Electronic Notes in Theoretical Computer Science 1 (1995)
URL: http://www.elsevier.nl/locate/entcs/volume1.html 21 pages

Bisimilarity as a Theory of Functional Programming

Andrew D. Gordon

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

Abstract

Morris-style contextual equivalence—invariance of termination under any context of ground type—is the usual notion of operational equivalence for deterministic functional languages such as FPC (PCF plus sums, products and recursive types). Contextual equivalence is hard to establish directly. Instead we define a labelled transition system for call-by-name FPC (and variants) and prove that CCS-style bisimilarity equals contextual equivalence—a form of operational extensionality. Using co-induction we establish equational laws for FPC. By considering variations of Milner's 'bisimulations up to $\sim$' we obtain a second co-inductive characterisation of contextual equivalence in terms of reduction behaviour and production of values. Hence we use co-inductive proofs to establish contextual equivalence in a series of stream-processing examples. Finally, we consider a form of Milner's original context lemma for FPC, but conclude that our form of bisimilarity supports simpler co-inductive proofs.

1 Objectives

The object of this paper is to offer a new perspective on the behaviour of functional programs based on CCS-style labelled transitions and bisimilarity.

Morris-style contextual equivalence is widely accepted as the natural notion of operational equivalence for PCF-like languages [24]. Two programs are contextually equivalent if they may be interchanged for one another in any larger program of integer type, without affecting whether evaluation of the whole program converges or not. The quantification over program contexts makes contextual equivalence hard to prove directly. One approach to this difficulty is to characterise contextual equivalence independently of the syntax and operational semantics of PCF. This is the 'full abstraction' problem for PCF; see Ong [18] for a discussion and review of the literature.

Instead, our approach is to characterise contextual equivalence as a form of bisimilarity, and to exploit operationally-based co-inductive proofs. Our point of departure is Milner's [16] entirely operational theory of CCS, based on labelled transitions and bisimilarity. A labelled transition takes the form

© 1995 Elsevier Science B. V. Open access under CC BY-NC-ND license.

$a \xrightarrow{\alpha} b$, where a and b are programs, and α is an *action*; the intended meaning of such a transition is that the atomic observation α can be made of program a to yield a successor b. In CCS, the actions represent possible communications. Given a definition of the possible labelled transitions for a language, any program gives rise to a (possibly infinite) *derivation tree*, whose nodes are programs and whose arcs are transitions, labelled by actions. Bisimilarity is based on the intuition that a derivation tree represents the behaviour of a program. We say two programs are *bisimilar* if their derivation trees are the same when one ignores the syntactic structure at the nodes. Hence bisimilarity is a way to compare behaviour, represented by actions, whilst discarding syntactic structure. Park [19] showed how bisimilarity could be defined co-inductively; the theory of CCS is heavily dependent on proofs by co-induction.

Bisimilarity has been applied to deterministic functional programming before, notably by Abramsky in his study of applicative bisimulation and lazy lambda-calculus [1] and by Howe [11], who invented a powerful method of showing that bisimilarity is a congruence. Both showed that their untyped forms of bisimilarity equalled contextual equivalence—a property known as *operational extensionality* [4]. If Ω is a divergent lambda-term, both these untyped formulations of bisimilarity distinguish $\lambda x.\,\Omega$ from Ω, because one converges and the other diverges. But in a typed call-by-name setting, contextual equivalence would identify these two functions, because they have the same behaviour on all arguments. Hence Turner [29, Preface] expressed concern that applicative bisimulation would fail to be operationally extensional for languages such as Miranda or Haskell.

We use Gunter's [10] FPC (PCF plus sums, products and recursive types; see Winskel [31] for a similar language) as the vehicle for this study. Our first main contribution is to answer Turner's concern by showing that by defining a labelled transition system for FPC and then defining bisimilarity exactly as in CCS, we obtain operational extensionality for call-by-name, call-by-name plus convergence testing, and call-by-value variants of FPC. In particular, in the call-by-name variant we have $\Omega^{A\to B}$ bisimilar to $\lambda x{:}A.\,\Omega^B$. Our second contribution is to investigate how operational methods developed in the theory of CCS apply to (deterministic) functional programming. We consider various refinements of co-induction, analogous to the idea of 'bisimulation up to $\sim$' in CCS. In particular, by taking advantage of determinism, we obtain a new co-inductive characterisation of contextual equivalence based on reduction behaviour and production of values.

Before Park's invention of bisimilarity, Milner [15] developed operational methods for proving contextual equivalence based on his context lemma for (combinatory) PCF. Our third contribution is to prove a generalisation of the context lemma for FPC, and show how it gives rise to another co-inductive characterisation of contextual equivalence. However we suggest that in a certain sense it is less useful than bisimilarity.

We begin by recalling the dual foundations of induction and co-induction in Section 2. We introduce the syntax and operational semantics of FPC and PCF in Section 3. Section 4 is the heart of the paper in which we define a

labelled transition system for call-by-name FPC, and replay the definition of bisimilarity from CCS. We prove that bisimilarity equals contextual equivalence, and develop an equational theory. We prove that bisimilarity is a congruence in Section 5, by adapting Howe's method. We derive a range of co-inductive characterisations of bisimilarity in Section 6, motivated by a collection of stream-processing examples. In Section 7 we generalise Milner's context lemma to FPC, to yield another co-inductive form of contextual equivalence. We sketch several variations of FPC in Section 8 and discuss related work and the significance of our results in Section 9.

2 Induction and Co-induction

We briefly recall how induction and co-induction principles derive from the Tarski-Knaster fixpoint theorem. Aczel [2] and Davey and Priestley [7] are good references. Let U be some universal set and $F : \wp(U) \to \wp(U)$ be a monotone function (that is, $F(X) \subseteq F(Y)$ whenever $X \subseteq Y$). We say a set $X \subseteq U$ is *F-closed* iff $F(X) \subseteq X$. Dually, a set $X \subseteq U$ is *F-dense* iff $X \subseteq F(X)$. A *fixpoint* of F is a solution of the equation $X = F(X)$. Let $\mu X.\, F(X)$ and $\nu X.\, F(X)$ be the following subsets of U.

$$\mu X.\, F(X) \stackrel{\text{def}}{=} \bigcap\{X \mid F(X) \subseteq X\}$$

$$\nu X.\, F(X) \stackrel{\text{def}}{=} \bigcup\{X \mid X \subseteq F(X)\}$$

Theorem 2.1 (Tarski-Knaster)

(1) *$\mu X.\, F(X)$ is the least fixpoint of F.*

(2) *$\nu X.\, F(X)$ is the greatest fixpoint of F.* □

We say that $\mu X.\, F(X)$, the least solution of $X = F(X)$, is the set *inductively defined* by F, and dually, that $\nu X.\, F(X)$, the greatest solution of $X = F(X)$, is the set *co-inductively defined* by F. We obtain two dual proof principles associated with these definitions.

Induction: $\mu X.\, F(X) \subseteq X$ if X is F-closed.

Co-induction: $X \subseteq \nu X.\, F(X)$ if X is F-dense.

Winskel [31], for instance, explains how structural and rule induction follow from this basic induction principle. Here we use co-induction extensively.

3 PCF and FPC

In this section we introduce two call-by-name languages: PCF—simply typed lambda-calculus plus arithmetic and recursion—and FPC—an extension of PCF with products, sums and recursive types. We define syntax, type assignment, a 'one-step' reduction relation, $\rightsquigarrow$, and a corresponding 'many-step' evaluation relation, $\Downarrow$.

Let X, Y, Z range over a countable set of *type variables*, and x, y, z over a countable set of (program) *variables*. The *type expressions*, E, and (program)

expressions, e, of PCF are given by the grammars

$$\begin{aligned} E &::= \mathtt{Num} \mid \mathtt{Bool} \mid E \to E \\ e &::= \underline{n} \mid \mathtt{succ}(e) \mid \mathtt{pred}(e) \mid \underline{bv} \mid \mathtt{zero}(e) \mid \mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e \\ &\mid \lambda x{:}E.\, e \mid e\, e \mid \mathtt{rec}\, x{:}E.\, e \end{aligned}$$

where $n \in \mathbb{N}$ and $bv \in \{tt, ff\}$. FPC is the PCF language extended with the following kinds of type and program expressions.

$$\begin{aligned} E &::= \mathtt{Unit} \mid E \times E \mid E + E \mid X \mid \mathtt{rec}\, X.\, E \\ e &::= \mathtt{unity} \mid (e, e) \mid \mathtt{split}\, e\, \mathtt{as}\, (x, x)\, \mathtt{in}\, e \\ &\mid \mathtt{inl}[E + E](e) \mid \mathtt{inr}[E + E](e) \\ &\mid \mathtt{case}\, e\, \mathtt{of}\, \mathtt{inl}(x) \Rightarrow e\, \mathtt{or}\, \mathtt{inr}(x) \Rightarrow e \\ &\mid \mathtt{intro}[\mathtt{rec}\, X.\, E](e) \mid \mathtt{elim}[\mathtt{rec}\, X.\, E](e) \end{aligned}$$

We identify (type and program) expressions up to alpha-conversion, that is, consistent renaming of bound variables. We write $e[{}^{e'}\!/_{x}]$ for the substitution of expression e' for each variable x free in expression e. Similarly $E[{}^{E'}\!/_{X}]$ denotes substitution of a type expression for a type variable. We write $fv(e)$ and $ftv(E)$ for the sets of program and type variables free in e and E, respectively. We often omit type information when writing program expressions.

Let a *type*, A or B, be a closed type expression. The *type assignment* relation is of the form $\Gamma \vdash e : A$ where Γ is an *environment*, a finite map from variables to types. If $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$, we write $Dom(\Gamma)$ for the domain of Γ, that is, $\{x_1, \ldots, x_n\}$. We write $\varnothing$ for the empty environment. We omit the type assignment rules, but they are similar to those in Gunter's book [10]. Given the type assignment relation, we can construct the following universal sets and relations.

$$\begin{aligned} Prog(A) &\stackrel{\text{def}}{=} \{e \mid \varnothing \vdash e : A\} \\ a, b \in Prog &\stackrel{\text{def}}{=} \textstyle\bigcup_{A \in Type} Prog(A) \\ Rel(A) &\stackrel{\text{def}}{=} \{(a, b) \mid a \in Prog(A)\ \&\ b \in Prog(A)\} \\ \mathcal{R}, \mathcal{S} \subseteq Rel &\stackrel{\text{def}}{=} \textstyle\bigcup_{A \in Type} Rel(A) \end{aligned}$$

If A is a type, $Prog(A)$ is the set of *programs* of type A, that is, closed, well-formed program expressions. $Prog$ is the set of programs of arbitrary type, ranged over by a and b. The type of each program is unique. We shall write $a_1, \ldots, a_n{:}A$ to mean $\{a_1, \ldots, a_n\} \subseteq Prog(A)$. If A is a type, $Rel(A)$ is the universal (total) relation between programs of type A, and Rel is the universal relation between programs of the same arbitrary type. We typically use $\mathcal{R}$ and $\mathcal{S}$ to denote arbitrary relations between programs of the same type.

The operational semantics is a one-step *reduction* relation, $\rightsquigarrow \subseteq Rel$. It is inductively defined by the axiom schemes in Table 1 closed under the structural rule that $\mathcal{E}[a] \rightsquigarrow \mathcal{E}[a']$ if $a \rightsquigarrow a'$ where $\mathcal{E}$ is an *experiment* (a kind of atomic *evaluation context* [8]), a context generated by the grammar

$$\texttt{if}\ \underline{bv}\ \texttt{then}\ a_{tt}\ \texttt{else}\ a_{ff} \rightsquigarrow a_{bv} \qquad \texttt{succ}(\underline{n}) \rightsquigarrow \underline{n+1}$$

$$\texttt{pred}(\underline{n}) \rightsquigarrow \begin{cases} \underline{0} & \text{if } n = 0 \\ \underline{n-1} & \text{otherwise} \end{cases} \qquad \texttt{zero}(\underline{n}) \rightsquigarrow \begin{cases} \underline{tt} \text{ if } n = 0 \\ \underline{ff} \text{ otherwise} \end{cases}$$

$$(\lambda x.\, e)\, a \rightsquigarrow e[a/x] \qquad \texttt{split}(a,b)\ \texttt{as}\ (x,y)\ \texttt{in}\ e \rightsquigarrow e[a,b/x,y]$$

$$(\texttt{case}\ \texttt{inl}(a)\ \texttt{of}\ \texttt{inl}(x_1) \Rightarrow e_1\ \texttt{or}\ \texttt{inr}(x_2) \Rightarrow e_2) \rightsquigarrow e_1[a/x_1]$$

$$(\texttt{case}\ \texttt{inr}(a)\ \texttt{of}\ \texttt{inl}(x_1) \Rightarrow e_1\ \texttt{or}\ \texttt{inr}(x_2) \Rightarrow e_2) \rightsquigarrow e_2[a/x_2]$$

$$(\texttt{rec}\, x.\, e) \rightsquigarrow e[^{\texttt{rec}\, x.\, e}/_x] \qquad \texttt{elim}(\texttt{intro}(a)) \rightsquigarrow a$$

Table 1 Axiom schemes for reduction

$$\begin{aligned} \mathcal{E} ::= {} & \texttt{succ}([\,]) \mid \texttt{pred}([\,]) \mid \texttt{zero}([\,]) \mid \texttt{if}\ [\,]\ \texttt{then}\ b_1\ \texttt{else}\ b_2 \mid [\,]\, b \\ \mid {} & \texttt{split}\ [\,]\ \texttt{as}\ (x,y)\ \texttt{in}\ e \mid \texttt{elim}([\,]) \\ \mid {} & \texttt{case}\ [\,]\ \texttt{of}\ \texttt{inl}(x_1) \Rightarrow e_1\ \texttt{or}\ \texttt{inr}(x_2) \Rightarrow e_2 \end{aligned}$$

Our choice of experiments gives rise to a deterministic, call-by-name evaluation strategy. We sketch call-by-value and other variations in Section 8. We define the usual notions of evaluation, convergence and divergence as follows.

$a \rightsquigarrow \stackrel{\text{def}}{=} \exists b(a \rightsquigarrow b)$	'a reduces'
$a \Downarrow b \stackrel{\text{def}}{=} a \rightsquigarrow^* b\ \&\ \neg(b \rightsquigarrow)$	'a evaluates to b'
$a\!\Downarrow \stackrel{\text{def}}{=} \exists b(a \Downarrow b)$	'a converges'
$a\!\Uparrow \stackrel{\text{def}}{=}$ whenever $a \rightsquigarrow^* b$, $b \rightsquigarrow$	'a diverges'

By expanding the definition we can easily check (in this deterministic setting) that $\Downarrow$ and $\Uparrow$ are complementary, that is, $a\!\Uparrow$ iff $\neg(a\!\Downarrow)$. There is a divergent term at every type. Define $\Omega^A \stackrel{\text{def}}{=} \texttt{rec}\, x{:}A.\, x$. We have $\Omega^A \rightsquigarrow \Omega^A$ and hence $\Omega^A\!\Uparrow$. Let the set of *values*, ranged over by u and v, be the set of programs generated by the following grammar.

$$v ::= \underline{\ell} \mid \lambda x.\, e \mid \texttt{unity} \mid (a,b) \mid \texttt{inl}(a) \mid \texttt{inr}(a) \mid \texttt{intro}(a)$$

It is not hard to check that a program a is a value iff it is $\rightsquigarrow$-normal, that is, that $\neg(a \rightsquigarrow)$. Hence the set of values is exactly the image of the evaluation relation, that is, $\{b \mid \exists a(a \Downarrow b)\}$.

Now we can define a form of Morris' contextual equivalence [17]. Let a *context*, $\mathcal{C}$, be a program expression possibly containing holes, each written as $[\,]$. Contexts are not identified up to alpha-conversion. *Contextual equivalence*, $\simeq \subseteq Rel$ is given by:

$$a \simeq b \text{ iff whenever } \mathcal{C}[a], \mathcal{C}[b]{:}\texttt{Num},\ \mathcal{C}[a]\!\Downarrow \text{ iff } \mathcal{C}[b]\!\Downarrow.$$

It would be equivalent but less wieldy to formulate contextual equivalence in terms of convergence to a particular integer.

4 Bisimilarity for FPC

We begin with a *labelled transition system* that characterises the immediate observations one can make of a program. It is a family of relations ($\xrightarrow{\alpha} \subseteq Prog \times Prog \mid \alpha \in Act$), indexed by the set Act of *actions*. If we let Lit, the set of *literals*, indexed by ℓ, be $\{tt, ff\} \cup \{0, 1, 2, \ldots\}$, then Act, ranged over by α, is the set

$$Lit \cup \{\mathtt{fst}, \mathtt{snd}, \mathtt{inl}, \mathtt{inr}, \mathtt{elim}\} \cup \{@a \mid a \in Prog\}.$$

We partition the set of types into active and passive types. The intention is that we can directly observe termination of programs of active type, but not those of passive type. Let a type be *active* iff it has the form `Bool`, `Num`, $A \times B$ or $A + B$. Let a type be *passive* iff it has the form `Unit`, $A \to B$ or $\mathtt{rec}\, X.\, E$. We define $\mathbf{0}$ to be some arbitrary divergent term of active type. Given these definitions, the labelled transition system may be defined inductively as follows.

$$\underline{\ell} \xrightarrow{\ell} \mathbf{0}$$

$$(a, b) \xrightarrow{\mathtt{fst}} a \qquad (a, b) \xrightarrow{\mathtt{snd}} b$$

$$\mathtt{inl}(a) \xrightarrow{\mathtt{inl}} a \qquad \mathtt{inr}(a) \xrightarrow{\mathtt{inr}} a$$

$$\frac{a{:}B \to A \qquad b{:}B}{a \xrightarrow{@b} a\, b} \qquad \frac{a{:}\mathtt{rec}\, X.\, E}{a \xrightarrow{\mathtt{elim}} \mathtt{elim}[\mathtt{rec}\, X.\, E](a)}$$

$$\frac{a{:}A \qquad A \text{ active} \qquad a \rightsquigarrow a'' \qquad a'' \xrightarrow{\alpha} a'}{a \xrightarrow{\alpha} a'}$$

The *derivation tree* of a program a is the potentially infinite tree whose nodes are programs, whose arcs are labelled transitions, and which is rooted at a. For instance, if A is an active type, the derivation tree of the combinator Ω^A is empty. In particular, the tree of $\mathbf{0}$ is empty. We use $\mathbf{0}$ in defining the transition system to indicate that after observing the value of a literal there is nothing more to observe. Following Milner [16], we wish to regard two programs as behaviourally equivalent iff their derivation trees are isomorphic when we ignore the syntactic structure of the programs labelling the nodes. We formalise this idea by requiring our behavioural equivalence to be a relation $\sim \subseteq Rel$ that satisfies property $(*)$: whenever $(a, b) \in Rel$, $a \sim b$ iff

(1) whenever $a \xrightarrow{\alpha} a'$ $\exists b'$ with $b \xrightarrow{\alpha} b'$ and $a' \sim b'$;

(2) whenever $b \xrightarrow{\alpha} b'$ $\exists a'$ with $a \xrightarrow{\alpha} a'$ and $a' \sim b'$.

As usual we can characterise this property as being a fixpoint of a certain monotone functional on relations, and then take bisimilarity to be the greatest. If $\mathcal{S} \subseteq Rel$, define $\langle \mathcal{S} \rangle \subseteq Rel$ such that $a \langle \mathcal{S} \rangle b$ iff

(1) whenever $a \xrightarrow{\alpha} a'$ $\exists b'$ with $b \xrightarrow{\alpha} b'$ and $a' \mathcal{S} b'$;

(2) whenever $b \xrightarrow{\alpha} b'$ $\exists a'$ with $a \xrightarrow{\alpha} a'$ and $a' \mathcal{S} b'$.

It is easy to check that function $\langle - \rangle$ is monotone. Let a *bisimulation* be a $\langle - \rangle$-dense relation, and let *bisimilarity*, $\sim \subseteq Rel$, be $\nu \mathcal{S}.\langle \mathcal{S} \rangle$, the greatest bisimulation. Clearly a relation satisfies property $(*)$ iff it is a fixpoint of function $\langle - \rangle$. By definition bisimilarity is such a relation, and indeed is the greatest.

Let *similarity*, $\lesssim$, be the preorder form of $\sim$, that is, the greatest fixpoint of the function obtained by omitting clause (2) of $\langle - \rangle$. We can easily establish the following basic facts.

Lemma 4.1

(1) $\lesssim$ *is a preorder and* $\sim$ *an equivalence relation.*

(2) $a \sim b$ *iff* $a \lesssim b$ *and* $b \lesssim a$.

(3) $\rightsquigarrow \subseteq \sim$ *and hence* $\Downarrow \subseteq \sim$. □

Parts (2) and (3) depend on the determinacy of $\rightsquigarrow$; they would fail, for instance, if we added nondeterministic choice to FPC.

4.1 Operational Extensionality

We have an obligation to show that bisimilarity, $\sim$, equals contextual equivalence, $\simeq$. The key fact we need is the following, that bisimilarity is a congruence.

Theorem 4.2 (Congruence) *If* $a \sim b$ *then* $\mathcal{C}[a] \sim \mathcal{C}[b]$ *for any context* $\mathcal{C}$.

We shall postpone the proof till Section 5. We now have operational extensionality:

Theorem 4.3 $\simeq = \sim$.

Proof. The proof of $\sim \subseteq \simeq$ follows from the congruence of $\sim$. The reverse inclusion follows by co-induction after showing that $\simeq$ is a bisimulation. For full details of a similar proof see Lemma 4.29 of Gordon [9], which was based on Theorem 3 of Howe [11]. If bisimilarity distinguished $\Omega^{A \to B}$ from $\lambda a.\, \Omega^B$ we would be unable to prove that $\simeq$ was a bisimulation. □

4.2 A Theory of Bisimilarity

We have defined bisimilarity as a greatest fixpoint and shown it to be a co-inductive characterisation of contextual equivalence. In this section we shall note without proof various equational properties needed in a theory of functional programming. Proofs of similar properties, but for a different form of bisimilarity, can be found in Gordon [9]. We noted already that $\rightsquigarrow \subseteq \sim$, which justifies a collection of beta laws. We can easily use co-induction to prove the following eta laws for passive types.

Proposition 4.4

(1) *If* $a{:}A \to B$, $a \sim \lambda x{:}A.\, a\, x$.

(2) *If* a:$\mathtt{rec}\, X.\, E$, $a \sim \mathtt{intro}(\mathtt{elim}(a))$. □

We have an unrestricted principle of extensionality.

Proposition 4.5 *Suppose* f, g:$A \to B$. *If* $f\, a \sim g\, a$ *for any* a:A, *then* $f \sim g$. □

Undefinedness propagates through experiments.

Proposition 4.6 $\mathcal{E}[\Omega] \sim \Omega$ *for any experiment* $\mathcal{E}$. □

We have the following adequacy result.

Proposition 4.7 *Suppose* a:A.

(1) *If* A *is active,* $a \sim \Omega^A$ *iff* $a{\Uparrow}$.

(2) *If* A *is passive,* $a \sim \Omega^A$ *if* $a{\Uparrow}$. □

As promised, we can prove that $\lambda x{:}A.\, \Omega^B \simeq \Omega^{A \to B}$, in fact by proving that $\lambda x{:}A.\, \Omega^B \sim \Omega^{A \to B}$. Consider any a:A. We have $(\lambda x{:}A.\, \Omega^B)\, a \sim \Omega^B$ by beta conversion and $\Omega^{A \to B}\, a \sim \Omega^B$ by Proposition 4.6. Hence $\lambda x{:}A.\, \Omega^B \sim \Omega^{A \to B}$ by extensionality. The converse of (2) is false, then, for $\lambda x{:}A.\, \Omega^B \sim \Omega^{A \to B}$ but $\lambda x{:}A.\, \Omega^B{\Downarrow}$.

Subject to the following conditions, every program has a value.

Proposition 4.8 *Suppose* a:A.

(1) *If* A *is active,* $\exists v(a \sim v)$ *iff* $a{\Downarrow}$.

(2) *If* A *is passive,* $\exists v(a \sim v)$ *unconditionally.* □

Finally, the value constructors are injective.

Proposition 4.9

(1) *If* $\underline{\ell} \sim \underline{\ell'}$ *then* $\ell = \ell'$.

(2) *If* $\lambda x{:}A.\, e \sim \lambda x{:}A.\, e'$ *then* $e[a/x] \sim e'[a/x]$ *for any* a:A.

(3) *If* $(a_1, a_2) \sim (b_1, b_2)$ *then* $a_1 \sim b_1$ *and* $a_2 \sim b_2$.

(4) *If* $\mathtt{inl}(a) \sim \mathtt{inl}(b)$ *then* $a \sim b$.

(5) *If* $\mathtt{inr}(a) \sim \mathtt{inr}(b)$ *then* $a \sim b$.

(6) *If* $\mathtt{intro}(a) \sim \mathtt{intro}(b)$ *then* $a \sim b$. □

5 Bisimilarity is a Congruence

In this section we shall sketch a proof that similarity is a precongruence, that is, preserved by arbitrary contexts. Since $\sim$ is the symmetrisation of $\lesssim$, it follows that bisimilarity is a congruence (a precongruence that is an equivalence), Theorem 4.2. Howe [11] originally proved that similarity was a precongruence for a broad class of 'lazy computation systems.' These were untyped and based on an evaluation relation. As in earlier work [6], we recast his proof in a typed setting and using labelled transitions. The proof in this section would not work for a nondeterministic calculus, where $\sim$ does not equal

$$\Gamma \vdash x\ \widehat{\mathcal{R}}\ x \qquad \Gamma \vdash \underline{\ell}\ \widehat{\mathcal{R}}\ \underline{\ell} \qquad \Gamma \vdash \mathtt{unity}\ \widehat{\mathcal{R}}\ \mathtt{unity}$$

$$\frac{\Gamma \vdash e\ \mathcal{R}\ e'}{\Gamma \vdash \kappa(e)\ \widehat{\mathcal{R}}\ \kappa(e')}\ \kappa \in \{\mathtt{succ}, \mathtt{pred}, \mathtt{zero}, \mathtt{inl}, \mathtt{inr}, \mathtt{intro}, \mathtt{elim}\}$$

$$\frac{\Gamma \vdash e_i\ \mathcal{R}\ e_i' \quad (i = 1,2,3)}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3\ \widehat{\mathcal{R}}\ \mathtt{if}\ e_1'\ \mathtt{then}\ e_2'\ \mathtt{else}\ e_3'}$$

$$\frac{\Gamma, x{:}A \vdash e\ \mathcal{R}\ e'}{\Gamma \vdash \lambda x{:}A.\, e\ \widehat{\mathcal{R}}\ \lambda x{:}A.\, e'} \qquad \frac{\Gamma, x{:}A \vdash e\ \mathcal{R}\ e'}{\Gamma \vdash \mathtt{rec}\, x{:}A.\, e\ \widehat{\mathcal{R}}\ \mathtt{rec}\, x{:}A.\, e'}$$

$$\frac{\Gamma \vdash e_1\ \mathcal{R}\ e_1' \qquad \Gamma \vdash e_2\ \mathcal{R}\ e_2'}{\Gamma \vdash (e_1, e_2)\ \widehat{\mathcal{R}}\ (e_1', e_2')}$$

$$\frac{\Gamma \vdash e_1\ \mathcal{R}\ e_1' : A_1 \times A_2 \qquad \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash e_2\ \mathcal{R}\ e_2'}{\Gamma \vdash \mathtt{split}\, e_1\ \mathtt{as}\ (x_1, x_2)\ \mathtt{in}\ e_2\ \widehat{\mathcal{R}}\ \mathtt{split}\, e_1'\ \mathtt{as}\ (x_1, x_2)\ \mathtt{in}\ e_2'}$$

$$\frac{\Gamma \vdash e_0\ \mathcal{R}\ e_0' : A_1 + A_2 \qquad \Gamma, x_i{:}A_i \vdash e_i\ \mathcal{R}\ e_i' \quad (i = 1,2)}{\Gamma \vdash \begin{pmatrix} \mathtt{case}\, e_0 \mathtt{of} \\ \mathtt{inl}(x_1) \Rightarrow e_1 \mathtt{or} \\ \mathtt{inr}(x_2) \Rightarrow e_2 \end{pmatrix}\ \widehat{\mathcal{R}}\ \begin{pmatrix} \mathtt{case}\, e_0' \mathtt{of} \\ \mathtt{inl}(x_1) \Rightarrow e_1' \mathtt{or} \\ \mathtt{inr}(x_2) \Rightarrow e_2' \end{pmatrix}}$$

Table 2 The compatible refinement of a relation

mutual similarity, that is, the symmetrisation of $\lesssim$. Howe [12] has recently shown how his method can be applied directly to bisimilarity, and hence is applicable to nondeterministic languages.

We need to extend relations such as bisimilarity to open expressions rather than simply programs. Let a *proved expression* be a triple (Γ, e, A) such that $\Gamma \vdash e : A$. If $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$, a Γ-*closure* is a substitution $\cdot[\vec{a}/\vec{x}]$ where each $a_i{:}A$. Now if $\mathcal{R} \subseteq Rel$, let its *open extension*, $\mathcal{R}^\circ$, be the least relation between proved expressions such that

$$(\Gamma, e, A)\ \mathcal{R}^\circ\ (\Gamma, e', A) \text{ iff } e[\vec{a}/\vec{x}]\ \mathcal{R}\ e'[\vec{a}/\vec{x}] \text{ for any } \Gamma\text{-closure } [\vec{a}/\vec{x}].$$

For instance, relation Rel° holds between any two proved expressions (Γ, e, A) and (Γ', e', A') provided only that $\Gamma = \Gamma'$ and $A = A'$. As a matter of notation we shall write $\Gamma \vdash e\ \mathcal{R}\ e' : A$ to mean that $(\Gamma, e, A)\ \mathcal{R}\ (\Gamma, e', A)$ and, in fact, we shall usually omit the type information.

We need the following notion, of compatible refinement, to characterise what it means for a relation on open expressions to be a precongruence. If $\mathcal{R} \subseteq Rel^\circ$, its *compatible refinement*, $\widehat{\mathcal{R}} \subseteq Rel^\circ$, is defined inductively by the rules in Table 2.

Define a relation $\mathcal{R} \subseteq Rel^\circ$ to be a *precongruence* iff it contains its own compatible refinement, that is, $\widehat{\mathcal{R}} \subseteq \mathcal{R}$. This definition is equivalent to saying that a relation is preserved by substitution into any context.

Lemma 5.1 *Assume that $\mathcal{R} \subseteq Rel^\circ$ is a preorder. $\mathcal{R}$ is a precongruence iff whenever $\Gamma \vdash e \,\mathcal{R}\, e'$ and $\mathcal{C}$ is a context, it holds that $\Gamma \vdash \mathcal{C}[e] \,\mathcal{R}\, \mathcal{C}[e']$.* □

Howe's general congruence proof does not apply to our form of similarity, based on a labelled transition system, but we can adapt it as follows. Inductively define relation $\lesssim^\bullet \subseteq Rel^\circ$ by the following rule.

$$\frac{\Gamma \vdash e \mathrel{\widehat{\lesssim^\bullet}} e'' \qquad \Gamma \vdash e'' \lesssim^\circ e'}{\Gamma \vdash e \lesssim^\bullet e'}$$

Following Sands [27], we can present some basic properties of $\lesssim^\bullet$ from Howe's paper as follows.

Lemma 5.2 *$\lesssim^\bullet$ is reflexive and the following rules are valid.*

$$\frac{\Gamma \vdash e \lesssim^\bullet e'' \qquad \Gamma \vdash e'' \lesssim^\circ e'}{\Gamma \vdash e \lesssim^\bullet e'} \qquad \frac{\Gamma \vdash e \mathrel{\widehat{\lesssim^\bullet}} e'}{\Gamma \vdash e \lesssim^\bullet e'} \qquad \frac{\Gamma \vdash e \lesssim^\circ e'}{\Gamma \vdash e \lesssim^\bullet e'}$$

Moreover, $\lesssim^\bullet$ is the least relation closed under the first two rules. □

The proof strategy is to show that $\lesssim^\circ = \lesssim^\bullet$, and then since $\lesssim^\bullet$ is a precongruence (by the previous lemma) it follows that $\lesssim^\circ$ is too, as desired. We have $\lesssim^\circ \subseteq \lesssim^\bullet$ already, so it remains to prove the reverse inclusion. We do so by co-induction. Here is the key lemma.

Lemma 5.3 *Let $\mathcal{S} \stackrel{\text{def}}{=} \{(a, b) \mid \varnothing \vdash a \lesssim^\bullet b\}$.*

(1) *Whenever $a \,\mathcal{S}\, b$ and $a \rightsquigarrow a'$ then $a' \,\mathcal{S}\, b$.*

(2) *Whenever $a \,\mathcal{S}\, b$ and $a \xrightarrow{\alpha} a'$ there is b' with $b \xrightarrow{\alpha} b'$ and $a' \,\mathcal{S}\, b'$.* □

The proofs are by induction on the depth of inference of reduction $a \rightsquigarrow a'$ and transition $a \xrightarrow{\alpha} a'$ respectively. Details of similar proofs may be found in Howe [11] and Gordon [9]. Given this lemma, it is routine to show that $\lesssim^\bullet \subseteq \lesssim^\circ$ and hence it follows that $\lesssim^\circ = \lesssim^\bullet$, and hence similarity is a precongruence.

6 Refining Bisimulation

We have developed equational laws of bisimilarity and shown it to be a co-inductive characterisation of contextual equivalence. The basic co-induction principle for bisimilarity is to prove $a \sim b$ by exhibition of a bisimulation $\mathcal{S}$ containing (a, b). Since $\sim$ is the union of all bisimulations, it follows that $(a, b) \in \sim$. Our purpose in this section is to illustrate co-inductive proofs about a derived FPC type of unbounded streams. We begin with a direct bisimulation proof, but then develop three techniques to simplify the details.

The FPC type of streams of type A is the following.

$$\mathtt{Stm}(A) \stackrel{\text{def}}{=} \mathtt{rec}\, X.\, \mathtt{Unit} + (A \times X)$$

$$\mathtt{Stm}(A) \stackrel{\text{def}}{=} \mathtt{rec}\, X.\mathtt{Unit} + (A \times X)$$

$$\mathtt{nil}[A] \stackrel{\text{def}}{=} \mathtt{intro}[\mathtt{Stm}(A)](\mathtt{inl}(\mathtt{unity}))$$

$$\mathtt{cons}[A](e) \stackrel{\text{def}}{=} \mathtt{intro}[\mathtt{Stm}(A)](\mathtt{inr}(e))$$

$$\mathtt{lcase}[A](e_1, e_2, e_3) \stackrel{\text{def}}{=} \mathtt{case}\ \mathtt{elim}[\mathtt{Stm}(A)](e_1)\mathtt{of}$$
$$\mathtt{inl}(x) \Rightarrow e_2\ \mathtt{or}$$
$$\mathtt{inr}(x\,y) \Rightarrow \mathtt{split}\, x\, y\, \mathtt{as}\, (x, y)\, \mathtt{in}\, e_3\, x\, y$$

$$\frac{}{\Gamma \vdash \mathtt{nil}[A] : \mathtt{Stm}(A)} \qquad \frac{\Gamma \vdash e : A \times \mathtt{Stm}(A)}{\Gamma \vdash \mathtt{cons}[A](e) : \mathtt{Stm}(A)}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Stm}(A) \qquad \Gamma \vdash e_2 : B \qquad \Gamma \vdash e_3 : A \to \mathtt{Stm}(A) \to B}{\Gamma \vdash \mathtt{lcase}[A](e_1, e_2, e_3) : B}$$

$$\mathtt{nil} \not\rightsquigarrow \qquad \mathtt{cons}(a) \not\rightsquigarrow$$

$$\mathtt{lcase}(a, b_1, b_2) \rightsquigarrow \mathtt{lcase}(a', b_1, b_2) \text{ if } a \rightsquigarrow a'$$

$$\mathtt{lcase}(\mathtt{nil}, b_1, b_2) \rightsquigarrow b_1$$

$$\mathtt{lcase}(\mathtt{cons}(a), b_1, b_2) \rightsquigarrow \mathtt{lcase}(\mathtt{cons}(a'), b_1, b_2) \text{ if } a \rightsquigarrow a'$$

$$\mathtt{lcase}(\mathtt{cons}((a_1, a_2)), b_1, b_2) \rightsquigarrow b_2\, a_1\, a_2$$

Table 3 Definition and properties of the FPC stream type

We show in Table 3 definitions of `nil` and `cons` constructors, and a Martin-Löf style `lcase` destructor. As in ML, we shall write $a :: as$ for $\mathtt{cons}(a, as)$ (but remember these are possibly unbounded streams). We need the following exhaustion lemma, provable from the theory in Section 4.

Lemma 6.1 *If* as:$\mathtt{Stm}(A)$ *then either (A)* $as \sim \Omega^{\mathtt{Stm}(A)}$*, (B)* $as \sim \mathtt{nil}$*, (C)* $as \sim \mathtt{cons}(\Omega^{A \times \mathtt{Stm}(A)})$ *or (D)* $as \Downarrow a :: as'$ *where* a:A *and* as':$\mathtt{Stm}(A)$*.* □

Suppose we have `map` and `iterate` combinators specified by the following equations.

```
map f nil = nil
map f (x::xs) = f x :: map f xs
iterate f x = x :: iterate f (f x))
```

These could easily be turned into formal definitions of two combinators. Pattern matching on streams would be accomplished using `lcase`, but we omit the details. Intuitively the streams

$$\mathtt{iterate}\, f\, (f\, x) \quad \text{and} \quad \mathtt{map}\, f\, (\mathtt{iterate}\, f\, x)$$

are equal, because they both consist of the sequence

$$f\, x, \quad f\, (f\, x), \quad f\, (f\, (f\, x)), \quad f\, (f\, (f\, (f\, x))), \quad \ldots$$

Here is how to prove this equality by co-induction.

Lemma 6.2 *If relations* $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S} \subseteq Rel$ *are*

$$\mathcal{S}_1 \stackrel{\text{def}}{=} \{(\mathtt{iterate}\, f\, (f\, c), \mathtt{map}\, f\, (\mathtt{iterate}\, f\, c)) \mid c{:}A \;\&\; f{:}A \to A\}$$
$$\mathcal{S}_2 \stackrel{\text{def}}{=} \{((c,a),(c,b)) \mid c{:}A \;\&\; (a,b) \in \mathcal{S}_1\}$$
$$\mathcal{S}_3 \stackrel{\text{def}}{=} \{(\mathtt{elim}(a), \mathtt{elim}(b)) \mid (a,b) \in \mathcal{S}_1\}$$
$$\mathcal{S} \stackrel{\text{def}}{=} \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3 \cup Id$$

(where $Id \subseteq Rel$ *is the relation of alpha-conversion restricted to* Rel*) then* $\mathcal{S}$ *is a bisimulation.*

Proof. Let property $(*)$ be $\mathcal{S}_3 \subseteq \langle \mathcal{S} \rangle$. For now we shall assume $(*)$ and hence show that $\mathcal{S}$ is a bisimulation; then we shall return and prove $(*)$. We consider each of the four ways in which $(a,b) \in \mathcal{S}$ and show that $(a,b) \in \langle \mathcal{S} \rangle$ in each case.

(1) $(a,b) \in \mathcal{S}_1$. Since the type of streams is a recursive type, the only transitions are $a \stackrel{\mathtt{elim}}{\longrightarrow} \mathtt{elim}(a)$ and $b \stackrel{\mathtt{elim}}{\longrightarrow} \mathtt{elim}(b)$, hence $(a,b) \in \langle \mathcal{S}_3 \rangle \subseteq \langle \mathcal{S} \rangle$.

(2) $(a,b) \in \mathcal{S}_2$. Both a and b are values of pair type, say (c,a') and (c,b') respectively, with $c{:}A$ and $(a',b') \in \mathcal{S}_1$. They each have two transitions.

$$a \stackrel{\mathtt{fst}}{\longrightarrow} c \qquad b \stackrel{\mathtt{fst}}{\longrightarrow} c \qquad a \stackrel{\mathtt{snd}}{\longrightarrow} a' \qquad b \stackrel{\mathtt{snd}}{\longrightarrow} b'$$

Hence $(a,b) \in \langle Id \cup \mathcal{S}_1 \rangle \subseteq \langle \mathcal{S} \rangle$.

(3) $(a,b) \in \mathcal{S}_3$. Our assumption $(*)$ is that $(a,b) \in \langle \mathcal{S} \rangle$.

(4) $(a,b) \in Id$. Trivially $(a,b) \in \langle Id \rangle \subseteq \langle \mathcal{S} \rangle$.

Hence it remains to prove $(*)$. Suppose then that $(a,b) \in \mathcal{S}_3$, in which case

$$a \equiv \mathtt{elim}(\mathtt{iterate}\, f\, (f\, c))$$
$$b \equiv \mathtt{elim}(\mathtt{map}\, f\, (\mathtt{iterate}\, f\, c))$$

for some $f{:}A \to A$ and $c{:}A$. By computing the reduction behaviour of a and b it is not hard to check the only transitions of a and b are

$$a \stackrel{\mathtt{inr}}{\longrightarrow} (f\, c, \mathtt{iterate}\, f\, (f\, (f\, c)))$$
$$b \stackrel{\mathtt{inr}}{\longrightarrow} (f\, c, \mathtt{map}\, f\, (\mathtt{iterate}\, f\, (f\, c))).$$

Property $(*)$ follows, then as $(a,b) \in \langle \mathcal{S}_2 \rangle \subseteq \langle \mathcal{S} \rangle$. □

Now, since $\mathcal{S}$ is a bisimulation it follows by co-induction that it, and indeed $\mathcal{S}_1$, is contained in bisimilarity. A corollary then is that

$$\mathtt{iterate}\, f\, (f\, c) \sim \mathtt{map}\, f\, (\mathtt{iterate}\, f\, c)$$

for any suitable f and c, what we set out to show.

6.1 Variant Greatest Fixpoints

We can refine the proof of Lemma 6.2 in various ways. First, the following lemma provides alternative characterisations of a greatest fixpoint.

Proposition 6.3 *Let U be an arbitrary universal set and let $F : \wp(U) \to \wp(U)$ be some monotone function. If $\nu \stackrel{\text{def}}{=} \nu X.\, F(X)$ we have:*

$$\begin{aligned} \nu &= \nu X.\, F(X) \cup \nu \qquad (\nu.\mathrm{I}) \\ &= \nu X.\, F(X \cup \nu) \qquad (\nu.\mathrm{II}) \\ &= \nu X.\, F(X \cup \nu) \cup \nu \quad (\nu.\mathrm{III}) \end{aligned}$$

□

These equations strengthen co-induction. For instance, we can slightly simplify the proof of Lemma 6.2 by setting $\mathcal{S}$ to be $\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$, but with no mention of Id. A replay of our calculations shows that $\mathcal{S} \subseteq \langle \mathcal{S} \cup Id \rangle$. Since $Id \subseteq \sim$ it follows that $\mathcal{S}$, though not a bisimulation, is dense with respect to the map $\mathcal{S} \mapsto \langle \mathcal{S} \cup \sim \rangle$. Hence by co-induction $\mathcal{S} \subseteq \nu \mathcal{S}.\, \langle \mathcal{S} \cup \sim \rangle$ and therefore $\mathcal{S} \subseteq \sim$ by $(\nu.\mathrm{II})$.

Paulson [20] implements co-induction principles based on these equations in Isabelle. Dual equations strengthen induction; for instance, the dual of $(\nu.\mathrm{II})$, $\mu = \mu X.\, F(X \cap \mu)$, corresponds to Melham's strong induction [14] in HOL.

6.2 Bisimulation via Values

Our second refinement further simplifies the proof of Lemma 6.2. If $\mathcal{S} \subseteq Rel$, define $\overline{\mathcal{S}} \subseteq Rel$ by

$$\underline{\ell}\ \overline{\mathcal{S}}\ \underline{\ell} \qquad \frac{e[a/x]\ \mathcal{S}\ e'[a/x] \quad (\forall a{:}A)}{\lambda x{:}A.\, e\ \overline{\mathcal{S}}\ \lambda x{:}A.\, e'}$$

$$\frac{a\ \mathcal{S}\ b}{\mathtt{intro}(a)\ \overline{\mathcal{S}}\ \mathtt{intro}(b)} \qquad \frac{a_1\ \mathcal{S}\ b_1 \qquad a_2\ \mathcal{S}\ b_2}{(a_1, a_2)\ \overline{\mathcal{S}}\ (b_1, b_2)}$$

$$\mathtt{unity}\ \overline{\mathcal{S}}\ \mathtt{unity} \qquad \frac{a\ \mathcal{S}\ b}{\mathtt{inl}(a)\ \overline{\mathcal{S}}\ \mathtt{inl}(b)} \qquad \frac{a\ \mathcal{S}\ b}{\mathtt{inr}(a)\ \overline{\mathcal{S}}\ \mathtt{inr}(b)}$$

If $\mathcal{S} \subseteq Rel$, define $\langle \mathcal{S} \rangle_V \subseteq Rel$ such that $(a, b) \in \langle \mathcal{S} \rangle_V$ iff $\exists u, v (a \sim u\ \overline{\mathcal{S}}\ v \sim b)$. Let $\nu_V \stackrel{\text{def}}{=} \nu \mathcal{S}.\, \langle \mathcal{S} \rangle_V$. We can prove that it approximates bisimilarity.

Proposition 6.4 $\nu_V \subset \sim$

Proof. The key lemma is that whenever $\mathcal{S} \subseteq \langle \mathcal{S} \rangle_o R$, then $\sim \mathcal{S} \sim \subseteq \lesssim$. Given this lemma and symmetry we have $\sim \nu_V \sim \subseteq \sim$. In fact $\nu_V \subseteq \sim$ since $Id \subseteq \sim$. The inclusion is strict because, for instance, $\mathbf{0} \sim \mathbf{0}$, although $(\mathbf{0}, \mathbf{0}) \notin \nu_V$ because no value is bisimilar to $\mathbf{0}$. □

Intuitively $(a, b) \in \nu_V$ iff a and b are bisimilar, and they both have a value, and so do their immediate subterms, 'all the way down'.

Co-induction with respect to $\langle - \rangle_V$ relies on matching of immediate subterms. We can allow matching via non-immediate subterms as follows. If we define $\langle \mathcal{S} \rangle_{V*} = \langle \mu \mathcal{R}.\, \mathcal{S} \cup \overline{\mathcal{R}} \rangle_V$ then by use of both induction and co-induction we can prove

Proposition 6.5 $\nu_V = \nu X.\, \langle X \rangle_{V*}$. □

Roughly speaking, unwinding the inner inductive definition permits arbitrary nesting of value constructors. Returning to Lemma 6.2, if we make the assumption that each $f^n\,c$ has a value, it is not hard to check that $\mathcal{S}_1 \subseteq \langle \mathcal{S}_1 \cup \nu_V \rangle_{V*}$, and hence by co-induction and (ν.II) that $\mathcal{S}_1 \subseteq \nu_V$ and indeed $\mathcal{S}_1 \subseteq \sim$. The reason for the restriction on each $f^n\,c$ is essentially that ν_V is an incomplete co-inductive characterisation of $\sim$. Our third refinement provides a complete such characterisation.

6.3 Bisimulation via Reductions

We begin with another functional, $\langle - \rangle_+$.

$$a\,\langle \mathcal{S} \rangle_+\, b \text{ iff } \exists a', b' (a \rightsquigarrow^+ a', b \rightsquigarrow^+ b' \;\&\; a'\,\mathcal{S}\,b')$$

If $\mathcal{S} \subseteq \langle \mathcal{S} \rangle_+$, starting from any pair in $\mathcal{S}$ we can make reductions in both programs to end up back in $\mathcal{S}$.

Proposition 6.6 *Let* $\nu_+ \stackrel{\text{def}}{=} \nu\mathcal{S}.\langle \mathcal{S} \rangle_+$.

(1) $(a,b) \in \nu_+$ *iff* $a{\Uparrow}$ *and* $b{\Uparrow}$.

(2) $\nu_+ \subset \sim$. □

The greatest fixpoints of both $\langle - \rangle_{V*}$ and $\langle - \rangle_+$ fall short of bisimilarity, but combining them we exactly match bisimilarity.

Theorem 6.7 $\sim \;= \nu\mathcal{S}.\langle \mathcal{S} \rangle_{V*} \cup \langle \mathcal{S} \rangle_+$. □

We omit the proof, but the significance of this equation is that it is a complete co-inductive characterisation of bisimilarity (and hence contextual equivalence) without mentioning labelled transitions. Let $F(\mathcal{S}) \stackrel{\text{def}}{=} \langle \mathcal{S} \rangle_{V*} \cup \langle \mathcal{S} \rangle_+$. Returning again to Lemma 6.2, we can easily check that $\mathcal{S}_1 \subseteq \langle \mathcal{S}_1 \cup \sim \rangle_{V*}$, indeed that $\mathcal{S}_1 \subseteq F(\mathcal{S}_1 \cup \sim)$ and hence by co-induction and (ν.II) that $\mathcal{S}_1 \subseteq \nu\mathcal{S}.F(\mathcal{S}) = \sim$. This time we need no restriction on each $f^n\,c$.

Here is an example that depends on matching reductions. If `filter` is defined by

```
filter f nil = nil
filter f (x::xs) =
  if f x then x :: filter f xs
         else      filter f xs
```

we can prove the following equation (where `o` is function composition).

Proposition 6.8 *For any* $f{:}B \to \texttt{Bool}$ *and* $g{:}B \to B$,

$$\texttt{filter}\,f \circ \texttt{map}\,g \sim \texttt{map}\,g \circ \texttt{filter}\,(f \circ g)$$

Proof. Let $\mathcal{S}$ be the following relation.

$$\{(\texttt{filter}\,f\,(\texttt{map}\,g\,as), \texttt{map}\,g\,(\texttt{filter}\,(f \circ g)\,as)) \mid as{:}\texttt{Stm}(B)\}$$

The result will follow if $\mathcal{S} \subseteq \sim$. We will show that

$$\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle_{V*} \cup \langle \mathcal{S} \rangle_+ \cup \sim$$

and hence by co-induction and (ν.III) that $\mathcal{S} \subseteq \sim$. Consider, then, any pair $(a, b) \in \mathcal{S}$,

$$a \equiv \mathtt{filter}\, f\, (\mathtt{map}\, g\, as)$$
$$b \equiv \mathtt{map}\, g\, (\mathtt{filter}\, (f \circ g)\, as).$$

We proceed by a case analysis of as according to Lemma 6.1. There are four cases: (A) $as \sim \Omega$, (B) $as \sim \mathtt{nil}$, (C) $as \sim \mathtt{cons}(\Omega)$ and (D) $as \Downarrow a' :: as'$. Only case (D) is of interest; the other cases follow easily. We must examine the three possible evaluations of $f(g\, a')$: (DA) $f(g\, a') \Uparrow$, (DB) $f(g\, a') \Downarrow \mathtt{true}$ and (DC) $f(g\, a') \Downarrow \mathtt{false}$. Only (DB) and (DC) are of interest. In case (DB) let u and v be the values

$$u \equiv g\, a' :: \mathtt{filter}\, f\, (\mathtt{map}\, g\, as')$$
$$v \equiv g\, a' :: \mathtt{map}\, g\, (\mathtt{filter}\, (f \circ g)\, as').$$

We have $a \sim u$ and $b \sim v$ and hence $(a, b) \in \langle \mathcal{S} \cup \sim \rangle_{V*}$. Finally, in case (DC) we cannot find matching values, but instead we have the matching reductions

$$a \rightsquigarrow^{+} \mathtt{filter}\, f\, (\mathtt{map}\, g\, as')$$
$$b \rightsquigarrow^{+} \mathtt{map}\, g\, (\mathtt{filter}\, (f \circ g)\, as')$$

and so have $(a, b) \in \langle \mathcal{S} \rangle_{+}$. By consideration of all these cases we have shown the desired inclusion and hence $\mathcal{S} \subseteq \sim$ follows by co-induction. □

Since `filter` is a partial function (think of $\mathtt{filter}\, (\lambda x.\, \mathtt{false})$) this example cannot be programmed in a co-recursive framework such as Paulson's [20].

We conclude with a more substantial example: a proof of the monad laws for streams [30]. Let `++` be the stream append operation, `join` the function that appends together a stream of streams, `id` the identity function and let $\mathtt{val}\, x = x :: \mathtt{nil}$.

Proposition 6.9

(1) $\mathtt{map}\, \mathtt{id} \sim \mathtt{id}$

(2) $\mathtt{map}\, (f \circ g) \sim \mathtt{map}\, f \circ \mathtt{map}\, g$

(3) $\mathtt{map}\, f \circ \mathtt{val} \sim \mathtt{val} \circ f$

(4) $\mathtt{map}\, f \circ \mathtt{join} \sim \mathtt{join} \circ \mathtt{map}\, (\mathtt{map}\, f)$

(5) $\mathtt{join} \circ \mathtt{val} \sim \mathtt{id}$

(6) $\mathtt{join} \circ \mathtt{map}\, \mathtt{val} \sim \mathtt{id}$

(7) $\mathtt{join} \circ \mathtt{map}\, \mathtt{join} \sim \mathtt{join} \circ \mathtt{join}$

Proof. Parts (3) and (5) follow by routine equational reasoning. Parts (1), (2) and (6) follow by straightforward co-inductions. If $\mathcal{S}_4$ is the relation

$$\{(\mathtt{map}\, f\, (\mathtt{join}\, ass), \mathtt{join}\, (\mathtt{map}\, (\mathtt{map}\, f)\, ass)) \mid ass{:}\mathtt{Stm}(\mathtt{Stm}(B))\}$$

it is possible to prove that $\mathcal{S}_4 \subseteq \langle \mathcal{S}_4 \cup \sim \rangle_{V*} \cup \langle \mathcal{S}_4 \rangle_{+}$ and hence part (4) follows

by co-induction, (ν.II) and extensionality. Finally, if $\mathcal{S}_7$ is the relation

$$\{(\mathtt{join}\,ass \mathbin{+\!\!+} \mathtt{join}\,(\mathtt{map}\,\mathtt{join}\,asss), \mathtt{join}(ass \mathbin{+\!\!+} \mathtt{join}\,asss)) \mid ass{:}\mathtt{Stm}(\mathtt{Stm}(B)), asss{:}\mathtt{Stm}(\mathtt{Stm}(\mathtt{Stm}(B)))\}$$

we can prove $\mathcal{S}_7 \subseteq \langle \mathcal{S}_7 \cup \sim \rangle_{V*} \cup \langle \mathcal{S}_7 \rangle_+ \cup \sim$ and hence part (7) follows by co-induction, (ν.III) and extensionality. □

7 A Context Lemma for FPC

Our final contribution is to rework Milner's context lemma for FPC and show it yields yet another co-inductive characterisation of contextual equivalence, but one that is less wieldly than bisimilarity. Milner [15] showed that contextual equivalence on PCF is unchanged if we restrict attention to 'applicative contexts' of the form $[\,]\,a_1 \ldots a_n$. The analogue in FPC is an evaluation context of the form $\vec{\mathcal{E}}[\,]$, where if $\vec{\mathcal{E}} = \mathcal{E}_1, \ldots, \mathcal{E}_n$ then $\vec{\mathcal{E}}[\,]$ is the context $\mathcal{E}_1[\cdots \mathcal{E}_n[\,]\cdots]$. Let *experimental equivalence*, $\approx \subseteq Rel$ be the relation such that

$$a \approx b \text{ iff whenever } \vec{\mathcal{E}}[a], \vec{\mathcal{E}}[b]{:}\mathtt{Num}, \text{ that } \vec{\mathcal{E}}[a]{\Downarrow} \text{ iff } \vec{\mathcal{E}}[b]{\Downarrow}.$$

By a straightforward modification of Milner's argument, we can prove the following context lemma by induction on n.

Lemma 7.1 *Suppose $a \approx b$ and that $\mathcal{C}[a], \mathcal{C}[b]{:}\mathtt{Num}$. If $\mathcal{C}[a]{\Downarrow}$ in n steps, then $\mathcal{C}[b]{\Downarrow}$ too.* □

An easy corollary is that $\approx \;=\; \simeq$. Since it is straightforward to prove that $\rightsquigarrow \subseteq \approx$, for instance, experimental equivalence and the context lemma form a useful technique for establishing equational properties of contextual equivalence, independently of bisimilarity.

Furthermore, we can co-inductively characterise experimental equivalence as follows. If $\mathcal{S} \subseteq Rel$, define functional[1] $F(\mathcal{S}) \subseteq Rel$ such that $(a, b) \in F(\mathcal{S})$ iff

(1) if $a, b{:}\mathtt{Num}$ then $a{\Downarrow}$ iff $b{\Downarrow}$;

(2) whenever $\mathcal{E}[a], \mathcal{E}[b] \in Prog$, $(\mathcal{E}[a], \mathcal{E}[b]) \in \mathcal{S}$.

Proposition 7.2 $\approx \;= \nu\mathcal{S}.\, F(\mathcal{S})$.

Proof. Let $\nu = \nu\mathcal{S}.\, F(\mathcal{S})$. It is easy to see that $\approx$ is F-dense and so $\approx \subseteq \nu$ by co-induction. For the reverse inclusion, suppose that $(a, b) \in \nu$, $\vec{\mathcal{E}}[a], \vec{\mathcal{E}}[b]{:}\mathtt{Num}$ and $\vec{\mathcal{E}}[a]{\Downarrow}$. Since $\nu = F(\nu)$, it follows by induction on the size of $\vec{\mathcal{E}}$ that $(\vec{\mathcal{E}}[a], \vec{\mathcal{E}}[b]) \in \nu$. Hence if $\vec{\mathcal{E}}[a]{\Downarrow}$ it must be that $\vec{\mathcal{E}}[b]{\Downarrow}$, by clause (1) of the definition of F. Hence $\nu \subseteq \approx$. □

This yields a co-induction principle for contextual equivalence, but we can improve it as follows.

[1] We took atomic experiments as primitive—rather than compound evaluation contexts—to allow a simple presentation of this functional.

Proposition 7.3 $\approx = \nu\mathcal{S}.\, F(\approx\mathcal{S}\approx)$. □

The proof is a variation on the proof that in CCS a 'bisimulation up to $\sim$' is contained in bisimilarity [16, p93]. On the face of it, this yields a useful co-induction principle, intuitively via 'matching experiments.' To show $\mathcal{S}$ is contained in experimental equivalence, it suffices to show that $\mathcal{S} \subseteq F(\approx\mathcal{S}\approx)$. For instance, if our candidate relation $\mathcal{S}$ contains a pair (a, b) of function type, we must show for every experiment $\mathcal{E}$ of form $[\,]\,c$ that $\mathcal{E}[a] \equiv a\,c \approx\mathcal{S}\approx b\,c \equiv \mathcal{E}[b]$, which is equivalent to the bisimulation condition. But suppose $\mathcal{S}$ contains a pair $(\mathtt{inl}(a), \mathtt{inl}(b))$; we must show that $\mathcal{E}[\mathtt{inl}(a)] \approx\mathcal{S}\approx \mathcal{E}[\mathtt{inl}(b)]$ for all suitable experiments, $\mathcal{E}$, which must be of the form

$$\mathtt{case}\,[\,]\,\mathtt{of}\,\mathtt{inl}(x_1) \Rightarrow e_1\,\mathtt{or}\,\mathtt{inr}(x_2) \Rightarrow e_2.$$

Hence we must show $e_1[{}^{a}/_{x_1}] \approx\mathcal{S}\approx e_1[{}^{b}/_{x_1}]$ which, because of the quantification over the arbitrary term e_1 is almost as hard as proving contextual equivalence directly, and certainly harder than proving $(a, b) \in \mathcal{S}$, the condition for $\mathcal{S}$ to be a bisimulation. This is evidence that although the context lemma justifies a certain co-inductive characterisation of contextual equivalence, it is harder to apply than bisimilarity.

8 Variations on FPC

We have presented one particular form of call-by-name FPC in detail. Our main results hold under several variations of the language.

As case (C) of Lemma 6.1 shows, our type of streams contains junk programs such as $\mathtt{cons}(\Omega)$. Miranda and Haskell have primitive sum-of-product types on the grounds that the possibility of such programs causes implementation inefficiency [21]. If we include primitive sums-of-products we can rule out case (C) of Lemma 6.1 and our type of streams becomes isomorphic to that in Miranda or Haskell.

Gunter [10] has `fst` and `snd` operations on pairs instead of `split`. In the absence of sums-of-products we needed `split`—which gives control of evaluation of pairs—to simplify proofs about streams. If we had `fst` and `snd` operations instead of `split` we could make the product type passive, modify the labelled transition system to allow unconditional `fst` and `snd` transitions, and hence derive a surjective pairing law, that $a \sim (\mathtt{fst}\,a, \mathtt{snd}\,a)$ whenever $a{:}A \times B$.

In our language there are no experiments to determine whether programs of passive type terminate. We can add a convergence testing operation, $\mathtt{seq}(a, b)$, which first evaluates a—of arbitrary type—and if it terminates, evaluates b and returns its value. This is sometimes known as a 'lazy' variation [25], though implementations of call-by-name using lazy evaluation do not depend on convergence testing. Contexts can now distinguish $\Omega^{A\to B}$ and $\lambda x{:}A.\,\Omega^{B}$, for instance. We can still prove operational extensionality, but we must modify the labelled transition system so that every transition $a \xrightarrow{\alpha} b$ is contingent on convergence of a. Every type must be active.

Similarly we can obtain a call-by-value version and prove operational ex-

tensionality. Every type is active. Variables stand for values, not arbitrary programs. We must eliminate the PCF recursion expression, $\texttt{rec}\, x{:}E.\, e$, because although it is not a value its reduction rule involves substitution of itself for the variable x. Fixpoint combinators can be coded in FPC anyway using contravariant recursive types [10]. Recursion (and hence divergence) can be recovered in call-by-value PCF by adding recursively-defined constants.

9 Discussion and Related Work

We have developed a 'CCS-view of lambda-calculus.' Using a novel labelled transition system for FPC, we replayed the definition of bisimilarity from CCS and proved that it equals contextual equivalence. Hence we answered Turner's [29, Preface] concern that in a typed, call-by-name setting, Abramsky's applicative bisimulation makes more distinctions than observable by well-typed contexts. We developed some refinements of the bisimulation proof technique that take advantage of the determinacy of our language, and demonstrated their utility on a series of stream-processing examples. Finally, we generalised Milner's context lemma from PCF to FPC, to yield another co-inductive form of contextual equivalence, but offered evidence that it yields a weaker co-induction principle than bisimilarity.

The main novelty of our work relative to earlier work on application bisimulation [1,9,11,27] is our use of a labelled transition system to match contextual equivalence exactly in a typed setting, and our refinements of bisimulation in Section 6. These refinements ought to be applicable to recent work on applicative bisimulation for deterministic languages with state [22,26]. Mason, Smith and Talcott [13] also advocate operational methods for functional programming. Their work is based on a form of the context lemma, indeed they derive a form of fixpoint induction, but they do not emphasise co-induction.

Bernstein and Stark [3] also use a labelled transition system for a functional language. Their system is more complex than the one of this paper in that they represent substitutions explicitly using labels.

Domain theory is the classical foundation of languages such as FPC, and indeed Pitts [23] shows how to derive a co-induction principle for recursively defined domains. In contrast our approach is based on the operational definition of our language. Working directly with program texts rather than with abstract denotations has some modest rewards. For instance the idea of 'bisimulation via reductions,' which formalises a simple intensional intuition, has no counterpart in Pitts' work.

Sangiorgi [28] has generalised various refinements of co-induction found in concurrency theory, in terms of his notion of respectful functions on relations. The functions $\langle - \rangle_{V*}$ and $\langle - \rangle_{+}$ do not directly fit Sangiorgi's framework, but the possible connections are worth pursuing.

Our approach to proofs about infinite streams rests on Tarski's impredicative proof of the existence of greatest fixpoints (Theorem 2.1)—the greatest fixpoint is defined as the union of a set of relations which includes itself. Coquand [5] is developing a predicative type theory that explains seemingly

impredicative definitions—for instance of infinite streams—in purely inductive terms.

Acknowledgement

The idea of defining bisimilarity on a deterministic functional language via a labelled transition system arose in joint work with Roy Crole [6]. I am grateful for many conversations with colleagues at Cambridge, Chalmers and Glasgow. John Hatcliff and Søren Lassen pointed out errors in an earlier version of this paper. This work was supported by a Royal Society University Research Fellowship.

References

[1] Samson Abramsky. The lazy lambda calculus. In Turner [29], pages 65–116.

[2] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[3] Karen L. Bernstein and Eugene W. Stark. Operational semantics of a focusing debugger. In *Eleventh Annual Conference on Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B.V., 1995.

[4] Bard Bloom. Can LCF be topped? Flat lattice models of typed lambda calculus. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 282–295. IEEE Computer Society Press, 1988.

[5] Thierry Coquand. Infinite objects in type theory. In *Types of Proofs and Programs*, pages 62–78, volume 806 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[6] Roy L. Crole and Andrew D. Gordon. A sound metalogical semantics for input/output effects. In *Computer Science Logic'94, Kazimierz, Poland, September 1994*, volume 933 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1995.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[8] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[9] Andrew D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, 1994.

[10] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, Mass., 1992.

[11] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[12] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. Preprint, 1994.

[13] I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. Submitted for publication, 1994.

[14] Thomas F. Melham. A package for inductive relation definitions in HOL. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, California*, pages 350–357. IEEE Computer Society Press, 1991.

[15] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–23, 1977.

[16] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[17] James H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, December 1968.

[18] C.-H. Luke Ong. Correspondence between operational and denotational semantics: The full abstraction problem for PCF. Submitted to *Handbook of Logic in Computer Science* Volume 3, OUP 1994, January 1994.

[19] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, March 1981.

[20] Lawrence C. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report 304, University of Cambridge Computer Laboratory, 1993.

[21] Simon L. Peyton Jones. FLIC—a functional language intermediate code. *ACM SIGPLAN Notices*, 23(8):30–48, August 1988.

[22] Andrew Pitts and Ian Stark. On the observable properties of higher order functions that dynamically create local names (preliminary report). In *SIPL'93: ACM SIGPLAN Workshop on State in Programming Languages*, pages 31–45, June 1993.

[23] Andrew M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[24] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[25] Jon G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 3:387–415, 1993.

[26] Eike Ritter and Andrew M. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In *Proceedings TLCA'95, Edinburgh*, 1995.

[27] David Sands. Operational theories of improvement in functional languages (extended abstract). In *Functional Programming, Glasgow 1991*, Workshops in Computing, pages 298–311. Springer-Verlag, 1992.

[28] Davide Sangiorgi. On the bisimulation proof method. Technical Report ECS–LFCS–94–299, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, August 1994.

[29] David Turner, editor. *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[30] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[31] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Mass., 1993.