Factoring an Adequacy Proof (Preliminary Version)

Roy L. Crole Andrew D. Gordon

1994

Abstract

This paper contributes to the methodology of using metalogics for reasoning about programming languages. As a concrete example we consider a fragment of ML corresponding to call-by-value PCF and translate it into a metalogic which contains (amongst other types) computation types and a fixpoint type. The main result is a soundness property ($\star$): if the denotations of two programs are provably equal in the metalogic, they have the same operationally observable behaviour. As usual, this follows from a computational adequacy result. In early notes, Plotkin showed how such proofs could be factored into two stages, the first non-trivial and the second (essentially) routine; our contribution is to rework his suggestion within a new framework. We define a metalogic, which incorporates computation and fixpoint types, and specify a modular translation of the ML fragment. Our proof of ($\star$) factors into two parts. First, the term language of the metalogic is equipped with an operational semantics and a (generic) computational adequacy result obtained. Second, a simple syntactic argument establishes a correspondence between the operational behaviour of an object program and of its denotation. The first part is not routine but is proved once and for all. The second is a detailed but essentially trivial calculation that is easily adaptable to other object languages. Such a factored proof is important because it promises to scale up more easily than a monolithic one. We show that it may be adapted to an object language with call-by-name functions and one with a simple exception mechanism.

1 Motivation and Background

The motivation for this work is to contribute to the long term project of using denotational semantics to prove properties of programs written in realistic languages like Standard ML or Haskell.

Our contribution is to the methodology of expressing and reasoning about denotational semantics using what we term *metalogics*. By metalogic we mean a formal system (such as a type theory or logic) intended to express denotational semantics, that is implementable in a theorem-prover, and which is equipped

with general proof principles, such as natural number induction or fixpoint induction. The intention is that the general proof infrastructure of the metalogic can be applied to a particular object language via a translation into the metalogic. We shall define a metalogic (called $\mathcal{M}$) which has a basic syntax of types and terms, and is equipped with both an equational and an operational semantics. $\mathcal{M}$ is closely allied to Crole and Pitts' FIX-logic [4, 3], a metalogic based on ideas from Moggi's *computational let-calculus* [10, 11] and a precursor of Pitts' *evaluation logic* [15]. An important principle underlying each of these metalogics is to distinguish simple data values from computations; more precisely, each uses *computation types*, of the form $T\sigma$, to represent computations that may return data values of type σ. Each of these metalogics is *monadic* in the sense that a computation type is modelled by a strong monad.

We consider the general problem of how to prove soundness of metalogical reasoning about an object language. The object language we use as a vehicle for this problem is $\mathcal{O}$, a small fragment of ML corresponding to call-by-value PCF. We give a translation into $\mathcal{M}$ and prove its soundness: if the denotations of two programs are provably equal in the metalogic, then the two are in fact observationally equivalent. Given that ML is defined operationally, the result justifies use of the metalogic to derive properties of the object language.

It is standard to derive such a soundness result from a proof that the denotational semantics respects evaluation in a sense known as computational adequacy [8]. Proofs of computational adequacy are, usually, directly linked to the denotational semantics of the object language in question. In his CSLI notes [17], Plotkin showed how such proofs can be factored in two, via an operational semantics for the metalogic. Our contribution is to rework (the idea of) such a factorisation in the setting of a metalogic endowed with computation and fixpoint types. First, we prove a (non-routine) adequacy result which relates the equational and operational semantics of $\mathcal{M}$. Second, we obtain soundness of metalogical reasoning for $\mathcal{O}$ via a simple but detailed proof of a correspondence between the operational behaviour of each $\mathcal{O}$ program and its denotation, which utilises the adequacy result for $\mathcal{M}$. The computational adequacy of $\mathcal{M}$ is *generic* in the sense that a computational adequacy result for a new programming language will only require a reworking of the second (and simpler) proof stage. We present two variants of $\mathcal{O}$ as evidence for this genericity: one with call-by-name functions and another with a simple exception mechanism. The use of computation types gives an elegant structure to the form of $\mathcal{O}$'s denotational semantics in general, and the fixpoint type gives rise to a uniform denotational semantics for $\mathcal{O}$'s recursive function declarations in particular. This factored proof is important as it is likely to scale up more easily than a monolithic one, and the monadic presentation ought to be of use as monadic metalogics are mechanised and applied to more realistic object languages.

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{ValL}(M) : \sigma_\perp} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{EvalL}(M) : \sigma_\perp}\ (\sigma \text{ total}) \qquad \frac{\Gamma \vdash E : \sigma_\perp \qquad \Gamma, x{:}\sigma \vdash F : \tau_\perp}{\Gamma \vdash \mathsf{LetL}\ x \Leftarrow E\ \mathsf{in}\ F : \tau_\perp}$$

$$\frac{\Gamma, e{:}\sigma_\perp \vdash F : \sigma \qquad \Gamma \vdash N : \mathit{fix}}{\Gamma \vdash \mathsf{It}(e.F, N) : \sigma} \qquad \frac{}{\Gamma \vdash \omega : \mathit{fix}_\perp} \qquad \frac{\Gamma \vdash E : \mathit{fix}_\perp}{\Gamma \vdash \mathsf{Inc}(E) : \mathit{fix}}$$

Table 1: Type assignment for $\mathcal{M}$

2 An Equational Metalogic $\mathcal{M}$

We now present a Martin-Löf style (simple) type theory $\mathcal{M}$ which will be viewed as a programming metalogic. This is based on the term language of the FIX-logic [3, 4], though $\mathcal{M}$ has a few crucial differences. The term language of $\mathcal{M}$ is well known, except, perhaps, for the fragment associated with the fixpoint type and lifted types $\sigma_\perp$. Let us summarise the types and (raw) terms:

$$\begin{array}{rcl}
\sigma & ::= & \mathit{unit} \mid \mathit{nat} \mid \mathit{fix} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma \mid \sigma_\perp \\
M & ::= & x \mid \langle\rangle \mid \mathsf{Z} \mid \mathsf{S}(M) \mid (x.M)^M(M) \mid \omega \mid \mathsf{Inc}(M) \mid \mathsf{It}(x.M, M) \mid \\
 & & \langle M, M\rangle \mid \mathsf{Split}(M, x.x.M) \mid \mathsf{Inl}(M) \mid \mathsf{Inr}(M) \mid \mathsf{Case}(M, x.M, x.M) \mid \\
 & & \lambda x.M \mid M\,M \mid \mathsf{ValL}(M) \mid \mathsf{EvalL}(M) \mid \mathsf{LetL}\ x \Leftarrow M\ \mathsf{in}\ M
\end{array}$$

The types are given by a unit type, natural numbers, fixpoint type, (co)products, exponentials, and lifted types. In the cases of the unit type, natural numbers, products, coproducts and exponentials, the raw terms are the usual ones of Martin-Löf's (simple) type theory—for background see Nordström *et al* [12]. The raw terms of the fixpoint type are described elsewhere [3, 4].

The type $\sigma_\perp$ is thought of as the type of partial computations with values of type σ. If M is any term, then $\mathsf{ValL}(M)$ is a computation which evaluates immediately yielding result M. $\mathsf{EvalL}(M)$ is similar, but M is never a partial computation. If E and F are both partial computations, then $\mathsf{LetL}\ x \Leftarrow E\ \mathsf{in}\ F$ can be thought of as the computation $F[^M/_x]$ provided that the partial computation E is defined and evaluates with result M, and is undefined if E is undefined. Up to provable equality $\mathsf{EvalL}(M)$ is the same as $\mathsf{ValL}(M)$; the two will be distinguished by their operational semantics. Evaluation of the former forces evaluation of M but evaluation of the latter does not. EvalL will be vital later on in obtaining precise relationships between operational and denotational semantics.

We give a type assignment system for $\mathcal{M}$. This will consist of rules for deriving judgments of the form $\Gamma \vdash M : \sigma$ where the *environment* Γ is a finite list of (variable, type) pairs. We shall refer to σ as the type *assigned* to the raw term M in the environment Γ. Well formed judgments of the form $\Gamma \vdash M : \sigma$ are generated by the rules for simply typed lambda-calculus with unit, natural

numbers, (co)products, plus the additional rules in Table 1. We write $\sigma \rightharpoonup \tau$ for $\sigma \to \tau_\perp$. We refer to a type σ, where $\sigma \not\equiv \tau_\perp$ for any τ, as a *total* type.

We can use the syntax of $\mathcal{M}$ as the basis for a pure equational type theory [4] in which theorems take the form $\Gamma \vdash M = M' : \sigma$; in this abstract we omit the rules for proving such judgments.

We can give a categorical semantics to $\mathcal{M}$ in the usual way. We shall interpret $\mathcal{M}$ in the FIX-category [3, 4] $\omega\mathcal{CPO}$ of ωcpos and Scott continuous functions, equipped with the lifting monad, and topped vertical natural numbers with the successor structure map as FPO. We shall write $D_\perp \stackrel{\text{def}}{=} \{\perp\} \cup \{[d] \mid d \in D\}$ for (the underlying set of) the lift of the ωcpo D, with unit $\eta_D \stackrel{\text{def}}{=} \lambda d \in D.[d] : D \to D_\perp$. We denote the FPO by $(\mathbb{N}^\infty, \infty, s)$ (using an obvious notation). If $f : C \times D_\perp \to D$ is a Scott continuous function, then the mediating morphism of the (indexed) FPO will be written $it(f) : C \times \mathbb{N}^\infty \to D$. We summarise the semantics of the less well known fragment of $\mathcal{M}$:

- $[\![\sigma_\perp]\!] \stackrel{\text{def}}{=} [\![\sigma]\!]_\perp$, $[\![\mathit{fix}]\!] \stackrel{\text{def}}{=} \mathbb{N}^\infty$,
- $[\![\Gamma \vdash \mathsf{EvalL}(M) : \sigma_\perp]\!] = [\![\Gamma \vdash \mathsf{ValL}(M) : \sigma_\perp]\!] \stackrel{\text{def}}{=} \eta_{[\![\sigma]\!]} \circ [\![\Gamma \vdash M : \sigma]\!]$,
- $[\![\Gamma \vdash \mathsf{LetL}\, x \Leftarrow E \,\mathsf{in}\, F : \tau_\perp]\!] \stackrel{\text{def}}{=} [\![\Gamma, x : \sigma \vdash F : \sigma_\perp]\!]_\perp \circ \langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash E : \sigma_\perp]\!] \rangle$, (where $f_\perp$ is the indexed Kliesli lifting of f in $\omega\mathcal{CPO}$),
- $[\![\Gamma \vdash \omega : \mathit{fix}_\perp]\!] \stackrel{\text{def}}{=} \infty \circ !$, where we have $[\![\Gamma]\!] \stackrel{!}{\longrightarrow} 1 \stackrel{* \mapsto \infty}{\longrightarrow} \mathbb{N}^\infty_\perp$, 1 terminal in $\omega\mathcal{CPO}$,
- $[\![\Gamma \vdash \mathsf{Inc}(E) : \mathit{fix}_\perp]\!] \stackrel{\text{def}}{=} s \circ [\![\Gamma \vdash E : \mathit{fix}_\perp]\!]$,
- $[\![\Gamma \vdash \mathsf{It}(e.F, N) : \sigma]\!] \stackrel{\text{def}}{=} it([\![\Gamma, e : \sigma_\perp \vdash F : \sigma]\!]) \circ \langle id_{[\![\Gamma]\!]}, [\![\Gamma \vdash N : \mathit{fix}]\!] \rangle$.

3 Computation Types and the Let-Calculus

Moggi [10, 11] introduced a (simple) type theory which is now often referred to as the computational let-calculus. His calculus captures the following intuitions about (many) programming languages. First, it is sensible to separate out the notions of computations of values from values themselves; second that any value may be regarded as a "trivial" computation yielding itself as its value; and third, that computations may be composed sequentially. In fact this type theory corresponds with the notion of a category (with finite products) equipped with a strong monad. For these reasons an instance of the computional let-calculus is often referred to as a computational monad. More precisely, a computational monad is specified by a triple $(T, \mathsf{Val}, \mathsf{Let})$ where T is a type constructor and Val and Let are term constructors, that satisfy the following type assignment rules,

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{Val}(M) : T\sigma} \qquad \frac{\Gamma \vdash M : T\sigma \qquad \Gamma, x : \sigma \vdash N : T\sigma'}{\Gamma \vdash \mathsf{Let}\, x \Leftarrow M \,\mathsf{in}\, N : T\sigma'}$$

and the following rules for deducing equational theorems.

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma, x{:}\sigma \vdash N : T\tau}{\Gamma \vdash \mathsf{Let}\, x \Leftarrow \mathsf{Val}(M)\, \mathsf{in}\, N = N[M/x] : T\sigma'} \qquad \frac{\Gamma \vdash M : T\sigma}{\Gamma \vdash \mathsf{Let}\, x \Leftarrow M\, \mathsf{in}\, \mathsf{Val}(x) = M : T\sigma}$$

$$\frac{\Gamma \vdash M : T\sigma \qquad \Gamma, x{:}\sigma \vdash N : T\sigma' \qquad \Gamma, y{:}\sigma' \vdash P : T\sigma''}{\Gamma \vdash \mathsf{Let}\, x \Leftarrow M\, \mathsf{in}\, (\mathsf{Let}\, y \Leftarrow N\, \mathsf{in}\, P) = \mathsf{Let}\, y \Leftarrow (\mathsf{Let}\, x \Leftarrow M\, \mathsf{in}\, N)\, \mathsf{in}\, P : T\sigma''}$$

As expected, the triple $((-)_\perp, \mathsf{ValL}, \mathsf{LetL})$ is a computational monad, usually known as the lifting monad. The equational logic of $\mathcal{M}$ must be omitted here for the sake of brevity, but (the appropriate instance of) the three (equational) rules above are indeed present in $\mathcal{M}$ [3, 4]. We include a small example at the end of the paper in Section 8 to illustrate the use of a monad other than the lifting monad.

4 An Operational Semantics for $\mathcal{M}$

Let us specify an operational semantics for (the term language of) $\mathcal{M}$, as done for a similar metalogic by Gordon [5]. We shall need a few auxiliary definitions. The *canonical* raw terms are given by the grammar

$$V \;::=\; \langle\rangle \mid \mathsf{Z} \mid \mathsf{S}(V) \mid \langle M, M\rangle \mid \mathsf{Inl}(M) \mid \mathsf{Inr}(M) \mid \lambda x.M \mid \mathsf{ValL}(M) \mid \mathsf{Inc}(M).$$

We write $\mathcal{M}_\sigma$ for the set of $\mathcal{M}$-*programs*, that is, raw terms M for which $\vdash M{:}\sigma$ is provable, and $\mathcal{M}_\sigma^{can} \subseteq \mathcal{M}_\sigma$ for the subset of canonical terms. We specify an operational semantics in two equivalent ways: as a 'big step' *evaluation relation*, $M \Downarrow V$, and a 'small step' *reduction relation*, $M \to N$, where M, N and V are programs and V is canonical. The two relations are generated in the usual way, except for the syntax involving lifted types and the fixpoint type; the rules for the evaluation and reduction relations appear in Table 2. It is straightforward to prove that $M \Downarrow V$ if and only if $M \to^* V$ (where $\to^*$ means reflexive, transitive closure of $\to$). The semantics is deterministic and call-by-name. If $M \in \mathcal{M}_\sigma$ and $M \to N$ then $N \in \mathcal{M}_\sigma$ too. It is easy to prove the following result by rule induction.

Proposition 1 *If $M \in \mathcal{M}_\sigma$ and also $M \Downarrow V$, then $V \in \mathcal{M}_\sigma^{can}$ and $\vdash M = V{:}\sigma$, where the latter judgment holds in the equational metalogic.*

Our principal aim here is to prove the following (generic) adequacy theorem.

Theorem 2 (Generic Adequacy) *The equational metalogic is computationally adequate for the operational semantics in the following sense. If $E \in \mathcal{M}_{\sigma_\perp}$ and $M \in \mathcal{M}_\sigma$ are such that $\vdash E = \mathsf{ValL}(M) : \sigma_\perp$ is provable, then there is $V \in \mathcal{M}_{\sigma_\perp}^{can}$ for which $E \Downarrow V$.*

We also prove a normalisation result at total types.

Theorem 3 (Normalisation) *If $M \in \mathcal{M}_\sigma$ with σ total, then evaluation of M converges, that is, $\exists V.\ M \Downarrow V$.*

$$\frac{}{V \Downarrow V}$$

$$\frac{M \Downarrow V}{\mathsf{S}(M) \Downarrow \mathsf{S}(V)}$$

$$\frac{N \Downarrow \mathsf{Z} \qquad M \Downarrow V}{(x.\, F)^{N}(M) \Downarrow V}$$

$$\frac{N \Downarrow \mathsf{S}(U) \qquad F[(x.\, F)^{U}(M)/x] \Downarrow V}{(x.\, F)^{N}(M) \Downarrow V}$$

$$\frac{P \Downarrow \langle M, N \rangle \qquad F[M, N/x, y] \Downarrow V}{\mathsf{Split}(P, x.\, y.\, F) \Downarrow V}$$

$$\frac{C \Downarrow \mathsf{Inl}(M) \qquad F[M/x] \Downarrow V}{\mathsf{Case}(C, x.\, F, x.\, G) \Downarrow V} \qquad \frac{C \Downarrow \mathsf{Inr}(M) \qquad G[M/x] \Downarrow V}{\mathsf{Case}(C, x.\, F, x.\, G) \Downarrow V}$$

$$\frac{F \Downarrow \lambda x.\, N \qquad N[M/x] \Downarrow V}{F\, M \Downarrow V}$$

$$\frac{M \Downarrow V}{\mathsf{EvalL}(M) \Downarrow \mathsf{ValL}(V)} \qquad \frac{E \Downarrow \mathsf{ValL}(M) \qquad F[M/x] \Downarrow V}{\mathsf{LetL}\, x \Leftarrow E \,\mathsf{in}\, F \Downarrow V}$$

$$\frac{N \Downarrow \mathsf{Inc}(E) \qquad M[\mathsf{LetL}\, y \Leftarrow E \,\mathsf{in}\, \mathsf{ValL}(\mathsf{It}(e.\, M, y))/e] \Downarrow V}{\mathsf{It}(e.\, M, N) \Downarrow V} \qquad \frac{}{\omega \Downarrow \mathsf{ValL}(\mathsf{Inc}(\omega))}$$

$$(x.F)^{\mathrm{Z}}(M) \to M$$

$$(x.F)^{\mathrm{S}(V)}(M) \to F[(x.F)^{V}(M)/x]$$

$$\mathsf{Split}(\langle M, N \rangle, x.\, y.\, F) \to F[M, N/x, y]$$

$$\mathsf{Case}(\mathsf{Inl}(M), x.\, F, x.\, G) \to F[M/x]$$

$$\mathsf{Case}(\mathsf{Inr}(M), x.\, F, x.\, G) \to G[M/x]$$

$$(\lambda x.\, N)M \to N[M/x]$$

$$\mathsf{LetL}\, x \Leftarrow \mathsf{ValL}(M) \,\mathsf{in}\, F \to F[M/x]$$

$$\mathsf{It}(e.\, M, \mathsf{Inc}(E)) \to M[\mathsf{LetL}\, y \Leftarrow E \,\mathsf{in}\, \mathsf{ValL}(\mathsf{It}(e.\, M, y))/e]$$

$$\omega \to \mathsf{ValL}(\mathsf{Inc}(\omega))$$

$$\mathsf{EvalL}(V) \to \mathsf{ValL}(V)$$

$$\frac{M \to N}{\mathcal{E}[M] \to \mathcal{E}[N]}$$

where $\mathcal{E}[-] ::= \mathsf{S}(-) \mid (x.F)^{-}(M) \mid \mathsf{Split}(-, x.\, y.\, F) \mid \mathsf{Case}(-, x.\, F, x.\, G) \mid (-\, M) \mid \mathsf{LetL}\, x \Leftarrow - \,\mathsf{in}\, E \mid \mathsf{It}(e.\, M, -) \mid \mathsf{EvalL}(-)$

Table 2: Operational semantics for $\mathcal{M}$

To prove this we need some technical machinery. Let $[\![-]\!]:\mathcal{M} \to \omega\mathcal{CPO}$ refer to the semantics of $\mathcal{M}$ in the FIX-category $\omega\mathcal{CPO}$. We now define a logical relation which takes the form $\lhd_\sigma \subseteq [\![\sigma]\!] \times \mathcal{M}_\sigma$ for all types σ, and is defined through certain inductive clauses of which we give just a few examples:

• $e \lhd_{\sigma_\perp} E$ iff whenever $e = [d]$ for $d \in [\![\sigma]\!]$ then $\exists M.\ E \Downarrow \mathsf{ValL}(M)$ and $d \lhd_\sigma M$.

• $\infty \lhd_{fix} N$ iff $\forall n \in \mathbb{N}^\infty \setminus \{\infty\}$ we have $n \lhd_{fix} N$.

• $n+1 \lhd_{fix} N$ iff $\exists E_n.\ N \Downarrow \mathsf{Inc}(E_n)$, $\exists N_n.\ E_n \Downarrow \mathsf{ValL}(N_n)$ and also for $1 \leq i \leq n$ we have $\exists E_{i-1}.\ N_i \Downarrow \mathsf{Inc}(E_{i-1})$, $\exists N_{i-1}.\ E_{i-1} \Downarrow \mathsf{ValL}(N_{i-1})$ and $\exists E.\ N_0 \Downarrow \mathsf{Inc}(E)$.

• $0 \lhd_{fix} N$ iff $\exists E.\ N \Downarrow \mathsf{Inc}(E)$.

We shall also need the following lemmas and proposition, for which we sketch the proof of the latter:

Lemma 4 *Suppose that $d \lhd_\sigma M$ and also $M' \to^* M$; then $d \lhd_\sigma M'$.*

Lemma 5 *Let $(d_i \mid i \in \omega)$ be an ω-chain in $[\![\sigma]\!]$ for any σ. If $d_i \lhd_\sigma M$ for some M and each $i \in \omega$, then $\bigvee_{i\in\omega} d_i \lhd_\sigma M$.*

Proposition 6 *Let $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash M : \sigma$ be provable, let $M_i \in \mathcal{M}_{\sigma_i}$ for $1 \leq i \leq n$, and let $d_i \in [\![\sigma_i]\!]$ for $1 \leq i \leq n$ with $d_i \lhd_{\sigma_i} M_i$. Then it is the case that*

$$[\![x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash M : \sigma]\!](\vec{d}) \lhd_\sigma M[M_1, \ldots, M_n/x_1, \ldots, x_n]$$

where $\vec{d} \stackrel{\text{def}}{=} (d_1, \ldots, d_n) \in \Pi_1^n [\![\sigma_i]\!]$.

Proof

The proof proceeds by induction on the structure of M, and uses Lemma 4 and Lemma 5. We just give some example cases. We shall adopt the following convention: if (for example) $\Gamma \vdash M{:}\sigma$ is provable, we write $m{:}[\![\Gamma]\!] \to [\![\sigma]\!]$ for the morphism (here continuous function) $[\![\Gamma \vdash M{:}\sigma]\!]$. We shall also write (for example) $\tilde{M}$ for $M[M_1, \ldots, M_n/x_1, \ldots, x_n]$. We give two inductive cases:

(*Case M is* $\mathsf{It}(e.F, N)$): We need to prove that

$$[\![x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash \mathsf{It}(e.F, N){:}\sigma]\!] = it(f)(\vec{d}, n(\vec{d})) \lhd_\sigma \mathsf{It}(e.\tilde{F}, \tilde{N}),$$

where $f{:}\Pi_1^n [\![\sigma_i]\!] \times [\![\sigma]\!]_\perp \to [\![\sigma]\!]$ and thus $it(f){:}\Pi_1^n [\![\sigma_i]\!] \times \mathbb{N}^\infty \to [\![\sigma]\!]$. By induction we know that $n(\vec{d}) \lhd_{fix} \tilde{N}$ (1) and

$$e \lhd_{\sigma_\perp} E \text{ implies } f(\vec{d}, e) \lhd_\sigma \tilde{F}[E/e]. \tag{2}$$

We consider the case when $n(\vec{d})$ is $\check{n}+1 \in \mathbb{N}^\infty \setminus \{\infty\}$. We have $\check{n}+1 \lhd_{fix} \tilde{N}$ from (1), and so $\tilde{N} \Downarrow \mathsf{Inc}(E_{\check{n}})$, $E_{\check{n}} \Downarrow \mathsf{ValL}(N_{\check{n}})$ and so on to $N_0 \Downarrow \mathsf{Inc}(E)$. It can be shown that

$$it(f)(\vec{d}, 0) \lhd_\sigma \mathsf{It}(e.\tilde{F}, N_0).$$

Now let $0 \leq r \leq \check{n}$. Write $N_{\check{n}+1} \stackrel{\text{def}}{=} \tilde{N}$; we prove that if

$$it(f)(\vec{d}, r) \lhd_\sigma \mathsf{It}(e.\tilde{F}, N_r) \tag{3}$$

then $it(f)(\vec{d}, r+1) \lhd_\sigma \mathsf{It}(e.\tilde{F}, N_{r+1})$. From (3) and that $E_r \Downarrow \mathsf{ValL}(N_r)$ we may deduce

$$it(f)(\vec{d}, r+1) \lhd_\sigma \tilde{F}[\mathsf{LetL}\ y \Leftarrow E_r\ \mathsf{in}\ \mathsf{ValL}(\mathsf{It}(e.\tilde{F}, y))/e].$$

But certainly $N_{r+1} \rightarrow^* \mathsf{Inc}(E_r)$ and hence $\mathsf{It}(e.\tilde{F}, N_{r+1}) \rightarrow^* \tilde{F}[\mathsf{LetL}\ E_r\ \mathsf{in}\ \mathsf{ValL}(\mathsf{It}(e.\tilde{F}, y))/e]$, so we are done by appeal to Lemma 4.

(*Case M is* $\mathsf{EvalL}(M)$): We wish to prove that

$$[\![x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \vdash \mathsf{EvalL}(M){:}\sigma_\perp]\!](\vec{d}) = [m(\vec{d})] \lhd_{\sigma_\perp} \mathsf{EvalL}(\tilde{M})$$

and so we need to show that there is $V \in \mathcal{M}_\sigma^{can}$ for which $\mathsf{EvalL}(\tilde{M}) \Downarrow \mathsf{ValL}(V)$ with $m(\vec{d}) \lhd_\sigma V$. By the induction hypothesis we have $m(\vec{d}) \lhd_\sigma \tilde{M}$, and by inspecting the clauses of the logical relation one can deduce that $\tilde{M} \Downarrow V$ for some V with $m(\vec{d}) \lhd_\sigma V$ and so we are done. ■

Proof [*of Theorem 2*] Follows because the categorical semantics of $\mathcal{M}$ is sound, together with the application of Proposition 6 with $i = 1$ and $d_1 \equiv * \lhd_{unit} \langle\rangle \equiv M_1$. More precisely, we may deduce

$$[\![x{:}unit \vdash E{:}\sigma_\perp]\!](*) = [[\![x{:}unit \vdash M{:}\sigma]\!](*)] \lhd_{\sigma_\perp} E[\langle\rangle/x] \equiv E$$

and so $\exists V.E \Downarrow V$ by inspecting the clause defining the logical relation at lifted types. ■

Proof [*of Theorem 3*] Just as in the last proof, apply Proposition 6 to deduce that the relation $[\![\vdash M{:}\sigma]\!](*) \lhd_\sigma M$. By inspection of the clauses defining relation $\lhd_\sigma$ it will follow that evaluation of M converges. ■

5 The Object Language $\mathcal{O}$

The object language $\mathcal{O}$ is essentially a call-by-value form of PCF. Syntactically it is a tiny fragment of Standard ML. The $\mathcal{O}$-*types*, denoted by σ or τ, are generated from ground types bool of Booleans, int of numbers, by forming function types σ `->` τ. Let metavariable ℓ range over a set, $\mathbb{N} \cup \{tt, ff\}$, of *literals*, and metavariable $\oplus$ over a set of *operators*, $\{+, -, \times, =, <\}$. The $\mathcal{O}$-*terms*, e, are generated by the following BNF grammar, which also defines *canonical* $\mathcal{O}$-terms, c.

$$\begin{array}{lcl} e & ::= & c \mid e \underline{\oplus} e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid e\ e \\ c & ::= & x \mid \mathsf{()} \mid \underline{\ell} \mid \mathsf{fn}\, x \,\mathsf{=>}\, e \mid \mathsf{let\ fun}\, f\, x = e\, \mathsf{in}\, f\, \mathsf{end} \end{array}$$

The last canonical term is a form of SML notation for a recursively defined function, named f and with argument x, which are both bound in e.

The type assignment system for $\mathcal{O}$ consists of a collection of rules for proving judgments of the form $\Gamma \vdash e : \sigma$ where Γ is a finite list of (variable, type) pairs; we shall write $x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ for Γ. The typing rules for $\mathcal{O}$ are a straightforward

$$\frac{}{c \Downarrow c} \qquad \frac{e_1 \Downarrow \underline{\ell_1} \qquad e_2 \Downarrow \underline{\ell_2}}{e_1 \underline{\oplus} e_2 \Downarrow \underline{\ell_1 \oplus \ell_2}} \qquad \frac{e_1 \Downarrow \underline{tt} \qquad e_2 \Downarrow c}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow c} \qquad \frac{e_1 \Downarrow \underline{ff} \qquad e_3 \Downarrow c}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow c}$$

$$\frac{e_1 \Downarrow \text{fn } x \texttt{ => } e \qquad e_2 \Downarrow c_2 \qquad e[c_2/x] \Downarrow c}{e_1\ e_2 \Downarrow c}$$

$$\frac{e_1 \Downarrow \text{let fun } f\ x = e \text{ in } f \text{ end} \qquad e_2 \Downarrow c_2 \qquad e[\text{let fun } f\ x = e \text{ in } f \text{ end}, c_2/f, x] \Downarrow c}{e_1\ e_2 \Downarrow c}$$

Table 3: Operational semantics for $\mathcal{O}$

extension of those for simply typed lambda-calculus; we omit them all apart from the rule for recursively defined functions.

$$\frac{\Gamma, f{:}\sigma \texttt{ -> } \tau, x{:}\sigma \vdash e : \tau}{\Gamma \vdash \text{let fun } f\ x = e \text{ in } f \text{ end} : \sigma \texttt{ -> } \tau}$$

For each $\mathcal{O}$-type σ, let $\mathcal{O}_\sigma$ be the set of $\mathcal{O}$-terms e for which $\vdash e : \sigma$ is provable; such a term is known as a *program*. Let $\mathcal{O}_\sigma^{can}$ be the set of all programs from $\mathcal{O}_\sigma$ that are canonical.

The operational semantics of $\mathcal{O}$ is specified as an evaluation relation, consisting of 'big step' judgments of the form $e \Downarrow c$, where e and c are programs, the latter canonical. One should think of the canonical terms as values which are returned by (non-divergent) programs. The relation is given inductively by the rules in Table 3. One can deduce from the rules for forming such judgments that evaluation is deterministic and that if $e \in \mathcal{O}_\sigma$ and $e \Downarrow c$, then $c \in \mathcal{O}_\sigma^{can}$. If e is a program, write $e{\Downarrow}$ to mean $\exists c.\, e{\Downarrow}c$, in which case we say that program c *converges*.

The evaluation relation induces a Morris-style contextual equivalence between $\mathcal{O}$-terms: two terms are equivalent if each can replace the other in any program without changing its convergence behaviour. Informally, let a *context*, $\mathcal{C}[-]$, be a term some of whose subterms have been replaced by a *hole*, $-$, and let $\mathcal{C}[e]$ be the term obtained by filling each hole with the term e. Suppose that e_1 and e_2 are both members of one of the sets $\mathcal{O}_\sigma$. Write $e_1 \approx e_2$ to mean that $\mathcal{C}[e_1]{\Downarrow}$ iff $\mathcal{C}[e_2]{\Downarrow}$ whenever $\mathcal{C}[e_1]$, $\mathcal{C}[e_2]$ are both members of $\mathcal{O}_\tau$ for some type τ. In this case we say the two terms are *observationally equivalent*.

6 Translation of $\mathcal{O}$ into $\mathcal{M}$

In this section we give a denotational semantics for $\mathcal{O}$ using $\mathcal{M}$. The intention is that via this semantics, the metalogic can be applied to prove observational equivalences between $\mathcal{O}$-terms. This intention is vindicated by Theorem 12, the main result of the paper, which says that if the denotations of two $\mathcal{O}$-terms are provably equal in the metalogic, then in fact the two terms are observationally equivalent.

We adopt the usual inductive definition of numerals, $\lfloor 0 \rfloor \stackrel{\text{def}}{=} \mathsf{Z}$ and $\lfloor n+1 \rfloor \stackrel{\text{def}}{=} \mathsf{S}(\lfloor n \rfloor)$. Booleans are represented by terms of type $\mathit{unit} + \mathit{unit}$; let $\lfloor \mathit{tt} \rfloor \stackrel{\text{def}}{=} \mathsf{Inl}(\langle\rangle)$ and $\lfloor \mathit{ff} \rfloor \stackrel{\text{def}}{=} \mathsf{Inr}(\langle\rangle)$. For each arithmetic or relational operator, $\oplus \in \{+, -, \times, =, <\}$, we need an encoding $(M \lfloor + \rfloor N)$ in $\mathcal{M}$ such that suitably typed $(\lfloor \ell \rfloor \lfloor \oplus \rfloor \lfloor \ell' \rfloor)$ evaluates to $\lfloor \ell \oplus \ell' \rfloor$. Given the iteration on *nat* in $\mathcal{M}$, it is routine to do so; for instance, $(M \lfloor + \rfloor N)$ is $(x.\mathsf{S}(x))^N(M)$.

Using iterations on the *fix* type we can define an operation $\mathsf{Fix}(-)$, that we will need for recursively defining partial functions of the form $\sigma \rightharpoonup \tau$.

Lemma 7 *There is a term-former* $\mathsf{Fix}(-)$ *that satisfies the typing rule*

$$\frac{\Gamma \vdash M : (\sigma \rightharpoonup \tau) \to (\sigma \rightharpoonup \tau)}{\Gamma \vdash \mathsf{Fix}(M) : \sigma \rightharpoonup \tau}$$

and for any M *and* N, $\mathsf{Fix}(M)\, N \to^* M\, (\mathsf{Fix}(M))\, N$.

Proof Let $\mathsf{Fix}'(M, N) \stackrel{\text{def}}{=} \mathsf{It}(e.\, \lambda x.\, \mathsf{LetL}\ y \Leftarrow e\ \mathsf{in}\ M\, y\, x, N)$ and then set $\mathsf{Fix}(M) \stackrel{\text{def}}{=} \mathsf{Fix}'(M, \mathsf{Inc}(\omega))$. It is straightforward to check that $\mathsf{Fix}(M)$ has the expected type. From the calculation,

$$\begin{array}{rcl}
\mathsf{Fix}(M)\, N & \equiv & \mathsf{It}(e.\, \lambda x.\, \mathsf{LetL}\ y \Leftarrow e\ \mathsf{in}\ M\, y\, x, \mathsf{Inc}(\omega))\, N \\
& \to & (\lambda x.\, \mathsf{LetL}\ y \Leftarrow (\mathsf{LetL}\ z \Leftarrow \omega\ \mathsf{in}\ \mathsf{ValL}(\mathsf{Fix}'(M, z)))\ \mathsf{in}\ M\, y\, x)\, N \\
& \to & \mathsf{LetL}\ y \Leftarrow (\mathsf{LetL}\ z \Leftarrow \omega\ \mathsf{in}\ \mathsf{ValL}(\mathsf{Fix}'(M, z)))\ \mathsf{in}\ M\, y\, N \\
& \to & \mathsf{LetL}\ y \Leftarrow (\mathsf{LetL}\ z \Leftarrow \mathsf{ValL}(\mathsf{Inc}(\omega))\ \mathsf{in}\ \mathsf{ValL}(\mathsf{Fix}'(M, z)))\ \mathsf{in}\ M\, y\, N \\
& \to & \mathsf{LetL}\ y \Leftarrow \mathsf{ValL}(\mathsf{Fix}'(M, \mathsf{Inc}(\omega)))\ \mathsf{in}\ M\, y\, N \\
& \equiv & \mathsf{LetL}\ y \Leftarrow \mathsf{ValL}(\mathsf{Fix}(M))\ \mathsf{in}\ M\, y\, N \\
& \to & M\, (\mathsf{Fix}(M))\, N
\end{array}$$

one sees that $\mathsf{Fix}(M)$ has the expected reduction behaviour too. ■

We give the translation of $\mathcal{O}$ into $\mathcal{M}$ in terms of an arbitrary computational monad $(T, \mathsf{Val}, \mathsf{Let})$. In this section and the next this computational monad will be an instance of the lifting monad, $((-)_\perp, \mathsf{ValL}, \mathsf{LetL})$. In Section 8 we will use a different monad to admit the possibility of an exception being raised.

The translation of $\mathcal{O}$ into $\mathcal{M}$ follows the pattern set by Pitts [15]. Map each $\mathcal{O}$-type σ to an $\mathcal{M}$-type $[\![\sigma]\!]$ inductively as follows: ground types bool and int are mapped to $\mathit{unit} + \mathit{unit}$ and *nat* respectively, while σ `->` τ is mapped inductively to $[\![\sigma]\!] \to T[\![\tau]\!]$. Each $\mathcal{O}$-environment $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ is mapped to $[\![\Gamma]\!] \stackrel{\text{def}}{=} x_1{:}[\![\sigma_1]\!], \ldots, x_n{:}[\![\sigma_n]\!]$. To begin the translation of $\mathcal{O}$ terms into $\mathcal{M}$, to each canonical $\mathcal{O}$-term c we assign a canonical $\mathcal{M}$-term $|c|$ as follows.

$$\begin{array}{rcl}
|x| & \stackrel{\text{def}}{=} & x \\
|\underline{\ell}| & \stackrel{\text{def}}{=} & \lfloor \ell \rfloor \\
|\text{fn } x \texttt{ => } e| & \stackrel{\text{def}}{=} & \lambda x.\, [\![e]\!] \\
|\text{let fun } f\, x = e \text{ in } f \text{ end}| & \stackrel{\text{def}}{=} & \mathsf{Fix}(\lambda f.\, \lambda x.\, [\![e]\!])
\end{array}$$

In keeping with the intuition that arbitrary $\mathcal{O}$-terms express computations rather than simple values, we assign to each $\mathcal{O}$-term e an $\mathcal{M}$-term $[\![e]\!]$ of computation type.

$$\begin{array}{rcl} [\![c]\!] & \stackrel{\text{def}}{=} & \mathsf{Val}(|c|) \\ [\![e_1 \underline{\oplus} e_2]\!] & \stackrel{\text{def}}{=} & \mathsf{Let}\, x_1 \Leftarrow [\![e_1]\!]\, \mathsf{in}\, (\mathsf{Let}\, x_2 \Leftarrow [\![e_2]\!]\, \mathsf{in}\, \mathsf{EvalL}(x_1 \lfloor \oplus \rfloor x_2)) \\ [\![\mathrm{if}\, e_1\, \mathrm{then}\, e_2\, \mathrm{else}\, e_3]\!] & \stackrel{\text{def}}{=} & \mathsf{Let}\, b \Leftarrow [\![e_1]\!]\, \mathsf{in}\, \mathsf{Case}(b, u.\, [\![e_1]\!], u.\, [\![e_1]\!]) \\ [\![e_1\, e_2]\!] & \stackrel{\text{def}}{=} & \mathsf{Let}\, f \Leftarrow [\![e_1]\!]\, \mathsf{in}\, (\mathsf{Let}\, x \Leftarrow [\![e_2]\!]\, \mathsf{in}\, f\, x) \end{array}$$

The translation respects the type systems of $\mathcal{O}$ and $\mathcal{M}$ in the following sense, easily proved by structural induction.

Proposition 8 (Static Adequacy)

(1) *Whenever* $\Gamma \vdash e : \sigma$ *is provable in* $\mathcal{O}$*, then* $[\![\Gamma]\!] \vdash [\![e]\!] : T[\![\sigma]\!]$ *is provable in* $\mathcal{M}$*.*

(2) *Furthermore, if* e *is canonical, then* $[\![e]\!]$ *is canonical and* $[\![\Gamma]\!] \vdash |e| : [\![\sigma]\!]$ *is provable in* $\mathcal{M}$*.*

Since the translation is compositional it is straightforward to prove the congruence and substitution parts of the following lemma by structural induction.

Lemma 9

(1) *If* $\Gamma \vdash [\![e_1]\!] = [\![e_2]\!] : \sigma$*,* $\Gamma' \vdash [\![\mathcal{C}[e_1]]\!] : \tau$ *and* $\Gamma' \vdash [\![\mathcal{C}[e_2]]\!] : \tau$ *are provable in* $\mathcal{M}$*,*
then $\Gamma' \vdash [\![\mathcal{C}[e_1]]\!] = [\![\mathcal{C}[e_2]]\!] : \tau$ *is provable in* $\mathcal{M}$ *too.*

(2) *Furthermore, for any* $\mathcal{O}$*-term* e *and canonical* $\mathcal{O}$*-term* c*,* $[\![e]\!][|c|/x] \equiv [\![e[c/x]]\!]$.

We can establish the following exact correspondence between evaluation of any configuration and its translation by rule inductions and appeal to Lemmas 7 and 9.

Lemma 10 *Suppose* $e \in \mathcal{O}_\sigma$*.*

(1) *Whenever* $e \Downarrow c$ *for some* $c \in \mathcal{O}_\sigma^{can}$*, then* $[\![e]\!] \Downarrow [\![c]\!]$*.*

(2) *Whenever* $[\![e]\!] \Downarrow V$ *for some* $V \in \mathcal{M}_{T[\![\sigma]\!]}^{can}$*, there is* c *with* $e \Downarrow c$ *and* $V \equiv [\![c]\!]$*.*

Now we can prove adequacy for $\mathcal{O}$. This is the crux of the factorisation, where adequacy for $\mathcal{M}$—obtained from a domain-theoretic logical relations argument—is combined with the previous correspondence lemma—obtained by comparatively routine albeit detailed syntactic calculations.

Proposition 11 (Dynamic Adequacy) *Suppose* $e \in \mathcal{O}_\sigma$ *and* $c \in \mathcal{O}_\sigma^{can}$*.*

(1) *Whenever* $e \Downarrow c$ *then* $\vdash [\![e]\!] = [\![c]\!] : T[\![\sigma]\!]$ *is provable.*

(2) *Whenever* $\vdash [\![e]\!] = [\![c]\!] : T[\![\sigma]\!]$ *is provable, then* $e \Downarrow$*.*

$$\frac{e_1 \Downarrow \mathrm{fn}\, x \texttt{=>} e \qquad e[e_2/x] \Downarrow c}{e_1\, e_2 \Downarrow c}$$

$$\frac{e_1 \Downarrow \mathrm{let\,fun}\, f\, x = e\, \mathrm{in}\, f\, \mathrm{end} \qquad e[\mathrm{let\,fun}\, f\, x = e\, \mathrm{in}\, f\, \mathrm{end}, e_2/f, x] \Downarrow c}{e_1\, e_2 \Downarrow c}$$

Table 4: Rules for call-by-name function application in $\mathcal{O}$

Proof (1) Suppose $e \Downarrow c$. By Lemma 10(1) we have $[\![e]\!] \Downarrow [\![c]\!]$. Hence we have $\vdash [\![e]\!] = [\![c]\!] : T[\![\sigma]\!]$ by Proposition 1. (2) Suppose $\vdash [\![e]\!] = [\![c]\!] : T[\![\sigma]\!]$. From Proposition 8(2) we know that $[\![c]\!]$ is canonical, so by Theorem 2, there is V such that $[\![e]\!] \Downarrow V$. Then by Lemma 10(2) there is c such that that $e \Downarrow c$ as required. ■

We now obtain soundness from adequacy as usual [8].

Theorem 12 (Soundness) *If* $e_1, e_2 \in \mathcal{O}_\sigma$ *and* $\vdash [\![e_1]\!] = [\![e_2]\!] : T[\![\sigma]\!]$ *then* $e_1 \approx e_2$.

Proof Suppose $\vdash [\![e_1]\!] = [\![e_2]\!] : T[\![\sigma]\!]$. We are to show for all contexts $\mathcal{C}[-]$ such that $\mathcal{C}[e_1]$ and $\mathcal{C}[e_2]$ are both $\mathcal{O}$-programs of some type τ, that $\mathcal{C}[e_1]\Downarrow$ iff $\mathcal{C}[e_2]\Downarrow$. We show the forwards direction; the reverse follows by symmetry. Suppose then that each $\mathcal{C}[e_i] \in \mathcal{O}_\tau$, and that $\mathcal{C}[e_1] \Downarrow c$ for some canonical $c \in \mathcal{O}_\tau$. By Proposition 11(1), we have $\vdash [\![\mathcal{C}[e_1]]\!] = [\![c]\!] : T[\![\tau]\!]$. By Lemma 9(1) we have $\vdash [\![\mathcal{C}[e_1]]\!] = [\![\mathcal{C}[e_2]]\!] : T[\![\tau]\!]$. By transitivity and symmetry, $\vdash [\![\mathcal{C}[e_2]]\!] = [\![c]\!]$, and then since c is canonical $\mathcal{C}[e_2]\Downarrow$ by Proposition 11(2), as required. ■

7 Variant 1: call-by-name $\mathcal{O}$

In this and the following section we show how our modular proof of adequacy can easily be modified to apply to two variants of the object language $\mathcal{O}$.

The first of these is a form of $\mathcal{O}$ in which functions are applied using a call-by-name instead of a call-by-value strategy. The syntax and type system of $\mathcal{O}$ is unchanged except that variables are no longer included among the canonical $\mathcal{O}$-terms.

$$c ::= \underline{\ell} \mid \mathrm{fn}\, x \texttt{=>} e \mid \mathrm{let\,fun}\, f\, x = e\, \mathrm{in}\, f\, \mathrm{end}.$$

The evaluation relation, $e \Downarrow c$, is given inductively by the original rules from Table 3, except that the two rules for the two kinds of function application are replaced by call-by-name forms in Table 4. One can easily deduce that evaluation is deterministic and preserves types as before. We adopt the same notions of program convergence and observational equivalence as before.

We must also modify the translation of $\mathcal{O}$ into $\mathcal{M}$. As before, ground types bool and int are mapped to $unit + unit$ and nat respectively, but this time each (call-by-name) function type $\sigma \texttt{->} \tau$ is mapped to $T[\![\sigma]\!] \texttt{->} T[\![\tau]\!]$ (instead of

$[\![\sigma]\!]$ -> $T[\![\tau]\!]$ in the call-by-value case). A call-by-name strategy applies functions to computations rather than values. Each $\mathcal{O}$-environment $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$ is mapped to $\mathcal{M}$-environment $[\![\Gamma]\!] \stackrel{\text{def}}{=} x_1{:}T[\![\sigma_1]\!], \ldots, x_n{:}T[\![\sigma_n]\!]$. The mapping of environments reflects a change in denotation of object variables—in the call-by-value setting they denoted values; here they denote computations. The translations of terms are the same as before except for the following changes. The rules for recursive functions and applications are changed, the rule for canonical variables is dropped, and a new one for non-canonical variables is introduced.

$$\begin{array}{rcl} |\text{let fun } f\, x = e \text{ in } f \text{ end}| & \stackrel{\text{def}}{=} & \mathsf{Fix}(\lambda g.\, \lambda x.\, (\lambda f.\, [\![e]\!])(\mathsf{Val}(g))) \\ [\![x]\!] & \stackrel{\text{def}}{=} & x \\ [\![e_1\, e_2]\!] & \stackrel{\text{def}}{=} & \mathsf{Let}\ f \Leftarrow [\![e_1]\!]\ \mathsf{in}\ f\, [\![e_2]\!] \end{array}$$

The third of these equations effects the call-by-name strategy, while the remaining two reflect the change in denotation of object variables—from values to computations.

Given the change in translation of environments, the following static adequacy result is easily proved.

Proposition 13 (Static Adequacy)

(1) *Whenever* $\Gamma \vdash e : \sigma$ *is provable in* $\mathcal{O}$*, then* $[\![\Gamma]\!] \vdash [\![e]\!] : T[\![\sigma]\!]$ *is provable in* $\mathcal{M}$*.*

(2) *Furthermore, if* e *is canonical, then* $[\![e]\!]$ *is canonical and* $[\![\Gamma]\!] \vdash |e| : [\![\sigma]\!]$ *is provable in* $\mathcal{M}$*.*

The congruence lemma is stated and proved as before, but the substitution lemma this time concerns the substitution of arbitrary $\mathcal{O}$-terms, rather than simply canonical ones.

Lemma 14

(1) *If* $\Gamma \vdash [\![e_1]\!] = [\![e_2]\!] : \sigma$*,* $\Gamma' \vdash [\![\mathcal{C}[e_1]]\!] : \tau$ *and* $\Gamma' \vdash [\![\mathcal{C}[e_2]]\!] : \tau$ *are provable in* $\mathcal{M}$*,*
then $\Gamma' \vdash [\![\mathcal{C}[e_1]]\!] = [\![\mathcal{C}[e_2]]\!] : \tau$ *is provable in* $\mathcal{M}$ *too.*

(2) *Furthermore, for any* $\mathcal{O}$*-terms* e *and* e'*,* $[\![e]\!][[\![e']\!]/x] \equiv [\![e[e'/x]]\!]$*.*

The correspondence between evaluation of an $\mathcal{O}$-program and its denotation is proved much as before, by a detailed but routine rule induction and appeal to the previous lemmas.

Lemma 15 *Suppose* $e \in \mathcal{O}_\sigma$*.*

(1) *Whenever* $e \Downarrow c$ *for some* $c \in \mathcal{O}_\sigma^{can}$*, then* $[\![e]\!] \Downarrow [\![c]\!]$*.*

(2) *Whenever* $[\![e]\!] \Downarrow V$ *for some* $V \in \mathcal{M}_{[\![\sigma]\!]}^{can}$*, there is* c *with* $e \Downarrow c$ *and* $V \equiv [\![c]\!]$*.*

$$\frac{e_1 \Downarrow \text{wrong}}{e_1 \underline{\oplus} e_2 \Downarrow \text{wrong}} \qquad \frac{e_1 \Downarrow \text{wrong}}{e_1\ e_2 \Downarrow \text{wrong}} \qquad \frac{e_1 \Downarrow \text{wrong}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow \text{wrong}}$$

$$\frac{e_1 \Downarrow \underline{\ell} \qquad e_2 \Downarrow \text{wrong}}{e_1 \underline{\oplus} e_2 \Downarrow \text{wrong}} \qquad \frac{e_1 \Downarrow c \qquad c \not\equiv \text{wrong} \qquad e_2 \Downarrow \text{wrong}}{e_1\ e_2 \Downarrow \text{wrong}}$$

Table 5: Rules for a single exception in $\mathcal{O}$

Once these detailed calculations are complete the soundness theorem follows exactly as before.

Theorem 16 (Soundness) *If* $e_1, e_2 \in \mathcal{O}_\sigma$ *and* $\vdash [\![e_1]\!] = [\![e_2]\!] : T[\![\sigma]\!]$ *then* $e_1 \approx e_2$.

Proof From Lemma 15 by the same arguments that established Proposition 11 and Theorem 12 in the call-by-value case. ■

8 Variant 2: $\mathcal{O}$ plus an exception

Our second variant consists in adding a single exception, `wrong`, to the original call-by-value $\mathcal{O}$. The point of this variant is that our methods work for computational monads other than the lifting monad.

We introduce a new canonical expression, `wrong`, and extend the typing relation so that judgment $\Gamma \vdash \text{wrong} : \sigma$ is provable for any type σ and (well-formed) environment Γ. The evaluation relation is generated from the original call-by-value rules in Table 3—amended with the side-condition on both rules for function application that $c_2 \not\equiv \text{wrong}$—together with the new rules in Table 5. Convergence and observational equivalence are defined as before.

To give the translation of $\mathcal{O}$ into $\mathcal{M}$, we shall need a new computational monad $(T, \mathsf{Val}, \mathsf{Let})$. Intuitively, computations may either converge to a value, diverge, or go wrong, that is, converge to the exceptional value `wrong`. To model such computations set $T\sigma \stackrel{\text{def}}{=} (\sigma + 1)_\perp$. A computation that goes wrong will be modelled by $\mathsf{Wrong} \stackrel{\text{def}}{=} \mathsf{ValL}(\mathsf{Inr}(\langle\rangle))$. The terms of the computational monad are defined by

$$\begin{array}{rcl} \mathsf{Val}(M) & \stackrel{\text{def}}{=} & \mathsf{ValL}(\mathsf{Inl}(M)) \\ \mathsf{Let}\, x \Leftarrow M \,\mathsf{in}\, N & \stackrel{\text{def}}{=} & \mathsf{LetL}\, y \Leftarrow M \,\mathsf{in}\, \mathsf{Case}(y, x.\, M, u.\, \mathsf{Wrong}). \end{array}$$

Given the metalogic's coproduct rules [3, 4], it is routine to check that this is indeed a well defined computational monad in the sense given earlier.

Apart from the re-interpretation of the triple $(T, \mathsf{Val}, \mathsf{Let})$ there are two other changes needed to the translation of $\mathcal{O}$ into $\mathcal{M}$. First, we need to change the translation of canonical terms, $[\![c]\!]$, to the following.

$$[\![c]\!] \stackrel{\text{def}}{=} \begin{cases} \mathsf{Wrong} & \text{if } c \equiv \text{wrong} \\ \mathsf{Val}(|c|) & \text{otherwise} \end{cases}$$

Note that the interpretation of each canonical term $c \in \mathcal{O}_\sigma$ as a value $|c| \in \mathcal{O}_{[\![\sigma]\!]}$ does not need to be extended to `wrong`—and in fact would not make sense. Second, we need to change the translation of $[\![e_1 \underline{\oplus} e_2]\!]$ to the following,

$$[\![e_1 \underline{\oplus} e_2]\!] \stackrel{\text{def}}{=} \mathsf{Let}\, x_1 \Leftarrow [\![e_1]\!]\, \mathsf{in}\, (\mathsf{Let}\, x_2 \Leftarrow [\![e_2]\!]\, \mathsf{in}\, \mathsf{Eval}(x_1 \underline{\oplus} x_2))$$

where $\mathsf{Eval}(M) \stackrel{\text{def}}{=} \mathsf{LetL}\, y \Leftarrow \mathsf{EvalL}(M)\, \mathsf{in}\, \mathsf{Val}(y)$ for any M. There is no need to re-interpret the definition of recursive functions using Fix because although the definition of $T\sigma$ is different, the interpretation of an $\mathcal{O}$-function σ `->` τ is still a partial function, this time $[\![\sigma]\!] \rightharpoonup [\![\tau]\!] + 1$.

Given these modifications to the denotational semantics, the soundness argument goes through much as before. Static adequacy (Proposition 8) holds as before, except that the judgment in part (2) holds only when $e \not\equiv$ wrong. Compositionality, Lemma 9(1) holds as before, and so does the second part, substitution, except again for a side-condition that $c \not\equiv$ wrong—but the side-condition on function applications ensures that `wrong` is never substituted for a variable in the operational semantics. The correspondence between the evaluation of each $\mathcal{O}$-program and its denotation, Lemma 10, holds as before. The proof requires the easily verified facts that

$$\begin{array}{rcl} \mathsf{Let}\, x \Leftarrow \mathsf{Val}(M)\, \mathsf{in}\, N & \to^+ & N[^M\!/_x] \\ \mathsf{Let}\, x \Leftarrow \mathsf{Wrong}\, \mathsf{in}\, M & \to^+ & \mathsf{Wrong} \\ \mathsf{Eval}(\lfloor \ell \rfloor \lfloor \oplus \rfloor \lfloor \ell' \rfloor) & \to^+ & \mathsf{Val}(\lfloor \ell \oplus \ell \rfloor) \end{array}$$

for suitably-typed terms M, N and literals ℓ and ℓ' (where $\to^+$ is the transitive closure of $\to$). Given these lemmas, dynamic adequacy and soundness (Proposition 11 and Theorem 12) follow exactly as before.

9 Related and Future Work

This paper has shown how adequacy (and hence soundness) for a call-by-value object language $\mathcal{O}$ may be factored into two parts: a (non-routine) adequacy proof for a metalogic coupled with a comparatively routine proof of correspondence between the evaluation of each object program and its denotation in the metalogic. The factorisation is of interest because the second part can easily be adapted to other object languages without needing to repeat the first part. Variants of $\mathcal{O}$ with call-by-name functions and with a simple exception mechanism illustrated this genericity.

Apart from the unpublished notes [17, Chapter 3] that inspired this reworking, the only previous work to factor an adequacy proof via a metalogic is in Gordon's dissertation [5]. There the meaning of the metalogic was given using Abramsky's applicative bisimulation rather than domain theoretically. Others have equipped a metalogic with an operational semantics [1, 6, 7, 8, 9, 14, 20]

but none has reworked Plotkin's original factorisation. Crole presents unfactored adequacy proofs for two PCF style languages mapped into the FIX-logic [2]. Apart from their application to denotational semantics, monads have been popularised by Wadler and others as a way to incorporate imperative features into lazy functional programming [13, 18, 19].

The key feature of this paper is the reworking of Plaotkin's idea for factoring adequacy, but within a more structured and foundational framework; nonetheless the absence of recursive types from $\mathcal{M}$ prohibits modelling of many object language features. Hence in future work we intend to extend $\mathcal{M}$ with recursive types; Pitts' recent advances [16] will be highly relevant to extending Theorem 2. A further goal is to mechanise some form of $\mathcal{M}$ in a theorem-prover, and hence take advantage of this paper's soundness result to prove operational equivalences of $\mathcal{O}$-programs mechanically. Although space has prevented its full exposition here, application of $\mathcal{M}$ to verification of functional programs was the specific motivation for the theory in this paper.

Acknowledgements We wish to thank everyone at the Ayr workshop who discussed this work, and to the referees for their detailed comments. Roy Crole holds a SERC Research Fellowship at Imperial College. During this work Andrew Gordon was a member of the Programming Methodology Group in Chalmers University of Technology, Gothenburg. The work was begun while we were visitors at the University of Cambridge Computer Laboratory. We thank Andrew Pitts, our host, and everyone else at the Lab for their hospitality. Roy Crole thanks the SERC and the CEC CLICS project for providing funding to visit Cambridge and Gothenburg. Andrew Gordon thanks Mary Sheeran for arranging his visit to the PMG.

Crole's address: Imperial College, Department of Computing, Huxley Building, 180 Queen's Gate, London SW7 2BZ, United Kingdom. `rlc@@doc.ic.ac.uk`. Gordon's address: University of Cambridge Computer Laboratory, New Museums Site, Cambridge CB2 3QG, United Kingdom. `adg@@cl.cam.ac.uk`.

References

[1] Peter Nicholas Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, University of Cambridge Computer Laboratory, August 1993. Available as Technical Report 309.

[2] R. L. Crole. Computational adequacy for the FIX-Logic. *Theoretical Computer Science.* Accepted. (To appear in 1994.).

[3] 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. Earlier version in LICS'90.

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

[5] Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, August 1992. To appear in Cambridge University Press' series Distinguished Dissertations in Computer Science.

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

[7] Claire Jones. *Probabilistic Non-determinism*. PhD thesis, University of Edinburgh, 1990. Available as Technical Report CST–63–90, Computer Science Department, University of Edinburgh.

[8] Albert R. Meyer and Stavros S. Cosmadakis. Semantical paradigms: Notes for an invited lecture. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 236–253, July 1988.

[9] Eugenio Moggi. *The Partial Lambda-Calculus*. PhD thesis, Department of Computer Science, University of Edinburgh, August 1988. Available as Technical report CST–53–88.

[10] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, June 1989.

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

[12] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *The International Series of Monographs in Computer Science*. Clarendon Press, Oxford, 1990.

[13] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*. ACM Press, 1993.

[14] Andrew M. Pitts. Notes on the call-by-value and call-by-name translation of the simply typed lambda-calculus into the computational lambda-calculus. Manuscript, October 1990.

[15] Andrew M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer-Verlag, 1991. Available as University of Cambridge Computer Laboratory Technical Report 198, August 1990.

[16] Andrew M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *MFPS IX, New Orleans*, 1993.

[17] Gordon D. Plotkin. Denotational semantics with partial functions. Unpublished lecture notes, CSLI, Stanford University, July 1985.

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

[19] Philip Wadler. The essence of functional programming. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, 1992.

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