

# An Observationally Complete Program Logic for Imperative Higher-Order Functions\*

Kohei Honda<sup>†</sup>

Nobuko Yoshida<sup>‡</sup>

Martin Berger<sup>†</sup>

## Abstract

*We propose a simple compositional program logic for an imperative extension of call-by-value PCF, built on Hoare logic and our preceding work on program logics for pure higher-order functions. A systematic use of names and operations on them allows precise and general description of complex higher-order imperative behaviour. The logic offers a foundation for general treatment of aliasing and local state on its basis, with minimal extensions. After establishing soundness, we prove that valid assertions for programs completely characterise their behaviour up to observational congruence, which is proved using a variant of finite canonical forms. The use of the logic is illustrated through reasoning examples which are hard to assert and infer using existing program logics.*

## 1. Introduction

Imperative extensions of higher-order functions, syntactically embodied by imperative extensions of the  $\lambda$ -calculus, have been one of the major topics in the study of semantics and types of programming languages for decades. They are a cornerstone of typed functional programming languages such as ML and Haskell and are central to the semantic analysis of procedural, object-oriented and even low-level languages [1, 14, 33, 36, 41]. The significance of combining imperative features and higher-order functions lies in their distilled presentation of key elements of sequential program behaviour, amenable for theoretical inquiry. This analytical nature makes it possible to develop rigorous operational semantics [25, 32, 38], a rich class of type disciplines [32, 36] and powerful operational reasoning techniques [28, 37].

Given these achievements, a natural question is if we can carry out a similar development for logical methods for reasoning, in particular those in the tradition of Hoare logic [12, 18]. In Hoare logic, assertions on programs offer a method for precisely describing properties of programs independent from the latter’s textual details, with proof rules enabling verification of valid assertions following the syntactic structure of target programs. Hoare logic has however been mainly developed for first-order imperative programs: its extension to accommodate general higher-order procedures has been known to be a subtle problem [6, 10, 30, 31].

The present paper introduces a simple compositional

program logic for an imperative extension of call-by-value PCF, built on Hoare logic and our preceding work on logics for pure higher-order functions [19, 22]. The assertions in the logic precisely describe behaviour of imperative higher-order procedures up to the observational equivalence, while proof rules enable compositional derivation of valid assertions. As far as we know, this is the first time a compositional program logic for imperative higher-order functions in full type hierarchy and with stored procedures has been developed. The logical articulation of higher-order behaviour is rigorously stratified, starting from pure functions [19, 22] and treating each significant imperative element, including state change, aliasing and local state, with an incremental enrichment of the assertion language and proof rules. The logic enjoys clean semantic status in the sense that valid assertions for a program precisely characterise its observational behaviour up to the contextual congruence [17, 30].

A syntactically simple extension of the Floyd-Hoare tradition for treating higher-order behaviour is that assertions in our logic not only talk about first-order data stored in imperative variables, as in Hoare’s logic and its standard extensions, but also about arbitrary higher-order imperative behaviours, which may be fed as arguments to procedures, denoted by functional variables and stored in imperative variables. Having programs’ behaviour as part of the universe of discourse is essential for reasoning about practical programs since functionalities of a higher-order program often crucially depend on the combined behaviour of the programs it uses. Our logical language fully embraces higher-order behaviours and data structures as target of description, which is done by naming behaviours by variables and asserting on them, rather than having their textual representation (programs) in assertions.

Let us present three simple, but non-trivial programs, clean logical description of whose behaviours is set to be one of the challenges in our present inquiry.

```
closureFact  $\stackrel{\text{def}}{=} \mu f^{\text{Nat} \Rightarrow \text{Unit}}. \lambda x^{\text{Nat}}. \text{if } x = 0$   
  then  $y := \lambda(). 1$   
  else  $y := \lambda(). (f(x - 1); x \times (!y)())$ 
```

Above and henceforth we use notations from standard textbooks [15, 36].  $()$  is the unique constant of type `Unit` and  $\lambda().N$  denotes  $\lambda z^{\text{Unit}}.N$  with  $z$  fresh. When invoked as e.g.

\*Work partially supported by EPSRC GR/R03075/01, GR/T04236/01, GR/S55538/01, GR/T04724/01 and GR/T03208/01. <sup>†</sup> Queen Mary, University of London. <sup>‡</sup> Imperial College London.

`closureFact` 3, the program stores a procedure in the imperative variable  $y$ . If we further 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 intension is that this final value should be the factorial of 3. The observable behaviour of `closureFact` can be informally described as: *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, but terser, factorial program, using Landin's idea [26] to realise a recursion by circular references.

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

It is easy to see that, after executing `circFact`,  $(!x)n$  returns the factorial of  $n$ . In more detail, 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 an inherent circularity of this description — How can we logically describe such a behaviour, and how can we derive it compositionally?

As the third example, let us consider:

$$\text{scheduler} \stackrel{\text{def}}{=} \text{map } (\lambda y. (\alpha \Rightarrow \text{Unit}) \times \alpha. (\pi_1(y)(\pi_2(y))))$$

where `map` is the standard higher-order map function:

$$\text{map} \stackrel{\text{def}}{=} \lambda f^{X \Rightarrow Y}. \mu m^{\text{List}(X) \Rightarrow \text{List}(Y)}. \lambda l^{\text{List}(X)}. \text{case}(l) \text{ of Nil} \Rightarrow \text{Nil} \mid x :: y \Rightarrow (f x) :: (m y)$$

Above  $x :: y$  is the list whose head is  $x$  and whose tail is  $y$ . The program `scheduler` receives a list of jobs, where each job is a pair of a function and its argument, and executes these functions with corresponding arguments sequentially. Assuming each function may have side effects, what would be the specification of the scheduler, parameterised by properties of stored programs, and how can we derive it from the program text? A possible informal description would be: *Given a list of jobs  $\langle f_1, x_1 \rangle, \langle f_2, x_2 \rangle, \dots, \langle f_{n-1}, x_{n-1} \rangle$ , if applying  $f_1$  to  $x_1$  changes the state  $\sigma_1$  to the state  $\sigma_2$ , applying  $f_2$  to  $x_2$  changes the state  $\sigma_2$  to the state  $\sigma_3$ , and so on, and finally applying  $f_{n-1}$  to  $x_{n-1}$  changes  $\sigma_{n-1}$  to  $\sigma_n$ , then feeding the scheduler with this job list starting from  $\sigma_1$  will eventually reach the state  $\sigma_n$ .* Compositional reasoning about such a program should treat higher-order functions, recursion, closures and products to derive an intended assertion from a program text. The proposed logic offers a

simple language to specify such complex behaviour with precision, combined with syntax directed proof rules for deriving judgements compositionally.

**Summary of Technical Results** In the following we summarise the main technical results of the paper.

1. Introduction of a compositional program logic for higher-order functions with global state, extending the logic for pure higher-order functions studied in [19], allowing natural descriptions of complex imperative higher-order behaviours and their compositional verification.
2. Study of the semantic foundations of the logic with respect to a naturally defined model. After establishing soundness of the proof rules, sound and complete characterisation of observational equivalence by validity is proved, using a proof method inspired by game semantics [5, 21, 23].
3. Exploration of the proposed assertional method and its proof rules through reasoning examples, including an illustration of verifications for three programming examples presented above.

The full version [2] presents detailed definitions, further results, more examples and all missing proofs.

## 2. Preview

This section illustrates the key ideas of assertions for imperative higher-order functions, starting from a brief review of the logic for pure functions presented in [19, 22].

**Pure Higher-Order Functions.** In the present approach to program logics, behaviour is asserted by naming them. Consider a simple program which computes a doubling function,  $N \stackrel{\text{def}}{=} \lambda x^{\text{Nat}}. x + x$ , where  $\text{Nat}$  is the type of natural numbers. If we apply  $N$  to 5, 10 is returned. More generally, the result of applying  $N$  to any natural number is always even. To represent these properties using logical formulae, we do not mention  $N$  itself, but describe its properties by *naming* it as, say,  $f$ . Thus we can write  $f \bullet 5 = 10$  as a property of  $N$ , named as  $f$ . Similarly we can write

$$\forall x^{\text{Nat}}. \text{Even}(f \bullet x) \tag{2.1}$$

where  $\text{Even}(n)$  is the predicate saying  $n$  is even (e.g.  $\text{Even}(x) \stackrel{\text{def}}{=} \exists n. (x = 2 \times n)$ ). The operator  $\bullet$  is left associative and non-commutative, and may be understood as an analogue of application in applicative structures. Formulae may be combined using all the standard logical connectives and quantifiers, just as in Hoare logic.

Using these formulae (ranged over  $C, C', D, \dots$ ), the judgement of the logic has the following shape.

$$\{C\}M :_f \{C'\}$$

which can be read as: if  $M$ , named as  $f$ , can rely on  $C$  as the behaviour of an environment, then the program combined with the environment can guarantee  $C'$ . The name  $f$  is called *anchor*. It can be any fresh name not occurring in  $M$  and  $C$ . An anchor is used to represent  $M$ 's point of operation, hence of specification. As an example, the specification for  $N$  is:

$$\{\top\}N :_f \{\forall x^{\text{Nat}}. \text{Even}(f \bullet x)\} \quad (2.2)$$

which says that the program  $N$  named  $f$ , under the trivial assumption  $\top$ , satisfies  $\forall x^{\text{Nat}}. \text{Even}(f \bullet x)$ . By having names in assertions, we can compositionally derive a specification with a non-trivial assumption on higher-order variables based on a simple operation: when the function  $f$  is applied to an argument, the result  $f \bullet x$  is *peeled-off* and replaced by a new anchor name  $u$ . E.g., we can derive

$$\{\forall x^{\text{Nat}}. \text{Even}(f \bullet x)\} f 3 :_u \{\text{Even}(u)\}$$

from two smaller specifications (1)  $\{\forall x^{\text{Nat}}. \text{Even}(f \bullet x)\} f :_m \{\forall x^{\text{Nat}}. \text{Even}(m \bullet x)\}$  (an instance of the axiom for variable:  $f$  named as  $m$  satisfies the same predicate as the one assumed for  $f$ ); and (2)  $\{\top\} 3 :_m \{m = 3\}$  (“3” named as  $m$  satisfies a predicate  $m = 3$ ) and by combining them together as  $(\forall x^{\text{Nat}}. \text{Even}(f \bullet x) \wedge m = 3) \supset \text{Even}(f \bullet m)$  (which is a simple instance of  $(A(x, x) \wedge x = y) \supset A(x, y)$ , the standard axiom in predicate logic [29, §2.8]). The same framework works for higher-order programs where multiple variables share assumptions.

$$\{\forall x^{\text{Nat}}. \text{Even}(f \bullet x)\} f 3 + f(f 5) + 1 :_u \{\text{Odd}(u)\} \quad (2.3)$$

where  $\text{Odd}(n)$  says  $n$  is odd. Now by combining two specifications for  $N$  in (2.2) and  $L \stackrel{\text{def}}{=} f 3 + f(f 5) + 1$  in (2.3), we arrive at:

$$\{\top\} \text{let } f = N \text{ in } L :_u \{\text{Odd}(u)\} \quad (2.4)$$

where  $\text{let } f = N \text{ in } L$  is encoded as  $(\lambda f. L)N$ . The property which is guaranteed by  $N$  is simply plugged into the assumption for  $L$ . This derivation is similar to a composition rule of Hoare logic, where we infer  $\{C\}P_1; P_2\{C'\}$  from  $\{C\}P_1\{C_1\}$  and  $\{C_1\}P_2\{C'\}$ .

**Mutable Higher-Order Functions.** The idea of naming behaviours is naturally extended to stateful computation. A typical example is the following specification of a program that reads the number 7 from a global storage cell  $x$  and then returns 9.

$$\{!x = 7\} 2 + !x :_u \{u = 9 \wedge !x = 7\}$$

where the logical term  $!x$  represents dereferencing  $x$  [32]. The resulting behaviour is located at the anchor name  $u$ . The assertion says: the program  $2 + !x$  returns 9 whenever  $x$  initially stores 7, and it does not change this content of  $x$ . As another example, this time with side-effects:

$$\{!x = 3\} x := (2 + !x) ; !x :_u \{u = 5 \wedge !x = 5\}$$

where “;” is sequential composition (encodable into call-by-value application).

We now move to assertions describing more complex behaviour where functions cause side-effects during evaluation. Let  $W \stackrel{\text{def}}{=} \lambda x. (w := (1 + !w) ; x + x)$ , slightly modified from  $\lambda x. x + x$  in the previous paragraph. Recalling  $L$  from (2.3), we further let  $M \stackrel{\text{def}}{=} \text{let } f = W \text{ in } L$ . Now this function not only satisfies  $\text{Odd}(u)$ , but also *changes a memory cell*  $w$  when invoked. Hence we would expect the following:

$$\{!w = 0\} \text{let } f = W \text{ in } L :_u \{\text{Odd}(u) \wedge !w = 3\} \quad (2.5)$$

How can we specify the behaviour of  $W$  to reach (2.5)? A simple method is to attach pre and post-conditions to invocations of functions by an argument, and assert them as a single predicate. Thus we write:

$$\{C\} f \bullet x \searrow_u \{C'\}$$

This assertion reads: if the state of memory and the environment satisfy  $C$ , the invocation of  $f$  with an argument  $x$  yields a value named  $u$  and a final state, together satisfying  $C'$ . “ $\searrow$ ” indicates the evaluation of  $f \bullet x$  resulting in  $u$ , which is asymmetric unlike the equality  $e = e'$ . This is due to the non-reversibility of state change. Based on this idea,  $W$  named as  $f$  has the following specification:

$$\text{EvenS}(w, f) \stackrel{\text{def}}{=} \forall x. \forall i. \{!w = i\} f \bullet x \searrow_u \{\text{Even}(u) \wedge !w = i + 1\}$$

$\text{EvenS}(w, f)$  specifies a procedure which, when invoked, would not only increment  $w$  but also return an even number: if  $f$  is called when  $!w = 0$ , then  $f$ 's return value is even and  $!w = 1$ . In fact from  $\text{EvenS}(w, f)$  we can derive:

$$\forall x. \{!w = 0\} f \bullet x \searrow_u \{\text{Even}(u) \wedge !w = 1\}$$

using standard axioms for universal quantification. Now assume  $f$  satisfies the above specification. Then we can derive the following judgement.

$$\{\text{EvenS}(w, f) \wedge !w = 0\} f 5 :_v \{\text{Even}(v) \wedge !w = 1\}$$

The key idea here is that when the function (named  $f$ ) is applied to the argument 5, not only is the result  $u$  replaced by a new anchor  $v$ , but we also *split* the assumption ( $\text{EvenS}(w, f)$ ) in two pieces, its pre-condition  $C$  added to the pre-condition of the judgement and its post-condition  $C'$  in the post-condition of the judgement. Repeating this, we

can derive (2.5) in a compositional way, using essentially the same `let`-rule as for stateless functions.

When working with higher-order functions, assertions with pre/post-conditions can be nested repeatedly and may appear in the pre/post-conditions of other assertions. For example, let  $V \stackrel{\text{def}}{=} \lambda y.(!x)y$ . If  $x$  stores a function with side effects, like  $W$  above, then calling  $V$  may involve writing to memory. Thus we may assert:

$$\{ \top \} \lambda y.(!x)y :_u \{ \{ \text{EvenS}(w, !x) \wedge !w = 0 \} \\ u \bullet n \searrow z \{ \text{Even}(z) \wedge \text{EvenS}(w, !x) \wedge !w = 1 \} \}$$

which says: if  $V$  is applied to a natural number  $n$  under the condition that  $x$  stores a function which satisfies  $\text{EvenS}(w, u)$ , and  $w$  initially stores 0, then resulting term evaluates to an even number, with  $w$ 's final state being 1.

A merit of the present approach in comparison with existing methods is that it can directly assert on the combined behaviour of two or more (possibly higher-order) procedures. For example, consider the following program:

$$M \stackrel{\text{def}}{=} \lambda x^{\text{Nat}}. (y := x ; g(f) ; g(f) ; !y + 1) \quad (2.6)$$

where  $f$  and  $g$  are of types  $\text{Unit} \Rightarrow \text{Unit}$  and  $(\text{Unit} \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$ , respectively. Assume we *only* know the abstract property of  $f$  and  $g$  which says: if we apply  $g$  to  $f$ , then the content of  $y$  changes its parity, i.e. if it is initially even then it becomes odd and vice versa. The proposed logic formally describes this property as follows, omitting return values:

$$\{ \text{Odd}(!y) \} f \bullet g \{ \text{Even}(!y) \} \wedge \{ \text{Even}(!y) \} f \bullet g \{ \text{Odd}(!y) \}.$$

Let us denote the above assertion by  $A(fg)$ . Then a property of  $M$  may be asserted as:

$$\{ A(fg) \} M :_u \{ \forall x^{\text{Nat}}. \{ \text{Even}(x) \} u \bullet x \searrow z \{ \text{Odd}(z) \wedge \text{Even}(!y) \} \} \quad (2.7)$$

which says: under the assumption about  $f$  and  $g$  as given, if the argument is even, then the result is odd and the content of  $y$  is even. Let the above post-condition be  $\text{Even\_then\_Odd}(u, y)$ . From the same pre-condition, we can also infer the dual property  $\text{Odd\_then\_Even}(u, y) \stackrel{\text{def}}{=} \forall x^{\text{Nat}}. \{ \text{Odd}(x) \} u \bullet x \searrow z \{ \text{Even}(z) \wedge \text{Odd}(!y) \}$  or even a conjunction of the two,  $\text{Even\_then\_Odd}(u, y) \wedge \text{Odd\_then\_Even}(u, y)$ , as its post-condition. This specification relies on the property of the combined behaviour of  $f$  and  $g$  and demonstrates a practical benefit of having named higher-order procedures and specifications of their behaviour as an integral part of assertions: we can transparently specify and reason about the complex interplay among two or more procedures which may call each other and which as a whole demonstrate a specific behaviour of interest. Further examples will be treated in § 6, after formally introducing the logic and proof rules.

### 3. Logic for Imperative Call-by-Value PCF

#### 3.1. Imperative PCF

This subsection briefly reviews the programming language we use, call-by-value PCF with unit, sums and products, augmented with imperative variables (henceforth often

called *references*). The grammar of programs is standard [36], given below. We assume an infinite set of *variables*, also called *names*, ranged over by  $x, y, z, \dots$

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

The grammar uses types  $(\alpha, \beta, \dots)$ , which are given later. Constants are ranged over by  $c$ . Examples include the unit  $()$ , natural numbers  $n$  and booleans  $b$  (either  $f$  or  $t$ ).  $\text{op}(\vec{M})$  (where  $\vec{M}$  is a vector of programs) is a standard  $n$ -ary arithmetic or boolean operation, such as  $+$ ,  $-$ ,  $\times$ ,  $=$  (equality of two numbers/booleans),  $\neg$  (negation),  $\wedge$  and  $\vee$ .  $!x$  dereferences  $x$  while  $x := N$  is assignment. All these constructs are standard, cf. [15, 36].

A *store*  $(\sigma, \sigma', \dots)$  is a finite map from imperative variables to values. We write  $\sigma[x \mapsto V]$  for the store which maps  $x$  to  $V$  and otherwise agrees with  $\sigma$ . The call-by-value reduction, written  $(M, \sigma) \longrightarrow (M', \sigma')$ , is standard [15, 36]. We only list the rules for assignment and dereference. Below  $\sigma(x)$  and  $\sigma[x \mapsto V]$  indicate  $x \in \text{dom}(\sigma)$ .

$$(!x, \sigma) \rightarrow (\sigma(x), \sigma) \quad (x := V, \sigma) \rightarrow ((), \sigma[x \mapsto V])$$

We also write  $(M, \sigma) \Downarrow (V, \sigma')$  for  $(M, \sigma) \rightarrow^* (V, \sigma')$ , and  $M \Downarrow V$  for  $(M, \emptyset) \rightarrow^* (V, \emptyset)$ .

The grammar of types is also standard [15, 36].

$$\begin{array}{ll} \alpha, \beta & ::= \text{Unit} \mid \text{Bool} \mid \text{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \\ \rho & ::= \alpha \mid \text{Ref}(\alpha) \end{array}$$

We call  $\alpha, \beta, \dots$  *value types*, and  $\text{Ref}(\alpha), \dots$  *reference types*. Reference types are restricted to carrying only non-reference types. Lifting this restriction leads to a distinct class of behaviour which deserves a logical treatment in its own right, see § 7 for further discussions on these extensions. Note a reference can still carry arbitrary higher-order procedures. A *basis* is a finite map from names to types.  $\Gamma, \Gamma', \dots$  range over bases whose codomains are value types, while  $\Delta, \Delta', \dots$  range over bases whose codomains are reference types.  $\text{dom}(\Gamma)$  (resp.  $\text{dom}(\Delta)$ ) denotes the domain of  $\Gamma$  (resp. of  $\Delta$ ). The typing rules are standard [36] and omitted. We write  $\Gamma; \Delta \vdash M : \alpha$  when  $M$  has type  $\alpha$  under  $\Gamma$  and  $\Delta$ , with  $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$ .

#### 3.2. Terms and Formulae

The logical language is that of the first-order logic with equality [29, § 2.8] augmented with an assertion for the evaluation of stateful expressions. The grammar of terms

and formulae follows. The first set of expressions  $(e, e', \dots)$  are *terms* while the second set are *formulae*  $(A, B, C, \dots)$ .

$$\begin{aligned} e & ::= x^\alpha \mid () \mid \mathbf{c} \mid \mathbf{op}(\vec{e}) \\ & \quad \mid \langle e, e' \rangle \mid \pi_i(e) \mid \text{inj}_i^{\alpha+\beta}(e) \mid !(x^{\text{ref}(\alpha)}) \\ C & ::= e = e' \mid \neg C \mid C \wedge C' \mid C \vee C' \mid C \supset C' \\ & \quad \mid \forall x^\alpha. C \mid \exists x^\alpha. C \mid \{C\} e \bullet e' \searrow_x \{C'\} \end{aligned}$$

Terms, which are from [19, 22] except  $!x$ , include all the constants  $(\mathbf{c}, \mathbf{c}', \dots)$  and first-order operators of the language in §.3.1. We also have a paring, projection and injection operation.  $!x$  denotes the dereference of  $x$ .

The predicate  $\{C\} e \bullet e' \searrow_x \{C'\}$  is called *evaluation formula*, where the name  $x$  binds its free occurrences in  $C'$ . Intuitively,  $\{C\} e \bullet e' \searrow_x \{C'\}$  asserts that an invocation of  $e$  with an argument  $e'$  under the (hypothetical) initial state  $C$  terminates with a final state and a resulting value, named as  $x$ , both described by  $C'$ .  $\bullet$  is non-commutative.  $\text{fv}(e)$  denotes the free variables occurring in  $e$ . We define two kinds of capture-avoiding substitutions  $C[e/x]$  and  $C[e/!x]$ , see [2, § 3.3].

Terms and formulae are typed starting from type-annotated variables. Two occurrences of the same name should have the same type.  $!(x^\rho)$  is typed as  $\alpha$  iff  $\rho = \text{Ref}(\alpha)$ . If  $e_1, e_2$  and  $z$  are typed as  $\alpha \Rightarrow \beta, \alpha$  and  $\beta$ , respectively, then  $\{C\} e_1 \bullet e_2 \searrow_z \{C'\}$  is well-typed. The remaining well-typedness conditions are naturally given [2]. A boolean typed term is also used as a formula. Hereafter we only consider well-typed terms and formulae and often omit type annotations. We shall write  $\Theta \vdash C$  if  $C$  is well-typed with its free names typed following  $\Theta$ , where  $\Theta, \Theta', \dots$  combine two kinds of bases.

**Convention 1**  $C_1 \equiv C_2$  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  $\mathbf{T}$  (definable as  $1 = 1$ ) and falsity  $\mathbf{F}$  (which is  $\neg \mathbf{T}$ ). The standard binding convention is always assumed.  $\text{fv}(C)$  denotes the set of *free variables* in  $C$ .  $\{C\} e_1 \bullet e_2 \searrow_{e'} \{C'\}$  with  $e'$  not a variable, stands for  $\{C\} e_1 \bullet e_2 \searrow_x \{x = e' \wedge C'\}$  with  $x$  fresh; and  $\{C\} e_1 \bullet e_2 \{C'\}$  for  $\{C\} e_1 \bullet e_2 \searrow () \{C'\}$ .  $A, B$  denote formulae which do not contain dereferences except in pre/post conditions of evaluation formulae. Formulae are often called *assertions*.

Some small examples:  $y = 6$  is an assertion which says  $y$  is equal to 6;  $!y = 6$  says the content of a memory cell  $y$  is equal to 6.  $C \stackrel{\text{def}}{=} \forall i, n. \{!w = n\} !x \bullet i \searrow 2 \times i \{!w = n + 1\}$  says  $x$  stores a function which, when invoked, increments  $w$  and returns the double of the argument. This is satisfied when, for example,  $f(w) \stackrel{\text{def}}{=} \lambda z. (w := !w + 1; z \times 2)$  is stored in  $x$ .  $D \stackrel{\text{def}}{=} \{C \wedge !w = 0\} u \bullet 3 \searrow 6 \{C \wedge !w = 1\}$  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  $!w = 1$ . This is satisfied by  $\lambda y. (!x)y$  named  $u$ , with  $x$  storing  $f(w)$  above.

In addition to axioms and rules of (say) number theory, there are axioms proper to data types and evaluation formulae, as detailed in [2].

## 4. Judgement and Proof Rules

**Judgement.** Following Hoare [18], a judgement in the present program logic consists of a pair of formulae and a program augmented with a fresh name called *anchor*, which takes the following shape.

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

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  $\{C\} M^{\Gamma; \Delta; \alpha} :_u \{C'\}$ , we assume  $\Gamma; \Delta \vdash M : \alpha$ .  $u$  is called the *anchor* of the judgement and should not be in  $\text{dom}(\Gamma, \Delta) \cup \text{fv}(C)$ . The formula  $C$  is the *pre-condition*; and  $C'$  is the *post-condition*. We say  $\models \{C\} M^{\Gamma; \Delta; \alpha} :_u \{C'\}$  is *well-typed* if (1)  $\Gamma; \Delta \vdash M : \alpha$ ; and (2)  $\Gamma, \Delta, \Theta \vdash C$  and  $u : \alpha, \Gamma, \Delta, \Theta \vdash C'$  for some  $\Theta$  such that  $\text{dom}(\Theta) \cap (\text{dom}(\Gamma, \Delta) \cup \{u\}) = \emptyset$ . The same condition applies to judgements on provability. Then the *primary names* in this well-formed judgement are  $\text{dom}(\Gamma, \Delta) \cup \{u\}$ . The *auxiliary names* in this judgement are those free names in  $C$  and  $C'$  which are not primary (for example, in “ $\{x = i\} 2 \times x :_u \{u = 2 \times i\}$ ”,  $x$  and  $u$  are primary while  $i$  is auxiliary;  $u$  is in addition its anchor). We often omit the typing of a program from a judgement, writing  $\{C\} M :_u \{C'\}$ .

Intuitively,  $\{C\} M^{\Gamma; \Delta; \alpha} :_u \{C'\}$  says that: *if  $M$  is closed by values satisfying  $C$  (for functional variables), and is evaluated starting from a store satisfying  $C$  (for imperative variables), then it terminates with a value named  $u$  and final state, satisfying  $C'$ .*

**Proof Rules.** The proof rules are given in Figure 1. In each rule, we use the notational conventions from the preceding sections. In addition,  $C^{\bar{x}}$  means  $C$  in which no name from  $\bar{x}$  freely occurs. We assume all occurring judgements are well-typed, and no primary names in the premise(s) occur as auxiliary names in the conclusion.

Below we illustrate key aspects of these rules.

**[Var, Const]** say that, if we wish to assert  $C$  about a datum named  $u$ , we should assume the same property, with the datum substituted for  $u$ .

**[Add]** is the rule for the addition operator, which assumes the left-to-right evaluation order, indicating both the state change induced by evaluation and the resulting values. Similarly for other first-order operators.

**[Abs]** says: if we know, under the assumptions  $A$  (which does not talk about  $x$ ) and starting from  $C$ , evaluation of  $M$  always terminates with a result  $m$  and a state which together satisfy  $C'$ , then we can guarantee  $\lambda x. M$  named  $u$  satisfies the same property under the same assumption  $A$ , now presented

---

**Figure 1** Proof Rules
 

---

$$\begin{array}{c}
 \text{[Var]} \frac{}{\{C[x/u]\} \bar{x} :_u \{C\}} \quad \text{[Const]} \frac{}{\{C[c/u]\} \bar{c} :_u \{C\}} \\
 \\
 \text{[Add]} \frac{\{C\} M_1 :_{m_1} \{C_0\} \quad \{C_0\} M_2 :_{m_2} \{C'[m_1 + m_2/u]\}}{\{C\} M_1 + M_2 :_u \{C'\}} \\
 \\
 \text{[Abs]} \frac{\{C \wedge A^x\} M :_m \{C'\}}{\{A\} \lambda x.M :_u \{\forall x.\{C\} u \bullet x \searrow m \{C'\}\}} \\
 \\
 \text{[App]} \frac{\{C\} M :_m \{C_0\} \quad \{C_0\} N :_n \{C_1 \wedge \{C_1\} m \bullet n \searrow u \{C'\}\}}{\{C\} MN :_u \{C'\}} \\
 \\
 \text{[If]} \frac{\{C\} M :_b \{C_0\} \quad \{C_0[t/b]\} M_1 :_u \{C'\} \quad \{C_0[f/b]\} M_2 :_u \{C'\}}{\{C\} \text{if } M \text{ then } M_1 \text{ else } M_2 :_u \{C'\}} \\
 \\
 \text{[In}_1\text{]} \frac{\{C\} M :_v \{C'[\text{in}_1(v)/u]\}}{\{C\} \text{in}_1(M) :_u \{C'\}} \\
 \\
 \text{[Case]} \frac{\{C^{\bar{x}}\} M :_m \{C_0^{\bar{x}}\} \quad \{C_0[\text{in}_i(x_i)/m]\} M_i :_u \{C'^{\bar{x}}\}}{\{C\} \text{case } M \text{ of } \{\text{in}_i(x_i).M_i\}_{i \in \{1,2\}} :_u \{C'\}} \\
 \\
 \text{[Pair]} \frac{\{C\} M_1 :_{m_1} \{C_0\} \quad \{C_0\} M_2 :_{m_2} \{C'[(m_1, m_2)/u]\}}{\{C\} \langle M_1, M_2 \rangle :_u \{C'\}} \\
 \\
 \text{[Proj}_1\text{]} \frac{\{C\} M :_m \{C'[\pi_1(m)/u]\}}{\{C\} \pi_1(M) :_u \{C'\}} \\
 \\
 \text{[Deref]} \frac{}{\{C[!x/u]\} \bar{!x} :_u \{C\}} \quad \text{[Assign]} \frac{\{C\} M :_m \{C'[m/!x][()]/u\}}{\{C\} x := M :_u \{C'\}} \\
 \\
 \text{[Rec]} \frac{\{A^x \wedge \forall j \leq i. B(j)[x/u]\} \lambda y.M :_u \{B(i)\}}{\{A\} \mu x. \lambda y.M :_u \{\forall i. B(i)\}} \\
 \\
 \text{[Promote]} \frac{\{A\} V :_u \{B\}}{\{A \wedge C\} V :_u \{B \wedge C\}} \\
 \\
 \text{[Consequence]} \frac{C \supset C_0 \quad \{C_0\} M :_u \{C'_0\} \quad C'_0 \supset C'}{\{C\} M :_u \{C'\}}
 \end{array}$$


---

as an evaluation formula replacing  $M$  with a call to  $u$  by  $x$ ,  $u \bullet x$ , with the result  $m$ .

**[App]** says: if we know  $M$  reaches  $C_0$  starting from  $C$ , and  $N$  reaches  $C_1$  starting from  $C_0$ , and, moreover, we know putting them together and applying them reaches  $C'$  starting from  $C_1$ , then  $MN$  reaches  $C'$  starting from  $u$ .

**[If, In<sub>1</sub>, Case, Pair, Proj<sub>1</sub>]** are natural rules for standard data types, similar to **[Add]**.

**[Deref]** is understood as **[Var, Const]**. If we wish to have  $C$  for a program  $!x$  named  $u$ , then we should assume the same thing for the content of  $x$ , substituting  $!x$  for  $u$ .

**[Assign]** uses two substitutions  $C'[m/!x][()]/u$ . The notation  $[m/!x]$  stands for replacing all occurrences of  $!x$  by  $m$ , while  $[()]/u$  is the standard substitution of  $()$  for  $u$ . 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 substitution  $[()]/u$  says, in effect, the assignment command terminates (because  $()$  is the unique value of type Unit).

**[Rec]** is for the total correctness of recursion [19, 22]. It is based on mathematical induction, though by choosing an appropriate domain and a well-ordering, we can extend the rule to well-founded induction.

**[Promote]** extends the stateless pre/post-conditions to general ones by conjunction. The rule is sound because a value does not (immediately) cause state change.

**[Consequence]** The rule follows the standard consequence rule in Hoare logic. Checking the validity of entailment is in general intractable, though in practise one can often appeal to syntactic reasoning. Other structural rules are discussed in [2, §4].

## 5. Observational Completeness

### 5.1 Models and Soundness

We introduce an operationally oriented notion of models.

**Definition 1** (semi-closed programs [31])  $\Gamma; \Delta \vdash M : \alpha$  is *semi-closed* when  $\text{dom}(\Gamma) = \emptyset$ , often written  $\Delta \vdash M : \alpha$ .

Let  $\cong$  be the standard observational congruence for the imperative PCFv [15], based on convergence to semi-closed values. An *abstract value of type*  $\Delta; \alpha$  is a  $\cong$ -congruence class of semi-closed values typed  $\alpha$  under  $\Delta$ . We write  $[V]^{\Delta; \alpha}$  for an abstract value represented by  $\Delta \vdash V : \alpha$ . A model is defined using abstract values.

**Definition 2** (models) A *model of type*  $\Gamma; \Delta$  is a pair  $(\xi, \sigma)$  such that  $\xi$  is a finite map from  $\text{dom}(\Gamma)$  to abstract values such that each  $x \in \text{dom}(\Gamma)$  is mapped to an abstract value typed as  $[V]^{\Delta; \Gamma(x)}$ ; and  $\sigma$  is a finite map from  $\text{dom}(\Delta)$  to abstract values such that each  $x \in \text{dom}(\Delta)$  is mapped to an abstract value typed as  $[V]^{\Delta; \alpha}$  with  $\Delta(x) = \text{Ref}(\alpha)$ . We let  $\mathcal{M}, \dots$  range over models.

We write  $\Gamma; \Delta \vdash \mathcal{M}$  when  $\mathcal{M}$  is a model of type  $\Gamma; \Delta$ . Intuitively,  $\xi$  and  $\sigma$  in  $(\xi, \sigma)$  respectively denote a standard functional environment and a store, taken modulo  $\cong$ .

Assume given a formula  $C$  and a model  $\mathcal{M}$ , both typed under  $\Gamma; \Delta$ . Then each term in  $C$  is inductively interpreted under  $\mathcal{M}$  as an abstract value in the obvious way, except each name of a reference type is interpreted as that name itself (so that, in effect, we treat reference-typed names as constants). As examples, given  $\mathcal{M} = (\xi, \sigma)$ , a functional variable  $x^\alpha$  is interpreted as  $\xi(x)$ ; dereferencing  $!y$  is interpreted as  $\sigma(y)$ ; and a pair  $\langle e, e' \rangle$  is interpreted as a pair of abstract values interpreting  $e$  and  $e'$  (the full definition is found in [2, 22, §4.3]). We write  $\llbracket e \rrbracket \mathcal{M}$  for the interpretation of  $e$  under  $\mathcal{M}$ .

The satisfaction relation is written:  $\mathcal{M}^{\Gamma;\Delta} \models C$ , which is defined following the standard first-order logic with equality [29, §2.8] with the equality predicate interpreted as the identity relation, adding the following clause for evaluation formulae. Let  $\mathcal{M} = (\xi, \sigma_0)$ . Below  $\Downarrow$  is defined from that of concrete programs. We set  $\mathcal{M}^{\Gamma;\Delta} \models \{C\}_{e_1 \bullet e_2 \setminus x} \{C'\}$  if

$$\begin{aligned} \forall \sigma. (\Delta \vdash \sigma \wedge (\xi, \sigma) \models C \\ \supset \exists V, \sigma'. ([e_1] \mathcal{M} [e_2] \mathcal{M}, \sigma) \Downarrow ([V]^{\Delta;\beta}, \sigma') \\ \text{such that } (\xi \cup x : [V]^{\Delta;\beta}, \sigma') \models C') \end{aligned}$$

The left-hand side says: if the interpretation of  $e_1$  is invoked with that of  $e_2$  as an argument, then for any state  $\sigma$  satisfying  $C$ , the invocation starting from  $\sigma$  will converge with a value named  $x$  and a state  $\sigma'$ , together satisfying  $C'$ . Below  $M\xi$  denotes the substitution of values following  $\xi$ , confusing abstract values and concrete values.

**Definition 3** (semantics of judgement)  $\models \{C\} M^{\Gamma;\Delta;\alpha} :_u \{C'\}$  iff, for each  $\xi$  and  $\sigma$ , whenever  $(\xi, \sigma) \models C$ , we have  $(M\xi, \sigma) \Downarrow (V, \sigma')$  such that  $(\xi \cdot u : [V]^{\Delta;\alpha}, \sigma') \models C'$ .

One of the basic results on the proposed logic follows.

**Theorem 1** (soundness of proof rules)  
If  $\vdash \{C\} M^{\Gamma;\Delta;\alpha} :_u \{C'\}$  then  $\models \{C\} M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ .

## 5.2 Observational Completeness

Compositional semantics dictates that programs with the same contextual behaviour are in principle interchangeable without affecting the observable behaviour of whole software, thus offering foundations for modular software engineering. Compositional program logics extend this idea by further allowing programs with the same specifications to be interchangeable without affecting the observable behaviour of the whole, up to a required specification. For this to be materialised, it is essential that valid assertions for programs capture precisely the contextual behaviour of programs [17, 30, 31]. This criterion may be stated with different degrees of exactness:

1. Are two programs contextually equivalent if and only if they satisfy the same set of assertions? That is, are  $M_1 \cong M_2$  if and only if, for each  $A$ ,  $u : M_1 \models A$  implies  $u : M_2 \models A$  and vice versa?
2. For each program, is there an assertion (*characteristic formula*) which fully describes its behaviour? That is, for each  $M$ , can we find  $A$  such that  $u : M \models A$  and  $u : N \models A$  implies  $M \cong N$ ?

For brevity we only consider semi-closed values and confuse abstract and concrete values above. Clearly (2) entails (1). Further, these questions can also be asked at the level of provability. (1) may be regarded as an essential property

of any program logic which aims to capture observable behaviour of programs. The following establishes (1) for our logic. For (2) (including its provability version), see § 7.

For establishing (1), we proceed as follows.

**Step 1:** We introduce a variant of *finite canonical forms* (FCFs) [5, 21, 23] which represent a limited class of behaviours and whose properties are, therefore, more readily extracted.

**Step 2:** We show characteristic formulae of FCFs w.r.t. total correctness are derivable using our proof rules.

**Step 3:** By reducing a differentiating context of two terms 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 validity.

For our present purpose, it suffices to focus on the following class of assertions. Below  $\sqsubseteq$  is the standard contextual ordering.

**Definition 4** An assertion  $C$  is a *total correctness assertion* (TCA) at  $u$  if whenever  $(\xi \cdot u : \kappa, \sigma) \models C$  and  $\kappa \sqsubseteq \kappa'$ , we have  $(\xi \cdot u : \kappa', \sigma) \models C$ .

Intuitively, total correctness is a property which is closed upwards — if a program  $M$  satisfies one and there is a more defined program  $N$  then  $N$  also satisfies it, see [2, §6]. The notion of characteristic formulae needs be refined for total correctness:

**Definition 5** (characteristic formulae) Given a semi-closed  $V, C$ , a TCA at  $u$ , *characterises*  $V$  iff: (1)  $\models \{\top\} V^{\Delta;\alpha} :_u \{C\}$  and (2)  $\models \{\top\} W^{\Delta;\alpha} :_u \{C\}$  implies  $V \sqsubseteq W$ .

We now introduce FCFs. Henceforth we only consider Nat and arrow types for simplicity. This does not influence the arguments. Finite canonical forms (FCFs), ranged over by  $F, F', \dots$ , are a subset of typable terms given by the following grammar (with obvious translations).  $U, U', \dots$  range over FCFs which are values.

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

where in the case construct,  $X$  is a finite non-empty subset of natural numbers (it diverges for others); and  $\omega^\alpha$  stands for a diverging closed term of type  $\alpha$ . We also set  $\Omega^{\alpha \Rightarrow \beta} \stackrel{\text{def}}{=} \lambda x^\alpha. \omega^\beta$ . We omit the obvious induced typing rules. In the functional sublanguage, FCFs represent essentially finite behaviour. Here we use FCFs for their tractability to derive characteristic formulae.

**Proposition 1** For each semi-closed  $\Delta \vdash U : \alpha$ , we have  $\vdash \{\top\} U^{\Delta;\alpha} :_u \{C\}$  such that  $C$  characterises  $U$ .

The proof uses derived proof rules for extracting TCAs for FCFs, for which we inductively prove the property in Definition 5. Note Proposition 1 implies (relative) completeness of  $\vdash$  for FCFs w.r.t. total correctness.

Write  $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$  when  $\models \{C\}M_1^{\Gamma; \Delta; \alpha} :_u \{C'\}$  iff  $\models \{C\}M_2^{\Gamma; \Delta; \alpha} :_u \{C'\}$ . The main result follows.

**Theorem 2** (observational completeness)  
 $\Gamma; \Delta \vdash M_1 \cong M_2 : \alpha$  if and only if  $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$ .

One of the significant consequences of Theorem 2 is that a strongest post condition for total correctness of  $T$  w.r.t. each semi-closed value (if any), not restricted to FCFs, is its characteristic formula.

## 6. Reasoning Examples

**Simple Imperative Higher-Order Functions.** We further illustrate the use of proof rules with programs which correspond to the assertions in § 2 and § 3.2. Let  $Double(u) \stackrel{\text{def}}{=} \forall i. (u \bullet i = i \times 2)$ . Then we infer a function with dereference.

$$\frac{\{Double(!x)\} !x :_m \{Double(m)\}}{\{y = 3\} y :_n \{n = 3\}} \quad (\text{Deref})$$

$$\frac{\{Double(!x) \wedge y = 3\} (!x)y :_u \{Double(!x) \wedge u = 6\}}{\{T\} \lambda y. (!x)y :_u \{\{Double(!x)\} u \bullet 3 \searrow 6 \{Double(!x)\}\}} \quad (\text{App})$$

$$\frac{\{T\} \lambda y. (!x)y :_u \{\{Double(!x)\} u \bullet 3 \searrow 6 \{Double(!x)\}\}}{\{Double(!x)\} (\lambda y. (!x)y) 3 :_u \{u = 6 \wedge Double(!x)\}} \quad (\text{Abs})$$

$$\frac{\{Double(!x)\} (\lambda y. (!x)y) 3 :_u \{u = 6 \wedge Double(!x)\}}{\{Double(!x)\} (\lambda y. (!x)y) 3 :_u \{u = 6 \wedge Double(!x)\}} \quad (\text{App})$$

Next we use the following derived rule: below by definition we assume  $M$  has type  $\text{Unit}$ , and safely omit its anchor by  $C(!)/u \equiv C$ .

$$[Seq] \frac{\{C\}M \{C_0\} \quad \{C_0\}N :_u \{C'\}}{\{C\}M;N :_u \{C'\}}$$

Using  $[Seq]$ , we can plug-in the post and pre-conditions of the conclusions of  $(\lambda y. (!x)y)3$  and  $\{T\}x := \lambda z. (z \times 2) \{Double(!x)\}$  as:

$$\{T\} x := \lambda z. (z \times 2); (\lambda y. (!x)y)3 :_u \{u = 6 \wedge Double(!x)\}$$

By a similar reasoning, we obtain the following which corresponds to  $C$  in § 3.2.

$$\{T\} x := \lambda z. (w := !w + 1; z \times 2) \{ \forall i, n. \{!w = n\} !x \bullet i \searrow 2 \times i \{!w = n + 1\} \}$$

Then similarly, we can derive  $D$  in § 3.2.

$$\{T\} \lambda y. (!x)y :_u \{\{C \wedge !w = 0\} u \bullet 3 \searrow 6 \{C \wedge !w = 1\}\}$$

Combining these by  $[Seq]$  gives us:

$$\{C \wedge !w = 0\} x := \lambda z. (w := !w + 1; z \times 2); (\lambda y. (!x)y)3 :_u \{u = 6 \wedge C \wedge !w = 1\}$$

Finally we treat  $M$  in (2.6), § 2, using  $A(fg)$  given there.

$$\frac{\{A(fg) \wedge Even(x)\} y := x \{A(fg) \wedge Even(!y)\}}{\{A(fg) \wedge Even(!y)\} g(f) \{A(fg) \wedge Odd(!y)\}} \quad (\text{Assign})$$

$$\frac{\{A(fg) \wedge Even(!y)\} g(f) \{A(fg) \wedge Odd(!y)\}}{\{A(fg) \wedge Odd(!y)\} g(f) \{A(fg) \wedge Even(!y)\}} \quad (\text{Var, App})$$

$$\frac{\{A(fg) \wedge Odd(!y)\} g(f) \{A(fg) \wedge Even(!y)\}}{\{A(fg) \wedge Even(!y)\} !y + 1 :_z \{Odd(z) \wedge Even(!y)\}} \quad (\text{Op})$$

$$\frac{\{A(fg) \wedge Even(!y)\} !y + 1 :_z \{Odd(z) \wedge Even(!y)\}}{\{A(fg)\} M :_u \{Even.then.Odd(u, x)\}} \quad (\text{Seq, Abs})$$

**Three Programming Examples Revisited.** We revisit the examples from the introduction, offering their formal specifications and an inference for one of them.

First, the specification for `closureFact` can be made precise with the following judgement.

$$\{T\} \text{closureFact} :_u \{\forall i^{\text{Nat}}. \{T\} u \bullet i \{ \{T\} !y \bullet () \searrow z \{z = i!\} \}\}$$

Next the specification of `circFact` can be written down as follows, under  $x : \text{Ref}(\text{Nat} \Rightarrow \text{Nat})$ :

$$\{T\} \text{circFact} \{ \exists g. (\forall i. \{!x = g\} (!x) \bullet i \searrow !i \{!x = g\} \wedge !x = g) \}$$

The postcondition says: after executing `circFact`,  $x$  stores a procedure which would calculate a factorial if  $x$  indeed stores that behaviour itself, and that  $x$  does store that behaviour. For scheduler we have (with  $C(i)$  representing a sequence of states; we assume list operations  $::$  and  $\text{Nil}$  in the assertion language):

$$\text{Sched}(u) \stackrel{\text{def}}{=} \{C(0)\} u \bullet \text{Nil} \{C(0)\} \wedge \forall g^{\alpha \Rightarrow \text{Unit}}, a^{\alpha}, y^{\text{List}((\alpha \Rightarrow \text{Unit}) \times \alpha)} \{ \{C(i+1)\} g \bullet a \{C(i)\} \wedge \{C(i)\} u \bullet y \{C(0)\} \supset \{C(i+1)\} u \bullet ((g, a) :: y) \{C(0)\} \}$$

The assertion says: given for example a list  $l \stackrel{\text{def}}{=} [(f, a), (g, b)]$ , if we have  $\{C(0)\} fa \{C(1)\}$  and  $\{C(1)\} gb \{C(2)\}$ , then  $\{C(0)\} \text{scheduler } l \{C(2)\}$  holds.

For derivations we only consider `circFact`. For detailed derivations of all three examples see [2, §9]. Let

$$\begin{aligned} A(u, g, j) &\stackrel{\text{def}}{=} \{!x = g\} u \bullet j \searrow j! \{!x = g\}. \\ C(u, g, i) &\stackrel{\text{def}}{=} \forall j \leq i. A(u, g, j) \wedge !x = g. \\ M &\stackrel{\text{def}}{=} \text{if } y = 0 \text{ then } 1 \text{ else } y \times ((!x)(y-1)) \end{aligned}$$

The main derivation follows.

$$\frac{\{C(!x, g, y) \wedge y = 0\} 1 :_m \{m = y! \wedge C(!x, g, y)\}}{\{C(!x, g, y) \wedge y \geq 0\} y \times ((!x)(y-1)) :_m \{m = y! \wedge C(!x, g, y)\}} \quad \frac{\{C(!x, g, y) \wedge y \geq 0\} y \times ((!x)(y-1)) :_m \{m = y! \wedge C(!x, g, y)\}}{\{C(!x, g, y)\} M :_m \{m = y! \wedge C(!x, g, y)\}} \quad \frac{\{C(!x, g, y)\} M :_m \{m = y! \wedge C(!x, g, y)\}}{\{T\} \lambda y. M :_m \{ \forall yg. \{C(!x, g, y)\} u \bullet y \searrow y! \{C(!x, g, y)\} \}} \quad \frac{\{T\} \lambda y. M :_m \{ \forall yg. \{C(!x, g, y)\} u \bullet y \searrow y! \{C(!x, g, y)\} \}}{\{T\} \text{circFact} \{ \forall yg. \{C(!x, g, y)\} !x \bullet y \searrow y! \{C(!x, g, y)\} \}} \quad \frac{\{T\} \text{circFact} \{ \forall yg. \{C(!x, g, y)\} !x \bullet y \searrow y! \{C(!x, g, y)\} \}}{\{T\} \text{circFact} \{ \exists g. (\forall i. A(!x, g, i) \wedge !x = g) \}}$$

The second line infers that the resulting value is  $y!$  using the induction hypothesis  $C(!x, g, y)$  and multiplication. The last step is by the following inference:

$$\begin{aligned} &\forall yg. \{C(!x, y, g)\} !x \bullet y \searrow y! \{C(!x, g, y)\} \\ &\Rightarrow \forall yg. (\forall j \leq y. A(g, g, j) \supset A(!x, g, y)) \\ &\Rightarrow \exists g. (\forall y. (\forall j \leq y. A(g, g, j) \supset A(g, g, y)) \wedge !x = g) \\ &\Rightarrow \exists g. (\forall y. A(!x, g, y) \wedge !x = g). \end{aligned}$$

where the first deduction uses the following axiom:

$$\{A \wedge C\} e \bullet e' \searrow x \{A \wedge C'\} \equiv (A \supset \{C\} e \bullet e' \searrow x \{C'\}),$$

the second one  $\forall x. A \supset \exists x. (A \wedge x = y)$  (via  $\forall x. A \supset A[y/x]$ ), and the last inference the standard mathematical induction.

## 7. Further Topics and Related Works

**Inferential Completeness** As observed in §5.2, our proof system is (relatively) complete for semi-closed FCFs w.r.t. total correctness. Does this extend to the whole language? We believe so in the following sense.

**Conjecture.** (1) For each semi-closed  $V$ ,  $\{T\}V :_u \{C\}$  s.t.  $C$  characterises  $V$  in the sense of Def. 5. (2) For each TCA  $C'$ ,  $\models \{T\}V :_u \{C'\}$  implies  $\vdash \{T\}V :_u \{C'\}$ .

The statement says that the assertion language can pinpoint, and the proof rules can relatively justify, any upwards-closed set which has a semi-closed value as its least element. We also conjecture that the corresponding statement holds for the proof system for partial correctness (this proof system, which is essentially identical with the present system except for a suitable replacement of the recursion rule, can embed and justify known proof rules for partial correctness for imperative languages, such as the while rule and procedure rules in Hoare logic). For both total and partial correctness, our coming paper will discuss completeness results in detail.

**Aliasing and Local State** In § 3, it is observed that allowing reference types to be carried by other types (including arrow and reference types) leads to a distinct class of behaviour. Indeed, this generalisation induces a strong notion of aliasing, in the sense that a reference name returned from a procedural call (as well as from e.g. reading references) can textually coalesce reference names in a program text. This significantly increases complexity in behaviour, hence in its logical treatment. A clean and tractable logical treatment of this phenomenon is possible on the basis of the logic studied in the present work, incrementing its assertion language with mutually dual modal operators which function as quantifiers over content of references, rather than over references themselves. For details see [7]. On the basis of this stratification, local state is also incorporated by a simple logical enrichment, reminiscent of the  $v$ -operator in  $\pi$ -calculus. The full exploration of local state will be reported elsewhere. For simplicity, we have omitted polymorphism and recursive types in the present paper. Their integration is entirely straightforward following [22].

**Related Work** In the following comparative discussions, we focus on directly related work, leaving more extensive comparisons to [2, 20, 22]. Compositional program logics for imperative languages have been studied extensively since Hoare’s seminal work. In late 1970s and early 1980s, there were several attempts to extend Hoare logic to higher-order languages, mostly focussing on Algol and its derivatives. Clarke [9] shows that a sound and (relatively) com-

plete Hoare logic *cannot* exist for programming languages with a certain set of features, including arbitrary higher-order procedures. Clarke’s argument relies on a given logical language being first-order and allowing models to have a finite universe (which makes validity in assertions recursive). As Halpern pointed out [16], a sound and complete logic may exist for higher-order programming languages if we consider other classes of models, as we do here. The relationship with finiteness in models and descriptive power of assertions is further discussed in [2, §8].

Following Clarke’s result, Olderog [34, 35], German et al. [13], Damm and Josco [11], Halpern [16] and Trakhtenbrot et al. [42] study Algol-like languages with procedures as parameters, obtaining various inferential relative completeness results. None of these works (aim to) allow direct description of behaviour of higher-order expressions including those stored in imperative variables, which we do with evaluation formulae. This restriction partly reflects the nature of their target languages, which strictly separate commands from (first-order and higher-order) expressions.

Specification logic by Reynolds [39] is a program logic for Idealised Algol which, as in LCF, allows higher-order programs to appear textually in assertions. For reasoning about side effects, assertions also include Hoare-triple-like formulae for command types, though their pre/post conditions only assert on first-order state, unlike our evaluation formulae. Specification logic does not allow assertions on, and compositional reasoning for, higher-order expressions.

Reynolds, O’Hearn and others [40] 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.

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 the higher-order setting because of complex interplay between higher-order procedures. Experiment of the possible extensions in the context of an integrated verification framework such as JML [3] would be an interesting subject for further study.

Names have been used in Hoare logic since an early work by Kowaltowski [24], and are found in the work by von Oheimb [43], Leavens and Baker [27], Abadi and Leino [4] and Bierman and Parkinson [8], 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.

None of the related works discussed above reports observational completeness in the sense of Theorem 2. The precise correspondence between contextual behaviours and

logical descriptions becomes essential when we take assertions on higher-order behaviours in earnest, including in practical applications.

The origin of the assertions and judgements introduced in the present work is the logic for typed  $\pi$ -calculi [19, 20] where linear types lead to a compositional process logic. The known precise embeddings of high-level languages into these typed  $\pi$ -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. [19, 20] discuss process logics and their relationship to the program logics in detail.

## References

- [1] C- home page. <http://www.cminusminus.org>.
- [2] A full version of this paper. Available at: <http://www.dcs.qmul.ac.uk/~kohei/logics>.
- [3] The Java Modeling Language (JML) home page. <http://www.jmlspecs.org/>.
- [4] M. Abadi and R. Leino. A logic for object-oriented programs. In *Verification: Theory and Practice*, pages 11–41. Springer-Verlag, 2004.
- [5] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. & Comp.*, 163:409–470, 2000.
- [6] K. R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.
- [7] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. Available at: [www.dcs.qmul.ac.uk/~kohei/logics](http://www.dcs.qmul.ac.uk/~kohei/logics), 2005.
- [8] G. M. Bierman and M. J. Parkinson. Separation logic and abstractions. In *POPL’05*, 2005.
- [9] E. Clarke. Programming language constructs for which it is impossible to obtain good hoare axiom systems. *J. ACM*, 26(1):129–147, 1979.
- [10] P. Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.
- [11] W. Damm and B. Josko. A sound and relatively\* complete hoare-logic for a language with higher type procedures. *Acta Inf.*, 1983.
- [12] R. W. Floyd. Assigning meaning to programs. In *Symp. in Applied Mathematics*, volume 19, 1967.
- [13] S. M. German, E. M. Clarke, and J. Y. Halpern. Reasoning about procedures as parameters. In *IBM Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 206–220, 1981.
- [14] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI’02*. ACM, 2002.
- [15] C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
- [16] J. Y. Halpern. A good hoare axiom system for an algol-like language. In *11<sup>th</sup> POPL*, pages 262–271. ACM Press, 1984.
- [17] M. Hennessy and R. Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
- [18] T. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
- [19] K. Honda. From process logic to program logic. In *ICFP’04*, pages 163–174. ACM Press, 2004.
- [20] K. Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: [www.dcs.qmul.ac.uk/~kohei/logics](http://www.dcs.qmul.ac.uk/~kohei/logics), March 2004. Typescript, 234 pages.
- [21] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.
- [22] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP’04*, pages 191–202, 2004.
- [23] J. M. E. Hyland and C. H. L. Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
- [24] T. Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.
- [25] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
- [26] P. Landin. A correspondence between algol 60 and church’s lambda-notation. *Comm. ACM*, 8:2, 1965.
- [27] G. Leavens and A. L. Baker. Enhancing the pre- and post-condition technique for more expressive specifications. In *FM’99: World Congress on Formal Methods*. Springer, 1999.
- [28] I. Mason and C. Talcott. Equivalence in functional languages with effects. *JFP*, 1(3):287–327, 1991.
- [29] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
- [30] A. R. Meyer. Floyd-Hoare logic defines semantics (preface version). In *LICS’86*, 1986.
- [31] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL’88*, 1988.
- [32] R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [33] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
- [34] E.-R. Olderog. Sound and complete hoare-like calculi based on copy rules. *Acta Inf.*, 16:161–197, 1981.
- [35] E.-R. Olderog. A characterization of hoare’s logic for programs with pascal-like procedures. In *15<sup>th</sup> Theory of Computing*, pages 320–329, 1983.
- [36] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [37] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS’98*, CUP, pages 227–273, 1998.
- [38] G. D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI, Aarhus University, 1981.
- [39] J. C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
- [40] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS’02*, 2002.
- [41] Z. Shao. An overview of the FLINT/ML compiler. In *1997 ACM SIGPLAN Workshop on Types in Compilation (TIC’97)*, Amsterdam, The Netherlands, June 1997.
- [42] B. A. Trakhtenbrot, J. Y. Halpern, and A. R. Meyer. From denotational to operational and axiomatic semantics for algol-like languages. In *CMU Workshop on Logic of Programs*, pages 474–500, 1984.
- [43] D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001.