A Sound Metalogical Semantics for Input/Output Effects

Roy L. Crole[1] and Andrew D. Gordon[2]

[1] Dept. of Mathematics and Computer Science,
University of Leicester, University Road, Leicester LE1 7RH, United Kingdom.
rlc3@mcs.le.ac.uk
[2] University of Cambridge Computer Laboratory,
New Museums Site, Cambridge CB2 3QG, United Kingdom.
adg@cl.cam.ac.uk

Abstract. We study the longstanding problem of semantics for input/output (I/O) expressed using side-effects. Our vehicle is a small higher-order imperative language, with operations for interactive character I/O and based on ML syntax. Unlike previous theories, we present both operational and denotational semantics for I/O effects. We use a novel labelled transition system that uniformly expresses both applicative and imperative computation. We make a standard definition of bisimilarity and prove it is a congruence using Howe's method.
Next, we define a metalogical type theory $\mathcal{M}$ in which we may give a denotational semantics to $\mathcal{O}$. $\mathcal{M}$ generalises Crole and Pitts' FIX-logic by adding in a parameterised recursive datatype, which is used to model I/O. $\mathcal{M}$ comes equipped both with judgements of equality of expressions, and an operational semantics; $\mathcal{M}$ itself is given a domain-theoretic semantics in the category $\mathcal{CPPO}$ of cppos (bottom-pointed posets with joins of ω-chains) and Scott continuous functions. We use the $\mathcal{CPPO}$ semantics to prove that the equational theory is computationally adequate for the operational semantics using formal approximation relations. The existence of such relations uses key ideas from Pitts' recent work.
A monadic-style textual translation into $\mathcal{M}$ induces a denotational semantics on $\mathcal{O}$. Our final result justifies metalogical reasoning: if the denotations of two $\mathcal{O}$ programs are equal in $\mathcal{M}$ then the $\mathcal{O}$ programs are in fact operationally equivalent.

1 Motivation

Ever since McCarthy referred to the input/output (I/O) operations `READ` and `PRINT` in LISP 1.5 [15] as "pseudo-functions," I/O effects have been viewed with suspicion. LISP 1.5 was the original applicative language. Its core could be explained as applications of functions to arguments, but "pseudo-functions"—which effected "an action such as the operation of input-output"—could not. Explaining pseudo-functions that effect I/O is not a matter of semantic archaeology: although lazy functional programmers avoid unrestricted side-effects, this style of I/O is pervasive in imperative languages and persists in applicative ones such as LISP, Scheme and ML. But although both the latter are defined formally [17, 25] neither definition includes the I/O operations.

We address this longstanding but still pertinent problem by supplying both an operational and a denotational semantics for I/O effects. We work with a call-by-value PCF-like language, $\mathcal{O}$, equipped with interactive I/O operations analogous to those of LISP 1.5. We can think of $\mathcal{O}$ as a tiny higher-order imperative language, with an applicative syntax making it a fragment of ML. We adopt CCS-style bisimilarity as the natural operational equivalence on $\mathcal{O}$ programs. Our first theorem is congruence of bisimilarity, via Howe's method [14], justifying operationally-based equational reasoning about $\mathcal{O}$ programs.

The denotational semantics is specified in two stages. First, we give a denotational semantics to a metalogic $\mathcal{M}$ in the category $\mathcal{CPPO}$ of cppos and Scott continuous functions. Second, we give a formal translation of the types and expressions of $\mathcal{O}$ into those of $\mathcal{M}$. $\mathcal{M}$ is based on the equational fragment of Crole and Pitts' FIX-logic [5], but contains a single parameterised recursive datatype which is used to model computations engaged in I/O, and does not (explicitly) contain a fixpoint type. Following Plotkin's use of a metalogic to study object languages [24] we equip the programs (closed expressions) of $\mathcal{M}$ with an operational semantics. Our second theorem shows the 'good fit' between the domain-theoretic semantics of $\mathcal{M}$ and its operational semantics: we prove that the denotational semantics is sound and adequate with respect to the operational semantics.

To complete our study, we establish a close relationship between the operational semantics of each $\mathcal{O}$ program and that of its denotation. Hence we prove our third theorem: that if the denotations of two $\mathcal{O}$ programs are provably equal in the metalogic, the programs are in fact operationally equivalent. The proof is by co-induction: we can show that the relation between $\mathcal{O}$ programs of equal denotations is in fact a bisimulation, and hence contained in bisimilarity.

We overcame two principal difficulties in this study. First, although it is fairly straightforward to write down operational semantics rules for side-effects, the essential problem is to develop a useful operational equivalence. Witness the great current interest in ML plus concurrency primitives: there are many operational semantics [2, 13] but few if any developed notions of operational equivalence. Holmström [13] pioneered a stratified approach to mixing applicative and imperative features in which a CCS-style labelled transition system for the side-effects was defined in terms of a 'big-step' natural semantics for the applicative part of the language. But Holmström's approach fails for the languages of interest here, in which side-effects may be freely mixed with applicative computation. Instead, we solve the problem of finding a suitable operational equivalence by expressing both the applicative and the side-effecting aspects of $\mathcal{O}$ in a single labelled transition system, a family $(\xrightarrow{\alpha} \mid \alpha \in \mathit{Act})$, of binary relations on $\mathcal{O}$ programs indexed by a set of actions, Act. The actions correspond to the atomic observations one can make of an $\mathcal{O}$ program. Milner's classical definition of (strong) bisimilarity from CCS [16] generates a natural operational equivalence, which subsumes both Abramsky's applicative bisimulation [1] and the stratified equivalences suggested by Holmström's semantics [10, 11]. The second main difficulty was the construction of formal approximation relations in the proof of adequacy for $\mathcal{M}$. Proof of their existence is complicated by the presence in $\mathcal{M}$ of a parameterised recursive type needed to model $\mathcal{O}$ computations engaged in I/O. Our proof makes use of recent work on algebraic completeness by Freyd [9] and Pitts [21].

As usual, we identify phrases of syntax up to alpha-conversion, that is, renaming of bound variables. We write $\phi \equiv \psi$ to mean that phrases ϕ and ψ are alpha-convertible. We write $\phi[\psi/x]$ for the substitution of phrase ψ for each variable x free in phrase ϕ. A **context**, $\mathcal{C}$, is a phrase of syntax with one or more **holes**. A hole is written as $[\,]$ and we write $\mathcal{C}[\phi]$ for the outcome of filling each hole in $\mathcal{C}$ with the phrase ϕ. If $\mathcal{R}$ is a relation, $\mathcal{R}^+$ is its transitive closure, and $\mathcal{R}^*$ its reflexive and transitive closure.

2 The object language $\mathcal{O}$

$\mathcal{O}$ is a call-by-value version of PCF, including constants for I/O. The **types** of $\mathcal{O}$, ranged over by τ, consist of ground types `unit`, `bool`, `int` and compound types τ`->`τ' and τ`*`τ', with the same intended meanings as in ML. Let *Lit*, ranged over by ℓ, be the set $\{\mathtt{true}, \mathtt{false}\} \cup \{\ldots, \mathtt{-2}, \mathtt{-1}, \mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$ of Boolean and integer **literals**, and let *Rator*, be the set $\{\mathtt{+}, \mathtt{-}, \mathtt{*}, \mathtt{=}, \mathtt{<}\}$ of arithmetic **operators**. Let notations $\underline{b}$ ($b = tt, f\!f$), $\underline{i}$ ($i \in \mathbb{Z}$) and $\underline{\oplus}$ ($\oplus \in \{+, -, \times, =, <\}$) range over the sets *Lit* and *Rator*. Let k range over the set of $\mathcal{O}$ **constants**, equal to

$$\{\mathtt{()}, \mathtt{fst}, \mathtt{snd}, \delta, \Omega, \mathtt{read}, \mathtt{write}\} \cup \mathit{Lit} \cup \mathit{Rator}.$$

Here is the grammar for $\mathcal{O}$ **expressions**,

$$e ::= k^\tau \mid x \mid \lambda x{:}\tau.\, e \mid e\, e \mid (e, e) \mid \mathtt{if}\, e\, \mathtt{then}\, e\, \mathtt{else}\, e$$

where x ranges over a countable set of variables. For the sake of simplicity there is just one user-definable constant, δ, and we assume a user-supplied declaration. $\mathtt{fun}\,\delta(x{:}\tau_\delta){:}\tau'_\delta \overset{\text{def}}{=} e_\delta$. The expression Ω^τ is one whose evaluation diverges. This is a spartan programming language, but it suffices to illustrate the semantics of side-effecting I/O.

The **type assignment** judgements are of the form $\Gamma \vdash e : \tau$, where the **environment**, Γ, is a list of variable-type pairs, $x{:}\tau_1, \ldots, x{:}\tau_n$.The provable judgements are generated by the usual monomorphic typing rules for this fragment of ML, where $\Gamma \vdash k^\tau : \tau$ is provable just when $k : \tau$ is an instance of one of the following type schemes.

$$\begin{array}{ll}
\mathtt{()} : \mathtt{unit} \qquad \underline{i} : \mathtt{int} & \mathtt{true}, \mathtt{false} : \mathtt{bool} \\
\delta : \tau_\delta \;\mathtt{->}\; \tau'_\delta & \Omega : \tau \\
\mathtt{+}, \mathtt{*}, \mathtt{-} : \mathtt{int} \;\mathtt{*}\; \mathtt{int} \;\mathtt{->}\; \mathtt{int} & \mathtt{=}, \mathtt{<} : \mathtt{int} \;\mathtt{*}\; \mathtt{int} \;\mathtt{->}\; \mathtt{bool} \\
\mathtt{fst} : \tau_1 \;\mathtt{*}\; \tau_2 \;\mathtt{->}\; \tau_1 & \mathtt{snd} : \tau_1 \;\mathtt{*}\; \tau_2 \;\mathtt{->}\; \tau_2 \\
\mathtt{read} : \mathtt{unit} \;\mathtt{->}\; \mathtt{int} & \mathtt{write} : \mathtt{int} \;\mathtt{->}\; \mathtt{unit}
\end{array}$$

We assume that $x{:}\tau_\delta \vdash e_\delta : \tau'_\delta$ is provable.

The set of **programs**, ranged over by p and q, is $Prog^{\mathcal{O}} \overset{\text{def}}{=} \{e \mid \exists\tau\,(\varnothing \vdash e : \tau)\}$. A **value expression**, *ve*, is an expression that is either a variable, a constant (but not Ω), a lambda-abstraction or a pair of value expressions. The set of **values**, $\mathit{Value}^{\mathcal{O}}$, ranged over by v or u, consists of the value expressions that are programs. Each program has a unique type, given the type annotations on constants and lambda-abstractions, though for notational convenience we often omit these annotations.

Before defining the labelled transition system that induces a behavioural equivalence on $\mathcal{O}$, we need to define the applicative reductions of $\mathcal{O}$. We define a call-by-value 'small-step' reduction relation, $\rightarrow\, \subseteq Prog^{\mathcal{O}} \times Prog^{\mathcal{O}}$, by the following axioms

$$\begin{array}{cc}
(\lambda x.\, e)\, v \rightarrow e[{}^{v}\!/\!_{x}] & \underline{\oplus}(\underline{i}, \underline{j}) \rightarrow \underline{i \oplus j} \\
\delta\, v \rightarrow e_\delta[{}^{v}\!/\!_{x}] & \Omega \rightarrow \Omega \\
\mathtt{fst}\,(u, v) \rightarrow u & \mathtt{snd}\,(u, v) \rightarrow v \\
\mathtt{if\ true\ then}\, p\, \mathtt{else}\, q \rightarrow p & \mathtt{if\ false\ then}\, p\, \mathtt{else}\, q \rightarrow q
\end{array}$$

together with the inference rule

$$\frac{p \rightarrow q}{\mathcal{E}[p] \rightarrow \mathcal{E}[q]}$$

where $\mathcal{E}$ is an **experiment**, a context specified by the grammar

$$\mathcal{E} ::= [\,]\, p \mid v\, [\,] \mid \mathtt{if}\, [\,]\, \mathtt{then}\, p\, \mathtt{else}\, q \mid ([\,], p) \mid (v, [\,]).$$

The rules for δ and Ω introduce the possibility of non-termination into $\mathcal{O}$. One can easily verify that the relation $\rightarrow$ is a partial function, and that it preserves types in the expected way. A **communicator** is a program ready to engage in I/O, that is, one of the form $\mathcal{C}[\mathtt{read}\,()]$ or $\mathcal{C}[\mathtt{write}\, n]$, where $\mathcal{C}$ is an **evaluation context**, a context made up of zero or more experiments. More precisely, such contexts are given by the grammar $\mathcal{C} = [\,] \mid \mathcal{E}[\mathcal{C}]$. If we let the set of **active programs**, *Active*, ranged over by a and b, be the union of the communicators and the values, we can easily show that the active programs are the normal forms of $\rightarrow$, that is:

Lemma 1. $\mathit{Active} = \{p \mid \neg\exists q(p \rightarrow q)\}$.

Our behavioural equivalence is based on a set of atomic observations, or **actions**, that may be observed of a program. This set, ranged over by α, is given by

$$\mathit{Act} \overset{\text{def}}{=} \mathit{Lit} \cup \{\mathtt{fst}, \mathtt{snd}, @v \mid v \in \mathit{Value}^{\mathcal{O}}\} \cup \mathit{Msg}$$

where *Msg*, a set of **messages**, represents I/O effects. Let *Msg*, ranged over by μ, be $\mathit{Msg} \overset{\text{def}}{=} \{?n, !n \mid n \in \mathbb{N}\}$, where $?n$ represents input of a number n and $!n$ output of n.

The **labelled transition system** is a family $(\xrightarrow{\alpha} \subseteq \mathit{Prog}^{\mathcal{O}} \times \mathit{Prog}^{\mathcal{O}} \mid \alpha \in \mathit{Act})$ of relations indexed by actions. It is inductively defined by the following rules

$$\underline{\ell} \xrightarrow{\ell} \Omega \qquad (u,v) \xrightarrow{\mathtt{fst}} u \qquad (u,v) \xrightarrow{\mathtt{snd}} v$$

$$u \xrightarrow{@v} u\,v \text{ if } u\,v \in \mathit{Prog}^{\mathcal{O}} \qquad \mathtt{read}\,() \xrightarrow{?n} \underline{n} \qquad \mathtt{write}\,n \xrightarrow{!n} ()$$

$$\frac{p \to p'' \qquad p'' \xrightarrow{\alpha} p'}{p \xrightarrow{\alpha} p'} \qquad \frac{p \xrightarrow{\mu} q}{\mathcal{E}[p] \xrightarrow{\mu} \mathcal{E}[q]}.$$

The last rule allows messages—but not arbitrary actions—to be observed as side-effects of subterms. Each transition arises from reduction to an active program.

Lemma 2. *$p \xrightarrow{\alpha} q$ iff $\exists a \in \mathit{Active}\,(p \to^{*} a \xrightarrow{\alpha} q)$.*

We write $p{\downarrow}$ to mean $\exists q \in \mathit{Active}(p \to^{*} q)$. Unless $p{\downarrow}$, p has no transitions. So Ω, for instance, has no transitions.

We adopt bisimilarity from Milner's CCS [16] as our operational equivalence for $\mathcal{O}$.[3] Any program p is the root of a potentially infinite **derivation tree**, whose nodes are programs and whose branches are labelled transitions. We regard two programs as behaviourally equivalent if they have the same derivation trees. The labels on the trees must match exactly, but we completely disregard the syntactic structure at their nodes.

We say a relation $\mathcal{S} \subseteq \mathit{Prog}^{\mathcal{O}} \times \mathit{Prog}^{\mathcal{O}}$ is a **bisimulation** iff $p\,\mathcal{S}\,q$ implies:

(1) whenever $p \xrightarrow{\alpha} p'$, there is q' with $q \xrightarrow{\alpha} q'$ and $p'\,\mathcal{S}\,q'$;
(2) whenever $q \xrightarrow{\alpha} q'$, there is p' with $p \xrightarrow{\alpha} p'$ and $p'\,\mathcal{S}\,q'$.

Then **bisimilarity**, $\sim \subseteq \mathit{Prog}^{\mathcal{O}} \times \mathit{Prog}^{\mathcal{O}}$, is the union of all bisimulations. It is standard to prove that bisimilarity is itself a bisimulation, and hence we have what amounts to a principle of co-induction:

Lemma 3. *$p \sim q$ iff there is a bisimulation $\mathcal{S}$ with $p\,\mathcal{S}\,q$.*

The main objective of this paper is to give a denotational semantics of $\mathcal{O}$ so that our metalogic $\mathcal{M}$ may be used to establish operational equivalences. Nonetheless, just as in CCS, the availability of co-induction means a great deal can be achieved simply using operational methods, provided that $\sim$ is a congruence. This is our first main result, which we can be proved via an adaptation of Howe's method; similar proofs can be found elsewhere [10, 12, 14].

Theorem 1. *Bisimilarity is a congruence.*

[3] In PCF-like languages, we often define two programs p and q to be observationally equivalent iff there is no program context $\mathcal{C}$ such that $\mathcal{C}[p]$ converges and $\mathcal{C}[q]$ diverges, and vice versa. This is inappropriate for our calculus because (unlike in CCS, say) contexts cannot observe the side-effects of a program. Any two communicators are contextually equivalent whether or not they are bisimilar.

3 The metalogic $\mathcal{M}$

We outline a Martin-Löf style type theory which will be used as a metalogic, $\mathcal{M}$, into which $\mathcal{O}$ may be translated and reasoned about—it is based on ideas from the FIX-Logic [5, 6], though $\mathcal{M}$ does not explicitly contain a fixpoint type. The (simple) types of $\mathcal{M}$ are given by

$$\sigma ::= X_0 \mid \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \sigma \times \sigma \mid \sigma \to \sigma \mid \sigma_\bot \mid \mathsf{U}(\sigma)$$

together with a **single** top-level recursive datatype declaration

$$\mathsf{datatype}\ \mathsf{U}(X_0) = c_1\ \mathsf{of}\ \sigma_1 \mid \cdots \mid c_n\ \mathsf{of}\ \sigma_n$$

in which any type $\mathsf{U}(\sigma)$ occurring in the σ_i is of the form $\mathsf{U}(X_0)$, and each function type in any σ_i has the form $\sigma \to \sigma'_\bot$ (thus the function types in the body of the recursive type are required to be partial). The use of these types in the modelling of $\mathcal{O}$ is essentially standard, but note that the single recursive datatype will be used in Section 4 to model I/O. The collection of (raw) expressions of $\mathcal{M}$ is given by the grammar in Figure 1.

$$\begin{array}{rcll}
E & ::= & x & \text{(variable)} \\
 & \mid & \langle\rangle & \text{(unit value)} \\
 & \mid & \lfloor \ell \rfloor & \text{(literal value)} \\
 & \mid & E \lfloor \oplus \rfloor E & \text{(arithmetic)} \\
 & \mid & \mathsf{If}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E & \text{(conditional)} \\
 & \mid & \langle E, E \rangle & \text{(pair)} \\
 & \mid & \mathsf{Split}\ E\ \mathsf{as}\ \langle x, y \rangle\ \mathsf{in}\ E & \text{(projection)} \\
 & \mid & c(E) & \text{(recursive data)} \\
 & \mid & \mathsf{Case}\ E\ \mathsf{of}\ c(x) \to E \mid \cdots \mid c(x) \to E & \text{(case analysis)} \\
 & \mid & \lambda x{:}\sigma.\, E & \text{(abstraction)} \\
 & \mid & E\, E & \text{(application)} \\
 & \mid & \mathsf{Lift}(E) & \text{(lifted value)} \\
 & \mid & \mathsf{Drop}\ E\ \mathsf{to}\ x\ \mathsf{in}\ E & \text{(sequential composition)} \\
 & \mid & \mathsf{Rec}\ x\ \mathsf{in}\ E & \text{(recursion)}
\end{array}$$

Fig. 1. Raw Expressions of the Metalogic $\mathcal{M}$, ranged over by E

Most of the syntax of $\mathcal{M}$ is standard [6, 7]. The types are either a type variable, a unit type, Booleans, integers, products, exponentials, liftings, or a single, parameterised recursive datatype whose body consists of a disjoint sum of instances of the latter types. Here, the expressions $\mathsf{Lift}(E)$ and $\mathsf{Drop}\ E_1\ \mathsf{to}\ x\ \mathsf{in}\ E_2$ give rise to an instance of (the type theory corresponding to) the lifting computational monad [19]. A **closed** type σ is one in which there are no occurrences of the type variable X_0, and we omit the easy formal definition. We define a type assignment system [4] for $\mathcal{M}$ which consists of rules for generating judgements of the form $\Gamma \vdash E{:}\sigma$, called **proved expressions**, where σ is a closed type, and the **environment** Γ is a finite list $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ of (variable, closed type) pairs. Most of the rules for generating these judgements are fairly standard [6]. We give a few example rules in Figure 2.

$$\frac{\Gamma \vdash E_1{:}\mathsf{Bool} \quad \Gamma \vdash E_2{:}\sigma \quad \Gamma \vdash E_3{:}\sigma}{\Gamma \vdash \mathsf{If}\, E_1 \,\mathsf{then}\, E_2 \,\mathsf{else}\, E_3{:}\sigma} \qquad \frac{\Gamma, x{:}\sigma_\perp \vdash E{:}\sigma_\perp}{\Gamma \vdash \mathsf{Rec}\, x \,\mathsf{in}\, E{:}\sigma_\perp} \qquad \frac{\Gamma \vdash E{:}\sigma_i[{}^{\sigma}\!/X_0]}{\Gamma \vdash c_i(E){:}\mathsf{U}(\sigma)}$$

$$\frac{\Gamma \vdash E{:}\mathsf{U}(\sigma) \quad \Gamma, x_1{:}\sigma_1[{}^{\sigma}\!/X_0] \vdash E_1{:}\sigma' \;\ldots\; \Gamma, x_n{:}\sigma_n[{}^{\sigma}\!/X_0] \vdash E_n{:}\sigma'}{\Gamma \vdash \mathsf{Case}\, E \,\mathsf{of}\, c_1(x_0) \to E_1 \mid \ldots \mid c_n(X_0) \to E_n \,{:}\sigma'}$$

Fig. 2. Example Rules for Generating Proved Expressions in $\mathcal{M}$

There is an equational theory for $\mathcal{M}$. A **theorem** of $\mathcal{M}$ takes the form $\Gamma \vdash E = E'{:}\sigma$ (where necessarily $\Gamma \vdash E{:}\sigma$ and $\Gamma \vdash E'{:}\sigma$ are proved expressions). The rules for generating the theorems are also fairly standard, and are omitted except for the example rules which are given in Figure 3. In the case that the environment Γ is empty, we shall write $E{:}\sigma$ and $E_1 = E_2{:}\sigma$. The set of $\mathcal{M}$ **programs of type** σ is $\mathit{Prog}^{\mathcal{M}}_{\sigma} \overset{\text{def}}{=} \{P \mid \exists\sigma(P{:}\sigma)\}$ and $\mathit{Prog}^{\mathcal{M}} \overset{\text{def}}{=} \bigcup_\sigma \mathit{Prog}^{\mathcal{M}}_{\sigma}$ is the set of $\mathcal{M}$ **programs**. Canonical forms V are given by the grammar

$$V \;::=\; \langle\rangle \mid \lfloor l \rfloor \mid \langle E, E\rangle \mid \lambda x{:}\sigma.\, E \mid \mathsf{Lift}(E) \mid c(E).$$

The set of $\mathcal{M}$ **values of type** σ is given by $\mathit{Value}^{\mathcal{M}}_{\sigma} \overset{\text{def}}{=} \{V \mid \exists\sigma(V{:}\sigma)\}$ and $\mathit{Value}^{\mathcal{M}} \overset{\text{def}}{=} \bigcup_\sigma \mathit{Value}^{\mathcal{M}}_{\sigma}$ is the set of $\mathcal{M}$ **values**.

$$\frac{\Gamma \vdash E{:}\sigma_i \qquad \Gamma, x_1{:}\sigma_1 \vdash E_1{:}\sigma \qquad \ldots \qquad \Gamma, x_n{:}\sigma_n \vdash E_n{:}\sigma}{\Gamma \vdash \mathsf{Case}\, c_i(E) \,\mathsf{of}\, c_1(x_1) \to E_1 \mid \ldots \mid c_n(x_n) \to E_n = E_i[{}^{E}\!/x_i]{:}\sigma}$$

$$\frac{\Gamma \vdash E{:}\mathsf{U}(\sigma) \qquad \Gamma, z{:}\mathsf{U}(\sigma) \vdash E'{:}\sigma'}{\Gamma \vdash \mathsf{Case}\, E \,\mathsf{of}\, c_1(x_1) \to E'[{}^{c_1(x_1)}\!/z] \mid \ldots \mid c_n(x_n) \to E'[{}^{c_n(x_n)}\!/z] = E'[{}^{E}\!/z]{:}\sigma'}$$

Fig. 3. Example Rules for the Equational Theory of E

Finally, we equip the syntax of $\mathcal{M}$ with an operational semantics. This is specified in two ways, first in the style of natural semantics 'big-step' reduction relations, and second in 'small-step' reduction relations. The former is specified via judgements of the form $P \Downarrow V$ where $\Downarrow \subseteq \mathit{Prog}^{\mathcal{M}} \times \mathit{Value}^{\mathcal{M}}$. The latter reductions take the form $P_1 \to P_2$, with P_1 and P_2 both $\mathcal{M}$ programs. We omit most of the rules for generating the operational semantics, except those associated with recursion and the recursive datatype which appear in Figure 4. Given any program P, we write $P{\Downarrow}$ to mean that there is a value V for which $P{\Downarrow}V$. As usual for a deterministic language we can prove that $P \Downarrow V$ iff $P \to^* V$.

$$\frac{E[\mathsf{Rec}\, x \,\mathsf{in}\, E/x] \Downarrow V}{\mathsf{Rec}\, x \,\mathsf{in}\, E \Downarrow V} \qquad \frac{E \Downarrow c_i(E') \qquad E_i[E'/x_i] \Downarrow V}{\mathsf{Case}\, E \,\mathsf{of}\, c_1(x_1) \to E_1 \mid \ldots \mid c_n(x_n) \to E_n \Downarrow V}$$

$$\mathsf{Split}\,\langle E_1, E_2\rangle \,\mathsf{as}\, \langle x, y\rangle \,\mathsf{in}\, E \to E[E_1, E_2/x, y]$$

Fig. 4. Examples of the reduction relations for $\mathcal{M}$

Our aim is to prove the following theorem.

Theorem 2.

(1) *If $P \in \mathit{Prog}^{\mathcal{M}}_{\sigma}$ and $P \to P'$, then $P' \in \mathit{Prog}^{\mathcal{M}}_{\sigma}$ and moreover $P = P'{:}\sigma$ is a theorem of $\mathcal{M}$.*

(2) *If $P = \mathsf{Lift}(P'){:}\sigma_\perp$ is a theorem of $\mathcal{M}$ then there exists a value V for which $V \in \mathit{Value}^{\mathcal{M}}_{\sigma_\perp}$ and $P \Downarrow V$.*

It is easy to prove the first part by rule induction on $P \to P'$. A corollary is that whenever $P \Downarrow V$, $P = V{:}\sigma$ is a theorem of $\mathcal{M}$.

In order to prove the second part, we first give a denotational semantics to $\mathcal{M}$ in the category $\mathcal{CPPO}$ of complete pointed posets (cppos) and Scott continuous functions. For us, a cppo is a poset which is complete in the sense of having joins of all ω-chains and pointed in the sense of having a bottom element. Closed types will be modelled by cppos, and the proved terms by Scott continuous functions. In order to set up the denotational semantics, we define a set of functors, each functor being of the form

$$\begin{array}{rcl} F_\sigma{:}\mathcal{CPO}_\perp^{op} \times \mathcal{CPO}_\perp \times \mathcal{CPO}_\perp^{op} \times \mathcal{CPO}_\perp & \to & \mathcal{CPO}_\perp \\ (X^-, X^+, Q^-, Q^+) & \mapsto & F_\sigma(X^-, X^+, Q^-, Q^+) \end{array}$$

where $\mathcal{CPO}_\perp$ is the category of cppos and **strict** continuous functions. These functors are introduced to provide convenient machinery for specifying the semantics of types, and for inducing functions which arise when we later prove the existence of certain logical relations. The cppos X^- and X^+ will model the parameter type variable X_0, with X^- modelling negative occurrences of X_0 and X^+ modelling positive occurrences. The cppos Q^- and Q^+ will play a role in modelling the recursive datatype declaration. The reader should also note that these functors are on the category of cppos and **strict** continuous functions—this is to take advantage of the minimal invariant cppos of Freyd and Pitts [9, 21]. The functors are defined by clauses such as

- $F_{\mathsf{Unit}}(X^-, X^+, Q^-, Q^+) \overset{\text{def}}{=} 1_\perp$,
- $F_{\sigma \to \sigma'} \overset{\text{def}}{=} F_\sigma(X^+, X^-, Q^+, Q^-) \to F_{\sigma'}(X^-, X^+, Q^-, Q^+)$,
- $F_{\mathsf{U}(X_0)}(X^-, X^+, Q^-, Q^+) \overset{\text{def}}{=} Q^+$,
- $F_{X_0}(X^-, X^+, Q^-, Q^+) \overset{\text{def}}{=} X^+$, and
- $F(X^-, X^+, Q^-, Q^+) \overset{\text{def}}{=} (\Sigma_1^n F_{\sigma_i}(X^-, X^+, Q^-, Q^+))_\perp$, (where Σ denotes a coproduct of cpos, itself a cppo).

The remaining clauses are omitted. The definition of the semantics of the closed types σ, written $[\![\sigma]\!]$, are as the reader expects, except possibly for a recursive type $\mathsf{U}(\sigma)$. There is, for each pair (X^-, X^+) of cppos, a functor

$$\mathcal{CPO}_\perp^{op} \times \mathcal{CPO}_\perp \xrightarrow{(Q^-, Q^+) \mapsto F(X^-, X^+, Q^-, Q^+)} \mathcal{CPO}_\perp .$$

We can then exhibit a family of cppos $(D(X^+, X^-) \mid X^+, X^- \in \mathcal{CPO}_\perp)$ which possess the simultaneous minimal invariant property—see [22]. In particular, there are isomorphisms $i{:}F(X^-, X^+, D(X^-, X^+), D(X^-, X^+)) \cong D(X^-, X^+)$ in $\mathcal{CPO}_\perp$, and we define $[\![\mathsf{U}(\sigma)]\!] \overset{\text{def}}{=} D([\![\sigma]\!], [\![\sigma]\!])$. Given a environment Γ we define $[\![\Gamma]\!]$ to be the cppo which is the product of the denotations of the types appearing in Γ, and we then specify a continuous function $[\![\Gamma \vdash E{:}\sigma]\!]{:}[\![\Gamma]\!] \to [\![\sigma]\!]$. The definition of these semantic functions is quite standard and omitted, but we do give the meaning of expressions associated with recursive types:

- If $e_j \overset{\text{def}}{=} [\![\Gamma \vdash E_j{:}\sigma_j[{}^{\sigma}\!/X_0]]\!]{:}[\![\Gamma]\!] \to [\![\sigma_j[{}^{\sigma}\!/X_0]]\!]$, and $\xi \in [\![\Gamma]\!]$, then we shall set $[\![\Gamma \vdash c_j(E_j){:}\mathsf{U}(\sigma)]\!](\xi) \overset{\text{def}}{=} i([in_j(e_j(\xi))]) \in [\![\mathsf{U}(\sigma)]\!]$ where one can show that there is an isomorphism $i{:}(\Sigma[\![\sigma_j[{}^{\sigma}\!/X_0]]\!])_\perp \cong [\![\mathsf{U}(\sigma)]\!]$ and in is the expected insertion function into the disjoint sum.
- If $e \overset{\text{def}}{=} [\![\Gamma \vdash E{:}\mathsf{U}(\sigma)]\!]{:}[\![\Gamma]\!] \to [\![\mathsf{U}(\sigma)]\!]$ and

$$e_j \overset{\text{def}}{=} [\![\Gamma, x_j{:}\sigma_j[{}^{\sigma}\!/X_0] \vdash E_j{:}\sigma']\!]{:}[\![\Gamma]\!] \times [\![\sigma_j[{}^{\sigma}\!/X_0]]\!] \to [\![\sigma']\!],$$

then

$$[\![\Gamma \vdash \mathsf{Case}\, E\, \mathsf{of}\, c_1(x_1) \to E_1 \mid \ldots \mid c_n(x_n) \to E_n{:}\sigma']\!](\xi)$$
$$\overset{\text{def}}{=} \begin{cases} e_j(\xi, \perp) \text{ if } i^{-1}(e(\xi)) = \perp \\ e_j(\xi, a_j) \text{ if } i^{-1}(e(\xi)) = [in_j(a_j)] \end{cases}$$

To prove Theorem 2, we shall show that there is a type indexed family of relations $\lhd_\sigma \subseteq [\![\sigma]\!] \times Prog_\sigma^{\mathcal{M}}$ satisfying certain conditions. Such formal approximation relations are fairly standard (see for example [7, 22, 24]) so we simply give these conditions at lifted and recursive types:

$$e \lhd_{\sigma_\bot} P \text{ iff } \exists d \in [\![\sigma]\!]. e = \mathit{lift}(d) \text{ implies } \exists P_1. P \Downarrow \mathsf{Lift}(P_1) \text{ and } d \lhd_\sigma P_1,$$

$$r \lhd_{\mathsf{U}(\sigma)} P \text{ iff } r = \bot \text{ or } \exists P_j. P \Downarrow c_j(P_j) \text{ and } \exists d_j. r = in_j(d_j) \text{ and } d_j \lhd_{\sigma_j[\sigma/X_0]} P_j.$$

We shall need the following lemma:

Lemma 4. *Given such a family of relations, if* $x_1{:}\sigma_1, \ldots, x_m{:}\sigma_m \vdash E{:}\sigma$ *and* $(d_k \lhd_{\sigma_k} P_k \mid 1 \leq k \leq m)$ *then* $[\![\Gamma \vdash E{:}\sigma]\!](\vec{d}) \lhd_\sigma E[\vec{P}/\vec{x}]$.

Proof. The proof is by induction on the structure of E, which is routine and omitted. □

In particular, it follows from Lemma 4 that $[\![P{:}\sigma]\!] \lhd_\sigma P$ for any $P \in \mathit{Prog}^{\mathcal{M}}_\sigma$. We can now complete the proof of Theorem 2. For suppose that $P_1 = \mathsf{Val}(P_2){:}\sigma_\bot$. Then by soundness of the semantics we have $[\![P_1{:}\sigma_\bot]\!] = \mathit{lift}([\![P_2{:}\sigma]\!]) \neq \bot$ and from Lemma 4 we have $[\![P_1{:}\sigma_\bot]\!] \lhd_{\sigma_\bot} P_1$. Hence, from the property of $\lhd_{\sigma_\bot}$ we deduce $P_1 \Downarrow \mathsf{Val}(P_3)$ for some P_3 as required.

The existence of the formal approximation relations can be proved by techniques which appear in Plotkin's CSLI notes [24]. However, it is more elegant to adapt Pitts' method of admissible actions on relational structures. We give an outline of the method. Set $\mathit{\Pi} \stackrel{\text{def}}{=} \{P{:}\sigma \mid P \in \mathit{Prog}^{\mathcal{M}}_\sigma, \sigma \in \mathit{Type}\}$ and for any cppo X put

$$\mathcal{R}(X) \stackrel{\text{def}}{=} \\ \{R \in \mathcal{P}(X \times \mathit{\Pi}) \mid \text{ for each } P{:}\sigma, \{x \mid (x, P{:}\sigma) \in R\} \subseteq X \text{ is chain complete}\}.$$

Write $D \stackrel{\text{def}}{=} D(1,1)$. We then define (monotone) functions, at each **closed** type σ, where $F_\sigma{:}\mathcal{R}(D)^{op} \times \mathcal{R}(D) \to \mathcal{R}(F_\sigma(1,1,D,D))$, by inductive clauses such as

$$F_{\sigma\to\sigma'}(R,S) \stackrel{\text{def}}{=} \{(f, P{:}\sigma \to \sigma') \mid f = \bot \text{ or} \\ P \Downarrow \lambda x.\, E' \text{ and } \forall (d, P_1{:}\sigma) \in F_\sigma(S,R). (f(d), E'[P_1/x]{:}\sigma') \in F_{\sigma'}(R,S)\}.$$

Using these functions, and lifting the function

$$F_{\sigma_j}(1,1,D,D) \xrightarrow{in} F(1,1,D,D) \cong D$$

to a (monotone) function $\mathcal{R}(F_{\sigma_j}(1,1,D,D)) \to \mathcal{R}(D)$ in a similar fashion, we arrive at a (monotone) function $\mathit{\Psi}{:}\mathcal{R}(D)^{op} \times \mathcal{R}(D) \to \mathcal{R}(D)$, and by symmetrising $\mathit{\Psi}$ we can take its least fixed point $(\mathit{\Delta}^-, \mathit{\Delta}^+) \in \mathcal{R}(D)^{op} \times \mathcal{R}(D)$. In fact using the minimal invariant property associated with D, we can show $\mathit{\Delta}^- = \mathit{\Delta}^+$ (the lengthy proof is omitted due to lack of space). We set

$$\lhd_\sigma \stackrel{\text{def}}{=} \{(d, P) \mid (d, P{:}\sigma) \in F_\sigma(1,1,D,D)\}.$$

4 The translation of $\mathcal{O}$ into $\mathcal{M}$

Following Plotkin [24] we induce a denotational semantics on $\mathcal{O}$, via a textual translation $(-)^{\mathcal{O}}$ of its types and expressions into $\mathcal{M}$. Each $\mathcal{O}$ type τ is sent to an $\mathcal{M}$ type $(\tau)^{\mathcal{O}}$ that models $\mathcal{O}$ values of type τ. We have $(\texttt{unit})^{\mathcal{O}} \stackrel{\text{def}}{=} \mathsf{Unit}$, $(\texttt{bool})^{\mathcal{O}} \stackrel{\text{def}}{=} \mathsf{Bool}$, $(\texttt{int})^{\mathcal{O}} \stackrel{\text{def}}{=} \mathsf{Int}$ and $(\tau_1 \texttt{*} \tau_2)^{\mathcal{O}} \stackrel{\text{def}}{=} (\tau_1)^{\mathcal{O}} \times (\tau_2)^{\mathcal{O}}$. Our translation of an $\mathcal{O}$ function, $(\tau_1 \texttt{->} \tau_2)^{\mathcal{O}}$, must model the "pseudo-functions" `read` and `write`, and so cannot simply be $(\tau_1)^{\mathcal{O}} \to (\tau_2)^{\mathcal{O}}$ (as of course McCarthy realised) but must be $(\tau_1)^{\mathcal{O}} \to \mathsf{T}(\tau_2)^{\mathcal{O}}$, where the range is a type of **computations** [19]. If τ is an $\mathcal{O}$ type, $\mathcal{M}$ type $\mathsf{T}(\tau)^{\mathcal{O}}$ is to represent the behaviour of $\mathcal{O}$ programs of type τ, including divergent programs and communicators as well as values. Using an idea that dates at least to Plotkin's Pisa notes [23, Chapter 5, Exercise 4], we set $\mathsf{T}\sigma \stackrel{\text{def}}{=} (\mathsf{U}(\sigma))_\perp$ given the following top-level $\mathcal{M}$ declaration:

$$\begin{array}{rcl} \mathsf{datatype}\,\mathsf{U}(X_0) & = & c_{rd}\ \mathsf{of}\ \mathsf{Int} \to \mathsf{U}(X_0)_\perp \\ & | & c_{wr}\ \mathsf{of}\ \mathsf{Int} \times \mathsf{U}(X_0)_\perp \\ & | & c_{ret}\ \mathsf{of}\ X_0 \end{array}$$

We may form programs of type $\mathsf{T}\sigma$ using the following definitions:

$$\begin{array}{rcl} \mathsf{Read}(E) & \stackrel{\text{def}}{=} & \mathsf{Lift}(c_{rd}(E)) \\ \mathsf{Write}(E_1, E_2) & \stackrel{\text{def}}{=} & \mathsf{Lift}(c_{wr}(\langle E_1, E_2 \rangle)) \\ \mathsf{Return}(E) & \stackrel{\text{def}}{=} & \mathsf{Lift}(c_{ret}(E)) \end{array}$$

Roughly speaking, a computation of type $\mathsf{T}(\tau)^{\mathcal{O}}$ consists of potentially unbounded strings of Read's or Write's terminated with either $\perp$ or a Return bearing an element of type $(\tau)^{\mathcal{O}}$. Hence $\mathsf{T}(\tau)^{\mathcal{O}}$ is a suitable semantic domain to model the behaviour of arbitrary $\mathcal{O}$ programs of type τ. It better models the interleaving of input and output than early denotational semantics models that passed around a state containing input and output sequences (see Mosses [18]).

$\mathcal{O}$ expressions are inductively translated into $\mathcal{M}$ expressions following the monadic style pioneered by Moggi [19] and Pitts [20]. The translation is parameterised by a monad (in the type-theoretic sense of Wadler [26]) $(\mathsf{T}, \mathsf{Val}, \mathsf{Let})$ where Val and Let are $\mathcal{M}$ combinators with the following types.

$$\begin{array}{ll} \mathsf{Val}\colon & \sigma \to \mathsf{T}\sigma \\ \mathsf{Let}\colon & \mathsf{T}\sigma \to (\sigma \to \mathsf{T}\sigma') \to \mathsf{T}\sigma'. \end{array}$$

(Strictly, speaking these are type schemes, and Val and Let are type-indexed families of combinators.) The idea behind this monadic translation is that Val and Let correspond to immediate termination and sequential composition respectively. We can define $\mathsf{Val} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{Return}(x)$ and Let has a recursive definition that roughly speaking stitches together the strings of I/O operations denoted by its two arguments,

$$
\begin{array}{ll}
\mathsf{Let} \stackrel{\text{def}}{=} & \mathsf{Fix}(\lambda let.\,\lambda x.\,\mathsf{Split}\,x\,\mathsf{as}\,\langle \hat{\imath o}, f\rangle\,\mathsf{in} \\
& \quad \mathsf{Drop}\,\hat{\imath o}\,\mathsf{to}\,\imath o\,\mathsf{in} \\
& \quad\quad \mathsf{Case}\,\imath o\,\mathsf{of} \\
& \quad\quad\quad c_{rd}(g) \to \mathsf{Read}(\lambda x.\,let\,\langle g\,x, f\rangle) \\
& \quad\quad\quad c_{wr}(x) \to \mathsf{Split}\,x\,\mathsf{as}\,\langle y, \hat{\imath o}'\rangle\,\mathsf{in}\,\mathsf{Write}(y, let\,\langle \hat{\imath o}', f\rangle)) \\
& \quad\quad\quad c_{ret}(x) \to f\,x)
\end{array}
$$

where $\mathsf{Fix}{:}(\sigma \to \sigma'_\perp) \to (\sigma \to \sigma'_\perp)$ is a fixpoint combinator defined from Rec [10]. (Note that let, $\imath o$ and $\hat{\imath o}$ and their primed variants are simply $\mathcal{M}$ variables.)

We simultaneously define the translation $(-)^{\mathcal{O}}$ of arbitrary $\mathcal{O}$ expressions to $\mathcal{M}$ expressions, and an auxiliary translation $|-|^{\mathcal{O}}$ of $\mathcal{O}$ value expressions. Here are the rules for value expressions

$$
\begin{array}{rcl}
|x|^{\mathcal{O}} & \equiv & x \\
|\texttt{()}|^{\mathcal{O}} & \equiv & \langle\rangle \\
|\underline{\ell}|^{\mathcal{O}} & \equiv & \lfloor \ell \rfloor \\
|\underline{\oplus}|^{\mathcal{O}} & \equiv & \lambda x.\,\mathsf{Split}\,x\,\mathsf{as}\,\langle y, y'\rangle\,\mathsf{in}\,\mathsf{Drop}\,y\,\lfloor\oplus\rfloor y'\,\mathsf{to}\,z\,\mathsf{in}\,\mathsf{Val}\,z \\
|\texttt{fst}|^{\mathcal{O}} & \equiv & \lambda x.\,\mathsf{Split}\,x\,\mathsf{as}\,\langle y, z\rangle\,\mathsf{in}\,\mathsf{Val}\,y \\
|\texttt{snd}|^{\mathcal{O}} & \equiv & \lambda x.\,\mathsf{Split}\,x\,\mathsf{as}\,\langle y, z\rangle\,\mathsf{in}\,\mathsf{Val}\,z \\
|\delta|^{\mathcal{O}} & \equiv & \mathsf{Fix}(\lambda f_\delta.\,\lambda x.\,(e_\delta[{}^{f_\delta}\!/\!_\delta])^{\mathcal{O}}) \qquad \text{where } \texttt{fun}\,\delta(x{:}\tau_\delta){:}\tau'_\delta \stackrel{\text{def}}{=} e_\delta \\
|\texttt{(}v, u\texttt{)}|^{\mathcal{O}} & \equiv & \langle |v|^{\mathcal{O}}, |u|^{\mathcal{O}}\rangle \\
|\lambda x{:}\tau.\,e|^{\mathcal{O}} & \equiv & \lambda x{:}(\tau)^{\mathcal{O}}.\,(e)^{\mathcal{O}} \\
|\texttt{read}|^{\mathcal{O}} & \equiv & \lambda x{:}\mathsf{Unit}.\,\mathsf{Read}(\mathsf{Val}) \\
|\texttt{write}|^{\mathcal{O}} & \equiv & \lambda x{:}\mathsf{Int}.\,\mathsf{Write}(x, \mathsf{Val}\,\langle\rangle)
\end{array}
$$

and here are the rules for arbitrary expressions, where $\mathsf{Let}\,x \Leftarrow E\,\mathsf{in}\,E'$ is an abbreviation for $\mathsf{Let}\,\langle E, \lambda x.\,E'\rangle$.

$$
\begin{array}{rcll}
(ve)^{\mathcal{O}} & \equiv & \mathsf{Return}(|ve|^{\mathcal{O}}) & (*) \\
(\Omega)^{\mathcal{O}} & \equiv & \mathsf{Rec}\,x\,\mathsf{in}\,x & \\
(\texttt{if}\,e_1\,\texttt{then}\,e_2\,\texttt{else}\,e_3)^{\mathcal{O}} & \equiv & \mathsf{Let}\,x \Leftarrow (e_1)^{\mathcal{O}}\,\mathsf{in}\,\mathsf{If}\,x\,\mathsf{then}\,(e_2)^{\mathcal{O}}\,\mathsf{else}\,(e_3)^{\mathcal{O}} & \\
(ve\,e')^{\mathcal{O}} & \equiv & \mathsf{Let}\,x \Leftarrow (e')^{\mathcal{O}}\,\mathsf{in}\,|ve|^{\mathcal{O}}\,x & (**) \\
(e\,e')^{\mathcal{O}} & \equiv & \mathsf{Let}\,f \Leftarrow (e)^{\mathcal{O}}\,\mathsf{in}\,\mathsf{Let}\,x \Leftarrow (e')^{\mathcal{O}}\,\mathsf{in}\,f\,x & \\
(\texttt{(}ve, e'\texttt{)})^{\mathcal{O}} & \equiv & \mathsf{Let}\,y \Leftarrow (e')^{\mathcal{O}}\,\mathsf{in}\,\mathsf{Val}\langle |ve|^{\mathcal{O}}, y\rangle & (**) \\
(\texttt{(}e, e'\texttt{)})^{\mathcal{O}} & \equiv & \mathsf{Let}\,x \Leftarrow (e)^{\mathcal{O}}\,\mathsf{in}\,\mathsf{Let}\,y \Leftarrow (e')^{\mathcal{O}}\,\mathsf{in}\,\mathsf{Val}\langle x, y\rangle &
\end{array}
$$

Rules marked $(*)$ and $(**)$ take precedence over later rules.

Lemma 5 (Correspondence).

(1) $\mathsf{Let}\,x \Leftarrow \mathsf{Return}(P)\,\mathsf{in}\,E \to^{+} E[{}^{P}\!/\!_{x}]$

(2) *If* $\Gamma, x{:}\tau \vdash e : \tau'$ *and* $\Gamma \vdash ve : \tau$ *then* $(e)^{\mathcal{O}}[|ve|^{\mathcal{O}}/x] \equiv (e[ve/x])^{\mathcal{O}}$.

(3) *If* $p \to q$ *then* $(p)^{\mathcal{O}} \to^{+} (q)^{\mathcal{O}}$.

Proof. (1) is a routine calculation. (2) is by induction on the structure of e and ve and (3) by induction on the derivation of $p \to q$. □

Without the prioritised translation rules marked $(**)$, part (3) would fail. If $p \to q$ we would have $(v, p) \to (v, q)$ but not $((v, p))^{\mathcal{O}} \to^{+} ((v, q))^{\mathcal{O}}$. However one can easily show that all the rules are valid up to provable equality in $\mathcal{M}$. Part (3) as proved makes the proof of Lemma 7 particularly simple.

Lemma 6. *If* $\mathcal{C}[\texttt{write}\,\underline{n}]$ *and* $\mathcal{C}[\texttt{read}\,()]$ *are communicators and* v *is a value,*

$$\begin{aligned}(v)^{\mathcal{O}} &= \mathsf{Return}(|v|^{\mathcal{O}})\\ (\mathcal{C}[\texttt{read}\,()])^{\mathcal{O}} &= \mathsf{Read}(\lambda x{:}\mathsf{Int}.\,(\mathcal{C}[x])^{\mathcal{O}})\\ (\mathcal{C}[\texttt{write}\,\underline{n}])^{\mathcal{O}} &= \mathsf{Write}(\lfloor n \rfloor, (\mathcal{C}[()])^{\mathcal{O}})\end{aligned}$$

are all $\mathcal{M}$ *theorems.*

Proof. The first equation follows by inspection. Proofs of the other two are by induction on the number of experiments making up evaluation context $\mathcal{C}$. □

Lemma 7 (Adequacy). $p\!\downarrow$ *iff* $(p)^{\mathcal{O}}\Downarrow$.

Proof. Use the last two lemmas and Theorem 2.

Lemma 8. *If* $(a)^{\mathcal{O}} = (b)^{\mathcal{O}}$ *and* $a \xrightarrow{\alpha} p$ *there is* q *with* $b \xrightarrow{\alpha} q$ *and* $(p)^{\mathcal{O}} = (q)^{\mathcal{O}}$.

Proof. By a simple case analysis and Lemma 6.

Lemma 9. *Relation* $\mathcal{S} \overset{\text{def}}{=} \{(p, q) \mid (p)^{\mathcal{O}} = (q)^{\mathcal{O}}\}$ *is a bisimulation.*

Proof. Suppose that $p\mathcal{S}q$ and that $p \xrightarrow{\alpha} p'$. By Lemma 2 there is a with $p \to^{*} a$ and $a \xrightarrow{\alpha} p'$. By Lemma 5 we have $(p)^{\mathcal{O}} \to^{*} (a)^{\mathcal{O}}$ and therefore $(p)^{\mathcal{O}} = (a)^{\mathcal{O}}$ by Theorem 2. By transitivity $(q)^{\mathcal{O}} = (a)^{\mathcal{O}}$ is derivable, so by Theorem 2 and Lemma 6 we have $(q)^{\mathcal{O}}\Downarrow$. Hence $q\!\downarrow$ by Lemma 7, that is, there is b with $q \to^{*} b$. By Lemma 5 and Theorem 2 we have $(q)^{\mathcal{O}} = (b)^{\mathcal{O}}$ and so $(a)^{\mathcal{O}} = (b)^{\mathcal{O}}$ by transitivity. Hence by Lemma 8 there is q' with $b \xrightarrow{\alpha} q'$ and $(p')^{\mathcal{O}} = (q')^{\mathcal{O}}$. Altogether we have $q \xrightarrow{\alpha} q'$ and $p' \, \mathcal{S} \, q'$. A symmetric argument shows that q can match any action of p, hence $\mathcal{S}$ is a bisimulation. □

The soundness of metalogical reasoning follows by co-induction, Lemma 3.

Theorem 3 (Soundness). $(p)^{\mathcal{O}} = (q)^{\mathcal{O}}$ *implies* $p \sim q$.

5 Discussion

By consolidating prior work on operational semantics, bisimulation equivalence and metalogics for denotational semantics, we have presented the most comprehensive study yet of I/O via side-effects. Previous work has treated denotational or operational semantics in isolation. Our study combines the two to admit proofs of programs based either on direct operational calculations (Theorem 1) or metalogical inference (Theorem 3).

Williams and Wimmers' paper [27] is perhaps the only other to consider an equational theory for a strict functional language with what amounts to side-effecting I/O, but they do not consider operational semantics. Similarly, the semantic domains for I/O studied in early work in the Scott-Strachey tradition of denotational semantics [18, 23] were not related to operational semantics. In his CSLI lecture notes, Plotkin [24] showed how Scott-Strachey denotational semantics could be reconciled with operational semantics by equipping his metalanguage (analogous to our $\mathcal{M}$) with an operational semantics. He showed for a given object language (analogous to $\mathcal{O}$) that the adequacy proof for the object language (analogous to Lemma 7) could be factored into an adequacy result for the metalanguage (analogous to Theorem 2) together with comparatively routine calculations about the operational semantics. Moggi [19] pioneered a monadic approach to modularising semantics. In an earlier study [7] we reworked Plotkin's framework in a monadic setting, for a simple applicative language.

We have made two main contributions to Plotkin's framework. First, by adapting recent advances in techniques for showing the existence of formal approximation relations we have a relatively straightforward proof of computational adequacy for a type theory with a parameterised recursive type. This avoids the direct construction of formal approximation relations using the limit/colimit coincidence (see for example [8]). Instead we use the minimal invariant property which characterises the (smallest) coincidence. Second, we use the adequacy result for $\mathcal{O}$ (Lemma 7) and co-induction to prove the soundness of metalogical reasoning with respect to operational equivalence (Theorem 3).

The idea of using a labelled transition system for a functional language, together with co-inductively defined bisimilarity, is perhaps the most important but the least familiar in this paper. It appears earlier in Boudol's concurrent γ-calculus [3], but Boudol does not establish whether bisimilarity on his calculus is a congruence. Abramsky's applicative bisimulation [1] is another co-inductively defined equivalence on functional languages but based on a 'big-step' natural semantics. Labelled transitions better express I/O, and hence are preferable to natural semantics for defining languages with I/O.

Acknowledgements We thank Simon Gay, Andrew Pitts and Eike Ritter for useful discussions. Roy Crole was supported by a Research Fellowship from the EPSRC. Andrew Gordon was funded by the Types BRA. This work was partially supported by the CLICS BRA.

References

1. Samson Abramsky and Luke Ong. Full abstraction in the lazy lambda calculus. **Information and Computation** 105:159–267, 1993.

2. Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In **19th POPL**, pages 119–129, 1992.
3. Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In **TAPSOFT'89**, Springer LNCS 351, 1989.
4. Roy. L. Crole. **Categories for Types**. CUP, 1993.
5. Roy. L. Crole and A. M. Pitts. New foundations for fixpoint computations: FIX hyperdoctrines and the FIX-logic. **Information and Computation**, 98:171–210, 1992.
6. Roy L. Crole. **Programming Metalogics with a Fixpoint Type**. PhD thesis, University of Cambridge, 1992.
7. Roy L. Crole and Andrew D. Gordon. Factoring an adequacy proof (preliminary report). In **Functional Programming, Glasgow 1993**, Springer 1994.
8. Marcello P. Fiore and Gordon D. Plotkin. An Axiomatisation of Computationally Adequate Domain Theoretic Models of FPC. In **9th LICS**, 1994.
9. P. Freyd. Algebraically complete categories. In **1990 Como Category Theory Conference**, Springer Lecture Notes in Mathematics, 1991.
10. Andrew D. Gordon. **Functional Programming and Input/Output**. CUP, 1994.
11. Andrew D. Gordon. An operational semantics for I/O in a lazy functional language. In **FPCA'93**, pages 136–145. 1993.
12. Andrew D. Gordon. A tutorial on co-induction and functional programming. In **Functional Programming, Glasgow 1994**. Springer Workshops in Computing.
13. Sören Holmström. PFL: A functional language for parallel programming. Report 7, Chalmers PMG. 1983.
14. Douglas J. Howe. Equality in lazy computation systems. In **4th LICS**, 1989.
15. John McCarthy et al. **LISP 1.5 Programmer's Manual**. MIT Press, 1962.
16. Robin Milner. **Communication and Concurrency**. Prentice-Hall, 1989.
17. R. Milner, M. Tofte and R. Harper. **The Definition of SML**. MIT Press, 1990.
18. Peter D. Mosses. Denotational semantics. In Jan Van Leeuven, editor, **Handbook of Theoretical Computer Science**, pages 575–631. Elsevier 1990.
19. Eugenio Moggi. Notions of computations and monads. **TCS**, 93:55–92, 1989.
20. Andrew M. Pitts. Evaluation logic. In **IVth Higher Order Workshop, Banff 1990**, pages 162–189. Springer 1991.
21. Andrew M. Pitts. Relational properties of domains. Tech. Report 321, University of Cambridge Computer Laboratory, December 1993.
22. Andrew M. Pitts. Computational adequacy via 'mixed' inductive definitions. In **MFPS IX, New Orleans 1993**, pages 72–82, Springer LNCS 802, 1994.
23. Gordon D. Plotkin. Pisa notes on domains, June 1978.
24. Gordon D. Plotkin. Denotational semantics with partial functions. Stanford CSLI 1985.
25. Jonathan Rees and William Clinger. Revised^3 report on the algorithmic language scheme. **ACM SIGPLAN Notices**, 21(12):37–79, December 1986.
26. Philip Wadler. The essence of functional programming. In **19th POPL**, 1992.
27. John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In **15th POPL**, 1988.

This article was processed using the LaTeX macro package with LLNCS style