Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 21 pages

Similarity and Bisimilarity for Countable Non-Determinism and Higher-Order Functions (Extended Abstract)

Søren B. Lassen [1]

University of Cambridge Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, United Kingdom
Soeren.Lassen@cl.cam.ac.uk

Corin S. Pitcher [2]

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom
Corin.Pitcher@comlab.ox.ac.uk

Abstract

This paper investigates operationally-based theories of a simply-typed functional programming language with countable non-determinism. The theories are based upon lower, upper, and convex variants of applicative similarity and bisimilarity, and the main result presented here is that these relations are compatible. The differences between the relations are illustrated by simple examples, and their continuity properties are discussed. It is also shown that, in some cases, the addition of countable non-determinism to a programming language with finite non-determinism alters the theory of the language.

Key words: lambda-calculus, applicative bisimilarity, countable non-determinism.

1 Introduction

Non-deterministic programs are used in the study of concurrent systems, to abstract from scheduling details, and in methodologies for program construction, where specifications are regarded as non-deterministic programs. In recent years, several non-deterministic higher-order languages have been proposed in the literature in these areas (see, e.g., [28,4]). Non-determinism is

[1] Supported by a grant from the Danish Natural Science Research Council and grant number GR/L38356 from the UK EPSRC.
[2] Partially supported by a UK EPSRC studentship.

also found as an integrated feature of the higher-order, operationally-based semantic meta-language of action semantics [19]. In this paper we use operational techniques to study the interaction between non-determinism and higher-order functions in an idealised, minimal programming language.

We investigate three variants, lower, upper, and convex, of applicative similarity and bisimilarity for a simply-typed functional programming language with countable non-determinism. This builds upon work by Abramsky, Howe, and Ong [1,11,12,21] for deterministic and finitely non-deterministic higher-order calculi.

The variants of the relations correspond to the different constructions on preorders that are used to characterise the lower, upper and convex powerdomains. Their definitions refer to an inductively defined may convergence relation between terms, and also a co-inductively defined may divergence predicate on terms, for the upper and convex variants. For each variant there is an applicative similarity preorder and an applicative bisimilarity equivalence relation, both defined by co-induction. In addition, the applicative similarity preorders determine mutual applicative similarity equivalence relations that do not coincide with applicative bisimilarity. The proliferation of preorders and equivalences reflects the conflicting requirements of different applications for semantic theories of non-determinism. This complexity is not apparent in the absence of non-determinism, because the nine relations defined here collapse to just two.

It is of fundamental importance to know whether the relations are compatible, i.e., are they preserved by the constructors of the language? We prove that this is the case for all of the relations, extending methods due to Howe and Ong that were previously restricted to finitely non-deterministic languages for the upper and convex variants. By the use of induction on the derivation of a must convergence judgement (the complement of the may divergence predicate) their methods are extended smoothly to a language with countable non-determinism.

Must convergence is defined inductively via a finite collection of infinitary rule schema, and so ordinal heights can be assigned to the derivation trees of must convergence judgements in the usual way. Such trees have heights strictly less than ω for finite non-determinism, and heights strictly less than the least non-recursive ordinal ω_1^{CK} for countable non-determinism. This allows us to prove unwinding theorems for fixed points terms with respect to must convergence: ω-unwinding in the case of finitely non-deterministic terms, and a more unusual ω_1^{CK}-unwinding for countably non-deterministic terms.

2 A Functional Language with Non-Determinism

The vehicle for the examples and results in this paper is a variant of the language of Moggi's computational lambda-calculus [17,7]. Within the computational lambda-calculus there is a distinction between values and computations that is enforced by the type system through a type-constructor for computation types. There are mechanisms for creating and composing the programs of computation types.

The language is extended with an operator $?\mathbb{N}$ to choose any natural number. The new construct is the sole source of non-determinism in the language, and, because it is assigned an appropriate computation type, non-determinism is restricted to the computation types. This restriction is convenient because the mechanism for composing computations can be used to control when non-determinism is resolved—an alternative is to incorporate both call-by-name and call-by-value abstractions (see, e.g., [22]). In addition, although the examples presented here have analogues at function types, they are simpler at computation types.

The types of the language are:

$$\sigma, \tau ::= \mathsf{unit} \mid \mathsf{nat} \mid \sigma \to \tau \mid \mathsf{P}(\sigma)$$

The computation types are those of the form $\mathsf{P}(\sigma)$, and the remaining types are called deterministic types.

The terms of the language are:

$$\begin{aligned} L, M, N ::= {} & x \mid \star \mid n \mid \mathsf{uop}\,(M) \mid \mathsf{bop}\,(M, N) \mid \\ & \mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \mid \lambda x : \sigma.\, M \mid M\, N \mid \\ & [M] \mid \mathsf{let}\, x : \sigma \Leftarrow M\, \mathsf{in}\, N \mid \mathsf{fix}\, x : \sigma.\, M \mid ?\mathbb{N} \end{aligned}$$

where x ranges over a countably infinite set of variables, n ranges over $\mathbb{N}$, and uop and bop range over a suitable set of symbols representing, respectively, unary and binary primitive recursive functions, e.g., not, plus, leq. For the sake of economy, booleans are represented by natural numbers: 0 for false, and 1 for true. The primitive recursive functions are assumed to follow this representation, and are denoted, e.g., $(\!|\mathsf{not}|\!) : \mathbb{N} \to \mathbb{N}$, $(\!|\mathsf{plus}|\!), (\!|\mathsf{leq}|\!) : \mathbb{N}^2 \to \mathbb{N}$. Variable binding terms follow the usual conventions for scope, α-conversion, and type annotations to ensure uniqueness of typing. The notation $M[N/x]$ denotes the capture-free substitution of N for free occurrences of x in M. The canonical terms are:

$$K ::= \star \mid n \mid \lambda x : \sigma.\, M \mid [M]$$

The type assignment rules in figure 1 are based on those of the computational lambda-calculus. We follow the convention that environments are partial functions, and that $\Gamma, x : \sigma$ is only defined when x is not in the domain of Γ.

The sets of terms and canonical terms that are closed and well-typed are *Exp* and *Can* respectively. A term that is closed and well-typed is called a program. The set of programs of type σ is Exp_σ.

$$\Gamma \vdash x : \sigma \quad (\Gamma(x) = \sigma) \qquad \Gamma \vdash \star : \mathsf{unit} \qquad \Gamma \vdash n : \mathsf{nat} \quad (n \in \mathbb{N})$$

$$\frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{uop}\,(M) : \mathsf{nat}} \qquad \frac{\Gamma \vdash M : \mathsf{nat} \qquad \Gamma \vdash N : \mathsf{nat}}{\Gamma \vdash \mathsf{bop}\,(M, N) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash L : \mathsf{nat} \qquad \Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\, M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\, N : \tau}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash [M] : \mathsf{P}(\sigma)} \qquad \frac{\Gamma \vdash M : \mathsf{P}(\sigma) \qquad \Gamma, x : \sigma \vdash N : \mathsf{P}(\tau)}{\Gamma \vdash \mathsf{let}\, x : \sigma \Leftarrow M\, \mathsf{in}\, N : \mathsf{P}(\tau)}$$

$$\frac{\Gamma, x : \mathsf{P}(\sigma) \vdash M : \mathsf{P}(\sigma)}{\Gamma \vdash \mathsf{fix}\, x : \mathsf{P}(\sigma).\, M : \mathsf{P}(\sigma)} \qquad \Gamma \vdash ?\mathbb{N} : \mathsf{P}(\mathsf{nat})$$

Fig. 1. Type Assignment

Many of the examples that we give do not depend on the existence of distinct canonical programs at a base type, and in such cases we use the unit type in preference to nat.

The operational semantics is presented as an inductively defined evaluation relation $\Downarrow^{\mathrm{may}}$, called may convergence, between programs and canonical closed terms. The rules are shown in figure 2. The may convergence relation is not a partial function because of the rule that allows $?\mathbb{N}$ to converge to any natural number.

In contrast to the situation for deterministic programs, the divergent (non-terminating) behaviour of a non-deterministic program is not determined by its convergent behaviour. Following [6,8,18] we define a may divergence predicate $\Uparrow^{\mathrm{may}}$ on programs by co-induction. The may divergence rules are given in figure 3. The symbol $(-)$ at the side of each rule is used to indicate that the may divergence predicate is the greatest fixed point of the monotone function determined by the rules. Note that there is some redundancy in the may divergence rules, because it can be shown that programs of deterministic types cannot diverge.

Examples 2.1 and 2.2 highlight properties of the programming language that are relevant in the sequel.

Example 2.1 *The construct for sequencing computations in the computational lambda-calculus provides an additional degree of control over the resolution of non-determinism. For example, a call-by-value abstraction is definable at computation types (where y is fresh for M):*

$$\lambda^v x : \sigma.\, M \stackrel{\mathrm{def}}{=} \lambda y : \mathsf{P}(\sigma).\, \mathsf{let}\, x : \sigma \Leftarrow y\, \mathsf{in}\, M$$

The call-by-value abstraction exhibits (weak) call-time choice, i.e., a non-

$$K \Downarrow^{\text{may}} K \quad (K \in Can)$$

$$\frac{M \Downarrow^{\text{may}} n}{\mathsf{uop}\,(M) \Downarrow^{\text{may}} (\!|\mathsf{uop}|\!)(n)} \qquad \frac{M \Downarrow^{\text{may}} n_1 \qquad N \Downarrow^{\text{may}} n_2}{\mathsf{bop}\,(M, N) \Downarrow^{\text{may}} (\!|\mathsf{bop}|\!)(n_1, n_2)}$$

$$\frac{L \Downarrow^{\text{may}} n+1 \qquad M \Downarrow^{\text{may}} K}{\mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \Downarrow^{\text{may}} K}\ (n \in \mathbb{N}) \qquad \frac{L \Downarrow^{\text{may}} 0 \qquad N \Downarrow^{\text{may}} K}{\mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \Downarrow^{\text{may}} K}$$

$$\frac{M \Downarrow^{\text{may}} \lambda x.\, M_1 \qquad M_1[N/x] \Downarrow^{\text{may}} K}{M\, N \Downarrow^{\text{may}} K}$$

$$\frac{M \Downarrow^{\text{may}} [M_1] \qquad N[M_1/x] \Downarrow^{\text{may}} K}{\mathsf{let}\, x \Leftarrow M\, \mathsf{in}\, N \Downarrow^{\text{may}} K}$$

$$\frac{M[\mathsf{fix}\, x.\, M/x] \Downarrow^{\text{may}} K}{\mathsf{fix}\, x.\, M \Downarrow^{\text{may}} K} \qquad ?\mathbb{N} \Downarrow^{\text{may}} [n] \quad (n \in \mathbb{N})$$

Fig. 2. May Convergence

deterministic program of type $\mathsf{P}(\sigma)$ *is resolved to program of type* σ *at the time that it is passed as an argument.*

Example 2.2 *Recursion is only available at computation types, but given a term* $\Gamma, f : \sigma \to \mathsf{P}(\tau) \vdash M : \sigma \to \mathsf{P}(\tau)$ *the following acts as a fixed point (where* g *and* x *are fresh for* M*):*

$$\Gamma \vdash \lambda x.\, \mathsf{let}\, f \Leftarrow \mathsf{fix}\, g.\ [\lambda x.\, \mathsf{let}\, f \Leftarrow g\, \mathsf{in}\, M\, x]\ \mathsf{in}\, M\, x : \sigma \to \mathsf{P}(\tau)$$

Non-determinism is often introduced via a binary operator, binary erratic choice. It can be defined in the programming language in terms of $?\mathbb{N}$.

Example 2.3 *The binary erratic choice of programs* M *and* N *of the same computation type is defined to be (where* y *is fresh for* M *and* N*):*

$$(M\ \mathsf{or}\ N) \stackrel{\text{def}}{=} \mathsf{let}\, y \Leftarrow ?\mathbb{N}\ \mathsf{in}\ \mathsf{if}\, y\, \mathsf{then}\, M\, \mathsf{else}\, N$$

Non-determinism is informally classified by the cardinalities of the sets of convergent behaviours of programs that cannot diverge (see section 6 also). For example, the binary erratic choice of deterministic terms is said to be finitely non-deterministic, whereas $?\mathbb{N}$ is said to be countably non-deterministic. König's lemma ensures that recursion does not provide a route from finite to countable non-determinism.

Example 2.4 *The program below can converge to any natural number, but may also diverge:*

$$\vdash \mathsf{fix}\, x.\, ([0]\ \mathsf{or}\ \mathsf{let}\, y \Leftarrow x\ \mathsf{in}\ [\mathsf{plus}\,(y, 1)]) : \mathsf{P}(\mathsf{nat})$$

It cannot be distinguished from $?\mathbb{N}$ *by equivalences that ignore divergent behaviour.*

$$\frac{M \Uparrow^{\mathrm{may}}}{\mathsf{uop}\,(M) \Uparrow^{\mathrm{may}}} - \qquad \frac{M \Uparrow^{\mathrm{may}}}{\mathsf{bop}\,(M, N) \Uparrow^{\mathrm{may}}} - \qquad \frac{N \Uparrow^{\mathrm{may}}}{\mathsf{bop}\,(M, N) \Uparrow^{\mathrm{may}}} -$$

$$\frac{L \Uparrow^{\mathrm{may}}}{\mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \Uparrow^{\mathrm{may}}} -$$

$$\frac{M \Uparrow^{\mathrm{may}}}{\mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \Uparrow^{\mathrm{may}}} - \quad (L \Downarrow^{\mathrm{may}} n+1 \text{ and } n \in \mathbb{N})$$

$$\frac{N \Uparrow^{\mathrm{may}}}{\mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \Uparrow^{\mathrm{may}}} - \quad (L \Downarrow^{\mathrm{may}} 0)$$

$$\frac{M \Uparrow^{\mathrm{may}}}{M\ N \Uparrow^{\mathrm{may}}} - \qquad \frac{M_1[N/x] \Uparrow^{\mathrm{may}}}{M\ N \Uparrow^{\mathrm{may}}} - \quad (M \Downarrow^{\mathrm{may}} \lambda x.\, M_1)$$

$$\frac{M \Uparrow^{\mathrm{may}}}{\mathsf{let}\, x \Leftarrow M\, \mathsf{in}\, N \Uparrow^{\mathrm{may}}} - \qquad \frac{N[M_1/x] \Uparrow^{\mathrm{may}}}{\mathsf{let}\, x \Leftarrow M\, \mathsf{in}\, N \Uparrow^{\mathrm{may}}} - \quad (M \Downarrow^{\mathrm{may}} [M_1])$$

$$\frac{M[\mathsf{fix}\, x.\, M/x] \Uparrow^{\mathrm{may}}}{\mathsf{fix}\, x.\, M \Uparrow^{\mathrm{may}}} -$$

Fig. 3. May Divergence

3 Similarity and Bisimilarity

Abramsky [1] develops a notion of applicative similarity for the untyped lazy lambda-calculus, building upon earlier work of Park and Milner [16] in the context of process calculi. The preorders and equivalences described in this section are based upon Abramsky's work and subsequent extensions to non-deterministic functional languages by Howe and Ong [12,21].

There are two fundamental differences between the deterministic and non-deterministic settings: applicative bisimilarity is not the same as mutual applicative similarity, and there are different ways of ordering non-deterministic programs that correspond to the constructions on preorders used to characterise the lower, upper, and convex powerdomains (see, e.g., [10,27]). This leads to nine distinct variations of applicative similarity and bisimilarity for non-deterministic programs, which collapse to just two relations on deterministic programs.

For the sake of brevity, "applicative" is implicit when similarity or bisimilarity are used in the sequel. The reader is also cautioned that terminology for (what we call) similarity or bisimilarity differs amongst authors. We use the following conventions: simulations and bisimulations are post-fixed points of a function; similarity and bisimilarity are the greatest simulations and bisimulations respectively; the prefix "bi" refers to a function on relations with a symmetric image; mutual similarity is the greatest symmetric relation contained in similarity.

The variants of similarity and bisimilarity are defined in terms of two functions of binary relations on programs. For a binary relation $\mathcal{R}$ on programs, we define binary relations on programs: $\langle\mathcal{R}\rangle_{\mathrm{LS}}$ and $\langle\mathcal{R}\rangle_{\mathrm{US}}$. The subscripts abbreviate lower similarity and upper similarity.

Definition 3.1 Let $\mathcal{R}$ be a binary relation on programs. The binary relations $\langle\mathcal{R}\rangle_{\mathrm{LS}}$ and $\langle\mathcal{R}\rangle_{\mathrm{US}}$ are defined by:

(i) $M, N \in \mathit{Exp}_\sigma$ are related by $\langle\mathcal{R}\rangle_{\mathrm{LS}}$ if:
 (a) $\sigma = \mathsf{unit}$; or
 (b) $\sigma = \mathsf{nat}$ and $\exists n \in \mathbb{N}.\ M \Downarrow^{\mathrm{may}} n \ \wedge\ N \Downarrow^{\mathrm{may}} n$; or
 (c) $\sigma = \tau_1 \to \tau_2$ and $\forall L \in \mathit{Exp}_{\tau_1}.\,(M\,L)\,\mathcal{R}\,(N\,L)$; or
 (d) $\sigma = \mathsf{P}(\tau)$ and $\forall M_1.M \Downarrow^{\mathrm{may}} [M_1] \Longrightarrow (\exists N_1.N \Downarrow^{\mathrm{may}} [N_1] \wedge M_1\,\mathcal{R}\,N_1)$.

(ii) $M, N \in \mathit{Exp}_\sigma$ are related by $\langle\mathcal{R}\rangle_{\mathrm{US}}$ if:
 (a) $\sigma = \mathsf{unit}$; or
 (b) $\sigma = \mathsf{nat}$ and $\exists n \in \mathbb{N}.\ M \Downarrow^{\mathrm{may}} n \ \wedge\ N \Downarrow^{\mathrm{may}} n$; or
 (c) $\sigma = \tau_1 \to \tau_2$ and $\forall L \in \mathit{Exp}_{\tau_1}.\,(M\,L)\,\mathcal{R}\,(N\,L)$; or
 (d) $\sigma = \mathsf{P}(\tau)$ and $\neg(M \Uparrow^{\mathrm{may}}) \Longrightarrow$
 $(\neg(N \Uparrow^{\mathrm{may}}) \wedge\ \forall N_1.N \Downarrow^{\mathrm{may}} [N_1] \Longrightarrow (\exists M_1.M \Downarrow^{\mathrm{may}} [M_1] \wedge M_1\,\mathcal{R}\,N_1))$.

The functions $\langle\cdot\rangle_{\mathrm{LS}}$ and $\langle\cdot\rangle_{\mathrm{US}}$ differ only in their action at computation types. If the assumption that divergent behaviour is less than any convergent behaviour is made explicit, then an immediate connection can be made with one of the methods used to construct the lower and upper powerdomains [21,23].

We are now in a position to define the nine variants of similarity and bisimilarity. Six of the relations are defined as the greatest fixed points of combinations of the functions defined above. However, it is easy to verify by induction that the simple type system of the computational lambda-calculus ensures that the greatest fixed points of the functions are also least fixed points. The remaining relations, the mutual similarities, are the greatest symmetric relations contained in the three similarities.

Definition 3.2 The similarity and bisimilarity relations are defined by (where $\nu\mathcal{R}.\phi(R)$ denotes the greatest fixed point of ϕ):

$$\begin{aligned}
\lesssim_{\mathrm{LS}} &\stackrel{\text{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}}\\
\lesssim_{\mathrm{US}} &\stackrel{\text{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{US}}\\
\lesssim_{\mathrm{CS}} &\stackrel{\text{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}} \cap \langle\mathcal{R}\rangle_{\mathrm{US}}\\
\simeq_{\mathrm{LB}} &\stackrel{\text{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle_{\mathrm{LS}}^{\mathrm{op}}\\
\simeq_{\mathrm{UB}} &\stackrel{\text{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{US}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle_{\mathrm{US}}^{\mathrm{op}}\\
\simeq_{\mathrm{CB}} &\stackrel{\text{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}} \cap \langle\mathcal{R}\rangle_{\mathrm{US}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle_{\mathrm{LS}}^{\mathrm{op}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle_{\mathrm{US}}^{\mathrm{op}}
\end{aligned}$$

In addition, the mutual similarities $\simeq_{\mathrm{LS}}$, $\simeq_{\mathrm{US}}$, and $\simeq_{\mathrm{CS}}$ are defined to be the greatest symmetric relations contained in $\lesssim_{\text{LS}}$, $\lesssim_{\text{US}}$, and $\lesssim_{\text{CS}}$ respectively. The names of the relations are summarised in the table below.

	Lower	Upper	Convex
Similarity	$\lesssim_{\text{LS}}$	$\lesssim_{\text{US}}$	$\lesssim_{\text{CS}}$
Mutual Similarity	$\simeq_{\text{LS}}$	$\simeq_{\text{US}}$	$\simeq_{\text{CS}}$
Bisimilarity	$\simeq_{\text{LB}}$	$\simeq_{\text{UB}}$	$\simeq_{\text{CB}}$

We refer to the tutorial papers [9,26] for the standard results concerning similarities and bisimilarities: each similarity is a preorder; each bisimilarity and mutual similarity is an equivalence; and the program that cannot converge, $\Omega \stackrel{\text{def}}{=} \mathsf{fix}\, x.\, x$, is a least element for each of the similarities. In addition, it is immediate from the definitions that programs related by any of the similarities or bisimilarities have the same type.

Although the method of definition of the similarities and bisimilarities is convenient for the proof of compatibility in section 4, it is helpful to have the unwound definition to mind. In the case of convex bisimilarity we have that, if M and N are programs of the same computation type, then $M \simeq_{\text{CB}} N$ if and only if:

(i) $\forall M_1.M \Downarrow^{\text{may}} [M_1] \Longrightarrow (\exists N_1.N \Downarrow^{\text{may}} [N_1] \wedge M_1 \simeq_{\text{CB}} N_1)$; and

(ii) $\forall N_1.N \Downarrow^{\text{may}} [N_1] \Longrightarrow (\exists M_1.M \Downarrow^{\text{may}} [M_1] \wedge M_1 \simeq_{\text{CB}} N_1)$; and

(iii) $M \Uparrow^{\text{may}} \Longleftrightarrow N \Uparrow^{\text{may}}$.

Lower bisimilarity follows the same pattern as convex bisimilarity with the exception that condition (iii) is dropped. We omit the unwinding of upper bisimilarity, but note that it identifies programs that can diverge, and that it does not identify a program that can diverge with one that does not. For example, the program in example 2.4 is identified with $?\mathbb{N}$ by lower similarity and bisimilarity, but not by the upper and convex variants of similarity and bisimilarity.

Lemmas 3.3 and 3.4 state elementary properties of, and relationships between, the different variants.

Lemma 3.3 *Erratic choice is the join operation for $\lesssim_{\text{LS}}$, and the meet operation for $\lesssim_{\text{US}}$ at the computation types, i.e., for all programs of a computation type L, M, N:*

(i) $(L \mathbin{\mathsf{or}} M) \lesssim_{\text{LS}} N$ *if and only if* $L \lesssim_{\text{LS}} N$ *and* $M \lesssim_{\text{LS}} N$.

(ii) $L \lesssim_{\text{US}} (M \mathbin{\mathsf{or}} N)$ *if and only if* $L \lesssim_{\text{US}} M$ *and* $L \lesssim_{\text{US}} N$.

Lemma 3.4 *The following inclusions hold:*

(i) $\lesssim_{\text{CS}} \subseteq \lesssim_{\text{LS}} \cap \lesssim_{\text{US}}$, $\simeq_{\text{CS}} \subseteq \simeq_{\text{LS}} \cap \simeq_{\text{US}}$, *and* $\simeq_{\text{CB}} \subseteq \simeq_{\text{LB}} \cap \simeq_{\text{UB}}$.

(ii) $\simeq_{\text{LB}} \subseteq \simeq_{\text{LS}}$, $\simeq_{\text{UB}} \subseteq \simeq_{\text{US}}$, *and* $\simeq_{\text{CB}} \subseteq \simeq_{\text{CS}}$.

$$\begin{tikzcd}
\lesssim_{\mathrm{LS}} \arrow[dr, dash] \arrow[dd, dash] & & \lesssim_{\mathrm{US}} \arrow[dl, dash] \arrow[dd, dash] \\
& \lesssim_{\mathrm{CS}} \arrow[dd, dash] & \\
\simeq_{\mathrm{LS}} \arrow[dr, dash] \arrow[dd, dash] & & \simeq_{\mathrm{US}} \arrow[dl, dash] \arrow[dd, dash] \\
& \simeq_{\mathrm{CS}} \arrow[dd, dash] & \\
\simeq_{\mathrm{LB}} \arrow[dr, dash] & & \simeq_{\mathrm{UB}} \arrow[dl, dash] \\
& \simeq_{\mathrm{CB}} &
\end{tikzcd}$$

Fig. 4. Inclusions between Similarities and Bisimilarities

Example 3.5 *The following examples demonstrate the strictness of the inclusions of lemma 3.4:*

(i) *For any program M, $(\Omega \,\mathsf{or}\, [[M]])$ and $(\Omega \,\mathsf{or}\, [(\Omega \,\mathsf{or}\, [M])])$ are related by: $(\simeq_{\mathrm{LB}} \cap \simeq_{\mathrm{UB}})$, $(\simeq_{\mathrm{LS}} \cap \simeq_{\mathrm{US}})$, and $(\lesssim_{\mathrm{LS}} \cap \lesssim_{\mathrm{US}})$; but not by: $\lesssim_{\mathrm{CS}}$, $\simeq_{\mathrm{CS}}$, and $\simeq_{\mathrm{CB}}$. From this we derive:*

$$(\Omega \,\mathsf{or}\, [[M]])\ ((\lesssim_{\mathrm{LS}} \cap \lesssim_{\mathrm{US}}) \setminus \lesssim_{\mathrm{CS}})\ (\Omega \,\mathsf{or}\, [(\Omega \,\mathsf{or}\, [M])])$$
$$(\Omega \,\mathsf{or}\, [[M]])\ ((\simeq_{\mathrm{LS}} \cap \simeq_{\mathrm{US}}) \setminus \simeq_{\mathrm{CS}})\ (\Omega \,\mathsf{or}\, [(\Omega \,\mathsf{or}\, [M])])$$
$$(\Omega \,\mathsf{or}\, [[M]])\ ((\simeq_{\mathrm{LB}} \cap \simeq_{\mathrm{UB}}) \setminus \simeq_{\mathrm{CB}})\ (\Omega \,\mathsf{or}\, [(\Omega \,\mathsf{or}\, [M])])$$

(ii) *If $M\ (\lesssim_{\mathrm{LS}} \setminus \lesssim_{\mathrm{LS}}^{\mathrm{op}})\ N$, then $([M] \,\mathsf{or}\, [N])\ (\simeq_{\mathrm{LS}} \setminus \simeq_{\mathrm{LB}})\ [N]$. Similarly, if we have $M\ (\lesssim_{\mathrm{US}} \setminus \lesssim_{\mathrm{US}}^{\mathrm{op}})\ N$, then $[M]\ (\simeq_{\mathrm{US}} \setminus \simeq_{\mathrm{UB}})\ ([M] \,\mathsf{or}\, [N])$. The assignment $M = \Omega$ and $N = [\star]$ satisfies both of the hypotheses. Finally, if $L\ (\lesssim_{\mathrm{CS}} \setminus \lesssim_{\mathrm{CS}}^{\mathrm{op}})\ M\ (\lesssim_{\mathrm{CS}} \setminus \lesssim_{\mathrm{CS}}^{\mathrm{op}})\ N$, then:*

$$([L] \,\mathsf{or}\, ([M] \,\mathsf{or}\, [N]))\ (\simeq_{\mathrm{CS}} \setminus \simeq_{\mathrm{CB}})\ ([L] \,\mathsf{or}\, [N])$$

A suitable assignment is: $L = \Omega$, $M = (\Omega \,\mathsf{or}\, [\star])$, and $N = [\star]$.

Figure 4 depicts the relationships between the similarities and bisimilarities described in lemma 3.4 and example 3.5. Every edge denotes a strict inclusion.

As previously stated, the similarities and bisimilarities collapse to a similarity preorder and a bisimilarity equivalence respectively if we remove $?\mathbb{N}$ from the programming language. It is easy to construct programs, see example 3.6, that demonstrate that the introduction of finite non-determinism is not conservative for any of the similarities and bisimilarities. Perhaps more surprising is that the upper and convex variants of similarity and bisimilarity are not conservatively extended when finite non-determinism is extended to countable non-determinism. This is discussed in example 3.7.

Example 3.6 *The following programs cannot be distinguished by application* *to deterministic programs:*

$$\vdash \lambda x : \mathsf{P}(\mathsf{nat}).\, \mathsf{let}\, y \Leftarrow x\, \mathsf{in}\, [\mathsf{plus}\,(y,y)] : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{nat})$$
$$\vdash \lambda x : \mathsf{P}(\mathsf{nat}).\, \mathsf{let}\, y \Leftarrow x\, \mathsf{in}\, \mathsf{let}\, z \Leftarrow x\, \mathsf{in}\, [\mathsf{plus}\,(y,z)] : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{nat})$$

They can be distinguished by applying them to a non-deterministic program such as $(0\, \mathsf{or}\, 1)$*, in which case the second program may converge to* $[\mathsf{plus}\,(0,1)]$.

Example 3.7 *The following programs cannot be distinguished by application to finitely non-deterministic programs:*

$$\vdash \lambda^v x.\, [\star] : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{unit})$$
$$\vdash f\, 0 : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{unit})$$
$$\textit{where } f\, x\, y \stackrel{\text{def}}{=} \mathsf{let}\, z \Leftarrow y\, \mathsf{in}\, \mathsf{if}\, (\mathsf{leq}\,(z,x))\, \mathsf{then}\, [\star]\, \mathsf{else}\, f\, z\, y$$

(the definition of f is intended to be formalised as in example 2.2). The programs can be distinguished by the upper and convex similarities and bisimilarities by applying them to $?\mathbb{N}$*. The first program is a strict constant function. The second program has only one convergent behaviour, will fail to terminate if its argument does, but, in addition, may diverge if it is possible to read an infinite, strictly increasing sequence of numbers from its argument.*

The similarity and bisimilarity relations extend in a standard way to relations on arbitrary typed terms by open extension. In general, the open extension of a relation on programs $\mathcal{R}$, denoted $\mathcal{R}^\circ$, relates typed terms $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$ if $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and

$$M[L_1/x_1] \ldots [L_n/x_n] \;\mathcal{R}\; N[L_1/x_1] \ldots [L_n/x_n]$$

for all $L_1 \in \mathit{Exp}_{\tau_1}, \ldots, L_n \in \mathit{Exp}_{\tau_n}$.

4 Compatibility

In this section we sketch a proof that the open extensions of the similarities and bisimilarities of section 3 are compatible for the programming language. A relation $\mathcal{R}$ is compatible for a language if it is preserved by every constructor θ of the language, that is, $\mathcal{R}$ is closed under the rule:

$$\frac{M_1\, \mathcal{R}\, N_1 \;\ldots\; M_n\, \mathcal{R}\, N_n}{\theta(M_1, \ldots, M_n)\; \mathcal{R}\; \theta(N_1, \ldots, N_n)}$$

where the arity of θ is n. Compatibility is of fundamental importance because it is a prerequisite for compositional reasoning.

Howe [11] describes a method using a congruence candidate for proving the compatibility of lower similarity. In later work, Howe [12] and Ong [21] extend the method to convex bisimilarity and convex similarity respectively.

Unfortunately, other methods (see, e.g., [1,25,5]) that have been used to prove compatibility of similarity for deterministic programming languages do not seem to be applicable here: there are difficulties with interpreting ?ℕ in the upper and convex powerdomains, and the methods based on syntactic logical relations use syntactic continuity (see section 5) to establish the fundamental property. Moreover, the compatibility of mutual similarity does not entail the compatibility of bisimilarity for a non-deterministic programming language.

We now sketch Howe and Ong's extension of Howe's congruence candidate method.

(i) The congruence candidate $\mathcal{R}^\bullet$ of a binary relation $\mathcal{R}$ on programs (which will range over the variants of similarity and bisimilarity) is an inductively defined binary relation on (potentially) open, well-typed terms. It is the least relation closed under the following rule, where θ ranges over constructors of the language, including variables, and the arity of θ is n:

$$\frac{L_1\,\mathcal{R}^\bullet\,M_1\;\ldots\;L_n\,\mathcal{R}^\bullet\,M_n\qquad\theta(M_1,\ldots,M_n)\,\mathcal{R}^\circ\,N}{\theta(L_1,\ldots,L_n)\,\mathcal{R}^\bullet\,N}$$

(ii) If $\mathcal{R}$ is a preorder, then the congruence candidate $\mathcal{R}^\circ$ satisfies:
(a) $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.
(b) $\mathcal{R}^\bullet;\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.
(c) $\mathcal{R}^\bullet$ is compatible.
(d) $M_1\,\mathcal{R}^\bullet\,N_1 \wedge M_2\,\mathcal{R}^\bullet\,N_2 \Longrightarrow M_1[M_2/x]\,\mathcal{R}^\bullet\,N_1[N_2/x]$.

(iii) If $\mathcal{R}$ is a preorder, the restriction to programs $\mathcal{R}^\bullet_0$ of the congruence candidate $\mathcal{R}^\bullet$ is a post-fixed point of $\langle\cdot\rangle_{\mathrm{LS}}$ or $\langle\cdot\rangle_{\mathrm{US}}$ if $\mathcal{R}$ is:
(a) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{LS}} \Longrightarrow \mathcal{R}^\bullet_0 \subseteq \langle\mathcal{R}^\bullet_0\rangle_{\mathrm{LS}}$.
(b) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{US}} \Longrightarrow \mathcal{R}^\bullet_0 \subseteq \langle\mathcal{R}^\bullet_0\rangle_{\mathrm{US}}$.
This is established by induction on the derivation of a may convergence judgement for (a), and on a natural number that is derived from a program that cannot diverge for (b)—although a problem is discussed below.

(iv) When $\mathcal{R}$ is lower, upper, or convex similarity, we deduce by co-induction that $\mathcal{R}^\bullet_0 \subseteq \mathcal{R}$, and thus $\mathcal{R}^\bullet = \mathcal{R}^\circ$. Consequently, the open extensions of lower, upper, and convex similarity are compatible, because the respective congruence candidates are. Compatibility of the mutual similarities follows immediately.

(v) The final step is to deduce that each of the bisimilarities are compatible. If $\mathcal{R}$ is an equivalence, it can be shown using induction that $\mathcal{R}^\bullet \subseteq \mathcal{R}^{\bullet+\mathrm{op}}$, where $\mathcal{R}^{\bullet+}$ denotes the transitive closure of the congruence candidate of $\mathcal{R}$, which is compatible by an easy induction. Hence, $\mathcal{R}^{\bullet+} \subseteq \mathcal{R}^{\bullet+\mathrm{op}}$, so $\mathcal{R}^{\bullet+}$ is symmetric. In addition, we can derive from (iii) that:
(a) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{LS}} \Longrightarrow \mathcal{R}^{\bullet+}_0 \subseteq \langle\mathcal{R}^\bullet_0\rangle^+_{\mathrm{LS}} \subseteq \langle\mathcal{R}^{\bullet+}_0\rangle_{\mathrm{LS}}$.
(b) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{US}} \Longrightarrow \mathcal{R}^{\bullet+}_0 \subseteq \langle\mathcal{R}^\bullet_0\rangle^+_{\mathrm{US}} \subseteq \langle\mathcal{R}^{\bullet+}_0\rangle_{\mathrm{US}}$.
As in (iv), co-induction can be used to show that $\mathcal{R}^{\bullet+}$ coincides with $\mathcal{R}$,

$$K \Downarrow^{\text{must}} \quad (K \in \mathit{Can})$$

$$\frac{M \Downarrow^{\text{must}}}{\mathsf{uop}\,(M) \Downarrow^{\text{must}}} \qquad \frac{M \Downarrow^{\text{must}} \qquad N \Downarrow^{\text{must}}}{\mathsf{bop}\,(M, N) \Downarrow^{\text{must}}}$$

$$\frac{L \Downarrow^{\text{must}} \qquad M \Downarrow^{\text{must}}}{\mathsf{if}\,L\,\mathsf{then}\,M\,\mathsf{else}\,N \Downarrow^{\text{must}}} \quad (L \Downarrow^{\text{may}} n+1 \text{ and } n \in \mathbb{N})$$

$$\frac{L \Downarrow^{\text{must}} \qquad N \Downarrow^{\text{must}}}{\mathsf{if}\,L\,\mathsf{then}\,M\,\mathsf{else}\,N \Downarrow^{\text{must}}} \quad (L \Downarrow^{\text{may}} 0)$$

$$\frac{M \Downarrow^{\text{must}} \qquad M_1[N/x] \Downarrow^{\text{must}}}{M\,N \Downarrow^{\text{must}}} \quad (M \Downarrow^{\text{may}} \lambda x.\, M_1)$$

$$\frac{M \Downarrow^{\text{must}} \qquad \{N[M_1/x] \Downarrow^{\text{must}} \mid M \Downarrow^{\text{may}} [M_1]\}}{\mathsf{let}\,x \Leftarrow M\,\mathsf{in}\,N \Downarrow^{\text{must}}}$$

$$\frac{M[\mathsf{fix}\,x.\,M/x] \Downarrow^{\text{must}}}{\mathsf{fix}\,x.\,M \Downarrow^{\text{must}}} \qquad ?\mathbb{N} \Downarrow^{\text{must}}$$

Fig. 5. Must Convergence

when $\mathcal{R}$ is lower, upper, or convex bisimilarity. Therefore, the bisimilarities are compatible.

It is worth noting that the method also works for recursive types and in the absence of types, and that the use of the computational lambda-calculus means that we do not need to use disjoint sets of call-by-name and call-by-value variables as in [12,21].

However, we have glossed over a problem with (iii)(b). Howe and Ong assigned natural numbers to programs that cannot diverge *and* that have only finitely many convergent behaviours. For this reason, their proofs only hold for programming languages with finite non-determinism.

The method can be extended to a language with countable non-determinism by using induction on the derivation of a must convergence judgement. The rules for must convergence $\Downarrow^{\text{must}}$ appear in figure 5. Using induction on these rules, the proof works smoothly for both finite and countable non-determinism. The only problem is, how do we know that, for any program M, $M \Uparrow^{\text{may}}$ if and only if $M \Downarrow^{\text{must}}$? This turns out to be trivial, because the complement of the greatest fixed point of a monotone function on a complete boolean lattice is the least fixed point of another monotone function that can be derived from the original function (see [2]), and the must convergence rules in figure 5 are derived in this way from the may divergence rules in figure 3.

Theorem 4.1 *The lower, upper, and convex variants of similarity, mutual similarity, and bisimilarity are compatible.*

5 Convergence and Continuity

This section describes unwinding properties of recursive programs with respect to may and must convergence, and examines related syntactic continuity properties of the lower and upper similarities. The first part covers may convergence and lower similarity, and the second part covers must convergence and upper similarity. The latter includes an analysis of the heights, measured by ordinals, of derivation trees associated to must convergence judgements.

Well-typed terms of the form $\mathsf{fix}\, x.\, M$, henceforth called fixed point expressions, satisfy a finite unwinding property with respect to may convergence: for any fixed point expression $\mathsf{fix}\, x.\, M$, let $\mathsf{fix}^{(n)} x.\, M$ denote the n'th unwinding, defined inductively by:

$$\begin{aligned} \mathsf{fix}^{(0)} x.\, M &\stackrel{\text{def}}{=} \Omega \\ \mathsf{fix}^{(n+1)} x.\, M &\stackrel{\text{def}}{=} M[\mathsf{fix}^{(n)} x.\, M/x] \end{aligned}$$

Then, whenever $x : \mathsf{P}(\sigma) \vdash M : \mathsf{P}(\sigma)$ and $x : \mathsf{P}(\sigma) \vdash N : \tau$,

$$N[\mathsf{fix}\, x.\, M/x] \Downarrow^{\text{may}} \text{ if and only if } \exists n < \omega.\ N[\mathsf{fix}^{(n)} x.\, M/x] \Downarrow^{\text{may}} \tag{1}$$

where $L \Downarrow^{\text{may}}$ if and only if $\exists K.\, L \Downarrow^{\text{may}} K$. The proof is the same as for deterministic languages (see, e.g., [15,26]).

A related result is a so-called syntactic continuity property of lower similarity on deterministic programs: for terms N and M, as above, and $L \in \mathit{Exp}_\tau$,

$$N[\mathsf{fix}\, x.\, M/x] \lesssim_{\text{LS}} L \text{ if and only if } \forall n < \omega.\ N[\mathsf{fix}^{(n)} x.\, M/x] \lesssim_{\text{LS}} L \tag{2}$$

See [14,26]. But syntactic continuity is not valid, in general, for non-deterministic programs:

Example 5.1 *Recall the program* $M \stackrel{\text{def}}{=} \mathsf{fix}\, x.\, ([0]\ \mathsf{or}\ \mathsf{let}\, y \Leftarrow x\, \mathsf{in}\ [\mathsf{plus}\,(y,1)])$, *from example 2.4. Let* $N \stackrel{\text{def}}{=} \mathsf{let}\, x \Leftarrow ?\mathbb{N}\ \mathsf{in}\, [\mathsf{let}\, y \Leftarrow ?\mathbb{N}\ \mathsf{in}\, [\mathsf{if}\ \mathsf{leq}\,(x,y)\ \mathsf{then}\, x\ \mathsf{else}\, y]]$. *Then, for every finite unwinding* $M^{(n)}$ *of* M, $\left[M^{(n)}\right]$ *is lower similar to* N. *But* $[M]$ *and* N *are not lower similar. (The calculations are straightforward.)*

We now turn our attention to must convergence. First, consider finitely non-deterministic programs where non-determinism only occurs in the form of binary erratic choice. In this case, the derivation trees of the must convergence judgements introduced in section 4 are only finitely branching. As a result, the finite unwinding property of fixed point expressions (1) also holds with respect to must convergence. Moreover, upper similarity satisfies the syntactic continuity property (2) (see [15]).

In general, must convergence derivation trees of programs involving countable choice are countably branching. The complexity of the trees can be measured by assigning ordinals to them in the usual way (a node is assigned the supremum of the successors of the ordinals associated with its children, see, e.g., [20]), and this allows us to give an ordinal bound to the induction used in the proof of theorem 4.1. The bound is simply the supremum of the ordinals associated to the derivations of must convergence judgements. Following work of Apt and Plotkin [3], the bound turns out to be ω_1^{CK}, the least non-recursive ordinal. We recall the definition of recursive ordinals below, but refer the reader to [29,20] for detailed accounts of the recursive ordinals.

Definition 5.2 An ordinal α is recursive if there exists a decidable order on the natural numbers that is order-isomorphic to α.

We first demonstrate that for each recursive ordinal α there is a program that cannot diverge, and that has a must convergence derivation tree with height at least α. Since α is a recursive ordinal, and it can be verified that every partial recursive function can be defined in the programming language, there is a program $M_\alpha : \mathsf{nat} \to \mathsf{nat} \to \mathsf{P}(\mathsf{nat})$ that does not diverge on any input, and the relation that it represents is order-isomorphic to α. Now we also need to construct a program slow that accepts as arguments a program representing an order on natural numbers, and a natural number. It then "counts down" from the given number until it reaches a minimal element, at which point it converges to $[\star]$. The type of the program is:

$$\vdash \mathsf{slow} : (\mathsf{nat} \to \mathsf{nat} \to \mathsf{P}(\mathsf{nat})) \to \mathsf{nat} \to \mathsf{P}(\mathsf{unit})$$

It is intended that the numeric argument, say n, is the code, with respect to the coding used by M_α, of an ordinal $\beta < \alpha$, and that the height of the derivation tree of $(\mathsf{slow}\, M_\alpha\, n) \Downarrow^{\mathrm{must}}$ is at least β. Intuitively, the must convergence derivation tree for this program should contain as sub-trees the derivation trees for $(\mathsf{slow}\, M_\alpha\, m) \Downarrow^{\mathrm{must}}$, where m codes an ordinal strictly less than β. The expressive power of $?\mathbb{N}$ can be used to do this: by choosing any natural number we are choosing the code of any ordinal strictly less than α. The decidability of the order on codes of ordinals strictly less than α allows us to then discard codes of ordinals that are greater than or equal to β. The following definition accomplishes this:

$$\mathsf{slow}\, f\, x \stackrel{\mathrm{def}}{=} \mathsf{let}\, y \Leftarrow ?\mathbb{N}\ \mathsf{in}\ \mathsf{let}\, z \Leftarrow f\, y\, x\ \mathsf{in}\ \mathsf{if}\ z\ \mathsf{then}\ (\mathsf{slow}\, f\, y)\ \mathsf{else}\ [\star]$$

Then, for each recursive ordinal α represented by M_α, we can define a program with a must convergence derivation tree of height α:

$$\mathsf{let}\, x \Leftarrow ?\mathbb{N}\ \mathsf{in}\ \mathsf{slow}\, M_\alpha\, x \tag{3}$$

In the other direction we have to show that the ordinal height of a must convergence derivation tree is always recursive. Suppose that M is a program that must converge, and that has a derivation tree with height α. The ordinals strictly less than α are represented by paths in the tree that start at the root of the tree, i.e., at M, together with annotations for the may convergence side-conditions. With the side conditions given, it is decidable whether an arbitrary path is a valid path from M by checking each component of the path against the rule schema of figure 5. With a suitable encoding of paths in the tree as sequences of natural numbers, the derivation tree of $M \Downarrow^{\mathrm{must}}$ is a recursive tree, and then the Kleene-Brouwer order on paths of the tree is both decidable and order-isomorphic to an ordinal greater than or equal to α. We refer the reader to [20] for the definition of recursive trees and the Kleene-Brouwer order.

In general, fixed point expressions in countably non-deterministic programs do not satisfy a finite unwinding property with respect to must convergence, because of the possibly transfinite heights of derivation trees; and the syntactic continuity property of upper similarity is invalid. For instance, if $\alpha \geq \omega$, the program (3) is a counterexample to both the finite unwinding and syntactic continuity properties. It is, however, possible to formulate an unwinding property for must convergence that holds for countable non-determinism by progressing to transfinite unwindings:

$$N[\mathsf{fix}\, x.\, M/x] \Downarrow^{\mathrm{must}} \text{ if and only if } \exists \alpha < \omega_1^{\mathrm{CK}}.\ N[\mathsf{fix}^{(\alpha)} x.\, M/x] \Downarrow^{\mathrm{must}} \quad (4)$$

In order to make sense of this assertion, we need to define the transfinite unwindings and their must convergence behaviour. We extend the syntax with new terms $\mathsf{fix}^{(\lambda)} x.\, M$, for all recursive limit ordinals λ, with the same typing rule as for ordinary fixed point expressions. Arbitrary recursive unwindings $\mathsf{fix}^{(\alpha)} x.\, M$, for $\alpha < \omega_1^{\mathrm{CK}}$, are defined if we let $\mathsf{fix}^{(0)} x.\, M \stackrel{\mathrm{def}}{=} \Omega$, as above, and, inductively, $\mathsf{fix}^{(\alpha+1)} x.\, M \stackrel{\mathrm{def}}{=} M[\mathsf{fix}^{(\alpha)} x.\, M/x]$, for all $\alpha < \omega_1^{\mathrm{CK}}$. Next, the definition of must convergence has to be extended to the new terms. Intuitively, we want the following rule which expresses that the must convergence at recursive limit ordinals is the best of all the must convergence behaviours at smaller ordinals:

$$\frac{\mathsf{fix}^{(\alpha)} x.\, M \Downarrow^{\mathrm{must}}}{\mathsf{fix}^{(\lambda)} x.\, M \Downarrow^{\mathrm{must}}} \quad (\alpha < \lambda < \omega_1^{\mathrm{CK}})$$

But, since the definition of must convergence in figure 5 depends on may convergence, it would be necessary to extend the may convergence relation on terms of computation type to the new terms as well, and it it is not clear how to do this. We get around this obstacle by giving a self-contained definition of must convergence at computation types without reference to may convergence. This can be achieved by means of either a "structurally inductive" definition of the must convergence predicate, $M \Downarrow^{\mathrm{must}}$, in the style of Pitts [24] or an inductively defined must convergence relation, $M \Downarrow^{\mathrm{must}} \mathcal{U}$, between terms M and sets of canonical terms $\mathcal{U}$. We sketch the second solution here. If M is a term in the original language, the meaning of $M \Downarrow^{\mathrm{must}} \mathcal{U}$ is that M must converge and that $\mathcal{U}$ is the set of canonical terms that M may converge to.

For example,

$$K \Downarrow^{\mathrm{must}} \{K\} \quad (K \in \mathit{Can}) \qquad\qquad ?\mathbb{N} \Downarrow^{\mathrm{must}} \{[n] \,|\, n \in \mathbb{N}\}$$

A rule for let can be given without reference to may convergence:

$$\frac{M \Downarrow^{\mathrm{must}} \mathcal{U} \qquad \{N[M_1/x] \Downarrow^{\mathrm{must}} \mathcal{V}_{M_1} \,|\, [M_1] \in \mathcal{U}\}}{\mathsf{let}\, x \Leftarrow M \,\mathsf{in}\, N \Downarrow^{\mathrm{must}} \bigcup \{\mathcal{V}_{M_1} \,|\, [M_1] \in \mathcal{U}\}}$$

The remaining rules are straightforward and make no reference to may convergence at computation types. The must convergence relation is extended to the new terms for transfinite unwindings of fixed point expressions by the rule:

$$\frac{\mathsf{fix}^{(\alpha)} x.\, M \Downarrow^{\mathrm{must}} \mathcal{U}}{\mathsf{fix}^{(\lambda)} x.\, M \Downarrow^{\mathrm{must}} \mathcal{U}} \quad (\alpha < \lambda < \omega_1^{\mathrm{CK}})$$

The analysis of the definition of the must convergence predicate in figure 5 shows that the closure ordinal of the rules for the must convergence relation is also ω_1^{CK}. The must convergence predicate, $M \Downarrow^{\mathrm{must}}$, is obtained from the must convergence relation as $M \Downarrow^{\mathrm{must}} \stackrel{\mathrm{def}}{\Leftrightarrow} \exists \mathcal{U}.\, M \Downarrow^{\mathrm{must}} \mathcal{U}$. This concludes the definition of the must convergence behaviour of the transfinite unwindings of fixed point expressions. The proof of (4) is by induction on the derivation of the must convergence judgment.

The definition of the upper similarity from section 3 can be extended to programs with occurrences of transfinite unwindings of fixed point expressions, by extending $\langle \mathcal{R} \rangle_{\mathrm{US}}$ to relate programs $M, N \in \mathit{Exp}_\sigma$ at computation types $\sigma = \mathsf{P}(\tau)$ if

$$\forall \mathcal{U}. M \Downarrow^{\mathrm{must}} \mathcal{U} \Longrightarrow (\exists \mathcal{V}. N \Downarrow^{\mathrm{must}} \mathcal{V} \wedge \forall N_1 \in \mathcal{V}. \exists M_1 \in \mathcal{U}. M_1 \,\mathcal{R}\, N_1)$$

The extension of upper similarity is obtained as the greatest fixed point of the extended definition of the function $\langle \cdot \rangle_{\mathrm{US}}$. It is compatible with respect to the extended language. The compatibility proof for upper similarity from section 4 carries over if the induction is now conducted on the derivation of $M \Downarrow^{\mathrm{must}} \mathcal{U}$.

We ask two questions about the extension of upper similarity to the extended language. First, is it a conservative extension, i.e., does it include the upper similarity relation defined in section 3 for the original language? Second, does it enjoy a transfinite syntactic continuity property? If both are answered affirmatively, we get a useful induction principle for reasoning about fixed point expressions with respect to upper similarity in the original language. The two questions are left as open problems.

6 Beyond Countable Choice

We have described two forms of non-determinism: the construct that we have taken as primitive $?\mathbb{N}$, and binary erratic choice. In this section, we outline two other possibilities that have been proposed in the literature.

The first is based on the observation that binary erratic choice has precisely the same expressiveness as a new choice construct $?\{0,1\}$ that may converge to either $[0]$ or $[1]$, but cannot diverge. It is natural to ask whether other forms of non-determinism can be obtained in a similar way, e.g., if X is a non-empty set of natural numbers, then what is the expressiveness of a choice construct $?X$ that may converge to $[n]$, for any $n \in X$, but cannot diverge? It turns out that choice constructs for countably infinite sets of natural numbers are not always equally expressive, because Apt and Plotkin [3] show that exactly the choice constructs for non-empty, recursively enumerable sets of natural numbers can be defined from $?\mathbb{N}$. This suggests that classifying non-deterministic programs by the cardinality of their convergent behaviour is misleading.

However, classification is not the only issue affected by the result of Apt and Plotkin. In the light of example 3.7, it is of interest to know whether the presence of additional forms of non-determinism further alters the upper and convex variants of similarity and bisimilarity. If this is the case, then a denotational model of non-determinism that can interpret sets of natural numbers that are not recursively enumerable will discriminate more than mutual similarity (or bisimilarity) for a programming language with only $?\mathbb{N}$.

In order to study these problems, the programming language given here can be extended with additional choice constructs of the form described above. The proofs of compatibility sketched in section 4 readily extend to more general forms of "erratic" non-determinism [23]. Roscoe [30] studies similar non-deterministic choice constructs in an extension of CSP.

McCarthy's ambiguous choice operator exhibits a very different form of non-determinism. The ambiguous choice of two programs has a natural, fair (also known as dove-tailing) implementation: run both programs in parallel, and return the value of the first to converge. The ambiguous choice of two programs can converge to any value that the programs can converge to, but only diverges when both programs can diverge.

Moran [18] studies a functional programming language extended with ambiguous choice and proves that lower similarity is compatible for the language. An example is given there that shows that convex similarity cannot be compatible in the presence of ambiguous choice. Similar examples can be used to show that upper similarity and bisimilarity also fail to be compatible. However, the compatibility of convex bisimilarity in the presence of ambiguous choice is an open problem. The method described in section 4 is not immediately applicable because it would imply the compatibility of convex similarity.

7 Conclusion

We have defined a simply-typed functional programming language with an operator that can converge to any natural number, and have introduced nine compatible relations on programs. The relations are lower, upper, and convex variants of applicative similarity and bisimilarity. Although some of the relations have been studied individually in the literature, we have emphasised that they can be constructed using only two functions, and that this affords a natural structure to the proofs of compatibility. In addition, we have mapped the inclusions between the relations, and have given characteristic examples of the differences between them.

Although the programming language is based on the computational lambda-calculus and non-determinism is restricted to computation types, the examples can be modified for programming languages with non-determinism at function or product types (with the assumption that convergence is observable at those types). We also note that the mechanism for creating and composing computations in the computational lambda-calculus provides an alternative, with the same expressive power, to using both call-by-name and call-by-value abstractions to control the resolution of non-determinism.

A different, interesting example demonstrates that the upper and convex variants of similarity and bisimilarity are sensitive to whether finite or countable non-determinism is present in the programming language, i.e., countable non-determinism can be used to distinguish programs of function type that cannot be distinguished by finitely non-deterministic programs.

Previous proofs of compatibility have been restricted to languages with finite non-determinism. We have extended them to a programming language with countable non-determinism by using a relationship between least and greatest fixed points in complete boolean lattices to transform a co-inductively defined may divergence predicate into an inductively defined must convergence predicate. The supremum of the ordinal heights of the must convergence derivation trees is the least non-recursive ordinal ω_1^{CK}.

In this paper we have concentrated on operational models based on co-inductively defined similarity and bisimilarity relations. It may be argued that the resulting models are finer-grained than is warranted by reasonable notions of observation. An alternative is to operate with Morris-style contextual approximation preorders and equivalence relations which are naturally defined on the basis of the may and must convergence predicates [13,15]. The compatibility of the similarity and bisimilarity relations considered here implies that they are all contained in corresponding contextual relations. The inclusions are strict, for different reasons [15]. For instance, the failure of syntactic continuity in example 5.1 distinguishes lower similarity from may contextual approximation which does satisfy the syntactic continuity property (2) for arbitrary non-deterministic programs. Lower and upper similarity are used as auxiliary relations in [13] to reason about contextual equivalences for the operationally-defined specification language of action semantics, action notation, which features countable non-determinism.

Acknowledgement

We would like to thank Ralph Loader, Peter Mosses, Luke Ong, Stan Wainer, and especially Andrew Moran for helpful conversations.

References

[1] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, pages 65–117. Addison-Wesley, 1990.

[2] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, number 90 in Studies in Logic. North-Holland Publishing Company, 1977.

[3] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, October 1986.

[4] R. J. Bird and O. de Moor. *The Algebra of Programming.* Prentice Hall, 1997.

[5] L. Birkedal and R. Harper. Operational interpretations of recursive types in an operational setting (summary). In M. Abadi and T. Ito, editors, *Symposium on Theoretical Aspects of Computer Science, Sendai, Japan*, volume 1281 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[6] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretations. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[7] A. D. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[8] A. D. Gordon. Bisimilarity as a theory of functional programming. BRICS Notes Series NS-95-3, Department of Computer Science, University of Aarhus, 1995.

[9] A. D. Gordon. A tutorial on co-induction and functional programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Workshops in Computing, 1995.

[10] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[11] D. J. Howe. Equality in lazy computation systems. In *Proceedings, 4th Annual Symposium on Logic in Computer Science*, pages 198–203. Computer Society Press, Washington, 1989.

[12] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[13] S. B. Lassen. Action semantics reasoning about functional programs. *Math. Struct. in Comp. Science*, pages 557–589, 1997.

[14] S. B. Lassen. Relational reasoning about contexts. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 91–135. Cambridge University Press, 1998.

[15] S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism*. PhD thesis, Department of Computer Science, University of Aarhus, February 1998. URL `http://www.cl.cam.ac.uk/users/sbl21/docs/phd.html`.

[16] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, New York, 1989.

[17] E. Moggi. Notions of computations and monads. *Information and Computation*, 93(1):55–92, 1991.

[18] A. Moran. Natural semantics for non-determinism. Licentiate thesis, Chalmers University of Technology and University of Göteborg, May 1994.

[19] P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[20] P. Odifreddi. *Classical Recursion Theory*, volume 125 of *Studies in Logic*. Elsevier Science Publishers B.V., 1989.

[21] C.-H. L. Ong. Concurrent lambda calculus, and a general pre-congruence theorem for applicative bisimulation. Preliminary version, August 1992.

[22] C.-H. L. Ong. Non-determinism in a functional setting. In *Proceedings, 8th Annual Symposium on Logic in Computer Science*, pages 275–286. Computer Society Press, Washington, 1993.

[23] C. S. Pitcher. *Functional Programming and Erratic Non-Determinism*. PhD thesis, Oxford University Computing Laboratory. In preparation (expected September 1998).

[24] A. M. Pitts. Parametric polymorphism and operational equivalence. Preliminary version. In this volume.

[25] A. M. Pitts. A note on logical relations between semantics and syntax. *Logic Journal of the Interest Group in Pure and Applied Logics*, 5(4):589–601, July 1997.

[26] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1997. Lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, September 1995.

[27] G. D. Plotkin. Domains. URL `http://hypatia.dcs.qmw.ac.uk/sites/other/domain.notes.other/`, 1983.

[28] S. Prasad, A. Giacalone, and P. Mishra. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In M. S. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 765–780. Springer-Verlag, 1990.

[29] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Series in Higher Mathematics. McGraw-Hill, 1967.

[30] A. W. Roscoe. Two papers on CSP. Technical Report PRG-67, Programming Resarch Group, Oxford University Computing Laboratory, July 1988. (An alternative order for the failures model & Unbounded nondeterminism in CSP).