Effectful applicative similarity for call-by-name lambda calculi

Ugo Dal Lago, Francesco Gavazzo, Ryo Tanaka

Effectful applicative similarity for call-by-name lambda calculi

Ugo Dal Lago[a,b], Francesco Gavazzo[a,b], Ryo Tanaka[c]

[a] *Università di Bologna*
[b] *INRIA Sophia Antipolis*
[c] *The University of Tokyo*

Abstract

We introduce a notion of applicative similarity in which not terms but *monadic values* arising from the evaluation of effectful terms, can be compared. We prove this notion to be fully abstract whenever terms are evaluated in call-by-name order. This is the first full-abstraction result for such a generic, coinductive methodology for program equivalence.

Keywords: Call-by-name λ-calculus, applicative similarity, Howe's method, Algebraic effects

1. Introduction

Program equivalence and refinement are crucial notions in the theory of programming languages, and their study is at the heart of programming language semantics. When higher-order functions are available, programs have a non-trivial interactive behaviour, and giving a *satisfactory* and at the same time *handy* definition of program equivalence becomes complicated. The problem has been approached, in many different ways, e.g. by denotational semantics or by contextual equivalence. These two approaches have their drawbacks, the first one relying on the existence of a denotational model, the latter quantifying over all contexts, thus making the task of proving programs equivalent quite hard. Handier methodologies for proving programs equivalent have been introduced along the years based on logical relations and applicative bisimilarity. Logical relations were originally devised for typed, normalising languages, but later generalised to more expressive formalisms, e.g., through step-indexing (Appel and McAllester, 2001) and biorthogonality (Benton et al., 2009). Starting from Abramsky's pioneering work on

applicative bisimilarity (Abramsky, 1990), coinduction has also been proved to be a useful methodology for program equivalence, and has been applied to a variety of calculi and language features (Lassen, 1998; Dal Lago et al., 2014; Ong, 1993).

The just described scenario also holds when the underlying calculus is not pure, but effectful. There have been many attempts to study effectful λ-calculi (Plotkin and Power, 2001; Moggi, 1989) by way of denotational semantics (de'Liguoro and Piperno, 1995; Jones, 1990), logical relations (Bizjak and Birkedal, 2015), and applicative bisimilarity (Lassen, 1998; Dal Lago et al., 2014; Crubillé and Dal Lago, 2014). But while the denotational and logical relation semantics of effectful calculi have been studied in the abstract (Goubault-Larrecq et al., 2008; Johann et al., 2010), the same cannot be said about applicative bisimilarity and related coinductive techniques. There is a growing body of literature on applicative bisimilarity for calculi with, e.g., nondeterministic (Lassen, 1998), and probabilistic effects (Dal Lago et al., 2014), but each notion of an effect has been studied independently, often getting different results. Distinct proofs of congruence for applicative bisimilarity, even if done through a common methodology, namely the so-called Howe's method (Howe, 1996), do not at all have the same difficulty in each of the cases cited above.

The observations above naturally lead to some questions. Is there any way to factor out the common part of the congruence proof for applicative bisimilarity in effectful calculi? Where do the limits on the correctness of applicative bisimilarity lie, in presence of effects?

This paper is part of a longstanding research effort directed to giving answers to the questions above. The first two authors, together with Paul Blain Levy, have recently introduced a general notion of applicative bisimilarity for lambda-calculi based on monads and relators (Dal Lago et al., 2017). This provides, under mild conditions, a sound methodology for checking contextual equivalence of programs. The central idea is to take simulations as relations on terms and to use a *relator* to lift a (candidate) simulation to a relation on the monadic images of values. There is however little hope to prove a generic full-abstraction result in such a setting, although for certain notions of an effect, full abstraction is already known to hold (Abramsky, 1990; Crubillé and Dal Lago, 2014).

In this paper, we study a different notion of simulation, which puts in relation not terms but their semantics. This way, interaction between terms and the environment can be modeled by an ordinary, deterministic, transition

system and, with minimal side conditions, similarity can be proved to be not only a *sound*, but also to coincide with the contextual preorder. This, however, only holds when terms are call-by-*name* evaluated.

The paper is organised as follows. We first informally introduce the main ideas behind the present work by means of an extended example. We will then enter the technical developments of the work by first introducing a computational call-by-name λ-calculus Λ_Σ enriched with arbitrary *algebraic effects à la* Plotkin and Power (Plotkin and Power, 2001). The latter are defined by means of a signature of effect-triggering operation symbols which are interpreted as algebraic operations with respect to a monad.

In Section 4 we define an abstract notion of observation on top of which we define contextual and CIU approximation, as well as applicative similarity. Our first main result (Theorem 2), namely the equivalence between applicative similarity and CIU approximation, is proved in Section 5. Section 6, instead, is dedicated to our second main result (Corollary 1), namely full abstraction of applicative similarity (and thus CIU approximation) for contextual approximation. This result relies on a suitable CIU theorem (Theorem 3) whose proof is based on a variation of the so-called Howe's method (Howe, 1996).

1.1. Lifting Transition Systems

In this section we introduce the main ideas behind our approach by means of an example, namely a call-by-name *nondeterministic* λ-calculus (Ong, 1993; Lassen, 1998), which we call $\Lambda^{\sqcup}$. The syntax of $\Lambda^{\sqcup}$ extends the syntax of the ordinary λ-calculus (Barendregt, 1984) with a binary nondeterministic choice operator $\sqcup$, the intended semantics of an expression $e \sqcup f$ being to nondeterministically evaluate either to e or to f. We denote the collection of expressions and values (i.e. expressions of the form $\lambda x.e$) by Λ and $\mathcal{V}$, respectively, and let letters $e, f, \ldots$ and $v, w, \ldots$ range over elements of Λ and $\mathcal{V}$, respectively. As usual, a *program* is a closed expression, i.e. an expression without free variables. We denote by Λ_0 the collection of all programs.

As we are in a nondeterministic setting we assume we can observe whether a program *may converge*[1], meaning that the observable (operational) behaviour of a program e is whether there exists a value v such that e evaluates to v. For instance, the program $(\lambda x.x) \sqcup \Omega$, where Ω is the purely divergent

[1] Although other choices, such as *must* or *may and must* convergence, are possible.

expression $(\lambda x.xx)(\lambda x.xx)$, may converge to $\lambda x.x$. We write $e \Downarrow v$ if e may converge to the value v, and $e\Downarrow$ if there exists a value v such that $e \Downarrow v$.

The notion of *contextual* approximation captures the idea of observational refinement in any possible environment. In our setting, as we have already remarked, observations are modelled by the predicate $\Downarrow$, whereas the role of the environment is played by *contexts*[2]. Accordingly, we say that a program e contextually approximates a program f, written $e \preceq^{ctx} f$, if for any context C, $C[e]\Downarrow$ implies $C[f]\Downarrow$. We refer to the relation $\preceq^{ctx}$ as *contextual approximation*.

Although intuitively appealing, the relation $\preceq^{ctx}$ comes with a major drawback, namely its universal quantification over all possible contexts. Proving that a program e contextually approximates a program f requires to test the behaviour of e and f in any possible environment. For this reason, several proof techniques for $\preceq^{ctx}$ have been proposed in the last decades. Such techniques often come in the form of a relation $\mathcal{R}$ between programs such that $\mathcal{R} \subseteq \preceq^{ctx}$, meaning that $e\,\mathcal{R}\,f$ implies $e \preceq^{ctx} f$. We say that $\mathcal{R}$ is *sound* for $\preceq^{ctx}$ if $\mathcal{R} \subseteq \preceq^{ctx}$, and that it is *fully abstract* for $\preceq^{ctx}$ if $\mathcal{R} = \preceq^{ctx}$.

Starting with the seminal work by Abramsky (Abramsky, 1990), *applicative similarity* has been proved to be a powerful proof technique for contextual approximation, both for the pure λ-calculus (Abramsky, 1990) and for its nondeterministic and probabilistic extensions (Ong, 1993; Lassen, 1998; Dal Lago et al., 2014). The notion of applicative similarity for $\Lambda^{\sqcup}$ is rooted in the observation that the operational semantics of $\Lambda^{\sqcup}$ defines a (labelled) transition system $\mathbb{T} = \langle \Lambda_0, \mathcal{L}, \rightarrow \rangle$ over programs. Accordingly, the set of states[3] is defined as the collection of programs, whereas the set of labels (or actions) $\mathcal{L}$ is defined as $\Lambda_0 \cup \{\tau\}$, where τ is a special symbol denoting evaluation. Finally, the transition relation $\rightarrow$ is defined as follows:

1. $e \xrightarrow{\tau} v$ if $e \Downarrow v$.
2. $\lambda x.f \xrightarrow{e} f[e/x]$, for any $e \in \Lambda_0$.

[2]Intuitively, a context is an expression with a hole $[-]$. We write $C[e]$ for the expression obtained by filling in the hole in C with e.

[3]From a formal perspective, we should work with a bi-labelled transition system distinguishing between programs and values. However, since the goal of this section is to give an informal exposition of the main ideas behind this work, such a level of mathematical accuracy is unnecessary.

Notice that due to the presence of nondeterminism, the above transition system is clearly nondeterministic. In general, a term e may converge to more than one value.

The notion of an applicative simulation is defined as ordinary simulation on $\mathbb{T}$. Spelling out the definition, it is not hard to see that a relation $\mathcal{R} \subseteq \Lambda_0 \times \Lambda_0$ is an applicative simulation if and only if $e\,\mathcal{R}\,f$ implies that for any value $\lambda x.g$ such that $e \Downarrow \lambda x.g$, there exists a value $\lambda x.h$ such that $f \Downarrow \lambda x.h$ and for any term k we have $g[k/x]\,\mathcal{R}\,h[k/x]$. *Applicative similarity* $\preceq^{app}$ is then coinductively defined as the largest applicative simulation. Notice that applicative similarity has better mathematical properties than $\preceq^{ctx}$. In fact, $\preceq^{app}$ does not universally quantify over all contexts, but only on programs passed as arguments to lambda abstractions. Furthermore, $\preceq^{app}$ comes with an associated *coinduction proof principle*: in order to prove that a program f is applicatively similar to a program e, it is sufficient to exhibit an applicative simulation $\mathcal{R}$ relating e and f.

Remarkably, applicative similarity is included in contextual approximation, meaning that $\preceq^{app}$ is a sound proof technique for $\preceq^{ctx}$. However, it is not hard to realise that full abstraction fails. In fact, we can easily convince ourselves that the program $e = \lambda x.(f \sqcup g)$ contextually approximates $h = \lambda x.f \sqcup \lambda x.g$, although $e \not\preceq^{app} h$.

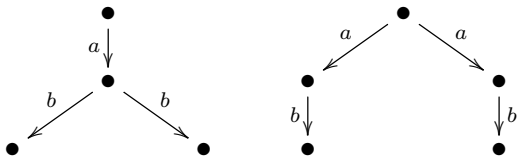
An informal argument supporting $e \preceq^{ctx} h$ is the following (a formal proof will be given later in full generality). As $\Lambda^{\sqcup}$ is a *call-by-name* calculus, a context C can only test a program k in *function position*. That is, during the evaluation of a program $C[k]$, k can only be evaluated in function position, i.e. in a situation of the form kc, where c may 'contain' other occurrences of k. Similarly, the result of the evaluation of kc can only be tested in function position. Iterating this observation, we see that testing a program k in a context C is equivalent to testing k against a finite sequence of inputs. This is the content of the so-called *CIU Theorem* (Mason and Talcott, 1991):

Theorem 1 (CIU, $\Lambda^{\sqcup}$-version)**.** *For all programs e, f, $e \preceq^{ctx} f$ holds if and only if for any finite sequence of programs $k_1, \ldots, k_n$, $ek_1 \cdots k_n \Downarrow$ implies $fk_1 \cdots k_n \Downarrow$.*

Notice that the CIU Theorem states that $\preceq^{ctx}$ is a form of *trace refinement*. Using Theorem 1 we can easily show $e \preceq^{ctx} h$. This comes with no surprise: contextual approximation being a form of trace refinement, it is not sensitive to 'branching behaviours', and thus by no means can tell e and h apart.

This is not the case for applicatively similarity which, instead, is sensitive to forms of branching. From an applicative simulation perspective, the difference between the expressions e and h as defined above is clear: e postpones the nondeterministic choice to the moment it receives an input, whereas h first makes the choice, and then waits to receive an input. Accordingly, it is sufficient to instantiate f as $x\Omega(\lambda x.x)$ and g as $x(\lambda x.x)\Omega$ to observe that $e \not\preceq^{app} h$. For suppose $\lambda x.(f \sqcup g) \preceq \lambda x.f \sqcup \lambda x.g$. In particular, there is a simulation $\mathcal{R}$ relating the two programs. By the very definition of applicative simulation, since $\lambda x.(f \sqcup g)$ evaluates to itself and $\lambda x.f \sqcup \lambda x.g$ evaluates to either $\lambda x.f$ or $\lambda x.g$, we must have $\lambda x.(f \sqcup g)\,\mathcal{R}\,\lambda x.f$ or $\lambda x.(f \sqcup g)\,\mathcal{R}\,\lambda x.g$. In the first case, since $\mathcal{R}$ is a simulation, we must have $(f[k/x] \sqcup g[k/x])\mathcal{R} f[k/x]$, for any program k. This, however, cannot be the case, as taking $k = \lambda xy.x$, we see that $f[k/x] \sqcup g[k/x]$ may converge to $\lambda x.x$, whereas $f[k/x]$ diverges. A similar argument shows that $\lambda x.(f \sqcup g)\,\mathcal{R}\,\lambda x.g$ cannot hold as well (take $k = \lambda xy.y$).

The reader might have recognised that e and h are nothing more than the encoding in $\Lambda_\sqcup$ of the labelled transition systems below, which are the standard example in concurrency theory showing that bisimilarity is strictly finer than trace equivalence.



Summing up, we have observed that in a nondeterministic setting, applicative similarity is sensitive to branching, whereas contextual approximation is not. This is because, in full generality, contextual approximation is a form of trace approximation, and thus intrinsically *deterministic*. This last observation suggests an easy way to make applicative similarity fully abstract for contextual approximation, namely to *determinise* the transition system $\mathbb{T}$, and thus the very notion of applicative similarity. Such a construction is well-known, and dates back at least to the powerset construction in automata theory (Rabin and Scott, 1959). Roughly speaking, the idea, which we are going to analyse in a more detailed fashion, is to make $\mathbb{T}$ deterministic by lifting the transition relation to *sets* of programs.

1.2. An Abstract View on Applicative Similarity

As we have already observed, $\mathbb{T}$ is a *nondeterministic* transition system. In fact, the transition relation $\xrightarrow{\tau} \subseteq \Lambda_0 \times \mathcal{V}_0$ can be equivalently described as the evaluation function $ev : \Lambda_0 \to \mathcal{P}\mathcal{V}_0$ mapping a program e to the set $\{v \in \mathcal{V}_0 \mid e \Downarrow v\}$ (notice that $ev(e)$ might be empty, finite, or infinite). Similarly, transitions of the form $\xrightarrow{e}$ determine a function $ap : \mathcal{V}_0 \to \Lambda_0^{\Lambda_0}$ defined by $ap(\lambda x.f, e) = f[e/x]$. As a consequence, we can equivalently describe $\mathbb{T}$ as the pair[4] $(ev : \Lambda_0 \to \mathcal{P}\mathcal{V}_0, ap : \mathcal{V}_0 \to \Lambda_0^{\Lambda_0})$.

Applicative similarity can be then defined relying on a suitable notion of *relation lifting*, following the abstract approach of (Dal Lago et al., 2017). A relation lifting is a map associating to any relation $\mathcal{R} \subseteq X \times Y$ a relation $\Gamma\mathcal{R} \subseteq \mathcal{P}(X) \times \mathcal{P}(Y)$. Fixed such a map Γ, an applicative simulation can be defined as a relation $\mathcal{R} \subseteq \Lambda_0 \times \Lambda_0$ such that $e\,\mathcal{R}\,f$ implies $ev(e)\ \Gamma\mathcal{R}\ ev(f)$, and $\lambda x.f\ \mathcal{R}\ \lambda x.g$ implies $ap(\lambda x.f, k)\,\mathcal{R}\,ap(\lambda x.g, k)$, for any program k. As a consequence, in order to recover the previous notion of applicative simulation for $\Lambda^{\sqcup}$, it is sufficient to instantiate Γ as follows:

$$V\ \Gamma\mathcal{R}\ W \iff \forall v \in V.\ \exists w \in W.\ v\,\mathcal{R}\,w.$$

In this paper we consider a different approach to applicative similarity, which we could summarise in the slogan 'lift the transition system, not the relation'. In fact, instead of lifting relations between programs to relations between sets of programs, we can lift the whole transition system $\mathbb{T}$ to a transition system $\mathbb{T}^*$ such that (i) states of $\mathbb{T}^*$ are *sets* of programs, and (ii) the transition relation (and thus the associated notion of simulation) of $\mathbb{T}^*$ is deterministic. We can build $\mathbb{T}^*$ by lifting the functions ev and ap to:

$$ev^* : \mathcal{P}(\Lambda_0) \to \mathcal{P}(\mathcal{V}_0) \qquad\qquad ap^* : \mathcal{P}(\mathcal{V}_0) \to \mathcal{P}(\mathcal{V}_0)^{\Lambda_0}.$$

The latter functions are defined in the obvious way, relying on the *(strong) monad* (MacLane, 1971) structure of $\mathcal{P}$. Furthermore, as we are interested in observing whether a program may convergence, we equip $\mathbb{T}^*$ with an observation function $ob : \mathcal{P}(\mathcal{V}_0) \to \{\bot, \top\}$, where $\{\bot, \top\}$ denotes the two-element Boolean algebra, defined by $ob(V) = \top$ if and only if $V \neq \emptyset$. Notice that we have $ob(ev(e)) = \top$ if and only if $e \Downarrow$.

[4]Using the terminology of (Abramsky, 1990) we refer to transition systems of this form as nondeterministic *applicative transition systems*.

Finally, we define a notion of applicative simulation between elements in $\mathcal{P}(\Lambda_0)$ as a relation $\mathcal{R} \subseteq \mathcal{P}(\Lambda_0) \times \mathcal{P}(\Lambda_0)$ such that:

1. $E \,\mathcal{R}\, F$ implies $ob(ev^*(E)) \leq ob(ev^*(F))$;
2. $V \,\mathcal{R}\, W$ and $V, W \in \mathcal{P}(\mathcal{V}_0)$ imply $ap^*(V, e) \,\mathcal{R}\, ap^*(W, e)$, for any program e.

As usual, applicative similarity $(\preceq^{app})^*$ is defined as the largest applicative simulation.

We immediately see that although coinductively defined, this new notion of applicative similarity is a trace-like refinement. For instance, it is straightforward to see that[5] $\lambda x.(f \sqcup g)\ (\preceq^{app})^*\ \lambda x.f \sqcup \lambda x.g$. Additionally, contrary to $\preceq^{app}$, $(\preceq^{app})^*$ is not only sound, but also fully abstract for $\preceq^{ctx}$, as we will prove in full generality in Section 6.

At this point it is natural to ask whether the above construction can be given for other effectful languages, such as calculi with primitives for probabilistic nondeterminism or input/output. In fact, the reader may have noticed that the central ingredients used to build $\mathbb{T}^*$ (at least from a mathematical perspective) were the (strong) monad structure of the powerset functor $\mathcal{P}$ and a suitable observation function ob. The rest of this paper answers the above question in the affirmative by generalising the above construction to a call-by-name λ-calculus enriched with algebraic effects *à la* Plotkin and Power (Plotkin and Power, 2001), this way providing a sound and complete characterisation of contextual approximation for a large class of effectful call-by-name calculi.

2. Mathematical Preliminaries

We assume the reader to be familiar with basic domain theory and category theory. In particular, we give for granted the notions of an ω-complete partial order (ω-cpo, hereafter) and of a pointed ω-cpo (ω-cppo, hereafter), as well as the notions of a monotone, continuous, and strict function. We also assume the reader to be familiar with the basic vocabulary of category theory, viz. the notions of a category, functor, and monad. Since we will work with the category Set of sets and functions only, we simply refer to

[5] Formally, we should write $\{\lambda x.(f \sqcup g)\}\ (\preceq^{app})^*\ \{\lambda x.f \sqcup \lambda x.g\}$.

functors and monads in place of functors and monads on Set, respectively. The reader can consult (Abramsky and Jung, 1994; Davey and Priestley, 1990; MacLane, 1971) for further details.

Following (Plotkin and Power, 2001, 2003), we consider (algebraic) operations as sources of side effects. Syntactically, such operations are described by means of a signature of operation symbols, i.e. a pair $\Sigma = (\mathcal{F}, \alpha)$ consisting of a set $\mathcal{F}$ of operation symbols and a map $\alpha : \mathcal{F} \to \mathbb{N}$, assigning to each operation symbol a (finite) arity. Semantically, operation symbols are interpreted as algebraic operations on a monad/Kleisli triple (MacLane, 1971; Moggi, 1989)[6].

Definition 1. *A* monad *is a triple $\mathbb{T} = \langle T, \eta, \gg\!= \rangle$ where T is a map — called the* carrier *of $\mathbb{T}$ — associating to each set X a set TX, η — called the* unit *of $\mathbb{T}$ — is a family of functions $\eta_X : X \to TX$, and $\gg\!=$ — called the* bind *of $\mathbb{T}$ — is an operation mapping each function $f : X \to TY$ to the function $\gg\!= f : TX \to TY$ subject to the following identities:*

$$(\gg\!= f) \circ \eta = f \qquad \gg\!= \eta = id \qquad \gg\!= (\gg\!= g \circ f) = \gg\!= g \circ \gg\!= f,$$

for f and g functions with appropriate domain and codomain.

As it is customary, we write $t \gg\!= f$ in place of $\gg\!= f(t)$, for $f : X \to TY$ and $t \in TX$. Moreover, when clear from the context we will omit subscripts, thus e.g. writing $\eta(x)$ in place of $\eta_X(x)$.

Definition 2. *Let Σ be a signature and $\mathbb{T} = \langle T, \eta, \gg\!= \rangle$ be a monad. We say that Σ is* interpreted *on $\mathbb{T}$ if associated to any n-ary operation symbol σ in Σ is a set-indexed family of functions $\sigma_X : (TX)^n \to TX$. We say that Σ is* algebraic *for $\mathbb{T}$ if the following identity holds, for any n-ary operation symbol $\sigma \in \Sigma$, map $f : X \to TY$, and all elements $t_1, \ldots, t_n \in TX$:*

$$\sigma_X(t_1, \ldots, t_n) \gg\!= f = \sigma_Y(t_1 \gg\!= f, \ldots, t_n \gg\!= f).$$

Notice that algebraicity of Σ for $\mathbb{T}$ is essentially requiring each TX to carry a Σ-algebra structure such that $\gg\!= f$ is a Σ-algebra homomorphism, for any set X and function $f : X \to TY$.

[6] We use the expression *monad* and *Kleisli triple* interchangeably.

Example 1. The following are examples of monads and algebraic operations on them. Due to space constraints, we are forced to omit many interesting examples, such as the exception and global state monad, for which we refer to (Dal Lago et al., 2017; Plotkin and Power, 2003).

1. The maybe monad $\mathbb{M}$ has carrier $X_\perp = \{\mathit{just}\ x \mid x \in X\} \cup \{\perp\}$, whereas unit and bind are thus defined:

$$\eta(x) = \mathit{just}\ x \qquad (\mathit{just}\ x) \gg\!= f = f(x) \qquad \perp \gg\!= f = \perp.$$

   The maybe monad is used to model partial computations: the symbol $\perp$ denotes divergence, whereas $\mathit{just}\ x$ denotes the result of a computation giving result x. Since the possibility of divergence is an intrinsic feature of any (Turing complete) programming language, no operation is required to produce it.

2. The output monad $\mathbb{O}$ has carrier $\mathcal{O}(X) = A^\infty \times X_\perp$, where A is a given alphabet and A^∞ denotes the set of finite and infinite strings over A. Unit and bind of $\mathbb{O}$ are thus defined:

$$\eta(x) = (\varepsilon, \mathit{just}\ x) \quad (u, \perp) \gg\!= f = (u, \perp) \quad (u, \mathit{just}\ x) \gg\!= f = (uw, a),$$

   where $(w, a) = f(x)$, ε denotes the empty string, and uw denotes the concatenation of u and w (as usual, if u is infinite we define uw as u). The output monad models partial computations with output. An element (u, a) represents the result of a computation printing the string u during its processing. Due to the possibility of divergence, the string u may be infinite (and in such a case we expect a to be $\perp$). We consider a signature containing a unary A-indexed operation symbol print_c, where $c \in A$, whose intended semantics is to print the character c and then continue with its argument. Formally, we interpret $\mathsf{print}_c((u, x))$ as (cu, x).

3. The non-empty powerset monad $\mathbb{P}$ has carrier $\mathcal{P}^{ne}(X) = \{U \subseteq X \mid U \neq \emptyset\}$, whereas unit and bind are thus defined:

$$\eta(x) = \{x\} \qquad\qquad U \gg\!= f = \bigcup_{x \in U} f(x).$$

   The non-empty powerset monad is used to model total nondeterministic computations. A set $U \subseteq X$ denotes the possible results of a

computation, meaning that if $x \in U$ then x is one of the possible, nondeterministic outcomes of the computation. Nondeterministic partial computations are modelled by post-composing $\mathbb{P}$ with $\mathbb{M}$. It is a routine exercise to verify that the latter is indeed a monad. We consider a signature containing a binary operation symbol $\sqcup$ for pure nondeterministic choice. We obtain the desired behaviour by interpreting $\sqcup$ as set-theoretic union.

4. The distribution monad $\mathbb{D}$ has carrier $\mathcal{D} = \{\mu \in [0,1]^X \mid \sum_x \mu(x) = 1\}$, where the map μ is assumed to have countable support[7]. Unit and bind of $\mathbb{D}$ are thus defined:

$$\eta(x)(x') = \begin{cases} 1 & \text{if } x = x' \\ 0 & \text{otherwise} \end{cases} \qquad (\mu \gg\!= f)(y) = \sum_{x \in X} \mu(x) \cdot f(x)(y).$$

Since the unit of $\mathbb{D}$ is given by the so-called Dirac distribution, we will denote it by δ. The distribution monad is used to model total probabilistic computations, with the intended meaning that a distribution μ gives the probability of convergence to values. Probabilistic partial computations are modelled by post-composing $\mathbb{D}$ with $\mathbb{M}$. It is a routine exercise to verify that the latter is indeed a monad, and actually isomorphic to the subdistribution monad. We can consider a signature containing a binary operation symbol or for fair probabilistic choice. We obtain the desired behaviour by interpreting or as the operation $\oplus$ defined by $(\mu \oplus \nu)(x) \triangleq \frac{1}{2}\mu(x) + \frac{1}{2}\nu(x)$.

Since we will define the evaluation of a program e as the limit of its finite evaluations, we require monads to come with a suitable domain-theoretic structure.

Definition 3. *Given a monad $\mathbb{T} = \langle T, \eta, \gg\!= \rangle$ and a signature Σ interpreted on $\mathbb{T}$, we say that $\mathbb{T}$ is* Σ-continuous *if for any set X, the set TX carries an ω-cppo structure (with order $\sqsubseteq_X$) such that $\gg\!=$ is strict in its first argument (i.e. $\bot \gg\!= f = \bot$), and:*

1. *For any k-ary operation symbol σ in Σ*

$$\sigma_X(\bigsqcup_{n<\omega} t_n^1, \ldots, \bigsqcup_{n<\omega} t_n^k) = \bigsqcup_{n<\omega} \sigma_X(t_n^1, \ldots, t_n^k).$$

[7]Recall that the support of $\mu \in [0,1]^X$ is the set $\mathsf{supp}(\mu) = \{x \in X \mid \mu(x) \neq 0\}$. We write $\sum_x \mu(x)$ in place of $\sum_{x \in \mathsf{supp}(\mu)} \mu(x)$.

2. *For any ω-chain $(t_n)_{n<\omega}$ in TX and any ω-chain $(f_n)_{n<\omega}$ in $X \to TY$ (notice that the latter inherits a ω-cppo structure from TY pointwise)*

$$(\bigsqcup_{n<\omega} t_n) \gg\!= (\bigsqcup_{n<\omega} f_n) = \bigsqcup_{n<\omega} (t_n \gg\!= f_n).$$

Example 2. All monads in Example 1 are Σ-continuous. In particular, $X_\perp$ is an ω-cppo with respect to the flat order $x \sqsubseteq y \iff x = \perp$ or $x = y$. To see that $\mathbb{PM}$ is Σ-continuous consider the ordering $U \sqsubseteq V \iff just^{-1}(U) \subseteq just^{-1}(V)$. Notice that we have $\{\perp\} \sqsubseteq V$ for any set V. Finally, for $\mathbb{DM}$ define the order $\mu \sqsubseteq \nu \iff \forall x.\ \mu(just\ x) \leq \nu(just\ x)$. Notice that we have $\eta(\perp) \sqsubseteq \mu$, for any $\mu \in \mathcal{D}(X_\perp)$. Checking continuity of $\gg\!=$ and of operations is a routine exercise. Ordering the output monad $\mathbb{O}$ requires a bit more attention, due to non-monotonicity of string concatenation with respect to the prefix order $\subseteq$ on A^∞ (see (Dal Lago et al., 2017)). We define $(u, a) \sqsubseteq (w, b)$ if and only if $a = \perp$ and $u \subseteq w$, or $a = just\ x$, $b = just\ y$, $u = w$, and $x = y$.

We can now introduce the effectful calculus Λ_Σ.

3. Syntax and Operational Semantics

In this section we introduce the computational language Λ_Σ, and give it *call-by-name* monadic operational semantics. In the rest of this section let $\mathbb{T} = \langle T, \eta, \gg\!= \rangle$ be a fixed Σ-continuous monad and Σ be a fixed algebraic signature on it.

Definition 4. *The collections Λ° and $\mathcal{V}^\circ$ of expressions and values of Λ_Σ are defined by the following grammar(s), where σ ranges over n-ary operation symbols in Σ (and x ranges over a fixed set of variables).*

$$v ::= x \mid \lambda x.e \qquad\qquad e ::= v \mid ee \mid \sigma(e, \ldots, e).$$

We adopt standard conventions as in (Barendregt, 1984). In particular, we denote by $FV(e)$ the collection of free variables of an expression e and identify expressions up to renaming of bound variables. We say that an expression e is closed (resp. open) if $FV(e) = \emptyset$ (resp. $FV(e) \neq \emptyset$) and refer to closed expressions as *programs*. We denote[8] by $\Lambda(\bar{x})$ the set of expressions

[8] Oftentimes we will write $\bar{s}$ to denote a finite sequence $s_1, \ldots, s_n$ of symbols s_i.

with free variables from $\bar{x}$ and write Λ in place of $\Lambda(\emptyset)$ (similar conventions apply to the set of values). Finally, we write $f[e/x]$ for the capture-free substitution of the expression e for all free occurrences of the variable x in the expression f.

Call-by-name evaluation contexts are expressions with a single hole $[-]$ defined by the grammar $E ::= [-] \mid Ee$. We say that E is closed if it has no free variables. We write $E[e]$ for the expression obtained by substituting the expression e for the hole $[-]$ in E. *Redexes* are programs of the form $(\lambda x.f)e$ or $\sigma(e_1, \ldots, e_n)$, the former producing a computation step, the latter producing the effect described by the operation σ. We notice that any program is either a value or a program of the form $E[r]$, for a redex r.

Operational semantics is defined by means of an evaluation function $[\![-]\!] : \Lambda \to T\mathcal{V}$ mapping each program $e \in \Lambda$ to a monadic value $[\![e]\!] \in T\mathcal{V}$. For instance, a probabilistic program is evaluated to a subdistribution of values (modelled via $\mathbb{DM}$).

Definition 5. *Define the $\mathbb{N}$-indexed family of maps $[\![-]\!]_n : \Lambda \to T\mathcal{V}$ as follows:*

$$\begin{aligned} [\![e]\!]_0 &= \bot \\ [\![v]\!]_{n+1} &= \eta(v) \\ [\![E[(\lambda x.f)e]]\!]_{n+1} &= [\![E[f[e/x]]]\!]_n \\ [\![E[\sigma(e_1, \ldots, e_k)]]\!]_{n+1} &= \sigma_{\mathcal{V}}([\![E[e_1]]\!]_n, \ldots, [\![E[e_k]]\!]_n) \end{aligned}$$

The sequence $([\![e]\!]_n)_{n<\omega}$ forms an ω-chain in $T\mathcal{V}$, so that we can define $[\![e]\!]$ as $\bigsqcup_{n<\omega} [\![e]\!]_n$.

Example 3. Let $\mathbb{T}$ be $\mathbb{DM}$, and let $e = Y(\lambda x.I \oplus x)$, where I is the identity combinator and Y is Curry's fixed point combinator. Writing a distribution as a formal sum[9], for any $n \geq 1$ we have $[\![e]\!]_n = \sum_{i=1}^{n} \frac{1}{2^n} \cdot I + (1 - \sum_{i=1}^{n} \frac{1}{2^n}) \cdot \bot$, and thus $[\![e]\!] = \sup_n [\![e]\!]_n = 1 \cdot I + 0 \cdot \bot$.

Before defining behavioural equivalences and refinements for Λ_Σ, it is useful to spell out some properties of the evaluation map $[\![-]\!]$.

[9] Any discrete distribution $\mu \in \mathcal{D}(X)$ can be written as an expression of the form $\sum_{i<\omega} p_i \cdot x_i$, where $\mu(x) = \sum_{x_i = x} p_i$.

Lemma 1. *The following identities hold.*

$$\begin{aligned}[\![v]\!] &= \eta(v)\\ [\![E[(\lambda x.e)f]]\!] &= [\![E[e[f/x]]]\!]\\ [\![E[\sigma(e_1,\ldots,e_k)]\!] &= \sigma_{\mathcal{V}}([\![E[e_1]]\!],\ldots,[\![E[e_k]]\!]).\end{aligned}$$

Proof. Simply observe that since $\sigma_{\mathcal{V}}$ and $\gg\!=$ are continuous then so is $[\![-]\!]$. □

Lemma 2. *For any program e and evaluation context E, we have:*

$$[\![E[e]]\!]_n \sqsubseteq [\![e]\!]_n \gg\!= (v \mapsto [\![E[v]]\!]_n) \qquad [\![E[e]]\!] = [\![e]\!] \gg\!= (v \mapsto [\![E[v]]\!]).$$

Proof sketch. The first inclusion can be easily proved by induction on n. For the second identity one shows by induction on n that for any $n \geq 0$ $[\![e]\!]_n \gg\!= (v \mapsto [\![E[v]]\!]) \sqsubseteq [\![E[e]]\!]$ holds. As a consequence, $[\![e]\!] \gg\!= (v \mapsto [\![E[v]]\!]) \sqsubseteq [\![E[e]]\!]$. Since from the first inclusion (and Lemma 1) follows $[\![E[e]]\!] \sqsubseteq [\![e]\!] \gg\!= (v \mapsto [\![E[v]]\!])$ we conclude $[\![E[e]]\!] = [\![e]\!] \gg\!= (v \mapsto [\![E[v]]\!])$. □

4. Program Refinement

In this section we define a general notion of observation on top of which we define several notions of program refinement, the latter being defined as relations between programs. In order to qualify as a notion of a program refinement, a relation $\mathcal{R}$ needs to satisfy some minimal desiderata. Clearly, $\mathcal{R}$ must be a preorder. Furthermore, if $e\,\mathcal{R}\,f$, then the observable behaviour of f, which is given through our general notion of observation, must refine the observable behaviour of e. This property is known as *preadequacy*. Finally, $\mathcal{R}$ must respect the way programs are (syntactically) constructed, a property known as *compatibility*. Notice that relying on compatibility one can show the refinement $C[e]\,\mathcal{R}\,C[f]$ *compositionally*, by proving $e\,\mathcal{R}\,f$. Summing up, we require program refinement relations to be *preadequate precongruence* relations.

Compatibility, however, requires to extend the notion of a program refinement to arbitrary expressions of the language. Indeed, dealing with a λ-abstraction $\lambda x.e$ compositionally, requires to work with the *open* expression e. As a consequence, we work with relations on arbitrary expressions — which we call *open relations* — rather than on programs only — which we refer to as *closed relations* (see (Pitts, 2011; Gordon, 1994; Lassen, 1998)).

Definition 6. *An* open relation *is a set $\mathcal{R}$ of triples $(\bar{x}, e, f)$, where $e, f \in \Lambda(\bar{x})$, which is closed under weakening, meaning that $(\bar{x}, e, f) \in \mathcal{R}$ implies $(\bar{x} \cup \{x\}, e, f) \in \mathcal{R}$.*

We use the infix notation and write $\bar{x} \vdash e \,\mathcal{R}\, f$ in place of $(\bar{x}, e, f) \in \mathcal{R}$ and abbreviate $\emptyset \vdash e \,\mathcal{R}\, f$ as $e \,\mathcal{R}\, f$. As a convention, when referring to *relations* we tacitly mean *open relations*. There is a canonical way to extend a closed relation to an open one.

Definition 7. *The* open extension *of a closed relation $\mathcal{R}$ is the open relation $\mathcal{R}^\circ$ defined as follows:*

$$\bar{x} \vdash e \,\mathcal{R}^\circ\, f \iff \forall \bar{k} \in \Lambda.\ e[\bar{k}/\bar{x}] \,\mathcal{R}\, f[\bar{k}/\bar{x}].$$

The notion of reflexivity, symmetry, and transitivity straightforwardly extends to open relations (see e.g. (Pitts, 2011)).

Definition 8. *Let $\mathcal{R}$ be an open relation. We say that $\mathcal{R}$ is:*

1. Compatible *if it satisfies the clauses in Figure 1.*

2. Substitutive *if:*

$$\bar{x} \cup \{x\} \vdash e \,\mathcal{R}\, f \wedge \bar{x} \vdash g \,\mathcal{R}\, h \implies \bar{x} \vdash e[g/x] \,\mathcal{R}\, f[h/x].$$

3. Closed under substitution *if:*

$$\bar{x} \cup \{x\} \vdash e \,\mathcal{R}\, f \wedge k \in \Lambda(\bar{x}) \implies \bar{x} \vdash e[k/x] \,\mathcal{R}\, f[k/x].$$

Notice that the open extension of a closed relation is always closed under substitution. If $\mathcal{R}$ is a compatible preorder, we say that $\mathcal{R}$ is a *precongruence*.

Having defined the notion of a precongruence relation, we now move to *preadequacy*. As already remarked, the latter builds on a suitable notion of *observation*. Intuitively, an observation is a function that takes an element in $T\mathcal{V}$ representing the result of an effectful computation, and returns what we can observe of such a computation. Following standard practice, we assume that the observable part of a computation consists of the side effects happened during such a computation. As a consequence, the observable part of a computation is uniquely determined by the semantics of the (algebraic) operations involved.

$$\frac{}{\bar{x} \vdash x \,\mathcal{R}\, x}\ \textsf{C-var}$$

$$\frac{\bar{x} \cup \{x\} \vdash e \,\mathcal{R}\, f}{\bar{x} \vdash \lambda x.e \,\mathcal{R}\, \lambda x.f}\ \textsf{C-abs} \qquad \frac{\bar{x} \vdash e \,\mathcal{R}\, f \qquad \bar{x} \vdash g \,\mathcal{R}\, h}{\bar{x} \vdash eg \,\mathcal{R}\, fh}\ \textsf{C-app}$$

$$\frac{\bar{x} \vdash e_1 \,\mathcal{R}\, f_1 \ldots \bar{x} \vdash e_n \,\mathcal{R}\, f_n}{\bar{x} \vdash \sigma(e_1, \ldots, e_n) \,\mathcal{R}\, \sigma(f_1, \ldots, f_n)}\ \textsf{C-op}$$

Figure 1: Compatibility clauses.

Definition 9. *Let $1 = \{*\}$ be the one element set and $!_X : X \to 1$ be the unique function mapping each element in X to $*$. Define the observation function $ob : T\mathcal{V} \to T1$ as $T(!_{\mathcal{V}})$. The map ob trivially extends to programs as $ob(e) = ob([\![e]\!])$.*

Proposition 1. *For all monadic values $\varphi_1, \ldots, \varphi_n$, ω-chain $(\varphi_n)_{n<\omega}$ in $T\mathcal{V}$, and value v, we have:*

$$\begin{aligned} ob(\bot) &= \bot_{T(1)} \\ ob(\eta_{\mathcal{V}}(v)) &= \eta_1(*) \\ ob(\sigma_{\mathcal{V}}(\varphi_1, \ldots, \varphi_n)) &= \sigma_1(ob(\varphi_1), \ldots, ob(\varphi_n)) \\ ob(\bigsqcup_{n<\omega} \varphi_n) &= \bigsqcup_{n<\omega} ob(\varphi_n). \end{aligned}$$

Proof. We first observe that by the very definition of monad we have $ob = \gg\!=(\eta_1 \circ\, !)$. As a consequence, we can rely on strictness and continuity of $\gg\!=$ to prove the first and last identities above. The second identity follows by Definition 1, whereas the third one is a direct consequence of algebraicity of operation symbols. □

We can now define an abstract notion of preadequacy.

Definition 10. *An open relation $\mathcal{R}$ is* preadequate *(with respect to ob) if $\emptyset \vdash e \,\mathcal{R}\, f$ implies $ob(e) \sqsubseteq ob(f)$.*

Having at our disposal the notion of a precongruence preadequate relation, we can introduce a canonical precongruence preadequate program refinement, namely Morris' *contextual approximation* (Morris, 1969).

Definition 11. *Recall that* contexts *are syntax trees defined by the grammar:*

$$C \ ::= \ [-] \mid eC \mid Ce \mid \lambda x.C.$$

As usual, we write $C[e]$ for the result of replacing the hole $[-]$ in C with the expression e. Given an expression e, we say that a context C closes e, if $C[e]$ is a program. We denote the collection of such contexts as $Cl(e)$. The open relation $\preceq^{ctx}$, called contextual approximation*, is thus defined:*

$$\bar{x} \vdash e \preceq^{ctx} f \iff \forall C \in Cl(e) \cap Cl(f).\ ob(C[e]) \sqsubseteq ob(C[f]).$$

Remark 1. Notice that in Definition 11 contexts are defined as syntax trees (i.e. expressions not considered modulo renaming of bound variables), and not as expressions. That means that when substituting an expression e for the hole $[-]$ in a context C, we cannot consider ordinary *capture-avoiding* substitution. For instance, for $C = \lambda x.[-]$ we require $C[x]$ to be $\lambda x.x$ (i.e. to capture the free variable x), whereas our definition of substitution gives $\lambda y.x$. In order to be formally precise, we should then give an explicit definition of $C[e]$, as it is done in (Pitts, 2011). Another, elegant, possibility is to characterise $\preceq^{ctx}$ as the largest compatible preadequate relation. In fact, under mild conditions on (algebraic) operations, it is easy to prove that the union of all compatible preadequate relations is itself preadequate and compatible, and thus the largest such relation (see (Dal Lago et al., 2017)).

Example 4. 1. For the maybe monad we have $ob : \mathcal{V}_\perp \to 1_\perp$ satisfying $ob(just\ v) = just\ *$ (and thus $ob(v) = just\ *$) and $ob(\perp) = \perp$. Therefore, $ob(e) = just\ *$ if and only e converges (notation $e \Downarrow$). As a consequence, $e \preceq^{ctx} f$ if and only if $\forall C.\ C[e] \Downarrow \implies C[f] \Downarrow$.

2. For the partial nondeterministic monad $\mathbb{PM}$ we have

$$\mathcal{P}^{ne}(1_\perp) = \{\{\perp\}, \{just\ *\}, \{\perp, just\ *\}\}$$

so that $ob(\perp) = \{\perp\}$, $ob(\{just\ v\}) = \{just\ *\}$ (and thus $ob(v) = \{just\ *\}$) and $ob(e \sqcup f) = ob(e) \cup ob(f)$. We thus have $ob(e) \neq \{\perp\}$ if and only if e may converge, and $\perp \notin ob(e)$ if and only if e must converge. As a consequence, according to the order defined in Example 2, we have $ob(e) \sqsubseteq ob(f)$ if and only $e \Downarrow \implies f \Downarrow$.

3. For the partial distribution monad $\mathbb{DM}$ of Example 1 we first observe that $\mathcal{D}(1_\perp) \cong [0, 1]$. It is then easy to see that we obtain $ob(\perp) = 0$, $ob(\delta(just\ v)) = 1$ (and thus $ob(v) = 1$) and $ob(e\ \mathsf{or}\ f) = ob(e) \oplus ob(f)$.

We see that $ob(e)$ gives the probability of convergence of e, meaning that $e \preceq^{ctx} f$ holds if and only if for any context C, the probability of converge of $C[f]$ is an upper bound of the probability of convergence of $C[e]$.

As stated in Remark 1, contextual approximation enjoys several nice properties, notably it is the largest preadequate compatible relation. For instance, since the composition $C_1[C_2[-]]$ of two contexts C_1, C_2 is itself a context, $\preceq^{ctx}$ is obviously compatible. Furthermore, since $\sqsubseteq$ is a preorder, then so is $\preceq^{ctx}$.

Before introducing applicative similarity, we define *CIU approximation* (Mason and Talcott, 1991), i.e. the restriction of contextual approximation to evaluation contexts.

Definition 12. *The* closed *relation* $\preceq^{ciu}$*, called* CIU approximation *is thus defined:*

$$e \preceq^{ciu} f \iff \forall E.\ ob(E[e]) \sqsubseteq ob(E[f]),$$

where E ranges over closed evaluation contexts. As usual, we can extend $\preceq^{ciu}$ to arbitrary expressions by taking its open extension.

It is immediate to see that since any evaluation context is of the form $[-]e_1 \cdots e_n$, for some programs $e_1, \ldots, e_n$, the relation $\preceq^{ciu}$ is a trace-like refinement which tests the applicative behaviour of programs against finite traces of the form $e_1, \ldots, e_n$.

5. Applicative Similarity

In this section we define *effectful applicative similarity* and prove its equivalence with $\preceq^{ciu}$. As already discussed in the introduction, at the very heart of our notion of effectful applicative similarity is the shift from the ordinary transition system of programs to the lifted transition system of 'monadic programs'. In order to qualify such a lifted system as an *applicative system*, according to the terminology of (Abramsky, 1990), we first need to lift the notion of an application from ordinary programs to monadic programs.

Definition 13. *Let e be a program and $\varphi \in T\mathcal{V}$ be a monadic value. Define* monadic application $\varphi \cdot e \in T\mathcal{V}$ *as* $\varphi \gg\!= (v \mapsto [\![ve]\!])$.

We immediately notice that $[\![ef]\!] = [\![e]\!] \cdot f$. Moreover, straightforward calculations give the following identities:

$$\bot \cdot e = \bot, \quad \eta(\lambda x.f) \cdot e = [\![f[e/x]]\!], \quad \sigma_{\mathcal{V}}(\varphi_1, \ldots, \varphi_n) \cdot e = \sigma_{\mathcal{V}}(\varphi_1 \cdot e, \ldots, \varphi_n \cdot e).$$

We can now define effectful applicative similarity.

Definition 14. *A relation $\mathcal{R} \subseteq T\mathcal{V} \times T\mathcal{V}$ is an* effectful applicative simulation *(hereafter applicative simulation) if $\varphi \,\mathcal{R}\, \psi$ implies:*

1. $ob(\varphi) \sqsubseteq ob(\psi)$.
2. $\forall e \in \Lambda.\ \varphi \cdot e \,\mathcal{R}\, \psi \cdot e$.

Applicative similarity $\preceq^{app}$ is defined as the largest *applicative simulation. We extend applicative similarity to programs as $e \preceq^{app} f \iff [\![e]\!] \preceq^{app} [\![f]\!]$.*

Notice that applicative similarity is well-defined since the defining clauses of applicative simulation in Definition 14 induce a monotone operator Φ on the complete lattice $2^{T(\mathcal{V})\times T(\mathcal{V})}$ of binary relations on monadic values. As a consequence, we can define applicative similarity as the greatest fixed point of Φ. Applicative similarity being defined coinductively, it comes with an associated *coinduction principle*:

$$\varphi \,\mathcal{R}\, \psi \wedge (\mathcal{R} \text{ applicative simulation }) \implies \varphi \preceq^{app} \psi.$$

Remark 2. Definition 14 gives the notion of an applicative simulation directly on monadic values, without explicitly defining the lifted transition system of monadic values (the transition system $\mathbb{T}^*$ of the introduction). However, it is easy to see that we can lift the transition system of programs $(ev : \Lambda \to T\mathcal{V}, ap : \mathcal{V} \to \Lambda^\Lambda)$, where $ev(e) = [\![e]\!]$ and $ap(\lambda x.f, e) = f[e/x]$, to the system $(ev^* : T\Lambda \to T\mathcal{V}, ap^* : T\mathcal{V} \to (T\Lambda)^\Lambda)$, where[10]:

$$ev^*(\xi) \triangleq \xi \gg\!= ev \qquad\qquad ap^*(\varphi, e) = \varphi \gg\!= ap(\_, e).$$

Proposition 2. *Applicative similarity is reflexive and transitive.*

Proof. The proof is by coinduction. We immediately notice that $\{(\varphi, \varphi) \mid \varphi \in T\mathcal{V}\}$ is an applicative simulation. Furthermore, we observe that if $\mathcal{R}$ and $\mathcal{S}$ are applicative simulations, then so is $\mathcal{R};\mathcal{S}$. In fact, $\varphi \,\mathcal{R}\, \psi$ and $\psi \,\mathcal{S}\, \xi$ imply $ob(\varphi) \sqsubseteq ob(\xi)$, since $\sqsubseteq$ is transitive. Additionally, $\mathcal{R}$ and $\mathcal{S}$ being applicative simulations, we have $\varphi \cdot e \,\mathcal{R}\, \psi \cdot e$ and $\psi \cdot e \,\mathcal{S}\, \xi \cdot e$, and thus $\varphi \cdot e \,(\mathcal{R};\mathcal{S})\, \xi \cdot e$, for any $e \in \Lambda$. □

[10] From a categorical perspective, our definition of lifting relies on the fact that any monad on the category Set is strong. That means that for every monad $\mathbb{T} = \langle T, \eta, \gg\!= \rangle$ on Set we can regard $\gg\!=$ as lifting any function $f : X \times Y \to TZ$ to $\gg\!= f : X \times TY \to TZ$.

We now prove the first of our main results, namely that $\preceq^{app}$ is a trace-like refinement, which actually coincides with $\preceq^{ciu}$. Such a result gives a coinductive characterisation of $\preceq^{ciu}$, which allows one to prove program refinement relying on both the induction and coinduction proof principle.

Theorem 2. *$\preceq^{app} = \preceq^{ciu}$ and $(\preceq^{app})^\circ = (\preceq^{ciu})^\circ$.*

Proof. To see that $\preceq^{app} \subseteq \preceq^{ciu}$ notice that, by the very definition of applicative simulation, $e \preceq^{app} f$ implies $\forall k \in \Lambda.\ ek \preceq^{app} fk$. As a consequence, for any evaluation context $E = [-]\bar{k}$ we have $e \preceq^{app} f$ implies $E[e] \preceq^{app} E[f]$, and thus $ob(E[e]) \sqsubseteq ob(E[f])$. Conversely, we prove $\preceq^{ciu} \subseteq \preceq^{app}$ by coinduction, showing that $\mathcal{R} \triangleq \{(\llbracket e \rrbracket \cdot \bar{k}, \llbracket f \rrbracket \cdot \bar{k}) \mid e \preceq^{ciu} f, \bar{k} \in \Lambda\}$ is an applicative simulation. Clearly $\mathcal{R}$ is closed under the lifting of application. Furthermore, we see that $ob(\llbracket e \rrbracket \cdot \bar{k}) \sqsubseteq ob(\llbracket f \rrbracket \cdot \bar{k})$, since $\llbracket e \rrbracket \cdot \bar{k} = \llbracket e\bar{k} \rrbracket$, $\llbracket f \rrbracket \cdot \bar{k} = \llbracket f\bar{k} \rrbracket$, and $e \preceq^{ciu} f$. Since $\preceq^{app} = \preceq^{ciu}$ we obviously have $(\preceq^{app})^\circ = (\preceq^{ciu})^\circ$. □

Example 5. Recall that according to standard applicative similarity a program of the form $(\lambda x.f) \sqcup (\lambda x.f)$ does not simulate $\lambda x.f \sqcup g$. We now prove that this is not the case for $\preceq^{app}$ as defined in Definition 14. Actually, we can even strengthen such a result showing that for any operation symbol such that $\eta(*) \sqsubseteq \sigma_1(\eta(*), \ldots, \eta(*))$ (lambda) abstraction distributes over σ, i.e. $\lambda x.\sigma(e_1, \ldots, e_n) \preceq^{app} \sigma(\lambda x.e_1, \ldots, \lambda x.e_n)$. To see that it is sufficient to observe that the relation

$$\mathcal{R} = \{(\eta(\lambda x.\sigma(e_1, \ldots, e_n)), \eta(\sigma(\lambda x.e_1, \ldots, \lambda x.e_n)))\} \cup \{(\varphi, \varphi) \mid \varphi \in T\mathcal{V}\}$$

is an applicative simulation. The clause on *ob* follows by hypothesis on σ, whereas for the clause on monadic application we simply notice that

$$\llbracket \lambda x.\sigma(e_1, \ldots, e_n) \rrbracket \cdot k = \sigma_{\mathcal{V}}(\llbracket e_1[k/x] \rrbracket, \ldots, \llbracket e_n[k/x] \rrbracket) = \llbracket \sigma(\lambda x.e_1, \ldots, \lambda x.e_n) \rrbracket \cdot k.$$

Finally, a similar argument shows that if $\sigma_1(\eta(*), \ldots, \eta(*)) \sqsubseteq \eta(*)$, then $\sigma(\lambda x.e_1, \ldots, \lambda x.e_n) \preceq^{app} \lambda x.\sigma(e_1, \ldots, e_n)$. Such a condition holds both for the powerset and the distribution monads with the operations in Example 1, but fails for the output monad with the print operation.

Up to this point we have introduced three notions of behavioural refinement: contextual approximation $\preceq^{ctx}$, CIU approximation $\preceq^{ciu}$, and applicative similarity $\preceq^{app}$. The former has been shown to be a precongruence, i.e. a compatible preorder, whereas the latter two to coincide and to be preorders. In the next section we show that (the open extensions of) $\preceq^{ciu}$ and $\preceq^{app}$ are precongruences too, and that $\preceq^{ctx}$, $(\preceq^{ciu})^\circ$, and $(\preceq^{app})^\circ$ coincide.

6. A CIU Theorem

In this section we prove a CIU theorem (Theorem 3) stating that (the open extension of) $\preceq^{ciu}$ coincides with $\preceq^{ctx}$. In virtue of Theorem 2 we will also obtain the equivalence between (the open extension of) $\preceq^{app}$ and $\preceq^{ctx}$, meaning that the former is fully abstract with respect to the latter. Our proof of Theorem 3 follows (Pitts, 2011) and it is based on a variation of *Howe's technique* (Howe, 1996), the standard relational construction to prove compatibility of applicative similarity. For readability, in the rest of this section we write $\preceq$ in place of $\preceq^{ciu}$.

The main idea behind Howe's technique is to extend $\preceq$ to an open relation $\preceq^H$ that is compatible and substitutive by construction. As a consequence, in order to prove that $\preceq^\circ$ is compatible, it is sufficient to prove $\preceq^\circ = \preceq^H$. Proving the inclusion $\preceq^\circ \subseteq \preceq^H$ is rather straightforward, whereas the other inclusion requires more effort.

Definition 15. *Given an open relation $\mathcal{R}$, the Howe extension $\mathcal{R}^H$ of $\mathcal{R}$ is inductively defined by the rules in Figure 2. The Howe extension of a closed relation $\mathcal{R}$ is defined as $(\mathcal{R}^\circ)^H$.*

$$\frac{\bar{x} \vdash x \,\mathcal{R}\, e}{\bar{x} \vdash x \,\mathcal{R}^H\, e}\ \textsf{H-var}$$

$$\frac{\bar{x} \cup \{x\} \vdash e \,\mathcal{R}^H\, g \qquad \bar{x} \vdash \lambda x.g \,\mathcal{R}\, f \qquad x \notin \bar{x}}{\bar{x} \vdash \lambda x.e \,\mathcal{R}^H\, f}\ \textsf{H-abs}$$

$$\frac{\bar{x} \vdash e \,\mathcal{R}^H\, k \qquad \bar{x} \vdash g \,\mathcal{R}^H\, h \qquad \bar{x} \vdash kh \,\mathcal{R}\, f}{\bar{x} \vdash eg \,\mathcal{R}^H\, f}\ \textsf{H-app}$$

$$\frac{\bar{x} \vdash e_i \,\mathcal{R}^H\, f_i \qquad \bar{x} \vdash \sigma(f_1, \ldots, f_n) \,\mathcal{R}\, g}{\bar{x} \vdash \sigma(e_1, \ldots, e_n) \,\mathcal{R}^H\, g}\ \textsf{H-op}$$

Figure 2: Howe extension.

When unambiguous, we will write $\mathcal{R}^H$ in place of $(\mathcal{R}^\circ)^H$ for a closed relation $\mathcal{R}$. The following result states some useful properties of Howe extension. The proof is standard and can be found in e.g. (Dal Lago et al., 2014).

Lemma 3. *Let $\mathcal{R}$ be a preorder closed under substitution. Then $\mathcal{R}^H$ is a compatible, reflexive, and substitutive open relation such that $\mathcal{R}^H;\mathcal{R} \subseteq \mathcal{R}^H$ and $\mathcal{R} \subseteq \mathcal{R}^H$.*

Since $\preceq$ is a preorder, and thus $\preceq^\circ$ is a preorder closed under substitution, by Lemma 3 $(\preceq)^H$ is a compatible and substitutive open relation containing $\preceq$. Before proving Theorem 3, we need to extend Howe's construction to evaluation contexts.

Definition 16. *Given an open relation $\mathcal{R}$, the Howe extension $\mathcal{R}^H_{\mathsf{E}}$ of $\mathcal{R}$ to (closed) evaluation contexts is defined in Figure 3.*

$$\frac{}{[-]\ \mathcal{R}^H_{\mathsf{E}}\ [-]}\ \mathsf{E}\text{-}nil \qquad \frac{\emptyset \vdash e\ \mathcal{R}^H\ f \qquad E\ \mathcal{R}^H_{\mathsf{E}}\ F}{Ee\ \mathcal{R}^H_{\mathsf{E}}\ Ff}\ \mathsf{E}\text{-}app$$

Figure 3: Howe extension: evaluation contexts.

Lemma 4. *Let $\mathcal{R}$ be a reflexive open relation. Then:*

$$\frac{E\ \mathcal{R}^H_{\mathsf{E}}\ F \qquad \bar{x} \vdash e\ \mathcal{R}^H\ f}{\bar{x} \vdash E[e]\ \mathcal{R}^H\ F[f]}$$

Proof. A straightforward induction on the derivation of $E\ \mathcal{R}^H_{\mathsf{E}}\ F$. □

Lemma 5. *If $E[e] \preceq^H f$, then there exist an evaluation context F and a term g such that $E \preceq^H_{\mathsf{E}} F$, $e \preceq^H g$, and $F[g] \preceq f$.*

Proof. We proceed by induction on E. If $E = [-]$, then we take $F = [-]$ and $g = f$. The thesis follows since $\preceq$ is reflexive. If E is of the form Gk, then $G[e]k \preceq^H f$ must be the conclusion of an instance of rule H-app. As a consequence, we have $G[e] \preceq^H c$, $k \preceq^H h$, and $ch \preceq f$. By induction hypothesis, from $G[e] \preceq^H c$ we obtain an evaluation context H and a term g such that $G \preceq^H_{\mathsf{E}} H$, $e \preceq^H g$, and $H[g] \preceq c$. From the latter we infer $H[g]h \preceq ch$, and thus $H[g]h \preceq f$, since $ch \preceq f$ and $\preceq$ is transitive. From $k \preceq^H h$ and $G \preceq^H_{\mathsf{E}} H$ we infer $Gk \preceq^H_{\mathsf{E}} Hh$. We are done by taking $F \equiv Hh$. □

The main technical part of the Howe's method is the so-called Key Lemma.

Lemma 6 (Key Lemma)**.** *For all (closed) evaluation contexts E, F and programs e, f, if $E \preceq^H_{\mathsf{E}} F$ and $e \preceq^H f$, then $ob(E[e]) \sqsubseteq ob(F[f])$.*

Proof. Since ob is continuous and $[\![E[e]]\!] = \bigsqcup_{n<\omega} [\![E[e]]\!]_n$ to prove the thesis it is sufficient to prove:

$$\forall n < \omega.\ ob([\![E[e]]\!]_n) \sqsubseteq ob([\![F[f]]\!]).$$

We proceed by induction on n. The case for $n = 0$ is trivial. Let $n > 0$ and suppose the thesis holds for any $m < n$. We proceed by case analysis on e.

Case 1. Suppose e is a value $\lambda x.k$. Since $e \preceq^H f$, the latter must be the conclusion of an inference of the form

$$\frac{\{x\} \vdash k \preceq^H h \qquad \lambda x.h \preceq f}{\lambda x.k \preceq^H f}\ \mathsf{H\text{-}abs}$$

We now proceed by case analysis on E. If $E \equiv [-]$, then $F \equiv [-]$ too, since $E \preceq^H_{\mathsf{E}} F$. By Proposition 1 we have:

$$ob([\![e]\!]_n) = ob(\eta(\lambda x.k)) = \eta_1(*) = ob(\eta(\lambda x.h)) \sqsubseteq ob([\![f]\!]),$$

since $\lambda x.h \preceq f$. If E is of the form $[-]g\bar{g}$, then F must be of the form $[-]c\bar{c}$ with $g \preceq^H c$ and $\bar{g} \preceq^H \bar{c}$. Moreover, we have:

$$[\![E[e]]\!]_n = [\![(\lambda x.k)g\bar{g}]\!]_n = [\![k[g/x]\bar{g}]\!]_{n-1}.$$

From $\{x\} \vdash k \preceq^H h$ and $g \preceq^H c$ we infer, by substitutivity, $k[g/x] \preceq^H h[c/x]$. The latter, together with $\bar{g} \preceq^H \bar{c}$, gives $k[g/x]\bar{g} \preceq^H h[c/x]\bar{c}$. We can now apply the induction hypothesis obtaining

$$ob([\![k[g/x]\bar{g}]\!]_{n-1}) \sqsubseteq ob([\![h[c/x]\bar{c}]\!]) = ob([\![(\lambda x.h)c\bar{c}]\!]).$$

We can now conclude the thesis since $\lambda x.h \preceq f$ implies $(\lambda x.h)c\bar{c} \preceq fc\bar{c}$.

Case 2. Suppose e is of the form $G[(\lambda x.k)g]$. By Lemma 5, from $G[(\lambda x.k)g] \preceq^H f$ we obtain the existence of an evaluation context H and a term h such that $G \preceq^H_{\mathsf{E}} H$, $(\lambda x.k)g \preceq^H h$, and $H[h] \preceq f$. In particular, $(\lambda x.k)g \preceq^H h$ must be the conclusion of an inference of the form:

$$\dfrac{\dfrac{\{x\} \vdash k \preceq^H c \qquad \lambda x.c \preceq d}{\lambda x.k \preceq^H d}\,\textsf{H-abs} \qquad \dfrac{g \preceq^H b \qquad db \preceq h}{(\lambda x.k)g \preceq^H h}\,\textsf{H-app}}{}$$

As a consequence, by substitutivity we obtain $k[g/x] \preceq^H c[b/x]$. Since $E \preceq^H_{\mathsf{E}} F$ and $G \preceq^H_{\mathsf{E}} H$ imply $E[G[-]] \preceq^H_{\mathsf{E}} F[H[-]]$ we can apply the induction hypothesis, obtaining:

$$\begin{aligned} ob([\![E[G[(\lambda x.k)g]]]\!]_n) &= ob([\![E[G[k[g/x]]]]\!]_{n-1}) \\ &\sqsubseteq ob([\![F[H[c[b/x]]]]\!]) \\ &= ob([\![F[H[(\lambda x.c)b]]]\!]) \\ &\sqsubseteq ob([\![F[H[db]]]\!]) \\ &\sqsubseteq ob([\![F[H[h]]]\!]) \\ &\sqsubseteq ob([\![F[f]]\!]). \end{aligned}$$

In the last inequality we have have used the following implications:

$$\lambda x.c \preceq d \implies (\lambda x.c)b \preceq db \implies F[H[(\lambda x.c)b]] \preceq F[H[db]]$$

and similarity for $db \preceq h$ and $H[h] \preceq f$.

Case 3. Suppose e is of the form $G[\sigma(c_1, \ldots, c_k)]$. We proceed exactly as in case 2. From $G[\sigma(c_1, \ldots, c_k)] \preceq^H f$ by Lemma 5 we obtain an evaluation context H and a term d such that $H[d] \preceq f$, $G \preceq^H_{\mathsf{E}} H$, and $\sigma(c_1, \ldots, c_k) \preceq^H d$. The latter must be the conclusion of an inference of the form:

$$\dfrac{c_i \preceq^H b_i \qquad \sigma(b_1, \ldots, b_k) \preceq d}{\sigma(c_1, \ldots, c_k) \preceq^H d}\,\textsf{H-op}$$

As in previous case we observe that $E \preceq^H_{\mathsf{E}} F$ and $G \preceq^H_{\mathsf{E}} H$ imply $E[G[-]] \preceq^H_{\mathsf{E}} F[H[-]]$. Reasoning as above we can now apply the induction hypothesis, obtaining:

$$\begin{aligned} ob([\![E[G[\sigma(c_1, \ldots, c_k)]]]\!]_n) &= \sigma_1(ob([\![E[G[c_1]]]\!]_{n-1}), \ldots, ob([\![E[G[c_k]]]\!]_{n-1})) \\ &\sqsubseteq \sigma_1(ob([\![F[H[b_1]]]\!]), \ldots, ob([\![F[H[b_k]]]\!])) \\ &= ob([\![F[H[\sigma(b_1, \ldots, b_k)]]]\!]) \\ &\sqsubseteq ob([\![F[H[d]]]\!]) \\ &\sqsubseteq ob([\![F[f]]\!]). \end{aligned}$$

□

Theorem 3 (CIU)**.** $\preceq^{\circ} = \preceq^{ctx}$.

Proof. To prove compatibility of $\preceq$ it is sufficient to show that $\preceq^{H}$ restricted to programs coincide with $\preceq$. The latter amounts to show that $\emptyset \vdash e \preceq^{H} f$ implies $ob(E[e]) \sqsubseteq ob(E[f])$, for any evaluation context E. This directly follows from Key Lemma, since $E \preceq^{H}_{\mathsf{E}} E$ always holds. Since $\preceq^{\circ}$ is obviously preadequate, we see that $\preceq^{\circ} \subseteq \preceq^{ctx}$. Since $\preceq^{ctx} \subseteq \preceq^{\circ}$ trivially holds, we conclude $\preceq^{\circ} = \preceq^{ctx}$. □

Corollary 1 (Full abstraction)**.** $(\preceq^{ciu})^{\circ} = \preceq^{ctx} = (\preceq^{app})^{\circ}$.

We conclude this section observing that applicative similarity, contextual approximation, and CIU approximation being trace-like refinement, we can straightforwardly extend Corollary 1 to their symmetric counterparts.

Corollary 2. *Define* applicative bisimilarity $\simeq^{app}$, contextual equivalence $\simeq^{ctx}$, *and* CIU equivalence $\simeq^{ciu}$ *as* $\preceq^{app} \cap (\preceq^{app})^{-1}$, $\preceq^{ctx} \cap (\preceq^{ctx})^{-1}$, *and* $\preceq^{ciu} \cap (\preceq^{ciu})^{-1}$, *respectively, where* $\mathcal{R}^{-1}$ *denotes the converse of* $\mathcal{R}$. *Then,* $(\simeq^{ciu})^{\circ} = \simeq^{ctx} = (\simeq^{app})^{\circ}$.

7. Call-by-name, Call-by-value, and Call-by-need

Corollaries 1 and 2 give coinductive characterisations of contextual approximation and equivalence, respectively. Such characterisations are proved through a CIU theorem, which takes advantage of the shape of call-by-name evaluation contexts. It is then natural to ask whether our results can be extended to encompass call-by-value calculi, the latter being more natural in presence of effects.

An effectful notion of applicative bisimilarity for a call-by-value λ-calculus with algebraic effects has been given in (Dal Lago et al., 2017), where an abstract soundness theorem for it is proved. Such a notion, however, is not fully abstract, as already observed in (Lassen, 1998) in the context of nondeterministic calculi. It is straightforward to see that the notion of applicative (bi)similarity proposed in the present paper can be easily given for a call-by-value calculus. Unfortunately, the resulting notion would not only invalidate Corollary 1 and Corollary 2, but would also be *unsound*, as witnessed by the probabilistic programs $\lambda x.(x \,\mathsf{or}\, \Omega)$ and $(\lambda x.x) \,\mathsf{or}\, (\lambda x.\Omega)$, and the context $(\lambda x.x(x(\lambda y.y)))[-]$.

Another issue that we have not addressed in this work is whether our techniques can be extended to call-by-need calculi. Concerning program equivalence and refinement, call-by-name and call-by-need calculi are equivalent in absence of computational effects (different from nontermination). Such an equivalence breaks in presence of effects such as pure and probabilistic nondeterminism, global states, and input-output. For instance, the program $(\lambda x.x \text{ or } x)((\lambda y.y) \text{ or } \Omega)$ converges with probability 0.5 if evaluated according to a call-by-need strategy, and with probability 0.25 if evaluated according to a call-by-name strategy.

Although our techniques seem to scale to specific effectful call-by-need languages (such as languages with primitives for raising exceptions), the study of coinductive techniques for call-by-need languages with algebraic languages is nontrivial, and we leave it for future work.

8. Conclusion and Future Work

We have shown how a notion of applicative similarity on call-by-name effectful lambda-calculi can be defined and proved fully-abstract. This is very much in line with logical relations as introduced in (Johann et al., 2010), but simpler and applicable to untyped λ-calculi. The underlying transition system has monadic values as states, and is essentially deterministic. This is indeed the reason the framework is only applicable to call-by-name (and not to call-by-value) calculi, in contrast with (Dal Lago et al., 2017).

As a future work, the authors would like to investigate other transition system-based techniques to obtain sound (and complete) notions of (bi)simulations for effectful calculi. A notable result in such a direction is (Gianantonio et al., 2009).

Abramsky, S., 1990. The lazy lambda calculus. In: Turner, D. (Ed.), Research Topics in Functional Programming. Addison Wesley, pp. 65–117.

Abramsky, S., Jung, A., 1994. Domain theory. In: Handbook of Logic in Computer Science. Clarendon Press, pp. 1–168.

Appel, A. W., McAllester, D. A., 2001. An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. 23 (5), 657–683.

Barendregt, H. P., 1984. The lambda calculus: its syntax and semantics. Studies in logic and the foundations of mathematics. North-Holland.

Benton, N., Kennedy, A., Beringer, L., Hofmann, M., 2009. Relational semantics for effect-based program transformations: higher-order store. In: Proc. of PPDP 2009. pp. 301–312.

Bizjak, A., Birkedal, L., 2015. Step-indexed logical relations for probability. In: Proc. of FOSSACS 2015. pp. 279–294.

Crubillé, R., Dal Lago, U., 2014. On probabilistic applicative bisimulation and call-by-value λ-calculi. In: Proc. of ESOP 2014. Vol. 8410 of LNCS. Springer, pp. 209–228.

Dal Lago, U., Gavazzo, F., Levy, P., 2017. Effectful applicative bisimilarity: Monads, relators, and Howe's method (long version), available at `https://arxiv.org/abs/1704.04647`.

Dal Lago, U., Sangiorgi, D., Alberti, M., 2014. On coinductive equivalences for higher-order probabilistic functional programs. In: Proc. of POPL 2014. pp. 297–308.

Davey, B. A., Priestley, H. A., 1990. Introduction to lattices and order. Cambridge University Press.

de'Liguoro, U., Piperno, A., 1995. Non deterministic extensions of untyped lambda-calculus. Inf. Comput. 122 (2), 149–177.

Gianantonio, P. D., Honsell, F., Lenisa, M., 2009. Rpo, second-order contexts, and lambda-calculus. Logical Methods in Computer Science 5 (3).

Gordon, A. D., September 1994. A tutorial on co-induction and functional programming. In: Workshops in Computing. Springer London, pp. 78–95. URL `https://www.microsoft.com/en-us/research/publication/a-tutorial-on-co-induction-and-functional-programming/`

Goubault-Larrecq, J., Lasota, S., Nowak, D., 2008. Logical relations for monadic types. Mathematical Structures in Computer Science 18 (6), 1169–1217.

Howe, D. J., 1996. Proving congruence of bisimulation in functional programming languages. Inf. Comput. 124 (2), 103–112.

Johann, P., Simpson, A., Voigtländer, J., 2010. A generic operational metatheory for algebraic effects. In: Proc. of LICS 2010. IEEE Computer Society, pp. 209–218.

Jones, C., 1990. Probabilistic non-determinism. Ph.D. thesis, University of Edinburgh, UK.

Lassen, S. B., May 1998. Relational reasoning about functions and nondeterminism. Ph.D. thesis, Dept. of Computer Science, University of Aarhus.

MacLane, S., 1971. Categories for the Working Mathematician. Springer-Verlag.

Mason, I. A., Talcott, C. L., 1991. Equivalence in functional languages with effects. J. Funct. Program. 1 (3), 287–327.

Moggi, E., 1989. Computational lambda-calculus and monads. In: Proc. of (LICS 1989. IEEE Computer Society, pp. 14–23.

Morris, J., 1969. Lambda calculus models of programming languages. Ph.D. thesis, MIT.

Ong, C. L., 1993. Non-determinism in a functional setting. In: Proc. of LICS 1993. IEEE Computer Society, pp. 275–286.

Pitts, A. M., Nov. 2011. Howe's method for higher-order languages. In: Sangiorgi, D., Rutten, J. (Eds.), Advanced Topics in Bisimulation and Coinduction. Vol. 52 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Ch. 5, pp. 197–232.

Plotkin, G. D., Power, J., 2001. Adequacy for algebraic effects. In: Proc. of FOSSACS 2001. pp. 1–24.

Plotkin, G. D., Power, J., 2003. Algebraic operations and generic effects. Applied Categorical Structures 11 (1), 69–94.

Rabin, M. O., Scott, D., 1959. Finite automata and their decision problems. IBM J. Res. Dev. 3 (2), 114–125.