To appear in the Proceedings of the 1994 Glasgow Workshop on Functional Programming, September 8–10, 1994, Ayr, Scotland. Springer Workshops in Computing, 1995.

A Tutorial on Co-induction and Functional Programming

Andrew D. Gordon*
University of Cambridge Computer Laboratory,
New Museums Site, Cambridge CB2 3QG, United Kingdom.
adg@cl.cam.ac.uk

Abstract

Co-induction is an important tool for reasoning about unbounded structures. This tutorial explains the foundations of co-induction, and shows how it justifies intuitive arguments about lazy streams, of central importance to lazy functional programmers. We explain from first principles a theory based on a new formulation of bisimilarity for functional programs, which coincides exactly with Morris-style contextual equivalence. We show how to prove properties of lazy streams by co-induction and derive Bird and Wadler's Take Lemma, a well-known proof technique for lazy streams.

The aim of this paper is to explain why co-inductive definitions and proofs by co-induction are useful to functional programmers.

Co-induction is dual to induction. To say a set is **inductively defined** just means it is the least solution of a certain form of inequation. For instance, the set of natural numbers $\mathbb{N}$ is the least solution (ordered by set inclusion, $\subseteq$) of the inequation

$$\{0\} \cup \{S(x) \mid x \in X\} \;\subseteq\; X. \tag{1}$$

The corresponding **induction principle** just says that if some other set satisfies the inequation, then it contains the inductively defined set. To prove a property of all numbers, let X be the set of numbers with that property and show that X satisfies inequation (1). If so then $\mathbb{N} \subseteq X$, since $\mathbb{N}$ is the least such set. This is simply mathematical induction.

Dually, a set is **co-inductively defined** if it is the greatest solution of a certain form of inequation. For instance, suppose that $\leadsto$ is the reduction relation in a functional language. The set of divergent programs, $\uparrow$, is the greatest solution of the inequation

$$X \;\subseteq\; \{a \mid \exists b(a \leadsto b \,\&\, b \in X)\}. \tag{2}$$

The corresponding **co-induction principle** is just that if some other set satisfies the inequation, then the co-inductively defined set contains it. For instance,

*Royal Society University Research Fellow.

suppose that program Ω reduces to itself, that is, $\Omega \rightsquigarrow \Omega$. To see that Ω is contained in $\uparrow$, consider set $X = \{\Omega\}$. Since X satisfies inequation (2), $X \subseteq \uparrow$, as $\uparrow$ is the greatest such set. Hence Ω is a member of $\uparrow$.

Bisimilarity is an equality based on operational behaviour. This paper seeks to explain why bisimilarity is an important co-inductive definition for functional programmers. Bisimilarity was introduced into computer science by Park (1981) and developed by Milner in his theory of CCS (1989). Bisimilarity in CCS is based on labelled transitions. A transition $a \xrightarrow{\alpha} b$ means that program (process) a can perform an observable action α to become successor program b. Any program gives rise to a (possibly infinite) **derivation tree**, whose nodes are programs and whose arcs are transitions, labelled by actions. Two programs are bisimilar if they root the same derivation trees, when one ignores the syntactic structure at the nodes. Bisimilarity is a way to compare behaviour, represented by actions, whilst discarding syntactic structure.

Contextual equivalence (Morris 1968) is widely accepted as the natural notion of operational equivalence for PCF-like languages (Milner 1977; Plotkin 1977). Two programs are contextually equivalent if, whenever they are each inserted into a hole in a larger program of integer type, the resulting programs either both converge or both diverge. The main technical novelty of this paper is to show how to define a labelled transition system for PCF-like languages (for instance, Miranda and Haskell) such that bisimilarity—operationally-defined behavioural equivalence—coincides with Morris' contextual equivalence. By virtue of this characterisation of contextual equivalence we can prove properties of functional programs using co-induction. We intend in a series of examples to show how co-induction formally captures and justifies intuitive operational arguments.

We begin in Section 1 by showing how induction and co-induction derive, dually, from the Tarski-Knaster fixpoint theorem. Section 2 introduces the small call-by-name functional language, essentially PCF extended with pairing and streams, that is the vehicle for the paper. We make two conventional definitions of divergence and contextual equivalence. In Section 3 we make a co-inductive definition of divergence, prove it equals the conventional one, and give an example of a co-inductive proof. The heart of the paper is Section 4 in which we introduce bisimilarity and prove it coincides with contextual equivalence. We give examples of co-inductive proofs and state a collection of useful equational properties. We derive the Take Lemma of Bird and Wadler (1988) by co-induction. Section 5 explains why bisimilarity is a precongruence, that is, preserved by arbitrary contexts, using Howe's method (1989). We summarise the paper in Section 6 and discuss related work.

This paper is intended to introduce the basic ideas of bisimilarity and co-induction from first principles. It should be possible to apply the theory developed in Section 4 without working through the details of Section 5, the hardest of the paper. In a companion paper (Gordon 1994a) we develop further co-inductive tools for functional programs. For more examples of bisimulation proofs see Milner (1989) or Gordon (1994b), for instance.

Here are our mathematical conventions. As usual we regard a **relation** $\mathcal{R}$ on a set X to be a subset of $X \times X$. If $\mathcal{R}$ is a relation then we write $x \mathrel{\mathcal{R}} y$ to mean $(x, y) \in \mathcal{R}$. If $\mathcal{R}$ and $\mathcal{R}'$ are both relations on X then we write $\mathcal{R}\mathcal{R}'$ for their **relational composition**, that is, the relation such that $x\mathcal{R}\mathcal{R}'y$ iff there is z such that $x \mathrel{\mathcal{R}} z$ and $z \mathrel{\mathcal{R}'} y$. If $\mathcal{R}$ is a relation then $\mathcal{R}^{\mathrm{op}}$ is its **opposite**, the relation such that $x \mathrel{\mathcal{R}^{\mathrm{op}}} y$ iff $y \mathrel{\mathcal{R}} x$. If $\mathcal{R}$ is a relation, we write $\mathcal{R}^+$ for its transitive closure, and $\mathcal{R}^*$ for its reflexive and transitive closure.

1 A Tutorial on Induction and Co-induction

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 1

(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 1 (Tarski-Knaster)

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

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

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 1. 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.

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

Let us revisit the example of mathematical induction, mentioned in the introduce. 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

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

In other words, mathematical induction is a special case of this general framework. Winskel (1993) shows in detail how structural induction and rule induction, proof principles familiar to computer scientists, are induction principles obtained from particular kinds of inductive definition. As for examples of co-induction, Sections 3 and 4 are devoted to co-inductive definitions of program divergence and equivalence respectively. Aczel (1977) is the standard reference on inductive definitions. Davey and Priestley (1990) give a more recent account of fixpoint theory, including the Tarski-Knaster theorem.

2 A Small Functional Language

In this section we introduce a small call-by-name functional language. It is PCF extended with pairing and streams, a core fragment of a lazy language like Miranda or Haskell. We define its syntax, a type assignment relation, a 'one-step' reduction relation, $\leadsto$, and a 'big-step' evaluation relation, $\Downarrow$.

Let x and y range over a countable set of **variables**. The **types**, A, B, and **expressions**, e, are given by the following grammars.

$$\begin{array}{rcl} A, B & ::= & \mathtt{Int} \mid \mathtt{Bool} \mid A \to A \mid (A, A) \mid [A] \\ e & ::= & x \mid e\, e \mid \lambda x{:}A.\, e \mid \mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e \mid k^A \mid \Omega^A \mid \delta^A \end{array}$$

where k ranges over a finite collection of **builtin constants**, Ω is the **divergent constant** and δ ranges over a finite collection of **user-defined constants**. We assume these include `map`, `iterate`, `take` and `filter`; we give informal definitions below. The builtin constants are listed below. We say that k^A is

admissable if $k{:}A$ is an instance of one of the following schemas.

$$\begin{array}{ll} \underline{tt}, \underline{ff} : \mathtt{Bool} & \underline{i} : \mathtt{Int} \\ \mathtt{succ}, \mathtt{pred} : \mathtt{Int} \to \mathtt{Int} & \mathtt{zero} : \mathtt{Int} \to \mathtt{Bool} \\ \mathtt{fst} : (A, B) \to A & \mathtt{snd} : (A, B) \to B \\ \mathtt{Pair} : A \to B \to (A, B) & \mathtt{Nil} : [A] \\ \mathtt{Cons} : A \to [A] \to [A] & \mathtt{scase} : B \to (A \to [A] \to B) \to [A] \to B \end{array}$$

For each user-defined constant δ we assume given a definition $\delta{:}A \stackrel{\text{def}}{=} e_\delta$. In effect these are definitions by mutual recursion, as each body e_δ can contain occurrences of any constant; hence there is no need for an explicit `fix` operator. We identify expressions up to alpha-conversion; that is, renaming of bound variables. We write $e[e'/x]$ for the substitution of expression e' for each variable x free in expression e. A **context**, $\mathcal{C}$, is an expression with one or more **holes**. A hole is written as $[\,]$ and we write $\mathcal{C}[e]$ for the outcome of filling each hole in $\mathcal{C}$ with the expression e.

The **type assignment** relation

$$\Gamma \vdash e : A \qquad \text{where } \Gamma \text{ is } x_1{:}A_1, \ldots, x_n{:}A_n,$$

is given inductively by rules of simply typed λ-calculus plus

$$\frac{k^A \text{ admissable}}{\Gamma \vdash k^A : A} \qquad \frac{}{\Gamma \vdash \Omega^A : A} \qquad \frac{\delta{:}A \stackrel{\text{def}}{=} e}{\Gamma \vdash \delta : A}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Bool} \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_3 : A}{\Gamma \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 : A}$$

We assume that $\varnothing \vdash e_\delta : A$ is derivable whenever $\delta{:}A \stackrel{\text{def}}{=} e_\delta$ is a definition of a user-defined constant. Type assignment is unique in the sense that whenever $\Gamma \vdash e : A$ and $\Gamma \vdash e : B$, then $A = B$.

Given the type assignment relation, we can construct the following universal sets and relations.

$$\begin{array}{rcll} \mathit{Prog}(A) & \stackrel{\text{def}}{=} & \{e \mid \varnothing \vdash e : A\} & \text{(programs of type } A) \\ a, b \in \mathit{Prog} & \stackrel{\text{def}}{=} & \bigcup_A \mathit{Prog}(A) & \text{(programs of any type)} \\ \mathit{Rel}(A) & \stackrel{\text{def}}{=} & \{(a,b) \mid \{a,b\} \subseteq \mathit{Prog}(A)\} & \text{(total relation on } A \text{ programs)} \\ \mathcal{R}, \mathcal{S} \subseteq \mathit{Rel} & \stackrel{\text{def}}{=} & \bigcup_A \mathit{Rel}(A) & \text{(total relation on programs)} \end{array}$$

The operational semantics is a one-step reduction relation, $\rightsquigarrow \subseteq \mathit{Rel}$. It is inductively defined by the axiom schemes

$$\begin{array}{cc} (\lambda x.\, e)a \rightsquigarrow e[a/x] & \delta_i \rightsquigarrow e_i \quad \text{if } \delta_i \stackrel{\text{def}}{=} e_i \\ \Omega \rightsquigarrow \Omega & \mathtt{if}\, \underline{\ell}\, \mathtt{then}\, a_{tt}\, \mathtt{else}\, a_{ff} \rightsquigarrow a_\ell \quad \ell \in \{tt, ff\} \\ \mathtt{succ}\, \underline{i} \rightsquigarrow \underline{i+1} & \mathtt{pred}\, \underline{i+1} \rightsquigarrow \underline{i} \\ \mathtt{zero}\, \underline{0} \rightsquigarrow \underline{tt} & \mathtt{zero}\, \underline{i} \rightsquigarrow \underline{ff} \quad \text{if } i \neq 0 \\ \mathtt{fst}\, (\mathtt{Pair}\, a\, b) \rightsquigarrow a & \mathtt{snd}\, (\mathtt{Pair}\, a\, b) \rightsquigarrow b \\ \mathtt{scase}\, f\, b\, \mathtt{Nil} \rightsquigarrow b & \mathtt{scase}\, f\, b\, (\mathtt{Cons}\, a\, as) \rightsquigarrow f\, a\, as \end{array}$$

together with the scheme of structural rules

$$\frac{a \rightsquigarrow b}{\mathcal{E}[a] \rightsquigarrow \mathcal{E}[b]}$$

where $\mathcal{E}$ is an **experiment** (a kind of atomic **evaluation context** (Felleisen and Friedman 1986)), a context generated by the grammar

$$\begin{array}{rcl} \mathcal{E} & ::= & [\,]\,a \mid \texttt{succ}\,[\,] \mid \texttt{pred}\,[\,] \mid \texttt{zero}\,[\,] \mid \texttt{if}\,[\,]\,\texttt{then}\,a\,\texttt{else}\,b \\ & \mid & \texttt{fst}\,[\,] \mid \texttt{snd}\,[\,] \mid \texttt{scase}\,a\,b\,[\,]. \end{array}$$

In other words the single structural rule above abbreviates eight different rules, one for each kind of experiment. Together they specify a deterministic, call-by-name evaluation strategy. Now we can make the usual definitions of evaluation, convergence and divergence.

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

By expanding the definition we can easily check that $\Downarrow$ and $\Uparrow$ are complementary, that is, $a\Uparrow$ iff $\neg a\Downarrow$. We can characterise the answers returned by the evaluation relation, $\Downarrow$, as follows. Let an $\rightsquigarrow$ **normal program** be a program a such that $\neg(a \rightsquigarrow)$. Let a **value**, u or v, be a program generated by the grammar

$$v ::= \lambda x.\, e \mid k \mid k_2\, a \mid k_2\, a\, b \text{ where } k_2 \in \{\texttt{Pair}, \texttt{Cons}, \texttt{scase}\}.$$

Lemma 2 *A program is a value iff it is $\rightsquigarrow$ normal.*

Proof By inspection, each value is clearly $\rightsquigarrow$ normal. For the other direction, one can easily prove by structural induction on a, that a is a value if it is $\rightsquigarrow$ normal. ■

Two programs are contextually equivalent if they can be freely interchanged for one another in a larger program, without changing its observable behaviour. This is a form of Morris' "extensional equivalence" (Morris 1968). Here is the formal definition of **contextual equivalence**, $\simeq\ \subseteq \mathit{Rel}$. Recall that $\mathcal{C}$ stands for contexts.

$$\begin{array}{l} a \sqsubseteq b \text{ iff whenever } (\mathcal{C}[a], \mathcal{C}[b]) \in \mathit{Rel}(\texttt{Int})\text{, that } \mathcal{C}[a]\Downarrow \text{ implies } \mathcal{C}[b]\Downarrow. \\ a \simeq b \text{ iff } a \sqsubseteq b \text{ and } b \sqsubseteq a. \end{array}$$

We have formalised 'observable behaviour' as termination at integer type. The relation is unchanged if we specify that $\mathcal{C}[a]$ and $\mathcal{C}[b]$ should both evaluate to the same integer. Contextual equivalence does not discriminate on grounds of termination at function or pair type. For instance, we will be able to prove that $\Omega^{A\to B} \simeq \lambda x{:}A.\, \Omega^B$. The two would be distinguished in a call-by-value setting, since one diverges and the other converges, but in our call-by-name setting no context of integer type can tell them apart.

We have introduced the syntax and operational semantics of a small functional language. Our definitions of divergence and contextual equivalence are natural and intuitive, but do not lend themselves to proof. In the next two sections we develop co-inductive characterisations of both divergence and contextual equivalence. Hence we obtain a theory admitting proofs of program properties by co-induction.

3 A Co-inductive Definition of Divergence

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

$$\begin{aligned} \mathcal{D}(X) &\stackrel{\text{def}}{=} \{a \mid \exists b(a \rightsquigarrow b \;\&\; b \in X)\} \\ \uparrow &\stackrel{\text{def}}{=} \nu X.\, \mathcal{D}(X) \end{aligned}$$

We can easily see that $\mathcal{D}$ is monotone. Hence by its co-inductive definition we have:

$\uparrow$ is the greatest $\mathcal{D}$-dense set and $\uparrow = \mathcal{D}(\uparrow)$.

Hughes and Moran (1993) give an alternative, 'big-step', co-inductive formulation of divergence.

As a simple example we can show that $\Omega\uparrow$. Let $X_\Omega \stackrel{\text{def}}{=} \{\Omega\}$. X_Ω is $\mathcal{D}$-dense, that is, $X_\Omega \subseteq \mathcal{D}(X_\Omega)$, because $\Omega \rightsquigarrow \Omega$ and $\Omega \in X_\Omega$. So $X_\Omega \subseteq \uparrow$ by co-induction, and therefore $\Omega\uparrow$.

We have an obligation to show that this co-inductive definition matches the earlier one, that $a\Uparrow$ iff whenever $a \rightsquigarrow^* b$, then $b \rightsquigarrow$.

Theorem 2 $\Uparrow = \uparrow$.

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

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

4 A Co-inductive Definition of Equivalence

We begin with a **labelled transition system** that characterises the immediate observations one can make of a program. It is defined in terms of the one-step operational semantics, and in some sense characterises the interface between the language's interpreter and the outside world. It is a a family of relations $(\xrightarrow{\alpha} \subseteq \mathit{Prog} \times \mathit{Prog} \mid \alpha \in \mathit{Act})$, indexed by the set Act of **actions**. If we let Lit, the set of **literals**, indexed by ℓ, be $\{\mathit{tt}, \mathit{ff}\} \cup \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the actions are given as follows.

$$\alpha, \beta \in Act \stackrel{\text{def}}{=} Lit \cup \{@a \mid a \in Prog\} \cup \{\texttt{fst}, \texttt{snd}, \texttt{Nil}, \texttt{hd}, \texttt{tl}\}$$

We partition the set of types into active and passive types. The intention is that we can directly observe termination of programs of active type, but not those of passive type. Let a type be **active** iff it has the form `Bool`, `Int` or $[A]$. Let a type be **passive** iff it has the form $A \to B$ or $\texttt{Pair}\,A\,B$. Arbitrarily we define $\mathbf{0} \stackrel{\text{def}}{=} \Omega^{\texttt{Int}}$. Given these definitions, the labelled transition system may be defined inductively as follows.

$$\underline{\ell} \xrightarrow{\ell} \mathbf{0} \qquad \texttt{Nil} \xrightarrow{\texttt{Nil}} \mathbf{0} \qquad \texttt{Cons}\,a\,b \xrightarrow{\texttt{hd}} a \qquad \texttt{Cons}\,a\,b \xrightarrow{\texttt{tl}} b$$

$$\frac{a\,b \in Prog}{a \xrightarrow{@b} a\,b} \qquad \frac{a \in Prog((A,B))}{a \xrightarrow{\texttt{fst}} \texttt{fst}\,a} \qquad \frac{a \in Prog((A,B))}{a \xrightarrow{\texttt{snd}} \texttt{snd}\,a}$$

$$\frac{a \rightsquigarrow a'' \qquad a'' \xrightarrow{\alpha} a'}{a \xrightarrow{\alpha} a'} \begin{cases} a \in Prog(A) \\ A \text{ active} \end{cases}$$

The **derivation tree** of a program a is the potentially infinite tree whose nodes are programs, whose arcs are labelled transitions, and which is rooted at a. For instance, the trees of the constant Ω^A are empty if A is active. In particular, the tree of $\mathbf{0}$ is empty. We use $\mathbf{0}$ in defining the transition system to indicate that after observing the value of a literal there is nothing more to observe. Following Milner (1989), we wish to regard two programs as behaviourally equivalent iff their derivation trees are isomorphic when we ignore the syntactic structure of the programs labelling the nodes. We formalise this idea by requiring our behavioural equivalence to be a relation $\sim \subseteq Rel$ that satisfies property $(*)$: whenever $(a,b) \in Rel$, $a \sim b$ iff

(1) Whenever $a \xrightarrow{\alpha} a'$ there is b' with $b \xrightarrow{\alpha} b'$ and $a' \sim b'$;

(2) Whenever $b \xrightarrow{\alpha} b'$ there is 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 an $\langle - \rangle$-dense relation.

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

- **Similarity**, $\lesssim\, \subseteq \mathit{Rel}$, is $\nu\mathcal{S}.\,[\mathcal{S}]$.
- A **simulation** is an $[-]$-dense relation.

We can easily establish the following basic facts.

Lemma 3

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

(2) $\sim\, = \,\lesssim \cap \lesssim^{\mathrm{op}}$.

(3) *Both* $\rightsquigarrow\, \subseteq\, \sim$ *and* $\Downarrow\, \subseteq\, \sim$.

Proof These are easily proved by co-induction. We omit the details. Parts (2) and (3) depend on the determinacy of $\rightsquigarrow$. Part (1) corresponds to Proposition 4.2 of Milner (1989). ■

4.1 A co-inductive proof about lazy streams

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 (Cons x xs) = Cons (f x) (map f xs)
iterate f x = Cons x (iterate f (f x))
```

These could easily be turned into formal definitions of two user-defined constants, but we omit the details. Pattern matching on streams would be accomplished using `scase`. Intuitively the streams

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

are equal, because they both consist of the sequence

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

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 4 *If* $\mathcal{S} \subseteq \mathit{Rel}$ *is*

$$\{(\mathtt{iterate}\,f\,(f\,x), \mathtt{map}\,f\,(\mathtt{iterate}\,f\,x)) \mid \\ \exists A(x \in \mathit{Prog}(A)\ \&\ f \in \mathit{Prog}(A \to A))\}$$

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

Proof It suffices to show that $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle$ and $\sim \subseteq \langle \mathcal{S} \cup \sim \rangle$. The latter is obvious, as $\sim = \langle \sim \rangle$. To show $\mathcal{S} \subseteq \langle \mathcal{S} \cup \sim \rangle$ we must consider arbitrary a and b such that $a \mathrel{\mathcal{S}} b$, and establish that each transition $a \xrightarrow{\alpha} a'$ is matched by a transition $b \xrightarrow{\alpha} b'$, such that either $a' \mathrel{\mathcal{S}} b'$ 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.

$$\begin{array}{rl} a & \rightsquigarrow^{+}\ \texttt{Cons}\,(f\,x)\,(\texttt{iterate}\,f\,(f\,(f\,x))) \\ b & \rightsquigarrow^{+}\ \texttt{Cons}\,(f\,x)\,(\texttt{map}\,f\,(\texttt{iterate}\,f\,(f\,x))) \end{array}$$

Whenever $a \rightsquigarrow^{*} a'$ we can check that $a \xrightarrow{\alpha} a''$ iff $a' \xrightarrow{\alpha} a''$. 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 $\texttt{iterate}\,f\,(f\,(f\,x)) \mathrel{\mathcal{S}} \texttt{map}\,f\,(\texttt{iterate}\,f\,(f\,x))$. ■

Since $\mathcal{S} \cup \sim$ is $\langle - \rangle$-dense, it follows that $(\mathcal{S} \cup \sim) \subseteq \sim$. 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.

4.2 Operational Extensionality

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

Theorem 3 (Precongruence) *If $a \sim b$ then $\mathcal{C}[a] \sim \mathcal{C}[b]$ for any suitable context $\mathcal{C}$. The same holds for similarity, $\lesssim$.*

The proof is non-trivial; we shall postpone it till Section 5.

Lemma 5 $\sqsubseteq\!\!\!\!\sim \; = \; \lesssim$.

Proof ($\lesssim \subseteq \sqsubseteq\!\!\!\!\sim$) Suppose $a \lesssim b$, that $(\mathcal{C}[a], \mathcal{C}[b]) \in \mathit{Rel}(\texttt{Int})$ and that $\mathcal{C}[a]\Downarrow$. By precongruence, $\mathcal{C}[a] \lesssim \mathcal{C}[b]$, so $\mathcal{C}[b]\Downarrow$ too. Hence $a \sqsubseteq\!\!\!\!\sim b$ as required.

($\sqsubseteq\!\!\!\!\sim \subseteq \lesssim$) This follows if we can prove that contextual order $\sqsubseteq\!\!\!\!\sim$ is a simulation. The details are not hard, and we omit them. For full details of a similar proof see Lemma 4.29 of Gordon (1994b), which was based on Theorem 3 of Howe (1989). ■

Contextual equivalence and bisimilarity are the symmetrisations of contextual order and similarity, respectively. Hence a corollary, usually known as **operational extensionality** (Bloom 1988), is that bisimilarity equals contextual equivalence.

Theorem 4 (Operational Extensionality) $\simeq = \sim$.

4.3 A Theory of Bisimilarity

We have defined bisimilarity as a greatest fixpoint, shown it to be a co-inductive characterisation of contextual equivalence, and illustrated how it admits co-inductive proofs of lazy streams. In this section we shall note without proof various equational properties needed in a theory of functional programming. Proofs of similar properties, but for a different form of bisimilarity, can be found in Gordon (1994b). We noted already that $\rightsquigarrow \subseteq \sim$, which justifies a collection of beta laws. We can easily prove the following unrestricted eta laws by co-induction.

Proposition 1 (Eta) *If $a \in \mathit{Prog}(A \to B)$, $a \sim \lambda x.\, a\, x$.*

Proposition 2 (Surjective Pairing)
If $a \in \mathit{Prog}((A, B))$, $a \sim \mathtt{Pair}\,(\mathtt{fst}\, a)\,(\mathtt{snd}\, a)$.

Furthermore we have an unrestricted principle of extensionality for functions.

Proposition 3 (Extensionality) *Suppose $\{f, g\} \subseteq \mathit{Prog}(A \to B)$. If $f\, a \sim g\, a$ for any $a \in \mathit{Prog}(A)$, then $f \sim g$.*

Here are two properties relating Ω and divergence.

Proposition 4 (Divergence)

(1) *$\mathcal{E}[\Omega] \sim \Omega$ for any experiment $\mathcal{E}$.*

(2) *If $a\Uparrow$ then $a \sim \Omega$.*

As promised, we can prove that $\lambda x{:}A.\, \Omega^B \simeq \Omega^{A \to B}$, in fact by proving $\lambda x{:}A.\, \Omega^B \sim \Omega^{A \to B}$. Consider any $a \in \mathit{Prog}(A)$. We have $(\lambda x{:}A.\, \Omega^B)\, a \sim \Omega^B$ by beta reduction and $\Omega^{A \to B}\, a \sim \Omega^B$ by part (1) of the last proposition. Hence $\lambda x{:}A.\, \Omega^B \sim \Omega^{A \to B}$ by extensionality. In fact, then, the converse of (2) is false, for $\lambda x{:}A.\, \Omega^B \sim \Omega^{A \to B}$ but $\lambda x{:}A.\, \Omega^B\Downarrow$.

We can easily prove the following adequacy result.

Proposition 5 (Adequacy) *If $a \in \mathit{Prog}(A)$ and A is active, $a\Uparrow$ iff $a \sim \Omega$.*

The condition that A be active is critical, because of our example $\lambda x{:}A.\, \Omega^B \sim \Omega^{A \to B}$, for instance.

Every convergent program equals a value, but the syntax of values includes partial applications of curried function constants. Instead we can characterise each of the types by the simpler grammar of **canonical programs**.

$$c ::= \underline{\ell} \mid \lambda x.\, e \mid \mathtt{Pair}\, a\, b \mid \mathtt{Nil} \mid \mathtt{Cons}\, a\, b.$$

Proposition 6 (Exhaustion) *For any program $a \in \mathit{Prog}(A)$ there is a canonical program c with $a \sim c$ iff either a converges or A is passive.*

The λ, `Pair` and `Cons` operations are injective in the following sense.

Proposition 7 (Canonical Freeness)

(1) *If* $\lambda x{:}A.\, e \sim \lambda x{:}A.\, e'$ *then* $e[{}^{a}\!/\!{}_{x}] \sim e'[{}^{a}\!/\!{}_{x}]$ *for any* $a \in \mathit{Prog}(A)$.

(2) *If* $\mathtt{Pair}\, a_1\, a_2 \sim \mathtt{Pair}\, b_1\, b_2$ *then* $a_1 \sim b_1$ *and* $a_2 \sim b_2$.

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

4.4 Bird and Wadler's Take Lemma

Our final example in this paper is to derive Bird and Wadler's Take Lemma (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) (Cons x xs) = Cons x (take n xs)
```

Here is the key lemma.

Lemma 6 *Define* $\mathcal{S} \subseteq \mathit{Rel}$ *by* $a\,\mathcal{S}\,b$ *iff* $\forall n \in \mathbb{N}(\mathtt{take}\ \underline{n+1}\ a \sim \mathtt{take}\ \underline{n+1}\ b)$.

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

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

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

Proof Recall that values of stream type take the form `Nil` or $\mathtt{Cons}\, a\, b$. For any program, a, of stream type, either $a\Uparrow$ or there is a value v with $a \Downarrow v$. Hence for any stream a, either $a \sim \Omega$ (from $a\Uparrow$ by adequacy, Proposition 5) or $a \Downarrow \mathtt{Nil}$ or $a \Downarrow \mathtt{Cons}\, a'\, a''$. Note also the following easily proved lemma about transitions of programs of active type, such as streams.

Whenever $a \in \mathit{Prog}(A)$ and A active, $a \xrightarrow{\alpha} b$ iff $\exists$value v $(a \Downarrow v \xrightarrow{\alpha} b)$.

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

(2) We have

$$\mathtt{take}\ \underline{n+1}\ (\mathtt{Cons}\ a'\ a'') \sim \mathtt{take}\ \underline{n+1}\ b.$$

With $n = 0$ we have

$$\mathtt{Cons}\ a'\ \mathtt{Nil} \sim \mathtt{take}\ \underline{1}\ b$$

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

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

for any n, and hence $a' \sim b'$ and $a''\ \mathcal{S}\ b''$ by canonical freeness, Proposition 7.

(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.

(1) 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.

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

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

This completes the proof of (3). ∎

The Take Lemma is a corollary of (3) by co-induction.

Theorem 5 (Take Lemma) *Suppose $a, b \in Prog([A])$.*
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.

Example equations such as

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

(where $\circ$ is function composition) in which the stream processing function preserves the size of its argument are easily proved using either co-induction or the Take Lemma. In either case we proceed by a simple case analysis of whether $as\Uparrow$, $as \Downarrow \texttt{Nil}$ or $as \Downarrow \texttt{Cons}\,a\,as'$. Suppose however that $\texttt{filter}\,f$ is the stream processing function that returns a stream of all the elements a of its argument such that $f\,a \Downarrow \underline{tt}$. Intuitively the following equation should hold

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

but straightforward attacks on this problem using either the Take Lemma or co-induction in the style of Lemma 4 fail. The trouble is that the result stream may not have as many elements as the argument stream.

These proof attempts can be repaired by resorting to a more sophisticated analysis of *as* than above. Lack of space prevents their inclusion, but in this way we can obtain proofs of the equation using either the Take Lemma or a simple co-induction. Alternatively, by more refined forms of co-induction—developed elsewhere (Gordon 1994a)—we can prove such equations using a simple-minded case analysis of the behaviour of *as*. These proof principles need more effort to justify than the Take Lemma, but in problems like the `map/filter` equation are easier to use.

5 Proof that Bisimilarity is a Precongruence

In this section we make good our promise to show that bisimilarity and similarity are precongruences, Theorem 3. We need to extend relations such as bisimilarity to open expressions rather than simply programs. Let a **proved expression** be a triple (Γ, e, A) such that $\Gamma \vdash e : A$. If $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$, a **$\Gamma$-closure** is a substitution $\cdot[\vec{a}/\vec{x}]$ where each $a_i \in \mathit{Prog}(A_i)$. Now if $\mathcal{R} \subseteq \mathit{Rel}$, let its **open extension**, $\mathcal{R}^\circ$, be the least relation between proved expressions such that

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

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

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

$$\Gamma \vdash e\, \widehat{\mathcal{R}}\, e \text{ if } e \in \{x, k, \Omega, \delta_j\}$$

$$\frac{\Gamma, x{:}A \vdash e\, \mathcal{R}\, e'}{\Gamma \vdash \lambda x{:}A.\, e\, \widehat{\mathcal{R}}\, \lambda x{:}A.\, e'} \qquad \frac{\Gamma \vdash e_1\, \mathcal{R}\, e_1' \qquad \Gamma \vdash e_2\, \mathcal{R}\, e_2'}{\Gamma \vdash e_1\, e_2\, \widehat{\mathcal{R}}\, e_1'\, e_2'}$$

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

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

Lemma 7 *Assume that $\mathcal{R} \subseteq \mathit{Rel}^\circ$ is a preorder. $\mathcal{R}$ is a precongruence iff $\Gamma \vdash \mathcal{C}[e]\, \mathcal{R}\, \mathcal{C}[e']$ whenever $\Gamma \vdash e\, \mathcal{R}\, e'$ and $\mathcal{C}$ is a context.*

The proof of the 'only if' direction is by induction on the size of context $\mathcal{C}$; the other direction is straightforward. Note that whenever a and b are programs of type A, that $a \sim b$ iff $(\varnothing, a, A) \sim^\circ (\varnothing, b, A)$, and similarly for similarity, $\lesssim$. Hence given the Lemma 7, to prove Theorem 3 it will be enough to show that $\sim^\circ$ and $\lesssim^\circ$ are precongruences, that is $\widehat{\sim^\circ} \subseteq \sim^\circ$ and $\widehat{\lesssim^\circ} \subseteq \lesssim^\circ$.

We shall use a general method established by Howe (1989). First we prove that the open extension of similarity is a precongruence. We define a second relation $\lesssim^\bullet$, which by construction satisfies $\widehat{\lesssim^\bullet} \subseteq \lesssim^\bullet$ and $\lesssim^\circ \subseteq \lesssim^\bullet$. We prove by co-induction that $\lesssim^\bullet \subseteq \lesssim^\circ$. Hence $\lesssim^\bullet$ and $\lesssim^\circ$ are one and the same relation, and $\lesssim^\circ$ is a precongruence because $\lesssim^\bullet$ is.

Second we prove that the open extension of bisimilarity is a precongruence. Let $\gtrsim\, = \,\lesssim^{\mathrm{op}}$. Recall Lemma 3(2), that $\sim\, = \,\lesssim \cap \gtrsim$. Furthermore $\sim^\circ\, = \,\lesssim^\circ \cap \gtrsim^\circ$ follows by definition of open extension. We can easily prove another fact, that $\widehat{\mathcal{R} \cap \mathcal{S}} = \widehat{\mathcal{R}} \cap \widehat{\mathcal{S}}$ whenever $\mathcal{R}, \mathcal{S} \subseteq \mathit{Rel}^\circ$. We have

$$\widehat{\sim^\circ} \; = \; (\widehat{\lesssim^\circ \cap \gtrsim^\circ}) \; = \; \widehat{\lesssim^\circ} \cap \widehat{\gtrsim^\circ} \; \subseteq \; \lesssim^\circ \cap \gtrsim^\circ \; = \; \sim^\circ$$

which is to say that $\sim^\circ$ is a precongruence. Indeed, being an equivalence relation, it is a congruence.

We have only sketched the first part, that $\lesssim^\circ$ is a precongruence. We devote the remainder of this section to a more detailed account. Compatible refinment, $\widehat{-}$, permits a concise inductive induction of Howe's relation $\lesssim^\bullet \subseteq \mathit{Rel}^\circ$ as $\mu\mathcal{S}.\,\widehat{\mathcal{S}}\lesssim^\circ$, which is to say that $\lesssim^\bullet$ is the least relation to satisfy the rule

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

Sands (1992) found the following neat presentation of some basic properties of $\lesssim^\bullet$ from Howe's paper.

Lemma 8 (Sands) *$\lesssim^\bullet$ is the least relation closed under the rules*

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

We claimed earlier that $\widehat{\lesssim^\bullet} \subseteq \lesssim^\bullet$ and $\lesssim^\circ \subseteq \lesssim^\bullet$; these follow from the lemma. The proof is routine, as is that of the following substitution lemma.

Lemma 9 *If $\Gamma, x{:}B \vdash e_1 \lesssim^\bullet e_2$ and $\Gamma \vdash e_1' \lesssim^\bullet e_2' : B$ then $\Gamma \vdash e_1[{}^{e_1'}\!/\!_x] \lesssim^\bullet e_2[{}^{e_2'}\!/\!_x]$.*

What remains of Howe's method is to prove that $\lesssim^\bullet \subseteq \lesssim^\circ$, which we do by co-induction. First note the following lemma—which is the crux of the proof—relating $\lesssim^\bullet$ and transition.

Lemma 10 *Let $\mathcal{S} \stackrel{\mathrm{def}}{=} \{(a,b) \mid \varnothing \vdash a \lesssim^\bullet b\}$.*

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

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

Proof The proofs are induction on the depth of inference of reduction $a \rightsquigarrow a'$ and transition $a \xrightarrow{\alpha} a'$ respectively. Details of similar proofs may be found in Howe (1989) and Gordon (1994b). ■

By this lemma, $\mathcal{S}$ 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 (Lemma 9) and the reflexivity of $\lesssim^\bullet$ (Lemma 8 and reflexivity of $\lesssim^\circ$). Hence we have $\lesssim^\bullet \subseteq \lesssim^\circ$. But the reverse inclusion follows from Lemma 8, so in fact $\lesssim^\bullet = \lesssim^\circ$ and hence $\lesssim^\circ$ is a precongruence.

6 Summary and Related Work

We explained the dual foundations of induction and co-induction. We defined notions of divergence and contextual equivalence for a small functional language, an extension of PCF. We gave co-inductive characterisations of both divergence and contextual equivalence, and illustrated their utility by a series of examples and properties. In particular we derived the 'Take Lemma' of Bird and Wadler (1988). We explained Howe's method for proving that bisimilarity, our co-inductive formulation of contextual equivalence, is a precongruence. We hope to have shown both by general principles and specific examples that there is an easy path leading from the reduction rules that define a functional language to a powerful theory of program equivalence based on co-induction.

Although our particular formulation is new, bisimilarity for functional languages is not. Often it is 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[1] (Abramsky and Ong 1993). Other work includes papers by Howe (1989), Smith (1991), Sands (1992, 1994), Ong (1993), Pitts and Stark (1993), Ritter and Pitts (1994), Crole and Gordon (1994) and my book (1994b). The present formulation is the first to coincide with contextual equivalence for PCF-like languages. It amounts to a co-inductive generalisation of Milner's original term model for PCF (1977). Since it equals contextual equivalence it answers Turner's (1990, Preface) concern that Abramsky's applicative bisimulation makes more distinctions than are observable by well-typed program contexts.

Domain theory is one perspective on the foundations of lazy functional programming; this paper offers another. Any subject benefits from multiple perspectives. In this case the two are of about equal expressiveness. Domain theory is independent of syntax and operational semantics, and provides fixpoint induction for proving program properties. If we take care to distinguish denotations from texts of programs, the theory of bisimilarity set out in Section 4 can be paralleled by a theory based on a domain-theoretic denotational semantics. Winskel (1993), for instance, shows how to prove adequacy for a lazy language with recursive types (albeit one in which functions and pairs are active types). Pitts (1994) develops a co-induction principle from domain theory. On the other hand, Smith (1991) shows how operational methods based on a form of bisimilarity can support fixpoint induction. One advantage of the operational approach is that bisimilarity coincides exactly with contextual equivalence. The corresponding property of a denotational semantics—full abstraction—is notoriously hard to achieve (Ong 1994).

[1] 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."

Acknowledgements

The idea of defining bisimilarity on a deterministic functional language via a labelled transition system arose in joint work with Roy Crole (1994). Martin Coen pointed out the `map/filter` example to me. I hold a Royal Society University Research Fellowship. This work has been partially supported by the CEC TYPES BRA, but was begun while I was a member of the Programming Methodology Group at Chalmers. I benefitted greatly from presenting a tutorial on this work to the Functional Programming group at Glasgow University. I am grateful to colleagues at the Ayr workshop, and at Chalmers and Cambridge, for many useful conversations.

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.

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

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

Bloom, B. (1988). Can LCF be topped? Flat lattice models of typed lambda calculus. In **Proceedings 3rd LICS**, pp. 282–295.

Crole, R. L. and A. D. Gordon (1994, September). A sound metalogical semantics for input/output effects. In **Computer Science Logic'94, Kazimierz, Poland**. Proceedings to appear in Springer LNCS.

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.

Gordon, A. D. (1994a). Bisimilarity as a theory of functional programming. Submitted for publication.

Gordon, A. D. (1994b). **Functional Programming and Input/Output**. Cambridge University Press. Revision of 1992 PhD dissertation.

Howe, D. J. (1989). Equality in lazy computation systems. In **Proceedings 4th LICS**, pp. 198–203.

Hughes, J. and A. Moran (1993, June). Natural semantics for nondeterminism. In **Proceedings of El Wintermöte**, pp. 211–222. Chalmers PMG. Available as Report 73.

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

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

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 8th LICS**, 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**, Volume 104 of **Lecture Notes in Computer Science**, pp. 167–183. Springer-Verlag.

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**, pp. 31–45.

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

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

Ritter, E. and A. M. Pitts (1994, September). A fully abstract translation between a λ-calculus with reference types and Standard ML. To appear in TLCA'95.

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

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. (1994, May). Total correctness and improvement in the transformation of functional programs (1st draft). DIKU, University of Copenhagen.

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.

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

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