An Observationally Complete Program Logic for Imperative Higher-Order Functions

Kohei Honda[1] Nobuko Yoshida[2] Martin Berger[3]

[1] Department of Computer Science, Queen Mary, University of London
[2] Department of Computing, Imperial College London
[3] Department of Informatics, University of Sussex

Abstract. We establish a strong completeness property called *observational completeness* of the program logic for imperative, higher-order functions introduced in [40]. Observational completeness states that valid assertions characterise program behaviour up to observational congruence, giving a precise correspondence between operational and axiomatic semantics. The proof layout for the observational completeness which uses a restricted syntactic structure called finite canonical forms originally introduced in game-based semantics, and characteristic formulae originally introduced in the process calculi, is generally applicable for a precise axiomatic characterisation of more complex program behaviour, such as aliasing and local state.

1 Introduction

Imperative higher-order functions, syntactically embodied by imperative extensions of the λ-calculus, have been one of the major topics in the study of semantics and types of programming languages. They are a cornerstone of richly typed languages such as ML [57] and Haskell [3], and are central to the semantic analysis of procedural, object-oriented and low-level languages [1, 49, 58, 65, 73]. The significance of combining imperative features and higher-order functions lies in their distilled presentation of key elements of sequential program behaviour, making them amenable to theoretical analysis. This analytical nature contributes to semantic studies [48, 57, 67], type-theoretic studies [57, 65] and the study of operational reasoning techniques [53, 66]. Imperative higher-order functions also enjoy rich computational behaviour, e.g. stored higher-order functions which can encode general recursion, cf. [48].

In Hoare logic [21, 31, 60], assertions on programs describe program properties independent of the latter's textual details, with proof rules, based on the the syntactic structure of programs, enabling verification of valid assertions. Hoare logic enjoys a clear observational basis, which we may call *observational completeness*: the set of pre/post conditions precisely characterise observable properties of programs. Thus specifications in Hoare logic capture no more and no less than the observational behaviour of programs, and we can compare two programs which satisfy the same specification but differ in efficiency, modularity, and other intensional features. This paper formalises and proves the observational completeness in a program logic for imperative higher-order functions introduced in [40]. This solves the long-standing open problem of matching axiomatic with operational semantics for this important class of imperative higher-order functional behaviour.

As a target programming language, we consider call-by-value PCF with global references, where references can store higher-order procedures. This language already exhibits all key problems arising in matching axiomatic with operational semantics for languages with higher-order state. The language and the logic presented in this paper can be extended to those with aliasing and local references [10, 79].

To our knowledge, this is the first time observational completeness is obtained for imperative higher-order functions in full type hierarchy, accommodating stored higher-order procedures. It is also the first time that proof techniques used to establish observational completeness appear in the literature.

Two examples of programs expressible with imperative higher-order functions

To introduce the logic used in this paper, we present two simple programs that exhibit the expressive power of our chosen programming language (we use notations from standard textbooks [25, 65]).

$$\begin{array}{lll}\texttt{closureFact} & \stackrel{\text{def}}{=} \mu f^{\mathsf{Nat}\Rightarrow\mathsf{Unit}}.\lambda x^{\mathsf{Nat}}. & \texttt{if}\ x=0\ \texttt{then}\ y:=\lambda().1 \\ & & \texttt{else}\ \ y:=\lambda().\,(f(x-1)\,;x\times(!y)()\,)\end{array}$$

Above $()$ is the unique constant of type Unit; while $\lambda().N$ denotes $\lambda z^{\mathsf{Unit}}.N$ with z fresh. When invoked as e.g. `closureFact` 3, the program stores a procedure in the imperative variable y. If we then invoke this stored procedure as $(!y)()$, then `closureFact` is called again with the argument $3-1=2$, after which a program stored in y *at that time* is invoked, so that the multiple of x and the value returned by that program is calculated and is given as the final return value. The intention of the program is that this final value should be the factorial of 3. The observable behaviour of `closureFact` can be informally described as follows.

When the program is fed with a number n, it stores in y a closure which, when invoked with $()$*, will return the factorial of n.*

Note that inside the body of `closureFact`, a free variable f and the content of an imperative variable y are used non-trivially. In particular, the correctness of this program crucially depends on how y is updated sequentially in an orderly manner.

Next we consider another nonstandard, more terse factorial program, using Landin's idea [48] to realise a recursion by circular references ("recursion through the store").

$$\texttt{circFact} \stackrel{\text{def}}{=} x := \lambda z.\texttt{if}\ z=0\ \texttt{then}\ 1\ \texttt{else}\ z\times(!x)(z-1)$$

After executing `circFact`, $(!x)n$ returns the factorial of n. But specifying the content of x this way does not give a full description of its behaviour, since x is still free so that the functionality of a procedure in question, the factorial, depends on the state of x (for example, if a program reads from x and store it in another variable, say y, assigns a diverging function to x, and feeds the content of y with 3, then the program diverges rather than returning 6). Taking care of this aspect, the state after executing `circFact` may be informally described thus:

x stores a procedure which computes the factorial of its argument using a procedure stored in x: that procedure should calculate the factorial, and x does store that procedure.

Note the inherent circularity of this description — how can we logically describe such a behaviour, and how can we derive it compositionally?

Technical Contributions and Outline

The following summarises the main technical contributions of this paper.

- Establishment of sound and complete characterisation of observational equivalence by the logic, using a restricted syntactic structure called *finite canonical forms* originally introduced in the study of game-based semantics [6, 38, 42].
- Derivation of characteristic formulae of finite canonical forms with respect to total correctness using our proof rules. By reducing differentiating contexts of two observationally distinct programs to finite canonical forms, and further to their characteristic formulae, we show any semantically distinct programs can be differentiated by an assertion, leading to the characterisation of the observational congruence by logical validity.

In the remainder, Section 2 introduces the target language and logic used in the paper. Section 3 illustrates compositional proof rules for the logic. Section 4 establishes soundness of proof rules and proves the observational completeness. Section 5 presents a few abridged reasoning examples. Section 6 discusses related work.

2 Logic for Imperative Call-by-Value PCF

2.1 Imperative PCF

This subsection briefly reviews call-by-value PCF, the programming language we use in the present study. We augment the language with unit, sums and products, and with imperative variables, henceforth called *references*. The grammar of programs is standard [65], given below, assuming given an infinite set of *variables* ($x, y, z, \ldots$, also called *names*).

$$\begin{array}{ll} \text{(value)} & V, W ::= \mathtt{c} \mid x \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M \mid \langle V, W \rangle \mid \mathtt{in}_i(V) \\ \text{(program)} & M, N ::= V \mid MN \mid x := N \mid !x \mid \mathtt{op}(\vec{M}) \mid \pi_i(M) \mid \langle M, N \rangle \mid \mathtt{in}_i(M) \\ & \qquad \mid \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \mid \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}} \end{array}$$

The grammar uses types ($\alpha, \beta, \ldots$), given later. Binding is standard and $\mathsf{fv}(M)$ denotes the set of free variables. Types annotating bound variables are often omitted. Constants ($\mathtt{c}, \mathtt{c}', \ldots$) include the unit $()$, natural numbers $\mathtt{n}$ and booleans $\mathtt{b}$ (either truth $\mathtt{t}$ or false $\mathtt{f}$). As usual $\mathtt{op}(\vec{M})$ (where $\vec{M}$ is a vector of programs) is a standard n-ary arithmetic or boolean operation, e.g. $+$, $-$, $\times$, $=$ (equality of two numbers), $\neg$ (negation), $\wedge$ and $\vee$. Dereferencing a variable x is written $!x$, and assignments of the form $x := N$.

The operational semantics of the language is given by the standard call-by-value reduction rules with stores [25, 65]. A *store* $(\sigma, \sigma', ...)$ is a finite map from references to values. A *configuration* is a pair of a program and a store. The reduction is the binary relation over configurations, written $(M, \sigma) \longrightarrow (M', \sigma')$, generated by the following rules [25, 65].

$$
\begin{aligned}
((\lambda x.M)V, \sigma) &\to (M[V/x], \sigma) && (2.1)\\
(\pi_1(\langle V_1, V_2\rangle), \sigma) &\to (V_1, \sigma) && (2.2)\\
(\mathtt{case}\ \mathtt{in}_1(W)\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}}, \sigma) &\to (M_1[W/x_1], \sigma) && (2.3)\\
(\mathtt{if}\ \mathtt{t}\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2, \sigma) &\to (M_1, \sigma) && (2.4)\\
((\mu f.\lambda g.N)W, \sigma) &\to (N[W/g][\mu f.\lambda g.N/f], \sigma) && (2.5)\\
(!x,\ \sigma) &\to (\sigma(x),\ \sigma) && (2.6)\\
(x := V,\ \sigma) &\to ((),\ \sigma[x \mapsto V]) && (2.7)
\end{aligned}
$$

(2.1–2.5) are from call-by-value PCF and do not involve the store (we omit obvious symmetric rules and the rules for first-order operators). (2.6) and (2.7) are for imperative constructs, assuming $x \in \mathsf{dom}(\sigma)$ in both. In (2.7), $\sigma[x \mapsto V]$ denotes the store which maps x to V and otherwise agrees with σ. Finally we have the contextual rule:

$$(\mathcal{E}[M], \sigma) \to (\mathcal{E}[M'], \sigma') \quad \textit{if} \quad (M, \sigma) \to (M', \sigma') \qquad (2.8)$$

where $\mathcal{E}[\cdot]$ ranges over left-to-right eager evaluation contexts, given by:

$$
\begin{aligned}
\mathcal{E}[\cdot] ::= {} & (\mathcal{E}[\cdot]M) \mid (V\mathcal{E}[\cdot]) \mid \mathtt{op}(\vec{V}, \mathcal{E}[\cdot], \vec{M}) \mid \pi_i(\mathcal{E}[\cdot]) \mid \mathtt{in}_i(\mathcal{E}[\cdot]) \mid !\mathcal{E}[\cdot]\\
& \mid x := \mathcal{E}[\cdot] \mid \mathtt{if}\ \mathcal{E}[\cdot]\ \mathtt{then}\ M\ \mathtt{else}\ N \mid \mathtt{case}\ \mathcal{E}[\cdot]\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}}
\end{aligned}
$$

We write $(M, \sigma) \Downarrow (V, \sigma')$ iff $(M, \sigma) \longrightarrow^* (V, \sigma')$, $(M, \sigma) \Downarrow$ iff $(M, \sigma) \Downarrow (V, \sigma')$ for some V and σ', and $(M, \sigma) \Uparrow$ iff $(M, \sigma) \longrightarrow^n$ for numeral n.

Types and typing rules. Types [25, 65] are given by the following grammar.

$$
\begin{aligned}
\alpha, \beta \quad &::= \quad \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta\\
\rho \quad &::= \quad \alpha \mid \mathsf{Ref}(\alpha)
\end{aligned}
$$

We call $\alpha, \beta, \ldots$ *value types*, and $\mathsf{Ref}(\alpha), \ldots$ *reference types*. A *basis* is a finite map from names to types. $\Gamma, \Gamma'\ldots.$ range over bases whose codomains are value types (which we sometimes call *environment basis*), while $\Delta, \Delta', \ldots$ range over bases whose codomains are reference types (which we sometimes call *reference basis*). $\mathsf{dom}(\Gamma)$ (resp. $\mathsf{dom}(\Delta)$) denotes the domain of Γ (resp. of Δ).

The typing rules use the sequent $\Gamma; \Delta \vdash M : \alpha$ ("M has type α under Γ and Δ"), and are standard [65], listed in Figure 1. In $\Gamma; \Delta \vdash M : \alpha$, we always assume $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. We extend typing to stores, writing $\Gamma; \Delta \vdash \sigma$ if: $\mathsf{dom}(\Delta) = \mathsf{dom}(\sigma)$ and $\Gamma; \Delta \vdash \sigma(x) : \alpha$ iff $\Delta(x) = \mathsf{Ref}(\alpha)$, for each $x \in \mathsf{dom}(\sigma)$. A configuration (M, σ) is *well-typed* if we have $\Gamma; \Delta \vdash M : \alpha$ and $\Gamma; \Delta \vdash \sigma$ for some Γ and Δ (if so and if $(M, \sigma) \longrightarrow (M', \sigma')$, we also have $\Gamma; \Delta \vdash M' : \alpha$ and $\Gamma; \Delta \vdash \sigma'$ by subject reduction).

Fig. 1 Typing rules for the core language

$$[\mathit{Var}]\ \frac{\Gamma(x)=\alpha}{\Gamma;\Delta\vdash x:\alpha}\qquad [\mathit{Unit}]\ \frac{-}{\Gamma;\Delta\vdash ():\mathsf{Unit}}\qquad [\mathit{Bool}]\ \frac{-}{\Gamma;\Delta\vdash \mathtt{b}:\mathsf{Bool}}\qquad [\mathit{Num}]\ \frac{-}{\Gamma;\Delta\vdash \mathtt{n}:\mathsf{Nat}}$$

$$[\mathit{Eq}]\ \frac{\Gamma;\Delta\vdash M_{1,2}:\mathsf{Nat}}{\Gamma;\Delta\vdash M_1=M_2:\mathsf{Bool}}\qquad [\mathit{Abs}]\ \frac{\Gamma,x\!:\!\alpha;\Delta\vdash M:\beta}{\Gamma;\Delta\vdash \lambda x^{\alpha}.M:\alpha\Rightarrow\beta}\qquad [\mathit{Rec}]\ \frac{\Gamma,x\!:\!\alpha\Rightarrow\beta;\Delta\vdash \lambda y^{\alpha}.M:\alpha\Rightarrow\beta}{\Gamma;\Delta\vdash \mu x^{\alpha\Rightarrow\beta}.\lambda y^{\alpha}.M:\alpha\Rightarrow\beta}$$

$$[\mathit{App}]\ \frac{\Gamma;\Delta\vdash M:\alpha\Rightarrow\beta\quad \Gamma;\Delta\vdash N:\alpha}{\Gamma;\Delta\vdash MN:\beta}\qquad [\mathit{If}]\ \frac{\Gamma;\Delta\vdash M:\mathsf{Bool}\quad \Gamma;\Delta\vdash N_i:\alpha_i\ (i=1,2)}{\Gamma;\Delta\vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2:\alpha}$$

$$[\mathit{Inj}]\ \frac{\Gamma;\Delta\vdash M:\alpha_i}{\Gamma;\Delta\vdash \mathtt{in}_i(M):\alpha_1+\alpha_2}\qquad [\mathit{Case}]\ \frac{\Gamma;\Delta\vdash M:\alpha_1+\alpha_2\quad \Gamma,x_i\!:\!\alpha_i;\Delta\vdash N_i:\beta}{\Gamma;\Delta\vdash \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).N_i\}_{i\in\{1,2\}}:\beta}$$

$$[\mathit{Pair}]\ \frac{\Gamma;\Delta\vdash M_i:\alpha_i\ (i=1,2)}{\Gamma;\Delta\vdash \langle M_1,M_2\rangle:\alpha_1\times\alpha_2}\qquad [\mathit{Proj}]\ \frac{\Gamma;\Delta\vdash M:\alpha_1\times\alpha_2}{\Gamma;\Delta\vdash \pi_i(M):\alpha_i\ (i=1,2)}$$

$$[\mathit{Deref}]\ \frac{\Delta(x)=\mathsf{Ref}(\alpha)}{\Gamma;\Delta\vdash !x:\alpha}\qquad [\mathit{Assign}]\ \frac{\Gamma;\Delta\vdash M:\alpha\quad \Delta(x)=\mathsf{Ref}(\alpha)}{\Gamma;\Delta\vdash x:=M:\mathsf{Unit}}$$

Remark 2.1 In spite of the restriction on types (i.e. reference types are not carried in other types), the language allows arbitrary imperative higher-order procedures to be carried as parameters of procedures and stored in references. Lifting this restriction means references can be used as parameters and return values of procedures, as well as content of other references, leading to a distinct class of behaviours which deserve treatment on their own right: see [10].

The following notion becomes important when we consider the semantics of programs.

Definition 2.2 (closed programs [56]) $\Gamma;\Delta\vdash M:\alpha$ is *closed* when $\mathsf{dom}(\Gamma)=\emptyset$, often written $\Delta\vdash M:\alpha$. The notation $\Delta\vdash\sigma$ is understood similarly.

For brevity, henceforth we work under the following convention.

Convention 2.3

1. *We only consider well-typed programs and configurations. Further we assume configurations only use stores whose values are closed.*
2. *We write $M\overset{def}{=}N$ to indicate M and N are definitionally equal up to the α-equality.*
3. *We write $\lambda().M$ for $\lambda z^{\mathsf{Unit}}.M$ with $z\notin\mathsf{fv}(M)$, $\mathtt{let}\ x=M\ \mathtt{in}\ N$ for $(\lambda x.N)M$, and $M;N$ for $(\lambda().N)M$.*

2.2 Terms and Formulae

Terms and Formulae. The logical language is that of first-order logic with equality [54, § 2.8] together with an assertion for the evaluation of stateful expressions. The grammar of terms and formulae is given below.

$$e \ ::= \ x^{\rho} \mid () \mid \mathsf{c} \mid \mathsf{op}(\vec{e}) \mid \langle e, e' \rangle \mid \pi_i(e) \mid \mathsf{inj}_i^{\alpha+\beta}(e) \mid !e$$

$$C \ ::= \ e = e' \mid \neg C \mid C \wedge C' \mid C \vee C' \mid C \supset C' \mid \forall x^{\alpha}.C \mid \exists x^{\alpha}.C \mid [C]\, e \bullet e' = x\, [C']$$

The first set of expressions (ranged over by $e, e', \ldots$) are *terms* while the second set are *formulae* (ranged over by $A, B, C, C' \ldots$). Terms, which are from the logics for pure functions studied in [34, 39] except for $!x$, include all the constants (ranged over by $\mathsf{c}, \mathsf{c}', \ldots$) including the natural numbers n and the boolean b (either the truth t or false f) and first-order operations (including multiplication) of the target programming language. In the grammar of terms, we have pairing, projection and injection operation. The final term, $!e$, denotes the dereference of e. $\mathsf{fv}(e)$ denotes the free variables occurring in e. Note that expressions in general, and variables in particular, will always denote values in our models, see Section 4.2. That means that non-termination of terms in formulae cannot occur. As in the first-order logic, the denotation of an expression depends only on the denotations of the free variables of expression, and likewise for formulae.

Fig. 2 Typing Rules for Terms and Formulae

$$\frac{(\Gamma \cup \Delta)(x) = \rho}{\Gamma; \Delta \vdash x : \rho} \quad \frac{-}{\Gamma; \Delta \vdash () : \mathsf{Unit}} \quad \frac{-}{\Gamma; \Delta \vdash \mathsf{n} : \mathsf{Nat}} \quad \frac{-}{\Gamma; \Delta \vdash \mathsf{b} : \mathsf{Bool}} \quad \frac{\Gamma; \Delta \vdash e : \mathsf{Bool}}{\Gamma; \Delta \vdash \neg e : \mathsf{Bool}}$$

$$\frac{\Gamma; \Delta \vdash e_i : \alpha_i \ (i = 1,2)}{\Gamma; \Delta \vdash (e_1, e_2) : \alpha_1 \times \alpha_2} \quad \frac{\Gamma; \Delta \vdash e : \alpha_1 \times \alpha_2}{\Gamma; \Delta \vdash \pi_i(e) : \alpha_i} \quad \frac{\Gamma; \Delta \vdash e : \alpha_i \ (i \in \{1,2\})}{\Gamma; \Delta \vdash \mathsf{inj}_i^{\alpha_1+\alpha_2}(e) : \alpha_1 + \alpha_2} \quad \frac{\Gamma; \Delta \vdash e : \mathsf{Ref}(\alpha)}{\Gamma; \Delta \vdash !e : \alpha}$$

$$\frac{\Gamma; \Delta \vdash e_{1,2} : \rho}{\Gamma; \Delta \vdash e_1 = e_2} \quad \frac{\Gamma; \Delta \vdash A_{1,2}}{\Gamma; \Delta \vdash A_1 \star A_2} (\star \in \{\wedge, \vee, \supset\}) \quad \frac{\Gamma, x{:}\alpha; \Delta \vdash A}{\Gamma; \Delta \vdash \forall x^{\alpha}.A} \quad \frac{\Gamma, x{:}\alpha; \Delta \vdash A}{\Gamma; \Delta \vdash \exists x^{\alpha}.A}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \alpha \Rightarrow \beta \quad \Gamma; \Delta \vdash e_2 : \alpha \quad \Gamma; \Delta \vdash C \quad \Gamma, z{:}\beta; \Delta \vdash C'}{\Gamma; \Delta \vdash [C]\, e_1 \bullet e_2 = z\, [C']}$$

The predicate $[C]\, e \bullet e' = x\, [C']$ is called *evaluation formula*, where the name x binds its free occurrences in C'. Intuitively, $[C]\, e \bullet e' = x\, [C']$ asserts that an invocation of e with an argument e' under the initial state C terminates with a final state and a resulting value, named as x, both described by C'. Note that $\bullet$ is non-commutative. Note that we can assert divergence by *negating* evaluation formulae: e.g. $\forall x.\neg[\mathsf{T}]\, f \bullet x = y\, [\mathsf{F}]$ cannot hold of a function that terminates anywhere.

Terms and formulae are typed starting from type-annotated variables. The typing rules are given in Figure 2 (we list only a couple of cases for constants and first-order operators). We write $\Gamma; \Delta \vdash e : \rho$ when e has type ρ under $\Gamma; \Delta$, and $\Gamma; \Delta \vdash C$ when C is well-typed under $\Gamma; \Delta$.

Convention 2.4 (formulae and terms)

1. We often write $\Theta \vdash C$ instead of $\Gamma;\Delta \vdash C$ with $\Theta = \Gamma \cup \Delta$. $\Theta, \Theta', \ldots$ range over finite maps combining two kinds of bases.
2. Logical connectives are used with their standard precedence/association. For example, $\neg A \wedge B \supset \forall x.C \vee D \supset E$ is parsed as $((\neg A) \wedge B) \supset (((\forall x.C) \vee D) \supset E)$.
3. $C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$, the logical equivalence of C_1 and C_2. We use truth T (definable as $1 = 1$) and falsity F (which is $\neg\mathsf{T}$).
4. The standard binding convention is always assumed, and $\mathsf{fv}(C)$ denotes the set of *free variables* in C.
5. $C^{\text{-}\vec{x}}$ is C in which no name from $\vec{x}$ freely occurs.
6. $[C]\ e_1 \bullet e_2 = e'\ [C']$ with e' not a variable, stands for $[C]\ e_1 \bullet e_2 = x\ [x = e' \wedge C']$ with x fresh; and $[C]\ e_1 \bullet e_2\ [C']$ for $[C]\ e_1 \bullet e_2 = ()\ [C']$.
7. Hereafter we only consider well-typed terms and formulae and often omit type annotations. Formulae are often called *assertions*.

Example 2.5 (assertions and their types)

1. Let $Double(u) \stackrel{\text{def}}{=} \forall n^{\mathsf{Nat}}.[\mathsf{T}]\ u \bullet n = 2 \times n\ [\mathsf{T}]$. Then the assertion

$$[Double(!x)]\ u \bullet 3 = 6\ [Double(!x)]$$

says that, if 3 is fed to the function denoted by u with the precondition $Double(!x)$ (i.e. x stores a doubling function), then the result is 6, without changing the content of x. This assertion is typed under $u : \mathsf{Nat} \Rightarrow \mathsf{Nat}\ ;\ x : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat})$ where the type of u indicates not only its argument and target are natural numbers but also an invocation may access x. The assertion is satisfied by, for example, $\lambda y.(!x)y$.
2. The assertion
$$C \stackrel{\text{def}}{=} \forall i,n.\ [!w = n]\ !x \bullet i = 2 \times i\ [!w = n+1]$$
typed under $\emptyset\ ;\ x : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat}), w : \mathsf{Ref}(\mathsf{Nat})$, says that an imperative variable x stores a function (procedure) which, when invoked, would increment w as well as returning the double of the argument. This assertion is satisfied when a procedure $f(w) \stackrel{\text{def}}{=} \lambda z.(w := !w + 1; z \times 2)$ is stored in x.
3. Using C above, let:

$$C' \stackrel{\text{def}}{=} [C \wedge\ !w = 0]\ u \bullet 3 = 6\ [C \wedge\ !w = 1]$$

This predicate says that, if u is invoked with 3 in a state satisfying $!w = 0$ as well as C, then the returned value is 6 and the final state is $!w = 1$. The formula is typed under $u : \mathsf{Nat} \Rightarrow \mathsf{Nat}\ ;\ x : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat}),\ w : \mathsf{Ref}(\mathsf{Nat})$, and is satisfiable by $\lambda y.(!x)y$ named as u, with x storing $f(w)$ above.

We introduce an important subclass of the well-typed formulae, *stateless formulae*. A formula is stateless if its validity does not depend on the current state of the store.

Definition 2.6 (stateless formulae) A formula say C is *stateless* if a name of a reference type in C only occurs inside the pre/post conditions of evaluation formulae. $A, B, \ldots$ range over stateless formulae. We sometimes also call well-typed formulae *stateful formulae* to emphasise that they may make assertions about the content of references.

An evaluation formula is stateless even if it contains dereferences in its pre/post conditions because they describe hypothetical state, *not* the current state. For example, if f denotes a procedure which always increments the content of a reference x, whether or not this fact holds does not depend on the current content of x, or on any other state. Thus $x = 3$ and $[Even(!y)]\, x \bullet ()\, [Odd(!y)]$ are both stateless, but neither $[Even(!y)]\, x \bullet ()\, [Odd(!y)] \wedge\, !y = 3$ nor $!y = 3$ are stateless.

2.3 Name Capture Avoiding Substitutions

In logics with equality and/or quantifications, capture-avoiding syntactic substitutions play a key role in deduction [54, §2]. Due to the existence of evaluation formulae, there is a subtlety in the interplay between substitution and evaluation formulae. Consider:

$$C \quad \stackrel{\text{def}}{=} \quad m = 0 \ \wedge\ \forall i.[m = 0 \wedge\, !y = i]\, f \bullet ()\, [!y = i + 1] \tag{2.9}$$

C says that the value of m is 0 and that f would, when invoked, increment the content of y at the time of invocation, whatever y might store. Note the first $m = 0$ is a stateless assertion, hence it should continue to be true in the precondition in the second conjunct. Hence this is equivalent to, under any reasonable interpretation,

$$C \quad \equiv \quad m = 0 \wedge \forall i.[!y = i]\, f \bullet ()\, [!y = i + 1] \tag{2.10}$$

Now suppose we wish to substitute $!y$ for m. Since (2.9) and (2.10) are logically equivalent, it should be the case that the results of applying the same substitution to both should again be logically equivalent. However if we substitute m for y in (2.9) we get:

$$C[!y/m] \quad \stackrel{\text{def}}{=} \quad !y = 0 \ \wedge\ \forall i.[!y = 0 \wedge\, !y = i]\, f \bullet ()\, [!y = i + 1]$$

which says the value *currently* stored in y is 0, and that f would, when invoked, increment the content of y at the time of invocation, *if y stores* 0. But if we apply the same substitution to (2.10), the result is a quite different:

$$!y = 0 \wedge \forall i.[!y = i]\, f \bullet ()\, [!y = i + 1]$$

which omits the condition y should store 0 in the precondition for f to increment the content of y. This is because assertions in pre/post conditions in evaluation formulae describe *hypothetical state of stores*, necessary to describing behaviour of λ-abstractions. We extend the standard notion "e is free for x in C" [54, §2.1], see [41] for details.

In the proof rule for assignment given later, we also use a substitution of the form $C[e/!x]$, in which e is substituted for each "free" dereference $!x$ occurring in C. This substitution should not affect the occurrences of $!x$ in pre/post conditions of evaluation formulae in C since they are about hypothetical states. For example, let C be $!x = 3 \ \wedge$ $\forall i.[!x = i]\, f \bullet ()\, [!x = i + 1]$. Then the second conjunct of C is stateless (its validity does not depend on the current state of the store) hence the substitution say $[3/!x]$ should only change the first conjunct, not the second. Hence the substitution rule for this substitution is:

$$([C]\, e_1 \bullet e_2 = z\, [C'])[e/!x] \quad \stackrel{\text{def}}{=} \quad [C]\, (e_1[e/!x]) \bullet (e_2[e/!x]) = z\, [C']$$

and homomorphically for other constructs (note that, by our grammar, quantifiers do not bind reference variables). Under this substitution, the notion that a term e^{α} is *free for* $(!x)^{\alpha}$ *in* C is defined exactly as in the standard notion [54, §2.1], avoiding the capture under quantification, e.g. in $(\exists y.C)[e/!x]$, if y occurs in e, then we first alpha-convert y into a fresh variable.

Convention 2.7 *Whenever we write $C[e/!x]$, we assume e is free for $!x$ in C. Likewise, whenever we write $C[e/x]$ we assume e is free for x in C.*

2.4 Judgement

Following Hoare [31], a judgement for total correctness in the present program logic consists of a program sandwiched by a pair of formulae, augmented with a fresh name called *anchor*, written as follows.

$$[C]\ M^{\Gamma;\Delta;\alpha} :_u\ [C']$$

This sequent is used for both validity and provability. If we wish to be specific, we prefix it with either $\vdash$ (for provability) or $\models$ (for validity). In the judgement above, M is the *subject* of the judgement, u its *anchor*, C its *pre-condition*, and C' its *post-condition*. Intuitively, the judgement says:

> *if the free non-reference variables of M are instantiated into values satisfying C and gets evaluated starting from a store satisfying C, then it terminates with the final state and the resulting value, named u, together satisfying C'.*

Definition 2.8 (primary/auxiliary names in a judgement) Let $[C]\,M^{\Gamma;\Delta;\alpha} :_u [C']$ be well-typed. Then the *primary names* in this judgement are the members of $\mathsf{dom}(\Gamma,\Delta) \cup \{u\}$. The *auxiliary names* in the judgement are those free names in C and C' that are not primary. *Henceforth we assume auxiliary names do not include reference names.*

Judgements are typed as expected:

Definition 2.9 We say $[C]\,M^{\Gamma;\Delta;\alpha} :_u [C']$ is *well-typed* iff (1) $\Gamma;\Delta \vdash M : \alpha$ and (2) $\Gamma,\Delta,\Theta \vdash C$ and $u\!:\!\alpha,\Gamma,\Delta,\Theta \vdash C'$ for some Θ such that $\mathsf{dom}(\Theta) \cap (\mathsf{dom}(\Gamma,\Delta) \cup \{u\}) = \emptyset$.

Convention 2.10 *Henceforth we assume a given judgement is well-typed. For brevity, we often omit the typing from a judgement when it is understood from the context, writing $[C]\,M :_u [C']$.*

3 Proof Rules

The proof rules for the present logic are divided into those which precisely follow the structure of programs (compositional rules) and those which do not (structural rules). We first list the former in Figure 3. Two key structural rules are given in Figure 4. The remaining structural rules as well as the axioms are standard [10, 79], and can be found in [41]. We use the following conventions.

Convention 3.1 (convention on the use of names)

Fig. 3 Compostional Proof Rules

$$[\mathit{Var}]\ \frac{-}{[C[x/u]]\ x :_u [C]} \qquad [\mathit{Const}]\ \frac{-}{[C[\mathsf{c}/u]]\ \mathsf{c} :_u [C]}$$

$$[\mathit{Op}]\ \frac{C_0 \overset{\text{def}}{=} C \quad [C_i]\, M_i :_{m_i} [C_{i+1}]\ (0 \le i \le n-1) \quad C_n \overset{\text{def}}{=} C'[\mathsf{op}(m_0..m_{n-1})/u]}{[C]\, \mathsf{op}(M_0..M_{n-1})\ :_u\ [C']}$$

$$[\mathit{Abs}]\ \frac{[C \wedge A^{\text{-}x}]\ M :_m [C']}{[A]\ \lambda x.M :_u [\forall x.[C]\, u \bullet x = m\, [C']]} \qquad [\mathit{App}]\ \frac{\begin{array}{c}[C]\ M :_m [C_0] \\ [C_0]\ N :_n [\ C_1 \wedge\ [C_1]\ m \bullet n = u\ [C']\,]\end{array}}{[C]\ MN :_u [C']}$$

$$[\mathit{If}]\ \frac{[C]\ M :_b [C_0] \quad [C_0[\mathtt{t}/b]]\ M_1 :_u [C'] \quad [C_0[\mathtt{f}/b]]\ M_2 :_u [C']}{[C]\ \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u [C']}$$

$$[\mathit{In}_1]\ \frac{[C]\ M :_v [C'[\mathtt{in}_1(v)/u]]}{[C]\ \mathtt{in}_1(M) :_u [C']} \qquad [\mathit{Case}]\ \frac{[C^{\text{-}\vec{x}}]\ M :_m [C_0^{\text{-}\vec{x}}] \quad [C_0[\mathtt{in}_i(x_i)/m]]\ M_i :_u [C'^{\,\text{-}\vec{x}}]}{[C]\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i \in \{1,2\}} :_u [C']}$$

$$[\mathit{Pair}]\ \frac{[C]\ M_1 :_{m_1} [C_0] \quad [C_0]\ M_2 :_{m_2} [C'[\langle m_1, m_2 \rangle/u]]}{[C]\ \langle M_1, M_2 \rangle :_u [C']} \qquad [\mathit{Proj}_1]\ \frac{[C]\ M :_m [C'[\pi_1(m)/u]]}{[C]\ \pi_1(M) :_u [C']}$$

$$[\mathit{Deref}]\ \frac{-}{[C[!x/u]]\ !x :_u [C]} \qquad [\mathit{Assign}]\ \frac{[C]\ M :_m [C'[m/\,!x][()/u]]}{[C]\ x := M :_u [C']}$$

$$[\mathit{Rec}]\ \frac{[A^{\text{-}xi} \wedge \forall j \lneq i.B(j)[x/u]]\ \lambda y.M :_u [B(i)^{\text{-}x}]}{[A]\ \mu x.\lambda y.M :_u [\forall i.B(i)]}$$

- Free $i, j, \ldots$ exclusively range over auxiliary names.
- In each proof rule, we assume all occurring judgements are well-typed, and no primary names in the premise(s) occur as auxiliary names in the conclusion (this may be considered as a variant of the standard bound name convention).

We now explain some key rules. For a more detailed explanation, see [41].

[*Deref*] is similar to **[*Var, Const*]**, using substitution. We assume $!x$ is free for m in C. The rule says that, if we wish for C to hold for the dereference of x named u, then we should assume the same for its content.

[*Assign*] uses two substitutions discussed, $[m/!x]$ and $[()/u]$ The first substitution $C'[m/!x]$ says the result of the assignment $x := M$ is turning what is stated about m in $C'[m/!x]$ into the property of $!x$. The second one $[()/u]$ says, in effect, the assignment command terminates (note $()$ is the unique value of type Unit).

[*Consequence-Aux*] is the rule which was originally introduced for the standard Hoare Logic by Kleymann [46] (which captures the semantics of auxiliary names).

4 Soundness and Observational Completeness

We begin this section with a quick summary of our notion of model for assertions and judgements based on the usual observational congruence for call-by-value PCF. We

Fig. 4 Selected Structural Rules

$$[\mathit{Invariance}]\ \frac{[C]\, M^{\Gamma;\Delta;\alpha} :_m [C'] \quad \Gamma;\Delta_0 \vdash C_0 \quad (\Delta_0 \text{ disjoint from } \Delta)}{[C \wedge C_0]\, M^{\Gamma;\Delta,\Delta_0;\alpha} :_m [C' \wedge C_0]}$$

$$[\mathit{Consequence\text{-}Aux}]\ \frac{[C_0]\, M^{\Gamma;\Delta;\alpha} :_u [C_0'] \quad C \supset \exists \vec{j}.(\, C_0[\vec{j}/\vec{i}] \wedge (C_0'[\vec{y}/\vec{x}][\vec{j}/\vec{i}] \supset C'[\vec{y}/\vec{x}])\,)}{[C]\, M :_u [C']}$$

In $[\mathit{Consequence\text{-}Aux}]$, we set $\{\vec{x}\} = \mathsf{dom}(\Gamma,\Delta) \cup \{u\}$, $\{\vec{i}\} = \mathsf{fv}(C, C', C_0, C_0') \backslash \{\vec{x}\}$, and $\vec{j}$ (resp. $\vec{y}$) are fresh. We assume no auxiliary reference names occur.

also establish soundness of axioms and proof rules and establish observational completeness. Proofs except the observational completeness are can be found in [41].

4.1 Observational Congruence

A *typed congruence* is an equivalence on typed terms with identical bases and types, closed under the compatibility rules corresponding to the typing rules. We write $\Gamma;\Delta \vdash M_1 \ \mathcal{R}\ M_2 : \alpha$ when $\Gamma;\Delta \vdash M_1 : \alpha$ and $\Gamma;\Delta \vdash M_2 : \alpha$ are related by a typed relation $\mathcal{R}$.

Definition 4.1 (observational congruence) Let $\Gamma;\Delta \vdash M_{1,2} : \alpha$. Then $\Gamma;\Delta \vdash M_1 \cong M_2 : \alpha$ is the maximum typed congruence such that for each semi-closed $\Delta \vdash M_{1,2} : \mathsf{Unit}$ and for each σ such that $\Delta \vdash \sigma$, we have $(M_1, \sigma) \Downarrow$ iff $(M_2, \sigma) \Downarrow$.

Convention 4.2 *Below and henceforth we let ω^α stand for a(ny) diverging closed term of type α: e.g. we can take $\omega^\alpha \stackrel{def}{=} (\mu x^{\alpha \Rightarrow \alpha}.\lambda y.xy)V$ with V any closed value typed α. Further we write, after fixing ω^α for each α, $\Omega^{\alpha \Rightarrow \beta}$ for $\lambda x^\alpha.\omega^\beta$, which is the least value at each arrow type (note it immediately diverges after invocation).*

Example 4.3 Note that the choice of basis affects the contextual congruence. For example with

$$M \stackrel{\text{def}}{=} \lambda y^\alpha.\mathtt{let}\ z = y()\ \mathtt{in}\ 3 \qquad N \stackrel{\text{def}}{=} \lambda y^\alpha.\mathtt{let}\ z = y()\ \mathtt{in}\ \mathtt{let}\ z' = y()\ \mathtt{in}\ 3$$

with $\alpha = \mathsf{Unit} \Rightarrow \mathsf{Unit}$. Then we have $\vdash M \cong N : \alpha \Rightarrow \mathsf{Nat}$, but $x : \mathsf{Ref}(\mathsf{Nat}) \vdash M \ncong N : \alpha \Rightarrow \mathsf{Nat}$. To check the latter, take $C[\,\cdot\,] \stackrel{\text{def}}{=} ([\,\cdot\,]L); \mathtt{if}\ !x = 1\ \mathtt{then}\ ()\ \mathtt{else}\ \omega$ with $L \stackrel{\text{def}}{=} \lambda().x := x+1$. Then $(C[M], x \mapsto 0)$ converges and $(C[N], x \mapsto 0)$ diverges.

Later we shall use the following ordering corresponding to $\cong$. Below a *typed precongruence* is a typed preorder closed under the compatibility rules.

Definition 4.4 (contextual ordering) Let $\Gamma;\Delta \vdash M_{1,2} : \alpha$. Then $\Gamma;\Delta \vdash M_1 \sqsubseteq M_2 : \alpha$ is the maximum typed precongruence satisfying, for each $\Delta \vdash M_{1,2} : \mathsf{Unit}$ and for each σ such that $\Delta \vdash \sigma$, $(M_1, \sigma) \Downarrow$ implies $(M_2, \sigma) \Downarrow$. We write $\sqsupseteq$ for the inverse of $\sqsubseteq$.

$\sqsubseteq$ is the preorder corresponding to $\cong$, i.e. $\sqsubseteq \cap \sqsupseteq = \cong$, and induces a partial order on the congruence classes of $\cong$.

4.2 Models and Soundness

Definition 4.5 (models) A *model of type* $\Gamma;\Delta$ is a pair (ξ,σ) such that ξ is a finite map from $\mathsf{dom}(\Gamma)$ to semi-closed values such that each $x \in \mathsf{dom}(\Gamma)$ is mapped to a semi-closed value such that $\Delta \vdash x : \Gamma(x)$; and σ is a finite map from $\mathsf{dom}(\Delta)$ to closed values such that each $x \in \mathsf{dom}(\Delta)$ is mapped to a semi-closed value $\Delta \vdash V : \alpha$ with $\Delta(x) = \mathsf{Ref}(\alpha)$. We let $\mathcal{M},\ldots$ range over models.

We write $\Gamma;\Delta \vdash \mathcal{M}$ or $\mathcal{M}^{\Gamma;\Delta}$ when $\mathcal{M}$ is a model of type $\Gamma;\Delta$. Intuitively, ξ and σ in (ξ,σ) respectively denote a standard functional environment and a store.

We now formalise the semantics of assertions. First we interpret terms under a model. We use the following notations.

Notation 4.6 1. Given $\Gamma;\Delta \vdash \mathcal{M}$ such that $\mathcal{M} = (\xi,\sigma)$, we write $x : V \in \mathcal{M}$ when $\xi(x) = V$; and $x \mapsto V \in \mathcal{M}$ when $\Delta(x) = \mathsf{Ref}(\alpha)$ and $\sigma(x) = V$ with $\Delta \vdash V : \alpha$.

2. Let $\Gamma;\Delta \vdash \mathcal{M}$ such that $\mathcal{M} = (\xi,\sigma)$ below.
   (a) We write $\mathcal{M}[x : V]$ for the result of replacing the target of x in ξ with V, assuming $x \in \mathsf{dom}(\xi)$. Similarly we define $\mathcal{M}[x \mapsto V]$, $\xi[x : V]$ and $\sigma[x \mapsto V]$.
   (b) We write $\mathcal{M}\cdot x : V$ for $(\xi \cup \{x : V\},\sigma)$, assuming simultaneously $x \notin \mathsf{dom}(\xi \cup \sigma)$. Similarly we define $\mathcal{M}\cdot[x \mapsto V]$, $\xi \cdot x : V$ and $\sigma \cdot [x \mapsto V]$.
3. $\xi\backslash\vec{x}$ removes from ξ all entries mapping elements of $\vec{x}$. Similarly we write $\mathcal{M}\backslash\vec{x}$ for the result of taking off $\vec{x}$-elements from the components of $\mathcal{M}$.

Definition 4.7 (interpretation of terms) Let $\Gamma;\Delta \vdash e : \rho$ for some ρ and $\Gamma;\Delta \vdash \mathcal{M}$. Then the *interpretation of e under* $\mathcal{M}$, denoted $\mathcal{M}[\![e]\!]$, is given by the following clauses.

$$\begin{array}{rll@{\qquad}rl}
\mathcal{M}[\![x^{\alpha}]\!] & \stackrel{\text{def}}{=} V & (x^{\alpha} : V \in \mathcal{M}) & \mathcal{M}[\![\mathtt{op}(\vec{e})]\!] & \stackrel{\text{def}}{=} \mathtt{op}(\mathcal{M}[\![\vec{e}]\!]) \\
\mathcal{M}[\![!x^{\mathsf{Ref}(\alpha)}]\!] & \stackrel{\text{def}}{=} V & (x^{\mathsf{Ref}(\alpha)} \mapsto V \in \mathcal{M}) & \mathcal{M}[\![\langle e,e'\rangle]\!] & \stackrel{\text{def}}{=} \langle \mathcal{M}[\![e]\!],\mathcal{M}[\![e']\!]\rangle \\
\mathcal{M}[\![x^{\mathsf{Ref}(\alpha)}]\!] & \stackrel{\text{def}}{=} x & & \mathcal{M}[\![\pi_i(e)]\!] & \stackrel{\text{def}}{=} \pi_i(\mathcal{M}[\![e]\!]) \\
\mathcal{M}[\![\mathtt{c}]\!] & \stackrel{\text{def}}{=} \mathtt{c} & & \mathcal{M}[\![\mathtt{in}_i(e)]\!] & \stackrel{\text{def}}{=} \mathtt{in}_i(\mathcal{M}[\![e]\!])
\end{array}$$

By construction of models, all terms are interpreted as semi-closed values *except* for reference names. A reference name is interpreted as itself, which indicates a reference name is in fact treated as a constant (one may observe that a value does mention reference names it may access, which indicate they are treated as formal part of the universe of behaviours, unlike function variables). All distinct reference names are considered to be distinct constants. This treatment is also reflected in the lack of quantifiers for reference names in the present logic.

Definition 4.8 (satisfaction) Given $\Gamma;\Delta \vdash \mathcal{M}$ and $\Gamma;\Delta \vdash C$, the relation $\mathcal{M} \models C$ (read: *$\mathcal{M}$ satisfies C*) is generated from the following clauses.

$$
\begin{array}{rll}
\mathcal{M} \models e_1 = e_2 & \text{if} & \mathcal{M}[\![e_1]\!] \;\cong\; \mathcal{M}[\![e_2]\!] \\
\mathcal{M} \models C_1 \wedge C_2 & \text{if} & (\mathcal{M} \models C_1) \;\wedge\; (\mathcal{M} \models C_2) \\
\mathcal{M} \models C_1 \vee C_2 & \text{if} & (\mathcal{M} \models C_1) \;\vee\; (\mathcal{M} \models C_2) \\
\mathcal{M} \models C_1 \supset C_2 & \text{if} & (\mathcal{M} \models C_1) \;\supset\; (\mathcal{M} \models C_2) \\
\mathcal{M} \models \neg C & \text{if} & \neg\, (\mathcal{M} \models C) \\
\mathcal{M} \models \forall x^{\alpha}.C & \text{if} & \forall \Delta \vdash V : \alpha.\ \mathcal{M} \cdot x : V \models C \\
\mathcal{M} \models \exists x^{\alpha}.C & \text{if} & \exists \Delta \vdash V : \alpha.\ \mathcal{M} \cdot x : V \models C \\
\mathcal{M} \models [C] e_1 \bullet e_2 = x[C'] & \text{if} & \forall \sigma.\ (\ \Delta \vdash \sigma \;\wedge\; (\xi,\sigma) \models C \quad \supset \\
 & & \quad \exists V, \sigma'.\ ((\mathcal{M}[\![e_1]\!])(\mathcal{M}[\![e_2]\!]),\ \sigma) \;\Downarrow\; (V, \sigma') \\
 & & \quad \text{such that } (\xi \cup x : V, \sigma') \models C'\ )
\end{array}
$$

These definitions are similar to [10, 79] where they are discussed in more detail. We are now ready to formalise the semantics of judgements. Below $M\xi$ denotes the substitution of values following ξ, e.g. $(x+y)\xi = 2+3$, provided $\xi(x) = 2, \xi(y) = 3$.

Definition 4.9 (semantics of judgement) $\models [C]\, M :_u [C']$ iff for each well-typed model (ξ,σ): $(\xi,\sigma) \models C$ implies both, $(M\xi,\sigma) \Downarrow (V,\sigma')$ and $(\xi \cdot u : V, \sigma') \models C'$

Proposition 4.10 (soundness of axioms) *All axioms of our logic are true under arbitrary (well-typed) models.*

Theorem 4.11 (soundness of proof rules) *If* $\vdash [C]\, M :_u [C']$ *by the proof rules in Figures 3 and 4, then* $\models [C]\, M :_u [C']$.

Proofs of soundness can be found in [41].

4.3 Observability and Program Logics

In languages with compositional semantics, program components with the same contextual behaviour are interchangeable without affecting the observable behaviour of the programs they are part of, thus offering foundations for modular software engineering. Compositional program logics extend this idea by further allowing programs with the same specifications to be interchangeable in a larger program, without affecting the observable behaviour of the whole, up to the latter's specification. Thus, ideally, valid assertions for programs should capture exactly the observable behaviour of programs [29, 55, 56]. Formally, we may ask: are two programs contextually equivalent if and only if they satisfy the same set of assertions? An affirmative answer reassures us that the logic enables us to reason about all observational properties, but no more. We call logics with this property *observationally complete*.

In the following we show that our logic is indeed observational complete, using the following steps:

1. We introduce a variant of *finite canonical forms* (FCFs) [6, 38, 42] which represent a limited class of behaviours.

2. We show that for each FCF characteristic formulae (w.r.t. total correctness) can be derived. They are formulae that capture the whole behaviour of the FCF.
3. By reducing differentiating contexts of two observationally distinct programs to FCFs, and further to their characteristic formulae, we show any semantically distinct programs can be differentiated by an assertion, leading to the characterisation of $\cong$ by logical validity.

The next two subsections introduce characteristic formulae and FCFs. For simplicity, and without loss of generality, we work under the following convention.

Convention 4.12 *Throughout the present section we only consider* Nat*, arrow types and induced reference types. Accordingly the conditional (*$\texttt{if } M \texttt{ then } N \texttt{ else } N'$*) branches on zero or non-zero, and each assignment has the shape* $(x := M); N$*.*

Definition 4.13 (TCAs) An assertion C is a *total correctness assertion (TCA) at u* if whenever $(\xi \cdot u : V, \sigma) \models C$ and $V \sqsubseteq V'$, we have $(\xi \cdot u : V', \sigma) \models C$.

Logics for total correctness properties are about upwards-closed properties. That means that if $\models [C]\ M :_m [C']$ and the program M is less defined than the program N, then also $\models [C]\ N :_m [C']$. For example, with Ω being a non-terminating program of integer type, $\lambda x.\Omega$ is less defined than $\lambda x.17$, and $\models [\mathsf{T}]\ \lambda x.\Omega :_m [\mathsf{T}]$ as well as $\models [\mathsf{T}]\ \lambda x.17 :_m [\mathsf{T}]$. This is because logics of total correctness cannot talk about non-termination. If we see program properties as given by a pair of pre-condition and post-condition, they must be upwards closed. See [37] for more details.

Characteristic formulae in the present logic are defined as:

Definition 4.14 (characteristic formulae) Given $\Delta \vdash V : \alpha$, a TCA C at u *characterises* V iff: (1) $\models [\mathsf{T}]\ V^{\Delta;\alpha} :_u [C]$ and (2) $\models [\mathsf{T}]\ W^{\Delta;\alpha} :_u [C]$ implies $\Delta \vdash V \sqsubseteq W : \alpha$.

In the technical development later, we need to consider characteristic formulae of open programs, extending Definition 4.14.

Definition 4.15 (characteristic assertion pair) We say a pair (C, C') is *a characteristic assertion pair (CAP) for* $\Gamma;\Delta \vdash M : \alpha$ *at u* iff we have: (0) C' is a TCA at u; (1) $\models [C]\ M^{\Gamma;\Delta;\alpha} :_u [C']$ and (2) $\models [C]\ N^{\Gamma;\Delta;\alpha} :_u [C']$ implies $\Gamma;\Delta \vdash M \sqsubseteq N : \alpha$. We also say (C, C') *characterise* $\Gamma;\Delta \vdash M : \alpha$ *at u* when (C, C') is a CAP for $\Gamma;\Delta \vdash M : \alpha$ at u.

4.4 Finite Canonical Forms

If $(C[M_1], \sigma)$ converges and $(C[M_2], \sigma)$ diverges, the convergent program can only explore a finite part of $C[\ \cdot\ ]$'s and σ's behaviour because the number of reduction steps to reach a value is finite. We can thus always make $C[\ \cdot\ ]$ and σ as little defined as possible, up to the point they have barely necessary constructs for convergence. Since the resulting minimal context and store are less defined than the original ones, it still lets M_2 diverge. In the functional sublanguage, finiteness can be easily captured as *finite canonical forms* [38] (cf. [6, 42]). Below we extend the construction in [38] to the present language.

Finite canonical forms (FCFs), ranged over by $F, F', \ldots$, are a subset of typable terms given by the following grammar (which are read as programs in imperative PCFv in the obvious way). $U, U', \ldots$ range over FCFs which are values.

$$\begin{array}{lll} F & ::= & \mathtt{n} \mid \omega^{\alpha} \mid \lambda x.F \mid \mathtt{let}\ x = yU\ \mathtt{in}\ F \mid \mathtt{case}\ x\ \mathtt{of}\ \langle \mathtt{n_i} : F_i \rangle_{i \in X} \\ & \mid & \mathtt{let}\ x = !y\ \mathtt{in}\ F \mid x := U; F \end{array}$$

Note that $x := U; F$ should be read as $(x := U); F$. Recall from Convention 4.2 that ω is a diverging and closed term. In the `case`-statement, the index set X is a finite, non-empty subset of natural numbers. The `case` construct diverges for values not in X. In $\mathtt{let}\ x = yU\ \mathtt{in}\ F$ (resp. $\mathtt{case}\ x\ \mathtt{of}\ \langle \mathtt{n_i} : F_i \rangle_i$), x should not be free in U (resp. F_i).

Clearly, FCFs can easily and naturally be translated into our imperative PCFv variant, and is typed following this translation, cf. [38].

The rest of our development relies on this straightforward fact about FCFs. A proof can be found in [41].

Lemma 4.16 *Let $M_{1,2}$ be values and $\Delta \vdash M_1 \not\cong M_2 : \alpha$. Then there exist semi-closed FCF F and $\vec{U}$, which are also values, such that, with $(i, j) = (1, 2)$ or $(i, j) = (2, 1)$:*

$$(FM_i, \vec{r} \mapsto \vec{U}) \Downarrow \qquad \textit{and} \qquad (FM_j, \vec{r} \mapsto \vec{U}) \Uparrow$$

with $\{\vec{r}\} \supset \mathsf{dom}(\Delta)$.

4.5 Characteristic Formulae for FCFs

We move to the derivation of CAPs for imperative FCFs. No change in the rules is necessary except for the let-application which now needs to mention state. In addition, we introduce one rule for each of dereference and assignment. We also need *weakening rule for values* which fills pre/post conditions with assertions on the invariance of states for values (which allows us to have clean derivations for values). Figure 5 presents the derivation rules, using stateful formulae. We observe:

- A CAP of n at u is $(\mathsf{T}, u = \mathsf{n})$, saying: whenever a program, say M, satisfies $[\mathsf{T}]M :_u [u = \mathsf{n}]$, M is contextually equal to n. E.g. under $x : \mathsf{Nat}$, $\mathtt{if}\ x\ \mathtt{then}\ \mathsf{n}\ \mathtt{else}\ \mathsf{n}$ has this property.
- For the case construct, given a CAP (A_i, B_i) at u of each F_i, we make the weakest precondition for the resulting term to converge, which is the i-indexed disjunction of $x = \mathsf{n}_i$ and A_i. For each i-th case, it can guarantee what F_i guarantees.
- A CAP for ω at u is (F, F): since this FCF never terminates, we can do nothing but assume absurdity. Compared with any program, ω is the least, so it is indeed a CAP of this program.
- For a CAP of the let-application, first, each value always has the precondition T, so there is no loss of generality in assuming (T, A) is a CAP for U. We further assume (C, C') is a CAP of F. The termination guarantee for F is obtained by extracting the "current state" by $!\vec{r} = \vec{j}$ (this equality does not violate TCA since $\vec{j}$ are quantified). The precondition can equivalently be written as $\exists \vec{j}.\,(\,!\vec{r} = \vec{j} \wedge \forall z.[A \wedge !\vec{r} = \vec{j}]\, f \bullet z = x\, [C]\,)$.

Fig. 5 Derivation Rules for Characteristic Assertions of FCFs

$$\frac{-}{[\mathsf{T}]\ \mathtt{n} :_u [u = \mathsf{n}]} \qquad \frac{[C_i]\ F_i :_u [C_i']}{[\vee_i(x = \mathsf{n}_i \wedge C_i)]\ \mathtt{case}\ x\ \mathtt{of}\ \langle \mathtt{n_i} : F_i \rangle_i :_u [\vee_i(x = \mathsf{n}_i \wedge C_i')]}$$

$$\frac{-}{[\mathsf{F}]\ \omega :_u [\mathsf{F}]} \qquad \frac{[C]\ F^{\Gamma, x:\alpha;\Delta;\beta} :_m [C']}{[\mathsf{T}]\ \lambda x.F :_u [\forall x.[C]u \bullet x = m[C']]} \qquad \frac{[C]\ F :_u [C']}{[C[!x/y]]\ \mathtt{let}\ y = !x\ \mathtt{in}\ F :_u [C']}$$

$$\frac{[\mathsf{T}]\ U :_z [A] \quad [C]F :_u [C'] \qquad j \text{ fresh}}{[\forall \vec{j}.\ (\,!\vec{r} = \vec{j}\ \supset\ \forall z.[A \wedge !\vec{r} = \vec{j}]f \bullet z = x[C \wedge x = j]\,)]\ \mathtt{let}\ x = yU\ \mathtt{in}\ F :_u [C'[j/x]]}$$

$$\frac{[\mathsf{T}]\ U^{\Delta;\alpha} :_z [A] \quad [C]\ F :_u [C'] \quad \mathsf{fv}(A) \subset [z] \cup \mathsf{dom}(\Delta)}{[\forall z.(A \supset C[z/!x])]\ x := U\, ;\, F :_u [C']} \qquad \frac{[\mathsf{T}]\ U^{\Gamma;\Delta;\alpha} :_u [A] \quad \mathsf{dom}(\Delta) = \vec{r}}{[!\vec{r} = \vec{i}]\ U^{\Gamma;\Delta;\alpha} :_u [A \wedge !\vec{r} = \vec{i}]}$$

- The last rule, the weakening rule for values, fills the pre/post-conditions with the same assertion on state, indicating the stateless nature of values: this is needed to precisely capture their behaviour in the stateful contexts. We assume:
  - If $[\mathsf{T}]\ U :_z [B]$ etc. is in the premise, we assume the judgement is directly obtained from the rules for values (numerals and abstraction).
  - If $[C]\ F :_u [C']$ etc. is in the premise and F is a value, that judgement should come immediately after this filling rule (preceding by the rules for values).
- The rule for dereference starts from a CAP (C, C') for F, and adjoins an additional constraint on $!x$ from that of y by syntactic substitution, to obtain $(C[!x/y], C')$ as a new CAP (this additional constraint is propagated to C' via auxiliary variables).
- In the rule for assignment, we derive a CAP of a program which writes U to x then behaves as F. The judgement assumes, by the third premise, that A has no free auxiliary names. This does not loose generality by universal closure. The rule may look simple, but its precondition in the conclusion deserves some inspection.
  - The assertion $\forall z.(A \supset C[z/!x])$ may be most easily understood from the viewpoint of an MTC (minimal terminating condition) for the resulting program, $x := U; F$. For this program to converge, the assertion $A[!x/z]$, which will hold after $x := U$, should be stronger than C at x, since if not F would diverge — given that C is an MTC for F. For example, C may demand the content of x increments 1, 2 and 3, while A may only guarantee that z (i.e. U) increments 1 but not others, giving only a weaker condition than C: or C may demand x stores 1, while A may say z is 2, guaranteeing a condition contradictory to C. To avoid such situations, we require A to be stronger than $C[z/!x]$ in a given initial state.
  - A further understanding of the precondition may be obtained by realising that, while not explicitly present, a consequence of $\models [\mathsf{T}]\ U :_z [A]$ (from the premise) is that the assertion $\exists z.A$ comes free (it is a tautology in the sense that it holds in any model). Hence, combined with the explicitly given precondition, we have $\exists z.(A \wedge C[z/!x])$. Note that its second conjunct stipulates the original precondi-

tion for F except at x for which we stipulate none (there is no point in stipulating anything about the content of a variable that is going to be overwritten). The conjunction also indicates that the information on U which A describes is communicated to C as the state at x *after* the initial assignment, albeit under the name z. This is then propagated to the postcondition C' through the closure property of the strong CAP (C, C') for F.

We write $\vdash_{\mathsf{char}} [C]\ F :_u [C']$ when $[C]\ F :_u [C']$ is derivable from the rules in Figure 5 except, when F is a value, we take the result of applying the weakening rule. We now observe (with $\sigma' \sqsubseteq \sigma'_0$ denoting the point-wise extension of $\sqsubseteq$):

Proposition 4.17 *If* $\vdash_{\mathsf{char}} [C]\ F :_u [C']$*, then* (C, C') *satisfies the following conditions.*

1. *(soundness)* $\models [C]\, F :_u [C']$ *with B being a TCA at u.*
2. *(MTC, minimal terminating condition)* $(F\xi, \sigma) \Downarrow$ *if and only if* $(\xi, \sigma) \models C$.
3. *(closure) If* $\models [C_0]\, M :_u [C']$ *such that* $C_0 \supset C$*, then if* $(M\xi, \sigma) \Downarrow (V_0, \sigma'_0)$ *and* $(F\xi, \sigma) \Downarrow (V, \sigma')$*, we have* $V \sqsubseteq V_0$ *and* $\sigma' \sqsubseteq \sigma'_0$*, for each* $(\xi, \sigma) \models C_0$.

Proof See Appendix A.2. □

4.6 Observational Completeness

We conclude this section by establishing observational completeness. We first define the standard logical equivalence, cf. [29].

Definition 4.18 (logical equivalence) Write $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$ when $\models [C]\ M_1^{\Gamma;\Delta\alpha} :_u [C']$ iff $\models [C]\ M_2^{\Gamma;\Delta;\alpha} :_u [C']$.

Note that the definition of $\cong_{\mathcal{L}}$ does not restrict the class of formulae to TCAs. The main result of this section follows.

Theorem 4.19 *Let* $\Gamma; \Delta \vdash M_{1,2} : \alpha$*. Then* $\Gamma; \Delta \vdash M_1 \cong M_2 : \alpha$ *iff* $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$.

Proof The "only if" direction is direct from the definition of the model. For the "if" direction, we prove the contrapositive. Suppose $M_1 \cong_{\mathcal{L}} M_2$ but $M_1 \not\cong M_2$. By abstraction, we can safely assume $M_{1,2}$ are semi-closed values. By Lemma 4.16, there exist semi-closed FCF values F and $\vec{U}$ such that, say,

$$(FM_1, \vec{r} \mapsto \vec{U}) \Downarrow \qquad \text{and} \qquad (FM_2, \vec{r} \mapsto \vec{U}) \Uparrow . \tag{4.1}$$

By Proposition 4.17, there are assertions which characterise F and $\vec{U}$ (in the sense of Definition 4.14). Let the characteristic formula for F at f be written $[\![F]\!](f)$. We now reason:

$$\begin{array}{ll} (FM_1,\ \vec{r} \mapsto \vec{U}) \Downarrow & \\ \qquad \Rightarrow & f : [F] \cdot m : [M_1]\ \models\ [\wedge_i [\![U_i]\!](!r_i)]\ f \bullet m = z\ [\mathsf{T}] \\ \qquad \Rightarrow & \forall V.\ (f : V \models [\![F]\!]_f \text{ implies } f : V \cdot m : [M_1]\ \models\ [\wedge_i [\![U_i]\!]_{!r_i}]\ f \bullet m = z\ [\mathsf{T}]) \\ \qquad \Rightarrow & \models [\mathsf{T}]\ M_1 :_m [\forall f.[[\![F]\!](f)\ \wedge\ (\wedge_i [\![U_i]\!](!r_i))]\ f \bullet m = z\ [\mathsf{T}]] \end{array}$$

But by (4.1) we have

$$\not\models [\mathsf{T}]\, M_2 :_m [\forall f.[[\![F]\!](f) \;\wedge\; (\wedge_i [\![U_i]\!](!r_i))]\, f \bullet m = z\, [\mathsf{T}]]$$

that is $M_1 \not\cong_{\mathcal{L}} M_2$, a contradiction. Thus we conclude $M_1 \cong M_2$, as required. $\square$

We mention a corollary of Theorem 4.19 which says that the strongest post condition always gives a CAP for a semi-closed value, after a definition.

Definition 4.20 *Given* $\Gamma;\Delta \vdash M : \alpha$ *and a TCA* C*, the set of* strongest post conditions of M w.r.t. M at u *(for total correctness), written* $\mathsf{sp}(C, M, u)$*, is the set of TCAs, say* C'*, such that: (1)* $[C]\, M :_u [C']$ *and (2) whenever* $[C]\, M :_u [C'']$ *we have* $C' \supset C''$*.*

Corollary 4.21 *Let* $A \in \mathsf{sp}(\mathsf{T}, V^{\Delta:\alpha}, u)$*. Then* A *characterises* V*.*

Proof We show V is the least element of the property described by A. Assume not, then there is W such that $W \not\sqsupseteq V$ but $[\mathsf{T}]\, W :_u [A]$. By Theorem 4.19, there is B such that $\models [\mathsf{T}]\, V :_u [B]$ and $\not\models [\mathsf{T}]\, W :_u [B]$. By assumption we have $A \supset B$. Hence $[\mathsf{T}]\, W :_u [B]$, a contradiction. $\square$

Note this says a strong post condition of T w.r.t. a semi-closed term V is always an up-closed set with the least element being (the congruence class of) V.

5 Reasoning Examples

5.1 Deriving Hoare Logic for Total Correctness

We first embed the standard proof rules of Hoare logic for total correctness with recursive procedures [46] in the logic presented above, thus establishing a precise link between the proposed logic and traditional Hoare logics for total correctness. Then we show a generalisation of these rules.

The syntax of programs is given as follows. Let $p, q, \ldots$ range over procedure labels.

$$\begin{array}{rcl} e & ::= & \mathtt{c} \mid\; !x \mid \mathtt{op}(e_1,...,e_n) \\ P, Q, .. & ::= & \mathtt{skip} \mid x := e \mid P;Q \mid \mathtt{if}\; e\; \mathtt{then}\; P\; \mathtt{else}\; Q \mid \mathtt{while}\; e\; \mathtt{do}\; P \\ & \mid & \mathtt{call}\; p \mid \mathtt{proc}\; p = P\; \mathtt{in}\; Q \end{array}$$

In $\mathtt{proc}\; p = P\; \mathtt{in}\; Q$, a procedure body P is named p, where we allow calls to p to occur in P. The reduction rules are standard [77], hence are omitted. Procedures are parameterless and do not return values. We still use the explicit dereference notation $!x$ since it clarifies the correspondence with imperative PCFv. Assertions, still ranged over by $C, C', \ldots$, are a proper subset of those of Section 2, having only natural numbers and references to storing natural numbers as data typed. Moreover, the new logic omits evaluation formulae

Let $\star \in \{\wedge, \vee, \supset\}$ and $\mathcal{Q} \in \{\forall, \exists\}$.

$$\begin{array}{rcl} e & ::= & i^{\mathsf{Nat}} \mid\; !x \mid \mathtt{n} \mid \mathtt{op}(e_1,...,e_n) \\ C & ::= & e_1 = e_2 \mid C_1 \star C_2 \mid \neg C \mid \mathcal{Q} i^{\mathsf{Nat}}.C \end{array}$$

Hereafter we shall safely confuse logical terms and expressions (as in Hoare logic). Moreover, auxiliary (function) variables are exclusively ranged over by $i, j, \ldots$

The judgement takes the shape $\Sigma \vdash [C]\,P\,[C']$, where $[C]\,P\,[C']$ is the standard Hoare triple and Σ is a finite map from procedural labels to pairs of formulae, writing each element of a map as a triple $[C]p[C']$. The meaning of $[C]p[C']$ is understood just as a Hoare triple, saying: *calling p at an initial state C will terminate with a final state C'.* The logic uses these triples as an assumption on the behaviour of procedures a program may use, and infer the resulting behaviour of the program.

Fig. 6 Hoare Logic with Recursive Procedure (total correctness)

$$[\mathit{Skip}]\,\frac{-}{\Sigma \vdash [C]\mathtt{skip}[C]} \qquad [\mathit{AssignH}]\,\frac{-}{\Sigma \vdash [C[e/!x]]x := e[C]} \qquad [\mathit{Seq}]\,\frac{\Sigma \vdash [C]P[C_0] \quad \Sigma \vdash [C_0]Q[C']}{\Sigma \vdash [C]\,P;Q\,[C']}$$

$$[\mathit{IfH}]\,\frac{\Sigma \vdash [C \wedge e]\,P_1\,[C'] \quad \Sigma \vdash [C \wedge \neg e]\,P_2\,[C']}{\Sigma \vdash [C]\,\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2\,[C']} \qquad [\mathit{While}]\,\frac{\begin{array}{c} C \wedge e \supset e' \gneqq 0 \\ \Sigma \vdash [C \wedge e \wedge e' = i]\,P\,[C \wedge e' \lneqq i] \end{array}}{\Sigma \vdash [C]\,\mathtt{while}\ e\ \mathtt{do}\ P\,[C \wedge \neg i]}$$

$$[\mathit{Call}]\,\frac{[C]p[C'] \in \Sigma}{\Sigma \vdash [C]\,\mathsf{call}\,p\ [C']} \qquad [\mathit{RecProc}]\,\frac{\begin{array}{c} \Sigma, [\exists j \lneqq i.C(j)]p[C_0] \vdash [C(i)]\,P\,[C_0] \\ \Sigma, [\exists i.C(i)]p[C_0] \vdash [C]\,Q\,[C'] \end{array}}{\Sigma \vdash [C]\mathtt{proc}\ p = P\ \mathtt{in}\ Q[C']}$$

$$[\mathit{Consequence\text{-}Aux}]\,\frac{[C_0]P[C_0'] \quad C \supset \exists \vec{j}.(C_0[\vec{j}/\vec{i}] \wedge (C_0'[\vec{y}/\vec{x}][\vec{j}/\vec{i}] \supset C'[\vec{y}/\vec{x}]))}{[C]\ M :_u [C']}$$

Figure 6 presents the proof rules. For simplicity of presentation, we use a single recursion in [*RecPro*] and mathematical induction in [*While*] and [*RecPro*] (their generalisation does not pose any technical difficulty). In [*While*] and [*RecPro*], we assume i, j are auxiliary and only occur in mentioned formulae and that the holes in $C(i)$ exhaust i. Among possible structural rules, we mention Kleymann's strengthened consequence rule [46], from which other known structural rules, such as the standard consequence rule and Hoare's adaptation rule, can be derived. In the rule, $\vec{i}$ (resp. $\vec{x}$) are the vector of auxiliary (resp. program) variables occurring in C_0 and C_0', while $\vec{j}$ (resp. $\vec{y}$) are the vector of fresh names of the same length as $\vec{i}$ (resp. $\vec{x}$).

We now embark on the embedding. The encoding of programs into imperative PCF is standard (procedure labels are simply taken to be variables).

$$[\![\mathtt{skip}]\!] \stackrel{\text{def}}{=} () \qquad [\![x := e]\!] \stackrel{\text{def}}{=} x := e \qquad [\![P;Q]\!] \stackrel{\text{def}}{=} [\![P]\!];[\![Q]\!] \ (\stackrel{\text{def}}{=} (\lambda().Q)P\ )$$
$$[\![\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q]\!] \stackrel{\text{def}}{=} \mathtt{if}\ e\ \mathtt{then}\ [\![P]\!]\ \mathtt{else}\ [\![Q]\!]$$
$$[\![\mathtt{while}\ e\ \mathtt{do}\ P]\!] \stackrel{\text{def}}{=} (\mu w.\lambda().\mathtt{if}\ e\ \mathtt{then}\ [\![P]\!];(w\,())\ \mathtt{else}\ ())\,()$$
$$[\![\mathtt{call}\ p]\!] \stackrel{\text{def}}{=} p() \qquad [\![\mathtt{proc}\ p = P\ \mathtt{in}\ Q]\!] \stackrel{\text{def}}{=} (\lambda p.[\![Q]\!])(\mu p.\lambda().[\![P]\!])$$

Note all commands have unit type. The judgement $\Sigma \vdash [C]P[C']$ is translated as:

$$[[\![\Sigma]\!] \wedge C][\![P]\!][C'] \qquad \text{with} \qquad [\![\emptyset]\!] \stackrel{\text{def}}{=} \mathsf{T} \text{ and } [\![\Sigma, [C]p[C']]\!] \stackrel{\text{def}}{=} [\![\Sigma]\!] \wedge [C]p \bullet ()[C']$$

Formulae are simply the subset of those for the imperative PCF. If we use the standard model of number theory [54, §3.1] for Hoare logic, validity of formulae also coincide for this subset of formulae. Thus the remaining task is to embed the proof rules. Below we show each rule in Figure 6 has a clean encoding into the logic for imperative PCF.

Proposition 5.1 (embedding of Hoare logic for total correctness) $\Sigma \vdash [C]P[C']$ *implies* $\vdash [[\![\Sigma]\!] \wedge [C]]\, [\![P]\!]\, [C']$.

The embedding is interesting because it immediately suggests new derived proof rules for imperative higher-order functions, in a coarser grain than the original ones. First let us consider how the while rule in Hoare logic can be extended to treat non-simple expressions as guard. The rule is to be considered as part of the proof rules for the imperative PCFv.

$$[\mathit{While\text{-}H}]\ \frac{[C]\,M :_b [B^b \wedge C] \quad C \wedge B[\mathsf{t}/b] \supset e' \gneq 0 \quad [C \wedge B[\mathsf{t}/b] \wedge e' = n]\,N\,[C \wedge e' \lneq n]}{[C]\,\mathtt{while}\,M\,\mathtt{do}\,N\,[C \wedge B[\mathsf{f}/b]]}$$

Note the rule can be used even when the guard M includes higher-order expressions, unlike the standard while rule. In that setting the while command can be considered as a macro, just as our preceding embedding does. An essentially identical inference proves its soundness through the soundness of the original rules.

Next we refine a recursion rule into the one for multiple recursion. The rule is easily encodable into the let-rec rule.

$$[\mathit{MRecProc}]\ \frac{\begin{array}{c}\Sigma, [\exists j \lneq i.C_1(j)]p_1[G_1], \dots [\exists j \lneq i.C_n(j)]p_n[G_n] \vdash [C(i)]\,P_h\,[G_h] \quad (1 \leq h \leq n) \\ \Sigma, [\exists j \lneq i.C(j)]p_1[G_1], \dots [\exists j \lneq i.C(j)]p_m[G_m] \vdash [C]\,Q\,[C']\end{array}}{\Sigma \vdash [C]\mathtt{proc}\ [p_1 = P_1, \dots, p_n = P_n]\ \mathtt{in}\ P[C']}$$

Similarly we can easily treat higher-order commands and expressions, based on the rules in Section 3, while respecting a distinction between expressions and commands (for the treatment of local variables, see Section 6).

5.2 Example Reasoning

Closure Factorial. Recall `closureFact` from Section 1. Its specification can be given as follows.

$$[\mathsf{T}]\ \mathtt{closureFact} :_u [\forall i^{\mathsf{Nat}}.\,[\mathsf{T}]\,u \bullet i\,[\,[\mathsf{T}]\,!y \bullet () = z\,[z = i!]\,]\,]$$

We use the following fact about factorials:

$$0! = 1 \quad \wedge \quad \forall i^{\mathsf{Nat}}.(i+1)! = (i+1) \times i!. \tag{5.1}$$

Let $A(g,i) \stackrel{\mathrm{def}}{=} [\mathsf{T}]g \bullet () = z[z = i!]$, $B(f,i) \stackrel{\mathrm{def}}{=} [\mathsf{T}]\,f \bullet i\,[\,[\mathsf{T}]\,!y \bullet () = z\,[z = i!]]$, and $B'(f,i) \stackrel{\mathrm{def}}{=} \forall j^{\mathsf{Nat}} \lneq i.B(f,j)$. By a straightforward application of the proof rules (see Appendix B for the detailed inference), we obtain, for $N \stackrel{\mathrm{def}}{=} \lambda x^{\mathsf{Nat}}.\,\mathtt{if}\ x = 0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2$:

$$[B'(f,i)]\ N :_u [\ \forall x^{\mathsf{Nat}}.(x = i\ \supset\ B(u,x))\,] \tag{5.2}$$

By applying the above we obtain:

$$[B'(f,i)]\ N :_u\ [B(u,i)] \tag{5.3}$$

We can now apply (Rec) to reach the required judgement.

Circular Factorial. Next we consider `circFact`:

$$\texttt{circFact} \stackrel{\text{def}}{=} x := \lambda z.\texttt{if } z = 0 \texttt{ then } 1 \texttt{ else } z \times (!x)(z-1)$$

Its specification may be written down as, under the typing $x : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat})$:

$$[\mathsf{T}]\ \texttt{circFact}\ [\exists g.(\forall i.[!x = g](!x) \bullet i = i![!x = g]\ \wedge\ !x = g)] \tag{5.4}$$

The specification says:

> *After executing* `circFact`*, x stores a procedure which would calculate a factorial if, as an assumption, x stores a program which has precisely that behaviour itself; and x does store that behaviour.*

The assertion tersely describes all we need to know about `circFact` including its circularity (for a precise understanding of this assertion, note $!x$ in the internal pre/post conditions of the evaluation formula is a hypothetical content of x, while $!x$ which nakedly occurs in the postcondition is the actual content of x, cf. §4.2, Page 13). The assertion makes it clear that calculation of a factorial by the stored procedure demands that x stores itself: if that stored procedure is stored in another variable, and if we change the content of x, it will no longer calculate a factorial.

For the derivation, let:

$$\begin{aligned} A(u,g,j) &\stackrel{\text{def}}{=} [!x = g]u \bullet j = j![!x = g]. \\ C(!x,g,i) &\stackrel{\text{def}}{=} \forall j \lneq i.\ A(!x,g,j) \wedge !x = g. \end{aligned}$$

We also set, for brevity:

$$M \stackrel{\text{def}}{=} \lambda y.\texttt{if } y = 0 \texttt{ then } 1 \texttt{ else } y \times (!x)(y-1)$$

Then a direct compositional inference leads to:

$$[\mathsf{T}]\ x := M\ [\ \forall yg.[C(!x,g,y)]!x \bullet y = y!\ [C(!x,g,y)]] \tag{5.5}$$

The key reasoning step is the following entailment:

$$\forall yg.[C(!x,g,y)]\ !x \bullet y = y!\ [C(!x,g,y)] \quad \supset \quad \exists g.\ (\forall i.A(!x,g,i) \wedge !x = g) \tag{5.6}$$

which is derived as follows:

$$\begin{aligned} &\forall yg.[C(!x,g,y)]\ !x \bullet y = y!\ [C(!x,g,y)] \\ &\quad \equiv \quad \forall y.\forall g.\,[\forall j \lneq y.A(g,g,j) \wedge !x = g]\ !x \bullet y = y!\ [\forall j \lneq y.A(g,g,j) \wedge !x = g] \\ &\quad \equiv \quad \forall g.\forall y.\,[\forall j \lneq y.A(g,g,j) \wedge !x = g]\ !x \bullet y = y!\ [\forall j \lneq y.A(g,g,j) \wedge !x = g] \\ &\quad \equiv \quad \forall g.\forall y.\ ((\forall j \lneq y.A(g,g,j)\ \supset\ [!x = g]\ !x \bullet y = y!\ [!x = g])) \qquad (\dagger,\ddagger) \\ &\quad \supset \quad \exists g.\ (\forall y.\ (\forall j \lneq y.A(g,g,j)\ \supset\ [!x = g]\ g \bullet y = y!\ [!x = g]) \wedge !x = g) \qquad (\star) \\ &\quad \stackrel{\text{def}}{=} \quad \exists g.(\forall y.\ (\forall j \lneq y.A(g,g,j)\ \supset\ A(g,g,y))\ \wedge\ !x = g) \\ &\quad \supset \quad \exists g.(\ \forall y.A(g,g,y)\ \wedge\ !x = g\ ) \\ &\quad \supset \quad \exists g.(\ \forall y.A(!x,g,y)\ \wedge\ !x = g\ ), \end{aligned}$$

In above, we use the following axioms of evaluation formulae in [41]:

$$
\begin{array}{lll}
(\dagger) & [A \wedge C]\, x \bullet y = z\, [C'] \equiv A \supset [C]\, x \bullet y = z\, [C'] & z \notin \mathsf{fv}(A) \\
(\ddagger) & [C_0] x \bullet y = z [C'_0] \supset [C]\, x \bullet y = z\, [C'] & \text{where } C \supset C_0 \text{ and } C'_0 \supset C'
\end{array}
$$

In $(\star)$, we have used the well-known axiom from predicate calculus with equality:

$$\forall x.A \quad \supset \quad A[y/x] \quad \equiv \quad \exists x.(A \wedge x = y)$$

which holds for an arbitrary y. By applying (5.5) to (5.6) we obtain (5.4), as required.

6 Discussion

Observational Completeness and Extensions The core of the paper answered the following question in the affirmative: *Can we build a Hoare logic for a programming language with higher-order state such that operational and axiomatic semantics coincide?* The key technical tool we used were *characteristic formulae* which capture the meaning of a program in a single pair of pre- and post-condition. Characteristic formulae arose in the context of concurrency theory [64], see [7] for an overview. An existence of characteristic formulae in a different theory tradition already suggests that the concept is not an artifact of the specific PCF-variant, but rather a general technique that can be used to build observationally complete program logics for substantially different kinds of programming languages (see [12]). We list our subsequent work related to the logic and observational completeness developed in this paper.

- Our work [37] refines the concept of characteristic formulae and presents observationally complete program logics with characteristic formulae for partial, as well as total correctness for call-by-value PCF. [37] also presents an observationally complete logic for call-by-value PCF with unrestricted higher-order state.
- The papers [79] extend the logic of the present paper to call-by-value PCF with unrestricted higher-order state (including aliasing) and unrestricted local memory, e.g. allowing programs like: `let` $x =$ `ref`(7) `in` $(x, \lambda f y.(x := f\, x\, y; !x))$. The resulting logic is for total correctness, is observationally complete and has characteristic formulae.
- In [9] a total correctness logic for call-by-value PCF extended with `callcc` and `throw` for unrestricted explicit control flow manipulation is presented. The logic is observationally complete, has characteristic formulae, and enables us to reason about the notorious `argfc` program `callcc` $\lambda k.($`throw` k $\lambda x.($`throw` $k\, \lambda y.x))$, which, when called once, returns twice.
- [13] presents an observationally complete logic with characteristic formulae for a variant of call-by-value PCF extended with meta-programming features.
- Finally, [11] introduces observationally complete Hoare logics with characteristic formulae for partial, total and generalised correctness for (essentially) arbitrary typed π-calculi. In this setting, typing disciplines are presented by logical axioms.

Characteristic formulae are interesting for another reason: if they can be inferred by induction on program syntax, as they can here and in all logics listed above, they enable a different, two-phased style of program verification. In conventional verification with program logics, typically, we build a proof tree whose root is of the form $[A]\ M :_m [B]$ where A, B express the desired program properties. Reasoning about programs is intertwined with reasoning in the ambient theory of mathematics, and Hoare's Consequence rule mediates between the two. With characteristic formulae, reasoning about programs has two separate phases.

- Computation of the characteristic assertion pair of a program, done by induction on the type derivation of a program.
- Verification of desired program properties from characteristic assertion pairs. This involves checking if the target properties are implied by its characteristic formula.

Note that tools for either phase can be specialised for their different purposes; in particular, tools for the second phase can be program language agnostic, and shared as 'back-ends' for different 'front-end' program logics.

One important task in making program logics usable for large-scale software verification is mechanisation. Recently Charguéraud has successfully developed program logics with characteristic formulae for PCF-like languages with and without state as extensions of higher-order logics [14, 15]. He gives shallow embeddings of his logics into Coq, and uses the embedding to verify a large number of highly non-trivial programs in the two-phased style described above.

6.1 Related Work

In the following we discuss related work focusing on logics for higher-order imperative languages. For comparisons in different contexts (for example, process logics, general aliasing and local references), see [10, 35, 36, 39].[4]

Equational Logics for Higher-Order Functions. Equational logics for the λ-calculi have been studied since the classical work by Curry and Church. LCF [24] augments the standard equational theory of the λ-calculus with Scott's fixed point induction. Our program logics for higher-order functions differ in that an assertion describes behavioural properties of programs rather than equates them, allowing specifications with arbitrary degrees of precision, as well as smoothly extending to non-functional behaviour.

The reasoning methods for λ-calculi have been studied focusing on the principles of parametricity using equational logics [4, 68]. The presented method differs in that it offers behavioural specifications for interface of a program, rather than directly equating or relating programs. It should however be noted that, for calculating validity of entailment, the present method does need to make resort to semantic arguments for polymorphic behaviours, see [39]. This suggests fruitful interplay between the present logical method, on the one hand, and the reasoning principles as developed in, and extending, [4, 68] on the other.

[4] This subsection was written by Kohei Honda. Since both reviewers and the editor are happy with this subsection in the first version, we keep this subsection in this revision to record his last writing on this topic.

Logical Expressiveness and Impossibility Result. Compositional program logics for imperative languages have been studied extensively since Hoare's seminal work. In late 1970s and early 1980s, there are a few attempts to extend Hoare logic to higher-order languages, mostly focusing on Algol and its derivatives. One of the basic works in this period is Clarke's work [16] (see also [18, §7.4.2.5]), which shows that a sound and (relatively) complete Hoare logic in the standard sense *cannot* exist for Algol-like (or Pascal-like) programming languages with the following set of features: (1) higher-order procedures, (2) recursion, (3) static scoping, (4) global variables, and (5) nested internal procedures. His argument can be briefly summarised as follows.

- Assume we have a sound and complete Hoare-like logic for partial correctness. This means we can prove $\{C\}P\{C'\}$ whenever it is true under any interpretation relative to true sentences of the underlying domain, cf. [17].
- Now assuming the language is standard first-order logic and take finite interpretations. Then validity of assertions is decidable. This makes provability in Hoare logic decidable. In particular, this holds for $\{\mathsf{T}\}P\{\mathsf{F}\}$, which witnesses P's divergence.
- But if the target language has the above five features, we can emulate a general computing device even under finite interpretations. This contradicts the recursiveness of validity of judgements, hence the proof system cannot be complete.

Clark's construction of general computing device under finite models relates to Jones and Muchnick's work in [44, 45], where they investigate decidability and complexity of programs with a fixed, finite number of memory locations, each of which can store only a finite amount of information with and without recursion (for example they showed [44] that undecidability can come from differences in the calling mechanisms).

Clarke's result indicates a fundamental discrepancy between the expressiveness of the assertion languages in Hoare logic for partial correctness (in the traditional sense) and the expressiveness of programming languages with rich features: the same simplification — making the data domain finite — has different effects on the tool for description (assertions) and the target of description (programs). Much subsequent work focuses on sublanguages of the Algol fragment Clarke proved incompleteness for, establishing their completeness.

How can we position the presented logic in the context of Clarke's work? We first note the following, which directly draw on Clarke's result. Below by "finite base types" we mean that Nat is interpreted as a non-trivial finite domain. By "static local variable" we mean a local variable declaration never exported outside of its scope.

Proposition 6.1 *The termination problem of the imperative PCFv in* §*2.1 extended with static local variables is undecidable under finite base types.*

Proof By Clarke's result (see also an alternative, and lucid, construction of Turing machine in Cousot's survey [18]). □

Note static local variables in the above sense can be easily captured in the present logic through the standard method in Hoare logic, cf.[8]. At this point we do not know whether imperative PCFv without static local variables has the same properties or not.

Proposition 6.1 indicates that Hoare-like logic in the traditional sense cannot be complete under finite models. The next result shows the other side of the coin, showing inherent complexity of the assertion language of the present logic. Below by *logic for the functional sublanguage* we mean the logic which only use the empty reference basis (i.e. semantically without stores and syntactically without dereferences).

Proposition 6.2 *The validity of assertions in the logic for the functional sublanguage is undecidable even under finite base types.*

Proof This result relies on:

1. The observational congruence of PCFv with finite bases and without recursion but with $\perp$ coincides with that of PCFv-programs with finite bases and with recursion.
2. For this language we can derive characteristic formulae following Section 4.

By adopting Loader's result [51], the contextual congruence of finitary PCFv with bottom (even without recursion) is undecidable. By the extensionality axiom noted in Section 2, this means that, in this finitary language, it is as difficult to calculate validity in this sublogic as calculating two programs are contextually equal or not.[5] $\square$

Corollary 6.3 *The validity of assertions in the present logic (for imperative PCFv) is undecidable even under finite base models.*

Proof By taking the empty reference basis. $\square$

Note this result crucially relies on direct description of higher-order behaviours in the logical language. While it is standard [27] to consider strong models (those which can represent all arithmetical relations) for total correctness, the above result shows how such description leads to inherent complexity of the presented logical language, under any non-trivial class of interpretations.

Program Logics for Sublanguages of Algol (1): Olderog's Analysis. In [62]. Olderog presents a sound and complete proof system for sub-languages of Algol with different variants of copy rules, treated uniformly based on the shape of call trees w.r.t. a given copy rule. Trakhtenbrot et al. [75] independently presented a sound and complete logic for an Algol-like language with copy rule. Later Olderog [63] presented a precise and uniform characterisation of existence of sound and (relatively) complete Hoare logic for sublanguages of a Pascal-like language, called $\mathcal{L}_{\text{pas}}$, which allows second-order procedures. His characterisation is amazingly simple: sound and complete Hoare-like logics exist for an admissible sublanguage of $\mathcal{L}_{\text{pas}}$ (here "admissibility" indicates closure under natural syntactic transformations respecting semantics inside $\mathcal{L}_{\text{pas}}$) if and only if its call trees are regular in the standard sense. An example of a program which does not have a regular call tree (even under finite interpretation), from [63], follows (we use letrec/let for readability).

$$\texttt{letrec}\ p = \lambda f.(\texttt{letrec}\ q = \lambda().f()\ \texttt{in}\ p(q); f())\ \texttt{in let}\ r = \lambda().\texttt{skip in}\ p(r)$$

[5] Loader's result is for call-by-name PCF with finite domains, with bottom and without recursion (the use of bottom is fundamental for his undecidability result). His argument is however easily converted to call-by-value PCF with finite base domains and bottom.

which is, modulo translation of let/letrec, easily a PCFv-program, strengthening our intuition behind Proposition 6.2.

As the above example shows, Olderog's results offer a deep analysis of the dynamics of languages with recursive higher-order procedures, uncovering structural information under Clarke's impossibility result. The same programming example also shows that imperative PCFv easily allows recursive calls which have nonregular call trees. If we aim at (at least) describing all semantic properties of a target programming language by the present program logic, then being able to describe behaviours with non-regular call structures may not be inhibited, at least as a starting point.

On its basis, however, we may pose the following question, based on Olderog's analysis: if we start from the use the presented logical language and imperative PCFv, how would the uniform restrictions considered in [62, 63] or analogous ones alter properties of the logic? One of our main concerns underlying this question is about tractability in reasoning (for example for model checking). We believe it is at least theoretically interesting and possibly pragmatically rewarding to reintroduce notions and results from his and others' studies on Algol-like languages in the present extended (and therefore far more intractable) setting.

Program Logics for Sublanguages of Algol (2): Damm and Josco's Logic. Algol and its sublanguages strictly separate commands from (first-order and higher-order) expressions. Further, variables only store first-order values such as integers. For this reason most of Hoare logics studied for these programming languages do not directly describe higher-order behaviour in assertions. One of the exceptions is work by Damm and Josco [19], where they use predicate variables (which represent e.g. postconditions) by instantiating them with a concrete predicate using a fixed correspondence in variables. For example, assume given an expression P of type $\alpha \Rightarrow Prg$ (Prg is the program type), they assert

$$\{C_{pre}\}P\{C_{post}\},$$

where C_{pre} and C_{post} are pre/post conditions of type $\alpha \Rightarrow Prg$, taking a predicate of type α. Thus the above formula in fact means that, for any expression Q of type α, and for any of its assertion in the shape similar to the standard most general formula [62], we have:

$$\{x = i\}Q\{C'\} \quad \supset \quad \{C_{post}(C')\}P(Q)\{C_{pre}(C')\}$$

which is now of type Prg, so that the judgement is an ordinary Hoare triple. There are three observations.

1. The use of a specific, and fixed, form of precondition of Q is crucial: since it wholly captures a state transformation by Q of interest, we can instantiate it into both pre/post conditions of the resulting command.
2. The instantiation is based on syntactic substitution of formulae using fixed variables, which works because the construction of higher-order formulae such as C_{post} and C_{pre} above reflects Algol's type structure: they are always built up from first-order state transformation one by one (so a formula of a higher-order type contains a sequence of substitutions broken down to first-order state transforms).

3. Because of (2), their approach is not directly extensible to stored higher-order procedures, so that (for example) the behaviour of `closureFact` and `circFact` cannot be asserted. More importantly, the framework may not allow description of generic higher-order behaviour like that of as `Map` and `App` as we did in Section 5 unless we alter the basic structure.

We believe the comparisons with our framework as given above (especially the third point) may suggest the effectiveness of evaluation formulae as a simple but powerful logical device to describe general stateful applicative behaviour.

Program Logics for Sublanguages of Algol (3): Halpern's Logic. German, Clarke and Halpern [22] and Halpern [26] studied completeness for Clarke's sub-language of Algol. Halpern [26] studied the language (called PRG83) using a separate class of assertion called *covering assertions* which roughly say which variables a program reads and writes. He then considers a judgement of the following form:

$$\mathcal{CA} \supset \{A\}P\{B\}$$

where $\mathcal{CA}$ is a covering assertion involving a program and identifiers it covers. His logic relies on the validity of such entailment, and, as such, is higher-order. He has shown, through the use of most general formulae for partial correctness, that his proof system is sound and complete with respect to strongly expressive models (i.e. those which can express weakest preconditions for arbitrary programs) and under the provision that all true judgements of the above form can be given by an oracle, He uses what he calls *store domains* where an element in a domain is equipped with its support (free names, or locations it uses), which is similar to the notion of models in Section 4. A main difference in this aspect is that, in our models, it is not an element but a domain which is equipped with free names, since type information of abstract values already includes its reference typing. We follow more recent approaches to operational and denotational semantics of programs which manipulate locations, as found in the work by Pitts and Stark [66].

Reynolds's Specification Logic. Specification logic by Reynolds [71] is a program logic for Idealised Algol which combine the traditions of both LCF and Hoare logic, where Hoare triples appear textually in assertions. It is a bold enterprise, since the logic aims to capture the whole of Idealised Algol including noninterference between expressions (needed to tame intractability of write effects in call-by-name evaluations).

The target language, Idealised Algol, is a purified form of Algol. As such, it has a strict separation between expressions (including abstraction, application and recursion) and commands (which is a special case of expressions), where only commands allow such constructs as loop and sequential composition. The judgement in Reynolds's logic (which he calls *specification*) uses, as its atomic formulae, a Hoare triple $\{C\}M\{C'\}$ (with M being a program text), equality of expressions, "noninterference" predicate and a "good variable" predicate, the latter two used for asserting on noninterference. These are combined with intuitionistic connectives (conjunction, entailment and falsity) and universal quantifiers over natural numbers. Note that, in this way, a judgement may contain many instances of program texts. Reynolds intends such a judgement to indicate a "predicate about environments in the sense of Landin", i.e. the set of all possible environments which satisfy the judgement.

Reynolds presented several proof rules. One interesting rule is essentially the following one (we write S for a judgement in Reynolds's logic and $S[M]$ for a judgement with a hole willed with an expression in it).

$$\frac{\vdash S[M] \quad M \cong N}{\vdash S[N]}$$

Note M can occur contravariantly in $S[\,\cdot\,]$. Other rules include, as in LCF, all standard logical inference rules, but they also combine rules for subtyping and the standard rules for Hoare triples (the assignment rule becomes complex due to the concern on noninterference). A major part of the efforts in [71] are done for formalising rules for noninterference. Subsequent studies on semantics of specification logics by O'Hearn [70], Tennent [74] and Ghica [23] also centre on precise formalisation of this notion.

Both specification logic and the logic studied in the present paper aim to capture a general class of imperative higher-order behaviours, albeit difference in the choice of languages. One technical similarity is a conceptual distinction between a store and an environment, which is explicit in the present logic because of the dereference notation. On the other hand, the main differences are:

1. Reynolds's logic is not (intended as) a compositional logic in Hoare's sense. This leads to two technical differences.
   - The present assertion language directly assert on and compositionally verify higher-order expressions, whereas Reynolds's logic does neither. This lack is partly compensated by the substitutivity rule listed above (note however this rule involves direct reasoning on $M \cong N$ at the level of programs).
   - Judgements in Reynolds's' logic may contain assertions on program texts whereas the present logic maintains strict distinction between a program and an assertion, the latter describing the behaviour of the former.
2. Effective reasoning principles for complex data types (starting from sums and products) is central to the present logic, which is not treated in specification logic. We believe their treatment may not be easy without using anchors.
3. There is also a minor technical difference in that the present logic is for total correctness while Reynolds's logic is for partial correctness, though much of Reynolds's technical development would equally work for total correctness, similarly the presented framework can cleanly accommodate partial correctness.

Reynolds's logic precedes the presented logic in that he tries to capture semantics of typed higher-order programs in a logical framework. As noted above, his logic does not (aim to) offer a compositional reasoning method for higher-order expressions and data structures, which is the main concern of the present work. An interesting topic which these comparisons may suggest is a possibility to extend the present framework to the logic for program development where we can combine programs and their specifications, as strongly advocated by Jifeng and Hoare [32]. As another interest, control of interference in the higher-order imperative call-by-name behaviours is central to Reynolds's logical framework as well as to subsequent studies on its semantics. It is an interesting subject of further study if, under the same setting as Reynolds, whether we can obtain a clean compositional logic following the present framework and its ramifications.

Other Related Work. For both typed and untyped λ-calculi, equational logics have been studied since the classical work by Church and Curry. LCF [24] augments the standard equational theories of the λ-calculus with Scott's fixpoint induction. In LCF-like logics, programs appear as terms of the logical language. These terms are equated by syntactic equations such as $\beta\eta$-conversions, for which statelessness of computation is essential. Mason [52] studies a LCF-like logic for imperative call-by-value functions, where imperative effects of programs are reasoned using small step reductions, a non-congruent syntactic equivalence and effect propagations. His logic is not (intended as) a compositional program logic but does allow certain contextual reasoning.

Dynamic Logic [28], introduced by Pratt [69] and studied by Harel and others [27], uses programs and predicates on them as part of formulae, facilitating detailed specifications of various properties of programs such as (non-)termination as well as intensional features. As far as we know, higher-order procedures have not been treated in Dynamic Logic, even though we believe part of the proposed method to treat higher-order functions would work consistently in their framework.

Names have been used in Hoare logic since an early work by Kowaltowski [47], and are found in the work by von Oheimb [76], Leavens and Baker [50] and Abadi and Leino [5], for treating parameter passing and return values. These works do not treat higher-order procedures and data types, which are uniformly captured in the present logic along with parameters and return values through the use of names. This generality comes from the fact that a large class of behaviours of programs are faithful representation as name passing processes which interact at names: our assertion language offers a concise way to describe such interactive behaviour in a logical framework.

Reynolds, O'Hearn and others [61, 72] study extensions of Hoare logic in which new logical connectives are used for reasoning about low-level operations such as garbage collection in the first-order setting. A clean logical treatment of low-level features and higher-order constructs would be an interesting topic for further study. One of the major aims of their work is to offer tractable reasoning for aliasing. This aspect of their work are extensively discussed in [10] and [79].

The characterisation of observational semantics by logical formulae are well-known in process logics [29] and is also discussed in Hoare logics [32, 55]. Nevertheless, none of the related work discussed above reports observational completeness in the sense of Theorem 4.19. We believe that, especially when a program logic treats assertions on higher-order programs (as in the present logic), precise correspondence between contextual behaviours and logical descriptions is important for various engineering concerns, for example substitutivity of modules through specifications. The notion of characteristic assertions in our sense is closely related with so-called most general formulae, cf. [8, 46].

The use of side-effect-free expressions when reasoning about assignment is a staple in compositional program logics. Freedom from side effects is however hard to maintain in a higher-order setting because of the complex interplay between higher-order procedures. The clean embedding of Hoare's assignment rule in §5 suggests that the presented framework effectively refines the standard approach while retaining its virtues in the original setting. It should be noted that in the context of an integrated verification framework JML [2], Leavens and others report engineering significance of the principle of the use of side-effect free expressions in practice. Experiment of the use of the proposed extensions in practical engineering settings would be an interesting subject for further study.

For solving some of the central issues associated with program development on a formal basis, a study on theories, calculi and practice of program/data refinement aims to build methodologies by which one can develop programs starting from general specifications and, through refinement of successively more concrete specifications, reach an executable program, cf. [20, 30, 32] (this line of study includes integrated software development frameworks such as VDM [43] and more recent Z notation[78]). One of the ideas strongly advocated in [32] in this context is a specification language in which we can combine programs and formulae using logical connectives and program constructs. While some trials to obtain such a calculus for higher-order procedures exist (for example see [59]), no tractable solutions have been known (Hoare and Jifeng [30, 32] noted difficulties to apply their framework to higher-order objects). Can the present theory contribute to the development of a simple, general and practical theory of refinement for programs and data types? By doing so, can it add anything to the existing integrated framework such as those using, for example, Z notation [78]? The semantic analysis of the proposed logic and its extensions, as partly discussed in Section 4, would offer a useful foundation for such an inquiry.

The origin of the assertions and judgements introduced in the present work is the logic for typed π-calculi [34, 36] where linear types lead to a compositional process logic. The known precise embeddings of high-level languages into these typed π-calculi can be used to determine the shape of name-based logics like the one presented here for the embedded languages. Once found, they can be embedded back with precision into the originating process logics. [33, 34, 36] discuss process logics and their relationship to the program logics in detail.

References

1. C– home page. http://www.cminusminus.org.
2. The Java Modeling Language (JML) home page. http://www.jmlspecs.org.
3. The Haskell home page. http://haskell.org.
4. Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. *TCS*, 121(1-2), 1993.
5. Martín Abadi and Rustan Leino. A logic for object-oriented programs. In *Verification: Theory and Practice*, pages 11–41. Springer-Verlag, 2004.
6. Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. & Comp.*, 163:409–470, 2000.
7. Luca Aceto and Anna Ingólfsdóttir. Characteristic formulae: From automata to logic. BRICS Report Series RS-07-2, BRICS, Department of Computer Science, University of Aarhus, 2007.
8. K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.
9. Martin Berger. Program Logics for Sequential Higher-Order Control. In *Proc. FSEN*, pages 194–211, 2009.

10. Martin Berger, Kohei Honda, and Nobuko Yoshida. A Logical Analysis of Aliasing in Imperative Higher-Order Functions. *J. Funct. Program.*, 17(4-5):473–546, 2007.
11. Martin Berger, Kohei Honda, and Nobuko Yoshida. Completeness and Logical Full Abstraction in Modal Logics for Typed Mobile Processes. In *Proc. ICALP*, pages 99–111, 2008.
12. Martin Berger, Kohei Honda, and Nobuko Yoshida. Logics for Imperative Higher-Order Functions with Aliasing and Local State: Three Completeness Results. Draft, available from `http://www.doc.ic.ac.uk/~yoshida/paper/threecompleteness.pdf`, 2008.
13. Martin Berger and Laurence Tratt. Program Logics for Homogeneous Metaprogramming. In *Proc. LPAR*, pages 64–81, 2010.
14. Arthur Charguéraud. Program verification through characteristic formulae. In *Proc. ICFP*, pages 321–332, 2010.
15. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proc. ICFP*, pages 418–430, 2011.
16. Edmund Clarke. Programming language constructs for which it is impossible to obtain good hoare axiom systems. *J. ACM*, 26(1):129–147, 1979.
17. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
18. Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.
19. Werner Damm and Bernhard Josko. A Sound and Relatively* Complete Hoare-Logic for a Language With Higher Type Procedures. *Acta Inf.*, 20:59–101, 1983.
20. Edsger W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. In *Proceedings of the international conference on Reliable software*, pages 2–2.13, 1975.
21. Robert W. Floyd. Assigning meaning to programs. In *Symp. in Applied Mathematics*, volume 19, 1967.
22. Steven M. German, Edmund M. Clarke, and Joseph Y. Halpern. Reasoning About Procedures as Parameters. In *IBM Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 206–220, 1981.
23. Dan R. Ghica. Semantical Analysis of Specification Logic, 3: An Operational Approach. In *Proc. ESOP*, volume 2986 of *LNCS*, pages 264–278, 2004.
24. Mike Gordon, Robin Milner, and Christopher Wadsworth. Edinburgh LCF. LNCS 78, Springer Verlag, 78.
25. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
26. Joseph Y. Halpern. A good hoare axiom system for an algol-like language. In *11th POPL*, pages 262–271. ACM Press, 1984.
27. David Harel. Proving the correctness of regular deterministic programs. *TCS*, 12(16):61–81, 1980.
28. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logigc*. The MIT Press, 2000.
29. Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
30. C. A. R. Hoare. Proof of correctness of data representations. pages 385–396, 2002.
31. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
32. Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
33. Kohei Honda. Sequential process logics: Soundness proofs. Available at: www.dcs.qmul.ac.uk/~kohei/logics, November 2003. Typescript, 50 pages.
34. Kohei Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.
35. Kohei Honda. From process logic to program logic (full version of [34]). Available at: www.dcs.qmul.ac.uk/~kohei/logics, November 2004. Typescript, 52 pages.

36. Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2004. Typescript, 234 pages.
37. Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and Relative Completeness of Logics for Higher-Order Functions. In *Proc. ICALP*, pages 360–371, 2006.
38. Kohei Honda and Nobuko Yoshida. Game-Theoretic Analysis of Call-by-Value Computation. *TCS*, 221:393–456, 1999.
39. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04, 6th ACM-SIGPLAN International Confernce on Prin ciples and Practice of Decralative Programming*, pages 191–202, 2004.
40. Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*, pages 270–279, 2005.
41. Kohei Honda, Nobuko Yoshida, and Martin Berger. An Observationally Complete Program Logic for Imperative Higher-Order Functions. Technical Report DTR-13-2, Imperial College, Department of Computing, 2013.
42. J. Martin E. Hyland and C. H. Luke Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
43. C B Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., 1986.
44. Niel Jones and S Muchnick. Complexity of finite memory programs with recursion. *Journal of ACM*, 25(2):312–321, 1977.
45. Niel Jones and S Muchnick. Even simple programs are hard to analyze. *Journal of ACM*, 24(2):338–350, 1977.
46. Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.
47. Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.
48. Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
49. Peter Landin. A correspondence between algol 60 and church's lambda-notation. *Comm. ACM*, 8:2, 1965.
50. Gary Leavens and Alber L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *FM'99: World Congress on Formal Methods*. Springer, 1999.
51. Ralph Loader. Finitary PCF is not decidable. *Theor. Comput. Sci.*, 266(1-2):341–364, 2001.
52. I A. Mason. A first order logic of effects. *Theoretical Computer Science*, 185(2):277–318, 1997.
53. Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *JFP*, 1(3):287–327, 1991.
54. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
55. Albert R. Meyer. Floyd-Hoare logic defines semantics (preface version). In *LICS'86*, 1986.
56. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *POPL'88*, 1988.
57. Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.
58. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
59. David A. Naumann. Soundness of data refinement for a higher-order imperative language. *Theor. Comput. Sci.*, 278(1-2):271–301, 2002.
60. P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.
61. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, pages 268–280, 2004.

62. Ernst-Rüdiger Olderog. Sound and complete hoare-like calculi based on copy rules. *Acta Inf.*, 16:161–197, 1981.
63. Ernst-Rüdiger Olderog. A characterization of hoare's logic for programs with pascal-like procedures. In *15th Theory of Computing*, pages 320–329, 1983.
64. David Park. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, pages 167–183, 1981.
65. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
66. Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *HOOTS'98*, CUP, pages 227–273, 1998.
67. Gordon D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI, Aarhus University, 1981.
68. Gordon D. Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 1993.
69. Vaughn R. Pratt. Six lectures on dynamic logic. In *Foundations of Computer Science III, Part 2*, number 109 in Mathematical Centre Tracts, pages 53–82, 1980.
70. John C. Reynolds. *The Craft of Programming*. Prentice-Hall International, 1981.
71. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
72. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.
73. Zhong Shao. An overview of the FLINT/ML compiler. In *1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
74. Robert D. Tennent. Semantical analysis of specification logic. *I & C*, 85(2):135–162, 1990.
75. Boris A. Trakhtenbrot, Joseph Y. Halpern, and Albert R. Meyer. From denotational to operational and axiomatic semantics for algol-like languages. In *CMU Workshop on Logic of Programs*, pages 474–500, 1984.
76. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001.
77. Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.
78. Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., 1996.
79. Nobuko Yoshida, Kohei Honda, and Martin Berger. Logical reasoning for higher-order functions with local state. *Logical Methods in Computer Science*, 4(2), 2008.

A Observational Completeness: Detailed Proofs

A.1 Supplement to Proof of Lemma 4.16

Assume $\Delta \vdash M_1 \ncong M_2 : \alpha$ and let $C[\,\cdot\,]$ and $\vec{V}$ be such that, for example:

$$(C[M_1],\ \vec{r} \mapsto \vec{V}) \Downarrow \qquad \text{and} \qquad (C[M_2],\ \vec{r} \mapsto \vec{V}) \Uparrow$$

which means, through the β_V-equality:

$$(WM_1,\ \vec{r} \mapsto \vec{V}) \Downarrow \qquad \text{and} \qquad (WM_2,\ \vec{r} \mapsto \vec{V}) \Uparrow$$

where we set $W \stackrel{\text{def}}{=} \lambda x.C[x]$. Note the convergence in $(WM_1,\ \vec{r} \mapsto \vec{V}) \Downarrow$ takes, by the very definition, only a finite number of reductions. Let it be n. Then (occurrences of)

λ-abstractions in W and $\vec{V}$ can only be applied up to n times, similarly for other destructors. Using this, we transform these programs into FCF values maintaining the above property. We illustrate the basic ideas. First, all recursion used in W and $\vec{V}$ are n times unfolded through the standard unfolding (e.g., given $\lambda x.M$, the 0th unfolding is Ω (cf. Convention 4.2), the 1st unfolding is $M[\Omega/x]$, the 2nd unfolding is $M[M[\Omega/x]/x]$, etc.), still maintaining convergence. Similarly β_v-redexes can be eliminated by performing reductions n times, while the "if" statement can be made less defined by pruning all branches which do not contribute to convergence. Variables of higher-order types are η-converted, while all Nat-typed variables are replaced by constants combined with the case construct, through inspection of their concrete usage during reductions. Applications are replaced by let-applications. For details of the transformation, see below. We now obtain (semi-closed) FCF values, which we set to be F and $\vec{U}$. Since the convergence/divergence behaviour of $(FM_1, \vec{r} \mapsto \vec{U})$ has not changed in comparison with $(WM_1, \vec{r} \mapsto \vec{V})$, and because $(FM_2, \vec{r} \mapsto \vec{U})$ is more prone to divergence than $(WM_2, \vec{r} \mapsto \vec{V})$, we still obtain:

$$(FM_1,\ \vec{r} \mapsto \vec{U}) \Downarrow \qquad \text{and} \qquad (FM_2,\ \vec{r} \mapsto \vec{U}) \Uparrow$$

as required.

In the following we present the translation of W into its corresponding FCF used above. We write:

1. $\eta^{\alpha}(x)$ for the η-expansion of a variable x of type α using `let`'s in the obvious way (e.g. $\eta^{\mathsf{Nat} \Rightarrow \mathsf{Nat}}(y) \overset{\text{def}}{=} \lambda x.\mathtt{let}\ z = yx\ \mathtt{in}\ z$). If $\alpha = \mathsf{Nat}$ then it is identity.
2. $\mathtt{case}^{\omega}\ x\ \mathtt{of}\ \langle \mathtt{i} : M_i \rangle$ is the case construct which allows infinite branching (which we later convert into finite branching).

Let the number of reduction steps needed to converge be n. The translation is in five stages, as given below. For brevity we assume only a single first-order operator, $\mathtt{succ}(M)$, is used in programs: generalisation to inclusion of other first-order operators is immediate.

Stage 1: unfolding. Unfold each recursion in W n times (as illustrated in the main proof). Let the resulting term be W'.

Stage 2: let-translation. On W' we perform the translation $\langle\langle W',\ x,\ x\rangle\rangle$ where $\langle\langle M,\ y,\ N\rangle\rangle$

is given by induction on M as follows.

$$\begin{array}{rcl}
\langle\!\langle x,\ y,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & N[x/y] \\
\langle\!\langle \mathtt{n},\ y,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & N[\mathtt{n}/y] \\
\langle\!\langle \lambda x.M,\ y,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & N[\lambda x.\langle\!\langle M,\ z,\ z\rangle\!\rangle/y] \\
\langle\!\langle \mathtt{succ}(M),\ y,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & \langle\!\langle M,\ x,\ N[\mathtt{succ}(x)/y]\rangle\!\rangle \\
\langle\!\langle M_1 M_2,\ y,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & \langle\!\langle M_1,\ f,\ \langle\!\langle M_2,\ x,\ \mathtt{let}\ y = f x\ \mathtt{in}\ N\rangle\!\rangle\rangle\!\rangle \\
\langle\!\langle \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2,\ y,\ N'\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & \langle\!\langle M,\ x,\ \mathtt{if}\ x\ \mathtt{then}\ \langle\!\langle N_1[\mathtt{t}/x],\ y,\ N'\rangle\!\rangle \\
 & & \qquad\qquad\quad \mathtt{else}\ \langle\!\langle N_2[\mathtt{f}/x],\ y,\ N'\rangle\!\rangle\rangle\!\rangle \\
\langle\!\langle \mathtt{let}\ y = M_1\ \mathtt{in}\ M_2,\ z,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & \langle\!\langle M,\ y,\ \langle\!\langle M_2,\ z,\ N\rangle\!\rangle\rangle\!\rangle \\
\langle\!\langle !x,\ y,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & \mathtt{let}\ y = !x\ \mathtt{in}\ N \\
\langle\!\langle x := M_1; M_2,\ y,\ N\rangle\!\rangle & \stackrel{\mathrm{def}}{=} & \langle\!\langle M_1,\ z,\ x := z; \langle\!\langle M_2,\ y,\ N\rangle\!\rangle\rangle\!\rangle
\end{array}$$

We can check that $\langle\!\langle M,\ y,\ N\rangle\!\rangle$ keeps or replicates β_V-redexes in M and N, possibly changing $(\lambda x.L_1)L_2$ into $(\lambda x.L_1')z$. We now repeat the following transformations n times.

1. Firstly, reduce all β_V-redexes in the resulting term, including those under λ-abstraction, simultaneously.
2. Secondly, letting the resulting term be (say) V, we calculate $\langle\!\langle V,\ x,\ x\rangle\!\rangle$ again, and let the resulting term be used for the next round (if we have not reached n).

After the n-th round, if there still remain any λV-redexes in the term, we replace them with ω (of the same type). Let the resulting program be W''.
Stage 3: η-conversion. On W'', we perform the following two transformations consecutively.

– Every subterm of W'' of the form $\lambda x^{\alpha\Rightarrow\beta}.C[x]_i$ where $C[x]_i$ enumerates all free occurrences of x in the body (if any) except those of the form xM, is simultaneously transformed into:

$$\lambda x^{\alpha\Rightarrow\beta}.C[\eta^{\alpha\Rightarrow\beta}(x)]_i,$$

thus eliminating all occurrences of arrow type variables except those occurring in the function positions of let-applications.

– Every subterm of W'' of the form $\lambda x^{\mathsf{Nat}}.C[x]_i$, where $C[x]_i$ enumerates all free occurrences of x in the body (if any), is simultaneously transformed into:

$$\lambda x^{\mathsf{Nat}}.\mathtt{case}^{\omega}\ x\ \mathtt{of}\ \langle\, \mathsf{n} : C[\mathsf{n}]_i \,\rangle\ ,$$

thus eliminating all occurrences of Nat-typed variables.

Let the resulting term be $W^{\dagger}$.

Stage 4: case branch pruning. By inspecting reductions starting from $W^{\dagger}M_1$ reaching convergence, we can witness which numerals (if ever) are fed to each subterm occurring in $W^\dagger$ of the form $\lambda x^{\mathsf{Nat}}.M$. This decides a finite number of numerals ever fed to each abstraction of the form $\lambda x^{\mathsf{Nat}}.M$ in $W^\ddagger$. By pruning all unnecessary branches, we now transform all infinite case constructs to finite constructs without changing behaviour (if none is fed we turn it into $\Omega \stackrel{\text{def}}{=} \lambda x.\omega$). Let the resulting term be $W^\ddagger$.

Stage 5: final cleanup. First observe that, in each subterm of the form $\mathtt{succ}(N)$, N is either a numeral or again of the form $\mathtt{succ}(N')$ (with obvious generalisation when other first-order operators are involved). Hence we can completely calculate away each successor (and other first-order operators).

Let the resulting term be F. Then FM_1 can precisely mimic reductions of WM_1 to reach convergence. The same transformation is performed for each V_i in $\vec{V}$, obtaining a FCF, named U_i. This concludes the transformation of W an $\vec{V}$ into desired FCFs.

A.2 Proofs for Proposition 4.17

In the subsequent proof of Proposition 4.17, we use the following notations for brevity.

Notation A.1

1. We write $(\xi \cdot u : M, \sigma) \Downarrow (\xi \cdot u : V, \sigma')$ for $(M\xi, \sigma) \Downarrow (V, \sigma')$.
2. We write $(M_1, \sigma_1) \sqsubseteq (M_2, \sigma_2)$ when we have $(M_i, \sigma_i) \Downarrow (V_i, \sigma_i')$ $(i = 1, 2)$ such that $V_1 \sqsubseteq V_2$ and $\sigma_1 \sqsubseteq \sigma_2$.

As before, it is easy to inductively verify (soundness). Below we show (MTC) and (closure).

(Numeral) Let $C \stackrel{\text{def}}{=} \vec{r} = \vec{i}$ and $C' \stackrel{\text{def}}{=} C \wedge u = \mathsf{n}$. MTC is trivial. For closure, assume $[C_0]M :_u [C']$ with $C_0 \supset C$ and let $(\xi, \sigma) \models C_0$. Then

$$(\xi \cdot u : M\xi, \sigma) \Downarrow (\xi \cdot u : V, \sigma_0') \models C' \quad \supset \quad V \cong \mathsf{n} \wedge \sigma \cong \sigma'$$

where $\sigma \cong \sigma'$ is by noting C' says the state is unchanged from the precondition C. Since $(\xi \cdot u : \mathsf{n}, \sigma) \Downarrow (\xi \cdot u : \mathsf{n}, \sigma)$, we are done.

(Case-n) Let $F' \stackrel{\text{def}}{=} \mathtt{case}\ x\ \mathtt{of}\ \langle \mathtt{n_i} : F_i \rangle_i$, $C \stackrel{\text{def}}{=} \vee_i (x = \mathsf{n}_i \wedge C_i)$ and $C' \stackrel{\text{def}}{=} \vee_i (x = \mathsf{n}_i \wedge C_i')$. By (IH), assume (C_i, C_i') satisfies (MTC) and (closure) w.r.t. F_i at u, for each i. Let ξ' be a model for the assumed basis and $\xi \stackrel{\text{def}}{=} \xi'/x$. For MTC we reason:

$$\begin{array}{rcl} (F'\xi', \sigma) \Downarrow & \Leftrightarrow & \vee_i (\xi'(x) = \mathsf{n}_i \ \wedge \ (F_i\xi, \sigma) \Downarrow) \\ & \Leftrightarrow & \vee_i (\xi'(x) = \mathsf{n}_i \ \wedge \ (\xi, \sigma) \models A_i \ ) \\ & \Leftrightarrow & \xi \models A. \end{array}$$

For (closure), let $E \supset C$ and assume $\models [E]M :_u [C']$. We have $E \wedge x = \mathsf{n_i} \ \supset \ C_i \wedge x = \mathsf{n_i}$. Note also we have, noting $x = \mathsf{n_i}$ is stateless:

$$\models [E \wedge x = \mathsf{n_i}]M :_u [C_i] \tag{A.1}$$

We can now reason:

$$\begin{array}{rcll}
(\xi', \sigma) \models E & \supset & \exists i. (\, \xi' \models E \wedge x = \mathsf{n}_i) & \\
& \supset & \exists i. (\xi(x) = \mathsf{n}_i \;\wedge\; F_i\xi \;\sqsubseteq\; M\xi) & \text{(IH)} \\
& \supset & F'\xi' \;\cong\; F_i\xi \;\sqsubseteq\; M\xi. &
\end{array}$$

(Omega) Straightforward.

(Abstraction) Similar to (Numeral) above.

(Dereference) Let $F' \stackrel{\text{def}}{=} \texttt{let}\ x = !y\ \texttt{in}\ F$ and assume by induction that (C, C') is a strong CAP of F at u. First we show $C[x/!x]$ is an MTC for F'. Below we assume ξ, σ etc. are appropriately typed.

$$\begin{array}{rcll}
(F'\xi,\ \sigma) \Downarrow & \Leftrightarrow & (F(\xi \cdot x : \sigma(y)),\ \sigma) \Downarrow & \text{(reduction)} \\
& \Leftrightarrow & (\xi \cdot x : \sigma(y),\ \sigma) \models C & \text{(IH: } C \text{ is an MTC for } F) \\
& \Leftrightarrow & (\xi,\ \sigma) \models C[!y/x] &
\end{array}$$

Next we show the closure property.

Below let $\xi' = \xi \cdot x : \sigma(y)$ and $C_0 \supset C[!y/x]$.

$$\begin{array}{ll}
\models [C_0]M[C'] \;\wedge\; (\xi, \sigma) \models C_0 & \\
\qquad\qquad \supset & \models [C \wedge C_0]M[C'] \;\wedge\; (\xi', \sigma) \models C \wedge C_0 \\
\qquad\qquad \supset & (F\xi', \sigma) \;\sqsubseteq\; (M\xi', \sigma) \\
\qquad\qquad \supset & (F', \sigma) \;\sqsubseteq\; (M\xi', \sigma)
\end{array}$$

The third line is by the closure condition for (C, C'), by being a strong CAP of F by our induction hypothesis.

(Assignment) Let

$$F' \stackrel{\text{def}}{=} x := U\,; F \qquad C_0 \stackrel{\text{def}}{=} \forall z.(A \supset C[z/!x]) \tag{A.2}$$

Further by induction we stipulate:

(IH1) (C, C') satisfies (MTC) and (closure) w.r.t. F at u;
(IH2) $(\top,\ A)$ satisfies (MTC) and (closure) w.r.t. U at z, assuming the auxiliary names in A are empty, without loss of generality.

From (IH2) we infer:

$$\forall \xi, \sigma.\ (\xi \cdot z\!:\!U\xi,\ \sigma) \models A \tag{A.3}$$

Hence also:

$$\forall \xi, \sigma.\ (\xi, \sigma) \models \exists z.A. \tag{A.4}$$

Assume ξ, σ etc. are appropriately typed and recall $\sigma[x \mapsto V]$ indicates the result of updating the content of x in σ with V.

Fig. 7 Asserted Programs for the Proof of Closure in Assignment Rule

$$
\begin{array}{l}
[C_0]\ \texttt{let}\ y = !x\ \texttt{in} \\
\qquad [C_0[y/!x]]\ x := U\ ; \\
\qquad\quad [C \wedge A[!x/z][y/!x]] \quad x := y\ ;\ M \quad :_u\ [C']
\end{array}
$$

$$
\begin{array}{l}
[C_0]\ \texttt{let}\ y = !x\ \texttt{in} \\
\qquad [C_0[y/!x]]\ x := U\ ; \\
\qquad\quad [C' \wedge A[!x/z][y/!x]] \quad F \quad :_u\ [C']
\end{array}
$$

Above we choose y to be fresh and recall $[\mathsf{T}]\ U :_z [A]$. Two coloured parts are inequated by the closure condition from the induction hypothesis (the lower is less defined).

We first show (C_0, C') is an MTC for F' under the given inductive hypotheses. We infer, for some I:

$$
\begin{array}{rcll}
(F'\xi,\ \sigma)\Downarrow & \Leftrightarrow & (F\xi,\ \sigma[x \mapsto U\xi])\Downarrow & \text{(reduction)} \\
& \Leftrightarrow & (\xi,\ \sigma[x \mapsto U\xi]) \models C & \text{(IH1)} \\
& \Leftrightarrow & (\xi \cdot z\!:\!U\xi,\ \sigma[x \mapsto U\xi]) \models A\ \wedge\ C & \text{(by (A.3) above)} \\
& \Leftrightarrow & (\xi \cdot z\!:\!U\xi,\ \sigma) \models A\ \wedge\ C[z/!x] & \text{(substitution)} \\
& \Leftrightarrow & (\xi \cdot z\!:\!U\xi,\ \sigma) \models A\ \wedge\ \forall z.(A \supset C[z/!x]) & (*) \\
& \Leftrightarrow & (\xi,\ \sigma) \models \exists z.A\ \wedge\ \forall z.(A \supset C[z/!x]) & \text{(by (A.3) above)} \\
& \Leftrightarrow & (\xi,\ \sigma) \models \forall z.(A \supset C[z/!x]) & \text{(by (A.4) above)}
\end{array}
$$

In each line above, a comment on the right-hand side of a formulae caters for both directions of implications. The logical equivalences directly connect the condition for convergence to the precondition under a given model, (ξ, σ). For $(*)$, the "if" direction (upwards) is immediate. For the "then" direction, we derive the second conjunct of the subsequent from that of the precedent. For an arbitrary (well-typed) W:

$$
\begin{array}{lll}
\multicolumn{3}{l}{(\xi\cdot z\!:\!U\xi,\ \sigma) \models C[z/!x]\ \wedge\ (\xi\cdot z\!:\!W, \sigma) \models A} \\
\Rightarrow & (\xi\cdot z\!:\!U\xi,\ \sigma) \models C[z/!x]\ \wedge\ [\mathsf{T}]\ W :_z [A] & \text{(definition of } \models) \\
\Rightarrow & (\xi\cdot z\!:\!U\xi,\ \sigma) \models C[z/!x]\ \wedge\ U\xi\ \sqsubseteq\ W & \text{(IH2)} \\
\Rightarrow & (\xi\cdot z\!:\!W,\ \sigma) \models C[z/!x] & \text{(IH1, } C \text{ is a TCA at } !x)
\end{array}
$$

For closure, we let $C_0 \supset \exists z.A \wedge \forall z.(A\ \supset\ C[z/!x])$ and assume:

$$[C_0]\ M :_u\ [C']. \qquad \text{(A.5)}$$

We use the following programs. We set $F' \stackrel{\text{def}}{=} x := U; F$ as before.

$$
\begin{array}{rcl}
N & \stackrel{\text{def}}{=} & \texttt{let}\ y = !x\ \texttt{in}\ x := U; x := y; M \\
L & \stackrel{\text{def}}{=} & \texttt{let}\ y = !x\ \texttt{in}\ F'
\end{array}
$$

Immediately:

$$N\ \cong\ M \qquad \text{and} \qquad L\ \cong\ F'. \qquad \text{(A.6)}$$

We start with an assertion on the subprogram of N, $x := y;M$, which we are going to compare with F. We first observe:

$$[C_0[y/!x]]\ x := y\ [C_0] \tag{A.7}$$

Combined with the assumption (A.5), we reach:

$$[C_0[y/!x]]\ x := y\,;M :_u [C'] \tag{A.8}$$

Further, as we have seen for the main inference for MTC, we have:

$$(\xi{\cdot}z : U\xi,\ \sigma[x \mapsto U\xi]) \models C \quad \Leftrightarrow \quad (\xi,\ \sigma) \models C_0. \tag{A.9}$$

Hence by **(IH2)** we reach, writing ξ' for $\xi{\cdot}z\,{:}\,U\xi$:

$$\forall \xi, \sigma.\ (\xi',\ \sigma[x \mapsto U\xi]) \models C[y/!x] \quad \supset \quad (F\xi',\ \sigma) \sqsubseteq (\ (x := y;M)\xi',\ \sigma) \tag{A.10}$$

We now reason, writing further $\sigma' = \sigma[x \mapsto U\xi]$:

$$\begin{array}{lcll}
(\xi,\sigma) \models C_0 & \supset & (\xi',\sigma') \models C_0[y/x] & \text{(A.9)} \\
 & \supset & (F\xi',\ \sigma') \sqsubseteq (\ (x := y;M)\xi',\ \sigma') & \text{(A.10)} \\
 & \supset & (F'\xi',\ \sigma) \sqsubseteq (\ (x := U;x := y;M)\xi',\ \sigma) & \text{(reduction)} \\
 & \supset & (L\xi,\ \sigma) \sqsubseteq (\ N\xi,\ \sigma) & \text{(reduction)} \\
 & \supset & (F'\xi,\ \sigma) \sqsubseteq (\ M\xi,\ \sigma) & \text{(A.6)}
\end{array}$$

(Let-Application) Let $F' \stackrel{\text{def}}{=} \texttt{let}\ x =\ fU\ \texttt{in}\ F\ \xi_0 = \vec{y} : \vec{V}$ and $\xi = \xi_0 \cdot f : W$, as well as $\sigma = \vec{r} \mapsto \vec{V}$. We assume:

(IH1) (C, C') satisfies (MTC) and (closure) w.r.t. u for F.
(IH2) (T, A) satisfies (MTC) and (closure) w.r.t. z for U.

We also let

$$C_1 \stackrel{\text{def}}{=} \exists \vec{j}.\ (\ !\vec{r} = \vec{j}\ \wedge\ \forall z.[A \wedge\ !\vec{r} = \vec{j}] f \bullet z = x[C]\ )$$

First we show C_1 is an MTC for F'. By (IH2) we have $\models [\mathsf{T}]U :_z [A]$ hence for any ξ_0 (omitting auxiliary I):

$$\xi_0 \cdot z\,{:}\,U\xi_0 \models A \tag{A.11}$$

Below we write $(\xi{\cdot}x : M, \sigma) \Downarrow (\xi{\cdot}x : V, \sigma')$ when $(M\xi, \sigma) \Downarrow (V, \sigma')$.

$$\begin{array}{lll}
(F'\xi,\sigma) \Downarrow & & \\
\Leftrightarrow & (\xi_0 \cdot x : WU,\ \sigma) \Downarrow (\xi \cdot x : S,\ \sigma) \models C & \text{(IH1, (A.11))} \\
\Leftrightarrow & \forall U_1 \sqsupseteq U\xi_0\ \supset\ (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma'_1) \models C & (C \text{ TCA at } x) \\
\Leftrightarrow & z : U_1 \cdot \xi_0 \models A\ \supset\ (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma'_1) \models C & \text{(IH2)} \\
\Leftrightarrow & z : U_1 \cdot \xi_0 \models A\ \supset\ (z : U_1 \cdot \xi \cdot \vec{j} : \vec{V},\ \sigma) \models [!\vec{r} = \vec{j}] f \bullet z = x[C] & \text{(Def-eval)} \\
\Leftrightarrow & (\xi \cdot \vec{j} : \vec{V},\ \sigma) \models [A \wedge !\vec{r} : \vec{j}] f \bullet z = x[C] & (\dagger) \\
\Leftrightarrow & (\xi,\ \sigma) \models \forall \vec{j}.(!\vec{r} = \vec{j} \supset [A \wedge !\vec{r} : \vec{j}] f \bullet z = x[C]) &
\end{array}$$

The last line's "then" direction is because, if $\vec{j}$ are not mapped to what are equivalent to $\vec{v}$, the premise of the entailment does not hold.

For the closure condition, let $\Gamma;\Delta \vdash M : \alpha$. Further let C_0 be such that:

$$C_0 \supset C_1$$

and assume:

$$[C_0]M :_u [C']. \tag{A.12}$$

Let a vector of names $\vec{z}$ be fresh below. We write $\mathtt{let}\ \vec{z} = !\vec{r}\ \mathtt{in}\ F$ for a sequence of let-derefs and $\vec{r} := \vec{V}$ for a sequence of assignments.

$$M_0 \quad \stackrel{\text{def}}{=} \quad \mathtt{let}\ \vec{z} = !\vec{r}\ \mathtt{in}\ \mathtt{let}\ x = yU\ \mathtt{in}\ (\vec{r} := \vec{z}\ ;\ M)$$

By checking the reduction we have $M \cong M_0$, hence we hereafter use M_0 instead of M without loss of precision. Now assume: $(\xi,\ \sigma) \models C_0$. By (A.12) we have:

$$\begin{aligned}(\xi{\cdot}u : M_0\xi,\ \sigma) &\longrightarrow^* (\xi{\cdot}u : (\vec{r} := \sigma(\vec{r}); M)\xi,\ \sigma_0) \models C \\ &\longrightarrow^* (\xi{\cdot}u : M\xi,\ \sigma) \models C_0 \\ &\longrightarrow^* (\xi{\cdot}u : V',\ \sigma') \models C'\end{aligned}$$

As the above reduction indicates, we can check:

$$[C]\,(\vec{r} := \sigma(\vec{r}); M)\xi,\ \sigma_0) :_u [C'] \tag{A.13}$$

We are almost there. Observe, by $\models [C]F :_u [C']$:

$$(\xi{\cdot}u : F'\xi,\ \sigma) \longrightarrow^* (\xi{\cdot}u : F\xi,\ \sigma_0) \models C \longrightarrow^* (\xi{\cdot}u : V'',\ \sigma'') \models C'$$

By (IH1) and (A.13) we know: $(\xi{\cdot}u : V'',\ \sigma'') \sqsubseteq (\xi{\cdot}u : V',\ \sigma')$, as required.

B Detailed Proof Derivations of Reasoning Examples

This section lists the detailed derivations omitted from Section 5. We use the following simple rules which are easily derivable in the proof rules in Section 3:

$$[\mathit{Seq'}]\ \frac{[C]\,M\,[C_0] \quad [C_0]\,N :_u [C']}{[C]\,M; N :_u [C']} \qquad [\mathit{Simple}]\frac{-}{[C[e/u]]e :_u [C]}$$

Closure Factorial The following derivation starts from the left branch of the conditional, followed by its right branch. We omit trivial application of (Consequence). (Sub) and (Mult) are proof rules for subtraction and multiplication given as $[Op]$ in Figure 3.

	$[1 = x!]\ 1 :_m\ [m = x!]$	(Const)
2.	$[B'(f,i) \wedge x = i \wedge x = 0]\ 1 :_m\ [m = x!]$	((5.1), Conseq)
3.	$[B'(f,i) \wedge x = i \wedge x = 0]\ \lambda().1 :_m\ [A(m,x)]$	(Abs)
4.	$[B'(f,i) \wedge x = i \wedge x = 0]\ y := \lambda().1 :_m\ [A(!y,x)]$	(Assign)
5.	$[B'(f,i) \wedge x = i \wedge x \neq 0]\ f :_m\ [B'(m,i)]$	(Var)
6.	$[B'(f,i) \wedge x = i \wedge x \neq 0]\ x - 1 :_n\ [n = (x-1)]$	(Simple, Conseq)
7.	$[B'(f,i) \wedge x = i \wedge x \neq 0]\ f(x-1)\ [A(!y,x-1)]$	(App)
8.	$[A(!y,x-1)]\ !y :_m\ [A(m,x-1)]$	(Deref)
9.	$[A(!y,x-1)]\ (!y)() :_v\ [v = (x-1)!]$	(8. Const, App)
10.	$[A(!y,i-1)]\ (!y)() \times x\ :_z\ [z = x!]$	(9, Var, Mult, (5.1), Conseq)
11.	$[B'(f,i) \wedge x = i \wedge x \neq 0]\ f(x-1)\ ;\ (!y)() :_z\ [z = x!]$	(7, 10, Seq)
12.	$[B'(f,i) \wedge x = i \wedge x \neq 0]\ \lambda().(\ f(x-1)\,;\,(!y)()\ ) :_m\ [A(m,x)]$	(Abs)
13.	$[B'(f,i) \wedge x = i \wedge x \neq 0]\ y := \lambda().(\ f(x-1)\,;\,(!y)()\ ) :_u\ [A(!y,x)]$	(Assign)
14.	$[B'(f,i) \wedge x = i]\ \mathtt{if}\ x = 0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u\ [A(!y,x)]$	(4, 13, IfH)
15.	$[B'(f,i)]\ \lambda x^{\mathsf{Nat}}.\mathtt{if}\ x = 0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u$ $[\ \forall x^{\mathsf{Nat}}.\ [x = i]\, u \bullet i\, [[\mathsf{T}]\ !y \bullet () = z\, [z = i!]\ ]$	(Abs)
16.	$[B'(f,i)]\ \lambda x^{\mathsf{Nat}}.\mathtt{if}\ x = 0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u$ $[\ \forall x^{\mathsf{Nat}}.(x = i\ \supset\ B(u,x))\,]$	(e5, Conseq)
17.	$[B'(f,i)]\ \lambda x^{\mathsf{Nat}}.\mathtt{if}\ x = 0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u\ [B(u,i)\,]$	(Conseq)
18.	$[\mathsf{T}]\ \mu f^{\mathsf{Nat} \Rightarrow \mathsf{Unit}}.\lambda x^{\mathsf{Nat}}.\mathtt{if}\ x = 0\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u\ [\forall i^{\mathsf{Nat}}.B(u,i)\,]$	(Rec)

Circular Factorial We set, for brevity:

$$M \overset{\text{def}}{=} \lambda y.\mathtt{if}\ y = 0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1)$$

We now infer, letting y be typed with Nat and omitting simple applications of Consequence Rule:

1.	$[C(!x,g,y)\ \wedge\ y=0]\ 1 :_m [m=y!\ \wedge\ C(!x,g,y)]$	(Simple)
2.	$[C(!x,g,y)\ \wedge\ \neg y=0]\ y\times(!x)(y-1) :_m [m=y!\ \wedge\ C(!x,g,y)]$	(Simple, App)
3.	$[C(!x,g,y)]\ \mathtt{if}\ y=0\ \mathtt{then}\ 1\ \mathtt{else}\ y\times(!x)(y-1) :_m [m=y!\ \wedge\ C(!x,g,y)]$	(IfH)
4.	$[\mathsf{T}]M :_u [\ \forall gy.[C(!x,g,y)]u\bullet y=y!\ [C(!x,g,y)]\ ]$	(Abs, $\forall$)
5.	$[\mathsf{T}]\ x:=M\ [\ \forall yg.[C(!x,g,y)]!x\bullet y=y!\ [C(!x,g,y)]\ ]$	(Assign)
6.	$[\mathsf{T}]\ \mathtt{circFact}\ [\ \exists g.\ (\forall i.A(!x,g,i)\wedge !x=g)\ ]$	(Conseq)

The application of (Consequence) in Line 6 uses the the entailment in the main section.