: Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. Electr. Notes Theor. Comput. Sci. **20**, 346–374 (1999)
38. Lassen, S.B.: Eager normal form bisimulation. In: Proceedings of LICS 2005. pp. 345–354 (2005)
39. Lassen, S.B.: Normal form simulation for mccarthy's amb. Electr. Notes Theor. Comput. Sci. **155**, 445–465 (2006)
40. Lawvere, W.F.: Functorial Semantics of Algebraic Theories. Ph.D. thesis (2004)
41. Leventis, T.: Probabilistic böhm trees and probabilistic separation. In: Proc. of LICS (2018), to appear.
42. Lévy, J.: An algebraic interpretation of the lambda beta - calculus and a labeled lambda - calculus. In: Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, Italy, March 25-27, 1975. pp. 147–165 (1975)
43. Levy, P.: Similarity quotients as final coalgebras. In: Proc. of FOSSACS 2011. LNCS, vol. 6604, pp. 27–41 (2011)

44. Levy, P., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Inf. Comput. **185**(2), 182–210 (2003)
45. Longo, G.: Set-theoretical models of lambda calculus: Theories, expansions, isomorphisms. Annals of Pure and Applied Logic **24**, 153–188 (1983)
46. MacLane, S.: Categories for the Working Mathematician. Springer-Verlag (1971)
47. Mason, I.A., Talcott, C.L.: Equivalence in functional languages with effects. J. Funct. Program. **1**(3), 287–327 (1991)
48. Milner, R.: Functions as processes. Mathematical Structures in Computer Science **2**(2), 119–141 (1992)
49. Moggi, E.: Computational lambda-calculus and monads. In: Proc. of LICS 1989. pp. 14–23. IEEE Computer Society (1989)
50. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
51. Morris, J.: Lambda Calculus Models of Programming Languages. Ph.D. thesis, MIT (1969)
52. Ong, C.L.: Non-determinism in a functional setting. In: Proc. of LICS 1993. pp. 275–286. IEEE Computer Society (1993)
53. Ong, C.: The Lazy Lambda Calculus: An Investigation Into the Foundations of Functional Programming. University of London. Imperial College of Science and Technology (1988)
54. Plotkin, G.: Call-by-name, call-by-value and the lambda-calculus. Theoretical Computer Science **1**(2), 125 – 159 (1975)
55. Plotkin, G.: Lambda-definability and logical relations (1973), technical Report SAI-RM-4, School of A.I., University of Edinburgh
56. Plotkin, G.D., Power, J.: Adequacy for algebraic effects. In: Proc. of FOSSACS 2001. pp. 1–24 (2001)
57. Plotkin, G.D., Power, J.: Notions of computation determine monads. In: Proc. of FOSSACS 2002. pp. 342–356 (2002)
58. Plotkin, G.D., Power, J.: Algebraic operations and generic effects. Applied Categorical Structures **11**(1), 69–94 (2003)
59. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Logical Methods in Computer Science **9**(4) (2013)
60. Pous, D., Sangiorgi, D.: Enhancements of the bisimulation proof method. In: Sangiorgi, D., Rutten, J. (eds.) Advanced Topics in Bisimulation and Coinduction. Cambridge University Press (2012)
61. Reynolds, J.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
62. Sands, D.: Improvement theory and its applications. In: Gordon, A.D., Pitts, A.M. (eds.) Higher Order Operational Techniques in Semantics, pp. 275–306. Publications of the Newton Institute, Cambridge University Press (1998)
63. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. ACM Trans. Program. Lang. Syst. **33**(1), 5:1–5:69 (2011)
64. Sangiorgi, D.: A theory of bisimulation for the pi-calculus. In: CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. pp. 127–142 (1993)
65. Sangiorgi, D.: The lazy lambda calculus in a concurrency scenario. Inf. Comput. **111**(1), 120–153 (1994)
66. Sangiorgi, D., Vignudelli, V.: Environmental bisimulations for probabilistic higher-order languages. In: Proceedings of POPL 2016. pp. 595–607 (2016)
67. Simpson, A., Voorneveld, N.: Behavioural equivalence via modalities for algebraic effects. In: Proc. of ESOP 2018. pp. 300–326 (2018)

68. Støvring, K., Lassen, S.B.: A complete, co-inductive syntactic theory of sequential control and state. In: Proceedings of POPL 2007. pp. 161–172 (2007)
69. Strassen, V.: The existence of probability measures with given marginals. Ann. Math. Statist. **36**(2), 423–439 (1965)
70. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2), 285–309 (1955)
71. Thijs, A.: Simulation and fixpoint semantics. Rijksuniversiteit Groningen (1996)
72. Villani, C.: Optimal Transport: Old and New. Grundlehren der mathematischen Wissenschaften, Springer Berlin Heidelberg (2008)
73. Wadler, P.: Monads for functional programming. In: Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992. pp. 233–264 (1992)