Relating Operational and Denotational Semantics for Input/Output Effects

Roy L. Crole
Department of Mathematics and Computer Science,
University of Leicester, Leicester, UK.
email: rlc3@mcs.le.ac.uk

and

Andrew D. Gordon
University of Cambridge Computer Laboratory,
Cambridge, UK.
email: adg@cl.cam.ac.uk

April 1996

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. We prove bisimilarity is a congruence using Howe's method.

Next, we define a metalanguage $\mathcal{M}$ 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 an operational semantics and 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 $\mathcal{M}$ is computationally adequate for the operational semantics using formal approximation relations. The existence of such relations is based on recent work of Pitts [Pit94b] for untyped languages, and uses the idea of minimal invariant objects due to Freyd.

A monadic-style textual translation into $\mathcal{M}$ induces a denotational semantics on $\mathcal{O}$. Our final result validates the denotational semantics: if the denotations of two $\mathcal{O}$ programs are equal then the $\mathcal{O}$ programs are in fact operationally equivalent.

1 Motivation

Ever since McCarthy [MAE$^+$62] referred to the input/output (I/O) operations `READ` and `PRINT` in LISP 1.5 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 [MTH90, RC86] 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. In this paper we shall:

- define a CCS-style labelled transition semantics for $\mathcal{O}$;
- show that the associated bisimilarity is a congruence;
- define a domain-theoretic denotational semantics for $\mathcal{O}$;
- prove that denotational equality implies bisimilarity.

Our aim is to present such an approach to I/O in detail for a simple language and to concentrate on small examples; let us discuss some motivation and detail:

Morris-style contextual equivalence is often adopted as operational equivalence for applicative languages without side-effects, such as PCF. Two programs p and q are *contextually equivalent* iff for any context $\mathcal{C}$ such that $\emptyset \vdash \mathcal{C}[p] : \texttt{bool}$ and $\emptyset \vdash \mathcal{C}[q] : \texttt{bool}$, then $\mathcal{C}[p]$ converges just when $\mathcal{C}[q]$ does. This is also known as observational congruence. It is inappropriate for our calculus because (unlike in CCS, say) contexts cannot observe the side-effects of a program. In fact, any two programs which are ready to engage in I/O are contextually equivalent because neither immediately converges to a value.

Thus, in order to set up a useful operational semantics and notion of equivalence of programs, we must seek a framework which can subsume the usual semantics of applicative languages, but at the same time provide a mechanism for the semantics of side-effects. A suitable framework is a labelled transition system, with assertions of the form $p \xrightarrow{\alpha} q$ meaning that program p performs action α to become program q. Using an appropriate labelled transition system, CCS-style bisimilarity provides the natural operational equivalence on $\mathcal{O}$ programs. Theorem 1 is that bisimilarity is a congruence. It follows that if two programs are bisimilar, they are also contextually equivalent. This is what we would hope: it would be disconcerting if bisimilarity equated two programs that were contextually distinct.

Another candidate for operational equivalence is trace equivalence. If $s = \alpha_1 \dots \alpha_n$ is a finite sequence of actions, we say that s is a *trace* of p iff $p \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n}$. Two programs are *trace equivalent* iff they have the same set of traces. As noted elsewhere [Gor93, Mil89] in a deterministic calculus such as $\mathcal{O}$ trace equivalence coincides with bisimilarity; we prefer to use bisimilarity because it admits proofs by co-induction.

The denotational semantics is specified in two stages. First, we give a denotational semantics to a metalanguage $\mathcal{M}$ 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 the FIX-logic of [CP92], 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 metalanguage to study object languages [Plo85] we equip the programs (closed expressions) of $\mathcal{M}$ with an operational semantics. Theorem 2 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 equal, 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 [BMT92, Hol83] but few developed notions of operational equivalence. Holmström [Hol83] 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, as we have described, 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, where the actions correspond to the atomic observations one can make of an $\mathcal{O}$ program. The classical definition of (strong) bisimilarity from CCS [Mil89] generates a natural operational equivalence, which subsumes both Abramsky's applicative bisimulation [AO92] and the stratified equivalences suggested by Holmström's semantics [Gor94, Gor93]. 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 construction is based on recent work of Pitts [Pit94b] for untyped languages, and uses the idea of minimal invariant objects due to Freyd.

Finally, some comments about notation. As usual, we identify phrases of syntax up to α-conversion, that is, renaming of bound variables. We write $\phi \equiv \psi$ to mean that phrases ϕ and ψ are α-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*, but not identified up to α-conversion. 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, $\mathcal{R}^*$ its reflexive and transitive closure, and $\mathcal{R}^{\mathrm{op}}$ its opposite, that is, $\{(y,x) \mid (x,y) \in \mathcal{R}\}$.

2 The object language $\mathcal{O}$

In this section we define the (object) programming language $\mathcal{O}$. First we give the types and expressions of $\mathcal{O}$. Then we specify the programs and values, and use these to present a "single-step" operational semantics. Next we highlight certain $\mathcal{O}$ expressions which are able to engage in I/O; these are used to develop a labelled transition system semantics, in which some of the actions (labels) amount to I/O effects. This labelled transition system induces a notion of program bisimilarity, which will be good for program reasoning provided bisimilarity is a congruence. We say a relation between $\mathcal{O}$-expressions is a *precongruence* iff it is preserved by all contexts, and a *congruence* if in addition it is an equivalence relation. We prove bisimilarity is a congruence by introducing a relation on $\mathcal{O}$-expressions which is clearly a precongruence, and which can be shown equal to similarity; the result follows by showing that bisimilarity is the symmetric interior of similarity.

$\mathcal{O}$ 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` and `int`, together with function and product types; these types have the same intended meanings as in ML, and are specified by the grammar

$$\tau ::= \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \tau\,\texttt{->}\,\tau' \mid \tau\,\texttt{*}\,\tau'.$$

Let *Lit*, ranged over by ℓ, be the set $\{\texttt{true},\texttt{false}\} \cup \{\ldots,\texttt{-2},\texttt{-1},\texttt{0},\texttt{1},\texttt{2},\ldots\}$ of Boolean and integer *literals*, and let *Rator*, be the set $\{\texttt{+},\texttt{-},\texttt{*},\texttt{=},\texttt{<}\}$ of arithmetic *operators*. It will be convenient to use the notations $\underline{b}$ ($b \in \{tt, f\!f\}$), $\underline{i}$ ($i \in \mathbb{Z}$) and $\underline{\oplus}$ ($\oplus \in \{+,-,\times,=,<\}$) to range over the sets *Lit* and *Rator*. We let k range over the set of $\mathcal{O}$ *constants*, given by

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

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

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

where x ranges over a countable set of variables, and k^τ is an expression if and only if $k{:}\tau$ is an instance of one of the following type schemes:

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

The intended meanings of expressions are those which the reader expects. For the sake of simplicity there is just one user-definable constant, δ, which provides a recursive program declaration as described shortly. 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 finite set of (variable, type) pairs, $\{\, x{:}\tau_1, \ldots, x{:}\tau_n \,\}$, in which the variables are required to be distinct. In such judgements, e will be an α-equivalence class of expressions. 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^τ is a valid expression. We shall write $e{:}\tau$ instead of $\emptyset \vdash e : \tau$. We assume there is a user-specified expression e_δ, determining the behaviour of the constant δ, for which we assume that $x{:}\tau_\delta \vdash e_\delta : \tau'_\delta$ is provable. It would be routine to extend $\mathcal{O}$ to allow finitely many user-definable constants, but for the sake of simplicity we allow just one.

We shall define the notions of program and value for $\mathcal{O}$. A *program* is a closed expression e for which there is a type τ where $e{:}\tau$. Each program has a unique type, given the type annotations on constants and lambda-abstractions, though for notational convenience we often omit these annotations. The metavariables p and q will range over programs. 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*, ranged over by v, u or w, consists of the value expressions that are programs; so values are those programs which appear in the grammar

$$v ::= k^\tau \mid \lambda x.\, e \mid (v, v)$$

where k^τ must be a valid expression and k is not Ω.

In order to specify various operational semantics for $\mathcal{O}$, we shall make heavy use of relationships between programs of the same type; with this in mind we shall introduce some notation. We shall write $\mathcal{U}_\tau$ for the largest binary relation on programs of type τ, that is $\mathcal{U}_\tau \overset{\text{def}}{=} \{\, e \mid e{:}\tau \,\} \times \{\, e \mid e{:}\tau \,\}$, and $\mathcal{U}$ is then defined to be the union of these relations over all types: $\mathcal{U} \overset{\text{def}}{=} \bigcup_\tau \mathcal{U}_\tau$.

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 between programs, $\rightarrow$, by the rules in Table 1. It is a standard and easy exercise to verify that in fact $\rightarrow \subseteq \mathcal{U}$, that is, $\rightarrow$ preserves types in the expected way.

$$(\lambda x.\, e)\, v \to e[{}^{v}\!/\!_{x}] \qquad\qquad \underline{\oplus}\,(\underline{i},\underline{j}) \to \underline{i \oplus j}$$

$$\delta\, v \to e_\delta[{}^{v}\!/\!_{x}] \qquad\qquad \Omega \to \Omega$$

$$\mathtt{fst}\,(u,v) \to u \qquad\qquad \mathtt{snd}\,(u,v) \to v$$

$$\mathtt{if}\ \mathtt{true}\ \mathtt{then}\, p\, \mathtt{else}\, q \to p \qquad\qquad \mathtt{if}\ \mathtt{false}\ \mathtt{then}\, p\, \mathtt{else}\, q \to q$$

together with the inference rule

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

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

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

Table 1: Rules for Generating the → Relation

The rules for δ and Ω introduce the possibility of non-termination into $\mathcal{O}$: observe how δ yields a recursive program provided that δ appears within the expression e_δ. One can easily verify that the relation $\to$ is a partial function.

A *communicator* is, informally, a program ready to engage in I/O. The elements of the set *Com* of communicators is specified by the grammar

$$c ::= \mathtt{read\ ()} \mid \mathtt{write}\, \underline{i} \mid \mathcal{E}[c].$$

A communicator is essentially specified by a finite nesting of experiments with a `read ()` or `write` $\underline{i}$ at the innermost level. It is quite easy to see that the set of communicators is disjoint from the set of values. Let us define the set of *active programs*, *Active*, ranged over by a and b, to be the (disjoint) union of the communicators and the values. We can easily show that the active programs are the normal forms of $\to$, that is:

Lemma 2.1 *Active = Normal, where Normal* $= \{p \mid \neg\exists q(p \to q)\}$

Proof We can show that $\mathit{Active} \subseteq \mathit{Normal}$ by proving that any communicator, or value, is normal; and this follows by showing that the set of normal forms is closed under the rules for defining the sets of values and communicators.

That $\mathit{Normal} \subseteq \mathit{Active}$ follows by structural induction on expressions; more precisely, we prove that

$$e \in \mathit{Normal} \quad \text{implies} \quad e \in \mathit{Active}$$

holds for all expressions by structural induction. □

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

$$u \xrightarrow{@v} u\,v \text{ if } u\,v \text{ a program} \qquad \mathtt{read}\,() \xrightarrow{?n} \underline{n} \qquad \mathtt{write}\,n \xrightarrow{!n} ()$$

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

Table 2: Rules for Generating the Labelled Transition System

Our behavioural equivalence is based on a set of atomic observations, or *actions*, that may be observed of a program. In particular, there are actions associated with both read and write effects. We let *Msg*, ranged over by μ, be $\mathit{Msg} \stackrel{\text{def}}{=} \{\Gamma i, !i \mid i \in \mathbb{Z}\}$, where Γi represents input of a number i and $!i$ output of i. Thus *Msg*, a set of *messages*, represents I/O effects. The set of actions, ranged over by α, is given by

$$\mathit{Act} \stackrel{\text{def}}{=} \mathit{Lit} \cup \{\mathtt{fst}, \mathtt{snd}, @v \mid v \text{ is a value }\} \cup \mathit{Msg}.$$

The *labelled transition system* is a ternary relation whose relationships will be written $p \xrightarrow{\alpha} p'$ where p and p' are programs, and α is an action. The labelled transition system is inductively defined by the rules in Table 2.

The last rule allows messages—but not arbitrary actions—to be observed as side-effects of subterms. Each transition $p \xrightarrow{\alpha} q$ can be factored as a (finite) sequence of applicative reductions, down to an active program, followed by an α transition; this fact is highly important, and is made precise in the following lemma.

Lemma 2.2 *$p \xrightarrow{\alpha} q$ iff $\exists a \in \mathit{Active}\,(p \to^* a \xrightarrow{\alpha} q)$.*

Proof Given the existence of a factorisation of an action via an active program, $p \to^* a \xrightarrow{\alpha} q$, it is easy to see that $p \xrightarrow{\alpha} q$ by applying the rule

$$\frac{p \to p'' \qquad p'' \xrightarrow{\alpha} p'}{p \xrightarrow{\alpha} p'}\ (\star)$$

Conversely, we use rule induction on the set of labelled transitions. Let us give one example case: consider the rule

$$\frac{p \xrightarrow{\mu} q}{\mathcal{E}[p] \xrightarrow{\mu} \mathcal{E}[q]}$$

for any arbitrary experiment $\mathcal{E}$. By induction, there is an active program a such that $p \to^* a \xrightarrow{\mu} q$, and so we have $\mathcal{E}[p] \to^* \mathcal{E}[a] \xrightarrow{\mu} \mathcal{E}[q]$. By inspecting the definition of the labelled transition system, if a is a value $a \xrightarrow{\mu} q$ could only be deduced from $(\star)$, implying that there is some p'' for which $a \to p''$. But this is not possible by Lemma 2.1. Hence a is a communicator, implying that $\mathcal{E}[a]$ is also a communicator as required. We omit the verifications for the remaining rules. □

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

We adopt bisimilarity from Milner's CCS [Mil89] as our operational equivalence for $\mathcal{O}$. Let q be an *α-derivative* of p iff $p \xrightarrow{\alpha} q$. We want two programs p and q to be behaviourally equivalent iff, for every action α, every α-derivative of p is behaviourally equivalent to some α-derivative of q, and vice versa. We shall assume that the reader is familiar with these ideas, at least in the setting of concurrency theory and process calculi. However, it will be convenient to give a terse summary of the notions of (bi)simulations, presented within our own framework.

Given a relation $\mathcal{S} \subseteq \mathcal{U}$ we define $[\mathcal{S}] \subseteq \mathcal{U}$ by deeming that $p[\mathcal{S}]q$ iff whenever $p \xrightarrow{\alpha} p'$, there is q' with $q \xrightarrow{\alpha} q'$ and $p' \mathcal{S} q'$. Note that this is well defined; that $[\mathcal{S}]$ really is a subset of $\mathcal{U}$ follows by inspecting the definition of the labelled transition system. We can define functions $\Phi_s, \Phi_b : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ where $\Phi_s(\mathcal{S}) \stackrel{\text{def}}{=} [\mathcal{S}]$ and $\Phi_b(\mathcal{S}) \stackrel{\text{def}}{=} [\mathcal{S}] \cap [\mathcal{S}^{\text{op}}]^{\text{op}}$. One can check that these functions are well-defined and monotone. We say that $\mathcal{S}$ is a *simulation* if $\mathcal{S} \subseteq \Phi_s(\mathcal{S})$ and that $\mathcal{S}$ is a *bisimulation* if $\mathcal{S} \subseteq \Phi_b(\mathcal{S})$.

We define *similarity*, $\lesssim \subseteq \mathcal{U}$, to be the greatest (post-)fixed point of Φ_s,

$$\lesssim \stackrel{\text{def}}{=} \nu(\Phi_s),$$

and *bisimilarity* to be the greatest (post-)fixed point of Φ_b,

$$\sim \stackrel{\text{def}}{=} \nu(\Phi_b).$$

If $p \lesssim q$ we say that p is *similar* to q, and if $p \sim q$ we say that p is *bisimilar* to q. We shall soon see that similarity is a preorder and that bisimilarity is an equivalence. It is immediate that (bi)similarity is the greatest (bi)simulation; in fact appealing to the (proof of the) Knaster-Tarski theorem we have

$$\begin{array}{rcl} \lesssim & = & \bigcup\{\, \mathcal{S} \mid \mathcal{S} \subseteq \Phi_s(\mathcal{S}) \,\} \\ \sim & = & \bigcup\{\, \mathcal{S} \mid \mathcal{S} \subseteq \Phi_b(\mathcal{S}) \,\} \end{array}$$

The following principles of co-induction are corollaries of the definitions of $\lesssim$ and $\sim$.

Lemma 2.3 *$p \lesssim q$ iff there is a simulation $\mathcal{S}$ with $p \mathcal{S} q$; and $p \sim q$ 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 metalanguage $\mathcal{M}$ 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, Theorem 1, which we shall prove via an adaptation of Howe's method; similar proofs can be found elsewhere [Gor94, Gor95a, How89].

The proof of this result is rather lengthy, involving a number of intermediate steps and definitions. We begin by observing that in order to prove Theorem 2.10 we may deal simply with similarity, rather than bisimilarity.

Lemma 2.4 *Bisimilarity is the symmetric interior of similarity, that is* $\sim \; = \; \lesssim \cap \lesssim^{\mathrm{op}}$.

Proof The proof depends on the easily verified fact that our labelled transition system is image singular in the sense that

$$\text{whenever } p \xrightarrow{\alpha} p' \text{ and } p \xrightarrow{\alpha} p'' \text{ then } p' \equiv p''.$$

Since $\sim \; = \Phi_b(\sim) = [\sim] \cap [\sim^{\mathrm{op}}]^{\mathrm{op}}$ we have $\sim \; \subseteq [\sim]$ and $\sim^{\mathrm{op}} \subseteq [\sim^{\mathrm{op}}]$. By co-induction $\sim \; \subseteq \; \lesssim$ and $\sim^{\mathrm{op}} \subseteq \; \lesssim$, hence $\sim \; \subseteq \; \lesssim \cap \lesssim^{\mathrm{op}}$. For the reverse inclusion it suffices to show that $\lesssim \cap \lesssim^{\mathrm{op}}$ is a simulation (as any symmetric simulation is a bisimulation). Consider p and q such that $p \lesssim q$ and $q \lesssim p$. Suppose that $p \xrightarrow{\alpha} p'$. From $p \lesssim q$ there must be a q' such that $q \xrightarrow{\alpha} q'$ and $p' \lesssim q'$. We need to show $q' \lesssim p'$ also. Since $q \xrightarrow{\alpha} q'$ there must be a p'' with $p \xrightarrow{\alpha} p''$ and $q' \lesssim p''$. But by the fact above, it must be that $p' \equiv p''$, so we are done. □

This lemma fails in a nondeterministic calculus such as CCS, where the labelled transition system is not image singular. Now, in order to prove Theorem 2.10, all we need do is show that $\lesssim$ is a precongruence; let us introduce some technical machinery in order to prove this.

We have given a definition of $\lesssim \; \subseteq \mathcal{U}$. This gives relationships between programs (of the same type). We will now extend the definition of $\lesssim$ to provide relationships between expressions. The restriction of this relation to programs will amount to similarity, so we denote it also by $\lesssim$. We define a relation $\lesssim$, with relationships denoted by $\Gamma \vdash e \lesssim e' : \tau$, and for which it will be implicit (by definition) that both e and e' are assigned the type τ in the environment Γ. We define $\Gamma \vdash e \lesssim e' : \tau$ iff

- $\Gamma \vdash e : \tau$,
- $\Gamma \vdash e' : \tau$ and
- if $\Gamma = \{\, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \,\}$, then for all finite sets of values $\{\, v_1{:}\tau_1, \ldots, v_n{:}\tau_n \,\}$ we have
$$e[\vec{v}/\vec{x}] \lesssim e'[\vec{v}/\vec{x}]$$
where here $\lesssim$ is similarity of programs as defined above.

Let us now define a relation $\lesssim^{\bullet}$, analogous in form to $\lesssim$, using the rules in Table 3. We call $\lesssim^{\bullet}$ *Howe's* relation. We have a lemma which gives some basic properties of Howe's relation, and (bi)similarity.

Lemma 2.5 (1) $\lesssim$ *is a preorder and* $\sim$ *is an equivalence;*

(2) *If* $p \lesssim q$ *and* $p \rightarrow^{*} p'$ *then* $p' \lesssim q$*;*

(3) *If* $\Gamma \vdash e_1 \lesssim^{\bullet} e_2 : \tau$ *and* $\Gamma \vdash e_2 \lesssim e_3 : \tau$ *then* $\Gamma \vdash e_1 \lesssim^{\bullet} e_3 : \tau$*;*

(4) $\Gamma \vdash e : \tau$ *implies* $\Gamma \vdash e \lesssim^{\bullet} e : \tau$*;*

$$\frac{}{\Gamma, x{:}\sigma \vdash x \lesssim^{\bullet} e : \tau} \quad (\Gamma, x{:}\sigma \vdash x \lesssim e : \tau)$$

$$\frac{}{\Gamma \vdash k \lesssim^{\bullet} e : \tau} \quad (\Gamma \vdash k \lesssim e : \tau)$$

$$\frac{\Gamma, x{:}\sigma \vdash e_1 \lesssim^{\bullet} e_2 : \tau}{\Gamma \vdash \lambda x{:}\sigma.\, e_1 \lesssim^{\bullet} e_3 : \sigma \texttt{ -> } \tau} \quad (\Gamma \vdash \lambda x{:}\sigma.\, e_2 \lesssim e_3 : \sigma \texttt{ -> } \tau)$$

$$\frac{\Gamma \vdash e_1 \lesssim^{\bullet} e_1' : \sigma \texttt{ -> } \tau \quad \Gamma \vdash e_2 \lesssim^{\bullet} e_2' : \sigma}{\Gamma \vdash e_1 e_2 \lesssim^{\bullet} e_3 : \tau} \quad (\Gamma \vdash e_1' e_2' \lesssim e_3 : \tau)$$

$$\frac{\Gamma \vdash e_1 \lesssim^{\bullet} e_1' : \sigma \quad \Gamma \vdash e_2 \lesssim^{\bullet} e_2' : \tau}{\Gamma \vdash \texttt{(}e_1, e_2\texttt{)} \lesssim^{\bullet} e_3 : \sigma \texttt{ * } \tau} \quad (\Gamma \vdash \texttt{(}e_1', e_2'\texttt{)} \lesssim e_3 : \sigma \texttt{ * } \tau)$$

$$\frac{\Gamma \vdash e_1 \lesssim^{\bullet} e_1' : \texttt{bool} \quad \Gamma \vdash e_2 \lesssim^{\bullet} e_2' : \tau \quad \Gamma \vdash e_3 \lesssim^{\bullet} e_3' : \tau}{\Gamma \vdash \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \lesssim^{\bullet} e_4 : \tau} \, (\Gamma \vdash \texttt{if}\, e_1' \,\texttt{then}\, e_2' \,\texttt{else}\, e_3' \lesssim^{\bullet} e_4 : \tau)$$

Table 3: Rules for Generating the Relation $\lesssim^{\bullet}$

(5) $\Gamma \vdash e_1 \lesssim e_2 : \tau$ *implies* $\Gamma \vdash e_1 \lesssim^{\bullet} e_2 : \tau$*;*

(6) *If* $\Gamma \vdash e \lesssim^{\bullet} e' : \sigma$ *and* $\Gamma, x{:}\sigma \vdash e_1 \lesssim^{\bullet} e_2 : \tau$ *then* $\Gamma \vdash e_1[{}^{e}\!/\!{x}] \lesssim^{\bullet} e_2[{}^{e'}\!/\!{x}] : \tau$*; and*

(7) $\lesssim^{\bullet}$ *is a precongruence.*

Proof

(1) It is easy to see that if $I \stackrel{\text{def}}{=} \{\, (p, p) \;\mid\; p \text{ is a program} \,\}$ then ($I \subseteq \mathcal{U}$ and) $I \subseteq \Phi_s(I)$. Thus $I \subseteq \lesssim$ implying that $\lesssim$ is reflexive. One can also prove routinely that for any simulation $\mathcal{S}$ we have $[\mathcal{S}] \circ [\mathcal{S}] \subseteq [\mathcal{S} \circ \mathcal{S}]$; that $\lesssim \,=\, [\lesssim]$ implies that $\lesssim \circ \lesssim \;\subseteq\; \lesssim$. So $\lesssim$ is a preorder and it is thus immediate from Lemma 2.4 that $\sim$ is an equivalence.

(2) This is immediate from the definition of $\lesssim$ plus rule ($\star$) of Table 2.

(3) Use induction on the derivation of $\Gamma \vdash e_1 \lesssim^{\bullet} e_2 : \tau$. One needs to appeal to the transitivity of $\lesssim$, proved in (1).

(4) Use induction on the derivation of $\Gamma \vdash e : \tau$.

(5) This follows from parts (3) and (4).

(6) Use induction on the derivation of $\Gamma, x{:}\sigma \vdash e_1 \lesssim^{\bullet} e_2 : \tau$, together with part (5).

(7) This follows from the definition of $\lesssim^{\bullet}$, plus part (4).

□

Lemma 2.6 *Whenever $v \lesssim^\bullet q$, there exists a value u for which*

$$v \lesssim^\bullet u \qquad \textit{and} \qquad q \to^* u.$$

In particular, if v is $\underline{l}$ then $q \to^* \underline{l}$.

Proof We use induction on the structure of value v.

(*Case v is $\lambda x{:}\sigma.\, e_1$*): Suppose that $\lambda x{:}\sigma.\, e_1 \lesssim^\bullet q$, where $\lambda x{:}\sigma.\, e_1 {:} \sigma$ `->` τ, say. Then there is an expression e_2 for which $x{:}\sigma \vdash e_1 \lesssim^\bullet e_2 : \tau$ and $\lambda x{:}\sigma.\, e_2 \lesssim q$. Each type σ is inhabited, in particular there is $v{:}\sigma$ for each σ. Thus we have

$$\begin{tikzcd}
\lambda x{:}\sigma.\, e_2 \arrow[r, phantom, "\lesssim"] \arrow[d, "@v"'] & q \arrow[d, dotted, "@v"] \\
(\lambda x{:}\sigma.\, e_2)v \arrow[r, phantom, "\lesssim"] & q'
\end{tikzcd}$$

and so appealing to Lemma 2.2 there is a value u for which $q \to^* u \xrightarrow{@v} uv = q'$; the definition of the labelled transition system ensures that u is indeed a value. Note that the only transitions $\lambda x{:}\sigma.\, e_2$ can make are of the form $@v$ for some $v{:}\sigma$; it follows that $\lambda x{:}\sigma.\, e_2 \lesssim u$ and hence, using Lemma 2.5 part (3), that $\lambda x{:}\sigma.\, e_1 \lesssim^\bullet u$.

(*Case v is (v, v')*): Suppose that $(v, v') \lesssim^\bullet q$. Then there are q_1 and q_1' for which $v \lesssim^\bullet q_1$, $v' \lesssim^\bullet q_1'$ and $(q_1, q_1') \lesssim q$. By induction there exist values u_1 and u_1' where $q_1 \to^* u_1$ and $q_1' \to^* u_1'$ and such that $v \lesssim^\bullet u_1$ and $v' \lesssim^\bullet u_1'$. Thus using Lemma 2.2 we have

$$\begin{tikzcd}
(q_1, q_1') \arrow[r, phantom, "\lesssim"] \arrow[d, "\mathtt{fst}"'] & q \arrow[d, dotted, "\mathtt{fst}"] & & (q_1, q_1') \arrow[r, phantom, "\lesssim"] \arrow[d, "\mathtt{snd}"'] & q \arrow[d, dotted, "\mathtt{snd}"] \\
u_1 \arrow[r, phantom, "\lesssim"] & w & & u_2 \arrow[r, phantom, "\lesssim"] & w'
\end{tikzcd}$$

and so $q \to^* (w, w')$ where $u_1 \lesssim w$ and $u_1' \lesssim w'$. It follows that $(v, v') \lesssim^\bullet (w, w')$.

The remaining cases consist of the value constants: each case is quite similar, relying on Lemma 2.2. We give just two examples:

(*Case v is $\underline{l}$*): If $\underline{l} \lesssim^\bullet p$ then $\underline{l} \lesssim p$ and thus there is a program p' for which $p \xrightarrow{l} p'$. Hence we must have $p \to^* a \xrightarrow{l} \Omega = p'$ and hence $a = \underline{l}$ because a has to be a value of type `int`.

(*Case v is* `read`): Suppose that $\texttt{read} \lesssim^\bullet p$, so that

$$\begin{tikzcd}
\texttt{read} \arrow[d, "\texttt{@()}"'] & \lesssim & p \arrow[d, dotted, "\texttt{@()}"] \\
\texttt{read()} & \lesssim^\bullet & p'
\end{tikzcd}$$

Using Lemma 2.2 there is an active a for which $p \rightarrow^* a \xrightarrow{\texttt{@()}} a\texttt{()}$. Thus a must be a value, with $\texttt{read} \lesssim^\bullet a$.

□

Lemma 2.7 *Whenever $p \rightarrow p'$ and $p \lesssim^\bullet q$, then $p' \lesssim^\bullet q$.*

Proof We induct on the derivation of $p \rightarrow p'$.

(*Case* $\texttt{fst}(u,v) \rightarrow u$): Suppose that we have $\texttt{fst}(u,v) \lesssim^\bullet q$. Then appealing to Lemma 2.6, there are programs p_1, p_2 and values v_1, v_2 and v_2' for which

- $\texttt{fst} \lesssim^\bullet p_1 \rightarrow^* v_1$ where $\texttt{fst} \lesssim^\bullet v_1$;
- $(u,v) \lesssim^\bullet p_2 \rightarrow^* (v_2,v_2')$ where $(u,v) \lesssim^\bullet (v_2,v_2')$ and
- $\texttt{fst}(u,v) \lesssim^\bullet p_1 p_2 \lesssim q$.

It follows that

$$\begin{tikzcd}
 & & \texttt{fst} \arrow[d, "\texttt{@}(v_2,v_2')"'] & \lesssim & v_1 \arrow[d, "\texttt{@}(v_2,v_2')"] & & p_1p_2 \arrow[d, "*"] & \lesssim & q \\
 & & \texttt{fst}(v_2,v_2') \arrow[d] & \lesssim & v_1(v_2,v_2') & = & v_1(v_2,v_2') & & \\
u & \lesssim^\bullet & v_2 & & & & & &
\end{tikzcd}$$

and so $u \lesssim^\bullet q$ as required by Lemma 2.5 parts (1), (2) and (3).

(*Case* $\texttt{snd}(u,v) \rightarrow v$): symmetric to the previous case.

(*Case* $(\lambda x{:}\sigma.\, e)v \rightarrow e[{}^{v}\!/\!{}_{x}]$): Suppose that $(\lambda x{:}\sigma.\, e)v \lesssim^\bullet q$. Then using Lemma 2.6, there are p_1, p_2 and v_2 such that

- $\lambda x{:}\sigma.\, e \lesssim^\bullet p_1$;
- $v \lesssim^\bullet p_2 \rightarrow^* v_2$ where $v \lesssim^\bullet v_2$ and

- $p_1 p_2 \lesssim q$.

Thus there is e' such that $x{:}\sigma \vdash e \lesssim^{\bullet} e'$ and $\lambda x{:}\sigma.\, e' \lesssim p_1$. So from Lemma 2.2 and from Lemma 2.5 part (6) we have

$$\begin{tikzcd}
 & & \lambda x{:}\sigma.\, e' \arrow[d, swap, "@v_2"] & \lesssim & p_1 \arrow[d, "@v_2"] & & p_1 p_2 \arrow[d, "*"] & \lesssim q \\
 & & (\lambda x{:}\sigma.\, e')v_2 \arrow[d] & \lesssim & v_1 v_2 \arrow[rr, equal] & & v_1 v_2 & \\
e[{}^{v}\!/_{x}] & \lesssim^{\bullet} & e'[{}^{v_2}\!/_{x}] & & & & &
\end{tikzcd}$$

and so $e[{}^{v}\!/_{x}] \lesssim^{\bullet} q$ by Lemma 2.5 parts (1), (2) and (3).

(*Case* $\delta v \to e_\delta[{}^{v}\!/_{x}]$)**:** Suppose that $\delta v \lesssim^{\bullet} q$. Thus using Lemma 2.6, there are p_1, p_2, v_1 and v_2 such that

- $\delta \lesssim^{\bullet} p_1 \to^{*} v_1$ where $\delta \lesssim^{\bullet} v_1$;
- $v \lesssim^{\bullet} p_2 \to^{*} v_2$ with $v \lesssim^{\bullet} v_2$ and
- $p_1 p_2 \lesssim q$.

Hence by Lemma 2.5 part (6)

$$\begin{tikzcd}
 & & \delta \arrow[d, swap, "@v_2"] & \lesssim & v_1 \arrow[d, "@v_2"] & & p_1 p_2 \arrow[d, "*"] & \lesssim q \\
 & & \delta v_2 \arrow[d] & \lesssim & v_1 v_2 \arrow[rr, equal] & & v_1 v_2 & \\
e_\delta[{}^{v}\!/_{x}] & \lesssim^{\bullet} & e_\delta[{}^{v_2}\!/_{x}] & & & & &
\end{tikzcd}$$

and so $e_\delta[{}^{v}\!/_{x}] \lesssim q$ by Lemma 2.5 parts (1), (2) and (3).

□

Proposition 2.8 $\lesssim^{\bullet}$ is a simulation.

Proof We have to verify that if $p \lesssim^{\bullet} q$ and $p \xrightarrow{\alpha} p'$ then there exists q' where $q \xrightarrow{\alpha} q'$ and $p' \lesssim^{\bullet} q'$. We induct on the derivation of the labelled transitions.

(*Case* $\underline{l} \xrightarrow{l} \Omega$): This is immediate because we have $\underline{l} \lesssim q$ whenever $\underline{l} \lesssim^\bullet q$.

(*Case* $(u, v) \xrightarrow{\mathtt{fst}} u$): There exist values u' and v' for which

$$\begin{tikzcd}
(u, v) \arrow[r, phantom, "\lesssim^\bullet"] \arrow[d, "\mathtt{fst}"'] & (u', v') \arrow[r, phantom, "\lesssim"] \arrow[d, "\mathtt{fst}"] & q \arrow[d, dotted, "\mathtt{fst}"] \\
u \arrow[r, phantom, "\lesssim^\bullet"] & u' \arrow[r, phantom, "\lesssim"] & q'
\end{tikzcd}$$

and so from Lemma 2.5 part (3) we deduce $u \lesssim^\bullet q'$.

(*Case* $(u, v) \xrightarrow{\mathtt{snd}} u$): Similar to `fst`.

(*Case* $u \xrightarrow{@v} uv$): Let $u \lesssim^\bullet q$. Then $q \to^* w$ with $u \lesssim^\bullet w$ for some w, using Lemma 2.6. Thus u and w have the same type. Note that from Lemma 2.5 part (4) we have $v \lesssim^\bullet v$ and so appealing to Lemma 2.2 we have

$$\begin{tikzcd}
u \arrow[r, phantom, "\lesssim^\bullet"] \arrow[d, "@v"'] & q \arrow[d, "@v"] \\
uv \arrow[r, phantom, "\lesssim^\bullet"] & wv
\end{tikzcd}$$

(*Case* $\mathtt{write}\ \underline{i} \xrightarrow{!i} \mathtt{()}$): Suppose that $\mathtt{write}\ \underline{i} \lesssim^\bullet q$. Then using Lemma 2.6, there are p_1, p_2 and v_1 such that

- $\mathtt{write} \lesssim^\bullet p_1 \to^* v_1$ where $\mathtt{write} \lesssim^\bullet v_1$;
- $\underline{i} \lesssim^\bullet p_2 \to^* \underline{i}$ and
- $p_1 p_2 \lesssim q$.

Thus we have

$$\begin{tikzcd}
\mathtt{write} \arrow[r, phantom, "\lesssim"] \arrow[d, "@\underline{i}"'] & v_1 \arrow[d, "@\underline{i}"] & p_1 p_2 \arrow[r, phantom, "\lesssim"] \arrow[d, "*"] & q \arrow[dd, "!i"] \\
\mathtt{write}\ \underline{i} \arrow[r, phantom, "\lesssim"] \arrow[d, "!i"'] & v_1 \underline{i} \arrow[r, equal] \arrow[d, "!i"] & v_1 \underline{i} & \\
\mathtt{()} \arrow[r, phantom, "\lesssim"] & p_3 \arrow[rr, phantom, "\lesssim"] & & q'
\end{tikzcd}$$

and so $\mathtt{()} \lesssim^\bullet q'$ by Lemma 2.5 part (5).

(*Case* `read ()` $\xrightarrow{?i} \underline{i}$): Similar to the previous case.

(*Case* $\dfrac{p \rightarrow p'' \qquad p'' \xrightarrow{\alpha} p'}{p \xrightarrow{\alpha} p'}\ (\star)$): This is immediate from Lemma 2.7.

(*Case* $\dfrac{p \xrightarrow{\mu} q}{\mathcal{E}[p] \xrightarrow{\mu} \mathcal{E}[q]}$): We need to consider the various experiments. Let us consider $(v, [\,])$. Let $(v,p) \xrightarrow{\mu} (v,p')$ and $(v,p) \lesssim^{\bullet} q$. Hence (as usual) we have p_1, p_2 and v_1 such that

- $v \lesssim^{\bullet} p_1 \rightarrow^{*} v_1$ where $v \lesssim^{\bullet} v_1$;
- $p \lesssim^{\bullet} p_2$ and
- $(p_1, p_2) \lesssim q$.

By induction,

$$\begin{tikzcd}
p \arrow[r, phantom, "\lesssim^{\bullet}"] \arrow[d, "\mu"'] & p_2 \arrow[d, "\mu"] \\
p' \arrow[r, phantom, "\lesssim^{\bullet}"] & p_2'
\end{tikzcd}$$

Putting everything together and using Lemma 2.2 we get

$$\begin{tikzcd}
(v,p) \arrow[r, phantom, "\lesssim^{\bullet}"] \arrow[d, "\mu"'] & (p_1,p_2) \arrow[r, phantom, "\lesssim"] \arrow[d, "\mu" description] & q \arrow[d, dotted, "\mu"] \\
(v,p') \arrow[r, phantom, "\lesssim^{\bullet}"] & (v_1,p_2') \arrow[r, phantom, "\lesssim"] & q'
\end{tikzcd}$$

and thus $(v,p') \lesssim^{\bullet} q'$ (as usual!).

□

Proposition 2.9 $\lesssim$ is a precongruence.

Proof First, note that $\lesssim \subseteq \lesssim^{\bullet}$ follows from Lemma 2.5 part (5). Proposition 2.8 shows that $\lesssim^{\bullet}$ is a simulation, and thus $\lesssim^{\bullet} \subseteq \lesssim$. Hence $\lesssim = \lesssim^{\bullet}$, and appealing to Lemma 2.5 part (7) we see that $\lesssim$ is a precongruence. □

Thus we can now prove the central theorem of this section:

Theorem 2.10 Bisimilarity is a congruence.

Proof This follows from Proposition 2.9 and Lemma 2.4. □

3 The Metalanguage $\mathcal{M}$ and its Computational Adequacy

In this section we begin by defining the metalanguage $\mathcal{M}$, describing its types, expressions, proved expressions, and its operational theory. Theorem 2 is stated, asserting a computational adequacy result for $\mathcal{M}$. Next we outline some categorical methods which will be used to give a denotational semantics to $\mathcal{M}$. These methods have their origins in Scott's work on models of the lambda-calculus, and also adapt the results of Freyd and Pitts on minimal invariant objects. Next we specify the denotational semantics, which is essentially quite standard—types are modelled by complete pointed partial orders, and proved expressions by Scott continuous functions. We prove that certain formal approximation relations exist using the properties of minimal invariant objects. Finally, we prove Theorem 2 using the formal approximation relations.

We outline a Martin-Löf style type theory which will be used as a metalanguage, $\mathcal{M}$, into which $\mathcal{O}$ may be translated and reasoned about—it is based on ideas from the FIX-Logic [CP92, Cro92], though $\mathcal{M}$ does not explicitly contain a fixpoint type. For a general account of similar type theories and their semantics, see for example [Cro93].

First we describe the types of $\mathcal{M}$. The (open and simple) types are given by the grammar

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

where X_0 is a fixed type variable, together with a *single* top-level recursive datatype declaration

$$\mathsf{datatype}\,\mathsf{U}(X_0) = c_1 \text{ of } \sigma_1 \mid \cdots \mid c_a \text{ of } \sigma_a$$

where $a > 0$ and 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). Note that the positive integer a is fixed, as are each of the types σ_i. However, a and the types σ_i are essentially arbitrary, and have in fact specified a family of type systems—in Section 4, we shall choose a specific type-system in which the recursive datatype $\mathsf{U}(X_0)$ is used to model I/O.

Informally, the (open) 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 (finite) disjoint sum of (a) instances of the latter types. These types will be used in the expected way when modelling the object types of $\mathcal{O}$.

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, noting that there are no type variable binding operations, and indeed just one type variable. We shall make use of type substitution, and will write $\sigma(\sigma')$ for $\sigma[\sigma'/X_0]$, where the latter has the obvious definition.

The collection of expressions of $\mathcal{M}$ is given by the grammar in Table 4. Most of the syntax of $\mathcal{M}$ is standard [Cro92, CG93]. 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

$$
\begin{array}{rcll}
E & ::= & x & \text{(variable)} \\
  & \mid & () & \text{(unit value)} \\
  & \mid & \lfloor \ell \rfloor & \text{(literal value)} \\
  & \mid & E \lfloor \oplus \rfloor E & \text{(arithmetic)} \\
  & \mid & \mathsf{If}\, E \,\mathsf{then}\, E \,\mathsf{else}\, E & \text{(conditional)} \\
  & \mid & (E, E) & \text{(pair)} \\
  & \mid & \mathsf{Split}\, E \,\mathsf{as}\, (x, y) \,\mathsf{in}\, E & \text{(projection)} \\
  & \mid & c(E) & \text{(recursive data)} \\
  & \mid & \mathsf{Case}\, E \,\mathsf{of}\, c_1(x) \to E \mid \cdots \mid c_a(x) \to E & \text{(case analysis)} \\
  & \mid & \lambda x{:}\sigma.\, E & \text{(abstraction)} \\
  & \mid & E\, E & \text{(application)} \\
  & \mid & \mathsf{Lift}(E) & \text{(lifted value)} \\
  & \mid & \mathsf{Drop}\, E \,\mathsf{to}\, x \,\mathsf{in}\, E & \text{(sequential composition)} \\
  & \mid & \mathsf{Rec}\, x{:}\sigma \,\mathsf{in}\, E & \text{(recursion)}
\end{array}
$$

$$\ell \in \mathbb{B} \cup \mathbb{Z} \qquad c \in \{\, c_1, \ldots, c_a \,\}$$

Table 4: Expressions of the Metalanguage $\mathcal{M}$, ranged over by E

lifting computational monad [Mog89]. The expression $\mathsf{Split}\, E_1 \,\mathsf{as}\, (x, y) \,\mathsf{in}\, E_2$ is the usual one for decomposing binary product expressions. Further details can be found in [NPS90].

We define a type assignment system for $\mathcal{M}$ which consists of rules for generating judgements of the form $\Gamma \vdash E{:}\sigma$, where σ is a closed type, and the *environment* Γ is a finite set $\{\, x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \,\}$ of (variable, closed type) pairs in which the variables are required to be distinct. In such judgements, which we call *proved expressions*, E is formally an α-equivalence class of expressions (the latter defined in Table 4). Usual scope rules apply. Most of the rules for generating these judgements are fairly standard, though for completeness they are given in Table 5. The type of an arithmetic expression is lifted so that its value can be forced using Drop. In the case that the environment Γ is empty, we shall write $E{:}\sigma$ for the type assignment.

We can equip $\mathcal{M}$ with a standard equational theory, which includes β, η and congruence rules. The judgements take the form $\Gamma \vdash E = E'{:}\sigma$, which we call *theorems*. Having given the full set of rules for type assignment, we omit the rules for deriving theorems. In the case that the environment Γ is empty, we shall write a theorem as $E = E'{:}\sigma$, or even $E = E'$ if no confusion is likely to occur.

An $\mathcal{M}$ *program* is a closed expression P for which there exists a (closed) type σ where $P{:}\sigma$. The set of $\mathcal{M}$ *value expressions* is given by the grammar

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

and *values* are those V which are programs.

Finally, we equip the syntax of $\mathcal{M}$ with an operational semantics. This is specified by 'small-step' reduction relations which take the form $P_1 \to P_2$. The rules for

$$\frac{}{\Gamma, x{:}\sigma, \Gamma' \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash () : \mathsf{Unit}} \qquad \frac{\ell \in \mathbb{B}}{\Gamma \vdash \lfloor \ell \rfloor : \mathsf{Bool}} \qquad \frac{\ell \in \mathbb{Z}}{\Gamma \vdash \lfloor \ell \rfloor : \mathsf{Int}}$$

$$\frac{\Gamma \vdash E_1 : \mathsf{Int} \qquad \Gamma \vdash E_2 : \mathsf{Int}}{\Gamma \vdash E_1 \lfloor \oplus \rfloor E_2 : \mathsf{Int}_\bot} \qquad \frac{\Gamma \vdash E_1 : \mathsf{Bool} \qquad \Gamma \vdash E_2 : \sigma \qquad \Gamma \vdash E_3 : \sigma}{\Gamma \vdash \mathsf{If}\, E_1 \,\mathsf{then}\, E_2 \,\mathsf{else}\, E_3 : \sigma}$$

$$\frac{\Gamma \vdash E_i : \sigma_i \quad (i = 1, 2)}{\Gamma \vdash (E_1, E_2) : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash E_1 : \sigma_1 \times \sigma_2 \qquad \Gamma, x_1{:}\sigma_1, x_2{:}\sigma_2 \vdash E_2 : \sigma}{\Gamma \vdash \mathsf{Split}\, E_1 \,\mathsf{as}\, (x_1, x_2) \,\mathsf{in}\, E_2 : \sigma}$$

$$\frac{\Gamma \vdash E : \sigma_i[{}^{\sigma}\!/X_0]}{\Gamma \vdash c_i(E) : \mathsf{U}(\sigma)}\, 1 \leq i \leq a \qquad \frac{\Gamma \vdash E : \mathsf{U}(\sigma) \qquad \Gamma, x{:}\sigma_i[{}^{\sigma}\!/X_0] \vdash E_i : \sigma'}{\Gamma \vdash \mathsf{Case}\, E \,\mathsf{of}\, c_1(x) \to E_1 \mid \cdots \mid c_a(x) \to E_a : \sigma'}$$

$$\frac{\Gamma, x{:}\sigma' \vdash E : \sigma}{\Gamma \vdash (\lambda x{:}\sigma'.\, E) : \sigma' \to \sigma} \qquad \frac{\Gamma \vdash E : \sigma' \to \sigma \qquad \Gamma \vdash E' : \sigma'}{\Gamma \vdash E\, E' : \sigma}$$

$$\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash \mathsf{Lift}(E) : \sigma_\bot} \qquad \frac{\Gamma \vdash E : \sigma_\bot \qquad \Gamma, x{:}\sigma \vdash E' : \sigma'}{\Gamma \vdash \mathsf{Drop}\, E \,\mathsf{to}\, x \,\mathsf{in}\, E' : \sigma'} \qquad \frac{\Gamma, x{:}\sigma_\bot \vdash E : \sigma_\bot}{\Gamma \vdash \mathsf{Rec}\, x{:}\sigma_\bot \,\mathsf{in}\, E : \sigma_\bot}$$

Table 5: Generation of proved expresssions in $\mathcal{M}$

generating the operational semantics appear in Table 6. The operational semantics of $\mathcal{M}$ is lazy in the sense that constructors do not evaluate their arguments.

Given any program P, we write $P\Downarrow$ to mean that there is a value V for which $P \to^+ V$. Note that $\mathcal{M}$ is deterministic: every program P which reduces to some value V, must reduce to a unique value V up to α-equivalence.

In the rest of this section, our aim is to construct a domain-theoretic denotational semantics for $\mathcal{M}$, assigning a denotation $[\![P]\!]$ to each program P, and to prove the following theorem.

Theorem 3.1

(1) If P is a program of type σ and $P \to P'$, then P' is also a program of type σ and moreover $[\![P]\!] = [\![P']\!] \in [\![\sigma]\!]$.

(2) If P is a program of type $\sigma_\bot$ and $[\![P]\!] \neq -$ then there exists a value V of type $\sigma_\bot$ and $P \to^* V$.

(3) The denotational semantics is sound for the equational theory of $\mathcal{M}$, that is if $P = P'$ is a theorem, then $[\![P]\!] = [\![P']\!]$.

Part (1) is soundness of the operational semantics of $\mathcal{M}$: it preserves denotation. Part (2) is adequacy: if a program does not denote $-$ then its evaluation converges. In Section 4, we obtain a denotational semantics for $\mathcal{O}$ indirectly via a textual

$$\begin{array}{rcl}
\lfloor \ell_1 \rfloor \lfloor \oplus \rfloor \lfloor \ell_2 \rfloor & \to & \lfloor \ell_1 \oplus \ell_2 \rfloor \\
\mathsf{If}\ \underline{tt}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 & \to & P_1 \\
\mathsf{If}\ \underline{ff}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 & \to & P_2 \\
\mathsf{Split}\ (P_1, P_2)\ \mathsf{as}\ (x, y)\ \mathsf{in}\ E & \to & E[{}^{P_1, P_2}/_{x, y}] \\
\mathsf{Case}\ c_i(P)\ \mathsf{of}\ c_1(x_1) \to E_1 \mid \ldots \mid c_a(x_a) \to E_a & \to & E_i[{}^{P}/_{x_i}] \\
(\lambda x{:}\sigma.\, E)\ P & \to & E[{}^{P}/_{x}] \\
\mathsf{Drop}\ \mathsf{Lift}(P)\ \mathsf{to}\ x\ \mathsf{in}\ E & \to & E[{}^{P}/_{x}] \\
\mathsf{Rec}\ x\ \mathsf{in}\ E \to E[{}^{\mathsf{Rec}\ x\ \mathsf{in}\ E}/_{x}] & &
\end{array}$$

together with the inference rule

$$\frac{P_1 \to P_2}{\mathcal{E}[P_1] \to \mathcal{E}[P_2]}$$

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

$$\begin{array}{rcl}
\mathcal{E} & ::= & [\,] \lfloor \oplus \rfloor P \\
 & \mid & V \lfloor \oplus \rfloor [\,] \\
 & \mid & \mathsf{If}\ [\,]\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \\
 & \mid & \mathsf{Split}\ [\,]\ \mathsf{as}\ (x, y)\ \mathsf{in}\ E \\
 & \mid & \mathsf{Case}\ [\,]\ \mathsf{of}\ c_1(x_0) \to E_1 \mid \ldots \mid c_a(x_0) \to E_a \\
 & \mid & [\,]\ P \\
 & \mid & V\ [\,] \\
 & \mid & \mathsf{Lift}([\,]) \\
 & \mid & \mathsf{Drop}\ [\,]\ \mathsf{to}\ x\ \mathsf{in}\ E
\end{array}$$

Table 6: The reduction relation for $\mathcal{M}$

translation into into $\mathcal{M}$. We need Theorem 2 to show that if the induced denotations of two $\mathcal{O}$ programs are equal, then the two programs are bisimilar. The rest of this section is devoted to the proof of Theorem 2.

Ultimately, we shall 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 expressions by Scott continuous functions. However, in order to set up our denotational semantics, we shall make use of some domain-theoretic constructions in other categories. The following categories will be employed:

- The category $\mathcal{CPO}$ with objects all cpos and morphisms all continuous functions;

- the category $\mathcal{CPPO}$ with objects all cppos and morphisms all continuous functions;

- the category $\mathcal{D}om$ with objects all cppos and morphisms all strict continuous functions; and

- the category $\mathcal{D}om^{\mathcal{T}}$ where $\mathcal{T}$ is any set and we define

  $$\mathcal{D}om^{\mathcal{T}} \stackrel{\text{def}}{=} \Pi_{\sigma\in\mathcal{T}}\mathcal{D}om$$

  to be the $\mathcal{T}$-indexed product category. For the time being $\mathcal{T}$ can be any set; but later it will be the set of all closed types in $\mathcal{M}$, and we shall use σ to denote elements of $\mathcal{T}$. We write $(A^\sigma \mid \sigma\in\mathcal{T})$ for an object of $\mathcal{D}om^{\mathcal{T}}$, and recall that by definition, hom-sets in $\mathcal{D}om^{\mathcal{T}}$ are given by

  $$\mathcal{D}om^{\mathcal{T}}(A, B) \stackrel{\text{def}}{=} \Pi_{\sigma\in\mathcal{T}}\mathcal{D}om(A^\sigma, B^\sigma).$$

We shall make use of the following inclusion diagram

$$\begin{tikzcd} \mathcal{D}om \arrow[r, "incl"] & \mathcal{CPPO} \arrow[r, "incl"] & \mathcal{CPO} \end{tikzcd}$$

where the first inclusion yields a lluf subcategory, and the second a full subcategory. We write $- : \mathcal{CPO} \to \mathcal{CPO}$ for the (functor part of the) lifting monad, which maps any cpo X to the lifted cppo

$$X_\perp \stackrel{\text{def}}{=} \{\,[x] \mid x\in X\,\} \cup \{\,-\,\}.$$

Note that each of the above categories is a $\mathcal{CPO}$-enriched category. As usual, the hom-sets of the first two categories, whose elements are continuous functions, are given the pointwise order. The hom-sets of $\mathcal{D}om^{\mathcal{T}}$ are products of c(p)pos—hence cpos. We shall write $-_A \stackrel{\text{def}}{=} (-_{A^\sigma} \mid \sigma \in \mathcal{T})$ for the bottom element of A in $\mathcal{D}om^{\mathcal{T}}$. While we shall only make use of $\mathcal{CPO}$-enrichment, note that each hom-set

$\mathcal{D}om^{\mathcal{T}}(A,B)$ is a pointed cpo; we write $-_{A,B} \overset{\text{def}}{=} (-_{A^\sigma,B^\sigma} \mid \sigma \in \mathcal{T})$ for the bottom of $\mathcal{D}om^{\mathcal{T}}(A,B)$ where $-_{A^\sigma,B^\sigma} \in \mathcal{D}om(A^\sigma, B^\sigma)$ is the function with constant value $-_{B^\sigma}$.

The terminal object $1 \in \mathcal{D}om^{\mathcal{T}}$ is $(\{\,-\,\} \mid \sigma \in \mathcal{T})$. Finally, if $f{:}A \to B$ in $\mathcal{D}om^{\mathcal{T}}$ and $a \in \Pi_{\sigma\in\mathcal{T}} A^\sigma$ then we define $f(a) \in \Pi_{\sigma\in\mathcal{T}} B^\sigma$ by $f(a)^\sigma \overset{\text{def}}{=} f^\sigma(a^\sigma)$.

Our wish is to give a semantics to $\mathcal{M}$ in $\mathcal{CPPO}$, using functions which are not necessarily strict to model proved expressions because $\mathcal{M}$ is a lazy type theory. However, while the semantics is specified in $\mathcal{CPPO}$, we wish to exploit the "minimal invariant" properties associated with the lluf subcategory $\mathcal{D}om$ of $\mathcal{CPPO}$.

Note that $(\mathcal{D}om^{\mathcal{T}})^{op} \times \mathcal{D}om^{\mathcal{T}}$ is a $\mathcal{CPO}$-category. Let

$$F : (\mathcal{D}om^{\mathcal{T}})^{op} \times \mathcal{D}om^{\mathcal{T}} \to \mathcal{D}om^{\mathcal{T}}$$

be a $\mathcal{CPO}$-functor. A *(parametrised) minimal invariant* for F is given by an object D of $\mathcal{D}om^{\mathcal{T}}$, and an isomorphism $i : F(D,D) \cong D : j$ in $\mathcal{D}om^{\mathcal{T}}$ for which the (continuous) function

$$\delta : \mathcal{D}om^{\mathcal{T}}(D,D) \to \mathcal{D}om^{\mathcal{T}}(D,D) \qquad e \mapsto i \circ F(e,e) \circ j$$

satisfies $\mu(\delta) = id_D$ in $\mathcal{D}om^{\mathcal{T}}(D,D)$. The reader can verify that δ is continuous—this follows from the facts that F is a $\mathcal{CPO}$-functor and that each i^σ and j^σ are continuous.

Proposition 3.2 Any $\mathcal{CPO}$-functor $F : (\mathcal{D}om^{\mathcal{T}})^{op} \times \mathcal{D}om^{\mathcal{T}} \to \mathcal{D}om^{\mathcal{T}}$ has a minimal invariant.

Proof The essence of the proof boils down to Scott's original construction of a model for lambda-calculus [Sco69]. We shall sketch out the important constructions in the proof, and leave detailed verifications to the reader.

For each $n \in \mathbb{N}$ there is a commutative diagram in $\mathcal{D}om^{\mathcal{T}}$ of the form

$$\begin{tikzcd}
D_{n+1} \arrow[rr, shift left, "F(p_n, e_n)"] \arrow[d, equal] \arrow[drr, phantom, "(*)"] & & F(D,D) \arrow[ll, shift left, "F(e_n, p_n)"] \arrow[d, equal] \\
D_{n+1} \arrow[r, shift left, "e_{n+1}"] \arrow[d, shift left, "r_n"] & D \arrow[l, shift left, "p_{n+1}"] \arrow[r, shift left, "j"] & F(D,D) \arrow[l, shift left, "i"] \\
D_n \arrow[u, shift left, "i_n"] \arrow[r, shift left, "e_n"] & D \arrow[l, shift left, "p_n"] \arrow[r, shift left, "j"] & F(D,D) \arrow[l, shift left, "i"]
\end{tikzcd}$$

Let us give the definitions of the objects and morphisms in this diagram:
(*Definition of D in $\mathcal{D}om^{\mathcal{T}}$*) Set

$$\begin{aligned} D_0 &\overset{\text{def}}{=} 1 \in \mathcal{D}om^{\mathcal{T}} \\ D_{n+1} &\overset{\text{def}}{=} F(D_n, D_n) \in \mathcal{D}om^{\mathcal{T}} \end{aligned}$$

for each $n \in \mathbb{N}$. Define morphisms

$$i_n : D_n \to D_{n+1} \qquad\qquad \text{and} \qquad\qquad r_n : D_{n+1} \to D_n$$

by

$$\begin{aligned} i_0 &\overset{\text{def}}{=} -_{D_0,D_1} \\ r_0 &\overset{\text{def}}{=} -_{D_1,D_0} \\ i_{n+1} &\overset{\text{def}}{=} F(r_n, i_n) \\ r_{n+1} &\overset{\text{def}}{=} F(i_n, r_n). \end{aligned}$$

Now define

$$D^\sigma \overset{\text{def}}{=} \{\, (d_n^\sigma \mid n < \omega) \in \Pi_{n<\omega} D_n^\sigma \;\mid\; r_n^\sigma(d_{n+1}^\sigma) = d_n^\sigma \,\}$$

for each $\sigma \in \mathcal{T}$. Order each D^σ pointwise, and note that D^σ is a cppo because each r_n^σ is strict continuous; here, $-_{D^\sigma} = (-_{D_n^\sigma} \mid n < \omega)$. Hence we define $D \overset{\text{def}}{=} (D^\sigma \mid \sigma \in \mathcal{T})$, an object of $\mathcal{D}om^{\mathcal{T}}$.

(*Definition of* e_n) We set $e_n \overset{\text{def}}{=} (e_n^\sigma \mid \sigma \in \mathcal{T})$ where

$$e_n^\sigma : D_n^\sigma \to D^\sigma \qquad d_n^\sigma \mapsto (e_n^\sigma(d_n^\sigma)_m \mid m < \omega)$$

and

$$e_n^\sigma(d_n^\sigma)_m \overset{\text{def}}{=} \begin{cases} r_{n,m}^\sigma(d_n^\sigma) & \text{if} \quad m < n \\ x^\sigma & \text{if} \quad m = n \\ i_{n,m}^\sigma(d_n^\sigma) & \text{if} \quad m > n \end{cases}$$

Here, if $m < n$, then $r_{n,m} \overset{\text{def}}{=} r_m \circ \ldots \circ r_n$ and $i_{m,n} \overset{\text{def}}{=} i_n \circ \ldots \circ i_m$. It is easy to verify that each e_n^σ is indeed a strict continuous function.

(*Definition of* p_n) These are the (strict continuous) projections, with p_n^σ mapping $(d_n^\sigma \mid n < \omega)$ to d_n^σ.

(*Definition of* i) We set $i = (i^\sigma \mid \sigma \in \mathcal{T})$ where

$$i^\sigma : F(D,D)^\sigma \to D^\sigma \qquad\qquad i^\sigma \overset{\text{def}}{=} \bigvee_{n<\omega} e_{n+1}^\sigma \circ F(e_n, p_n)^\sigma$$

(*Definition of* j) We set $j = (j^\sigma \mid \sigma \in \mathcal{T})$ where

$$j^\sigma : D^\sigma \to F(D,D)^\sigma \qquad\qquad j^\sigma \overset{\text{def}}{=} \bigvee_{n<\omega} F(p_n, e_n)^\sigma \circ p_{n+1}^\sigma$$

One can show that these definitions yield commutative diagrams of the form given on page 20, and that each (e_n^σ, p_n^σ) is an embedding-projection pair in $\mathcal{D}om$. Using the square $(*)$ we can prove that $e_n \circ p_n = \delta^n(-_{D,D})$ for each $n < \omega$, and thus

$$\mu(\delta) = \bigvee_{n<\omega} \delta^n(-_{D,D}) = \bigvee_{n<\omega} e_n \circ p_n = id_D$$

with the final equality following from the basic properties of embedding-projection pairs. □

Let us now assign a denotational semantics to the closed types of $\mathcal{M}$ where we write $\mathcal{T}$ for the set of all closed types. We shall first define a $\mathcal{T}$-indexed family of functors

$$(\, F_\sigma{:}(\mathcal{D}om^{\mathcal{T}})^{op} \times \mathcal{D}om^{\mathcal{T}} \to \mathcal{D}om^{\mathcal{T}} \mid \sigma \in \mathcal{T} \,)$$

through the following clauses:

- $F_{\mathsf{Unit}}(A,B) \overset{\text{def}}{=} \{\, 0 \,\}_\bot$;
- $F_{\mathsf{Bool}}(A,B) \overset{\text{def}}{=} \mathbb{B}_\bot$;
- $F_{\mathsf{Int}}(A,B) \overset{\text{def}}{=} \mathbb{Z}_\bot$;
- $F_{\sigma\times\sigma'}(A,B) \overset{\text{def}}{=} F_\sigma(A,B) \times F_{\sigma'}(A,B)$;
- $F_{\sigma\Rightarrow\sigma'}(A,B) \overset{\text{def}}{=} F_\sigma(B,A) \Rightarrow F_{\sigma'}(A,B)$;
- $F_{\sigma_\bot}(A,B) \overset{\text{def}}{=} F_\sigma(A,B)_\bot$; and
- $F_{\mathsf{U}(\sigma)}(A,B) \overset{\text{def}}{=} B^\sigma$,

where at base types, $F_\sigma(-,+)$ maps morphisms to identity morphisms. Note that $\times$ and $\Rightarrow$ are the product and exponential functors in $\mathcal{CPO}$, restricted to the category $\mathcal{D}om$. We also define a functor

$$F : (\mathcal{D}om^{\mathcal{T}})^{op} \times \mathcal{D}om^{\mathcal{T}} \longrightarrow \mathcal{D}om^{\mathcal{T}}$$

by setting

$$F(A,B) \overset{\text{def}}{=} (LS(F_{\sigma_1(\sigma)}(A,B), \ldots, F_{\sigma_a(\sigma)}(A,B)) \mid \sigma{\in}\mathcal{T})$$

where $LS(-) : \mathcal{D}om^a \to \mathcal{D}om$ is the functor given by

$$\mathcal{D}om^a \xrightarrow{incl} \mathcal{CPO}^a \xrightarrow{+} \mathcal{CPO} \xrightarrow{\bot} \mathcal{CPO} \xrightarrow{incl} \mathcal{D}om$$

with $+$ being coproduct (of a objects) and $-$ being the lifting monad on $\mathcal{CPO}$. In general, we write

$$in_j : A_j \longrightarrow A_1 + \ldots + A_a$$

for coproduct insertion. Note that this is a sensible definition of F as σ is a closed type, and thus so is each $\sigma_i(\sigma)$. We leave the verification that the functors F_σ and F are indeed $\mathcal{CPO}$-functors to the reader.

Appealing to Proposition 3.2 there is a minimal invariant D for F, equipped with an isomorphism $i : F(D,D) \to D$ in $\mathcal{D}om^{\mathcal{T}}$. We define

$$[\![\sigma]\!] \overset{\text{def}}{=} F_\sigma(D,D)$$

for each $\sigma \in \mathcal{T}$. Note the following consequences of this definition:

- $[\![\mathsf{Unit}]\!] = \{\, 0 \,\}_\bot$;

- $\llbracket \mathsf{Bool} \rrbracket = \mathbb{B}_\bot$;
- $\llbracket \mathsf{Int} \rrbracket = \mathbb{Z}_\bot$;
- $\llbracket \sigma \times \sigma' \rrbracket = \llbracket \sigma \rrbracket \times \llbracket \sigma' \rrbracket$;
- $\llbracket \sigma \to \sigma' \rrbracket = \llbracket \sigma \rrbracket \Rightarrow \llbracket \sigma' \rrbracket$;
- $\llbracket \sigma_\bot \rrbracket = \llbracket \sigma \rrbracket_\bot$; and
- $\llbracket \mathsf{U}(\sigma) \rrbracket = F_{\mathsf{U}(\sigma)}(D, D) = D^\sigma$.

Note that in $\mathcal{D}om$ we have

$$\begin{tikzcd}
F(D,D)^\sigma \arrow[d, "i^\sigma"'] & = & \ldots & = & LS(\llbracket \sigma_1(\sigma) \rrbracket, \ldots, \llbracket \sigma_a(\sigma) \rrbracket) \arrow[d, "i^\sigma"] \\
D^\sigma & = & \ldots & = & \llbracket \mathsf{U}(\sigma) \rrbracket
\end{tikzcd}$$

Given an environment Γ we define $\llbracket \Gamma \rrbracket$ to be the cppo which is the product of the denotations of the types appearing in Γ, and we then specify a continuous function $\llbracket \Gamma \vdash E{:}\sigma \rrbracket{:}\llbracket \Gamma \rrbracket \to \llbracket \sigma \rrbracket$ for each proved expression. Note that if Γ is empty, we define $\llbracket \Gamma \rrbracket \overset{\text{def}}{=} \{-\}$, any one-point cppo. The definition of these semantic functions is quite standard; we simply give the meaning of expressions associated with functions, recursion and cases:

- If $e \overset{\text{def}}{=} \llbracket \Gamma, x{:}\sigma \vdash E{:}\sigma' \rrbracket : (\llbracket \Gamma \rrbracket \times \llbracket \sigma \rrbracket) \to \llbracket \sigma' \rrbracket$ and $\xi \in \llbracket \Gamma \rrbracket$, then we set
$$\llbracket \Gamma \vdash \lambda x.\, E{:}\sigma \Rightarrow \sigma' \rrbracket(\xi) \overset{\text{def}}{=} \lambda x \in \llbracket \sigma \rrbracket . e(\xi, x) : \llbracket \sigma \rrbracket \to \llbracket \sigma' \rrbracket.$$
- If $\llbracket \Gamma, x{:}\sigma_\bot \vdash E{:}\sigma_\bot \rrbracket : (\llbracket \Gamma \rrbracket \times \llbracket \sigma_\bot \rrbracket) \to \llbracket \sigma_\bot \rrbracket$, and $\lambda(e)$ denotes exponential transpose (currying), then
$$\llbracket \Gamma \vdash \mathsf{Rec}\, x \,\mathsf{in}\, E{:}\sigma_\bot \rrbracket(\xi) \overset{\text{def}}{=} \bigvee_{n<\omega} \lambda(e)(\xi)^n(-_{\llbracket \sigma_\bot \rrbracket}).$$
- If $e \overset{\text{def}}{=} \llbracket \Gamma \vdash E{:}\sigma_j(\sigma) \rrbracket : \llbracket \Gamma \rrbracket \to \llbracket \sigma_j(\sigma) \rrbracket$, and $\xi \in \llbracket \Gamma \rrbracket$, then we shall set
$$\llbracket \Gamma \vdash c_j(E){:}\mathsf{U}(\sigma) \rrbracket(\xi) \overset{\text{def}}{=} i^\sigma([in_j(e(\xi))]) \in \llbracket \mathsf{U}(\sigma) \rrbracket$$
- If $e \overset{\text{def}}{=} \llbracket \Gamma \vdash E{:}\mathsf{U}(\sigma) \rrbracket{:}\llbracket \Gamma \rrbracket \to \llbracket \mathsf{U}(\sigma) \rrbracket$ and
$$e_j \overset{\text{def}}{=} \llbracket \Gamma, x_j{:}\sigma_j(\sigma) \vdash E_j{:}\sigma' \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket \sigma_j(\sigma) \rrbracket \to \llbracket \sigma' \rrbracket,$$
then
$$\llbracket \Gamma \vdash \mathsf{Case}\, E \,\mathsf{of}\, c_1(x_1) \to E_1 \mid \ldots \mid c_n(x_n) \to E_n{:}\sigma' \rrbracket(\xi)$$
$$\overset{\text{def}}{=} \begin{cases} e_j(\xi, -) \text{ if } j^\sigma(e(\xi)) = - \\ e_j(\xi, d_j^\sigma) \text{ if } j^\sigma(e(\xi)) = [in_j(d_j^\sigma)] \end{cases}$$

We finish this section by noting that we have set up some machinery which caters for the possibility that the body of the recursive datatype contains a contravariant type variable. However, in our application to I/O, there is no such contravariance. We could slightly simplify both this section and the next by restricting attention to such recursive types; however, the simplification is not particularly significant. Furthermore, the present formulation of $\mathcal{M}$ makes it suitable for other applications, such as a denotational semantics of a language with a store, where such contravariance is essential.

In this section we introduce some simple category theory that will play a key role in the proof of Theorem 3.1. We shall show that there is a $\mathcal{T}$ indexed family of relations

$$(\ \lhd_\sigma \subseteq [\![\sigma]\!] \times \{\, P \ \mid\ \exists\sigma(P{:}\sigma)\,\} \mid \sigma \in \mathcal{T}\ )$$

satisfying certain conditions. Such formal approximation relations are fairly standard (see for example [CG93], [Pit94b] and [Plo85]) so we simply give these conditions at function, lifted and recursive types:

- $f \lhd_{\sigma\Rightarrow\sigma'} P$ iff $\quad f = -$ or $\exists E.\ P \rightarrow^* \lambda x.\, E$ and $\forall d \lhd_\sigma P'.\ f(d) \lhd_{\sigma'} E[{}^{P'}\!/x]$,
- $e \lhd_{\sigma_\bot} P$ iff $\quad \exists d \in [\![\sigma]\!].\ e = [d]$ implies $\exists P'.\ P \rightarrow^* \mathsf{Lift}(P')$ and $d \lhd_\sigma P'$,
- $r^\sigma \lhd_{\mathsf{U}(\sigma)} P$ iff $\quad r^\sigma = -_{D^\sigma}$ or
  $\exists P_j.\ P \rightarrow^* c_j(P_j)$ and $\exists d_j^\sigma \in [\![\sigma_j(\sigma)]\!].\ r^\sigma = i^\sigma([in_j(d_j^\sigma)])$ and $d_j^\sigma \lhd_{\sigma_j(\sigma)} P_j$.

Proposition 3.3 There exists a family of formal approximation relations

$$(\lhd_\sigma \mid \sigma \in \mathcal{T})$$

enjoying the above properties.

The existence of the formal approximation relations can be proved by techniques which appear in Plotkin's CSLI notes [Plo85]. However, it is more elegant to adapt Pitts' method of admissible actions on relational structures. We give an outline of the method. Set $\mathit{TyProgs} \overset{\text{def}}{=} \{P{:}\sigma \mid\ P \text{ is a program of type } \sigma\}$, regard the set $\mathit{TyProgs}$ as a discrete cpo, and for any cppo X put

$$\mathcal{R}(X) \overset{\text{def}}{=} \quad \{R \in \mathcal{P}(X \times \mathit{TyProgs}) \mid R \text{ is an } \omega\text{-chain complete subset}\}.$$

We define $\mathcal{R}(A) \overset{\text{def}}{=} \Pi_{\sigma\in\mathcal{T}}(A^\sigma)$ where A is an object of $\mathcal{D}om^{\mathcal{T}}$. We shall use the letters R and S to range over elements of both $\mathcal{R}(A)$ and $\mathcal{R}(X)$. In the former case, R^σ will denote the σ-th component of R.

Lemma 3.4 *Both $\mathcal{R}(X)$ and $\mathcal{R}(A)$, where X is an object of $\mathcal{D}om$ and A is an object of $\mathcal{D}om^{\mathcal{T}}$, are complete lattices.*

Proof Note that $\mathcal{R}(X)$ is a complete lattice with the inclusion order, where arbitrary meets are given by set-theoretic intersection. It follows that $\mathcal{R}(A)$ is a complete lattice with the product ordering. □

Let D be the minimal invariant of F defined on page 22, and for each $\sigma \in \mathcal{T}$ we shall define a monotone function

$$F_\sigma : \mathcal{R}(D)^{op} \times \mathcal{R}(D) \longrightarrow \mathcal{R}(\llbracket \sigma \rrbracket)$$

through the following clauses:

- $F_{\mathsf{Unit}}(R,S) \stackrel{\text{def}}{=} \{\, (d, P{:}\mathsf{Unit}) \mid d = - \quad \text{or} \quad (d = [0] \quad \mathit{and} \quad P \to^* ()) \,\};$
- $F_{\mathsf{Bool}}(R,S) \stackrel{\text{def}}{=} \{\, (d, P{:}\mathsf{Bool}) \mid d = - \quad \text{or}$ $(d = [0] \quad \mathit{and} \quad P \to^* \lfloor tt \rfloor) \quad \text{or} \quad (d = [1] \quad \mathit{and} \quad P \to^* \lfloor \mathit{ff} \rfloor) \,\};$
- $F_{\mathsf{Int}}(R,S) \stackrel{\text{def}}{=} \{\, (d, P{:}\mathsf{Int}) \mid d = - \quad \text{or} \quad (\exists z \in \mathbb{Z}.\, d = [z] \quad \mathit{and} \quad P \to^* \lfloor z \rfloor) \,\};$
- $F_{\sigma \times \sigma'}(R,S) \stackrel{\text{def}}{=} \{\, (p, P_1{:}\sigma \times \sigma') \mid p = - \quad \text{or}$ $(\exists (d,d') \in \llbracket \sigma \rrbracket \times \llbracket \sigma' \rrbracket.\, p = (d,d') \quad \mathit{and} \quad \exists P, P'.\, P \to^* (P, P') \quad \mathit{and}$ $(d, P{:}\sigma) \in F_\sigma(R,S) \quad \mathit{and} \quad (d', P'{:}\sigma') \in F_{\sigma'}(R,S)) \,\};$
- $F_{\sigma \to \sigma'}(R,S) \stackrel{\text{def}}{=} \{\, (f, P{:}\sigma \to \sigma') \mid f = - \text{ or } (\exists E'.\, P \to^* \lambda x.\, E'$ $\mathit{and} \quad \forall (d, P{:}\sigma) \in F_\sigma(S,R).\, (f(d), E'[{}^{P}\!/\!x]{:}\sigma') \in F_{\sigma'}(R,S)) \,\};$
- $F_{\sigma_\bot} \stackrel{\text{def}}{=} \{\, (e, P{:}\sigma_\bot) \mid e = - \quad \text{or} \quad (\exists d \in \llbracket \sigma \rrbracket.\, e = [d] \quad \mathit{and}$ $\exists P'.\, P \to^* \mathsf{Lift}(P') \quad \mathit{and} \quad (d, P'{:}\sigma) \in F_\sigma(R,S)) \,\};$
- $F_{\mathsf{U}(\sigma)}(R,S) \stackrel{\text{def}}{=} S^\sigma$ where of course $S^\sigma \in \mathcal{R}(D^\sigma) = \mathcal{R}(\llbracket \mathsf{U}(\sigma) \rrbracket)$.

We leave the reader to verify that $(F_\sigma \mid \sigma \in \mathcal{T})$ is a family of monotone functions.

Next we define a monotone function

$$F : \mathcal{R}(D)^{op} \times \mathcal{R}(D) \longrightarrow \mathcal{R}(F(D,D))$$

by setting its components to be

$$\begin{aligned} F(R,S)^\sigma \stackrel{\text{def}}{=} \{\, (x^\sigma, P{:}\mathsf{U}(\sigma)) \mid x^\sigma = -_{F(D,D)^\sigma} \quad \text{or} \\ \exists P_j.\, P \to^* c_j(P_j) \quad \mathit{and} \quad \exists d^\sigma_j \in \llbracket \sigma_j(\sigma) \rrbracket.\, x^\sigma = [in_j(d^\sigma_j)] \quad \mathit{and} \\ (d^\sigma_j, P_j{:}\sigma_j(\sigma)) \in F_{\sigma_j(\sigma)}(R,S) \,\}. \end{aligned}$$

We also define a monotone function

$$L : \mathcal{R}(D)^{op} \times \mathcal{R}(D) \longrightarrow \mathcal{R}(D)$$

by setting

$$\begin{aligned} L(R,S) \stackrel{\text{def}}{=} \{\, (u^\sigma, P{:}\tau) \mid u^\sigma = -_{D^\sigma} \quad \text{or} \\ \exists x^\sigma \in F(D,D)^\sigma.\, u^\sigma = i^\sigma(x^\sigma) \quad \mathit{and} \quad (x^\sigma, P{:}\tau) \in F(R,S)^\sigma \,\} \end{aligned}$$

Define

$$L^{sym} : \mathcal{R}(D)^{op} \times \mathcal{R}(D) \longrightarrow \mathcal{R}(D)^{op} \times \mathcal{R}(D)$$

by setting $L^{sym}(R,S) \stackrel{\text{def}}{=} (L(S,R), L(R,S))$. Note that using Lemma 3.4 we can deduce that $\mathcal{R}(D)^{op} \times \mathcal{R}(D)$ is a complete lattice, and hence by Knaster-Tarski there is an element $(R_\perp, R_+) \in \mathcal{R}(D)^{op} \times \mathcal{R}(D)$ which is the least fixed point of L^{sym}. It follows from the equality

$$(R_\perp, R_+) = (L(R_+, R_\perp), L(R_\perp, R_+))$$

and leastness of $(R_\perp, R_+)$, that $L^{sym}(R_+, R_\perp) \leq (R_+, R_\perp)$ and hence that

$$R_+ \leq R_\perp \qquad (\star)$$

in the lattice $\mathcal{R}(D)$.

We shall now set out to prove that $R_\perp \leq R_+$. This will involve some further machinery. We shall write $e : R \preceq S$ to mean

- $e \in \mathcal{D}om^{\mathcal{T}}(D,D)$;
- $R \in \mathcal{R}(D)$ and $S \in \mathcal{R}(D)$; and
- for every $(u^\sigma, P{:}\tau) \in R^\sigma$ we have $(e^\sigma(u^\sigma), P{:}\tau) \in S^\sigma$.

Lemma 3.5 *If $e : R \preceq S$ then*

$$(d, P{:}\tau) \in F_\tau(S,R) \quad \textit{implies} \quad (F_\tau(e,e)(d), P{:}\tau) \in F_\tau(R,S).$$

Proof The result follows from a simple induction on the closed type τ. We consider one simple case:

(*Case τ is* $\mathsf{U}(\sigma)$) Let $(r^\sigma, P{:}\mathsf{U}(\sigma)) \in F_{\mathsf{U}(\sigma)}(S,R) = R^\sigma$. Recall that $F_{\mathsf{U}(\sigma)}(e,e)^\sigma = e^\sigma$. We have $(e^\sigma(r^\sigma), P{:}\mathsf{U}(\sigma)) \in S^\sigma = F_{\mathsf{U}(\sigma)}(R,S)$ and so we are done. □

Lemma 3.6 *Whenever $e : R \preceq S$ we have $\delta(e) : L(S,R) \preceq L(R,S)$.*

Proof Suppose that $(u^\sigma, P{:}\tau) \in L(S,R)^\sigma$. We wish to show that

$$(\delta(e)^\sigma(u^\sigma), P{:}\tau) \in L(R,S)^\sigma.$$

If $\delta(e)^\sigma(u^\sigma) = -_{D^\sigma}$ we are done. If not, we know that u^σ is non-bottom, and thus there is a non-bottom $x^\sigma \in F(D,D)^\sigma$ for which

$$\delta(e)^\sigma(u^\sigma) = i^\sigma \circ F(e,e)^\sigma(x^\sigma).$$

Thus it remains to show that

$$(F(e,e)^\sigma(x^\sigma), P{:}\tau) \in F(R,S)^\sigma \qquad (\dagger)$$

By induction, it follows that $(x^\sigma, P{:}\tau) \in F(R,S)^\sigma$ and hence τ must be of the form $\mathsf{U}(\sigma)$, $P \to^* c_j(P_j)\ldots(1)$ and $x^\sigma = [in_j(d_j^\sigma)]$ where $(d_j^\sigma, P_j{:}\sigma_j(\sigma)) \in F_{\sigma_j(\sigma)}(S,R)$. Using Lemma 3.5, we have

$$(F_{\sigma_j(\sigma)}(e,e)(d_j^\sigma), P_j{:}\sigma_j(\sigma)) \in F_{\sigma_j(\sigma)}(R,S)\ldots(2)$$

Thus (†) will follow from (1) and (2) using the following computation:

$$\begin{array}{rcl} F(e,e)^\sigma(x^\sigma) & = & LS((F_{\sigma_1(\sigma)}(e,e),\ldots,F_{\sigma_a(\sigma)}(e,e)))([in_j(d_j^\sigma)]) \\ & = & [(F_{\sigma_1(\sigma)}(e,e)+\ldots+F_{\sigma_a(\sigma)}(e,e))(in_j(d_j^\sigma))] \\ & = & [in_j(F_{\sigma_j(\sigma)}(e,e)(d_j^\sigma))] \end{array}$$

where we have used the definition of $F(e,e)$, naturality of the unit of the lifting monad, and the universal property of coproducts. □

Let

$$Z \stackrel{\text{def}}{=} \{\, e \in \mathcal{D}om^{\mathcal{T}}(D,D) \mid e : R_\bot \preceq R_+ \,\}.$$

Using Lemma 3.6 we see that for any $e \in Z$ we have $\delta(e) \in Z$. Also, $-_{D,D} \in Z$, for if $(d^\sigma, P{:}\tau) \in R_\bot^\sigma$ then $(-_{D^\sigma}, P{:}\tau) \in L(R_\bot, R_+)^\sigma = R_+^\sigma$. One can check that Z is ω-chain complete, and hence it follows that

$$id_D = \mu(\delta) = \bigvee_{n<\omega} \delta^n(-_{D,D}) \in Z.$$

Hence $id_D : R_\bot \preceq R_+$, that is $R_\bot^\sigma \subseteq R_+^\sigma$ for each $\sigma \in \mathcal{T}$, which amounts to $R_\bot \leq R_+$ in $\mathcal{R}(D)$. Recalling assertion $(\star)$ from page 26, we can set $R_{\mathit{fix}} \stackrel{\text{def}}{=} R_\bot = R_+$ and finally

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

Thus we have proved the existence of the required family of formal approximation relations—it is very easy to see that the required properties hold.

We shall need the following lemmas:

Lemma 3.7 *Suppose that $P{:}\sigma$, $P \to^* P'$ and $d \lhd_\sigma P'$. Whenever we have these data, $d \lhd_\sigma P$.*

Proof The proof is a simple induction on the structure of σ. □

Lemma 3.8 *Whenever $y_1{:}\sigma_1,\ldots,y_m{:}\sigma_m \vdash E{:}\sigma$ and $(d_k \lhd_{\sigma_k} P_k \mid 1 \leq k \leq m)$ then*

$$[\![\Gamma \vdash E{:}\sigma]\!](\vec{d}) \lhd_\sigma E[\vec{P}/\vec{y}].$$

Proof The proof proceeds by induction on the structure of the expression E.

(*Case E is* $\mathsf{Rec}\,x\,\mathsf{in}\,E$) We have to prove that

$$[\![\Gamma \vdash \mathsf{Rec}\,x\,\mathsf{in}\,E{:}\sigma_\bot]\!](\vec{d}) \lhd_{\sigma_\bot} P,$$

where $P \overset{\text{def}}{=} \mathsf{Rec}\, x \,\mathsf{in}\, E[\vec{P}/\vec{y}]$. Suppose that $[\![\Gamma \vdash \mathsf{Rec}\, x \,\mathsf{in}\, E{:}\sigma_\bot]\!](\vec{d}) \neq -_{[\![\sigma_\bot]\!]}$, say

$$[\![\Gamma \vdash \mathsf{Rec}\, x \,\mathsf{in}\, E{:}\sigma_\bot]\!](\vec{d}) = [a] \in [\![\sigma_\bot]\!].$$

It remains to prove that $P \twoheadrightarrow^* \mathsf{Lift}(P')$ for some $P'{:}\sigma$, and that $a \lhd_\sigma P'$.

We have $- \lhd_{\sigma_\bot} P$. From this, we can use induction on $\mathbb{N}$ to show that $\lambda(e)(\vec{d})^n(-) \lhd_{\sigma_\bot} P[{}^{\mathsf{Rec}\, x \,\mathsf{in}\, E}/x]$ holds for all $n \in \mathbb{N}$. Note also that the domain elements indexed by n form an ω-chain in $[\![\sigma_\bot]\!]$. Appealing to the definition of the denotational semantics we see that

$$[a] = \bigvee_{n<\omega} \lambda(e)(\vec{d})^n(-)$$

and so there is $n_0 \in \mathbb{N}$, and $a_m \in [\![\sigma]\!]$ for every $m \geq n_0$, with $\lambda(e)(\vec{d})^m(-) = a_m$. Therefore $[a_m] \lhd_{\sigma_\bot} P[{}^{\mathsf{Rec}\, x \,\mathsf{in}\, E}/x]$, implying that $P[{}^{\mathsf{Rec}\, x \,\mathsf{in}\, E}/x] \twoheadrightarrow^* \mathsf{Lift}(P')$ for some $P'{:}\sigma$ and that $a_m \lhd_\sigma P'$ for all $m \geq n_0$. Note that we make crucial use of the determinacy of $\twoheadrightarrow^*$ here—each a_m yields the same P'—and $P \twoheadrightarrow^* \mathsf{Lift}(P')$ follows from the definition of $\twoheadrightarrow^*$. Certainly $(a_m \mid m \geq n_0)$ is an ω-chain in $[\![\sigma]\!]$, and thus

$$a = \bigvee_{n<\omega} a_n = \bigvee_{m \geq n_0} a_m \lhd_\sigma P'$$

as $\lhd_\sigma$ is chain complete. □

We can now complete the proof of Theorem 3.1. It is easy to prove the first part by rule induction on $P \twoheadrightarrow P'$. A corollary is that whenever $P \twoheadrightarrow^* V$, $[\![P]\!] = [\![V]\!] \in [\![\sigma]\!]$. For the second part, note that it follows from Lemma 3.8 that $[\![P{:}\sigma]\!] \lhd_\sigma P$ for any P of type σ. To see this, just note that $0 \lhd_{\mathsf{Unit}} ()$, and observe that one can prove $[\![x{:}\mathsf{Unit} \vdash P{:}\sigma]\!] = [\![P{:}\sigma]\!]$ for any program $P \in \{\, P \mid \exists\sigma(P{:}\sigma)\,\}$. Now suppose that we have $[\![P{:}\sigma_\bot]\!] \neq -$ and from Lemma 3.8 we have $[\![P{:}\sigma_\bot]\!] \lhd_{\sigma_\bot} P$. Hence, from the property of $\lhd_{\sigma_\bot}$ we deduce $P \twoheadrightarrow^* \mathsf{Lift}(P')$ for some P' as required. Finally, the third part of Theorem 3.1, that the denotational semantics is sound for the equational theory, follows as usual by a routine induction on the derivation of proved expressions.

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

Following [Plo85] we induce a denotational semantics on $\mathcal{O}$, via a textual translation $\langle\!\langle - \rangle\!\rangle$ of its types and expressions into $\mathcal{M}$. Each $\mathcal{O}$ type τ is sent to an $\mathcal{M}$ type $\langle\!\langle \tau \rangle\!\rangle$ that models $\mathcal{O}$ values of type τ. We have $\langle\!\langle \texttt{unit} \rangle\!\rangle \overset{\text{def}}{=} \mathsf{Unit}$, $\langle\!\langle \texttt{bool} \rangle\!\rangle \overset{\text{def}}{=} \mathsf{Bool}$, $\langle\!\langle \texttt{int} \rangle\!\rangle \overset{\text{def}}{=} \mathsf{Int}$ and $\langle\!\langle \tau_1 \texttt{ * } \tau_2 \rangle\!\rangle \overset{\text{def}}{=} \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle$. Our translation of an $\mathcal{O}$ function, $\langle\!\langle \tau_1 \texttt{ -> } \tau_2 \rangle\!\rangle$, must model the "pseudo-functions" `read` and `write`, and so cannot simply be $\langle\!\langle \tau_1 \rangle\!\rangle \rightarrow \langle\!\langle \tau_2 \rangle\!\rangle$ but must be $\langle\!\langle \tau_1 \rangle\!\rangle \rightarrow \mathsf{T}\langle\!\langle \tau_2 \rangle\!\rangle$, where the range is a type of *computations* [Mog89]. If τ is an $\mathcal{O}$ type, $\mathcal{M}$ type $\mathsf{T}\langle\!\langle \tau \rangle\!\rangle$ 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 the Pisa notes [Plo78, Chapter 5, Exercise 4], we set $\mathsf{T}\sigma \overset{\text{def}}{=} (\mathsf{U}(\sigma))_\perp$ 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 abbreviations:

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

Roughly speaking, a computation of type $\mathsf{T}\langle\!\langle\tau\rangle\!\rangle$ consists of potentially unbounded strings of Read's or Write's terminated with either – or a Return bearing an element of type $\langle\!\langle\tau\rangle\!\rangle$. Hence $\mathsf{T}\langle\!\langle\tau\rangle\!\rangle$ 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 [Mos90]).

We need also a sequential composition, an $\mathcal{M}$ program, Let, that runs one computation after another, with the following type.

$$\mathsf{Let}\colon\quad \mathsf{T}\sigma \times (\sigma \to \mathsf{T}\sigma') \to \mathsf{T}\sigma'$$

(Strictly speaking, this is a type scheme, and Let is a type-indexed family of programs.) We shall define Let recursively using a fixpoint program. It is routine to derive such a program, with the following properties, from the Rec operator.

Proposition 4.1 For each program P of type $(\sigma \to \tau_\perp) \to (\sigma \to \tau_\perp)$ there is a program $\mathsf{Fix}\,P$ of type $\sigma \to \tau_\perp$ such that $(\mathsf{Fix}\,P)\,Q \to^+ P\,(\mathsf{Fix}\,P)\,Q$ for any program Q of type σ.

Proof Omitted. See [Gor94, p61] for a proof. □

The program Let has the following recursive definition that roughly speaking stitches together the strings of I/O operations denoted by its two arguments.

```
Let  ≝  Fix(λlet. λx. Split x as (îo, f) in
          Drop îo to io in
            Case io of
              c_rd(g) → Read(λy. let (g y, f))
              c_wr(x) → Split x as (y, îo') in Write(y, let (îo', f))
              c_ret(x) → f x)
```

Note that *let*, $\imath o$ and $\widehat{\imath o}$ and their primed variants are simply $\mathcal{M}$ variables. Program Let has the following reduction behaviour.

$$\begin{array}{ll}\mathsf{Let}(P, \lambda x.\, E) \to^{+} \mathsf{Drop}\ P\ \mathsf{to}\ \imath o\ \mathsf{in}\ \mathsf{Case}\ \imath o\ \mathsf{of} & \\ \qquad c_{rd}(g) \to \mathsf{Read}(\lambda y.\, \mathsf{Let}(g\, y, \lambda x.\, E)) & \\ \qquad c_{wr}(x) \to \mathsf{Split}\ x\ \mathsf{as}\ (y, \widehat{\imath o}')\ \mathsf{in}\ \mathsf{Write}(y, \mathsf{Let}(\widehat{\imath o}', \lambda x.\, E)) & \\ \qquad c_{ret}(x) \to (\lambda x.\, E)\, x & \end{array}$$

Lemma 4.2 (1) $\mathsf{Let}(\mathsf{Return}(P), \lambda x.\, E) \to^{+} E[{}^{P}\!/\!_{x}]$

(2) $\mathsf{Let}(\mathsf{Write}(P, Q), \lambda x.\, E) \to^{+} \mathsf{Write}(P, \mathsf{Let}(Q, \lambda x.\, E))$

(3) $\mathsf{Let}(\mathsf{Read}(P), \lambda x.\, E) \to^{+} \mathsf{Read}(\lambda y.\, \mathsf{Let}(P(y), \lambda x.\, E))$

Proof (1) and (3) follow immediately from the reduction above. For (2) we have

$$\begin{array}{ll}\mathsf{Let}(\mathsf{Write}(P, Q), \lambda x.\, E) & . \\ \quad \to^{+} \quad \mathsf{Split}\ (P, Q)\ \mathsf{as}\ (y, \widehat{\imath o}')\ \mathsf{in}\ \mathsf{Write}(y, \mathsf{Let}(\widehat{\imath o}', \lambda x.\, E)) & \\ \quad \to^{+} \quad \mathsf{Write}(P, \mathsf{Let}(Q, \lambda x.\, E)) & \end{array}$$

□

$\mathcal{O}$ expressions are inductively translated into $\mathcal{M}$ expressions, following the monadic style pioneered by [Mog89] and [Pit91]. We simultaneously define the translation $\langle\!\langle - \rangle\!\rangle$ of arbitrary $\mathcal{O}$ expressions to $\mathcal{M}$ expressions, and an auxiliary translation $\llparenthesis - \rrparenthesis$ of $\mathcal{O}$ value expressions. Here are the rules for value expressions.

$$\begin{array}{rcl}
\llparenthesis x \rrparenthesis & \equiv & x \\
\llparenthesis \texttt{()} \rrparenthesis & \equiv & () \\
\llparenthesis \underline{\ell} \rrparenthesis & \equiv & \lfloor \ell \rfloor \\
\llparenthesis \underline{\oplus} \rrparenthesis & \equiv & \lambda x.\, \mathsf{Split}\ x\ \mathsf{as}\ (y, y')\ \mathsf{in}\ \mathsf{Drop}\ y \lfloor \oplus \rfloor y'\ \mathsf{to}\ z\ \mathsf{in}\ \mathsf{Return}(z) \\
\llparenthesis \texttt{fst} \rrparenthesis & \equiv & \lambda x.\, \mathsf{Split}\ x\ \mathsf{as}\ (y, z)\ \mathsf{in}\ \mathsf{Return}(y) \\
\llparenthesis \texttt{snd} \rrparenthesis & \equiv & \lambda x.\, \mathsf{Split}\ x\ \mathsf{as}\ (y, z)\ \mathsf{in}\ \mathsf{Return}(z) \\
\llparenthesis \delta \rrparenthesis & \equiv & \mathsf{Fix}(\lambda f_{\delta}.\, \lambda x.\, \langle\!\langle e_{\delta}[{}^{f_{\delta}}\!/\!_{\delta}] \rangle\!\rangle) \qquad \text{given } x{:}\tau_{\sigma} \vdash e_{\delta} : \tau'_{\sigma} \\
\llparenthesis \texttt{(}v, u\texttt{)} \rrparenthesis & \equiv & (\llparenthesis v \rrparenthesis, \llparenthesis u \rrparenthesis) \\
\llparenthesis \texttt{(}\lambda x{:}\tau.\, e\texttt{)} \rrparenthesis & \equiv & \lambda x{:}\langle\!\langle \tau \rangle\!\rangle.\, \langle\!\langle e \rangle\!\rangle \\
\llparenthesis \texttt{read} \rrparenthesis & \equiv & \lambda x{:}\mathsf{Unit}.\, \mathsf{Read}(\lambda y.\, \mathsf{Return}(y)) \\
\llparenthesis \texttt{write} \rrparenthesis & \equiv & \lambda x{:}\mathsf{Int}.\, \mathsf{Write}(x, \mathsf{Return}(()))
\end{array}$$

Here are the rules for expressions. Since value expressions, ranged over by *ve*, are also expressions, ranged over by *e*, there is overlap between the rules marked $(*)$

and some later rules. In case of overlap, a rule marked $(*)$ takes precedence over any later rule.

$$
\begin{array}{rcll}
\langle\!\langle ve \rangle\!\rangle & \equiv & \mathsf{Return}(\llparenthesis ve \rrparenthesis) & (*)\\
\langle\!\langle \Omega \rangle\!\rangle & \equiv & \mathsf{Rec}\, x\, \mathsf{in}\, x & \\
\langle\!\langle \mathtt{if}\; e_1\, \mathtt{then}\, e_2\, \mathtt{else}\; e_3 \rangle\!\rangle & \equiv & \mathsf{Let}(\langle\!\langle e_1 \rangle\!\rangle, \lambda x.\, \mathsf{If}\; x\, \mathsf{then}\, \langle\!\langle e_2 \rangle\!\rangle\; \mathsf{else}\; \langle\!\langle e_3 \rangle\!\rangle) & \\
\langle\!\langle ve_1\, e_2 \rangle\!\rangle & \equiv & \mathsf{Let}(\langle\!\langle e_2 \rangle\!\rangle, \lambda x.\, (\llparenthesis ve_1 \rrparenthesis\; x) & (*)\\
\langle\!\langle e_1\, e_2 \rangle\!\rangle & \equiv & \mathsf{Let}(\langle\!\langle e_1 \rangle\!\rangle, \lambda f.\, \mathsf{Let}(\langle\!\langle e_2 \rangle\!\rangle, \lambda x.\, f\; x)) & \\
\langle\!\langle \mathtt{(}ve_1, e_2\mathtt{)} \rangle\!\rangle & \equiv & \mathsf{Let}(\langle\!\langle e_2 \rangle\!\rangle, \lambda y.\, \mathsf{Return}((\llparenthesis e_1 \rrparenthesis, y))) & (*)\\
\langle\!\langle \mathtt{(}e_1, e_2\mathtt{)} \rangle\!\rangle & \equiv & \mathsf{Let}(\langle\!\langle e_1 \rangle\!\rangle, \lambda x.\, \mathsf{Let}(\langle\!\langle e_2 \rangle\!\rangle, \lambda y.\, \mathsf{Return}((x, y)))) &
\end{array}
$$

Lemma 4.3

(1) *If* $x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash ve : \tau$, *then* $x_1{:}\langle\!\langle \tau_1 \rangle\!\rangle, \ldots, x_n{:}\langle\!\langle \tau_n \rangle\!\rangle \vdash \llparenthesis ve \rrparenthesis : \langle\!\langle \tau \rangle\!\rangle$ *too.*

(2) *If* $x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash e : \tau$, *then* $x_1{:}\langle\!\langle \tau_1 \rangle\!\rangle, \ldots, x_n{:}\langle\!\langle \tau_n \rangle\!\rangle \vdash \langle\!\langle e \rangle\!\rangle : T\langle\!\langle \tau \rangle\!\rangle$ *too.*

Proof By a simultaneous induction on the derivations of $x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash ve : \tau$ and $x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash e : \tau$. □

Lemma 4.4

(1) *Whenever* $\Gamma, x{:}\tau \vdash e : \tau'$ *and* $\Gamma \vdash ve : \tau$, e *is a value expression iff* $e[{}^{ve}\!/\!_{x}]$.

(2) *If* $\Gamma, x{:}\tau \vdash e : \tau'$ *and* $\Gamma \vdash ve : \tau$ *then* $\langle\!\langle e \rangle\!\rangle[{}^{\llparenthesis ve \rrparenthesis}\!/\!_{x}] \equiv \langle\!\langle e[{}^{ve}\!/\!_{x}] \rangle\!\rangle$.

(3) *If* $p \to q$ *then* $\langle\!\langle p \rangle\!\rangle \to^{+} \langle\!\langle q \rangle\!\rangle$.

Proof (1) follows by induction on the derivation of $\Gamma, x{:}\tau \vdash e : \tau'$, and likewise (2), which depends on (1) for the cases of the translation $\langle\!\langle - \rangle\!\rangle$ that are conditional on whether an expression is a value expression. (3) follows by induction on the derivation of $p \to q$. □

Part (3) makes the proof of Lemma 4.6 particularly simple. Without the conditional translation rules for applications and pairs part (3) would fail. If $p \to q$ we would have $\mathtt{(}v, p\mathtt{)} \to \mathtt{(}v, q\mathtt{)}$ but not $\langle\!\langle \mathtt{(}v, p\mathtt{)} \rangle\!\rangle \to^{+} \langle\!\langle \mathtt{(}v, q\mathtt{)} \rangle\!\rangle$.

Lemma 4.5 *If* $\mathcal{C}[\mathtt{write}\; \underline{n}]$ *and* $\mathcal{C}[\mathtt{read\;()}]$ *are communicators and* v *is a value,*

$$
\begin{array}{rcl}
\langle\!\langle v \rangle\!\rangle & = & \mathsf{Return}(\llparenthesis v \rrparenthesis)\\
\langle\!\langle \mathcal{C}[\mathtt{read\;()}] \rangle\!\rangle & = & \mathsf{Read}(\lambda x{:}\mathsf{Int}.\, \langle\!\langle \mathcal{C}[x] \rangle\!\rangle)\\
\langle\!\langle \mathcal{C}[\mathtt{write}\; \underline{n}] \rangle\!\rangle & = & \mathsf{Write}(\lfloor n \rfloor, \langle\!\langle \mathcal{C}[\mathtt{()}] \rangle\!\rangle)
\end{array}
$$

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

Proof The first equation follows by definition of $\langle\!\langle v \rangle\!\rangle$. We can prove the second by induction on the number of experiments making up evaluation context $\mathcal{C}$. For the base case, when $\mathcal{C}$ is simply a hole, $[\,]$, we have the following.

$$\begin{aligned}
\langle\!\langle \mathtt{read}\,() \rangle\!\rangle &\equiv \mathsf{Let}(\langle\!\langle () \rangle\!\rangle, \lambda x.\, (\!|\mathtt{read}|\!)\, x) \\
&= (\lambda x.\, \mathsf{Read}(\lambda y.\, \mathsf{Return}(y)))\, () \\
&= \mathsf{Read}(\lambda x.\, \mathsf{Return}(x)) \\
&= \mathsf{Read}(\lambda x.\, \langle\!\langle x \rangle\!\rangle)
\end{aligned}$$

In the inductive case, the context $\mathcal{C}$ takes the form $\mathcal{E}[\mathcal{C}']$ where $\mathcal{E}$ is a single experiment and $\mathcal{C}'$ a smaller context. We shall only consider the case where $\mathcal{E}$ is an application of the form $(v\,[\,])$.

$$\begin{aligned}
\langle\!\langle \mathcal{C}[(\mathtt{read}\,())] \rangle\!\rangle &\equiv \langle\!\langle v\,(\mathtt{read}\,()) \rangle\!\rangle \\
&= \mathsf{Let}(\langle\!\langle \mathcal{C}'[\mathtt{read}\,()] \rangle\!\rangle, \lambda x.\, (\!|v|\!)\, x) \\
&= \mathsf{Let}(\mathsf{Read}(\lambda y.\, \langle\!\langle \mathcal{C}'[y] \rangle\!\rangle), \lambda x.\, (\!|v|\!)\, x) \quad \text{(induction hypothesis)} \\
&= \mathsf{Read}(\lambda y.\, \mathsf{Let}(\langle\!\langle \mathcal{C}'[y] \rangle\!\rangle, \lambda x.\, (\!|v|\!)\, x)) \quad \text{(Lemma 4.2)} \\
&= \mathsf{Read}(\lambda y.\, \langle\!\langle v\, \mathcal{C}'[y] \rangle\!\rangle) \\
&= \mathsf{Read}(\lambda x.\, \langle\!\langle \mathcal{C}[x] \rangle\!\rangle)
\end{aligned}$$

The other cases are similar. The third equation can be proved similarly. □

Lemma 4.6 *$p\Downarrow$ iff $\langle\!\langle p \rangle\!\rangle\Downarrow$.*

Proof

(Only If) Suppose $p \to^* a$. So $\langle\!\langle p \rangle\!\rangle = \langle\!\langle a \rangle\!\rangle$ by Lemma 4.4 and Theorem 3.1. But then $\langle\!\langle p \rangle\!\rangle\Downarrow$ by Lemma 4.5 and Theorem 3.1.

(If) We prove the contrapositive. If not $p\Downarrow$ there must be an infinite chain $p \to p_1 \to p_2 \to \cdots$ in $\mathcal{O}$. By Lemma 4.4 there is another infinite chain $\langle\!\langle p \rangle\!\rangle \to^+ \langle\!\langle p_1 \rangle\!\rangle \to^+ \langle\!\langle p_2 \rangle\!\rangle \to^+ \cdots$ in $\mathcal{M}$ and hence not $\langle\!\langle p \rangle\!\rangle\Downarrow$.

□

Lemma 4.7 *If $\langle\!\langle a \rangle\!\rangle = \langle\!\langle b \rangle\!\rangle$ and $a \xrightarrow{\alpha} p$ there is q with $b \xrightarrow{\alpha} q$ and $\langle\!\langle p \rangle\!\rangle = \langle\!\langle q \rangle\!\rangle$.*

Proof The proof is by a case analysis of how $a \xrightarrow{\alpha} p$ was derived. We first consider the two cases where a is a communicator.

- $a \xrightarrow{?n} \mathcal{C}[\underline{n}]$ if $a \equiv \mathcal{C}[\mathtt{read}\,()]$. So by Lemma 4.5 and the fact that $\mathsf{Read}(-)$, $\mathsf{Write}(-)$ and $\mathsf{Return}(-)$ have disjoint images, b must be a communicator of the form $\mathcal{D}[\mathtt{read}\,()]$. So $b \xrightarrow{?n} \mathcal{D}[\underline{n}]$. We have $\langle\!\langle a \rangle\!\rangle = \mathsf{Read}(\lambda x.\, \langle\!\langle \mathcal{C}[x] \rangle\!\rangle)$ and $\langle\!\langle b \rangle\!\rangle = \mathsf{Read}(\lambda x.\, \langle\!\langle \mathcal{D}[x] \rangle\!\rangle)$. Since $\mathsf{Read}(-)$ is injective we have $\lambda x.\, \langle\!\langle \mathcal{C}[x] \rangle\!\rangle = \lambda x.\, \langle\!\langle \mathcal{D}[x] \rangle\!\rangle$ and in particular $\langle\!\langle \mathcal{C}[\underline{n}] \rangle\!\rangle = \langle\!\langle \mathcal{D}[\underline{n}] \rangle\!\rangle$, as required.

- $a \xrightarrow{!n} \mathcal{C}[\texttt{()}]$ if $a \equiv \mathcal{C}[\texttt{write}\,\underline{n}]$. Again by Lemma 4.5, b must be a communicator of the form $\mathcal{D}[\texttt{write}\,\underline{n}]$. We have $\langle\!\langle a \rangle\!\rangle = \mathsf{Write}(\lfloor n \rfloor, \langle\!\langle \mathcal{C}[\texttt{()}] \rangle\!\rangle)$ and $\langle\!\langle b \rangle\!\rangle = \mathsf{Write}(\lfloor n \rfloor, \langle\!\langle \mathcal{D}[\texttt{()}] \rangle\!\rangle)$, and since $\mathsf{Write}(-)$ is injective, $\langle\!\langle \mathcal{C}[\texttt{()}] \rangle\!\rangle = \langle\!\langle \mathcal{D}[\texttt{()}] \rangle\!\rangle$ as required.

Now we consider the possibilities where a is a value.

- $a \xrightarrow{\ell} \Omega$ if $a \equiv \underline{\ell}$. Since $\langle\!\langle a \rangle\!\rangle = \langle\!\langle b \rangle\!\rangle$ and b is a value, $b \equiv \underline{\ell}$ too, and so $b \xrightarrow{\ell} \Omega$ too.

- $a \xrightarrow{\texttt{fst}} u_1$ if $a \equiv \texttt{(}u_1, u_2\texttt{)}$. In this case b must be a pair too, say $\texttt{(}v_1, v_2\texttt{)}$. Hence $a \xrightarrow{\texttt{fst}} v_1$ and $\langle\!\langle \texttt{(}u_1, u_2\texttt{)} \rangle\!\rangle = \langle\!\langle \texttt{(}v_1, v_2\texttt{)} \rangle\!\rangle$ implies that $\langle\!\langle u_1 \rangle\!\rangle = \langle\!\langle v_1 \rangle\!\rangle$.

- $a \xrightarrow{\texttt{snd}} u_2$ if $a \equiv \texttt{(}u_1, u_2\texttt{)}$. Symmetric to the previous case.

- $a \xrightarrow{@v} a\,v$ if $a\,v$ a program. Since a is a function so is b. Hence we have $b \xrightarrow{@v} b\,v$ and $\langle\!\langle a\,v \rangle\!\rangle = \langle\!\langle b\,v \rangle\!\rangle$ by compositionality.

□

Lemma 4.8 *Relation* $\mathcal{S} \stackrel{\text{def}}{=} \{(p, q) \mid \langle\!\langle p \rangle\!\rangle = \langle\!\langle q \rangle\!\rangle\}$ *is a bisimulation.*

Proof Suppose that $p \mathcal{S} q$ and that $p \xrightarrow{\alpha} p'$. By Lemma 2.2 there is a with $p \to^* a$ and $a \xrightarrow{\alpha} p'$. By Lemma 4.4 we have $\langle\!\langle p \rangle\!\rangle \to^* \langle\!\langle a \rangle\!\rangle$ and therefore $\langle\!\langle p \rangle\!\rangle = \langle\!\langle a \rangle\!\rangle$ by Theorem 3.1. By transitivity $\langle\!\langle q \rangle\!\rangle = \langle\!\langle a \rangle\!\rangle$ holds, so by Theorem 3.1 and Lemma 4.5 we have $\langle\!\langle q \rangle\!\rangle\Downarrow$. Hence $q\Downarrow$ by Lemma 4.6, that is, there is active b with $q \to^* b$. By Lemma 4.4 and Theorem 3.1 we have $\langle\!\langle q \rangle\!\rangle = \langle\!\langle b \rangle\!\rangle$ and so $\langle\!\langle a \rangle\!\rangle = \langle\!\langle b \rangle\!\rangle$ by transitivity. Hence by Lemma 4.7 there is q' with $b \xrightarrow{\alpha} q'$ and $\langle\!\langle p' \rangle\!\rangle = \langle\!\langle q' \rangle\!\rangle$. 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. □

Theorem 4.9 $\langle\!\langle p \rangle\!\rangle = \langle\!\langle q \rangle\!\rangle$ implies $p \sim q$.

Proof Suppose $\langle\!\langle p \rangle\!\rangle = \langle\!\langle q \rangle\!\rangle$. Then (p, q) is a member of a bisimulation, the $\mathcal{S}$ of Lemma 4.8. So $p \sim q$ since every bisimulation is included in $\sim$, by its definition. □

5 Discussion

By consolidating prior work on operational semantics, bisimulation equivalence and metalanguages 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 equality of denotations (Theorem 3).

Williams and Wimmers [WW88] are perhaps the only others 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 [Mos90, Plo78] were not related to operational semantics. In his CSLI lecture notes, Plotkin [Plo85] 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 4.6) could be factored into an adequacy result for the metalanguage (analogous to Theorem 3.1) together with comparatively routine calculations about the operational semantics. Moggi [Mog89] pioneered a monadic approach to modularising semantics. In an earlier study [CG93] 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 [FP93]). Instead we use the minimal invariant property which characterises the (smallest) coincidence. Second, we use the adequacy result for $\mathcal{O}$ (Lemma 4.6) and co-induction to prove the soundness of denotational 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 the concurrent γ-calculus of Boudol [Bou89], but Boudol does not establish whether bisimilarity on his calculus is a congruence. Applicative bisimulation [AO92] 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.

Since the work reported here was completed, Gordon [Gor95b, Gor95a] has investigated a labelled transition system semantics for a variety of stateless functional languages, without I/O. A useful future project would be to extend the results of this paper to a language with nondeterminism and concurrency. Indeed, since this work was completed, Jeffrey has investigated monadic languages (analogous to our $\mathcal{M}$) with nondeterminism [Jef95a] and concurrency [Jef95b]. Based on the presentation in [Gor95b] of a labelled transition system form of Howe's congruence proof, Jeffrey showed that bisimilarity for his concurrent monadic language is a congruence. A useful next step would be to extend this result to a language, like our $\mathcal{O}$, in which side-effects are freely mixed with applicative computation.

Having worked through the details of both a classical denotational semantics for $\mathcal{O}$ and an entirely operational treatment of bisimilarity, we are in a position to compare the two approaches. Though we have not spelt out the details, both operational and denotational semantics can validate an equational theory for $\mathcal{O}$. Bisimilarity immediately offers a co-induction principle, and a domain-theoretic semantics a fixpoint induction principle. With more work co-induction can be derived from a denotational semantics [Pit94a] and fixpoint induction from an operational semantics [MST, Smi91]. Finally, we found that the intermediate metalanguage $\mathcal{M}$ usefully modularised the denotational semantics of $\mathcal{O}$; the details of sections 3 and 4 can be understood independently of one another.

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. The commutative diagrams were typeset using Paul Taylor's macros.

References

[AO92] Samson Abramsky and Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Control*, 105:159–267, 1992.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 119–129, 1992.

[Bou89] G. Boudol. Towards a lambda-calculus for concurrent and communicating systems. *Lecture Notes in Computer Science*, 351:149–161, 1989.

[CG93] R. L. Crole and A. D. Gordon. Factoring an adequacy proof. In C. J.van Rijsbergen, editor, *FP'93 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 9–25. Springer-Verlag, 1993.

[CP92] R.L. Crole and A.M. Pitts. New foundations for fixpoint computations: FIX hyperdoctrines and the FIX logic. *Information and Control*, 98:171–210, 1992.

[Cro92] Roy L. Crole. *Programming Metalogics with a Fixpoint Type*. PhD thesis, University of Cambridge Computer Laboratory, February 1992. Available as Technical Report 247.

[Cro93] R. L. Crole. *Categories for Types*. Cambridge Mathematical Textbooks. Cambridge University Press, 1993.

[FP93] M. P. Fiore and G. D. Plotkin. An axiomatisation of computationally adequate domain theoretic models of fpc. In *9th Annual Symposium on Logic in Computer Science*. I.E.E.E. Computer Society Press, 1993.

[Gor93] Andrew D. Gordon. An operational semantics for I/O in a lazy functional language. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 136–145. ACM Press, 1993.

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, 1994.

[Gor95a] A. D. Gordon. Bisimilarity as a theory of functional programming. Technical report, Aarhus University, Denmark, 1995. BRICS Notes Series NS–95–3, BRICS, Aarhus University.

[Gor95b] A. D. Gordon. Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[Hol83] Sören Holmström. PFL: A functional language for parallel programming. In *Declarative Programming Workshop*, pages 114–139. University College, London, 1983. Extended version published as Report 7, Programming Methodology Group, Chalmers University. September 1983.

[How89] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[Jef95a] A. Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *Tenth annual symposium on Logic In Computer Science*, pages 255–264. I.E.E.E. Computer Society Press, 1995.

[Jef95b] A. Jeffrey. A fully abstract semantics for a nondeterministic functional language with monadic types. *Electronic Notes in Theoretical Computer Science*, 1, 1995.

[MAE$^+$62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Mog89] E. Moggi. Notions of computation and monads. *Theoretical Computer Science*, 93:55–92, 1989.

[Mos90] Peter D. Mosses. Denotational semantics. In Jan Van Leeuven, editor, *Handbook of Theoretical Computer Science*, chapter 11, pages 575–631. Elsevier Science Publishers B. V., 1990. Volume B.

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

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1990.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *Monographs on Computer Science.* Oxford University Press, 1990.

[Pit91] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer-Verlag, Berlin, 1991.

[Pit94a] Andrew M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124:195–219, 1994.

[Pit94b] Andrew M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *Proceedings Mathematical Foundations of Programming Semantics IX, New Orleans 1993*, volume 802 of *Lecture Notes in Computer Science*, pages 72–82. Springer-Verlag, 1994.

[Plo78] Gordon D. Plotkin. The category of complete partial orders: a tool for making meanings. Unpublished lecture notes for the Summer School on Foundations of Artificial Intelligence and Computer Science, Pisa., June 1978.

[Plo85] G.D. Plotkin. Denotational semantics with partial functions. Unpublished lecture notes from CSLI summer school, 1985.

[RC86] Jonathan Rees and William Clinger. Revised report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[Sco69] D.S. Scott. Models of the lambda calculus. Unpublished manu- script, 1969.

[Smi91] Scott F. Smith. From operational to denotational semantics. In *MFPS VII, Pittsburgh*, volume 598 of *Lecture Notes in Computer Science*, pages 54–76. Springer-Verlag, 1991.

[WW88] John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the lineΓ In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 169–179, January 1988.