BRICS

Basic Research in Computer Science

BRICS NS-95-3 A. D. Gordon: Bisimilarity as a Theory of Functional Programming

Bisimilarity as a Theory of Functional Programming

Mini-Course

Andrew D. Gordon

BRICS Notes Series NS-95-3

ISSN 0909-3206 July 1995

See back inner page for a list of recent publications in the BRICS Notes Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

Bisimilarity as a Theory of Functional Programming. Mini-Course.*

Andrew D. Gordon
University of Cambridge Computer Laboratory

June 1995

Abstract

Operational intuition is central to computer science. These notes introduce functional programmers to operational semantics and operational equivalence. We show how the idea of bisimilarity from CCS may be applied to deterministic functional languages. On elementary operational grounds it justifies equational reasoning and proofs about infinite streams.

*Notes to accompany graduate lectures at BRICS in the Computer Science Department of Aarhus University, March 1995. © A. D. Gordon, 1995. Earlier versions of part of this material appeared in the proceedings of the Glasgow FP'94 (Gordon 1995b) and MFPS'95 (Gordon 1995a) conferences. This version of November 24, 1995 fixes minor errors in the version printed as number NS–95–3 in the BRICS Notes Series. If you find any mistakes or have any comments please email `adg@cl.cam.ac.uk`.

Contents

List of Tables

1 Objectives

The object of these notes is to offer a new perspective on the behaviour of functional programs based on CCS-style labelled transitions and bisimilarity. Apart from this new material, these notes are intended to introduce functional programmers to the basic ideas of operational semantics and co-inductive definitions. We have in mind a reader familiar with programming in lazy functional languages, such as Miranda or Haskell, but use a simpler language, PCF plus streams, as the vehicle for developing our theory. PCF is simply-typed lambda-calculus plus arithmetic (Plotkin 1977). Streams are one of the key ingredients of lazy functional programming (Hughes 1989). They unify many ideas: both infinite and finite lists, co-routines, multiple outcomes, standard input and output, files and so on. PCF plus streams is probably the minimal functional language that admits interesting co-inductive proofs of program equivalence. Operational intuition is central to computer science. These notes show how operational semantics, together with the elementary mathematics of inductive and co-inductive definitions, is enough to build from first principles an expressive theory of functional programming.

Morris-style contextual equivalence is widely accepted as the natural notion of operational equivalence for PCF-like languages. Two programs are contextually equivalent if they may be interchanged for one another in any larger program of ground 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, typically using some form of domain theory. This is the 'full abstraction' problem for PCF; see Ong (1994) 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 (1989) entirely operational theory of CCS, based on labelled transitions and bisimilarity. A labelled transition takes the form $a \xrightarrow{\alpha} b$, 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 (1981) 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 (Abramsky and Ong 1993) and by Howe (1989), who invented a powerful method of showing that bisimilarity is a congruence. Both showed that their untyped forms of bisimilarity equalled contextual equivalence. 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 (1990, Preface) expressed concern that applicative bisimulation would fall short of contextual equivalence for languages such as Miranda or Haskell.

We answer Turner's concern by showing that by defining a labelled transition system for PCF plus streams and then defining bisimilarity exactly as in CCS, we can prove it equals contextual equivalence. In particular, in the call-by-name variant we have $\Omega^{A \to B}$ bisimilar to $\lambda(x{:}A)\Omega^{B}$. This is the first new result presented in these notes. The second is the investigation in Section 4 of refinements of bisimulation that exploit determinacy.

Here is the structure of these notes.

- Section 2 begins by introducing PCF plus streams, and defining contextual equivalence. The problem then is to characterise contextual equivalence co-inductively. After explaining how principles of induction and co-induction follow from Tarski's theorem, we do so by giving a labelled transition system, and replaying the definition of bisimilarity from CCS. We conclude the section by illustrating bisimulation proofs about stream-processing programs and deriving basic properties of bisimilarity.

- Section 3 falls into two halves. In the first we prove bisimilarity equals contextual equivalence. Howe proved congruence of bisimilarity for a general class of **lazy computation systems**. We rework his method for our form of bisimilarity, based on a labelled transition system. In the second half we adapt Milner's context lemma (1977) to construct another co-inductive characterisation of contextual equivalence, which we call 'experimental equivalence.' The congruence proof for experimental equivalence is unrelated to Howe's. We find that the associated co-induction principle is weaker than bisimulation.

- In Section 4 we exploit the determinacy of reduction in PCF plus streams to obtain refinements of bisimulation, analogous to the idea of 'bisimulation up to $\sim$' in CCS. In particular we obtain a new co-inductive characterisation of contextual equivalence based on reduction behaviour and production of values.

- We sketch in Section 5 how the results of the previous sections extend to richer languages than PCF plus streams. In particular, our theory holds for a language of sums, products and recursive types (Gunter 1992; Winskel 1993), and for call-by-value variations.

- Section 6 concludes by reviewing related work and discussing the significance of these results.

2 Bisimilarity for PCF plus Streams

This section explains the central ideas of this operational theory. The language we work with, PCF plus streams, is given in a series of fragments in separate tables. In fact all the results of these notes hold for a richer language, FPC, which includes sums, products and recursive types. We will prove properties only for PCF plus streams. FPC is defined in Section 5, and call-by-value variants are described, but we leave extension of all the proofs as exercises.

2.1 Definition of PCF plus Streams

Definition 2.1 *Here is the syntax of types, A, and expressions, e,*

$$
\begin{array}{rcl}
A & ::= & \texttt{Bool} \mid \texttt{Num} \mid A \to A \mid \texttt{[}A\texttt{]} \\
e & ::= & x \mid \lambda(x{:}A)e \mid e(e) \mid \texttt{rec}(f{:}A \to B, x{:}A)e \mid \underline{n}(n \in \mathbb{N}) \mid \underline{bv}(bv \in \{tt, f\!f\}) \\
& \mid & \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{succ}(e) \mid \texttt{pred}(e) \mid \texttt{zero}(e) \\
& \mid & \texttt{nil} \mid e :: e \mid \texttt{scase}(e, e, e)
\end{array}
$$

Here are our metavariable conventions.

	variables	*closed*	*possibly-open*
Types	X, Y	A, B	
Expressions	x, y	a, b	e
Values	–	v	–

We usually write `true` and `false` for $\underline{tt}$ and $\underline{f\!f}$ respectively, and allow metavariable ℓ to range over all literals, $\mathbb{N} \cup \{tt, f\!f\}$. We identify phrases up to alpha-conversion, that is, consistent renaming of bound variables. We use $\equiv$ for syntactic identity. We write $\phi[{}^{\psi}\!/\!{}_{x}]$ for the substitution of phrases ψ for each occurrence of variable x in phrase ϕ. We write $fv(e)$ for the set of variables free in expression e.

Our syntax of PCF plus streams uses separate syntactic constructors, such as, $\texttt{succ}(-)$, $\texttt{pred}(-)$, $\texttt{rec}(f, x)(-)$, ..., for arithmetic, recursion and so on. Instead, PCF can be presented as a simply typed lambda-calculus with function constants for arithmetic and recursion (Scott 1993; Plotkin 1977). By using separate forms of syntax we simplify the presentation here of the operational semantics. But this is an issue of form rather than content. Elsewhere we present a call-by-value form of PCF using function constants (Crole and Gordon 1995).

We present the rules that define the semantics of call-by-name PCF plus streams as a collection of fragments: $\Delta_{\rightarrow}$ (functions, Table 1), Δ_A (arithmetic, Table 2), $\Delta_{[-]}$ (streams, Table 3) and $\Delta_{\mu\rightarrow}$ (recursive functions, Table 4). PCF itself is given by

$$\mathrm{PCF} = \Delta_{\rightarrow} \cup \Delta_A \cup \Delta_{\mu\rightarrow}$$

and PCF plus streams is $\mathrm{PCF} \cup \Delta_{[-]}$. For any of these languages, the static semantics is given as follows.

Definition 2.2 (Static Semantics) *The* **type assignment** *relation* $\Gamma \vdash e : A$ *is given inductively by the* (Exp $-$) *rules of the language, where* Γ *is an* **environment**, *a finite map from variables to types, of form* $x_1{:}A_1, \ldots, x_n{:}A_n$. *We assume implicitly that environments appearing in the rules of static semantics are well-formed. In general we define*

$$\begin{aligned} a, b \in \mathit{Prog}(A) &\stackrel{\text{def}}{=} \{e \mid \varnothing \vdash e : A\} \\ \mathit{Prog} &\stackrel{\text{def}}{=} \bigcup\{\mathit{Prog}(A) \mid A \textit{ is a type}\} \\ \mathit{Rel}(A) &\stackrel{\text{def}}{=} \{(a,b) \mid a \in \mathit{Prog}(A) \ \&\ b \in \mathit{Prog}(A)\} \\ \mathcal{R}, \mathcal{S} \subseteq \mathit{Rel} &\stackrel{\text{def}}{=} \bigcup\{\mathit{Rel}(A) \mid A \textit{ is a type}\} \end{aligned}$$

and we write $a_1, \ldots, a_n{:}A$ *to mean that* $\{a_1, \ldots, a_n\} \subseteq \mathit{Prog}(A)$.

Proposition 2.3

(1) *If* $\Gamma \vdash e : A$ *and* $\Gamma \vdash e : B$ *then* $A \equiv B$.

(2) *If* $\Gamma, x{:}A_2 \vdash e_1 : A_1$ *and* $\Gamma \vdash e_2 : A_2$ *then* $\Gamma \vdash e_1[{}^{e_2}\!/\!{}_{x}] : A_1$.

Proof

(1) By rule induction on the derivation of $\Gamma \vdash e : A$.

(2) By rule induction on the derivation of $\Gamma, x{:}A_2 \vdash e_1 : A_1$.

■

To define the dynamic semantics, we need the notion of a context, an expression-with-a-hole. It is enough for our purposes for the hole to be represented by a distinguished variable.

Definition 2.4 (Contexts) *Suppose there is some arbitrary variable,* $-$, *used to stand for a* **hole** *in an expression. Let a* **context** *be an expression* e *such that* $fv(e) \subseteq \{-\}$. *If* e *is a context, we write* $e[e']$ *short for* $e[{}^{e'}\!/\!{}_{-}]$.

Statics

$$\Gamma, x{:}A, \Gamma' \vdash x : A \quad \text{(Exp } x\text{)}$$

$$\frac{\Gamma, x{:}B \vdash e : A \qquad x \notin \mathit{Dom}(\Gamma)}{\Gamma \vdash \lambda(x{:}B)e : B \to A} \text{(Exp Fun)}$$

$$\frac{\Gamma \vdash e_1 : B \to A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1(e_2) : A} \text{(Exp Appl)}$$

Lazy Dynamics

$$\begin{array}{cc} v ::= \lambda(x{:}A)e & \text{(Value Fun)} \\ (\lambda(x{:}A)e)(a) \mapsto e[{}^{a}\!/\!{}_{x}] & \text{(Red Beta)} \\ \mathcal{E} ::= -(a) & \text{(Exper Rator)} \end{array}$$

Table 1: $\Delta_{\to}$: functions

The dynamic semantics is based on a small-step reduction relation, $\mapsto$. Its reduction strategy is determined by a set of **experiments**, with metavariable $\mathcal{E}$, the set of contexts under which reduction is closed.

Definition 2.5 (Dynamic Semantics) *The* **experiments** *and* **values** *are the contexts and programs, respectively, given by the* (Exper $-$) *and* (Value $-$) *rules of the language. The* **small-step reduction** *relation $a \mapsto b$ is given inductively by the* (Red $-$) *rules of the language together with the generic rule*

$$\frac{a \mapsto b}{\mathcal{E}[a] \mapsto \mathcal{E}[b]} \text{(Red Experiment)}$$

We define the usual notions of evaluation, convergence and divergence as follows.

$$\begin{array}{rcll} a \mapsto & \stackrel{\text{def}}{=} & \exists b(a \mapsto b) & \text{`}a\ \textit{reduces'} \\ a \Downarrow b & \stackrel{\text{def}}{=} & a \mapsto^* b\ \&\ \neg(b \mapsto) & \text{`}a\ \textit{evaluates to}\ b\text{'} \\ a\Downarrow & \stackrel{\text{def}}{=} & \exists b(a \Downarrow b) & \text{`}a\ \textit{converges'} \\ a\Uparrow & \stackrel{\text{def}}{=} & \textit{whenever}\ a \mapsto^* b,\ b \mapsto & \text{`}a\ \textit{diverges'} \end{array}$$

Statics

$$\Gamma \vdash \underline{n} : \texttt{Num} \quad \text{(Exp Num)} \qquad \Gamma \vdash \underline{bv} : \texttt{Bool} \quad \text{(Exp Bool)}$$

$$\frac{\Gamma \vdash e : \texttt{Num}}{\Gamma \vdash \texttt{succ}(e) : \texttt{Num}}\text{(Exp Succ)} \qquad \frac{\Gamma \vdash e : \texttt{Num}}{\Gamma \vdash \texttt{pred}(e) : \texttt{Num}}\text{(Exp Pred)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_3 : A}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : A}\text{(Exp If)}$$

$$\frac{\Gamma \vdash e : \texttt{Num}}{\Gamma \vdash \texttt{zero}(e) : \texttt{Bool}}\text{(Exp Zero)}$$

Dynamics

$$v ::= \underline{bv} \mid \underline{n} \quad \text{(Value Arith)}$$

$$\texttt{if}\, \underline{bv}\, \texttt{then}\, a_{tt}\, \texttt{else}\, a_{f\!f} \mapsto a_{bv} \quad \text{(Red If)} \qquad \texttt{succ}(\underline{n}) \mapsto \underline{n+1} \quad \text{(Red Succ)}$$

$$\texttt{pred}(\underline{0}) \mapsto \underline{0} \quad \text{(Red Pred 0)} \qquad \texttt{pred}(\underline{n+1}) \mapsto \underline{n} \quad \text{(Red Pred Succ)}$$

$$\texttt{zero}(\underline{0}) \mapsto \underline{tt} \quad \text{(Red Zero 0)} \qquad \texttt{zero}(\underline{n+1}) \mapsto \underline{f\!f} \quad \text{(Red Zero Succ)}$$

$$\mathcal{E} ::= \texttt{if} - \texttt{then}\, a\, \texttt{else}\, a \mid \texttt{succ}(-) \mid \texttt{pred}(-) \mid \texttt{zero}(-) \quad \text{(Exper Arith)}$$

Table 2: Δ_A: arithmetic

Statics

$$\Gamma \vdash \texttt{nil} : \texttt{[}A\texttt{]} \quad \text{(Exp Nil)}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : \texttt{[}A\texttt{]}}{\Gamma \vdash e_1 :: e_2 : \texttt{[}A\texttt{]}} \text{(Exp Cons)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{[}A\texttt{]} \qquad \Gamma \vdash e_2 : B \qquad \Gamma \vdash e_3 : A \to \texttt{[}A\texttt{]} \to B}{\Gamma \vdash \texttt{scase}(e_1, e_2, e_3) : B} \text{(Exp Scase)}$$

Lazy Dynamics

$$\begin{array}{cc}
v ::= \texttt{nil} \mid (a :: a) & \text{(Value Stm)} \\
\texttt{scase}(\texttt{nil}, b, c) \mapsto b & \text{(Red Scase Nil)} \\
\texttt{scase}(a_1 :: a_2, b, c) \mapsto c(a_1)(a_2) & \text{(Red Scase Cons)} \\
\mathcal{E} ::= \texttt{scase}(-, a, a) & \text{(Exper Stm)}
\end{array}$$

Table 3: $\Delta_{\texttt{[-]}}$: streams

Statics

$$\frac{\Gamma, f{:}A \to B, x{:}A \vdash e : B}{\Gamma \vdash \texttt{rec}(f{:}A \to B, x{:}A)e : A \to B} \text{(Exp Rec)}$$

Lazy Dynamics

$$\begin{array}{cc}
v ::= \texttt{rec}(f, x)e & \text{(Value Rec)} \\
(\texttt{rec}(f, x)e)(a) \mapsto e[\texttt{rec}(f, x)e, a/f, x] & \text{(Red Rec)}
\end{array}$$

Table 4: $\Delta_{\mu\to}$: recursive functions

An **evaluation context** is a context of the form $\vec{\mathcal{E}}[-]$, that is, $\mathcal{E}_1[\ldots\mathcal{E}_n[-]\ldots]$, where $\vec{\mathcal{E}}$ is a list of experiments $\mathcal{E}_1,\ldots,\mathcal{E}_n$ with $n \geq 0$. Felleisen and Friedman (1986) pioneered the use of evaluation contexts (though they originally used the term 'reduction contexts'). Taking experiments—atomic evaluation contexts—as primitive helps our discussion of experimental equivalence in Sections 3.4 and 5.1.

For all the languages considered in these notes, reduction is deterministic and preserves types.

Proposition 2.6

(1) *Whenever $a \mapsto b$ and $a \mapsto c$, $b \equiv c$.*

(2) *If $a{:}A$ and $a \mapsto b$ then $b{:}A$ too.*

A corollary of (1) is that $a\Uparrow$ iff $\neg(a\Downarrow)$. It is not hard to check that a program a is a value iff it is $\mapsto$-normal, which is to say that $\neg(a \mapsto)$. Hence the set of values is exactly the image of the evaluation relation, that is, $\{b \mid \exists a(a \Downarrow b)\}$.

Given the dynamic semantics, we can state a form of Morris (1968) contextual equivalence.

Definition 2.7 Contextual order, *$\sqsubseteq \subseteq Rel$, and* **contextual equivalence**, *$\simeq \subseteq Rel$, are given by:*

$a \sqsubsim b$ *iff whenever $a, b{:}A$ and $-{:}A \vdash e : \texttt{Bool}$, that $e[a]\Downarrow$ implies $e[b]\Downarrow$ too.*
$a \simeq b$ *iff $a \sqsubsim b$ and $b \sqsubsim a$.*

The idea is that two programs are equivalent if no amount of programming can tell them apart. This is also known as 'observational congruence' (Meyer and Cosmadakis 1988). Two programs are distinct if there is a context that distinguishes them. The central theme of these notes is to establish contextual equivalences by characterising contextual equivalence as a form of bisimilarity. It would be equivalent but less wieldy to formulate contextual equivalence in terms of convergence to `true` or to `false`, or to an integer. Another common variation is to define contextual equivalence on open expressions using variable-capturing contexts; we will show in Proposition 3.14 that this variation makes no difference.

2.2 Induction and Co-induction

This section recalls the dual foundations of induction and co-induction. We derive strong versions of both. 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 $X \subseteq Y$).

Induction and co-induction are dual proof principles that derive from the definition of a set to be the least or greatest solution, respectively, of equations of the form $X = F(X)$.

First some definitions. 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.

$$\begin{aligned} \mu X. F(X) &\stackrel{\text{def}}{=} \bigcap\{X \mid F(X) \subseteq X\} \\ \nu X. F(X) &\stackrel{\text{def}}{=} \bigcup\{X \mid X \subseteq F(X)\} \end{aligned}$$

Lemma 2.8

(1) *$\mu X. F(X)$ is the least F-closed set.*

(2) *$\nu X. F(X)$ is the greatest F-dense set.*

Proof We prove (2); (1) follows by a dual argument. Since $\nu X. F(X)$ contains every F-dense set by construction, we need only show that it is itself F-dense, for which the following lemma suffices.

If every X_i is F-dense, so is the union $\bigcup_i X_i$.

Since $X_i \subseteq F(X_i)$ for every i, $\bigcup_i X_i \subseteq \bigcup_i F(X_i)$. Since F is monotone, $F(X_i) \subseteq F(\bigcup_i X_i)$ for each i. Therefore $\bigcup_i F(X_i) \subseteq F(\bigcup_i X_i)$, and so we have $\bigcup_i X_i \subseteq F(\bigcup_i X_i)$ by transitivity, that is, $\bigcup_i X_i$ is F-dense. ■

Theorem 2.9 (Tarski)

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

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

Proof Again we prove (2) alone; (1) follows by a dual argument. Let $\nu = \nu X. F(X)$. We have $\nu \subseteq F(\nu)$ by Lemma 2.8. So $F(\nu) \subseteq F(F(\nu))$ by monotonicity of F. But then $F(\nu)$ is F-dense, and therefore $F(\nu) \subseteq \nu$. Combining the inequalities we have $\nu = F(\nu)$; it is the greatest fixpoint because any other is F-dense, and hence contained in ν. ■

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.

$$\begin{array}{lll} \textbf{Induction:} & \mu X.\, F(X) \subseteq X & \text{if } X \text{ is } F\text{-closed.} \\ \textbf{Co-induction:} & X \subseteq \nu X.\, F(X) & \text{if } X \text{ is } F\text{-dense.} \end{array}$$

Mathematical induction is a special case. Suppose there is an element $0 \in U$ and an injective function $S : U \to U$. If we define a monotone function $F : \wp(U) \to \wp(U)$ by

$$F(X) \stackrel{\text{def}}{=} \{0\} \cup \{S(x) \mid x \in X\}$$

and set $\mathbb{N} \stackrel{\text{def}}{=} \mu X.\, F(X)$, the associated principle of induction is that $\mathbb{N} \subseteq X$ if $F(X) \subseteq X$, which is to say that

$$\mathbb{N} \subseteq X \text{ if both } 0 \in X \text{ and } \forall x \in X(S(x) \in X).$$

We can refine co-induction (and dually induction) using the following equations.

Proposition 2.10 *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:*

$$\begin{array}{rcll} \nu & = & \nu X.\, F(X) \cup \nu & (\nu.\text{I}) \\ & = & \nu X.\, F(X \cup \nu) & (\nu.\text{II}) \\ & = & \nu X.\, F(X \cup \nu) \cup \nu & (\nu.\text{III}) \end{array}$$

Proof Make the following definitions.

$$\begin{array}{rcl} \nu_1 & \stackrel{\text{def}}{=} & \nu X.\, F(X) \cup \nu \\ \nu_2 & \stackrel{\text{def}}{=} & \nu X.\, F(X \cup \nu) \\ \nu_3 & \stackrel{\text{def}}{=} & \nu X.\, F(X \cup \nu) \cup \nu \end{array}$$

We must show that each ν_i equals ν. Since $\nu \subseteq F(\nu)$ it follows by co-induction that $\nu \subseteq \nu_i$ for each i. The reverse inclusions take a little more work.

$\nu_2 \subseteq \nu$. Since ν is a fixpoint of F, which is monotone, we have $\nu = F(\nu) \subseteq F(\nu_2 \cup \nu)$. Now $\nu_2 \subseteq F(\nu_2 \cup \nu)$ so $\nu_2 \cup \nu \subseteq F(\nu_2 \cup \nu)$, and therefore $\nu_2 \cup \nu \subseteq \nu$ by co-induction. Hence $\nu_2 \subseteq \nu$.

$\nu_1 \subseteq \nu_2$. We have $\nu_1 = F(\nu_1) \cup \nu = F(\nu_1) \cup F(\nu) \subseteq F(\nu_1 \cup \nu)$. So $\nu_1 \subseteq \nu X.\, F(X \cup \nu)$.

$\nu_3 \subseteq \nu_2$. We have $\nu_3 = F(\nu_3 \cup \nu) \cup \nu = F(\nu_3 \cup \nu) \cup F(\nu) = F(\nu_3 \cup \nu)$ since $F(\nu) \subseteq F(\nu_3 \cup \nu)$. Hence $\nu_3 \subseteq \nu X.\, F(X \cup \nu)$. ■

Paulson (1993) implements co-induction principles based on these equations in Isabelle. Dual equations strengthen induction; for instance, the dual of (ν.II), $\mu = \mu X. F(X \cap \mu)$, corresponds to the strong induction of Camilleri and Melham (1992) in HOL: $\mu \subseteq X$ if $F(X \cap \mu) \subseteq X$. Equation ($\nu$.III) and its dual justify the following principles, where μ and ν are the least and greatest fixpoints, respectively, of monotone F.

Strong Induction:	$\mu \subseteq X$	if $F(X \cap \mu) \cap \mu \subseteq X$.
Strong Co-induction:	$X \subseteq \nu$	if $X \subseteq F(X \cup \nu) \cup \nu$.

For numbers, our strong induction yields $\mathbb{N} \subseteq X$ if $\{0\} \cup \{S(x) \in \mathbb{N} \mid x \in X \;\&\; x \in \mathbb{N}\} \subseteq X$.

Aczel (1977) is the standard reference on inductive definitions. Davey and Priestley (1990) give a more recent account of fixpoint theory, including Tarski's theorem. Both Winskel (1993) and Pitts (1994b) explain how sets of rules give rise to inductive definitions of operational semantics, with associated principles of rule induction. Our use of 'closed' and 'dense' differs slightly from Aczel's; if R is a set of rules, he says a set is 'R-closed' or 'R-dense' to mean it is in our sense F-closed or F-dense, respectively, where F is the functional determined by the rule set R. Aczel mentions the dual of induction in passing, but 'co-induction' seems first to have been used by Milner and Tofte (1991).

The standard theory of inductive and co-inductive definitions is non-constructive in that it relies on Tarski's impredicative definitions of $\mu X. F(X)$ and $\nu X. F(X)$. They are formed by quantifying over classes of sets, the sets of F-closed and F-dense sets respectively, that include themselves. Cousot and Cousot (1979) develop a constructive version of fixpoint theory. Coquand (1993) is developing a predicative type theory that explains seemingly impredicative definitions—for instance of infinite streams—in purely inductive terms.

2.3 Divergence

We can characterise divergence co-inductively in terms of unbounded reduction. Let function $\mathcal{D} : \wp(\mathit{Prog}) \to \wp(\mathit{Prog})$ be such that

$$\mathcal{D}(X) \stackrel{\text{def}}{=} \{a \mid \exists b(a \mapsto b \;\&\; b \in X)\}.$$

We can easily see that $\mathcal{D}$ is monotone. Hence it possesses a greatest fixpoint, $\nu X. \mathcal{D}(X)$, which is the greatest $\mathcal{D}$-dense set. We can show that this co-inductive definition matches the one from Definition 2.5, that $a\Uparrow$ iff whenever $a \mapsto^* b$, then $b \mapsto$.

Theorem 2.11 $\Uparrow = \nu X.\, \mathcal{D}(X)$.

Proof Let ν be $\nu X.\, \mathcal{D}(X)$.

$\nu \subseteq \Uparrow$. Suppose that $a \in \nu$. We must show whenever $a \mapsto^* b$, that $b \mapsto$. If $a \in \nu$, then $a \in \mathcal{D}(\nu)$ so there is an a' with $a \mapsto a'$ and $a' \in \nu$. Furthermore since reduction is deterministic, a' is unique. By iterating this argument, whenever $a \in \nu$ and $a \mapsto^* b$ it must be that $b \in \nu$. Therefore $b \mapsto$.

$\Uparrow \subseteq \nu$. By co-induction it suffices to prove that set $\Uparrow$ is $\mathcal{D}$-dense. Suppose that $a\Uparrow$. Since $a \mapsto^* a$, we have $a \mapsto$, that is, $a \mapsto b$ for some b. But whenever $b \mapsto^* b'$ it must be that $a \mapsto^* b'$ too. In fact $b' \mapsto$ since $a\Uparrow$. Hence $b\Uparrow$ too, $a \in \mathcal{D}(\Uparrow)$ and therefore $\Uparrow$ is $\mathcal{D}$-dense.

■

PCF plus streams includes divergent programs, but only because of recursive functions, $\Delta_{\mu\to}$ in Table 4. Reduction in $\Delta_{\to} \cup \Delta_A \cup \Delta_{[-]}$ can be shown to be normalising by standard methods (see Girard, Taylor, and Lafont (1989), for instance). Given recursive functions, we can define a program Ω^A at any type A to be $(\mathtt{rec}(f{:}\mathtt{Num} \to A, x{:}\mathtt{Num})f(x))(\underline{0})$ and we have

$$(\mathtt{rec}(f{:}\mathtt{Num} \to A, x{:}\mathtt{Num})f(x))(\underline{0}) \mapsto (\mathtt{rec}(f{:}\mathtt{Num} \to A, x{:}\mathtt{Num})f(x))(\underline{0})$$

by (Red Rec). Hence the set $\{\Omega^A\}$ is $\mathcal{D}$-dense, that is, $\{\Omega^A\} \subseteq \mathcal{D}(\{\Omega^A\})$. So $\{\Omega^A\} \subseteq \Uparrow$ by co-induction, and therefore $\Omega^A\Uparrow$.

It is convenient in Section 4.4 to have a co-inductive characterisation of divergence in terms of many-step reductions. Recall that $\mathcal{R}^+$ is the transitive closure of relation $\mathcal{R}$. If map $\mathcal{D}_+ : \wp(\mathit{Prog}) \to \wp(\mathit{Prog})$ is

$$\mathcal{D}_+(X) \stackrel{\text{def}}{=} \{a \mid \exists b(a \mapsto^+ b \;\&\; b \in X)\}$$

it is straightforward to prove the following.

Proposition 2.12 $\Uparrow = \nu X.\, \mathcal{D}_+(X)$.

Proof Let ν be $\nu X.\, \mathcal{D}_+(X)$. By co-induction it suffices to show that $\Uparrow \subseteq \mathcal{D}_+(\Uparrow)$ and that $\nu \subseteq \mathcal{D}(\nu)$. The former is the easy inclusion, as $\Uparrow = \mathcal{D}(\Uparrow)$. As for the latter, suppose that $a \in \nu$. Since $\nu = \mathcal{D}_+(\nu)$ there is b such that $a \mapsto^{n+1} b$ and $b \in \nu$. If $n = 0$ then $a \in \mathcal{D}(\nu)$ immediately. Otherwise there is c with $a \to c$ and $c \to^+ b$. So $c \in \mathcal{D}_+(\nu)$ by definition, and hence $a \in \mathcal{D}(\nu)$. In either case $a \in \mathcal{D}(\nu)$, as required. ■

Observations

$$
\begin{array}{lll}
\alpha & ::= & @a \qquad\qquad (A \to B \text{ passive}) \\
& | & n \mid \texttt{true} \mid \texttt{false} \quad (\texttt{Num}, \texttt{Bool} \text{ active}) \\
& | & \texttt{nil} \mid \texttt{hd} \mid \texttt{tl} \qquad ([A] \text{ active})
\end{array}
$$

$$
\frac{a{:}A \qquad A \text{ active} \qquad a \mapsto a' \qquad a' \xrightarrow{\alpha} b}{a \xrightarrow{\alpha} b} \;(\text{Trans Red})
$$

$$
\begin{array}{cr}
a \xrightarrow{@b} a(b) \text{ if } a{:}B \to A \text{ and } b{:}B & (\text{Trans Fun}) \\
\underline{n} \xrightarrow{n} \mathbf{0} & (\text{Trans Num}) \\
\texttt{true} \xrightarrow{\texttt{true}} \mathbf{0} \qquad \texttt{false} \xrightarrow{\texttt{false}} \mathbf{0} & (\text{Trans Bool}) \\
\texttt{nil} \xrightarrow{\texttt{nil}} \mathbf{0} & (\text{Trans Nil}) \\
a :: b \xrightarrow{\texttt{hd}} a & (\text{Trans Hd}) \\
a :: b \xrightarrow{\texttt{tl}} b & (\text{Trans Tl})
\end{array}
$$

Table 5: Observations on PCF plus streams

Experiments propagate divergence. Using the fact that reduction is closed under any under experiment, we can easily check the following.

Proposition 2.13 *Suppose* $-{:}A \vdash \mathcal{E} : B$ *and* $a{:}A$*. If* $a\Uparrow$ *then* $\mathcal{E}[a]\Uparrow$ *too.*

Hughes and Moran (1995) give an alternative, 'big-step', co-inductive formulation of divergence.

2.4 Bisimilarity

To characterise contextual equivalence co-inductively, we begin with a **labelled transition system** that characterises the immediate observations one can make of a program.

Definition 2.14 (Labelled Transition System) *Table 5 specifies a set Act of* **actions***, ranged over by* α*, and a partition of the types into two kinds,* **active** *and* **passive***. Let* $\mathbf{0}$ *be some arbitrary divergent program of some active type. The labelled transition system is a family of relations* $(\xrightarrow{\alpha} \subseteq \mathit{Prog} \times \mathit{Prog} \mid \alpha \in \mathit{Act})$*, given inductively by the* (Trans -) *rules in Table 5.*

The transitions are designed to correspond to the observations contexts can make of programs. Programs of active type have transitions iff they converge. Programs of passive type unconditionally have transitions. We discuss the active/passive distinction further in Section 5.1.

Lemma 2.15 *Suppose $a{:}A$ and A is active. Then $a\Downarrow$ iff $\exists \alpha, b(a \xrightarrow{\alpha} b)$.*

Proof We can easily prove by rule induction on transition $a \xrightarrow{\alpha} b$ that $a\Downarrow$. On the other hand, any value has a transition, so if $a \Downarrow v$, a has a transition too, by iterating (Trans Red). ■

The labelled transition system is image-singular in the following sense.

Lemma 2.16 *Whenever $a \xrightarrow{\alpha} b$ and $a \xrightarrow{\alpha} c$ then $b \equiv c$.*

Proof Suppose $a{:}A$. If A is passive, then only one rule can derive either $a \xrightarrow{\alpha} b$ or $a \xrightarrow{\alpha} c$ and by inspection $b \equiv c$. If A is active, by rule induction on their derivations there are values u and v such that $a \Downarrow u$ and $u \xrightarrow{\alpha} b$, and $a \Downarrow v$ and $v \xrightarrow{\alpha} c$. By determinacy of $\Downarrow$, a consequence of Proposition 2.6(1), $u \equiv v$, and then by inspection it follows that $b \equiv c$. ■

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. Following Milner (1989), we wish to regard two programs as behaviourally equivalent iff their derivation trees are the same 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 $a' \sim b'$.

In fact there are many such relations; the empty set is one. We are after the largest or most generous such relation. We can define it co-inductively as follows. First define two functions $[-], \langle - \rangle : \wp(Rel) \to \wp(Rel)$ by

$$\begin{aligned} [\mathcal{S}] &\stackrel{\text{def}}{=} \{(a, b) \mid \text{whenever } a \xrightarrow{\alpha} a' \text{ there is } b' \text{ with } b \xrightarrow{\alpha} b' \text{ and } a' \,\mathcal{S}\, b'\} \\ \langle \mathcal{S} \rangle &\stackrel{\text{def}}{=} [\mathcal{S}] \cap [\mathcal{S}^{\text{op}}]^{\text{op}} \end{aligned}$$

where $\mathcal{S} \subseteq Rel$. By examining element-wise expansions of these definitions, it is not hard to check that a relation satisfies property $(*)$ iff it is a fixpoint of function $\langle - \rangle$. One can easily check that both functions $[-]$ and $\langle - \rangle$ are monotone. Hence what we seek, the greatest relation to satisfy $(*)$, does exist, and equals $\nu\mathcal{S}.\,\langle \mathcal{S} \rangle$, the greatest fixpoint of $\langle - \rangle$. We make the following standard definitions (Milner 1989).

- **Bisimilarity**, $\sim \subseteq Rel$, is $\nu\mathcal{S}.\,\langle\mathcal{S}\rangle$.
- A **bisimulation** is a $\langle - \rangle$-dense relation.

Bisimilarity is the greatest bisimulation and $\sim \, = \langle\sim\rangle$. Again by expanding the definitions we can see that relation $\mathcal{S} \subseteq Rel$ is a bisimulation iff $a\,\mathcal{S}\,b$ implies

- Whenever $a \xrightarrow{\alpha} a'$ there is b' with $b \xrightarrow{\alpha} b'$ and $a'\,\mathcal{S}\,b'$;
- Whenever $b \xrightarrow{\alpha} b'$ there is a' with $a \xrightarrow{\alpha} a'$ and $a'\,\mathcal{S}\,b'$.

An asymmetric version of bisimilarity is of interest too. Let a **simulation** be a $[-]$-dense relation. Let **similarity**, $\lesssim \subseteq Rel$, be $\nu\mathcal{S}.\,[\mathcal{S}]$, the greatest simulation.

From Lemma 2.15 we know that 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 used $\mathbf{0}$ in defining the transition system to indicate that after observing the value of a literal there is nothing more to observe.

2.5 The Map/Iterate Example

To motivate study of bisimilarity, let us see how straightforward it is to use co-induction to establish that two lazy streams are bisimilar. Suppose `map` and `iterate` are a couple of builtin constants 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, with each equation being a valid series of reductions, but we omit the details. Pattern matching on streams would be accomplished using `scase`. Intuitively the streams

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

are equal, because they both consist of the sequence

$$f\,x :: f^2\,x :: f^3\,x :: f^4\,x :: \cdots$$

We cannot directly prove this equality by induction, because there is no argument to induct on. Instead we can easily prove it by co-induction, via the following lemma.

Lemma 2.17 *If $\mathcal{S} \subseteq Rel$ is*

$$\{(\texttt{iterate}\, f\,(f\,x), \texttt{map}\, f\,(\texttt{iterate}\, f\,x)) \mid \exists A(x{:}A \,\&\, f{:}A \to A)\}$$

then $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle$.

Proof To show $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle$ we must consider any pair $(a, b) \in \mathcal{S}$ and check that each transition $a \xrightarrow{\alpha} a'$ is matched by a transition $b \xrightarrow{\alpha} b'$, such that either $a' \,\mathcal{S}\, b'$ or $a' \sim b'$, and vice versa. Suppose then that a is $\texttt{iterate}\, f\,(f\,x)$, and b is $\texttt{map}\, f\,(\texttt{iterate}\, f\,x)$. We can calculate the following reductions.

$$\begin{array}{rcl} a & \mapsto^{+} & f(x) :: (\texttt{iterate}\, f\,(f\,(f\,x))) \\ b & \mapsto^{+} & f(x) :: (\texttt{map}\, f\,(\texttt{iterate}\, f\,(f\,x))) \end{array}$$

Using the reductions above we can enumerate all the transitions of a and b.

$$a \xrightarrow{\texttt{hd}} f\,x \tag{1}$$

$$a \xrightarrow{\texttt{tl}} \texttt{iterate}\, f\,(f\,(f\,x)) \tag{2}$$

$$b \xrightarrow{\texttt{hd}} f\,x \tag{3}$$

$$b \xrightarrow{\texttt{tl}} \texttt{map}\, f\,(\texttt{iterate}\, f\,(f\,x)) \tag{4}$$

Now it is plain that $(a, b) \in \langle \mathcal{S} \cup \sim \rangle$. Transition (1) is matched by (3), and vice versa, with $f\,x \sim f\,x$ (since $\sim$ is reflexive). Transition (2) is matched by (4), and vice versa, with $\texttt{iterate}\, f\,(f\,(f\,x))\, \mathcal{S}\, \texttt{map}\, f\,(\texttt{iterate}\, f\,(f\,x))$. ■

Hence $\mathcal{S} \subseteq \sim$ by strong co-induction (Section 2.2). A corollary then is that

$$\texttt{iterate}\, f\,(f\,x) \sim \texttt{map}\, f\,(\texttt{iterate}\, f\,x)$$

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

2.6 Theory of Bisimilarity

We postpone till Section 3 the lengthy proof that bisimilarity is a congruence. We finish this section by proving a good many properties of bisimilarity mainly by easy co-inductions. First some general facts; we prove them for PCF plus streams but in fact they hold for all the languages considered in these notes.

Proposition 2.18

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

(2) $\mapsto \subseteq \sim$

(3) $\Downarrow \subseteq \sim$

(4) $\sim \subseteq \lesssim$

(5) $\lesssim \cap \lesssim^{\mathrm{op}} \subseteq \sim$

(6) $\{(a,b) \in Rel \mid a\Uparrow \ \&\ b\Uparrow\} \subseteq \sim$

Proof

(1) Similar to Proposition 4.2 of Milner (1989).

(2) If $\mathcal{S}$ is $\{(a,b) \mid a \mapsto b\}$, it is enough by strong co-induction to show that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle$. It suffices to consider the transitions of any $a, b{:}A$ with $a \mapsto b$. We first consider any arbitrary transition $a \xrightarrow{\alpha} a'$. If A is active, then by determinacy of $\mapsto$ it must be that $b \xrightarrow{\alpha} a'$ and we have $a' \sim a'$ by reflexivity. Otherwise we must analyse the transition rules for functions, the one passive type.

(Trans Fun) Here α is $@c$ and a' is $a(c)$. We can derive $b \xrightarrow{\alpha} b(c)$ and we have $a(c)\,\mathcal{S}\,b(c)$.

Dually, we must consider any arbitrary transition $b \xrightarrow{\alpha} b'$. If A is active by (Trans Red) it must be that $a \xrightarrow{\alpha} b'$ too, and $b'\,Id\,b'$. If A is passive, by a similar case analysis as before we can exhibit a' such that $a \xrightarrow{\alpha} a'$ and $a'\,\mathcal{S}\,b'$.

(3) Combine (2) and the fact that $\sim$ is a preorder.

(4) Since $\sim = \langle \sim \rangle = [\sim] \cap [\sim^{\mathrm{op}}]^{\mathrm{op}}$, $\sim \subseteq [\sim]$ and hence $\sim \subseteq \lesssim$.

(5) Let $\mathcal{S} = \lesssim \cup \lesssim^{\mathrm{op}}$. By showing that $\mathcal{S}$ is a bisimulation, we will have that $\lesssim \cup \lesssim^{\mathrm{op}} \subseteq \sim$. Since $\mathcal{S} = \mathcal{S}^{\mathrm{op}}$, it suffices to show that $\mathcal{S} \subseteq [\mathcal{S}]$. Consider any $(a,b) \in \mathcal{S}$ and any transition $a \xrightarrow{\alpha} a'$. Since $a \lesssim b$ there is b' with $b \xrightarrow{\alpha} b'$ and $a' \lesssim b'$. Then since $b \lesssim a$ there must be an a'' with $a \xrightarrow{\alpha} a''$ and $b \lesssim a''$. But since the transition relation is image-singular, Lemma 2.16, $a' \equiv a''$ and hence we have $a'\,\mathcal{S}\,b'$ and therefore $\mathcal{S} \subseteq [\mathcal{S}]$.

(6) It is easy to check that $\{(a,b) \in Rel \mid a\Uparrow \ \&\ b\Uparrow\}$ is a bisimulation. ■

Since $\sim$ is symmetric, (4) and (5) imply the equation $\sim = \lesssim \cap \lesssim^{\mathrm{op}}$. The latter relation is sometimes called 'mutual similarity'. In a nondeterministic calculus such as CCS, the transition system fails to be image singular and mutual similarity strictly contains bisimilarity, that is, property (5) fails.

For any type A define the following.

$$\begin{aligned} \mathit{Total}(A) &\stackrel{\text{def}}{=} \forall a{:}A \exists b{:}A (b \Downarrow b \mathbin{\&} a \sim b) \\ \Omega^A_\sim &\stackrel{\text{def}}{=} \{a \mid a{:}A \mathbin{\&} a \sim \Omega^A\} \end{aligned}$$

$\mathit{Total}(A)$ asserts that every program of type A equals a value. $\Omega^A_\sim$ is the $\sim$-equivalence class of Ω^A. A corollary of (6) above is that $\Uparrow^A \subseteq \Omega^A_\sim$ irrespective of the type A, where $\Uparrow^A = \{a{:}A \mid a\Uparrow\}$. The converse depends on A.

Proposition 2.19 *For any type A, the following are equivalent.*

(1) $\Omega^A_\sim \subseteq \Uparrow^A$, *that is,* $\forall a{:}A(a \sim \Omega^A \Rightarrow a\Uparrow)$

(2) $\forall a{:}A(a\Downarrow \Rightarrow a \not\sim \Omega^A)$

(3) $\forall a, b{:}A(a\Downarrow \mathbin{\&} a \sim b \Rightarrow b\Downarrow)$

(4) $\neg\, \mathit{Total}(A)$

Proof

(1) $\Leftrightarrow$ (2) Contrapositives of one another.

(2) $\Rightarrow$ (3) For a contradiction, suppose $a\Downarrow$, $a \sim b$ and $b\Uparrow$. Since $\Uparrow^A \subseteq \Omega^A_\sim$, $b \sim \Omega^A$, and $a \sim \Omega^A$ by transitivity. But since $a\Downarrow$ applying (2) yields $a \not\sim \Omega^A$. Contradiction.

(3) $\Rightarrow$ (2) Again for a contradiction, suppose $a\Downarrow$ and $a \sim \Omega^A$. Applying (3) yields $\Omega^A\Downarrow$, but $\Omega^A\Uparrow$ by definition.

(4) $\Rightarrow$ (1) Property (4) means there is an a such that $\neg\exists b{:}A(b\Downarrow \mathbin{\&} a \sim b)$, that is, a equals no convergent term. So $a\Uparrow$ in particular, since $a \sim a$. Therefore $a \sim \Omega^A$, since $\Uparrow^A \subseteq \Omega^A_\sim$. By transitivity, Ω^A equals no convergent term, that is, property (1), $\Omega^A_\sim \subseteq \Uparrow^A$.

$\neg$(4) $\Rightarrow$ $\neg$(1) Property $\mathit{Total}(A)$ applied to Ω^A yields $\exists b(\Omega^A \sim b \mathbin{\&} b\Downarrow)$, contradicting (1). ■

A corollary is that every type A satisfies one of $\mathit{Total}(A)$ or $\Omega^A_\sim \subseteq \Uparrow^A$, but not both. It is easy to see that every active type, A, satisfies $\Omega^A_\sim \subseteq \Uparrow^A$.

If $a{:}A \to B$ then $a \sim \lambda(x)a(x)$	(Fun Eta)
If $a, b{:}A \to B$ then $a \sim b$ if $\forall c{:}A(a(c) \sim b(c))$	(Fun Ext)
If $as{:}\texttt{[}A\texttt{]}$, then $as \sim \texttt{scase}(as, \texttt{nil}, \lambda(x)\lambda(xs)x :: xs)$.	(Stm Eta)
If $\underline{\ell} \sim \underline{\ell'}$ then $\ell = \ell'$.	(Inj Lit)
If $\lambda(x{:}A)e \sim \lambda(x{:}A)e'$ then $e[{}^{a}\!/\!{}_{x}] \sim e'[{}^{a}\!/\!{}_{x}]$ for any $a{:}A$.	(Inj Fun)
If $a :: as \sim b :: bs$ then $a \sim b$ and $as \sim bs$.	(Inj Stm)

Table 6: Properties Specific to PCF plus Streams

Proposition 2.20 *If A active, $\Omega^A_{\sim} \subseteq \Uparrow^A$.*

Proof Suppose $a{:}A$ and that A is active. From Lemma 2.15 it follows that if $a \sim \Omega$, a has no transitions, and hence that it diverges. ■

Now we turn to properties specific to PCF plus streams.

Proposition 2.21 *All the properties in Table 6 hold.*

Proof We examine each property in turn.

(Fun Eta) If $\mathcal{S} = \{(a, \lambda(x)a(x)) \mid a{:}A \to B\}$ we can show that $\mathcal{S} \subseteq \langle\sim\rangle$ and hence that $\mathcal{S} \subseteq \sim$. The only transitions of a and $\lambda(x)a(x)$ are of the form $a \xrightarrow{@c} a(c)$ and $\lambda(x)a(x) \xrightarrow{@c} (\lambda(x)a(x))(c)$ for arbitrary c. But by (Red Beta) we have $(\lambda(x)a(x))(c) \mapsto a(c)$, which establishes that $\mathcal{S} \subseteq \langle\sim\rangle$.

(Fun Ext) Suppose that $a(c) \sim b(c)$ for any c. Clearly then $(a, b) \in \langle\sim\rangle$.

(Stm Eta) Consider any $as{:}\texttt{[}A\texttt{]}$. Either $as\Uparrow$, $as \Downarrow \texttt{nil}$ or $as \Downarrow b :: bs$. In any case we can check that as is bisimilar to $\texttt{scase}(a, \texttt{nil}, \lambda(x)\lambda(xs)x :: xs)$. In the first case both diverge; in the other cases both evaluate to the same thing.

Finally (Inj Lit), (Inj Fun) and (Inj Stm) follow trivially. ■

The converses of (Fun Ext), (Inj Lit), (Inj Fun) and (Inj Stm) hold, and are instances of the congruence property of $\sim$ that we shall prove in the next section. In fact, these particular instances can be proved directly. A corollary of (Fun Eta) is that $Total(A \to B)$ holds for any function type $A \to B$. In PCF plus streams, then, $Total(A) \Leftrightarrow A$ passive.

For instance, we can prove $\lambda(x{:}A)\Omega^B \sim \Omega^{A \to B}$, which are distinguished by the standard form of applicative bisimulation. Consider any $a{:}A$. We have $(\lambda(x{:}A)\Omega^B)\,a \sim \Omega^B$ by (Red Beta) and $\Omega^{A \to B}\,a \sim \Omega^B$ since $\Uparrow^B \subseteq \Omega^B_\sim$. Hence $\lambda(x{:}A)\Omega^B \sim \Omega^{A \to B}$ by (Fun Ext).

3 Congruence of Bisimilarity

Here the main objective is to introduce Howe's method for proving that similarity is a precongruence, and the consequences that bisimilarity is a congruence, and that bisimilarity equals contextual equivalence. We also consider experimental equivalence, another co-inductive characterisation of contextual equivalence suggested by a generalisation of Milner's context lemma. But we show that co-induction based on experimental equivalence is a weaker principle than co-induction using bisimilarity.

3.1 Congruence and Precongruence

A congruence relation is an equivalence that is preserved by all contexts. To state this formally requires a little preliminary work.

Definition 3.1 *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* $a_i{:}A_i$. *Now if* $\mathcal{R} \subseteq Rel$, *let its* **open extension**, $\mathcal{R}^\circ$, *be the relation between proved expressions such that*

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

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.

Definition 3.2 *If* $\mathcal{R} \subseteq Rel^\circ$, *its* **compatible refinement**, $\widehat{\mathcal{R}} \subseteq Rel^\circ$, *is defined inductively for PCF plus streams by the rules in Table 7. Define a relation* $\mathcal{R} \subseteq Rel^\circ$ *to be a* **precongruence** *iff it contains its own compatible refinement, that is,* $\widehat{\mathcal{R}} \subseteq \mathcal{R}$. *Let a* **congruence** *be an equivalence relation that is a precongruence.*

The rules in Table 7 formalise the intention that two expressions are related by $\widehat{\mathcal{R}}$ when they have the same outermost form and their immediate subexpressions are related by $\mathcal{R}$.

The following lemma shows that this definition is equivalent to the more common one based on variable-capturing contexts. Let $\mathcal{C}$ stand for a variable-capturing context, that is, an expression, not identified up to alpha-conversion,

$$\frac{}{\Gamma \vdash x \mathrel{\widehat{\mathcal{R}}} x}\,(\text{Comp } x) \qquad \frac{\Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1' \qquad \Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2'}{\Gamma \vdash e_1(e_2) \mathrel{\widehat{\mathcal{R}}} e_1'(e_2')}\,(\text{Comp Appl})$$

$$\frac{\Gamma, x{:}A \vdash e \mathrel{\mathcal{R}} e'}{\Gamma \vdash \lambda(x{:}A)e \mathrel{\widehat{\mathcal{R}}} \lambda(x{:}A)e'}\,(\text{Comp Fun}) \qquad \frac{}{\Gamma \vdash \underline{\ell} \mathrel{\widehat{\mathcal{R}}} \underline{\ell}}\,(\text{Comp Lit})$$

$$\frac{\Gamma \vdash e \mathrel{\mathcal{R}} e' \qquad \kappa \in \{\mathtt{succ}, \mathtt{pred}, \mathtt{zero}\}}{\Gamma \vdash \kappa(e) \mathrel{\widehat{\mathcal{R}}} \kappa(e')}\,(\text{Comp } \kappa)$$

$$\frac{\Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1' \qquad \Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2' \qquad \Gamma \vdash e_3 \mathrel{\mathcal{R}} e_3'}{\Gamma \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \mathrel{\widehat{\mathcal{R}}} \mathtt{if}\, e_1'\, \mathtt{then}\, e_2'\, \mathtt{else}\, e_3'}\,(\text{Comp If})$$

$$\frac{\Gamma, f{:}A \to B, x{:}A \vdash e \mathrel{\mathcal{R}} e'}{\Gamma \vdash \mathtt{rec}(f{:}A \to B, x{:}A)e \mathrel{\widehat{\mathcal{R}}} \mathtt{rec}(f{:}A \to B, x{:}A)e'}\,(\text{Comp Rec})$$

$$\frac{\Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1' \qquad \Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2' \qquad \Gamma \vdash e_3 \mathrel{\mathcal{R}} e_3'}{\Gamma \vdash \mathtt{scase}(e_1, e_2, e_3) \mathrel{\widehat{\mathcal{R}}} \mathtt{scase}(e_1', e_2', e_3')}\,(\text{Comp Scase})$$

$$\frac{\Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1' \qquad \Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2'}{\Gamma \vdash e_1 :: e_2 \mathrel{\widehat{\mathcal{R}}} e_1' :: e_2'}\,(\text{Comp Cons}) \qquad \frac{}{\Gamma \vdash \mathtt{nil} \mathrel{\widehat{\mathcal{R}}} \mathtt{nil}}\,(\text{Comp Nil})$$

Table 7: The compatible refinement of a relation

whose bound variables are distinct, containing a single typed hole denoted by '$-_A$' and subject to the extra type assignment rule $\Gamma \vdash -_A : A$. We write $\mathcal{C}[e]$ for the outcome of filling the hole in the context $\mathcal{C}$ with the expression e. Free variables of e may be bound by this process.

Lemma 3.3 *Assume that $\mathcal{R} \subseteq Rel^\circ$ is a preorder. $\mathcal{R}$ is a precongruence iff* (Cong $\mathcal{R}$) *holds.* (Cong $\mathcal{R}$) *is given by*

$$\frac{\Gamma \vdash \mathcal{C}[-_A] : B \qquad \Gamma, \Gamma' \vdash e \mathrel{\mathcal{R}} e' : A}{\Gamma \vdash \mathcal{C}[e] \mathrel{\mathcal{R}} \mathcal{C}[e'] : B} \text{ (Cong } \mathcal{R})$$

where Γ' is the list of bound variables in $\mathcal{C}$ whose scope includes the hole.

Proof

(Only If) Suppose that preorder $\mathcal{R}$ satisfies $\widehat{\mathcal{R}} \subseteq \mathcal{R}$. We prove by induction on the derivation of $\Gamma \vdash \mathcal{C}[-_A] : B$ that if Γ' is the environment bound in $\mathcal{C}$, then $\Gamma, \Gamma' \vdash e \mathrel{\mathcal{R}} e'$ implies $\Gamma \vdash \mathcal{C}[e] \mathrel{\mathcal{R}} \mathcal{C}[e']$.

$\mathcal{C}$ is $-_A$. Such a context binds no variables, so $\Gamma' = \varnothing$. We have $\Gamma \vdash e \mathrel{\mathcal{R}} e'$ by assumption.

$\mathcal{C}$ is a variable Impossible because $\mathcal{C}$ has to contain a hole.

$\mathcal{C}$ is an application. Either $\mathcal{C}$ is $\mathcal{C}_0\, e_0$ or $e_0\, \mathcal{C}_0$, where $\mathcal{C}_0$ is a smaller context and e_0 is an ordinary expression (with no hole). In the first case we have $\Gamma \vdash \mathcal{C}_0 : C \to B$ and $\Gamma \vdash e_0 : C$ for some type C. By induction hypothesis we have $\Gamma \vdash \mathcal{C}_0[e] \mathrel{\mathcal{R}} \mathcal{C}_0[e'] : C \to B$ since $\mathcal{C}_0$ binds the same variables as $\mathcal{C}$. By (Comp Appl) and reflexivity we have $\Gamma \vdash \mathcal{C}_0[e]\, e_0 \mathrel{\mathcal{R}} \mathcal{C}_0[e']\, e_0 : B$, that is, $\Gamma \vdash \mathcal{C}[e_0] \mathrel{\mathcal{R}} \mathcal{C}[e'] : B$. The second case, when the context is $e_0\, \mathcal{C}_0$, is similar.

$\mathcal{C}$ is $\lambda(x{:}C_1)\mathcal{C}'$. The environment Γ' of variables bound by $\mathcal{C}$ must be $x{:}C_1, \Gamma''$ with $\Gamma, x{:}C_1 \vdash \mathcal{C}' : C_2$ where $B \equiv C_1 \to C_2$. By induction hypothesis we have $\Gamma, x{:}C_1 \vdash \mathcal{C}'[e] \mathrel{\mathcal{R}} \mathcal{C}'[e'] : C_2$ since $\Gamma, x{:}C_1, \Gamma'' \vdash e \mathrel{\mathcal{R}} e'$ and Γ'' is the variables bound in $\mathcal{C}'$. Rule (Comp Fun) now yields $\Gamma \vdash \mathcal{C}[e] \mathrel{\mathcal{R}} \mathcal{C}[e'] : B$.

Cases for the other forms of contexts follow similarly.

(If) Suppose that preorder $\mathcal{R}$ satisfies (Cong $\mathcal{R}$). We must show that $\widehat{\mathcal{R}} \subseteq \mathcal{R}$. We need to consider all the rules of compatible refinement from Table 7.

(Comp x) $\mathcal{R}$ is reflexive.

(Comp Lit) $\mathcal{R}$ is a reflexive.

(Comp Appl) Suppose that $\Gamma \vdash e_1 \mathrel{\mathcal{R}} e_1'$ and $\Gamma \vdash e_2 \mathrel{\mathcal{R}} e_2'$. By (Cong $\mathcal{R}$) we have $\Gamma \vdash e_1(e_2) \mathrel{\mathcal{R}} e_1'(e_2)$ and $\Gamma \vdash e_1'(e_2) \mathrel{\mathcal{R}} e_1'(e_2')$ and therefore by transitivity that $\Gamma \vdash e_1(e_2) \mathrel{\mathcal{R}} e_1'(e_2')$.

(Comp Fun) Suppose that $\Gamma, x{:}A \vdash e \mathrel{\mathcal{R}} e'$. Using context $\lambda(x{:}A){-}$ we have $\Gamma \vdash \lambda(x{:}A)e \mathrel{\widehat{\mathcal{R}}} \lambda(x{:}A)e'$ from (Cong $\mathcal{R}$).

The other cases are similar. ■

3.2 Similarity is a Precongruence

In this section we shall prove that similarity is a precongruence, that is, $\widehat{\lesssim^\circ} \subseteq \lesssim^\circ$. Since $\sim$ equals the symmetrisation of $\lesssim$, it follows that bisimilarity is a congruence (a precongruence that is an equivalence). Howe (1989) 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 (Crole and Gordon 1995), we recast his proof in a typed setting and using labelled transitions.

Definition 3.4 *Inductively define relation $\lesssim^\bullet \subseteq Rel^\circ$ by the following rule.*

$$\frac{\Gamma \vdash e \mathrel{\widehat{\lesssim^\bullet}} e'' \qquad \Gamma \vdash e'' \lesssim^\circ e'}{\Gamma \vdash e \lesssim^\bullet e'}\;(\text{Cand Def})$$

We can present some basic properties of $\lesssim^\bullet$ from Howe's paper as follows.

Lemma 3.5 *$\lesssim^\bullet$ is reflexive and the following rules are valid.*

$$\frac{\Gamma \vdash e \mathrel{\widehat{\lesssim^\bullet}} e'}{\Gamma \vdash e \lesssim^\bullet e'}\;(\text{Cand Cong}) \qquad \frac{\Gamma \vdash e \lesssim^\circ e'}{\Gamma \vdash e \lesssim^\bullet e'}\;(\text{Cand Sim})$$

$$\frac{\Gamma \vdash e \lesssim^\bullet e'' \qquad \Gamma \vdash e'' \lesssim^\circ e'}{\Gamma \vdash e \lesssim^\bullet e'}\;(\text{Cand Right})$$

$$\frac{\Gamma, x{:}A \vdash e_1 \lesssim^\bullet e_1' \qquad \Gamma \vdash e_2 \lesssim^\bullet e_2' : A}{\Gamma \vdash e_1[e_2/x] \lesssim^\bullet e_1'[e_2'/x]}\;(\text{Cand Subst})$$

Moreover, $\lesssim^\bullet$ is the least relation closed under (Cand Cong) *and* (Cand Right).

Proof First note that $\lesssim^\circ$ is a preorder, since similarity is a preorder and by definition of open extension. Given then that $\lesssim^\circ$ is reflexive, reflexivity of $\lesssim^\bullet$ follows by proving $\Gamma \vdash e \lesssim^\bullet e$ by structural induction on e. Rule (Cand Cong) follows at once from (Cand Def). Now the reflexivity of $\lesssim^\bullet$ means that $Id^\circ \subseteq$

$\lesssim^\bullet$. Compatible refinement is monotone, so if $Id = \{(a,b) \in Rel \mid a \equiv b\}$ we have $\widehat{Id^\circ} \subseteq \widehat{\lesssim^\bullet}$, which is to say $Id^\circ \subseteq \widehat{\lesssim^\bullet}$. Thus we have $\Gamma \vdash e \widehat{\lesssim^\bullet} e$ whenever $\Gamma \vdash e$, and hence (Cand Sim) follows from (Cand Def).

For (Cand Right), suppose that $\Gamma \vdash e \lesssim^\bullet e''$ and $\Gamma \vdash e'' \lesssim^\circ e'$. By (Cand Def) there must be an e''' with $\Gamma \vdash e \widehat{\lesssim^\bullet} e'''$ and $\Gamma \vdash e''' \lesssim^\circ e''$. By transitivity of $\lesssim^\circ$ we have $\Gamma \vdash e''' \lesssim^\circ e'$, and hence by (Cand Def) that $\Gamma \vdash e \lesssim^\bullet e'$ as required.

(Cand Subst) follows by a routine rule induction on the judgment $\Gamma, x{:}B \vdash e_1 \lesssim^\bullet e_1'$.

Finally suppose that $\mathcal{R}$ is another relation to satisfy both (Cand Cong) and (Cand Right). We must show that $\lesssim^\bullet \subseteq \mathcal{R}$. We prove by rule induction that $\Gamma \vdash e \lesssim^\bullet e'$ implies $\Gamma \vdash e \,\mathcal{R}\, e'$. Suppose then that $\Gamma \vdash e \lesssim^\bullet e'$, and hence by (Cand Def) that there is e'' with $\Gamma \vdash e \widehat{\lesssim^\bullet} e''$ and $\Gamma \vdash e'' \lesssim^\circ e'$. By the induction hypothesis we have $\Gamma \vdash e \widehat{\mathcal{R}} e''$. So by (Cand Cong) for $\mathcal{R}$ we have $\Gamma \vdash e \,\mathcal{R}\, e''$, and then by (Cand Right) for $\mathcal{R}$ that $\Gamma \vdash e \,\mathcal{R}\, e'$. ∎

The proof strategy is to show that $\lesssim^\circ \;=\; \lesssim^\bullet$, and then since $\lesssim^\bullet$ is a precongruence, (Cand Cong) 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 are the key lemmas. Let relation $\mathcal{S}$ be $\{(a,b) \mid \varnothing \vdash a \lesssim^\bullet b\}$, the restriction of $\lesssim^\bullet$ to programs.

Lemma 3.6 *Whenever $a \,\mathcal{S}\, b$ and $a \mapsto a'$ then $a' \,\mathcal{S}\, b$.*

Proof By rule induction on the relation $a \mapsto a'$. In any case we know from $a \,\mathcal{S}\, b$ and (Cand Def) there is a program c with $\varnothing \vdash a \widehat{\lesssim^\bullet} c \lesssim^\circ b$. We examine each rule in turn.

(Red Beta) Here a is $(\lambda(x{:}A)e)(a_2)$ and a' is $e[{}^{a_2}\!/\!{}_x]$. We start with an analysis of $\varnothing \vdash (\lambda(x)e)(a_2) \widehat{\lesssim^\bullet} c$.

- From (Comp Appl), c must be of the form $c_1(c_2)$ with $\varnothing \vdash \lambda(x)e \lesssim^\bullet c_1$ and $\varnothing \vdash a_2 \lesssim^\bullet c_2$.
- From (Cand Def) and (Comp Fun), there must be an intermediate $\lambda(x)e'$ with $\varnothing \vdash \lambda(x)e \widehat{\lesssim^\bullet} \lambda(x)e' \lesssim^\circ c_1$ and indeed $x{:}A \vdash e \lesssim^\bullet e'$.

We can fill in the diagram below starting from the similarity $\lambda(x)e' \lesssim c_1$, and using (Red Beta), Proposition 2.18(2) and (Cand Subst).

$$
\begin{tikzcd}
& \lambda(x)e' \arrow[d, "@c_2"'] & \lesssim & c_1 \arrow[d, "@c_2"] & \\
e[a_2/x] \lesssim^{\bullet} e'[c_2/x] \lesssim & (\lambda(x)e')(c_2) & \lesssim & c_1(c_2) & \lesssim b
\end{tikzcd}
$$

Now we have $a' \mathcal{S} b$ from (Cand Right).

(Red Rec) Here a is $a_1(a_2)$ where a_1 is $\texttt{rec}(f,x)e$, and a' is $e[a_1, a_2/f, x]$.

- From (Comp Appl), c takes the form $c_1(c_2)$ with $a_1 \lesssim^{\bullet} c_1$ and $a_2 \lesssim^{\bullet} c_2$.
- From (Cand Def) and (Comp Rec), there must be an intermediate $d \equiv \texttt{rec}(f,x)e'$ with $\varnothing \vdash a_1 \widehat{\lesssim^{\bullet}} d \lesssim^{\circ} c_1$.

As in the previous case we can fill in the following diagram.

$$
\begin{tikzcd}
& d \arrow[d, "@c_2"'] & \lesssim & c_1 \arrow[d, "@c_2"] & \\
e[a_1, a_2/f, x] \lesssim^{\bullet} e'[d, c_2/f, x] \lesssim & d(c_2) & \lesssim & c_1(c_2) & \lesssim b
\end{tikzcd}
$$

Hence $a' \mathcal{S} b$ from (Cand Right).

(Red If) Here a is $\texttt{if}\,\underline{bv}\,\texttt{then}\,a_{tt}\,\texttt{else}\,a_{f\!f}$, with $bv \in \{tt, f\!f\}$, and a' is a_{bv}. As before we analyse $\varnothing \vdash \texttt{if}\,\underline{bv}\,\texttt{then}\,a_{tt}\,\texttt{else}\,a_{f\!f} \widehat{\lesssim^{\bullet}} c$.

- From (Comp If), c takes the form $\texttt{if}\,c_1\,\texttt{then}\,c_{tt}\,\texttt{else}\,c_{f\!f}$ with $\varnothing \vdash \underline{bv} \lesssim^{\bullet} c_1$, $\varnothing \vdash a_{tt} \lesssim^{\bullet} c_{tt}$ and $\varnothing \vdash a_{f\!f} \lesssim^{\bullet} c_{f\!f}$.
- From (Cand Def) and (Comp Lit) it follows that $c_1 \sim \underline{bv}$. Hence $c_1 \Downarrow \underline{bv}$ and $c \mapsto^{*} c_{bv}$.

We have $\varnothing \vdash a' \equiv a_{bv} \lesssim^{\bullet} c_{bv} \lesssim c \lesssim b$, and so $a' \mathcal{S} b$ by (Cand Right).

(Red Experiment) Here a is $\mathcal{E}[a_1]$, such that $a_1 \mapsto a_1'$ and a' is $\mathcal{E}[a_1']$. Given $\varnothing \vdash \mathcal{E}[a_1] \widehat{\lesssim^{\bullet}} c$, whatever the form of $\mathcal{E}$ we can decompose c into the form $\mathcal{E}'[c_1]$ with $\varnothing \vdash a_1 \lesssim^{\bullet} c_1$ and $\varnothing \vdash \mathcal{E} \widehat{\lesssim^{\bullet}} \mathcal{E}'$. Since $a_1 \mapsto a_1'$ and $a_1 \mathcal{S} c_1$

we have $a'_1 \mathrel{\mathcal{S}} c_1$ by induction hypothesis. By considering the possible forms of $\mathcal{E}$, we have $\varnothing \vdash a' \equiv \mathcal{E}[a'_1] \mathrel{\widehat{\lesssim^\bullet}} \mathcal{E}'[c_1]$, and hence $\varnothing \vdash a' \mathrel{\widehat{\lesssim^\bullet}} c \lesssim^\circ b$, that is, $a' \mathrel{\mathcal{S}} b$ as required.

The other cases, for streams and arithmetic, are similar. ■

Lemma 3.7 *If $a \mathrel{\mathcal{S}} b$ and $a \xrightarrow{\alpha} a'$ there is b' with $b \xrightarrow{\alpha} b'$ and $a' \mathrel{\mathcal{S}} b'$.*

Proof By rule induction on the derivation of $a \xrightarrow{\alpha} a'$. In any case we know from $a \mathrel{\mathcal{S}} b$ and (Cand Def) there is a program c with $\varnothing \vdash a \mathrel{\widehat{\lesssim^\bullet}} c \lesssim^\circ b$.

(Trans Lit) Here a is $\underline{\ell}$, α is ℓ, and a' is $\mathbf{0}$. From (Comp ℓ) c must equal $\underline{\ell}$, and then from $\underline{\ell} \lesssim b$ it follows that $b \xrightarrow{\alpha} \mathbf{0}$ too, and certainly $\mathbf{0} \mathrel{\mathcal{S}} \mathbf{0}$, as $\mathcal{S}$ is reflexive.

(Trans Fun) Here a is of function type, α is $@c$ and a' is $a(c)$. Since b is of the same function type, (Trans Fun Passive) admits the transition $b \xrightarrow{\alpha} b(c)$, and since $\lesssim^\bullet$ is a precongruence, (Cand Cong), we have $a(c) \mathrel{\mathcal{S}} b(c)$ as required.

(Trans Red) Here $a \xrightarrow{\alpha} a'$ was derived from $a \mapsto a''$ and $a'' \xrightarrow{\alpha} a'$. By Lemma 3.6 we have $a'' \mathrel{\mathcal{S}} b$, so by induction hypothesis we have $b \xrightarrow{\alpha} b'$ with $a' \mathrel{\mathcal{S}} b'$, as required.

All cases considered, the result follows. ■

Theorem 3.8 *The open extension of similarity is a precongruence.*

Proof By Lemma 3.7, $\mathcal{S}$ is a simulation, and hence $\mathcal{S} \subseteq \lesssim$ by co-induction. Open extension is monotone, so $\mathcal{S}^\circ \subseteq \lesssim^\circ$. Now $\lesssim^\bullet \subseteq \mathcal{S}^\circ$ follows from the substitution lemma (Cand Subst) and the reflexivity of $\lesssim^\bullet$ (Lemma 3.5). Hence we have $\lesssim^\bullet \subseteq \lesssim^\circ$. Since (Cand Sim) furnishes the reverse inclusion, we have $\lesssim^\bullet = \lesssim^\circ$ and hence $\lesssim^\circ$ is a precongruence. ■

3.3 Bisimilarity equals Contextual Equivalence

Given that bisimilarity and similarity are precongruences, it is now easy to show that bisimilarity equals contextual equivalence.

Lemma 3.9 *Both $\lesssim \subseteq \sqsubseteq$ and $\sim \subseteq \simeq$.*

Proof First we shall prove $\lesssim \subseteq \sqsubseteq\!\!\!\sim$. Suppose that $a \lesssim b{:}A$. To see that $a \sqsubseteq\!\!\!\sim b$ we must consider an arbitrary expression e such that $-{:}A \vdash e : \mathtt{Bool}$ and show that $e[a]\Downarrow$ implies $e[b]\Downarrow$. By supposition we have $\varnothing \vdash a \lesssim^\circ b{:}A$, and so by the congruence property of $\lesssim^\circ$ (in fact just (Cand Subst)) it follows that $\varnothing \vdash e[a] \lesssim^\circ e[b] : \mathtt{Bool}$. In fact we have $e[a] \lesssim e[b]$. Now $e[a] \Downarrow \underline{bv}$ with $bv \in \{tt, \mathit{ff}\}$. So $e[a] \xrightarrow{bv} \mathbf{0}$. Since $e[a] \lesssim e[b]$ it must be that $e[b] \xrightarrow{bv} \mathbf{0}$ too, and therefore $e[b]\Downarrow$, as required, by Lemma 2.15. Having now shown that $\lesssim \subseteq \sqsubseteq\!\!\!\sim$, $\sim \subseteq \simeq$ follows by symmetry. ■

Lemma 3.10 *If type A is active there is a Boolean context $-{:}A \vdash e : \mathtt{Bool}$ such that for any $a{:}A$, $e[a]\Downarrow$ iff $a\Downarrow$.*

Proof Witness the following contexts.

`Bool`	$-$
`Num`	$\mathtt{zero}(-)$
`[`A`]`	$\mathtt{scase}(-, \mathtt{true}, \lambda(x)\lambda(xs)\mathtt{true})$

■

Lemma 3.11 *Suppose $a{:}A$ and A is active. If $a \sqsubseteq\!\!\!\sim b$ and $a \Downarrow u$ there is v with $b \Downarrow v$ and $u \sqsubseteq\!\!\!\sim v$.*

Proof A corollary of Lemma 3.10 is that $b\Downarrow$ follows from $a \sqsubseteq\!\!\!\sim b$ and $a\Downarrow$. The result then follows from $\Downarrow \subseteq \sim$ and Lemma 3.9. ■

Lemma 3.12 *Contextual order is a simulation.*

Proof Suppose then that $a{:}A$, $a \sqsubseteq\!\!\!\sim b$ and that $a \xrightarrow{\alpha} a'$. To show that $\sqsubseteq\!\!\!\sim$ is a simulation we must find some b' such that $b \xrightarrow{\alpha} b'$ with $a' \sqsubseteq\!\!\!\sim b'$.

Suppose first that A is active. So there must be an intermediate value v such that $a \Downarrow v$ and $v \xrightarrow{\alpha} a'$. By Lemma 3.11 there is a program u such that $b \Downarrow u$ and $v \sqsubseteq\!\!\!\sim u$. We proceed to analyse the derivation of $v \xrightarrow{\alpha} b$. It suffices to show that $u \xrightarrow{\alpha} b'$ with $a' \sqsubseteq\!\!\!\sim b'$, for then we can infer that $b \xrightarrow{\alpha} b'$ too. We examine some typical cases.

(Trans Lit) Here v is a literal $\underline{\ell}$, α is ℓ and a' is $\mathbf{0}$. We know $\underline{\ell} \sqsubseteq\!\!\!\sim u$. It must be that u is $\underline{\ell}$ or we could tell them apart using a conditional or an equality test. So setting b' to be $\mathbf{0}$, we have $u \xrightarrow{\alpha} b'$ and $a' \sqsubseteq\!\!\!\sim b'$.

(Trans Hd) Here v is $a_1 :: a_2$, α is $\mathtt{hd}$ and a' is a_1. We know $a_1 :: a_2 \sqsubseteq\!\!\!\sim u$ so using `scase` to test both sides we can establish that u must be $b_1 :: b_2$ with $a_i \sqsubseteq\!\!\!\sim b_i$ for each i. Let b' be b_1 and we have $u \xrightarrow{\alpha} b'$ with $a' \sqsubseteq\!\!\!\sim b'$.

Secondly we must consider the ways $a \xrightarrow{\alpha} a'$ can be derived if A is passive. In PCF plus streams only one rule applies.

(Trans Fun) Here a and b are of function type, α is $@c$ and a' is $a(c)$. (Trans Fun) admits the transition $b \xrightarrow{\alpha} b(c)$ too. Now contextual order is closed under contexts, so by appeal to the context $-(c)$ we have $a(c) \sqsubseteq\!\!\!\sim b(c)$, as required.

All cases considered, it follows that contextual order is a simulation. ∎

By combining Lemmas 3.12 and 3.9 we have that bisimilarity equals contextual equivalence.

Theorem 3.13 $\sim\, = \,\simeq$.

Here is a simple application. In Definition 2.7, we defined contextual equivalence on programs using expressions with a single variable, '$-$', as contexts. A common variation is to define it on open expressions using variable-binding contexts. We can use Theorem 3.13 to show the two versions are one.

Proposition 3.14 *Supposing that*

(1) $\simeq^\circ$ *is a congruence*

(2) $\mapsto \subseteq \simeq$

then $\Gamma \vdash e_1 \simeq^\circ e_2$ *iff for any variable-capturing context* $\mathcal{C}$ *with* $\mathcal{C}[e_1], \mathcal{C}[e_2]$:`Bool` *that* $\mathcal{C}[e_1]\Downarrow$ *iff* $\mathcal{C}[e_2]\Downarrow$.

Proof

(Only If) Suppose that $\Gamma \vdash e_1 \simeq^\circ e_2$. By Lemma 3.3, (1) and (Cong $\simeq^\circ$), $\varnothing \vdash \mathcal{C}[e_1] \simeq^\circ \mathcal{C}[e_2]$: `Bool` and therefore $\mathcal{C}[e_1]\Downarrow$ iff $\mathcal{C}[e_2]\Downarrow$.

(If) Suppose that $[{}^{a_1,\ldots,a_n}\!/\!{}_{x_1,\ldots,x_n}]$ is an arbitrary Γ-closure. We must establish that $e_1[\vec{a}/\vec{x}] \simeq e_2[\vec{a}/\vec{x}]$. Let variable-binding context $\mathcal{C}$ be

$$(\lambda(x_1)\cdots\lambda(x_n){-})(a_1)\cdots(a_n).$$

We have $\mathcal{C}[e_i] \mapsto^n e_i[\vec{a}/\vec{x}]$ for each i, and hence by (2) that $\mathcal{C}[e_i] \simeq e_i[\vec{a}/\vec{x}]$ for each i. Suppose then that $e[e_1[\vec{a}/\vec{x}]]\Downarrow$ for some (closed) context e. We must show that $e[e_2[\vec{a}/\vec{x}]]\Downarrow$ too. We have $e[e_1[\vec{a}/\vec{x}]] \simeq e[\mathcal{C}[e_1]$ since $\simeq$ is certainly preserved by a closed context such as e, so $e[\mathcal{C}[e_1]\Downarrow$. Since $e[\mathcal{C}]$ is itself a variable-capturing context of `Bool` type, we have $e[\mathcal{C}[e_2]\Downarrow$ by hypothesis. Now $e[\mathcal{C}[e_2]] \simeq e[e_2[\vec{a}/\vec{x}]]$, and therefore $e[e_2[\vec{a}/\vec{x}]]\Downarrow$, as required. ∎

Hypotheses (1) and (2) follow from Theorem 3.13.

3.4 Experimental Equivalence

We 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 (1977) showed that contextual equivalence on a combinatory-logic form of PCF is unchanged if we restrict attention to 'applicative contexts' of the form $- a_1 \ldots a_n$. Berry (1981) extended Milner's proof to the lambda-calculus form of PCF. Here the analogue of an applicative context 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

$$a \approx b{:}A \quad \text{iff} \quad \text{whenever } -{:}A \vdash \vec{\mathcal{E}}[-] : \texttt{Bool}, \text{ that } \vec{\mathcal{E}}[a]\Downarrow \text{ iff } \vec{\mathcal{E}}[b]\Downarrow.$$

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

Lemma 3.15 *Suppose $a \approx b{:}A$ and that $-{:}A \vdash e : \texttt{Bool}$. If $e[a]\Downarrow$ in n steps, then $e[b]\Downarrow$ too.*

Proof Similar to Lemma 4.17 on page 46 of Gordon (1994). ■

This property of a language, that experimental equivalence implies contextual equivalence, was named **operational extensionality** by Bloom (1988). (Bloom called $\approx$ 'applicative congruence'.) An easy corollary is that $\approx = \simeq$. Evaluation contexts make the same distinctions as full-blown contexts. Since it is straightforward to prove that $\mapsto \subseteq \approx$, for instance, experimental equivalence is a useful definition for establishing equational properties of contextual equivalence, independently of bisimilarity.

Lemma 3.16 $\mapsto \subseteq \approx$

Proof Suppose that $a \mapsto b$. We must show for any evaluation context $\vec{\mathcal{E}}$ that if $\vec{\mathcal{E}}[a]\Downarrow$ then so does $\vec{\mathcal{E}}[b]\Downarrow$, and the converse. Suppose that $\vec{\mathcal{E}}[a] \Downarrow v$, that is, $\vec{\mathcal{E}}[a] \mapsto^* v$. Program $\vec{\mathcal{E}}[a]$ cannot be a value, so there must be a' with $\vec{\mathcal{E}}[a] \mapsto a' \mapsto^* v$. But because $\vec{\mathcal{E}}[-]$ is an evaluation context and reduction is deterministic, it must be that $a' \equiv \vec{\mathcal{E}}[b]$, and a' converges. The converse follows easily. ■

Lemma 3.17 *If a, b:`Bool` or `Num` then $a \approx b$ iff $\forall \ell (a \Downarrow \underline{\ell} \Leftrightarrow b \Downarrow \underline{\ell})$.*

Proof

(Only If) Suppose that $a \approx b$ and $a \Downarrow \underline{\ell}$. For a contradiction, suppose either that $b\Uparrow$ or $b \Downarrow \underline{\ell'}$ with $\ell \neq \ell'$. The former gives rise to a contradiction immediately. For the latter, we may construct an evaluation context $\vec{\mathcal{E}}$, coding literal equality using `pred`, `zero` and `if`, such that $\vec{\mathcal{E}}[c]\Downarrow$ iff $c \Downarrow \underline{\ell}$. So $\vec{\mathcal{E}}[a]\Downarrow$ while $\vec{\mathcal{E}}[b]\Uparrow$. Contradiction.

(If) Either $a\Uparrow$ and $b\Uparrow$, or $a \Downarrow \underline{\ell}$ and $b \Downarrow \underline{\ell}$ for some literal ℓ. In the former case $a \approx b$ easily. In the latter case, suppose that $\vec{\mathcal{E}}[a]\Downarrow$. So $\vec{\mathcal{E}}[\underline{\ell}]\Downarrow$ too, and hence $\vec{\mathcal{E}}[b]\Downarrow$. ■

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:`Bool` that $a\Downarrow$ iff $b\Downarrow$;

(2) whenever $\mathcal{E}[a]$ and $\mathcal{E}[b]$ are well-typed, $(\mathcal{E}[a], \mathcal{E}[b]) \in \mathcal{S}$.

Proposition 3.18 *$\approx\; = \nu\mathcal{S}.\, F(\mathcal{S})$.*

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, first note that since $\nu = F(\nu)$, it follows that whenever $(a, b) \in \nu$ and $\mathcal{E}[a]$ and $\mathcal{E}[b]$ are well-typed, that $(\mathcal{E}[a], \mathcal{E}[b]) \in \nu$ too. Suppose then that $(a, b) \in \nu$, $\vec{\mathcal{E}}[a], \vec{\mathcal{E}}[b]$:`Bool` 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.

Proposition 3.19 *$\approx\; = \nu\mathcal{S}.\, F(\approx\mathcal{S}\approx)$.*

Proof Let ν be $\nu\mathcal{S}.\, F(\approx\mathcal{S}\approx)$. Since $\approx\; = \nu\mathcal{S}.\, F(\mathcal{S})$ it follows that $\approx\; \subseteq F(\approx) \subseteq F(\approx\approx\approx)$ (by transitivity of $\approx$ and monotonicity of F) and so $\approx\; \subseteq \nu$ by co-induction.

For the other direction, we shall show that $\approx\nu\approx$ is F-dense, and hence that $\approx\nu\approx\; \subseteq\; \approx$, and hence that $\nu \subseteq\; \approx$, since $Id \subseteq\; \approx$. Suppose then that $a \approx\nu\approx b$, that is, there are a' and b' with $a \approx a' \;\nu\; b' \approx b$.

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

(1) Suppose that a, b:`Bool` and $a\Downarrow$. We must show that $b\Downarrow$ too. It must be that a', b':`Bool` too, so since $a \approx a'$, $a'\Downarrow$. Now since $\nu \subseteq F(\approx\nu\approx)$ it follows that $b'\Downarrow$ too, and therefore $b\Downarrow$.

(2) Suppose that $\mathcal{E}[a]$ and $\mathcal{E}[b]$ are well-typed. We have $\mathcal{E}[a] \approx \mathcal{E}[a']$ from $a \approx b$, then $\mathcal{E}[a']{\approx}\nu{\approx}\mathcal{E}[b']$, since $\nu \subseteq F(\approx\nu\approx)$, and $\mathcal{E}[b'] \approx \mathcal{E}[b]$, as $b' \approx b$. In all $\mathcal{E}[a]{\approx}\nu{\approx}\mathcal{E}[b]$.

Hence $(a, b) \in F(\approx\nu\approx)$. ■

The proof is a variation on the proof that in CCS a 'bisimulation up to $\sim$' is contained in bisimilarity (see Proposition 4.1). 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 $(a_1 :: a_2, b_1 :: b_2)$; we must show that $\mathcal{E}[a_1 :: a_2] \approx\mathcal{S}\approx \mathcal{E}[b_1 :: b_2]$ for all suitable experiments, $\mathcal{E}$, which include the following

$$\texttt{scase}(-, b, \lambda(x)\lambda(xs)e).$$

Hence we must show $e[^{a_1, a_2}/_{x, xs}] \approx\mathcal{S}\approx e[^{b_1, b_2}/_{x, xs}]$ which, because of the quantification over the arbitrary expression e is almost as hard as proving contextual equivalence directly, and certainly harder than proving each $(a_i, b_i) \in \mathcal{S}$, 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.

4 Refining Bisimulation

In the theory of CCS, there are various forms of 'bisimulation up to $\sim$' that allow simpler proofs by co-induction than plain bisimulation. Our main objective here is to develop co-induction principles that simplify co-induction proofs by exploitation of the determinacy of PCF plus streams. There are two ideas: matching values (Section 4.2) and matching reductions (Section 4.4). Taken together they admit a co-inductive characterisation of bisimilarity without mention of transitions, only equations and reductions.

The Take Lemma of Bird and Wadler (1988) is a well-known technique for proving equivalence of two streams by inducting over their finite approximations. We derive the Take Lemma in Section 4.6 but show that it is less useful than the method of matching values and reductions.

4.1 Bisimulations up to $\sim$

Milner's idea of 'bisimulation up to $\sim$' works for any labelled transition system. Suppose $\mathcal{S} \subseteq Rel$. Let $\mathcal{S}$ be a **bisimulation up to** $\sim$ iff $a\,\mathcal{S}\,b$ implies

(1) whenever $a \xrightarrow{\alpha} a'$ there is b' such that $b \xrightarrow{\alpha} b'$ and $a' \sim\mathcal{S}\sim b'$; and

(2) whenever $b \xrightarrow{\alpha} b'$ there is a' such that $a \xrightarrow{\alpha} a'$ and $a' \sim\mathcal{S}\sim b'$.

In relational terms, $\mathcal{S} \subseteq \langle\sim\mathcal{S}\sim\rangle$. See Milner (1989, p93) for the standard proof of the following.

Proposition 4.1 (Milner) *If $\mathcal{S}$ is a bisimulation up to $\sim$ then $\sim\mathcal{S}\sim \subseteq \langle\sim\mathcal{S}\sim\rangle$, that is, $\sim\mathcal{S}\sim$ is a bisimulation.*

Corollaries are that $\sim\mathcal{S}\sim \subseteq \sim$ and $\mathcal{S} \subseteq \sim$. Since bisimilarity is trivially a bisimulation up to $\sim$ it follows that $\sim = \nu\mathcal{S}.\langle\sim\mathcal{S}\sim\rangle$. Milner's result is a co-inductive characterisation of bisimilarity. In the theory of CCS it often results in simpler proofs than plain bisimulation. But proofs via a bisimulation up to $\sim$ still require calculation of transitions.

4.2 Matching Values

In our functional programming context, it is tedious to check all the transitions of an expression, as in the proof of the `map/iterate` example, Lemma 2.17, because, in Strachey's (1967) wonderful phrase, the "characteristic feature of an expression is its value." All that matters about an expression is its value. We want a proof principle that for each pair (a, b) in the candidate relation $\mathcal{S}$ requires us just to find values for each of a and b—rather than find their transitions—such that the values have the same outermost form (both :: for instance) and have immediate subexpressions related by $\mathcal{S}$.

We begin by restricting compatible refinement to values. Let *Value* be the set of values, that is, closed programs given by the (Value $-$) rules. If $\mathcal{S} \subseteq Rel$, we define $\overline{\mathcal{S}}$, $\langle \mathcal{S} \rangle$ and ν_V, all subsets of Rel, as follows.

$$\begin{aligned} \overline{\mathcal{S}} &\stackrel{\text{def}}{=} \{(u,v) \in \mathit{Value} \times \mathit{Value} \mid \varnothing \vdash u\, \widehat{\mathcal{S}^\circ}\, v\} \\ \langle \mathcal{S} \rangle_V &\stackrel{\text{def}}{=} \sim\overline{\mathcal{S}}\sim \\ \nu_V &\stackrel{\text{def}}{=} \nu\mathcal{S}.\, \langle \mathcal{S} \rangle_V \end{aligned}$$

In elementwise terms, we have that ν_V equals the greatest binary relation $\mathcal{S} \subseteq Rel$ such that

$$a\, \mathcal{S}\, b \quad \Leftrightarrow \quad \exists u, v (a \sim u\, \widehat{\mathcal{S}^\circ}\, v \sim b).$$

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'. We can prove that ν_V approximates bisimilarity. As a notational convenience, if $\mathcal{R} \subseteq Rel$ we shall write $\mathcal{R}_\sim$ to mean the composition $\sim\mathcal{R}\sim$.

Lemma 4.2 *For any $\mathcal{S} \subseteq Rel$, $\overline{\mathcal{S}} \subseteq [\mathcal{S}_\sim \cup Id]$.*

Proof If $a\, \overline{\mathcal{S}}\, b$ we must show that whenever $a \xrightarrow{\alpha} a'$ there is b' such that $b \xrightarrow{\alpha} b'$ and $(a', b') \in \mathcal{S}_\sim \cup Id$. In fact a and b must be values satisfying $\varnothing \vdash u\, \widehat{\mathcal{S}^\circ}\, v$, that is, with the same outermost form and their immediate subexpressions related by $\mathcal{S}^\circ$. We consider the various kinds of values.

(Value Arith) Here both a and b are $\underline{\ell}$. The only transition of a is $a \xrightarrow{\ell} \mathbf{0}$, which is matched by $b \xrightarrow{\ell} \mathbf{0}$ and we have $(\mathbf{0}, \mathbf{0}) \in Id$.

(Value Fun) Here a is $\lambda(x{:}A)e$ and b is $\lambda(x{:}A)e'$, such that $e[{}^c\!/\!_x]\, \mathcal{S}\, e'[{}^c\!/\!_x]$ for any $c{:}A$. Each transition of a takes the form $a \xrightarrow{@c} a(c)$, which can be matched by $b \xrightarrow{@c} b(c)$. We can calculate $a(c) \sim e[{}^c\!/\!_x]\, \mathcal{S}\, e'[{}^c\!/\!_x] \sim b(c)$ and so $(a(c), b(c)) \in \mathcal{S}_\sim$.

In the other possibilities, (Value Stm) and (Value Rec), we can argue similarly. ∎

Lemma 4.3 *If $\mathcal{S} \subseteq \langle \mathcal{S} \rangle_V$, then $\mathcal{S}_\sim \subseteq \sim$.*

Proof If $(a, b) \in \mathcal{S}_\sim$, there must be a_0 and b_0 with $a \sim a_0 \, \mathcal{S} \, b_0 \sim b$. If we suppose $\mathcal{S} \subseteq \langle \mathcal{S} \rangle_V$ there must be u and v with $a_0 \sim u \, \overline{\mathcal{S}} \, v \sim b_0$. For any transition $a \xrightarrow{\alpha} a'$, given Lemma 4.2, we can fill in the following diagram from left to right.

$$\begin{tikzcd}
a \arrow[r, phantom, "\sim"] \arrow[d, "\alpha"'] & a_0 \arrow[r, phantom, "\sim"] \arrow[d, "\alpha"'] & u \arrow[r, phantom, "\overline{\mathcal{S}}"] \arrow[d, "\alpha"'] & v \arrow[r, phantom, "\sim"] \arrow[d, "\alpha"] & b_0 \arrow[r, phantom, "\sim"] \arrow[d, "\alpha"] & b \arrow[d, "\alpha"] \\
a' \arrow[r, phantom, "\sim"] & a'_0 \arrow[r, phantom, "\sim"] & a'' \arrow[r, phantom, "(\mathcal{S}_\sim \cup Id)"] & b'' \arrow[r, phantom, "\sim"] & b'_0 \arrow[r, phantom, "\sim"] & b'
\end{tikzcd}$$

We have $(a', b') \in (\mathcal{S}_\sim \cup \sim)$, from which $(a, b) \in [\mathcal{S}_\sim \cup \sim]$. Indeed by a symmetric argument we have $(a, b) \in \langle \mathcal{S}_\sim \cup \sim \rangle$. By strong co-induction we have $\mathcal{S}_\sim \subseteq \sim$ as required. ■

Proposition 4.4 $\nu_V \subset \sim$

Proof Since ν_V is $\langle - \rangle_V$ dense, we have $\sim\nu_V\sim \subseteq \sim$ by Lemma 4.3. 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 $\mathbf{0}$. ■

We can use ν_V to refine the proof of Lemma 2.17. Our original proof explicitly considered all the transitions of the programs in relation $\mathcal{S}$. If we make the assumption that $(f^n(c), (f^n(c)) \in \nu_V$, it is not hard to check that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \nu_V \rangle_V$, and hence by strong co-induction that $\mathcal{S} \subseteq \nu_V \subseteq \sim$. This proof avoids having to explicitly calculate transitions. The reason for the restriction on each $f^n \, c$ is essentially that ν_V is an incomplete co-inductive characterisation of $\sim$.

4.3 The Map/Filter Example

The idea of matching values is a step forward but it has shortcomings. Suppose `filter` is the usual function that deletes the elements of a stream that fail a predicate.

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

For any functions $f{:}B \to$ `Bool` and $g{:}A \to B$, `filter` ought to commute with `map` as follows, where `o` function composition.

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

For arbitrary f and g make the following definitions.

$$\begin{array}{rcl} e & \equiv & \mathtt{filter}\, f\, (\mathtt{map}\, g, -) \\ e' & \equiv & \mathtt{map}\, g\, (\mathtt{filter}\, (f \circ g) -) \\ \mathcal{S} & = & \{(e[cs], e'[cs]) \mid cs\colon \mathtt{[}A\mathtt{]}\} \end{array}$$

Equation $(*)$ will follow by (Fun Ext) if we can show that $\mathcal{S} \subseteq \sim$.

Lemma 4.5 *Whenever $e[cs]\, \mathcal{S}\, e'[cs]$ and $e[cs] \Downarrow v$ in n steps, there is u with $e'[cs] \Downarrow u$ and $v\, \overline{\mathcal{S}'}\, u$ if $\mathcal{S}' = \mathcal{S} \cup \sim$.*

Proof By mathematical induction on n. Suppose that $e[cs]\, \mathcal{S}\, e'[cs]$ and that $e[cs] \Downarrow v$ in n steps. Consider the evaluation behaviour of cs. Either (A) $cs\Uparrow$, (B) $cs \Downarrow \mathtt{nil}$ or (C) $cs \Downarrow c :: cs'$. Case (A) contradicts our assumption $e[cs] \Downarrow v$. In case (B) both $e[cs] \Downarrow \mathtt{nil}$ and $e'[cs] \Downarrow \mathtt{nil}$ so the hypothesis holds. In case (C) $e[cs]$ and $e'[cs]$ reduce as follows.

$$\begin{array}{rcl} e[cs] & \mapsto^{+} & \mathtt{if}\, f(g\, c)\, \mathtt{then}\, g(c) :: e[cs']\, \mathtt{else}\, e[cs'] \\ e'[cs] & \mapsto^{+} & \mathtt{if}\, f(g\, c)\, \mathtt{then}\, g(c) :: e'[cs']\, \mathtt{else}\, e'[cs'] \end{array}$$

Consider the evaluation behaviour of $f(g(c))$. Either (CA) $f(g\, c)\Uparrow$, (CB) $f(g\, c) \Downarrow \mathtt{true}$ and (CC) $f(g\, c) \Downarrow \mathtt{false}$. Again (CA) is contradictory. In case (CB) let

$$\begin{array}{rcl} u & \equiv & g(c) :: e[cs'] \\ v & \equiv & g(c) :: e'[cs'] \end{array}$$

and we have $e[cs] \Downarrow u$, $e'[cs] \Downarrow v$ and $v\, \overline{\mathcal{S}'}\, u$ as required. In case (CC) we have

$$\begin{array}{rcl} e[cs] & \mapsto^{+} & e[cs'] \\ e'[cs] & \mapsto^{+} & e'[cs']. \end{array}$$

By induction hypothesis there are values u and v with $e[cs'] \Downarrow u$, $e'[cs'] \Downarrow v$ and $v\, \overline{\mathcal{S}'}\, u$. But $e[cs] \Downarrow u$ and $e'[cs] \Downarrow v$ too. ■

A symmetric argument shows that the lemma holds for $\mathcal{S}^{\mathrm{op}}$ too. We can deduce that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle_V \cup \sim$. We cannot apply strong co-induction though, because $\sim\, \neq \nu_V$. But by calculating transitions we can see that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle \cup \sim$ and then $(*)$ does follow by strong co-induction.

Applying the matching values idea to a candidate relation such as $\mathcal{S}$ is problematic because not all the programs in $\mathcal{S}$ have a value, and furthermore, as in case (CC) not all of them immediately produce a value. Other examples also require an induction on the evaluation of a program. The next idea, matching reductions, caters for divergence in the candidate relation and in a sense incorporates the induction used in Lemma 4.5 into a co-induction principle, once and for all.

4.4 Matching Reductions

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

$$a \, \langle \mathcal{S} \rangle_+ \, b \quad \Leftrightarrow \quad \exists a', b'(a \mapsto^+ a', b \mapsto^+ b' \;\&\; a' \, \mathcal{S} \, b')$$

These are both monotone maps. 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 4.6 *Let* $\nu_+ \stackrel{\text{def}}{=} \nu\mathcal{S}.\, \langle \mathcal{S} \rangle_+$.

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

(2) $\nu_+ \subset \sim$.

Proof Recall the divergence functionals $\mathcal{D}$ and $\mathcal{D}_+$ from Section 2.3. Consider the following three sets.

$$\begin{aligned} X &\stackrel{\text{def}}{=} \{a \mid \exists b((a, b) \in \nu_+)\} \\ Y &\stackrel{\text{def}}{=} \{b \mid \exists a((a, b) \in \nu_+)\} \\ \mathcal{S} &\stackrel{\text{def}}{=} \{(a, b) \mid a \in \nu X.\, \mathcal{D}_+(X) \;\&\; b \in \nu X.\, \mathcal{D}_+(X)\} \end{aligned}$$

One can easily check that X and Y are both $\mathcal{D}$ dense, and that $\mathcal{S}$ is $\langle - \rangle_+$ dense. Hence by co-induction we have $X \subseteq \Uparrow$, $Y \subseteq \Uparrow$ and $\mathcal{S} \subseteq \nu_+$. Part (1) then follows. Inclusion $\nu_+ \subseteq \sim$ follows from (1); it is strict because of convergent programs: if $a\Downarrow$ then $(a, a) \notin \nu_+$ although $a \sim a$. ■

The rest of this section is devoted to the proof that bisimilarity equals $\nu\mathcal{S}.\, \langle \mathcal{S} \rangle_V \cup \langle \mathcal{S} \rangle_+$. The proof applies to all the languages considered in this paper. It depends on bisimilarity being an equivalence and general facts from Section 2.6. The point of combining matching values and matching reductions in this equation is to simplify earlier proofs.

- Consider the `map/iterate` example, Lemma 2.17. We can easily check that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle_V$. So $\mathcal{S} \subseteq \sim$ follows by strong co-induction with no restriction on each $f^n\, c$.

- Consider the `map/filter` example from Section 4.3. A simple modification of the argument in Lemma 4.5 shows that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle \cup \langle \mathcal{S} \rangle_+ \cup \sim$ but with no need for an induction on n. Hence $\mathcal{S} \subseteq \sim$. In case (CC) we have that $(e[cs], e'[cs]) \in \langle \mathcal{S} \rangle_+$.

Lemma 4.7 *Suppose* F *is a monotone function on* $\wp(Rel)$. *Then* $\nu X.\, F(X) = \nu X.\, F(X) \cup F(F(X))$.

Proof Let $\nu_1 = \nu X.\, F(X)$ and $\nu_2 = \nu X.\, F(X) \cup F(F(X))$. Since $\nu_1 = F(\nu_1)$ we have $\nu_1 \subseteq F(\nu_1) \cup F(F(\nu_1))$ and therefore $\nu_1 \subseteq \nu_2$. Since $F(\nu_2) \subseteq \nu_2$ (as $\nu_2 = F(\nu_2) \cup F(F(\nu_2)))$ we have $F(F(\nu_2)) \subseteq F(\nu_2)$ by monotonicity. So $\nu_2 = F(\nu_2) \cup F(F(\nu_2)) = F(\nu_2)$ and therefore $\nu_2 \subseteq \nu_1$. Hence $\nu_1 = \nu_2$. ■

Lemma 4.8 *Suppose F is a monotone function on $\wp(Rel)$. If*

$$\begin{aligned} \nu_1 &\stackrel{\text{def}}{=} \nu\mathcal{S}.\, F(\mathcal{S}) \cup \langle \mathcal{S} \rangle_+ \\ \nu_2 &\stackrel{\text{def}}{=} \nu\mathcal{S}.\, F(\mathcal{S}) \cup \langle F(\mathcal{S}) \rangle_+ \cup \nu_+ \end{aligned}$$

then $\nu_1 = \nu_2$.

Proof We begin with the easier inclusion, $\nu_2 \subseteq \nu_1$. First note that $\nu_+ \subseteq \nu_1$, because $\nu_+ \subseteq \langle \nu_+ \rangle_+$. By Lemma 4.7, ν_1 unwinds to

$$\nu_1 = \nu\mathcal{S}.\, F(\mathcal{S}) \cup \langle \mathcal{S} \rangle_+ \cup F(F(\mathcal{S}) \cup \langle F(\mathcal{S}) \rangle_+) \cup \langle F(\mathcal{S}) \cup \langle F(\mathcal{S}) \rangle_+ \rangle_+.$$

We shall show that ν_2 is dense with respect to the functional on the right-hand side. Starting from

$$\nu_2 \subseteq F(\nu_2) \cup \langle F(\nu_2) \rangle_+ \cup \nu_+$$

we obtain

$$\nu_2 \subseteq F(\nu_2) \cup \langle F(\nu_2) \cup \langle F(\nu_2) \rangle_+ \rangle_+ \cup \nu_1$$

from monotonicity of $\langle - \rangle_+$ and $\nu_+ \subseteq \nu_1$. From this inequation it follows by strong co-induction that $\nu_2 \subseteq \nu_1$. For the other direction we shall prove that

(i) $\mathcal{S}_\Omega \stackrel{\text{def}}{=} \{(a, b) \in \nu_1 \mid a\Uparrow \,\&\, b\Uparrow\} \subseteq \nu_+$

(ii) $\mathcal{S}_1 \stackrel{\text{def}}{=} \{(a, b) \in \nu_1 \mid a\Downarrow\} \subseteq F(\nu_1) \cup \langle F(\nu_1) \rangle_+$

(iii) $\mathcal{S}_2 \stackrel{\text{def}}{=} \{(a, b) \in \nu_1 \mid b\Downarrow\} \subseteq F(\nu_1) \cup \langle F(\nu_1) \rangle_+$

and hence it will follow that

$$\nu_1 = \mathcal{S}_\Omega \cup \mathcal{S}_1 \cup \mathcal{S}_2 \subseteq F(\nu_1) \cup \langle F(\nu_1) \rangle_+ \cup \nu_+$$

and hence by co-induction that $\nu_1 \subseteq \nu_2$.

(i) If $a\Uparrow$ and $b\Uparrow$ then $(a, b) \in \nu_+$ by Proposition 4.6.

(ii) We prove by induction on n that

if $(a, b) \in \mathcal{S}_1$ and $a\Downarrow$ in n steps, $(a, b) \in F(\nu_1) \cup \langle F(\nu_1) \rangle_+$.

Suppose then that $(a, b) \in \mathcal{S}_1$ and $a\Downarrow$ in n steps. From $\nu_1 = F(\nu_1) \cup \langle \nu_1 \rangle_+$ there are two cases to consider. If $(a, b) \in F(\nu_1)$ there is nothing more to say. Suppose, on the other hand, that $(a, b) \in \langle \nu_1 \rangle_+$, which is to say there are a' and b' with $a \mapsto^+ a'$, $b \mapsto^+ b'$ and $(a', b') \in \nu_1$. Since a converges, $(a', b') \in \mathcal{S}_1$ as a' must converge, and it does so in fewer steps than n. Therefore by induction hypothesis we have $(a', b') \in F(\nu_1) \cup \langle F(\nu_1) \rangle_+$. But then $(a, b) \in \langle F(\nu_1) \cup \langle F(\nu_1) \rangle_+ \rangle_+ = \langle F(\nu_1) \rangle_+ \cup \langle \langle F(\nu_1) \rangle_+ \rangle_+ \subseteq \langle F(\nu_1) \rangle_+$, as required.

Finally case (iii) follows by a symmetric argument. Via the two inclusions, then, we have $\nu_1 = \nu_2$. ■

Lemma 4.9 $\sim\; = \nu\mathcal{S}.\, \langle \mathcal{S} \rangle \cup \langle \mathcal{S} \rangle_V \cup \langle \mathcal{S} \rangle_+$.

Proof Let $F(\mathcal{S})$ be $\langle \mathcal{S} \rangle \cup \langle \mathcal{S} \rangle_V$ and ν be $\nu\mathcal{S}.\, F(\mathcal{S}) \cup \langle \mathcal{S} \rangle_+$, that is, $\nu\mathcal{S}.\, \langle \mathcal{S} \rangle \cup \langle \mathcal{S} \rangle_V \cup \langle \mathcal{S} \rangle_+$. The easy inclusion is $\sim\; \subseteq \nu$, which follows from $\sim\; = \langle \sim \rangle$. For the reverse inclusion, we shall prove

(1) $\sim\; = \nu\mathcal{S}.\, F(\mathcal{S})$

(2) $\nu_\sim \subseteq F(\nu_\sim) \cup \sim$

From (2) $\nu_\sim \subseteq \nu\mathcal{S}.\, F(\mathcal{S})$ follows by strong co-induction from (1). Hence $\nu \subseteq\; \sim$ follows.

(1) Let $\nu_1 = \nu\mathcal{S}.\, F(\mathcal{S})$. Inclusion $\sim\; \subseteq \nu_1$ follows from $\sim\; = \langle \sim \rangle$. It remains to show $\nu_1 \subseteq\; \sim$. Whenever $\mathcal{S} \subseteq \langle \mathcal{S} \rangle \cup \langle \mathcal{S} \rangle_V$ we can show that $\mathcal{S}_\sim \subseteq\; \sim$ by a simple modification of the argument for Lemma 4.3. Hence $\sim\nu_1\sim\; \subseteq\; \sim$ and indeed $\nu_1 \subseteq\; \sim$.

(2) Finally, suppose that $(a, b) \in \nu_\sim$, that is, $a \sim a_0 \,\nu\, b_0 \sim b$. We must show either that $a \sim b$ or $(a, b) \in F(\nu_\sim)$. Since $\nu = \nu\mathcal{S}.\, F(\mathcal{S}) \cup \langle F(\mathcal{S}) \rangle_+ \cup \nu_+$, by Lemma 4.8, either $(a_0, b_0) \in \nu_+$ or there is a pair $(a_1, b_1) \in F(\nu)$ with $a_0 \mapsto^* a_1$ and $b_0 \mapsto^* b_1$. In the first case we have $a_0 \sim b_0$, since $\nu_+ \subset\; \sim$, and so $a \sim b$ too. In the second case, pair (a_1, b_1) is either in (i) $\langle \nu \rangle_V$ or (ii) $\langle \nu \rangle$. If (i), we have $(a, b) \in \langle \nu \rangle_V \subseteq \langle \nu_\sim \rangle_V \subseteq F(\nu_\sim)$. Otherwise, suppose (ii), and that $a \xrightarrow{\alpha} a'$. Then we can fill in the following diagram

$$\begin{tikzcd}
a \arrow[d, "\alpha"'] & \sim & a_1 \arrow[d, "\alpha"'] & \langle \nu \rangle & b_1 \arrow[d, "\alpha"] & \sim & a \arrow[d, "\alpha"] \\
a' & \sim & a'_1 & \nu & b'_1 & \sim & a'
\end{tikzcd}$$

and a symmetric one to show that $(a, b) \in \langle \nu_{\sim} \rangle \subseteq F(\nu_{\sim})$. In any case, then, we have $(a, b) \in F(\nu_{\sim}) \cup \sim$ as required. ■

Lemma 4.10 $\sim \subseteq \langle \sim \rangle_V \cup \nu_+$

Proof Suppose $a \sim b{:}A$. Either (A) $a\Uparrow$ and A is active or (B) there is a value v with $a \sim v$. In case (A), $b\Uparrow$ too, so $(a, b) \in \nu_+$. In case (B), $b \sim v$ too, we have $a \sim v \overline{\sim} v \sim b$, so $(a, b) \in \langle \sim \rangle_V$. In any case $(a, b) \in \langle \sim \rangle_V \cup \nu_+$. ■

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

Theorem 4.11 $\sim = \nu \mathcal{S}. \langle \mathcal{S} \rangle_V \cup \langle \mathcal{S} \rangle_+$.

Proof Let ν be the right-hand side, $\nu \mathcal{S}. \langle \mathcal{S} \rangle_V \cup \langle \mathcal{S} \rangle_+$. From the following inclusion,

$$\begin{array}{rcll} \sim & \subseteq & \langle \sim \rangle_V \cup \nu_+ & \text{Lemma 4.10} \\ & \subseteq & \langle \sim \rangle_V \cup \nu & \text{since } \nu_+ \subseteq \nu \end{array}$$

it follows by strong co-induction $\sim \subseteq \nu$. Clearly $\nu \subseteq \nu \mathcal{S}. \langle \mathcal{S} \rangle \cup \langle \mathcal{S} \rangle_V \cup \langle \mathcal{S} \rangle_+$, and the latter by Lemma 4.9 equals $\sim$. So $\sim = \nu$. ■

The significance of this equation is that it is a complete co-inductive characterisation of bisimilarity (and hence contextual equivalence) without mentioning labelled transitions.

4.5 Proof of Monad Laws

Our final example is a proof of the monad laws for streams (Wadler 1992). Define the following combinators.

$$\begin{array}{rcl} (\texttt{++}) & \stackrel{\text{def}}{=} & \texttt{rec}(f, xs)\lambda(ys)\texttt{scase}(xs, ys, \lambda(x)\lambda(xs')x :: f\, xs'\, ys) \\ \texttt{join} & \stackrel{\text{def}}{=} & \texttt{rec}(f, xss)\texttt{scase}(xss, \texttt{nil}, \lambda(xs)\lambda(xss')xs \texttt{ ++ } f xss') \\ \texttt{id} & \stackrel{\text{def}}{=} & \lambda(x)x \\ \texttt{unit} & \stackrel{\text{def}}{=} & \lambda(x)x :: \texttt{nil} \end{array}$$

Strictly speaking these are families of monomorphic combinators indexed by types, but we will skate over the details of typing.

Proposition 4.12

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

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

(3) $\texttt{map}\,f \circ \texttt{unit} \sim \texttt{unit} \circ f$

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

(5) $\texttt{join} \circ \texttt{unit} \sim \texttt{id}$

(6) $\texttt{join} \circ \texttt{map}\,\texttt{unit} \sim \texttt{id}$

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

Proof

(3,5) Easy equational reasoning.

(1, 2, 6) For these three parts it suffices by extensionality to show that each of the following relations is contained in $\sim$.

$$\begin{array}{lcllr}
\mathcal{S}_1 & = & \{(\texttt{map}\,\texttt{id}\,as, & \texttt{id}\,as) & \mid as{:}\,\texttt{[}A\texttt{]}\} \\
\mathcal{S}_2 & = & \{(\texttt{map}\,(f \circ g)\,as, & (\texttt{map}\,f \circ \texttt{map}\,g)\,as) & \mid as{:}\,\texttt{[}A\texttt{]}\} \\
\mathcal{S}_6 & = & \{((\texttt{join} \circ \texttt{map}\,\texttt{unit})\,as, & \texttt{id}\,as) & \mid as{:}\,\texttt{[}A\texttt{]}\}
\end{array}$$

For each $\mathcal{S}_i$ we can easily check that $\mathcal{S}_i \subseteq \langle \mathcal{S}_i \cup \sim \rangle_V \cup \sim$, and hence by strong co-induction that $\mathcal{S}_i \subseteq \sim$.

(4) Again by extensionality it suffices to show that $\mathcal{S}_4 \subseteq \sim$.

$$\mathcal{S}_4 \;=\; \{(\texttt{map}\,f\,(\texttt{join}\,ass), \texttt{join}\,(\texttt{map}\,(\texttt{map}\,f)\,ass)) \mid ass{:}\,\texttt{[[}A\texttt{]]}\}$$

Suppose that $a\,\mathcal{S}_4\,b$. Take cases according to the operational behaviour of ass: (A) $ass\Uparrow$, (B) $ass \Downarrow \texttt{nil}$ or (C) $ass \Downarrow as :: ass'$. In case (A) we have $a \sim \Omega \sim b$ and in case (B), $a \sim \texttt{nil} \sim b$. In case (C) we have the following reductions.

$$\begin{array}{rcl}
a & \mapsto^{+} & \texttt{map}\,f\,(as \mathbin{\texttt{++}} \texttt{join}\,ass') \\
\texttt{map}\,(\texttt{map}\,f)\,ass & \mapsto^{+} & \texttt{map}\,f\,as :: \texttt{map}\,(\texttt{map}\,f)\,ass' \\
b & \mapsto^{+} & (\texttt{map}\,f\,as) \mathbin{\texttt{++}} \texttt{join}\,(\texttt{map}\,(\texttt{map}\,f)\,ass')
\end{array}$$

We examine the three possible behaviours of as: (CA) $as\Uparrow$, (CB) $as \Downarrow$ $\mathtt{nil}$ or (CC) $as \Downarrow a_0 :: as'$. In case (CA) we have $a \sim \Omega \sim b$. In case (CB) let

$$\begin{aligned} a' &\equiv \mathtt{map}\, f\, (\mathtt{join}\, ass') \\ b' &\equiv \mathtt{join}\, (\mathtt{map}\, (\mathtt{map}\, f)\, ass') \end{aligned}$$

and then we have $a \mapsto^+ a'$ and $b \mapsto^+ b'$ and $a'\, \mathcal{S}\, b'$, that is, $a \langle \mathcal{S} \rangle_+ b$. Finally, in case (CC) let

$$\begin{aligned} a' &\equiv \mathtt{map}\, f\, (\mathtt{join}\, (as' :: ass')) \\ b' &\equiv \mathtt{join}\, (\mathtt{map}\, (\mathtt{map}\, f)\, (as' :: ass')) \end{aligned}$$

and we have the following bisimilarities.

$$\begin{aligned} a &\sim \mathtt{map}\, f\, (a_0 :: (as' \mathbin{\mathtt{++}} \mathtt{join}\, ass')) \\ &\sim (f\, a_0) :: \mathtt{map}\, f\, (as' \mathbin{\mathtt{++}} \mathtt{join}\, ass') \\ &\sim (f\, a_0) :: a' \\ b &\sim (f\, a_0) :: ((\mathtt{map}\, f\, as') \mathbin{\mathtt{++}} \mathtt{join}\, (\mathtt{map}\, (\mathtt{map}\, f)\, ass')) \\ &\sim (f\, a_0) :: \mathtt{join}\, (\mathtt{map}\, f\, as' :: \mathtt{map}\, (\mathtt{map}\, f)\, ass') \\ &\sim (f\, a_0) :: b' \end{aligned}$$

Since $a'\, \mathcal{S}_4\, b'$ we have $(a, b) \in \langle \mathcal{S}_4 \cup \sim \rangle_V$. In all cases, then, we have established that $(a, b) \in \langle \mathcal{S}_4 \cup \sim \rangle_V \cup \langle \mathcal{S}_4 \rangle_+$. Hence $\mathcal{S}_4 \subseteq \sim$ by strong co-induction.

(7) The hardest. The most obvious co-inductive proof fails. Instead we can prove that $\mathcal{S}_7$ below is a subset of $\sim$. First we extract the following expressions from the definitions of `++` and `join` respectively.

$$\begin{aligned} \overline{\mathtt{append}}[as, bs] &\stackrel{\text{def}}{=} \mathtt{scase}(as, bs, \lambda(x)\lambda(xs)x :: (xs \mathbin{\mathtt{++}} bs)) \\ \overline{\mathtt{join}}[ass] &\stackrel{\text{def}}{=} \mathtt{scase}(ass, \mathtt{nil}, \lambda(xs)\lambda(xss)xs \mathbin{\mathtt{++}} \mathtt{join}(xss')) \end{aligned}$$

Note that $as \mathbin{\mathtt{++}} bs \mapsto^+ \overline{\mathtt{append}}[as, bs]$ and $\mathtt{join}(ass) \mapsto^+ \overline{\mathtt{join}}[ass]$. Now let

$$\begin{aligned} e_L[ass, asss] &\stackrel{\text{def}}{=} \overline{\mathtt{append}}[\overline{\mathtt{join}}[ass], \mathtt{join}\, (\mathtt{map}\, \mathtt{join}\, asss)] \\ e_R[ass, asss] &\stackrel{\text{def}}{=} \overline{\mathtt{join}}[ass \mathbin{\mathtt{++}} \mathtt{join}\, asss] \end{aligned}$$

and define $\mathcal{S}_7$ to be

$$\{(e_L[ass, asss], e_R[ass, asss]) \mid ass{:}\mathtt{[[}A\mathtt{]]}\ \&\ asss{:}\mathtt{[[[}A\mathtt{]]]}\}.$$

Suppose that $e_L[ass, asss] \, \mathcal{S}_7 \, e_R[ass, asss]$. We shall establish the following property

$$(e_L[ass, asss], e_R[ass, asss]) \in \langle \mathcal{S}_7 \cup \sim \rangle_V \cup \langle \mathcal{S}_7 \rangle_+ \cup \sim \qquad (*).$$

and then $\mathcal{S}_7 \subseteq \sim$ follows from $(*)$ by strong co-induction. Part (7) follows easily (take $ass \equiv \mathtt{nil}$). Consider then the three possible evaluation behaviours of ass: (A) $ass\Uparrow$, (B) $ass \Downarrow \mathtt{nil}$ or (C) $ass \Downarrow as{::}ass'$. In case (A) we have that $e_L[ass, asss] \sim \Omega \sim e_R[ass, asss]$. In case (B) we have the following reductions.

$$\begin{array}{lll} e_L[ass, asss] & \mapsto^+ & \mathtt{join}\,(\mathtt{map}\,\mathtt{join}\,asss) \\ e_R[ass, asss] & \mapsto^+ & \overline{\mathtt{join}}[\mathtt{join}\,asss] \end{array}$$

Now consider cases of the behaviour of $asss$: (BA) $asss\Uparrow$, (BB) $asss \Downarrow \mathtt{nil}$ and (BC) $asss \Downarrow ass'{::}asss'$. In case (BA) we have $e_L[ass, asss] \sim \Omega \sim e_R[ass, asss]$ and in case (BB) $e_L[ass, asss] \sim \mathtt{nil} \sim e_R[ass, asss]$. In case (BC),

$$\begin{array}{lll} e_L[ass, asss] & \mapsto^+ & \overline{\mathtt{join}}[\mathtt{join}\,ass' :: \mathtt{map}\,\mathtt{join}\,asss'] \\ & \mapsto^+ & \mathtt{join}\,ass' \,\mathtt{++}\, \mathtt{join}\,(\mathtt{map}\,\mathtt{join}\,asss') \\ & \mapsto^+ & e_L[ass', asss'] \\ e_R[ass, asss] & \mapsto^+ & \overline{\mathtt{join}}[\overline{\mathtt{join}}[ass'] :: asss')] \\ & \mapsto^+ & \overline{\mathtt{join}}[ass' \,\mathtt{++}\, \mathtt{join}\,asss'] \\ & \equiv & e_R[ass', asss'] \end{array}$$

so $(e_L[ass, asss], e_R[ass, asss]) \in \langle \mathcal{S}_7 \rangle_+$.

Returning to the original case analysis, in the final case, (C), we have the following reductions.

$$\begin{array}{lll} e_L[ass, asss] & \mapsto^+ & \overline{\mathtt{append}}[as \,\mathtt{++}\, \mathtt{join}\,ass', \mathtt{join}\,(\mathtt{map}\,\mathtt{join}\,asss)] \\ e_R[ass, asss] & \mapsto^+ & \overline{\mathtt{join}}[as :: (ass' \,\mathtt{++}\, \mathtt{join}\,asss)] \\ & \mapsto^+ & as \,\mathtt{++}\, \mathtt{join}\,(ass' \,\mathtt{++}\, \mathtt{join}\,asss) \end{array}$$

There are three behaviours of as to consider: (CA) $as\Uparrow$, (CB) $as \Downarrow \mathtt{nil}$ and (CC) $as \Downarrow a :: as'$. In case (CA) both sides diverge. In case (CB),

$$\begin{array}{lll} e_L[ass, asss] & \mapsto^+ & \overline{\mathtt{append}}[\overline{\mathtt{join}}[ass'], \mathtt{join}\,(\mathtt{map}\,\mathtt{join}\,asss)] \\ e_R[ass, asss] & \mapsto^+ & \overline{\mathtt{join}}\,[ass' \,\mathtt{++}\, \mathtt{join}\,asss] \end{array}$$

so $(e_L[ass, asss], e_R[ass, asss]) \in \langle \mathcal{S}_7 \rangle_+$. Finally, in case (CC), set a' and b' to be

$$\begin{array}{rcl} a' & \equiv & \texttt{join}\,(as' :: ass') \;\texttt{++}\; \texttt{join}\,(\texttt{map}\,\texttt{join}\, asss) \\ b' & \equiv & \texttt{join}((as :: ass') \;\texttt{++}\; \texttt{join}\, asss) \end{array}$$

and we can easily check that $e_L[ass, asss] \sim a_0 :: a'$, $e_R[ass, asss] \sim a_0 :: b'$ and $a' \sim\mathcal{S}_7\sim b'$, and hence that $(e_L[ass, asss], e_R[ass, asss]) \in \langle \mathcal{S}_7 \cup \sim \rangle_V$. In all cases we have established property $(*)$ and hence (7) follows. ■

4.6 Bird and Wadler's Take Lemma

We derive the Take Lemma of Bird and Wadler (1988) to illustrate how a proof principle usually derived by domain-theoretic fixpoint induction follows also from co-induction. We begin with the `take` function, which returns a finite approximation to an infinite list.

```
take 0 xs = nil
take n nil = nil
take (n+1) (x :: xs) = x :: (take n xs)
```

Here is the key lemma.

Lemma 4.13 *Define $\mathcal{S} \subseteq Rel$ by $a\mathcal{S}b$ iff $\forall n \in \mathbb{N}(\texttt{take}\,\underline{n+1}\,a \sim \texttt{take}\,\underline{n+1}\,b)$.*

(1) *Whenever $a\,\mathcal{S}\,b$ and $a \Downarrow \texttt{nil}$, $b \Downarrow \texttt{nil}$ too.*

(2) *Whenever $a\,\mathcal{S}\,b$ and $a \Downarrow a' :: a''$ there are b' and b'' with $b \Downarrow b' :: b''$, $a' \sim b'$ and $a''\,\mathcal{S}\,b''$.*

(3) $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle$.

Proof

(1) Using $a\,\mathcal{S}\,b$ and $n = 0$ we have $\texttt{take}\,\underline{1}\,a \sim \texttt{take}\,\underline{1}\,b$. Since $a \Downarrow \texttt{nil}$, we have $a \sim \texttt{nil}$, and in fact that $\texttt{nil} \sim \texttt{take}\,\underline{1}\,b$ by definition of `take`. We know that either $b \sim \Omega$, $b \Downarrow \texttt{nil}$ or $b \Downarrow b' :: b''$. The first and third possibilities would contradict $\texttt{nil} \sim \texttt{take}\,\underline{1}\,b$, so it must be that $b \Downarrow \texttt{nil}$.

(2) We have

$$\texttt{take}\,\underline{n+1}\,(a' :: a'') \sim \texttt{take}\,\underline{n+1}\,b.$$

With $n = 0$ we have

$$a' :: \texttt{nil} \sim \texttt{take}\,\underline{1}\,b$$

which rules out the possibilities that $b \sim \Omega$ or $b \Downarrow \texttt{nil}$, so it must be that $b \Downarrow b' :: b''$. So we have

$$a' :: (\texttt{take}\,\underline{n}\,a'') \sim b' :: (\texttt{take}\,\underline{n}\,b'')$$

for any n, and hence $a' \sim b'$ and $a'' \mathcal{S}\, b''$ by (Inj Stm).

(3) As before it suffices to prove that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle$. Suppose that $a \,\mathcal{S}\, b$. For each transition $a \xrightarrow{\alpha} a'$ we must exhibit b' satisfying $b \xrightarrow{\alpha} b'$ and either $a' \,\mathcal{S}\, b'$ or $a' \sim b'$. Since a and b are streams, there are three possible actions α to consider.

  (a) Action α is $\texttt{nil}$. Hence $a \Downarrow \texttt{nil}$ and a' is $\mathbf{0}$. By part (1), $b \Downarrow \texttt{nil}$ too. Hence $b \xrightarrow{\texttt{nil}} \mathbf{0}$, and $\mathbf{0} \sim \mathbf{0}$ as required.

  (b) Action α is $\texttt{hd}$. Hence $a \Downarrow a' :: a''$. By part (2), there are b' and b'' with $b \Downarrow b' :: b''$, hence $b \xrightarrow{\texttt{hd}} b'$, and in fact $a' \sim b'$ by part (2).

  (c) Action α is $\texttt{tl}$. Hence $a \Downarrow a' :: a''$. By part (2), there are b' and b'' with $b \Downarrow b' :: b''$, hence $b \xrightarrow{\texttt{tl}} b''$, and in fact $a'' \,\mathcal{S}\, b''$ by part (2). ■

The Take Lemma follows from (3) by strong co-induction.

Theorem 4.14 (Take Lemma) *Suppose $a, b\colon$*`[`*A*`]`*.*
Then $a \sim b$ iff $\forall n \in \mathbb{N}\,(\texttt{take}\,\underline{n+1}\,a \sim \texttt{take}\,\underline{n+1}\,b)$.

See Bird and Wadler (1988) and Sander (1992), for instance, for examples of how the Take Lemma reduces a proof of equality of infinite streams to an induction over all their finite approximations. We can use the Take Lemma to prove the `map`/`filter` example of Section 4.3 but as in Lemma 4.5 we need an induction on the length of evaluation. Co-induction based on the equation $\sim\, = \nu\mathcal{S}.\, \langle \mathcal{S} \rangle_V \cup \langle \mathcal{S} \rangle_+$ is more useful than the Take Lemma, in the sense that it avoids the induction on evaluation.

5 Variations

5.1 The Active/Passive Distinction

We distinguished between active and passive types when creating our labelled transition system for PCF plus streams. Programs of active types had to converge to a value before they would admit a transition. Programs of passive type had transitions whether or not they converged. We showed in Section 2.6 that $\mathit{Total}(A) \Leftrightarrow A$ passive. But this need not be so in every labelled transition system that generates a co-inductive characterisation of contextual equivalence. Recall experimental equivalence from Section 3.4. Consider the following labelled transition system for PCF plus streams,

$$\frac{a{:}A \qquad -:A \vdash \mathcal{E} : B}{a \xrightarrow{\mathcal{E}} \mathcal{E}[b]}\;(\text{Trans Exper}) \qquad \frac{a{:}\texttt{Bool} \qquad a\Downarrow}{a \xrightarrow{\texttt{val}} \mathbf{0}}\;(\text{Trans Val})$$

where an action is either an experiment, $\mathcal{E}$, or `val`, and $\mathbf{0}$ is disjoint from the set of programs. Clearly two programs are bisimilar according to this labelled transition system iff they are experimentally equivalent. In effect every type apart from `Bool` is passive. So there are types which do not satisfy $\mathit{Total}(-)$ that are passive.

The active/passive distinction is best seen as a device to optimise a labelled transition system for co-inductive bisimulation proofs. The generic system above would work for any operationally extensional calculus. We can view the transition rule for function types (the only passive types in PCF plus streams) as an instance of (Trans Exper). This rule is contingent only on the type of the observed program, not on whether it terminates or on the shape of its value. In contrast our rules for active types do depend on such information, and as we discussed in the case of streams, lead to simpler proofs. At least one rule must depend on termination behaviour or the shape of values. Without (Trans Val) the system above would be useless. On the other hand, if there are convergence testing contexts at all types—as in a call-by-value language—we can dispense with passive types and (Trans Exper) completely. But suppose a type A satisfies $\mathit{Total}(A)$. There is a value v contextually equivalent to Ω^A. If our transition system distinguishes between convergent and divergent programs—by making A active, for instance—bisimilarity would fall short of contextual equivalence. Although $v \simeq \Omega^A$ we would have $v \not\sim \Omega^A$.

5.2 Convergence Testing and Call-by-Value

Our presentation of PCF plus streams was lazy in the sense that functions were call-by-name and cons, ::, did not evaluate its arguments. There are two important variations.

- Suppose we add a convergence testing operation `seq` such that $\texttt{seq}(a, b)$ diverges if a diverges, but if a converges, $\texttt{seq}(a, b) \mapsto^+ b$. Programs a and b may be of any type and $\texttt{seq}(a, b)$ has the same type as b. Every type A, now satisfies $\Omega^A_\sim \subseteq \Uparrow^A$. In particular, $\Omega^{A\to B} \not\simeq \lambda(x{:}A)\Omega^B$. If we make function types active—so that every type is active—and replace (Trans Fun) from Table 5 by the following rule

$$\frac{v{:}B \to A \qquad b{:}B}{v \xrightarrow{@b} v(b)}$$

  we can rework Section 3 to show that bisimilarity equals contextual equivalence. This is sometimes known as a 'lazy' variation (Riecke 1993), though implementations of call-by-name using lazy evaluation do not depend on convergence testing. The presence or absence of `seq` is controversial amongst lazy functional programmers: it is desirable on grounds of efficiency but its presence puts awkward side-conditions on (Fun Eta) and (Fun Ext).

- Suppose we make evaluation eager in the sense that functions are call-by-value and cons evaluates its arguments.

$$v ::= \lambda(x{:}A)e \mid \texttt{rec}(f, x)e \mid \texttt{nil} \mid (v :: v)$$
$$(\lambda(x{:}A)e)(v) \mapsto e[{}^{v}\!/\!{}_{x}]$$
$$(\texttt{rec}(f, x)e)(v) \mapsto e[{}^{\texttt{rec}(f,x)e,\, v}\!/\!{}_{f,\, x}]$$
$$\texttt{scase}(\texttt{nil}, b, c) \mapsto b$$
$$\texttt{scase}(u :: v, b, c) \mapsto c(u)(v)$$
$$\mathcal{E} ::= -(a) \mid v(-) \mid \texttt{scase}(-, a, a) \mid (- :: a) \mid (v :: -)$$

  Arithmetic is unchanged. If we make all types active, then the following rules

$$\frac{v{:}B \to A \qquad u{:}B}{v \xrightarrow{@u} v(u)}$$

$$\texttt{nil} \xrightarrow{\texttt{nil}} \mathbf{0} \qquad u :: v \xrightarrow{\texttt{hd}} u \qquad u :: v \xrightarrow{\texttt{tl}} v$$

  together with (Trans Red) and the rules for literals from Table 5 lead to a definition of bisimilarity that equals contextual equivalence. Variables only stand for values. We must modify the definition of Γ-closure to only substitute values for variables.

Statics

$$\frac{\Gamma \vdash e_1 : A}{\Gamma \vdash \mathtt{inl}[A+B](e_1) : A+B}\,(\text{Exp Inl}) \qquad \frac{\Gamma \vdash e_2 : B}{\Gamma \vdash \mathtt{inr}[A+B](e_2) : A+B}\,(\text{Exp Inr})$$

$$\frac{\Gamma \vdash e_0 : A+B \qquad \Gamma, x_1{:}A \vdash e_1 : C \qquad \Gamma, x_2{:}B \vdash e_2 : C}{\Gamma \vdash \mathtt{case}\, e_0\, \mathtt{of}\ \mathtt{inl}(x_1) \Rightarrow e_1\ \mathtt{or}\ \mathtt{inr}(x_2) \Rightarrow e_2 : C}\,(\text{Exp Case})$$

Lazy Dynamics

$$\begin{array}{rr}
v ::= \mathtt{inl}(a) \mid \mathtt{inr}(a) & (\text{Value Sum}) \\
(\mathtt{case}\,\mathtt{inl}(a)\ \mathtt{of}\ \mathtt{inl}(x_1) \Rightarrow e_1\ \mathtt{or}\ \mathtt{inr}(x_2) \Rightarrow e_2) \mapsto e_1[{}^{a}\!/\!{}_{x_1}] & (\text{Red Inl}) \\
(\mathtt{case}\,\mathtt{inr}(a)\ \mathtt{of}\ \mathtt{inl}(x_1) \Rightarrow e_1\ \mathtt{or}\ \mathtt{inr}(x_2) \Rightarrow e_2) \mapsto e_2[{}^{a}\!/\!{}_{x_2}] & (\text{Red Inr}) \\
\mathcal{E} ::= \mathtt{case}\ -\ \mathtt{of}\,\mathtt{inl}(x_1) \Rightarrow e_1\ \mathtt{or}\ \mathtt{inr}(x_2) \Rightarrow e_2 & (\text{Exper Sum})
\end{array}$$

Active Observations

$$\begin{array}{rl}
\alpha ::= \mathtt{inl} \mid \mathtt{inr} & (\text{Act Sum}) \\
\mathtt{inl}(a) \xrightarrow{\mathtt{inl}} a & (\text{Trans Inl}) \\
\mathtt{inr}(a) \xrightarrow{\mathtt{inr}} a & (\text{Trans Inr})
\end{array}$$

Table 8: Δ_+: sums

5.3 FPC: Sums, Pairs, Recursive Types

We have presented a theory of bisimilarity in detail for PCF plus streams. All the results extend routinely to the usual first-order extensions of PCF. Tables 8, 9 and 10 show the rules for sums, Δ_+, pairs, $\Delta_\times$, and recursive types Δ_μ. The obvious rules of compatible refinement, $\widehat{-}$, are omitted. We let X range over a countable set of type variables. The new types are given by

$$A ::= X \mid \mathtt{rec}(X)A \mid A + A \mid A \times A.$$

Recursive types and pairs are passive; sums are active. The language

$$\Delta_\to \cup \Delta_+ \cup \Delta_\times \cup \Delta_\mu$$

corresponds to the lazy forms of Gunter's (1992) FPC and Winskel's (1993) language of recursive types. As discussed in these textbooks, arithmetic, Δ_A,

Statics

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B} \text{(Exp Pair)}$$

$$\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathtt{fst}(e) : A} \text{(Exp Fst)} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathtt{snd}(e) : B} \text{(Exp Snd)}$$

Lazy Dynamics

$$\begin{array}{rl} v ::= (a, b) & \text{(Value Pair)} \\ \mathtt{fst}((a, b)) \mapsto a & \text{(Red Fst)} \\ \mathtt{snd}((a, b)) \mapsto b & \text{(Red Snd)} \\ \mathcal{E} ::= \mathtt{fst}(-) \mid \mathtt{snd}(-) & \text{(Exper Pair)} \end{array}$$

Passive Observations

$$\begin{array}{rl} \alpha ::= \mathtt{fst} \mid \mathtt{snd} & \text{(Act Pair)} \\ a \xrightarrow{\mathtt{fst}} \mathtt{fst}(a) \text{ if } a{:}A \times B & \text{(Trans Fst)} \\ a \xrightarrow{\mathtt{snd}} \mathtt{snd}(a) \text{ if } a{:}A \times B & \text{(Trans Snd)} \end{array}$$

Table 9: $\Delta_\times$: pairs

Statics

$$\frac{\Gamma \vdash e : A[\mu/X] \qquad \mu \equiv \mathtt{rec}(X)A}{\Gamma \vdash \mathtt{intro}[\mu](e) : \mu}\,(\text{Exp Intro})$$

$$\frac{\Gamma \vdash e : \mu \qquad \mu \equiv \mathtt{rec}(X)A}{\Gamma \vdash \mathtt{elim}[\mu](e) : A[\mu/X]}\,(\text{Exp Elim})$$

Lazy Dynamics

$$\begin{array}{rcl} v ::= \mathtt{intro}(a) & & (\text{Value Rec}) \\ \mathtt{elim}(\mathtt{intro}(a)) \mapsto a & & (\text{Red Elim}) \\ \mathcal{E} ::= \mathtt{elim}(-) & & (\text{Exper Rec}) \end{array}$$

Passive Observations

$$\begin{array}{cc} \alpha ::= \mathtt{elim} & (\text{Act Rec}) \\ a \xrightarrow{\mathtt{elim}} \mathtt{elim}[\mathtt{rec}(X)A](a) \text{ if } a{:}\mathtt{rec}(X)A & (\text{Trans Elim}) \end{array}$$

Table 10: Δ_μ: recursive types

recursive functions, $\Delta_{\mu\to}$, and streams, $\Delta_{[-]}$, can all be encoded. As with PCF plus streams we have $\mathit{Total}(A) \Leftrightarrow A$ passive. We can prove the following eta laws by co-inductive arguments similar to those in Proposition 2.21.

$$\begin{array}{rcll} (\mathtt{fst}(a), \mathtt{snd}(a)) & \sim & a & \text{if } a{:}A \to B \\ (\mathtt{intro}(\mathtt{elim}(a))) & \sim & a & \text{if } a{:}\mathtt{rec}(X)A \end{array}$$

Call-by-name plus convergence testing and call-by-value variants exist for FPC also. We omit the details. There is one subtlety. As Gunter (1992, p238) shows, in call-by-value FPC there are no values of the recursive type $\mathtt{rec}(X)X$. (In our form of call-by-name FPC, there are, because $\mathtt{intro}(-)$ is not an experiment.) In fact there are many such types with no values. Hence to avoid equating convergent and divergent functions at any type $A \to B$ where there are no values of type A, we introduce the following variant of (Trans Val) at (all) function types.

$$\frac{v{:}A \to B}{v \xrightarrow{\mathtt{val}} \mathbf{0}}$$

5.4 Matching Value Contexts

Consider the following encoding of infinite streams in (lazy) FPC.

$$\begin{array}{rcl} \mathtt{Stm}(X) & \stackrel{\text{def}}{=} & \mathtt{rec}(Y)(X \times Y) \\ \mathtt{cons}(a, b) & \stackrel{\text{def}}{=} & \mathtt{intro}[\mathtt{Stm}(X)]((a, b)) \end{array}$$

Co-inductive proofs similar to those developed in Section 4 work well with this encoding. But we need to generalise the idea of matching values (via $\langle - \rangle_V$) to allow matching of value contexts.

$$\begin{array}{rcl} H(\mathcal{S}) & \stackrel{\text{def}}{=} & \mu\mathcal{R}.\, \mathcal{S} \cup \overline{\mathcal{R}} \\ \langle \mathcal{S} \rangle_{V*} & \stackrel{\text{def}}{=} & \langle H(\mathcal{S}) \rangle_V \end{array}$$

These are all monotone functions. Let ν_{V*} be $\nu\mathcal{S}.\, \langle \mathcal{S} \rangle_{V*}$. Roughly speaking, unwinding the inner inductive definition permits arbitrary nesting of value constructors. To justify this generalisation, suppose that $\mathcal{S}$ is $\{(a, a'), (b, b')\}$. Then $(\mathtt{cons}(a, b), \mathtt{cons}(a', b')) \notin \langle \mathcal{S} \rangle_V$ because $\mathtt{cons}$ is encoded by two value constructors, but we do have $(\mathtt{cons}(a, b), \mathtt{cons}(a', b')) \in \langle \mathcal{S} \rangle_{V*}$. We need the next few lemmas to show that $\nu_V = \nu_{V*}$.

Lemma 5.1 *For any $\mathcal{S}$, $\mathcal{S} \subseteq H(\mathcal{S})$.*

Proof $H(\mathcal{S}) = \mu\mathcal{R}.\, \mathcal{S} \cup \overline{\mathcal{R}} = \mathcal{S} \cup \overline{H(\mathcal{S})}$. ■

Lemma 5.2 $H(\nu_{V*}) \subseteq \nu_{V*}$

Proof Since $H(\nu_{V*})$ is the least set closed under $\mathcal{R} \mapsto \nu_{V*} \cup \overline{\mathcal{R}}$, $H(\nu_{V*}) \subseteq \nu_{V*}$ will follow by induction if $\nu_{V*} \cup \overline{\nu_{V*}} \subseteq \nu_{V*}$. It is enough to show that $\overline{\nu_{V*}} \subseteq \nu_{V*}$.

$$\begin{array}{rcll}
\overline{\nu_{V*}} & \subseteq & \sim\overline{\nu_{V*}}\sim & \text{because } Id \text{ reflexive} \\
& = & \langle \nu_{V*} \rangle_V & \\
& \subseteq & \langle H(\nu_{V*}) \rangle_V & \text{because } \nu_{V*} \subseteq H(\nu_{V*}) \text{ and } \langle - \rangle_V \text{ monotone} \\
& = & \nu_{V*} &
\end{array}$$

Hence $H(\nu_{V*}) \subseteq \nu_{V*}$ by induction. ■

Proposition 5.3 $\nu_V = \nu_{V*}$.

Proof We prove inclusions in both directions by co-induction. First, $\nu_V \subseteq \nu_{V*}$ follows from $\nu_V \subseteq \langle H(\nu_V) \rangle_V$, which holds because $\nu_V \subseteq \langle \nu_V \rangle_V \subseteq \langle H(\nu_V) \rangle_V$, using Lemma 5.1. For the reverse inclusion, we have

$$\begin{array}{rcll}
\nu_{V*} & = & \sim\overline{H(\nu_{V*})}\sim & \\
& \subseteq & \sim\overline{\nu_{V*}}\sim & \text{by Lemma 5.2} \\
& = & \langle \nu_{V*} \rangle_V &
\end{array}$$

and hence $\nu_{V*} \subseteq \nu_V$ by co-induction. ■

We can generalise Theorem 4.11 as follows. The proof is a routine modification of the original proof.

Theorem 5.4 $\sim\, = \nu\mathcal{S}.\, \langle \mathcal{S} \rangle_{V*} \cup \langle \mathcal{S} \rangle_{+}$.

This theorem is useful for co-inductive proofs about `Stm` types and other encodings in FPC. See Gordon (1995a) for an application.

6 Summary and Related Work

We have developed a 'CCS-view of lambda-calculus.' Using a novel labelled transition system we replayed the definition of bisimilarity from CCS and proved that it equals contextual equivalence. Hence we answered Turner's (1990, 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 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. We generalised Milner's context lemma to yield another co-inductive form of contextual equivalence, but offered evidence that it yields a weaker co-induction principle than bisimilarity.

Although our particular formulation is new, bisimilarity for functional languages is not. Often it is usually known as 'applicative bisimulation' and is based on a natural semantics style evaluation relation. The earliest reference I can find is to Abramsky's unpublished 1984 work on Martin-Löf's type theory, which eventually led to his study of lazy lambda-calculus[2] (Abramsky and Ong 1993). Other work includes papers by Howe (1989), Smith (1991), Sands (1992, 1995), Ong (1993), Pitts and Stark (1993), Rees (1994), Ritter and Pitts (1995), Crole and Gordon (1995) and my book (1994). The main novelty of the present work is our use of a labelled transition system to match contextual equivalence exactly in a typed setting, and our refinements of bisimulation in Section 4. It would be possible to rephrase our construction of $\sim$ directly in terms of $\Downarrow$, without using a labelled transition system, and still match contextual equivalence. We prefer the construction based on labels because it admits convenient proofs by diagram chasing, and emphasises the connection to CCS.

Bernstein and Stark (1995) 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. Their objective was to study debugging rather than to model contextual equivalence. In fact their form of bisimilarity is finer grained than contextual equivalence because individual reduction steps are observable. Mason, Smith, and Talcott (1994) 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. Felleisen and Friedman

[2]The earliest presentation of lazy lambda-calculus appears to be Abramsky's thesis (1987, Chapter 6), in which he explains that the "main results of Chapter 6 were obtained in the setting of Martin-Löf's Domain Interpretation of his Type Theory, during and shortly after a visit to Chalmers in March 1984."

(1989) also develop syntactic approaches to reasoning about programs.

Domain theory is the classical foundation of languages such as FPC, and indeed Pitts (1994a) 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 matching reductions, which formalises a simple intensional intuition, has no counterpart in Pitts' work. But there are certain properties, such as showing that a **Y** combinator in PCF returns the least fixpoint of its argument, that are more simply proved from a denotational than an operational semantics. The purpose of these notes is to show that much can be done from elementary operational foundations. No doubt more operational results will follow. Though purely operational methods are enough for many purposes, an adequate domain-theoretic denotational semantics is useful too. See Pitts' recent notes (1994b), for instance, for proofs of operational properties via denotational semantics.

Sangiorgi (1994) presents a refinement of co-induction, based on the notion of 'respectful functionals' on relations. Hopefully connections can be made with the present work, though $\langle - \rangle_{V*}$ and $\langle - \rangle_{+}$ of this paper do not directly take the form of respectful functionals.

Acknowledgements

The idea of defining bisimilarity on a deterministic functional language via a labelled transition system arose in joint work with Roy Crole (Crole and Gordon 1995). I am grateful for many conversations with colleagues at Cambridge and Chalmers, and for the opportunities to develop this material in lectures at Glasgow and Aarhus. I thank Glynn Winskel for the invitation to lecture at Aarhus, and Simon Peyton Jones and Phil Wadler for the invitation to Glasgow. Martin Coen suggested the `map/filter` example. John Hatcliff and Søren Bøgh Lassen pointed out errors in earlier versions of these notes. I began this work while a member of the Programming Methodology Group at Chalmers University in Gothenburg. At Cambridge I have been supported by the TYPES BRA and a University Research Fellowship from the Royal Society. Paul Taylor's prooftree and commutative diagrams packages helped typeset these notes.

References

Abramsky, S. (1987, October 5). **Domain Theory and the Logic of Observable Properties**. Ph. D. thesis, Queen Mary College, University of London.

Abramsky, S. and L. Ong (1993). Full abstraction in the lazy lambda calculus. **Information and Computation 105**, 159–267. Available as Technical Report 259, University of Cambridge Computer Laboratory.

Aczel, P. (1977). An introduction to inductive definitions. In J. Barwise (Ed.), **Handbook of Mathematical Logic**, pp. 739–782. North-Holland.

Bernstein, K. L. and E. W. Stark (1995). 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.

Berry, G. (1981). Some syntactic and categorical constructions of lambda-calculus models. Technical Report 80, INRIA.

Bird, R. and P. Wadler (1988). **Introduction to Functional Programming**. Prentice-Hall International.

Bloom, B. (1988). Can LCF be topped? Flat lattice models of typed lambda calculus. In **Proceedings of the 3rd IEEE Symposium on Logic in Computer Science**, pp. 282–295. IEEE Computer Society Press.

Camilleri, J. and T. Melham (1992, August). Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory.

Coquand, T. (1993). Infinite objects in type theory. In **Types of Proofs and Programs**, Volume 806 of **Lecture Notes in Computer Science**, pp. 62–78. Springer-Verlag.

Cousot, P. and R. Cousot (1979). Constructive versions of Tarski's fixed point theorems. **Pacific Journal of Mathematics 82**(1), 43–57.

Crole, R. L. and A. D. Gordon (1995, June). 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.

Davey, B. A. and H. A. Priestley (1990). **Introduction to Lattices and Order**. Cambridge University Press.

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

Felleisen, M. and D. P. Friedman (1989). A syntactic theory of sequential state. **Theoretical Computer Science 69**, 243–287.

Girard, J.-Y., P. Taylor, and Y. Lafont (1989). **Proofs and Types**. Cambridge University Press.

Gordon, A. D. (1994). **Functional Programming and Input/Output**. Cambridge University Press.

Gordon, A. D. (1995a). Bisimilarity as a theory of functional programming. In **Eleventh Annual Conference on Mathematical Foundations of Programming Semantics**, Volume 1 of **Electronic Notes in Theoretical Computer Science**. Elsevier Science Publishers B.V.

Gordon, A. D. (1995b). A tutorial on co-induction and functional programming. In **Functional Programming, Glasgow 1994**, Workshops in Computing, pp. 78–95. Springer-Verlag.

Gunter, C. A. (1992). **Semantics of Programming Languages: Structures and Techniques**. MIT Press, Cambridge, Mass.

Howe, D. J. (1989). Equality in lazy computation systems. In **Proceedings of the 4th IEEE Symposium on Logic in Computer Science**, pp. 198–203.

Hughes, J. (1989, April). Why functional programming matters. **The Computer Journal 32**(2), 98–107.

Hughes, J. and A. Moran (1995, June). Making choices lazily. In **FPCA'95, La Jolla**. ACM Press.

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

Meyer, A. R. and S. S. Cosmadakis (1988, July). Semantical paradigms: Notes for an invited lecture. In **Proceedings of the 3rd IEEE Symposium on Logic in Computer Science**, pp. 236–253.

Milner, R. (1977). Fully abstract models of typed lambda-calculi. **Theoretical Computer Science 4**, 1–23.

Milner, R. (1989). **Communication and Concurrency**. Prentice-Hall International.

Milner, R. and M. Tofte (1991). Co-induction in relational semantics. **Theoretical Computer Science 87**, 209–220.

Morris, J. H. (1968, December). **Lambda-Calculus Models of Programming Languages**. Ph. D. thesis, MIT.

Ong, C.-H. L. (1993, June). Non-determinism in a functional setting (extended abstract). In **Proceedings of the 8th IEEE Symposium on Logic in Computer Science**, pp. 275–286.

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

Park, D. (1981, March). Concurrency and automata on infinite sequences. In P. Deussen (Ed.), **Theoretical Computer Science: 5th GI-Conference, Karlsruhe**, Volume 104 of **Lecture Notes in Computer Science**, pp. 167–183. Springer-Verlag.

Paulson, L. C. (1993). Co-induction and co-recursion in higher-order logic. Technical Report 304, University of Cambridge Computer Laboratory.

Pitts, A. and I. Stark (1993, June). 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**, pp. 31–45.

Pitts, A. M. (1994a). A co-induction principle for recursively defined domains. **Theoretical Computer Science 124**, 195–219.

Pitts, A. M. (1994b, December). Some notes on inductive and co-inductive techniques in the semantics of functional programs. BRICS Notes Series NS–94–5, BRICS, Aarhus University.

Plotkin, G. D. (1977). LCF considered as a programming language. **Theoretical Computer Science 5**, 223–255.

Rees, G. (1994, April). Observational equivalence for a polymorphic lambda calculus. University of Cambridge Computer Laboratory. Available from the author.

Riecke, J. G. (1993). Fully abstract translations between functional languages. **Mathematical Structures in Computer Science 3**, 387–415.

Ritter, E. and A. M. Pitts (1995). A fully abstract translation between a λ-calculus with reference types and Standard ML. In **Proceedings TLCA'95, Edinburgh**.

Sander, H. (1992). **A Logic of Functional Programs with an Application to Concurrency**. Ph. D. thesis, Programming Methodology

Group, Chalmers University of Technology and University of Gothenburg.

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

Sands, D. (1995, January). Total correctness by local improvement in program transformation. In **Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)**. ACM Press.

Sangiorgi, D. (1994, August). On the bisimulation proof method. Technical Report ECS–LFCS–94–299, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Scott, D. S. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. **Theoretical Computer Science 121**, 411–440. Annotated reprint of Scott's 1969 manuscript.

Smith, S. F. (1991). From operational to denotational semantics. In **MFPS VII, Pittsburgh**, Volume 598 of **Lecture Notes in Computer Science**, pp. 54–76. Springer-Verlag.

Strachey, C. (1967, August). Fundamental concepts in programming languages. Unpublished lectures given at the International Summer School in Computer Programming, Copenhagen.

Turner, D. (Ed.) (1990). **Research Topics in Functional Programming**. Addison-Wesley.

Wadler, P. (1992). Comprehending monads. **Mathematical Structures in Computer Science 2**, 461–493.

Winskel, G. (1993). **The Formal Semantics of Programming Languages**. MIT Press, Cambridge, Mass.

Recent Publications in the BRICS Notes Series

NS-95-3 **Andrew D. Gordon.** ***Bisimilarity as a Theory of Functional Programming. Mini-Course*****. July 1995. iv+59 pp.**

NS-95-2 **Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors.** ***Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS*** **(Aarhus, Denmark, 19–20 May, 1995), May 1995. vi+334 pp.**

NS-95-1 **Igor Walukiewicz.** ***Notes on the Propositional μ-calculus: Completeness and Related Results*****. February 1995. 54 pp.**

NS-94-6 **Uffe H. Engberg, Kim G. Larsen, and Peter D. Mosses, editors.** ***Proceedings of the 6th Nordic Workshop on Programming Theory*** **(Aarhus, Denmark, 17–19 October, 1994), December 1994. v+483pp.**

NS-94-5 **Andrew M. Pitts.** ***Some Notes on Inductive and Co-Inductive Techniques in the Semantics of Functional Programs,*** **DRAFT VERSION. December 1994. vi+135 pp.**

NS-94-4 **Peter D. Mosses, editor.** ***Abstracts of the 6th Nordic Workshop on PROGRAMMING THEORY*** **(Aarhus, Denmark, 17–19 October, 1994), October 1994. v+52 pp.**

NS-94-3 **Sven Skyum, editor.** ***Complexity Theory: Present and Future*** **(Aarhus, Denmark, 15–18 August, 1994), September 1994. v+213 pp.**

NS-94-2 **David Basin.** ***Induction Based on Rippling and Proof Planning. Mini-Course*****. August 1994. 62 pp.**

NS-94-1 **Peter D. Mosses, editor.** ***Proc. 1st International Workshop on Action Semantics*** **(Edinburgh, 14 April, 1994), May 1994. 145 pp.**