

An Observationally Complete Program Logic for Imperative Higher-Order Functions

Kohei Honda¹, Nobuko Yoshida², and Martin Berger¹

¹ Department of Computer Science, Queen Mary, University of London

² Department of Computing, Imperial College London

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 proof rules of the logic exactly follow the syntax of the language and can cleanly embed, justify and extend the standard proof rules for total correctness of Hoare logic. 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.

Table of Contents

1	Introduction	1
2	Preview	6
3	Logic for Imperative Call-by-Value PCF	11
3.1	Imperative PCF	11
3.2	Terms and Formulae	13
3.3	Substitutions and Avoidance of Name Capture	16
3.4	Judgement	19
3.5	Axioms for Data Types and Evaluation Formulae	19
4	Proof Rules	23
4.1	Proof Rules (1): Composition Rules	23
4.2	Proof Rules (2): Structural Rules	27
5	Models and Soundness	30
5.1	Observational Equality	30
5.2	Models	31
6	Observational Completeness	34
6.1	Observability and Program Logics	34
6.2	Assertions for Total Correctness and Characteristic Formulae	34
6.3	Finite Canonical Forms	35
6.4	Characteristic Formulae for FCFs (1): functional sublanguage	36
6.5	Characteristic Formulae for FCFs (2): the imperative extension	39
6.6	Observational Completeness	41
7	Reasoning Examples	43

7.1	Deriving Hoare Logic for Total Correctness	43
7.2	Reasoning for Imperative Higher-Order Functions	46
7.3	Three Programming Examples Revisited	47
8	Discussion	52
8.1	Further Topics	52
8.2	Related Work	52
A	Typed Congruence and Soundness	64
A.1	Typed Congruence	64
A.2	Proof of Proposition 5.13	65
A.3	Proof of Theorem 5.14	67
B	Observational Completeness: Detailed Proofs	71
B.1	Supplement to Proof of Lemma 6.5	71
B.2	Proofs for Proposition 6.8	73
C	Derivations of Hoare Rules	78

1 Introduction

The purpose of the present paper is to introduce a basic compositional program logic for imperative higher-order functions and study its semantic foundations. 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 for decades. They are a cornerstone of richly typed functional programming languages such as ML [54] and Haskell [2] and are central to the semantic analysis of procedural, object-oriented and even low-level languages [1, 21, 44, 55, 61, 69]. 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 for their dynamics [43, 54, 64], a rich class of type disciplines [54, 61] and powerful operational reasoning techniques [48, 62].

Given these achievements, a natural question is if we can carry out a similar development in the context of logical methods for reasoning about programs, in particular those in the tradition of Hoare logic [16, 28, 57]. 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 [8, 13, 20, 50, 51].

The present paper introduces a simple compositional program logic for an imperative extension of call-by-value PCF, built on Hoare logic [28] and our preceding work on logics for pure higher-order functions [31, 35]. The assertions in the logic concisely describe the 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 the full type hierarchy which accommodates general stored procedures and data types has been developed, detailed comparisons with preceding work are found in Section 8. The logical articulation of higher-order behaviour is rigorously stratified, starting from pure functions [31, 35] 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 which a program satisfies precisely characterise its observational behaviour up to the standard contextual congruence [26, 50].

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. As an example, consider $\text{twice} \stackrel{\text{def}}{=} \lambda f^{\alpha \Rightarrow \alpha}. \lambda x^{\alpha}. (f(fx))$, with α being a higher-order type. The behaviour of this program depends on the potentially complex interplay between f and x , all the more

so if they have side effects. Thus 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 programming examples, a clean and rigorous behavioural description of which by a logical means is set to be one of the challenges in our present inquiry (we use notations from standard textbooks [22, 61]).

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

Above $()$ is the unique constant of type Unit and $\lambda().N$ denotes $\lambda z^{\text{Unit}}.N$ with z fresh. When invoked as e.g. $\text{closureFact } 3$, 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 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, but terser, factorial program, using Landin's idea [43] 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 . But just saying so 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 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 the following program.

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

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

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

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 $\langle f_1, x_1 \rangle, \langle f_2, x_2 \rangle, \dots, \langle f_{n-1}, x_{n-1} \rangle$, ($n \geq 1$), if applying f_1 to x_1 changes the state σ_1 to the state σ_2 , applying f_2 to x_2 changes the state σ_2 to the state σ_3 , and so on, and finally applying f_{n-1} to x_{n-1} changes σ_{n-1} to σ_n , then feeding the scheduler with this job list starting from σ_1 will eventually reach the state σ_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.

Observational Completeness

As has been noted already, a basic feature of the proposed logic is that valid judgements on programs talk about no more and no less than the latter's observable behaviour. This is an essential feature for any program logic which aims to capture observable behaviours of programs since if not the status of the descriptive language for assertions itself becomes unclear. In the context of compositional program logics, such a correspondence has a further significance, in that a given logical principle extends and refines, but does not violate, the underlying compositional semantics of observable behaviours of programs. We may also recall the correspondence between program/process behaviour and its logical description has been one of the main points of logics of processes since its inception [26] and is also stressed in the context of Hoare Logics [20, 29].

As initiated in [26], a precise formalisation of this idea can be given by demanding that two programs are observationally equivalent if and only if the collections of valid assertions for two programs coincide. In other words, if two programs have distinct contextual behaviours, then that difference, however small, can be depicted by the assertion language, but no more than that. In the present paper we shall present a transparent proof of this result based on syntax-directed arguments: the proof and associated construction shed further light on the semantic foundations of the proposed logic.

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 [31, 32], 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 restricted syntactic structure called *finite canonical forms* originally introduced in the study of game-based semantics [6, 34, 37]. Basic observations on relative completeness of the proof system are also presented at the end.
3. Exploration of the proposed assertional method and its proof rules through reasoning examples, including a sound embedding and extension of Hoare’s proof rules for total correctness, as well as an illustration of verifications for three programming examples presented above.

Status and Positioning of Presented Program Logic

Standard constructs for high-level programming languages range from arithmetic functions to first-order imperative constructs to higher-order procedures to primitives for concurrency and synchronisation. The presented program logic belongs to a family of program logics which capture the computational behaviours associated with those constructs in a stratified fashion through a logical means. Part of this family of logics have been presented in our foregoing studies [9, 31, 32, 36], arguing for a stratification of sequential programs in the following hierarchy.

- (1) Strongly normalising (pure) higher-order functional behaviour.
- (2) Possibly diverging (pure) higher-order functional behaviour.
- (3) Imperative higher-order functional behaviour without aliasing.
- (4) Imperative higher-order functional behaviour with potential aliasing.
- (5) Imperative higher-order functional behaviour with potential aliasing and local state.

A notable feature of this stratification is a logic in a higher-level stratification can faithfully embed a logic at a lower-level. Each logic at a higher-level hierarchy adds (essentially) a single logical primitive to capture a more complex behaviour. This addition of a logical primitive radically changes the nature of the logic, especially in the construction of models and, therefore, calculation of entailment, a focal point of any program logic.

The present paper treats the class of behaviours (3), after our initial report of its main results in [36] (for reference, the logic for (2) is discussed in [31, 33], (4) in [9] and (5) in a coming exposition). In spite of more complex logics subsuming simpler logics, it is best to consider each hierarchy in general and (3) in particular independently, for the following reasons.

- Generally speaking, reasoning for simpler behaviour is best done in a simpler logic: for example, if we know that the absence of aliasing is guaranteed for stated variables, we had better use the logic which does not assume aliasing almost invariably.

Or even if a program ultimately runs in a multi-threaded environment, if we know that variables in a part of the program are free from interference, we can reason about that part sequentially. Similarly, if we know programs and libraries only assume purely functional behaviour, we can use that fact to achieve considerably simpler logical calculation. Each class of behaviours leads to a distinct class of concerns in logical calculation and reasoning, leading to a distinct logical universe, hence had better be presented in its distilled form.

- More specifically, the logic in the present paper treats a basic class of imperative behaviour, extending standard first-order imperative programs to higher-order procedures. Theoretically, this means that the present logic treats a direct higher-order extension of the standard Hoare logic, thus offering a clear picture of how we may treat higher-order procedures as a precise extension of the standard Hoare logic (as we shall see later, the semantics and proof rules of the original Hoare logic is faithfully embeddable in those of Hoare logic). Practically, this means the present logic allows the reasoning on first-order imperative programs as in Hoare logic without any change, while seamlessly extending the scope of reasoning to arbitrary higher-order procedures when needed. This opens the possibility to use the accumulated results in Hoare logic to the present setting in a fairly straightforward fashion.

For these reasons, devoting each class of logic to an independent technical exposition would make a good sense. In the present work, we focus on those distinct aspects of the logical theory and key reasoning examples pertaining to this specific class of behaviours, so that it can be read independently from the full versions of [9, 32]. Technically the paper for the first time offers a full proof of observational completeness (the method is applicable to more complex logics).

Outline

Section 2 illustrates central ideas of the logic informally. Section 3 introduces the target language and the logic. Section 4 illustrates compositional proof rules for the logic. Section 5 establishes soundness of proof rules. Section 6 proves the sound and complete logical characterisation of the contextual congruence. Section 7 presents reasoning examples. Section 8 discusses related work, extensions and further topics.

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 [31, 35].

Pure Higher-Order Functions. In the present approach to program logics, behaviour is located at, and reasoned via, names. Consider a simple program which computes a doubling function, $N \stackrel{\text{def}}{=} \lambda x.x + x$, with type $\text{Nat} \Rightarrow \text{Nat}$ where Nat is the type for natural numbers. The type of N represents an abstract specification of this program, saying that if we apply a value of Nat , then it returns (if ever) a value of type Nat ; but if we apply a value of other types, it guarantees nothing. Thus a function can guarantee a type of the resulting value only relying on types of its arguments.

We extend the same compositional reasoning to more general behavioural properties, using Hoare-style assertions [16, 28]. Take the same N above; we observe that, if we apply N to 5, then it returns 10; more generally if we apply N to any natural number, it always returns even. And in more detail, it returns the double of an arbitrary argument whenever the latter is well-typed, that is as far as it is a natural number. Thus this function can guarantee a certain behaviour by relying on the property of the argument. We wish to represent these behavioural properties using logical formulae. To do this, we do not mention N itself in a formula, but rather describe its properties by *naming* it as, say, f (any fresh name will do). Thus we can write

$$f \bullet 5 = 10 \tag{2.1}$$

as a property of N (named as f). Similarly we can write

$$\forall x^{\text{Nat}}. \text{Even}(f \bullet x) \tag{2.2}$$

where $\text{Even}(n)$ is the predicate saying n is even (for example we can set $\text{Even}(x) \stackrel{\text{def}}{=} \exists n.(x = 2 \times n)$). The operator \bullet , left-associative and non-commutative, is best understood as the application in applicative structures. We can further refine the formula to say f computes the doubling function. Formulae may be combined using arbitrary standard logical connectives and quantifiers, just as in Hoare Logic.

Using these formulae (which we let range over C, C', \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 function combined with the environment can guarantee C' .

The name f is called *anchor*, and can be any fresh name not occurring in M . An anchor is used for representing the point of operation, hence of specification, of M . This idea – naming functions and programs – naturally comes from encodings of functions into the π -calculus [52, 53] where function M is mapped into an agent $[[M]]_f$ equipped with unique name, f to be referred and shared by other agents. Reflecting its process-theoretic origin, the specification method regards programs as interactive entities [52]

whose specifications are given via interrogating its behaviour one by one. As an example, we can write down the specification for N as:

$$\{\mathsf{T}\} N :_f \{\forall x. \text{Even}(f \bullet x)\} \quad (2.3)$$

which says that the program N named as f , under the trivial assumption T on its free variables, satisfies $\forall x. \text{Even}(f \bullet x)$ (the triviality of the assumption is because no free variables occur in M : the truth T in general indicates the weakest specification allowing any well-typed behaviour).

Let us present further examples of assertions, starting from a simple assertion for the identity function.

$$\{\mathsf{T}\} \lambda x^{\text{Nat}}. x :_u \{\forall y^{\text{Nat}}. u \bullet y = y\}. \quad (2.4)$$

The judgement says that the given program, when applied to any argument of type Nat , will return that same argument as a result.

When this function is applied to the argument, $u \bullet y$ is peeled-off and replaced by a new anchor name m .

$$\{\mathsf{T}\} (\lambda x^{\text{Nat}}. x) 2 :_m \{m = 2\} \quad (2.5)$$

Let us take a brief look at how we can derive (2.5) compositionally from assertions for component programs. We first need the predicate for the argument.

$$\{\mathsf{T}\} 2 :_z \{z = 2\} \quad (2.6)$$

which simply says that “2” as a program satisfies, when named as z , the predicate $z = 2$. We then infer:

$$\frac{(2.4) \quad (2.6) \quad (\forall y. u \bullet y = y) \wedge z = 2 \supset u \bullet z = 2}{\{\mathsf{T}\} (\lambda x^{\text{Nat}}. x) 2 :_m \{m = 2\}}$$

In the above derivation (which follows the proof rule for application, formally discussed later), the predicate in the conclusion, “ $m = 2$ ”, is obtained by substituting “ m ” for “ $u \bullet z$ ” in “ $u \bullet z = 2$ ”, the conclusion of the entailment in the antecedent. Note this entailment is a simple instance of the standard axiom $(A(x, x) \wedge x = y) \supset A(x, y)$ from predicate logic with equality (cf. [49, §2.8]). While validity of such entailment is in general incalculable, it *is* calculable using simple axioms in this case as well as in many practical examples. The nature of the above derivation may become clearer by contrasting it with the corresponding typing derivation.

$$\frac{\vdash \lambda x^{\text{Nat}}. x : \text{Nat} \Rightarrow \text{Nat} \quad \vdash 2 : \text{Nat}}{\vdash (\lambda x^{\text{Nat}}. x) 2 : \text{Nat}}$$

Here, instead of calculating validity of entailment, we simply match the first Nat of the function’s type $\text{Nat} \Rightarrow \text{Nat}$ with the argument’s type Nat , obtaining Nat . In correspondence with simpler specification, we have only to use simpler and essentially mechanical inferences.

Next we consider an example which uses a non-trivial assumption on a higher-order variable.

$$\{\forall x. \text{Even}(f \bullet x)\} f 3 + 1 :_u \{\text{Odd}(u)\} \quad (2.7)$$

where $Odd(n)$ says n is odd. Using the same environment, we can further derive:

$$\{\forall x.Even(f \bullet x)\} f3 + f(f5) + 1 :_u \{Odd(u)\} \quad (2.8)$$

By using name f , the term which contains multiple f can share a single specification in the environment. The derivation of (2.7) and (2.8) starts from the following instance of the axiom for a variable.

$$\{\forall x.Even(f \bullet x)\} f :_m \{\forall x.Even(m \bullet x)\} \quad (2.9)$$

The specification says that if we assume $\forall x.Even(f \bullet x)$ for the function which a variable f stands for, then f as a program named m satisfies precisely the same property (substituting m for f). Starting from (2.9), we can easily reach (2.7) and (2.8), using the rule for application as well as a similar rule for first-order operations.

Let $L \stackrel{\text{def}}{=} f3 + f(f5) + 1$ and recall $N \stackrel{\text{def}}{=} \lambda x.x + x$ before. We now consider the following specification.

$$\{\top\} \text{let } f = N \text{ in } L :_u \{Odd(u)\} \quad (2.10)$$

The “let” construct has the standard decomposition into abstraction and application, but has a special status in the present context in that its derivation clearly describes how we can “plug” a specification of a part into a specification of a whole (just as in the cut rule in the sequent calculus). The derivation of (2.10) follows.

$$\frac{(2.3), \quad (2.8), \quad \forall x.Even(f \bullet x) \supset \forall x.Even(f \bullet x)}{\{\top\} \text{let } f = N \text{ in } L :_u \{Odd(u)\}} \quad (2.11)$$

Note the third condition in the antecedent is a trivially valid tautology. In this tautology, the first $\forall x.Even(f \bullet x)$ is the conclusion of (2.3) while the second one is the assumption of (2.8): thus the property which is guaranteed by N is simply plugged into the assumption for L . This derivation may be understood as being analogous to a composition rule of Hoare Logic, where we infer $\{C\}P_1; P_2\{C'\}$ from $\{C\}P_1\{C_1\}$ and $\{C_2\}P_2\{C'\}$ such that $C_1 \supset C_2$. (2.11) may also be contrasted with the type derivation for the same program.

$$\frac{\vdash N : \text{Nat} \Rightarrow \text{Nat} \quad f : \text{Nat} \Rightarrow \text{Nat} \vdash L : \text{Nat}}{\vdash \text{let } f = N \text{ in } L : \text{Nat}} \quad (2.12)$$

which has a shape isomorphic to (2.11).

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 the dereferencing of x (this follows ML [54]: the use of dereferencing notation in Hoare logic is found in, for example, [71]). The resulting behaviour is located at the anchor 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, we can derive:

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

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 us consider $W \stackrel{\text{def}}{=} \lambda x. (w := (1+!w) ; x + x)$, which modifies $\lambda x. x + x$ in the previous paragraph. Recalling L from (2.8), we define $M \stackrel{\text{def}}{=} \text{let } f = W \text{ in } L$. Now this function does not only satisfy $\text{Odd}(u)$, but also *changes a memory cell* w when invoked. Hence we would expect the following:

$$\{!w = 0\} \text{let } f = W \text{ in } L :_u \{ \text{Odd}(u) \wedge !w = 3 \} \quad (2.13)$$

How can we specify the behaviour of W to reach (2.13)? 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:

$$\{C\} f \bullet x = u \{C'\}$$

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' . “ \bullet ” 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.

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

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

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

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 3, not only is the result u replaced by a new anchor v , but also we *split* the assumption ($\text{EvenS}(w, f)$) in two pieces, its pre-condition placed in the pre-condition of the resulting judgement (together with the assumption itself) and its post-condition placed in the post-condition of the judgement. Repeating this, we can derive (2.13) 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 in an arbitrary way 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 derive:

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

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

A merit of the present approach in comparison with conventional Hoare logic and its refinements is that ours can directly assert formulae about the combined behaviour of (possibly higher-order) procedures. Consider the following program:

$$M \stackrel{\text{def}}{=} \lambda x^{\text{Nat}}. (y := x ; g(f) ; g(f) ; !y + 1) \quad (2.14)$$

where f is of type $\text{Unit} \Rightarrow \text{Unit}$ and g of $(\text{Unit} \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$. 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 present logic can formally describe this property as follows (omitting return values).

$$A(fg) \stackrel{\text{def}}{=} \{Odd(!y)\} g \bullet f \{Even(!y)\} \wedge \{Even(!y)\} g \bullet f \{Odd(!y)\}$$

Then a property of M under these assumptions may be asserted as:

$$\{A(fg)\} M :_u \{ \forall x^{\text{Nat}}. \{Even(x)\} u \bullet x = z \{Odd(z) \wedge Even(!y)\} \} \quad (2.15)$$

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 written $Even_then_Odd(u, y)$. From the same pre-condition, we can also infer the dual property $Odd_then_Even(u, y) \stackrel{\text{def}}{=} \forall x^{\text{Nat}}. \{Odd(x)\} u \bullet x = z \{Even(z) \wedge Odd(!y)\}$ or even a conjunction of the two, $Even_then_Odd(u, y) \wedge Odd_then_Even(u, y)$, as its post-condition. This specification relies on the property of the combined behaviour of f and g and demonstrates the 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 of assertions will be treated in Sections 3, 4 and 7, after formally introducing the logic and proof rules in the following sections.

3 Logic for Imperative Call-by-Value PCF

3.1 Imperative PCF

This subsection briefly reviews the programming language we shall use in the present study, call-by-value PCF with unit, sums and products, augmented with imperative variables (henceforth also called *references*). The grammar of programs is standard [61] and given below. We assume an infinite set of *variables* (x, y, z, \dots , also called *names*).

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

The grammar uses types (α, β, \dots), which are given later. Binding is standard and $\text{fv}(M)$ denotes the set of free variables. Types annotating bound variables are often omitted. Constants (c, c', \dots) include the unit $()$, natural numbers \mathbf{n} and booleans \mathbf{b} (either truth \mathbf{t} or false \mathbf{f}). $\text{op}(\vec{M})$ (where \vec{M} is a vector of programs) is a standard n -ary arithmetic or boolean operation, such as $+$, $-$, \times , $=$ (equality of two numbers), \neg (negation), \wedge and \vee . We only treat the dereference of a variable, $!x$. This suffices by the underlying type structure, similarly for assignment $x := N$.

The dynamics of the language is given as the standard call-by-value reduction using a store [22, 61]. A *store* (σ, σ', \dots) is a finite map from imperative variables 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 [22, 61].

$$((\lambda x. M)V, \sigma) \rightarrow (M[V/x], \sigma) \quad (3.1)$$

$$(\pi_1(\langle V_1, V_2 \rangle), \sigma) \rightarrow (V_1, \sigma) \quad (3.2)$$

$$(\text{case in}_1(W) \text{ of } \{\text{in}_i(x_i). M_i\}_{i \in \{1,2\}}, \sigma) \rightarrow (M_1[W/x_1], \sigma) \quad (3.3)$$

$$(\text{if } \mathbf{t} \text{ then } M_1 \text{ else } M_2, \sigma) \rightarrow (M_1, \sigma) \quad (3.4)$$

$$((\mu f. \lambda g. N)W, \sigma) \rightarrow (N[W/g][\mu f. \lambda g. N/f], \sigma) \quad (3.5)$$

$$(!x, \sigma) \rightarrow (\sigma(x), \sigma) \quad (3.6)$$

$$(x := V, \sigma) \rightarrow ((), \sigma[x \mapsto V]) \quad (3.7)$$

(3.1–3.5) are from call-by-value PCF and do not involve the store (we omit obvious symmetric rules and the rules for first-order operators). (3.6) and (3.7) are for imperative constructs, assuming $x \in \text{dom}(\sigma)$ in both. In (3.7), $\sigma[x \mapsto V]$ denotes the store which maps x to V and otherwise agrees with σ . Finally we have the contextual rule:

$$(\mathcal{E}[M], \sigma) \rightarrow (\mathcal{E}[M'], \sigma') \quad \text{if} \quad (M, \sigma) \rightarrow (M', \sigma') \quad (3.8)$$

where $\mathcal{E}[\cdot]$ is the left-to-right eager evaluation context, given by:

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

Fig. 1 Typing Rules for the Core Language

$$\begin{array}{c}
 [Var] \frac{\Gamma(x) = \alpha}{\Gamma; \Delta \vdash x : \alpha} \quad [Unit] \frac{}{\Gamma; \Delta \vdash () : \text{Unit}} \quad [Bool] \frac{}{\Gamma; \Delta \vdash \mathbf{b} : \text{Bool}} \\
 \\
 [Num] \frac{}{\Gamma; \Delta \vdash \mathbf{n} : \text{Nat}} \quad [Add] \frac{\Gamma; \Delta \vdash M_{1,2} : \text{Nat}}{\Gamma; \Delta \vdash M_1 + M_2 : \text{Nat}} \\
 \\
 [Abs] \frac{\Gamma, x : \alpha; \Delta \vdash M : \beta}{\Gamma; \Delta \vdash \lambda x^\alpha. M : \alpha \Rightarrow \beta} \quad [Rec] \frac{\Gamma, x : \alpha \Rightarrow \beta; \Delta \vdash \lambda y^\alpha. M : \alpha \Rightarrow \beta}{\Gamma; \Delta \vdash \mu x^{\alpha \Rightarrow \beta}. \lambda y^\alpha. M : \alpha \Rightarrow \beta} \\
 \\
 [App] \frac{\Gamma; \Delta \vdash M : \alpha \Rightarrow \beta \quad \Gamma; \Delta \vdash N : \alpha}{\Gamma; \Delta \vdash MN : \beta} \\
 \\
 [If] \frac{\Gamma; \Delta \vdash M : \text{Bool} \quad \Gamma; \Delta \vdash N_i : \alpha \ (i = 1, 2)}{\Gamma; \Delta \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : \alpha} \quad [Inj] \frac{\Gamma; \Delta \vdash M : \alpha_i}{\Gamma; \Delta \vdash \text{in}_i(M) : \alpha_1 + \alpha_2} \\
 \\
 [Case] \frac{\Gamma; \Delta \vdash M : \alpha_1 + \alpha_2 \quad \Gamma, x_i : \alpha_i; \Delta \vdash N_i : \beta}{\Gamma; \Delta \vdash \text{case } M \text{ of } \{\text{in}_i(x_i^{\alpha_i}). N_i\}_{i \in \{1, 2\}} : \beta} \\
 \\
 [Pair] \frac{\Gamma; \Delta \vdash M_i : \alpha_i \ (i = 1, 2)}{\Gamma; \Delta \vdash \langle M_1, M_2 \rangle : \alpha_1 \times \alpha_2} \quad [Proj] \frac{\Gamma; \Delta \vdash M : \alpha_1 \times \alpha_2}{\Gamma; \Delta \vdash \pi_i(M) : \alpha_i \ (i = 1, 2)} \\
 \\
 [Deref] \frac{\Delta(x) = \text{Ref}(\alpha)}{\Gamma; \Delta \vdash !x : \alpha} \quad [Assign] \frac{\Gamma; \Delta \vdash M : \alpha \quad \Delta(x) = \text{Ref}(\alpha)}{\Gamma; \Delta \vdash x := M : \text{Unit}}
 \end{array}$$

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 all n . Here \longrightarrow^n is the n -fold relational composition of \longrightarrow .

The grammar of types [22, 61] is given by the following grammar.

$$\alpha, \beta ::= \text{Unit} \mid \text{Bool} \mid \text{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \quad \rho ::= \alpha \mid \text{Ref}(\alpha)$$

We call Unit , Bool and Nat *base types*, α, β, \dots *value types*, and $\text{Ref}(\alpha), \dots$ *reference types*.

A *basis* is a finite map from names to types. Γ, Γ', \dots range over bases whose codomains are value types (which we sometimes call *environment basis*), while Δ, Δ', \dots range over bases whose codomains are reference types (which we sometimes call *reference basis*). $\text{dom}(\Gamma)$ (resp. $\text{dom}(\Delta)$) 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 [61], listed in Figure 1 ([*Add*] is an instance of the rules for operators). In $\Gamma; \Delta \vdash M : \alpha$, we always assume $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$. We extend the typing to stores, writing $\Gamma; \Delta \vdash \sigma$ if: (1) $\text{dom}(\Delta) = \text{dom}(\sigma)$ and (2) $\Gamma; \Delta \vdash \sigma(x) : \alpha$ iff $\Delta(x) = \text{Ref}(\alpha)$, for each $x \in \text{dom}(\sigma)$. 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). We omit empty environments from the judgment: for example, $\Gamma \vdash M : \alpha$ denotes $\Gamma; \emptyset \vdash M : \alpha$.

Remark 3.1 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 [9] and Section 8.1 for further discussions.

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

Definition 3.2 (semi-closed programs [51]) $\Gamma; \Delta \vdash M : \alpha$ is *semi-closed* (resp. *closed*) when $\text{dom}(\Gamma) = \emptyset$ (resp. $\text{dom}(\Gamma) = \text{dom}(\Delta) = \emptyset$), often written $\Delta \vdash M : \alpha$ (resp. $\vdash M : \alpha$). The notation $\Delta \vdash \sigma$ is understood similarly.

For brevity, henceforth we work under the following convention.

Convention 3.3

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

3.2 Terms and Formulae

Terms and Formulae. The logical language is that of first-order logic with equality [49, § 2.8] together with an assertion for the evaluation of stateful expressions. The grammar of terms and formulae follows.

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

The first set of expressions (ranged over by e, e', \dots) are *terms* while the second set are *formulae* (ranged over by $A, B, C, C' \dots$). Terms, which are from the logics for pure functions studied in [31, 35] except for $!x$, include all the constants (ranged over by c, c', \dots) 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 paring, projection and injection operation. The final term, $!e$, denotes the dereference of e . $\text{fv}(e)$ denotes the free variables occurring in e .

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' . \bullet is non-commutative. Note that having an explicit name x to denote the result in C' is crucial; in order to represent higher-order behaviours, evaluation formulae should be arbitrarily nested (recall the examples in Section 2), for which assigning a distinct name to each result is essential.

Fig. 2 Typing Rules for Terms and Formulae

$$\begin{array}{c}
 \frac{(\Gamma \cup \Delta)(x) = \rho}{\Gamma; \Delta \vdash x : \rho} \quad \frac{-}{\Gamma; \Delta \vdash () : \text{Unit}} \quad \frac{-}{\Gamma; \Delta \vdash n : \text{Nat}} \quad \frac{-}{\Gamma; \Delta \vdash b : \text{Bool}} \quad \frac{\Gamma; \Delta \vdash e : \text{Bool}}{\Gamma; \Delta \vdash \neg e : \text{Bool}} \\
 \\
 \frac{\Gamma; \Delta \vdash e_i : \alpha_i \ (i = 1, 2)}{\Gamma; \Delta \vdash (e_1, e_2) : \alpha_1 \times \alpha_2} \quad \frac{\Gamma; \Delta \vdash e : \alpha_1 \times \alpha_2}{\Gamma; \Delta \vdash \pi_i(e) : \alpha_i} \quad \frac{\Gamma; \Delta \vdash e : \alpha_i \ (i \in \{1, 2\})}{\Gamma; \Delta \vdash \text{inj}_i^{\alpha_1 + \alpha_2}(e) : \alpha_1 + \alpha_2} \quad \frac{\Gamma; \Delta \vdash e : \text{Ref}(\alpha)}{\Gamma; \Delta \vdash !e : \alpha} \\
 \\
 \frac{\Gamma; \Delta \vdash e_{1,2} : \rho}{\Gamma; \Delta \vdash e_1 = e_2} \quad \frac{\Gamma; \Delta \vdash C_{1,2}}{\Gamma; \Delta \vdash C_1 \star C_2} \ (\star \in \{\wedge, \vee, \supset\}) \quad \frac{\Gamma, x : \alpha; \Delta \vdash C}{\Gamma; \Delta \vdash \forall x^\alpha. C} \quad \frac{\Gamma, x : \alpha; \Delta \vdash C}{\Gamma; \Delta \vdash \exists x^\alpha. C} \\
 \\
 \frac{\Gamma; \Delta \vdash e_1 : \alpha \Rightarrow \beta \quad \Gamma; \Delta \vdash e_2 : \alpha \quad \Gamma; \Delta \vdash C \quad \Gamma, z : \beta; \Delta \vdash C'}{\Gamma; \Delta \vdash \{C\} e_1 \bullet e_2 = z \{C'\}}
 \end{array}$$

This is a departure from other logics such as JML where the unique variable is used to denote the result. Note that we can assert divergence by negating evaluation formulae.³ We also note that e does not include any PCF programs such as abstractions, applications and assignment which involve non-trivial dynamics (possibly infinite reductions for simplification of assertions). It is also worthwhile pointing out that quantification is only over variables of value type, not of reference type.

Terms and formulae are naturally 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 3.4 *Hereafter we only consider well-typed terms and formulae and often omit type annotations. Formulae are often called assertions.*

Convention 3.5 (formulae and terms)

1. We often write $\Theta \vdash C$ instead of $\Gamma; \Delta \vdash C$ with $\Theta = \Gamma \cup \Delta$. Θ, Θ', \dots 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 \top (definable as $1 = 1$) and falsity \bot (which is $\neg \top$).
4. The standard binding convention is always assumed. $\text{fv}(C)$ denotes the set of *free variables* in C .
5. $C^{\vec{x}}$ is C in which no name from \vec{x} freely occurs.

³ In the logics for pure higher-order functions studied in [31, 35], terms include \perp^α for each α which denotes divergence. Rather than using it, we have chosen to use evaluation formulae which entail termination, and to use its negation for indicating divergence. By the exclusion of \perp , all terms denote total values, which facilitates semantic treatment of assertions.

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 $\{C\} e_1 \bullet e_2 = () \{C'\}$.

Example 3.6 (assertions and their types)

1. $y = 6$ is an assertion which says y is equal to 6. On the other hand, $!y = 6$ says the content of a memory cell y is equal to 6. In the former, y is typed as Nat , while, in the latter, y is typed as $\text{Ref}(\text{Nat})$. Thus we have $y : \text{Ref}(\text{Nat}) \vdash !y = 6$.
2. The assertion $\text{Double}(u) \stackrel{\text{def}}{=} \forall n^{\text{Nat}}. \{T\} u \bullet n = 2 \times n \{T\}$ says that a procedure named u with type $\text{Nat} \Rightarrow \text{Nat}$ always returns the double of its argument. This is satisfied by, for example, $\lambda z. 2 \times z$. The predicate is typed as: $u : \text{Nat} \Rightarrow \text{Nat} \vdash \text{Double}(u)$.
3. Let us define $A(m) \stackrel{\text{def}}{=} \exists y' z'. (m = \langle \text{inj}_1(y'), z' \rangle \wedge \text{Even}(y') \wedge \text{Odd}(z'))$. Then we have, for example, $A(\langle \text{inj}_1(y), z \rangle) \supset \text{Even}(y)$.
4. Using the predicate introduced in 2 above, the assertion $\text{Double}(!x)$ says that a reference x stores a doubling function. Note we have $x : \text{Ref}(\text{Nat} \Rightarrow \text{Nat}) \vdash \text{Double}(!x)$. This assertion is satisfied, for example, after $x := \lambda z. 2 \times z$ is executed.
5. The assertion $\{\text{Double}(!x)\} u \bullet 3 = 6 \{\text{Double}(!x)\}$ says that, if 3 is fed to the function named by u under the initial state $\text{Double}(!x)$, then the result is 6, with the same final state. We have a typing

$$u : \text{Nat} \Rightarrow \text{Nat} ; x : \text{Ref}(\text{Nat}) \vdash \{\text{Double}(!x)\} u \bullet 3 = 6 \{\text{Double}(!x)\}$$

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, $\lambda y. (!x)y$, named as u .

6. The assertion

$$C \stackrel{\text{def}}{=} \forall i, n. \{!w = n\} !x \bullet i = 2 \times i \{!w = n + 1\}$$

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. For typing, we have $\emptyset ; x : \text{Ref}(\text{Nat} \Rightarrow \text{Nat}), w : \text{Ref}(\text{Nat}) \vdash C$. This assertion is satisfied when a procedure $f(w) \stackrel{\text{def}}{=} \lambda z. (w := !w + 1; z \times 2)$ is stored in x .

7. Using C in (4) 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 $!w = 1$. For typing, we have

$$u : \text{Nat} \Rightarrow \text{Nat} ; x : \text{Ref}(\text{Nat} \Rightarrow \text{Nat}), w : \text{Ref}(\text{Nat}) \vdash C'$$

This is satisfied by $\lambda y. (!x)y$ named as u , with x storing $f(w)$ above.

The following notion which distinguishes stateless and stateful is useful for the subsequent technical development.

Definition 3.7 (stateless formulae) A formula is *stateless* if, in the formula, a name of a reference type occurs only inside the pre/post conditions of evaluation formulae. A, A', B, \dots range over stateless formulae.

Example 3.8 (stateless formulae)

1. Under $x : \text{Nat}; y : \text{Ref}(\text{Nat})$, the formula “ $x = 3$ ” is stateless.
2. Under $x : \text{Nat}; y : \text{Ref}(\text{Nat})$, the formula “ $!y = 3$ ” is *not* stateless.
3. Under $x : \text{Unit} \Rightarrow \text{Unit}; y : \text{Ref}(\text{Nat})$, the formula “ $\{\text{Even}(!y)\}x \bullet ()\{\text{Odd}(!y)\}$ ” is stateless, but “ $\{\text{Even}(!y)\}x \bullet ()\{\text{Odd}(!y)\} \wedge !y = 3$ ” is not.

For distinction, we sometimes call well-typed formulae in general, *stateful formulae*. Note a stateless formula can use a stateful formula inside an evaluation formula.

3.3 Substitutions and Avoidance of Name Capture

Substitution for variables. In logics with equality and/or quantifications, capture-avoiding syntactic substitutions play a fundamental role in deduction [49, §2]. Due to the existence of evaluation formulae, some care is needed for their usage in the present theory. As an example of subtlety in the interplay between substitution and evaluation formulae, consider the following assertion:

$$C \stackrel{\text{def}}{=} m = 0 \wedge \forall i. \{m = 0 \wedge !y = i\} f \bullet () \{!y = i + 1\} \quad (3.9)$$

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 it would be. Note that the stateless predicate $m = 0$ in the precondition of the evaluation formula in (3.9) is equivalent to \top since it is already stipulated outside of the evaluation formulae. Now suppose we wish to substitute $!y$ for m . Then we obtain:

$$C[!y/m] \stackrel{\text{def}}{=} !y = 0 \wedge \forall i. \{!y = 0 \wedge !y = i\} f \bullet () \{!y = i + 1\} \quad (3.10)$$

which says the value 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*. As we noted, the second $m = 0$ in (3.9) is insignificant, so we have:

$$C \equiv m = 0 \wedge \forall i. \{!y = i\} f \bullet () \{!y = i + 1\} \quad (3.11)$$

which, when we substitute $!y$ for m , results in a quite different assertion:

$$!y = 0 \wedge \forall i. \{!y = i\} f \bullet () \{!y = i + 1\} \quad (3.12)$$

This assertion says (1) y *currently* stores 0, and (2) f always increments the content of y , whatever it would be. Starting from the intuitive reading of (3.9), the assertion (3.12) is a natural consequence of substitution: (3.10) does not give us what we should expect. Note this anomaly comes from implicit universal quantification in evaluation formulae, so that they can discuss *hypothetical state of stores*, necessary to describing behaviour of λ -abstractions. For example, the behaviour of $\lambda().y := y + 1$ need be described not

only for a particular (for example current) content of y but also for an arbitrary content of y . For the formal account, see Section 5.

To define safe substitution formally, we first extend the standard notion “ e is free for x in C ” in [49, §2.1] to evaluation formulae.

Definition 3.9 We say a term e^α is *free for x^α in C* if one of the following clauses is inductively satisfied:

1. e is free for x in $e_1 = e_2$.
2. e is free for x in $\neg C$ if it is free for x in C .
3. e is free for x in $C_1 \star C_2$ with $\star \in \{\wedge, \vee, \supset\}$ if it is free for x in both C_1 and C_2 .
4. e is free for x in $Qy.C$ with $Q \in \{\forall, \exists\}$ if e is free for x in C , and, moreover, $y \in \text{fv}(e)$ implies $x \notin \text{fv}(C)$.
5. e is free for x in $\{C_1\} e_1 \bullet e_2 = y \{C_2\}$ if e is free for x in C_1 and C_2 , and moreover $!w \in e$ for some w implies $x \notin \text{fv}(C_1) \cup \text{fv}(C_2)$.

Here $!w \in e$ means that $!w$ is a subexpression of e .

The final condition in Definition 3.9 is not restrictive. In the standard quantification theory, we can always rename bound variables to avoid capture of names; we can use existential quantification to “flush out” all names in dangerous positions. As an example, we can transform C in (3.9) as:

$$C' \stackrel{\text{def}}{=} \exists z. (m = 0 \wedge \forall i. \{z = 0 \wedge !y = i\} f \bullet () \{!y = i + 1\} \wedge z = m) \quad (3.13)$$

on which we can safely perform the previous substitution:

$$C'[!y/m] \stackrel{\text{def}}{=} \exists z. (!y = 0 \wedge \forall i. \{z = 0 \wedge !y = i\} f \bullet () \{!y = i + 1\} \wedge z = !y) \quad (3.14)$$

Formally we observe:

Proposition 3.10 *For any well-typed C , e^α and x^α , we can always effectively find C' such that $C' \equiv C$ and e is free for x in C' , by applying the following procedure repeatedly.*

1. Suppose $Qy.C_0$ is a subformula of C and $x \in \text{fv}(Qy.C_0)$. Assume $y \in \text{fv}(e)$. Then we transform $Qy.C_0$ to $Qy'.C_0[y'/y]$ with y' fresh.
2. Suppose $!w \in e$ for some w and $\{C_1\} e_1 \bullet e_2 = z \{C_2\}$ is a subformula of C such that $x \in \text{fv}(C_1) \cup \text{fv}(C_2)$. Then we transform it to $\exists x'. (x' = x \wedge \{C_1[x'/x]\} e_1 \bullet e_2 = z \{C_2[x'/x]\})$ where x' is fresh and $[x'/x]$ is a literal syntactic substitution.

Proof By induction on the structure of formulae. The cases where a formula is conjunction, disjunction, entailment, negation or equality are immediate from induction hypothesis. If it is $Qy.C$ with e free for x in C by induction hypothesis, the first clause transforms it into $Qy'.C[y'/y]$ with y' fresh. Since $y' \notin \text{fv}(e)$, it satisfies Definition 3.9 (4). Finally suppose a formula is $\{C\} e_1 \bullet e_2 = z \{C'\}$ with e free for x in C and C' . If e does not contain a dereference, the formula vacuously satisfies Definition 3.9 (5). Otherwise, the second clause moves x to the outside of C and C' , hence done. \square

By Proposition 3.10 we shall safely stipulate:

Convention 3.11 *From now on, whenever we write $C[e/x]$ in statements and judgements, we assume e to be free for x in C , unless otherwise specified.*

Substitution for dereferences. For the proof rule for assignment which we shall discuss later, we use an alternative form of substitution, written $C[e/!x]$, in which e is substituted for each “free” dereference $!x$ occurring in C . As we noted, this substitution should *not* affect the occurrences of $!x$ in the pre/post conditions of evaluation formulae. For example, let C be given by:

$$C \stackrel{\text{def}}{=} !x = 3 \wedge \forall i. \{!x = i\} f \bullet () \{!x = i + 1\} \quad (3.15)$$

which can be, for example, a post-condition of the assignment command $x := 3$, in which case the corresponding pre-condition is given as $C[3/!x]$ (in the proof rule for assignment we present later). But if we perform the substitution literally, the result of substitution becomes $3 = 3 \wedge \forall i. \{3 = i\} f \bullet () \{3 = i + 1\}$, which is a sheer nonsense. Intuitively, the evaluation formula in C :

$$\forall i. \{!x = i\} f \bullet () \{!x = i + 1\} \quad (3.16)$$

says that whenever we invoke the function f , the reference x is incremented, whatever the stored value would be at the time of invocation. This is because the intention of a substitution for a dereference is always to have the *current* content of x be equated with e , not hypothetical ones in pre/post conditions of evaluation formulae. Therefore we expect the substitution to work in the following way:

$$C[3/!x] \stackrel{\text{def}}{=} 3 = 3 \wedge \forall i. \{!x = i\} f \bullet () \{!x = i + 1\}, \quad (3.17)$$

which now makes sense. For clarity, we give the definition of the substitution as:

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

and others are defined homomorphically. In the case of $Qy^\alpha.(C[e/!x])$, recall the grammar of formulae do not allow quantification over reference names. Since $[e/!x]$ as defined above never affects pre/post conditions of evaluation formulae, the only capture of names we need to consider is the one induced by quantifiers. Based on this observation, we can extend the idea of Definition 3.9 to dereferences as follows.

Definition 3.12 We say a term e^α is *free for* $(!x)^\alpha$ in C if one of the following clauses is inductively satisfied:

1. e is free for $!x$ in $e_1 = e_2$.
2. e is free for $!x$ in $\neg C$ if it is free for $!x$ in C .
3. e is free for $!x$ in $C_1 \star C_2$ with $\star \in \{\wedge, \vee, \supset\}$ if it is free for $!x$ in both C_1 and C_2 .
4. e is free for $!x$ in $Qy.C$ with $Q \in \{\forall, \exists\}$ if e is free for $!x$ in C and moreover, $y \in \text{fv}(e)$ implies $x \notin \text{fv}(C)$.
5. e is free for $!x$ in $\{C_1\} e_1 \bullet e_2 = y \{C_2\}$ if e is free for $!x$ in C_1 and C_2 .

Thus we only need the standard alpha-conversion to avoid the capture of names for this type of substitutions. We stipulate:

Convention 3.13 Whenever we write $C[e/!x]$, we assume e is free for $!x$ in C .

3.4 Judgement

Following Hoare [28], a judgement 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 M gets evaluated starting from a store satisfying C , then it terminates with the final state and the resulting value, named u , together satisfying C' .

As in Hoare logic, the distinction between primary names and auxiliary names plays an important role in both proof rules and semantics of the logic.

Definition 3.14 (primary/auxiliary names in a judgement) Let $\models \{C\} M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ be well-typed. Then the *primary names* in this judgement are $\text{dom}(\Gamma, \Delta) \cup \{u\}$. 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.*

For example, in a judgement “ $\{x = i\} 2 \times x^{x:\text{Nat}; \text{Nat}} :_u \{u = 2 \times i\}$ ”, x and u are primary while i is auxiliary. u is in addition the anchor.

Judgements are typed as follows:

Definition 3.15 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 $\text{dom}(\Theta) \cap (\text{dom}(\Gamma, \Delta) \cup \{u\}) = \emptyset$.

In brief, a judgement is well-typed if (1) its pre/post conditions are themselves well-typed and have types which conform to the typing of the program; and if (2) its pre/post conditions have a common typing for auxiliary variables. Since a formula is (formally) explicitly typed, we can easily check whether a given judgement is well-typed or not, and, if it is, can give a unique typing to it.

Convention 3.16 *Henceforth we assume that any given judgement is well-typed. For brevity, we often omit the typing from a judgement when it is understood from the context, writing $\{C\} M :_u \{C'\}$.*

3.5 Axioms for Data Types and Evaluation Formulae

For deriving judgements using the proof rules presented in the next section, one often needs to calculate validity of assertions (the class of models we consider will be discussed in Section 5). For syntactic calculation of validity, we shall freely use axioms, rules and theorems from propositional calculus, as well as first-order logic with equality [49, §2.8] (which includes the substitutivity rule, $x = y \wedge C(x, x) \supset C(x, y)$ where

$C(x, y)$ substitutes y for some of occurrences of x in $C(x, x)$ [49, §2.8]); and formal number theory (for example Peano arithmetic) and its theorems. In addition, there are also natural axioms for data types, as well as those for evaluation formulae, the main ones out of which we shall discuss in this subsection. These axioms will be justified through the semantics of assertions in Section 5.

Axioms for Data Types. We first list basic axioms for data types except for natural numbers, in Figure 3. (t1) is equivalent to:

$$C[()/x^{\text{Unit}}] \equiv C \quad (3.18)$$

which is often useful in practice. (t2) is the analogue of (t1) for boolean values. (t2) is equivalent to the following axiom, for an arbitrary (appropriately typed) C .

$$(C \equiv C[t/x^{\text{Bool}}]) \vee (C \equiv C[f/x^{\text{Bool}}]) \quad (3.19)$$

Fig. 3 Axioms for Data Types

- | | |
|---|--|
| <p>(t1) $x^{\text{Unit}} = ()$</p> <p>(t3) $t \neq f$</p> <p>(t5) $\langle \pi_1(x), \pi_2(x) \rangle = x$</p> <p>(t7) $\text{in}_1(x) \neq \text{in}_2(y)$</p> | <p>(t2) $x^{\text{Bool}} = t \vee x^{\text{Bool}} = f$</p> <p>(t4) $\pi_i(\langle x_1, x_2 \rangle) = x_i$</p> <p>(t6) $\text{in}_i(x) = \text{in}_i(y) \supset x = y$</p> <p>(t8) $x^{\text{Ref}(\alpha)} \neq y^{\text{Ref}(\alpha)}$ (x and y are distinct)</p> |
|---|--|
-

Further (t2) justifies the use of boolean terms as formulae, via the following “macros” . Below let $\star \in \{\wedge, \vee, \supset\}$, then we can define: $e^{\text{Bool}} \stackrel{\text{def}}{=} e = t, \neg e \equiv \neg e \equiv e = f, e_1 \star e_2 \equiv e_1 \star e_2$, and $e_1 = e_2 \equiv (e_1 = e_2)$. Note these translations are possible because all boolean terms denote total values. The use of boolean terms as formulae is often convenient in practice. (t4) and (t5) are the standard extensional axioms for pairs and projections. (t6) and (t7) are standard axioms for sums.

(t8) says we do not assume aliasing (i.e. non-trivial equations on reference names) *at the logical level*, which reflects the lack of aliasing at the level of programs’ behaviours in the target programming language. (t8) says any equations between top-level reference typed terms are vacuous — either it has the shape $x = x$ in which case it is always true, or it has the shape $x = y$ in which case it is always false⁴. Thus, in effect, there is no need to have reference names in formulae except in the shape of dereferencing (note further, by typing, reference-typed terms never appear as proper subterms in the logic except as dereferences). Thus, under (t8), we safely stipulate:

Convention 3.17 *Henceforth we assume each variable, say x , of a reference type always occurs in a formula as the subterm of ! x .*

⁴ Equations $x^{\text{Ref}(\alpha)} = y^{\text{Ref}(\beta)}$ for $\alpha \neq \beta$ are not well-typed.

Remark 3.18 Convention 3.17 suggests that, for a variable of a reference type x , we may as well write x to denote its content, just as in the standard Hoare logic. Using the explicit notation for dereference has the merit of clarity in typing, clear and rigorous understanding of substitutions, and its direct extensibility to the logic which allows aliasing. For the last point, it becomes essential to express aliasing *logically*, i.e. to have non-trivial equations on reference names in the logic, when the target behaviour involves aliasing. For example consider $\lambda x^{\text{Ref}(\text{Nat})}.x := 0$. The program affects the content of any reference it receives. We can describe the behaviour of this program (naming it as u) thus:

$$\forall x^{\text{Ref}(\text{Nat})}, i^{\text{Nat}}. \{!x = i\} u \bullet x = () \{!x = 0\} \quad (3.20)$$

In (3.20), the reference-typed variable x is quantified. Another description uses an inequation on references in addition:

$$\forall x^{\text{Ref}(\text{Nat})}, i^{\text{Nat}}. \{x \neq y \wedge !y = 1\} u \bullet x = () \{!y = 1\} \quad (3.21)$$

which says if this program receives a variable which is distinct from y , and if y originally stores 1, then y will continue to store 1 after invocation of this program terminates. The logic with aliasing, which involves a non-trivial extension to the present logic in addition to the erasure of (I2), is discussed in [9].

Fig. 4 Axioms for Evaluation Formulae

(e1)	$\{C_1\} x \bullet y = z \{C\} \wedge \{C_2\} x \bullet y = z \{C\}$	\equiv	$\{C_1 \vee C_2\} x \bullet y = z \{C\}$	
(e2)	$\{C\} x \bullet y = z \{C_1\} \wedge \{C\} x \bullet y = z \{C_2\}$	\equiv	$\{C\} x \bullet y = z \{C_1 \wedge C_2\}$	
(e3)	$\{\exists w. C\} x \bullet y = z \{C'\}$	\equiv	$\forall w. \{C\} x \bullet y = z \{C'\}$	$w \notin \text{fv}(C')$
(e4)	$\{C\} x \bullet y = z \{\forall w. C'\}$	\equiv	$\forall w. \{C\} x \bullet y = z \{C'\}$	$w \notin \text{fv}(C)$
(e5)	$\{A \wedge C\} x \bullet y = z \{C'\}$	\equiv	$A \supset \{C\} x \bullet y = z \{C'\}$	
(e6)	$\{C\} x \bullet y = z \{A \supset C'\}$	\supset	$A \supset \{C\} x \bullet y = z \{C'\}$	$z \notin \text{fv}(A)$
(e7)	$\{C\} x \bullet y = z \{C'\}$	\supset	$\{C \wedge A\} x \bullet y = z \{C' \wedge A\}$	
(e8)	$\{C_0\} x \bullet y = z \{C'_0\}$	\supset	$\{C\} x \bullet y = z \{C'\}$	where $C \supset C_0$ and $C'_0 \supset C'$.
(ext)	$\text{Ext}^{\Delta: \alpha \Rightarrow \beta}(x, y)$	\supset	$x = y$	

Axioms for Evaluation Formulae. The rules in Figure 4 are about evaluation formulae. (e1) and (e2) should read naturally. From these rules we obtain:

$$\{C_1\} x \bullet y = z \{C'_1\} \wedge \{C_2\} x \bullet y = z \{C'_2\} \supset \{C_1 \wedge C_2\} x \bullet y = z \{C'_1 \wedge C'_2\} \quad (3.22)$$

$$\{C_1\} x \bullet y = z \{C'_1\} \wedge \{C_2\} x \bullet y = z \{C'_2\} \supset \{C_1 \vee C_2\} x \bullet y = z \{C'_1 \vee C'_2\} \quad (3.23)$$

The axioms from (e3) to (e6) are also naturally read. In (e6), the converse (hence logical equivalence) does not generally hold. As an example, consider the following inference,

assuming the converse of (e6) were indeed valid (\Rightarrow indicates implications).

$$\begin{aligned} \top &\Rightarrow F \supset \{T\}x \bullet y = z \{T\} \\ &\Rightarrow \{T\}x \bullet y = z \{F \supset T\} \\ &\Rightarrow \{T\}x \bullet y = z \{T\} \end{aligned}$$

This (wrong) inference claims any programs x of type $\alpha \Rightarrow \beta$ will converge for any argument y of type α , an absurd statement.

In (e7), we add the conjunction with a stateless formula to both of the pre and post conditions: we cannot add a stateful formula since the evaluation may change a state about which it asserts.

(e8) uses a side condition on validity. Note we can substitute dereferences to non-dereferences to obtain the following proper axiom schema. Below we assume we are working under Δ such that $\text{dom}(\Delta) = \{\vec{r}\}$, let \vec{i} be fresh, and define $\sigma = [\vec{i}/!\vec{r}]$.

$$(\forall \vec{i}. (C\sigma \supset C_0\sigma)) \wedge \{C_0\}x \bullet y = z \{C'_0\} \wedge (\forall \vec{i}. (C'_0\sigma \supset C'\sigma)) \Rightarrow \{C\}x \bullet y = z \{C'\}$$

which is equivalent to the original law.

The final axiom, (ext), is about extensionality, i.e. if two programs behave equivalently as (imperative) functions then they should be equated in the logic. The axiom augments the corresponding one in [31] with side effects (read/writes on imperative variables). In the axiom, a reference type Δ is mentioned, which, when used in the judgement, should coincide with the basis of the program being discussed in the judgement. The axiom uses the following formula.

Definition 3.19 (extensionality formulae) Let $\Delta = \vec{r} : \text{Ref}(\vec{\gamma})$.

$$\begin{aligned} \text{Ext}^{\Delta; \alpha \Rightarrow \beta}(x, y) &\stackrel{\text{def}}{=} \forall h^\alpha, i^\beta, \vec{j}^{\vec{\gamma}}, \vec{k}^{\vec{\gamma}}. (\{!\vec{r} = \vec{j}\} x \bullet h = z \{z = i \wedge !\vec{r} = \vec{k}\} \\ &\equiv \{!\vec{r} = \vec{j}\} y \bullet h = w \{w = i \wedge !\vec{r} = \vec{k}\}) \end{aligned}$$

The predicate expresses an extensional equality of two procedures, saying:

Whenever x converges for some argument and for some stored value at r , then y converges for the same argument and for the same stored value at r , with the same return value and the same final state; and vice versa.

Note that extensionality is stipulated relative to Δ .

Fig. 5 Proof Rules (1): compositional rules

$$\begin{array}{c}
 [Var] \frac{}{\{C[x/u]\} \bar{x} :_u \{C\}} \quad [Const] \frac{}{\{C[c/u]\} \bar{c} :_u \{C\}} \\
 [Op] \frac{C_0 \stackrel{\text{def}}{=} C \quad \{C_i\} M_i :_{m_i} \{C_{i+1}\} \ (0 \leq i \leq n-1) \quad C_n \stackrel{\text{def}}{=} C'[\text{op}(m_0..m_{n-1})/u]}{\{C\} \text{op}(M_0..M_{n-1}) :_u \{C'\}} \\
 [Abs] \frac{\{C \wedge A^{\bar{x}}\} M :_m \{C'\}}{\{A\} \lambda x. M :_u \{\forall x. \{C\} u \bullet x = m \{C'\}\}} \quad [App] \frac{\{C\} M :_m \{C_0\} \quad \{C_0\} N :_n \{C_1 \wedge \{C_1\} m \bullet n = u \{C'\}\}}{\{C\} MN :_u \{C'\}} \\
 [If] \frac{\{C\} M :_b \{C_0\} \quad \{C_0[t/b]\} M_1 :_u \{C'\} \quad \{C_0[f/b]\} M_2 :_u \{C'\}}{\{C\} \text{if } M \text{ then } M_1 \text{ else } M_2 :_u \{C'\}} \\
 [In_1] \frac{\{C\} M :_v \{C'[\text{in}_1(v)/u]\}}{\{C\} \text{in}_1(M) :_u \{C'\}} \quad [Case] \frac{\{C^{\bar{x}}\} M :_m \{C_0^{\bar{x}}\} \quad \{C_0[\text{in}_i(x_i)/m]\} M_i :_u \{C'^{\bar{x}}\}}{\{C\} \text{case } M \text{ of } \{\text{in}_i(x_i). M_i\}_{i \in \{1,2\}} :_u \{C'\}} \\
 [Pair] \frac{\{C\} M_1 :_{m_1} \{C_0\} \quad \{C_0\} M_2 :_{m_2} \{C'[(m_1, m_2)/u]\}}{\{C\} \langle M_1, M_2 \rangle :_u \{C'\}} \quad [Proj_1] \frac{\{C\} M :_m \{C'[\pi_1(m)/u]\}}{\{C\} \pi_1(M) :_u \{C'\}} \\
 [Deref] \frac{}{\{C[!x/u]\} \bar{!x} :_u \{C\}} \quad [Assign] \frac{\{C\} M :_m \{C'[m/!x][()/u]\}}{\{C\} x := M :_u \{C'\}} \\
 [Rec] \frac{\{A^{\bar{x}i} \wedge \forall j \leq i. B(j)[x/u]\} \lambda y. M :_u \{B(i)^{\bar{x}}\}}{\{A\} \mu x. \lambda y. M :_u \{\forall i. B(i)\}}
 \end{array}$$

4 Proof Rules

4.1 Proof Rules (1): Composition Rules

In this section we introduce the proof rules for the present logic. When we derive a judgement for a program using these rules, the derivation precisely follows the structure of the program, except for intermittent use of structural rules. The proof rules which follow the structure of programs are given in Figure 5. We use Convention 3.5 and the following convention.

Convention 4.1 (convention on the use of names)

- Free i, j, \dots exclusively range over auxiliary names.
- In each proof rule, we assume all occurring judgements to be well-typed. In addition 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).

- When M has unit type, we can safely ignore the anchor u by the axiom $C'[(\)/u^{\text{Unit}}] \equiv C'$ in § 3.5, so that we shall write $\{C\}M\{C'\}$ for $\{C\}M :_u \{u = (\) \wedge C'\}$ with $u \notin \text{fv}(C')$ just like a Hoare triple.

In the following we illustrate the rules in Figure 5 one by one.

[Var] says that, if we wish to specify something for x as a program named u , then we should specify the same thing for x in the assumption, with substituting x for u . As simple instances, we infer:

$$\frac{}{\{x = 3\} x :_u \{u = 3\}} \quad \frac{}{\{x = x\} x :_u \{u = x\}} \quad (4.1)$$

In the second inference, note that $x = x$ is equivalent to \top . Combined with the consequence rule given later, we can conclude $\{\top\}x :_u \{u = x\}$ from (4.1).

[Const] says that, if we wish to specify something about a constant named u , then we should be able to say the same thing about that constant (which can be tested by putting that constant into that variable u). We give two instances of this axiom.

$$\frac{}{\{Even(2)\} 2 :_u \{Even(u)\}} \quad \frac{}{\{Odd(2)\} 2 :_u \{Odd(u)\}} \quad (4.2)$$

The first judgement is equivalent to $\{\top\} 2 :_u \{Even(u)\}$ since $Even(2)$ is always true. In contrast, $Odd(2)$ in the second judgement is always false. Thus we are assuming something impossible, so that concluding absurdly in the post-condition is permitted.

[Op] is a rule for standard first-order operators. The rule assumes op appears as a term constructor in the logical language. In the nullary case, the premise becomes $C \stackrel{\text{def}}{=} C'[c/u]$, so the rule is equivalent to **[Const]**. A simple instance of this rule follows.

$$[Add] \frac{\{C\} M :_m \{C_0\} \quad \{C_0\} N :_n \{C'[m+n/u]\}}{\{C\} M+N :_u \{C'\}}$$

As an example of application of this rule, we infer:

$$\frac{\{Odd(x)\} x :_m \{Odd(m)\} \quad \{Odd(m)\} 1 :_n \{Even(u)[m+n/u]\}}{\{Odd(x)\} x+1 :_u \{Even(u)\}} \quad (4.3)$$

[Abs] says that if, under the assumptions C and A (which is a stateless formula and does not have x), M named m returns a result and a state which together satisfy C' , then $\lambda x.M$ named as u satisfies, under A , if u is invoked by x at C , it terminates with a state and a resulting value (named m) which together satisfy C' . Let us see how we can use this rule, taking a simple example (we assume the premise has been inferred already).

$$\frac{\{\top \wedge z = 1\} !y = x+z :_m \{m = (\) \wedge !y = x+1\}}{\{z = 1\} \lambda x. !y = x+z :_u \{ \forall x. \{\top\} u \bullet x \{!y = x+1\} \}} \quad (4.4)$$

In the above inference, “ $z = 1$ ” in the premise is taken as the stateless formula (A in the rule), to be moved into the precondition of the conclusion. Note we can also regard this formula as part of C in the rule, in which case we obtain:

$$\frac{\{\top \wedge z = 1\} !y = x+z :_m \{m = (\) \wedge !y = x+1\}}{\{\top\} \lambda x. !y = x+z :_u \{ \forall x. \{z = 1\} u \bullet x \{!y = x+1\} \}} \quad (4.5)$$

which is semantically equivalent to (4.4) (we later discuss how we can formally mediate (4.4) and (4.5)).

[App] says that: if (1) starting from C , a program M is evaluated into a value named m which, together with the final state, satisfies C_0 ; and (2) starting from C_0 , a program N is evaluated into a value named n which reaches a final state C_1 as well as $\{C_1\}m \bullet n = z\{C'\}$ (which says starting from C_1 , applying n to m results in a value z which, together with a final state, satisfies C'). Then MN named as u will start from C and reaches C' . A simple example inference follows. Below we let

$$A \stackrel{\text{def}}{=} \forall xi. \{!y = i\} m \bullet x = u \{u = !y + !x + 1 \wedge !y = i + 1\}$$

Again we assume the premises are already inferred.

$$\frac{\begin{array}{l} \{!y = 1\} \lambda x. (y := !y + 1; x + !y) :_m \{!y = 1 \wedge A\} \\ \{!y = 1 \wedge A\} 1 :_n \{!y = 1 \wedge \{!y = 1\} m \bullet 1 = u \{u = 3 \wedge !y = 2\}\} \end{array}}{\{!y = 1\} (\lambda x. (y := !y + 1; x + !y)) 1 :_u \{u = 3 \wedge !y = 2\}}$$

From **[Abs]** and **[App]**, the proof rule for the let construct is derivable (the derivation is shown in Section 7.1 for a slightly simpler case).

$$[\text{Let}] \frac{\{C\} M :_x \{C_0\} \quad \{C_0\} N :_u \{C'\}}{\{C\} \text{let } x = M \text{ in } N :_u \{C'\}}$$

[Let] states that, if, starting from C , the evaluation of M , named x , terminates and reaches a state and a result which together satisfy C_0 , and if, starting from C_0 , the evaluation of N , named u , terminates and reaches C' , then, starting from C , the evaluation of $\text{let } x = M \text{ in } N$, also named u , will terminate and reach C' . Note x is not in C' by our convention. As an example, let $M \stackrel{\text{def}}{=} (!y := !y + 1; !y)$ and $N \stackrel{\text{def}}{=} (!y := !y + x; !y + 1)$. Then we infer:

$$\frac{\{!y = 0\} M :_x \{!y = 1 \wedge x = 1\} \quad \{!y = 1 \wedge x = 1\} N :_u \{!y = 2 \wedge u = 3\}}{\{!y = 0\} \text{let } x = M \text{ in } N :_u \{!y = 2 \wedge u = 3\}} \quad (4.6)$$

[If] says that, if: (1) a boolean-typed program M named b satisfies C_0 under C , (2) M_1 reaches C' starting from C_0 assuming it says b is true, and (3) M_2 reaches C' starting from C_0 assuming it says b is false, then the program $\text{if } M \text{ then } M_1 \text{ else } M_2$ will reach C' starting from the initial state C . As a simple example of its application, we infer:

$$\frac{\{x = t\} x :_b \{b = t\} \quad \{t = t\} y := 0 \{!y = 0\} \quad \{f = t\} y := 1 \{!y = 0\}}{\{x = t\} \text{if } x \text{ then } y := 0 \text{ else } y := 1 \{!y = 0\}} \quad (4.7)$$

In (4.7), the last judgement in the antecedent, $\{f = t\} y := 1 :_u \{u = () \wedge !y = 0\}$, holds since, under the absurd assumption $f = t$, we can conclude anything.

[Inj, Case, Pair, Proj] The proof rules for injection, pair and projection are similar to **[Op]**, while the proof rule for the case construct is similar to **[If]**. We only show a simple reasoning for injection (the derivation of the antecedent is treated later).

$$\frac{\{T\} () :_v \{in_1(v) = in_1(())\}}{\{T\} in_1(()):_u \{u = in_1(())\}} \quad (4.8)$$

[Deref] is understood as **[Var, Const]**, showing that dereferenced imperative variables and program variables work in similar ways. In the rule, we assume $!x$ is free for m in C (cf. Definition 3.9, page 17, §3.2). The rule says that, if we wish to have C for $!x$ (as a program) named u , then we should assume the same thing about the content of x , substituting $!x$ for u . As an example, we infer:

$$\frac{-}{\overline{\{Even(!x)\} !x :_u \{Even(u)\}}} \quad (4.9)$$

[Assign] uses two substitutions for reasoning about assignment, $[m/!x]$ and $[(\)/u]$. The notation $C[m/!x]$ stands for replacing all occurrences of $!x$ in C except in the pre/post conditions of evaluation formulae by m , assuming m is safe for $!x$ in C (for precise definitions, see Definition 3.12, page 18, §3.2); whereas $[(\)/u]$ is the standard substitution of $(\)$ for a unique anchor 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). The rule says that: if we wish to have, under the initial environment/state C , the result of executing the command $x := M$ (named u) to satisfy C' , then we demand, starting from C , M named m (the program to be assigned to x) evaluates to the state satisfied by the predicate C' with its occurrences of $!x$ substituted for m (since the state of $!x$ should satisfy what m satisfies) and substituting $(\)$ for u (since the assignment command always terminates to return the unique value $(\)$). Below we show a simple example of the use of **[Assign]**.

$$\frac{\{!y = 1\} !y + 1 :_m \{Even(m) \wedge (\) = (\)\}}{\{!y = 1\} y := !y + 1 \{Even(!y)\}} \quad (4.10)$$

Later we shall see how we can derive the standard assignment rule in Hoare Logic [28] from the above rule.

[Rec] is the proof rule for the total correctness of recursion (slightly strengthening the rule in [31, 35]). i in the premise should be auxiliary. Observe i is implicitly universally quantified, and that, if $i = 0$, it holds that (since j ranges over natural numbers) $\forall j \preceq i. B(j)[x/u]$ is logically equivalent to \top . We can thus read the rule as:

- (base case) Assuming A , the program named u satisfies, without assuming anything about x , already $B(0)$.
- (inductive case) Assuming A and $B(i)[x/u]$ for x for each $i \leq n$, the program named u satisfies $B(n+1)$.

Then we conclude that the program satisfies $B(n)$ for each n . A simple example of the use of the rule follows. Let M and $B(i)$ in the rule to be the following program and formula:

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

$$B(i) \stackrel{\text{def}}{=} \{\top\} u \bullet i \{!y = i!\}.$$

Then, by applying **[Rec]** using the same variables, we obtain:

$$\frac{\{\top\} \lambda y. M :_u \{\{\top\} u \bullet 0 \{!z = 1\}\} \quad \{\top \wedge \forall j \leq i. \{\{\top\} x \bullet j \{!z = j!\}\}\} \lambda y. M :_u \{\{\top\} u \bullet (i+1) \{!z = (i+1)!\}\}}{\{\top\} \mu x. \lambda y. M :_u \{\forall n. \{\top\} u \bullet n \{!z = n!\}\}}$$

[*Rec*] allows a simple extension to induction on an arbitrary well-ordering. Below i has type α not limited to Nat . $\llbracket \alpha \rrbracket$ denotes the closed values of type α , and $\preceq_{\llbracket \alpha \rrbracket}$ is a well-founded ordering defined over $\llbracket \alpha \rrbracket$ (for example $\text{Nat} \times \text{Nat}$ can be equipped with the lexicographic ordering: we may also refine so that a well-ordering is given to a subset of a domain).

$$[\text{Rec-Wf}] \frac{\{A \wedge \forall j^{\alpha} \preceq_{\llbracket \alpha \rrbracket} i. B(i)[x/u]\} \lambda y. M :_u \{B(i)\}}{\{A\} \mu x. \lambda y. M :_u \{\forall i^{\alpha}. B(i)\}}$$

By [*Rec*], we can also extend [*Let*] to a rule for multiple recursive lets.

$$[\text{LetRec}] \frac{\{A \wedge (\wedge_k \forall j \preceq i. E_k(j)[\vec{y}/\vec{x}])\} V_k :_{x_k} \{E_k(i)\} \quad (1 \leq k \leq n)}{\{A \wedge (\wedge_k \forall i. E_k(i)[\vec{y}/\vec{x}])\} M :_u \{B\}} \quad \{A\} \text{letrec } \vec{x} = \vec{V} \text{ in } M :_u \{B\}$$

In Section 7.1 later, we shall show how several known proof rules for total correctness on loop and recursive procedures can be derived using these recursion rules.

4.2 Proof Rules (2): Structural Rules

Figure 6 presents two basic structural rules, [*Promote*] and [*Consequence*], as well as a few additional ones, including strengthening of [*Consequence*] due to Kleymann [41]. Structural rules do not change the shape of the concerned program, manipulating only formulae.

In [*Promote*], we add a predicate on imperative variables to a judgement on values, which is originally inferred without state. Forming a conjunction with the same formula in both the precondition and postcondition can be inconsistent in standard Hoare Logic, but is consistent in the present setting, since values have no effects. As a simple example, under the typing $x : \text{Ref}(\text{Nat})$, we have:

$$\frac{\{T\} 3 :_u \{u = 3\}}{\{T \wedge !x = 2\} 3 :_u \{u = 3 \wedge !x = 2\}} \quad (4.11)$$

which adds the same state at x in the pre/post conditions.

[*Consequence*] is the standard consequence rule [28]. As is standard, the conditions $C \supset C_0$ and $C'_0 \supset C'$ indicate that the entailment holds under arbitrary models (as defined in Section 5).

[$\wedge \supset$] moves a stateless formula in the pre-condition to a negative position in the post-condition. Note a program is restricted values: if it were about general programs, then, for example, when $A = F$, the premise is vacuously true for any program even though the conclusion claims termination (note this relates to the requirement for total correctness). [$\supset \wedge$] does the reverse transfer, which holds for general programs which may not be values. As a simple application, we mediate the conclusions of the two inferences from the same premise, (4.4) and (4.5) in Page 24. First we infer:

$$\begin{array}{l} 1. \quad \{z = 1\} \lambda x. !y = x + z :_u \{ \forall x. \{T\} u \bullet x \{!y = x + 1\} \} \quad (4.4) \\ 2. \quad \{T\} \lambda x. !y = x + z :_u \{ z = 1 \supset \forall x. \{T\} u \bullet x \{!y = x + 1\} \} \quad (\wedge \supset) \\ 3. \quad \{T\} \lambda x. !y = x + z :_u \{ \forall x. \{z = 1\} u \bullet x \{!y = x + 1\} \} \quad (\text{Consequence}) \end{array}$$

Fig. 6 Structural Rules

$$\begin{array}{c}
\text{[Promote]} \frac{\{A\} V :_u \{B\}}{\{A \wedge C\} V :_u \{B \wedge C\}} \quad \text{[Consequence]} \frac{C \supset C_0 \quad \{C_0\} M :_u \{C'_0\} \quad C'_0 \supset C'}{\{C\} M :_u \{C'\}} \\
\text{[\wedge-\supset]} \frac{\{C \wedge A\} V :_u \{C'\}}{\{C\} V :_u \{A \supset C'\}} \quad \text{[\supset-\wedge]} \frac{\{C\} M :_u \{A \supset C'\}}{\{C \wedge A\} M :_u \{C'\}} \\
\text{[\vee-Pre]} \frac{\{C_1\} M :_u \{C\} \quad \{C_2\} M :_u \{C\}}{\{C_1 \vee C_2\} M :_u \{C\}} \quad \text{[\wedge-Post]} \frac{\{C\} M :_u \{C_1\} \quad \{C\} M :_u \{C_2\}}{\{C\} M :_u \{C_1 \wedge C_2\}} \\
\text{[Aux}\exists\text{]} \frac{\{C\} M :_u \{C'^i\}}{\{\exists i.C\} M :_u \{C'\}} \quad \text{[Aux}\forall\text{]} \frac{\{C'^i\} M :_u \{C'\}}{\{C\} M :_u \{\forall i.C'\}} \\
\text{[Aux}_{inst}\text{]} \frac{\{C(i^\alpha)\} M :_u \{C'(i^\alpha)\} \quad \alpha \text{ is base}}{\{C(c^\alpha)\} M :_u \{C'(c^\alpha)\}} \quad \text{[Aux}_{abst}\text{]} \frac{\{C(c^\alpha)\} M :_u \{C'(c^\alpha)\} \text{ for all } c \text{ in type } \alpha}{\{C(i)\} M :_u \{C'(i)\}} \\
\text{[Invariance]} \frac{\{C\} M^{\Gamma;\Delta;\alpha} :_m \{C'\}}{\{C \wedge A\} M^{\Gamma;\Delta;\alpha} :_m \{C' \wedge A\}} \\
\text{[Consequence-Aux]} \frac{\{C_0\} M^{\Gamma;\Delta;\alpha} :_u \{C'_0\} \quad C \supset \exists \vec{j}. (C_0[\vec{j}/\vec{i}] \wedge (C'_0[\vec{y}/\vec{x}][\vec{j}/\vec{i}] \supset C'[\vec{y}/\vec{x}]))}{\{C\} M :_u \{C'\}}
\end{array}$$

In $[\text{Consequence-Aux}]$, we set $\{\vec{x}\} = \text{dom}(\Gamma, \Delta) \cup \{u\}$, $\{\vec{i}\} = \text{fv}(C, C', C_0, C'_0) \setminus \{\vec{x}\}$, and \vec{j} (resp. \vec{y}) are fresh. We assume no auxiliary reference names occur.

The other direction just does the converse:

$$\begin{array}{l}
1. \quad \{T\} \lambda x. !y = x + z :_u \{ \forall x. \{z = 1\} u \bullet x \{!y = x + 1\} \} \quad (4.5) \\
\hline
2. \quad \{T\} \lambda x. !y = x + z :_u \{ z = 1 \supset \forall x. \{T\} u \bullet x \{!y = x + 1\} \} \quad (\text{Consequence}) \\
\hline
3. \quad \{z = 1\} \lambda x. !y = x + z :_u \{ \forall x. \{T\} u \bullet x \{!y = x + 1\} \} \quad (\supset-\wedge)
\end{array}$$

The applications of $[\text{Consequence}]$ in Line 3 of the first inference and Line 2 of the second are possible by the axiom (e5) for evaluation formulae given in §3.5.

$[\vee\text{-Pre}]$ says that, if a given program can guarantee C starting from any state satisfying C_1 as well as starting from any state satisfying C_2 , then the program can also guarantee C starting from a state which satisfies either C_1 or C_2 . $[\wedge\text{-Post}]$ is the dual of $[\vee\text{-Pre}]$. One of the uses of $[\wedge\text{-Post}]$ is when we infer for a closure, i.e. for λ -abstraction. Consider we wish to derive the following judgement.

$$\{T\} \lambda(). y := y + 1 :_u \{ \{ \text{Even}(!y) \} u \bullet () \{ \text{Odd}(!y) \} \wedge \{ \text{Odd}(!y) \} u \bullet () \{ \text{Even}(!y) \} \}$$

Even if we can make these two statements in the post-condition into one without changing meaning in the above case, it gives a natural specification, as in more practical cases.

If we wish to derive such a judgement directly, since $[Abs]$ has a conclusion of a single evaluation formula, $[\wedge-Post]$ is particularly useful. For example, we can reason as follows in the above case.

$$\frac{\begin{array}{c} \{T\} \lambda().y := y + 1 :_u \{ \{Even(!y)\}u \bullet () \{Odd(!y)\} \} \\ \{T\} \lambda().y := y + 1 :_u \{ \{Odd(!y)\}u \bullet () \{Even(!y)\} \} \end{array}}{\{T\} \lambda().y := y + 1 :_u \{ \{Even(!y)\}u \bullet () \{Odd(!y)\} \wedge \{Odd(!y)\}u \bullet () \{Even(!y)\} \} }$$

As an aside, we observe there is the following variant of $[\wedge-Post]$.

$$[\wedge-Pre/Post] \frac{\{C_1\}M :_u \{C'_1\} \quad \{C_2\}M :_u \{C'_2\}}{\{C_1 \wedge C_2\}M :_u \{C'_1 \wedge C'_2\}}$$

Similarly we have a variant of $[\vee-Pre]$. We can easily observe $[\wedge-Post]$ and $[\wedge-Pre/Post]$ are mutually derivable through the use of $[Consequence]$ (and similarly for the other pair). For deriving $[\wedge-Pre/Post]$ from $[\wedge-Post]$, we first strengthen the preconditions of both judgements in the premise of $[\wedge-Pre/Post]$ using $[Consequence]$, to have $C_1 \wedge C_2$ as the common precondition. Then we apply $[\wedge-Post]$. For the converse derivation, we set $C_1 = C_2 = C$ in $[\wedge-Pre/Post]$ and apply $[Consequence]$ using $C \wedge C \supset C$.

$[Aux_{\exists}]$ and $[Aux_{\forall}]$ (where i is auxiliary by Convention 4.1) are easily justifiable from the semantics of auxiliary variables. They may also be considered as the infinitary counterparts of $[\vee-Pre]$ and $[\wedge-Post]$, respectively, because, by the lack of i in the post-condition in $[Aux_{\exists}]$ (resp. in the pre-condition in $[Aux_{\forall}]$), we have virtually an infinite family of i -indexed predicates in the pre-condition (resp. in the post-condition), which are combined by infinitary disjunction (resp. infinitary conjunction) in the conclusion. We show below one of the typical examples where $[Aux_{\forall}]$ is useful.

$$\begin{array}{l} 1. \{!x+!y = i\} !x = !x+!y; y = !x \{!x+!y = 2 \times i\} \quad \text{Premise} \\ 2. \{T\} \lambda(). (!x = !x+!y; y = !x) \{ \{!x+!y = i\}u \bullet () \{!x+!y = 2 \times i\} \} \quad (Abs) \\ 3. \{T\} \lambda(). (!x = !x+!y; y = !x) \{ \forall i. \{!x+!y = i\}u \bullet () \{!x+!y = 2 \times i\} \} \quad (Aux_{\forall}) \end{array}$$

Note i should be left free before we close the behaviour by abstraction: once everything is on the right-hand side, we can use $[Aux_{\forall}]$ to universally abstract the variable, to obtain a closed specification.

$[Aux_{inst}]$ instantiates an (implicitly universally quantified) auxiliary variable into a concrete value with base type (i.e. either Unit, Nat or Bool), both in the pre-condition and the post-condition; while $[Aux_{abst}]$ does the converse. Validity of both rules is immediate. These rules are simple in character, but are sometimes useful for economical reasoning.

$[Consequence-Aux]$ is the rule which was originally introduced for the standard Hoare Logic by Kleymann [41], capturing elegantly the semantics of auxiliary names. $[Consequence]$ (of Figure 5), $[Aux_{\exists}]$ and $[Aux_{\forall}]$ are all special cases of this rule. We note that, in the present setting, many examples can be reasoned naturally using the main rules in Figure 5.

5 Models and Soundness

5.1 Observational Equality

This section gives a concise summary of models for assertions and judgements, and establishes the soundness of proof rules, including axioms stated in Section 3. The models are operational in nature: this has the merit of immediate faithfulness to programs' behaviours (e.g. a predicate claiming the existence of unrealisable functions is unsatisfiable); extensibility (e.g. polymorphisms); and conciseness. Different models are possible, using, for example, CPOs. [33] constructs a more uniform universe of models from typed processes.

We start by introducing a typed congruence for programs [61], which are used for defining an abstract notion of behaviour for the semantics of the logic. A *typed congruence* is an equivalence relation on typed terms with identical bases and types, closed under the compatibility rules corresponding to the typing rules. For reference, we give the compatibility rules in Figure 10 in Appendix A.1. 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 5.1 (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 satisfying, for each semi-closed $\Delta \vdash M_{1,2} : \text{Unit}$ and for each σ such that $\Delta \vdash \sigma$, we have $(M_1, \sigma) \Downarrow$ iff $(M_2, \sigma) \Downarrow$.

As before, when one or both of the bases are empty, we write e.g. $\Gamma \vdash M \cong N : \alpha$, $\Delta \vdash M \cong N : \alpha$, $\vdash M \cong N : \alpha$, etc. We offer a simple example of the effect of reference bases on the typed congruence.

Convention 5.2 *Below and henceforth we let ω^α stand for a(ny) diverging closed term of type α : e.g. we can take $\omega^\alpha \stackrel{\text{def}}{=} (\mu x^{\alpha \Rightarrow \alpha}. \lambda y. xy)V$ 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 5.3 Let $M \stackrel{\text{def}}{=} \lambda y^\alpha. \text{let } z = y() \text{ in } 3$ and $N \stackrel{\text{def}}{=} \lambda y^\alpha. \text{let } z = y() \text{ in let } z' = y() \text{ in } 3$ with $\alpha = \text{Unit} \Rightarrow \text{Unit}$. Then we have $\vdash M \cong N : \alpha \Rightarrow \text{Nat}$, but $x : \text{Ref}(\text{Nat}) \vdash M \not\cong N : \alpha \Rightarrow \text{Nat}$. To check the latter, take $C[\cdot] \stackrel{\text{def}}{=} ([\cdot]L); \text{if } !x = 1 \text{ then } () \text{ else } \omega$ 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 5.4 (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 $\Delta \vdash M_{1,2} : \text{Unit}$ 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 .

5.2 Models

Definition 5.5 (models) A *model of type* $\Gamma; \Delta$ is a pair (ξ, σ) such that ξ is a finite map from $\text{dom}(\Gamma)$ to semi-closed values such that each $x \in \text{dom}(\Gamma)$ is mapped to a semi-closed value V such that $\Delta \vdash V : \Gamma(x)$; and σ is a finite map from $\text{dom}(\Delta)$ to semi-closed values such that each $x \in \text{dom}(\Delta)$ is mapped to a semi-closed value $\Delta \vdash V : \alpha$ with $\Delta(x) = \text{Ref}(\alpha)$. We let \mathcal{M}, \dots range over models, and write $\mathcal{M}^{\Gamma; \Delta}$ to indicate that $\Gamma; \Delta \vdash \mathcal{M}$.

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 5.6**
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 $x : V \in \mathcal{M}$ when $\Delta(x) = \text{Ref}(\alpha)$ 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 $x \in \text{dom}(\xi)$. 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 $x \notin \text{dom}(\xi \cup \sigma)$. Similarly we define $\mathcal{M} \cdot [x \mapsto V]$, $\xi \cdot x : V$ and $\sigma \cdot [x \mapsto V]$.
 3. $\xi \setminus \vec{x}$ indicates the result of taking off the submap of ξ whose domain is $\{\vec{x}\}$. Similarly we write $\mathcal{M} \setminus \vec{x}$ for the result of taking off \vec{x} -elements from the components of \mathcal{M} .

Definition 5.7 (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 $\llbracket e \rrbracket_{\mathcal{M}}$, is given by the following clauses.

$$\begin{array}{ll}
\llbracket x^\alpha \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} V & (x^\alpha : V \in \mathcal{M}) & \llbracket \text{op}(\vec{e}) \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} \text{op}(\llbracket \vec{e} \rrbracket_{\mathcal{M}}) \\
\llbracket !x^{\text{Ref}(\alpha)} \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} V & (x^{\text{Ref}(\alpha)} \mapsto V \in \mathcal{M}) & \llbracket \langle e, e' \rangle \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} \langle \llbracket e \rrbracket_{\mathcal{M}}, \llbracket e' \rrbracket_{\mathcal{M}} \rangle \\
\llbracket x^{\text{Ref}(\alpha)} \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} x & & \llbracket \pi_i(e) \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} \pi_i(\llbracket e \rrbracket_{\mathcal{M}}) \\
\llbracket c \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} c & & \llbracket \text{in}_i(e) \rrbracket_{\mathcal{M}} \stackrel{\text{def}}{=} \text{in}_i(\llbracket e \rrbracket_{\mathcal{M}})
\end{array}$$

Here writing e.g. $\text{op}(\llbracket \vec{e} \rrbracket_{\mathcal{M}})$ is an abbreviation for the unique (up to α -equivalence) V such that $\text{op}(\llbracket \vec{e} \rrbracket_{\mathcal{M}}) \Downarrow V$. By construction of models, all terms are interpreted as semi-closed values *except* a reference name. 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 5.8 (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, assuming that $\mathcal{M} = (\xi, \sigma)$.

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

where, in each clause, a logical connective/quantifier on the right-hand side (if any) is used semantically.

The satisfaction of an invocation formula, $\mathcal{M}^{\Gamma; \Delta} \models \{C\}e_1 \bullet e_2 = x\{C'\}$, means that:

If the interpretation of e_1 is invoked with that of e_2 as an argument, and if an initial state at Δ satisfies C , then it will converge with a value named x , where x and the final state at Δ together satisfy C' .

Note in the satisfaction, we take any state at Δ satisfying C , even if it contradicts the current state as given in \mathcal{M} . This is because we need to describe the behaviour of invocation under hypothetical states; for example, the specification of the behaviour of $\lambda().x := !x + 1$ means that it increments a value stored in x in an arbitrary state of x .

We are now ready to formalise the semantics of judgements. Below $M\xi$ denotes the substitution of values following ξ .

Definition 5.9 (semantics of judgement) $\models \{C\}M :_u \{C'\}$ iff $(\xi, \sigma) \models C$ implies $(M\xi, \sigma) \Downarrow (V, \sigma')$ and $(\xi \cdot u : V, \sigma') \models C'$ for each well-typed (ξ, σ) .

The description of programs by valid judgements is extensional, in the following sense.

Proposition 5.10 *If $\models \{C\}M :_u \{C'\}$ and $\Gamma; \Delta \vdash M \cong N : \alpha$, then $\models \{C\}N :_u \{C'\}$.*

The proof is immediate from the definition (formally by Proposition A.1). We use the following notations in the rest of the paper. We also assume, as before, that all judgement, terms, models, etc. are implicitly well-typed. Basic properties of the satisfaction relation follow.

Lemma 5.11 *Let $\Gamma; \Delta \vdash C$ and assume given $\Gamma; \Delta \vdash \mathcal{M}$,*

1. (renaming) *For each injective renaming ϕ , $\mathcal{M} \models C$ iff $\mathcal{M}\phi \models C\phi$.*
2. (thinning) *If $x \notin \text{dom}(\Gamma, \Delta)$, then $\mathcal{M} \cdot x : V \models C$ implies $\mathcal{M} \models C$.*
3. (weakening) *If $x \notin \text{dom}(\Gamma, \Delta)$, then $\mathcal{M} \models C$ implies $\mathcal{M} \cdot x : V \models C$.*

Proof Each using the corresponding clauses in Proposition A.1 (1–3). □

Many of the proof rules in Figure 5 use substitutions. The following is their standard properties. Below if $\mathcal{M} = (\xi, \sigma)$, we write $\text{dom}(\mathcal{M})$ for $\text{dom}(\xi) \cup \text{dom}(\sigma)$. For (1)-a/b, we only need these special cases (in comparison with the standard ones) for our technical development. Below let \mathcal{M} be arbitrarily chosen (except assuming well-typedness).

Lemma 5.12 (substitution) *Let $\mathcal{M} = (\xi, \sigma)$.*

1. (a) *If $x \notin \text{dom}(\mathcal{M}) \cup \text{fv}(e)$, then $\mathcal{M} \models C[e^\alpha/x^\alpha]$ iff $\mathcal{M} \cdot x : (\llbracket e \rrbracket_{\mathcal{M}}) \models C$.*
 (b) *If $x \in \text{dom}(\xi) \setminus \text{fv}(e)$, then $\mathcal{M} \models C[e^\alpha/x^\alpha]$ iff $\mathcal{M}[x : (\llbracket e \rrbracket_{\mathcal{M}})] \models C$.*
2. *If $x \in \text{dom}(\sigma)$, then $\mathcal{M} \models C[e^\alpha/!x^{\text{Ref}(\alpha)}]$ iff $\mathcal{M}[x \mapsto (\llbracket e \rrbracket_{\mathcal{M}})] \models C$.*

Proof While the proof of (1-a/b) is essentially a special case of that of (2), we give direct proofs. For (1-a), if x is not in C the result is vacuous. Suppose $x \in \text{fv}(C)$.

$$\begin{aligned} \mathcal{M} \models C &\equiv \mathcal{M} \cdot x : \llbracket e \rrbracket_{\mathcal{M}} \models C \wedge x = e && \text{(Lem 5.11-3)} \\ &\equiv \mathcal{M} \cdot x : \llbracket e \rrbracket_{\mathcal{M}} \models C[e/x] \wedge x = e && \text{(substitutivity)} \\ &\equiv \mathcal{M} \cdot x : \llbracket e \rrbracket_{\mathcal{M}} \models C[e/x] && (\mathcal{M} \cdot x : \llbracket e \rrbracket_{\mathcal{M}} \models x = e \text{ always}) \end{aligned}$$

The reasoning for (1-b) is identical, replacing \mathcal{M} above with $\mathcal{M} \setminus x$. (2) is by induction on C , starting from an equation. For the base case we note:

$$\llbracket e'[e/!x] \rrbracket_{\mathcal{M}} = \llbracket e' \rrbracket_{\mathcal{M}[x \mapsto \llbracket e \rrbracket_{\mathcal{M}}]} \quad (5.1)$$

which is immediate. Thus we infer, writing σ for $[e/!x]$:

$$\begin{aligned} \mathcal{M} \models (e_1 = e_2)\sigma &\equiv \llbracket e_1\sigma \rrbracket_{\mathcal{M}} = \llbracket e_2\sigma \rrbracket_{\mathcal{M}} && \text{(Def. 5.8)} \\ &\equiv \llbracket e_1 \rrbracket_{\mathcal{M}[x \mapsto \llbracket e \rrbracket_{\mathcal{M}}]} = \llbracket e_2 \rrbracket_{\mathcal{M}[x \mapsto \llbracket e \rrbracket_{\mathcal{M}}]} && (5.1) \\ &\equiv \mathcal{M}[x \mapsto \llbracket e \rrbracket_{\mathcal{M}}] \models e_1 = e_2 && \text{(Def. 5.8)} \end{aligned}$$

The inductive cases are immediate (for negation we use the logical equivalence). \square

The main results of this paper follow. The first result says we can safely use the axioms in Figures 3 and 4, Section 3.5, Page 20, as the basis of reasoning.

Proposition 5.13 (soundness of axioms) *All axioms in Figures 3 and 4 are true under arbitrary (well-typed) models.*

Proof See Appendix A.2. \square

The second one says that we can safely use the proof rules in Figures 5 and 6, or other derivable rules from them, for deriving properties of programs.

Theorem 5.14 (soundness of proof rules) *If $\vdash \{C\} M :_u \{C'\}$ by the proof rules in Figures 5 and 6, then $\models \{C\} M :_u \{C'\}$.*

Proof See Appendix A.3. \square

6 Observational Completeness

6.1 Observability and Program Logics

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 (local) specifications to be interchangeable without affecting the observable behaviour of the whole, up to a required (global) specification. For this to be truly materialised, it is essential that valid assertions for programs capture precisely the contextual behaviour of programs [26, 50, 51]. 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 (i) $u : M \models A$ and (ii) $u : N \models A$ implies $M \cong N$?

Clearly (2) entails (1) when models are given up to the observational congruence (as they are now, cf. Section 5: and if not, there is no possibility (1) generally holds). The properties can be easily refined for general programs. 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. In the following, we establish (1) for the proposed logic. For (2), see Section 8. For establishing (1), we proceed as follows.

Step 1: We introduce a variant of *finite canonical forms* (FCFs) [6, 34, 37] which represent a limited class of behaviours.

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 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 validity.

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

Convention 6.1 *Throughout the present section we only consider Nat, arrow types and induced reference types. Accordingly the “if” construct branches on zero or non-zero, and assignment has the shape $x := M ; N$.*

6.2 Assertions for Total Correctness and Characteristic Formulae

For our present purpose, we only need to use the following class of assertions. Below \sqsubseteq is the partial order induced by \sqsubseteq on programs (cf. Definition 5.4).

Definition 6.2 (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 $(\xi \cdot u : V', \sigma) \models C$.

Intuitively, total correctness is a property which is closed upwards — if a program M satisfies it and there is a more defined program N then N also satisfies it.⁵ At the end of the present section we discuss the status of total correctness in this sense in the present logic. Characteristic formulae in the present logic are defined as:

Definition 6.3 (characteristic formulae) Given $\Delta \vdash V : \alpha$, a TCA C at u *characterises* V iff: (1) $\models \{\top\}V^{\Delta;\alpha} :_u \{C\}$ and (2) $\models \{\top\}W^{\Delta;\alpha} :_u \{C\}$ implies $\Delta \vdash V \sqsubseteq W : \alpha$.

In the technical development later, we need to consider characteristic formulae to open programs, extending Definition 6.3.

Definition 6.4 (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 .

6.3 Finite Canonical Forms

If $(C[M_1], \sigma)$ converges and $(C[M_2], \sigma)$ diverges, the convergence uses only finite part of $C[\cdot]$ and σ , since 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 that they have barely the 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* [34] (cf. [6, 37]). Below we extend the construction in [34] for the imperative PCFv.

Finite canonical forms (FCFs), ranged over by F, F', \dots , 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', \dots range over FCFs which are values.

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

Above ω is a diverging closed term, cf. Convention 5.2. In the case, X is a finite non-empty subset of natural numbers (it diverges for other values); and in $\text{let } x = yU \text{ in } F$ (resp. $\text{case } x \text{ of } \langle n_i : F_i \rangle_i$), y (resp. x) should not be free in U (resp. F_i). FCFs are typed following their translation to imperative PCFv-terms, cf. [34].

Lemma 6.5 *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 $(i, j) = (2, 1)$:*

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

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

Proof See Appendix B.1. □

⁵ There are assertions which describe partial correctness rather than total correctness. For example, $\forall x.(u \bullet x = x! \vee u \bullet x \Uparrow)$ is a partial correctness assertion for a factorial.

Fig. 7 Derivation Rules for CAPs of FCFs (functional sublanguage)

$$\begin{array}{c}
\frac{-}{\{T\} n :_u \{u = n\}} \quad \frac{\{A_i\} F_i :_u \{B_i\}}{\{\forall_i(x = n_i \wedge A_i)\} \text{ case } x \text{ of } \langle n_i : F_i \rangle_i :_u \{\forall_i(x = n_i \wedge B_i)\}} \\
\frac{-}{\{F\} \omega :_u \{F\}} \quad \frac{\{A\} F :_m \{B\}}{\{T\} \lambda x.F :_u \{\forall x.\{A\}u \bullet x = m\{B\}\}} \\
\frac{\{T\} U :_z \{A_0\} \quad \{A_1\} F :_u \{B\} \quad j \text{ fresh}}{\{\forall z.\{A_0\}f \bullet z = x\{A_1 \wedge x = j\}\} \text{ let } x = fU \text{ in } F :_u \{B[j/x]\}}
\end{array}$$

6.4 Characteristic Formulae for FCFs (1): functional sublanguage

In this subsection and the next, we introduce derivation rules by which we can derive characteristic formulae (CAPs) of FCFs. For clearer and incremental presentation, this subsection focusses on FCFs of the functional sublanguage (i.e. we take off assignment, dereference and reference types/bases). Accordingly, the logical language does not use dereference, and models lose their store part. The key ideas are more simply illustrated using the functional sublanguage: the addition to its imperative extension is incremental. In Figure 7, we present the derivation rules for CAPs for the FCFs for the functional sublanguage. We write:

$$\vdash_{\text{char}} \{A\}F :_u \{B\}$$

if $\{A\}F :_u \{B\}$ is derivable from these rules. In each rule we assume: (1) each premise should be derived in this system, not in the proof rules in Section 4; (2) programs, assertions, etc. are appropriately typed; and (3) newly introduced names are always fresh. Below we informally illustrate each rule in Figure 7 one by one.

1. A CAP of n at u is $(T, u = n)$, saying: whenever a program, say M , satisfies $\{T\}M :_u \{u = n\}$, M may as well be equal to n . For example, under $x : \text{Nat}$, `if x then n else n` has this property, and it is indeed \cong -equal to n .
2. 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 $x = n_i$ and A_i . For each i -th case, it can guarantee what F_i guarantees.
3. 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.
4. A CAP of $\lambda x.F$ is made directly from a CAP for F . Since (A, B) is a CAP of F at m , and because $\lambda x.F$ has already terminated, we need no precondition for convergence, and may as well set $\forall x.\{A\}u \bullet x = m\{B\}$ as the desired CAP.
5. 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_0) is a CAP for U . We further assume (A_1, B) is a CAP of F . Now for `let $x = fU$ in F` to terminate and to

guarantee the x -portion of A_1 (which is $\exists \vec{y}.A_1$), we should assume:

$$\forall z. \{A_0\} f \bullet z = x \{A_1\}$$

The evaluation formula says that any argument satisfying the characteristic formula for U at z can be fed to f and result in a value satisfying (the x -portion of) A_1 , adding the constraint to f so that, when it is invoked, it does terminate. Note A_1 may assert on primary names other than x , which is necessary for F after the let to terminate. Under any model we consider, the formula is equivalent to:

$$\forall z. \exists x'. (\{A_0\} f \bullet z = x' \{x' = x\} \wedge A_1)$$

This is because A_0 is always satisfiable (with a witness U). We can also check the resulting assertions are still a TCA pair (in the sense of Convention 6.1) since a free variable of U , which are negatively used primary names in A , is dualised in the formula, making it a TCA at these names.

Example 6.6 (derivations of characteristic assertions for FCFs)

1. Recall $\Omega^{\alpha \Rightarrow \beta} \stackrel{\text{def}}{=} \lambda x^\alpha \omega^\beta$ from Convention 5.2. We infer:

$$\frac{1. \{F\} \omega :_m \{F\}}{2. \{T\} \Omega :_u \{\forall x. \{F\} u \bullet x = m\{F\}\}}$$

Using axioms for evaluation formulae in Figure 4 (page 21), we calculate:

$$\begin{aligned} \forall x. \{F\} u \bullet x = m\{F\} &\Leftrightarrow \forall x. \{F \wedge F\} u \bullet x = m\{F\} & (\text{e7}) \\ &\Leftrightarrow \forall x. (F \supset \{F\} u \bullet x = m\{F\}) & (\text{e5}) \end{aligned}$$

Thus, up to logical equivalence, the derivation says (T, T) is a CAP of Ω .

2. Let $F \stackrel{\text{def}}{=} \text{let } x = f3 \text{ in case } x \text{ of } \langle 1 : 1, 2 : 2 \rangle$. We derive:

$$\frac{\frac{1. \{T\} 1 :_u \{u = 1\}}{2. \{T\} 2 :_u \{u = 2\}}}{3. \{x = j = 1 \vee x = j = 2\} \text{ case } x \text{ of } \langle i : i \rangle_{i=1,2} :_u \{u = j = 1 \vee u = j = 2\}}}{4. \{T\} 3 :_z \{z = 3\}}}{5. \{\forall z. \{z = 3\} f \bullet z = x \{x = 1 \vee x = 2 \wedge x = j\}\} F :_u \{u = j = 1 \vee u = j = 2\}}$$

The final judgement captures the behaviour of F by saying:

Relying on the assumption that f returns 1 or 2 when fed with 3, we can guarantee that F will evaluate to that very value f returns as its own result.

Note how the condition on f guarantees the termination of the let construct.

We are now ready to prove the main result of this subsection. The conditions for inductive arguments to go through are made slightly stronger than CAP.

Proposition 6.7 If $\vdash_{\text{char}} \{A\} F :_u \{B\}$, (A, B) satisfies the following three conditions.

1. (soundness) $\models \{A\} F :_u \{B\}$ with B being a TA at u .
2. (MTC, minimal terminating condition) $F\xi \Downarrow$ if and only if $\xi \models A$.
3. (closure) If $\models \{E\}M :_u \{B\}$ such that $E \supset A$, then $F\xi \sqsubseteq M\xi$ for each $\xi \models E$.

Below and henceforth (IH) stands for “induction hypothesis”.

Proof (Soundness) is mechanical by checking each derivation rule is justifiable w.r.t. the proof rules in Section 4 and that it produces a TA. Below we show (MTC) and (closure). Throughout the proofs:

- We let ξ' stand for an appropriately typed model for the derived precondition.
- We treat environments as their corresponding substitutions, writing e.g. $F\xi$.

(Numeral) MTC of \top is immediate since n is a value. For closure, we reason:

$$\xi' \cdot u : M\xi' \models u = n \quad \Rightarrow \quad M\xi' \cong n \quad \Rightarrow \quad n \sqsubseteq M\xi'$$

(Case-n) Let, for brevity, $F' \stackrel{\text{def}}{=} \text{case } x \text{ of } \langle n_i : F_i \rangle_i$, $A \stackrel{\text{def}}{=} \bigvee_i (x = n_i \wedge A_i)$ and $B \stackrel{\text{def}}{=} \bigvee_i (x = n_i \wedge B_i)$. By (IH), assume (A_i, B_i) satisfies (MTC) and (closure) w.r.t. F_i at u , for each i . We show (A, B) do the same w.r.t. F' at u . Let ξ' be a model for the assumed basis and $\xi \stackrel{\text{def}}{=} \xi' / x$ (taking off x -component). For MTC, we reason:

$$\begin{aligned} F'\xi' \Downarrow &\Leftrightarrow \bigvee_i (\xi'(x) = n_i \wedge F_i\xi \Downarrow) \\ &\Leftrightarrow \bigvee_i (\xi(x) = n_i \wedge \xi \models A_i) \quad \Leftrightarrow \quad \xi \models A. \end{aligned}$$

For (closure), let $E \supset A$ and $\models \{E\}M :_u \{B\}$, hence $\models \{E \wedge x = n_i\}M :_u \{B_i\}$. Note that $A \wedge x = n_i \equiv A_i \wedge x = n_i$, hence $E \wedge x = n_i \supset A_i$.

$$\begin{aligned} \xi' \models E &\Rightarrow \exists i. (\xi' \models E \wedge x = n_i) \\ &\Rightarrow \exists i. (\xi(x) = n_i \wedge F_i\xi \sqsubseteq M\xi) \quad (\text{IH}) \\ &\Rightarrow F'\xi' \cong F_i\xi \sqsubseteq M\xi. \end{aligned}$$

(Omega) F is an MTC for ω since ω never terminates under any substitution. It satisfies (closure) since ω is always the minimum element of a given type.

(Abstraction) Let $F' \stackrel{\text{def}}{=} \lambda x. F$. Since F' is a value, \top is an MTC. For (closure), let $A' \stackrel{\text{def}}{=} \forall x. \{A\}u \bullet x = m\{B\}$ and assume $\{E\}M :_u \{A'\}$.

$$\begin{aligned} \xi' \cdot u : M\xi' \models A' &\Rightarrow \forall W. (\xi' \cdot x : W \models A \Rightarrow \xi' \cdot m : (M\xi')W \models B) \\ &\Rightarrow \forall W. (\xi' \cdot x : W \models A \Rightarrow F(\xi' \cdot x : W) \sqsubseteq (M\xi')W) \\ &\Rightarrow F'\xi' \sqsubseteq M\xi'. \end{aligned}$$

In the last line we can cancel W because, noting A is an MTC for F by (IH):

$$\xi' \cdot x : W \not\models A \quad \Rightarrow \quad (F'\xi')W \longrightarrow F(\xi' \cdot x : W) \uparrow \quad \Rightarrow \quad \forall N. (F'\xi')W \sqsubseteq N.$$

(Let-Application) Let

$$F' \stackrel{\text{def}}{=} \text{let } x = fU \text{ in } F \quad A \stackrel{\text{def}}{=} \forall z. \{A_0\}f \bullet z = h\{A_1\}$$

By induction hypothesis we assume:

- (IH1) (A_1, B) satisfies (MTC) and (closure) w.r.t. F at u .
 (IH2) (T, A_0) satisfies (MTC) and (closure) w.r.t. U at z .

Further w.l.o.g. we assume the auxiliary names of A_0 are empty (by universal closure).

First we show A is an MTC for F' . Let $\xi = \xi'/f$ and $\xi'(f) = W$. By (IH2) we have $\xi \models \{T\}U :_z \{A\}$, that is:

$$\xi \cdot z : U\xi \models A \quad (6.1)$$

We now reason:

$$\begin{aligned} F'\xi' \Downarrow & \\ \Leftrightarrow \exists S. (WU\xi \Downarrow S \wedge \xi' \cdot x : S \models A_1) & \quad (\text{IH1, (6.1)}) \\ \Leftrightarrow \xi' \cdot z : U\xi' \models \{T\}f \bullet z = x\{A_1\} & \quad (\text{by definition}) \\ \Leftrightarrow \forall U'. (U' \sqsupseteq U\xi_0 \supset \xi \cdot z : U' \models \{T\}f \bullet z = x\{A_1\}) & \quad (A_0 \text{ TCA at } z) \\ \Leftrightarrow \forall U'. (z : U' \cdot \xi_0 \models A_0 \supset z : U' \cdot \xi' \models \{T\}f \bullet z = x\{A_1\}) & \quad (\text{IH2}) \\ \Leftrightarrow \xi' \models \{A_0\}f \bullet z = x\{A_1\} & \quad (\text{e5 in §3.5}) \end{aligned}$$

Second we show (closure). Fix E such that $E \supset A$. Assume:

$$\xi' \models E \quad (6.2)$$

as well as:

$$\{E\}M :_u \{B\} \quad (6.3)$$

Let $\xi'(f) = W$. From (6.2) and (IH2), we know WU converges and that:

$$\xi' \cdot x : WU \models E \wedge A_1 \quad (6.4)$$

From (6.3) (noting f does not occur in B) we have:

$$\{E \wedge A_1\}M_u \{B\}. \quad (6.5)$$

By (6.4), (6.5) and (IH1), we obtain $F(\xi' \cdot x : WU) \sqsubseteq M\xi'$. That is: $F'\xi' \sqsubseteq M\xi'$, as required. We have now exhausted all cases. \square

6.5 Characteristic Formulae for FCFs (2): the imperative extension

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 8 presents the derivation rules, using stateful formulae. We observe:

1. The rules for numeral, case, ω and abstraction are identical with those in Figure 7.
2. The let rule refines the corresponding rule in Figure 7 by requiring that the termination guarantee for F is done 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. 8 Derivation Rules for Characteristic Assertions of FCFs

$$\begin{array}{c}
 \frac{}{\{T\} n :_u \{u = n\}} \quad \frac{\{C_i\} F_i :_u \{C'_i\}}{\{\forall_i(x = n_i \wedge C_i)\} \text{ case } x \text{ of } \langle n_i : F_i \rangle_i :_u \{\forall_i(x = n_i \wedge C'_i)\}} \\
 \frac{}{\{F\} \omega :_u \{F\}} \quad \frac{\{C\} F^{\Gamma, x: \alpha; \Delta; \beta} :_m \{C'\}}{\{T\} \lambda x. F :_u \{\forall x. \{C\} u \bullet x = m \{C'\}\}} \\
 \frac{\{T\} U :_z \{A\} \quad \{C\} F :_u \{C'\} \quad j \text{ fresh}}{\{\forall \vec{j}. (!\vec{r} = \vec{j} \supset \forall z. \{A \wedge !\vec{r} = \vec{j}\} f \bullet z = x \{C \wedge x = j\})\} \text{ let } x = yU \text{ in } F :_u \{C'[j/x]\}} \\
 \frac{\{C\} F :_u \{C'\}}{\{C[!x/y]\} \text{ let } y = !x \text{ in } F :_u \{C'\}} \\
 \frac{\{T\} U^{\Delta; \alpha} :_z \{A\} \quad \{C\} F :_u \{C'\} \quad \text{fv}(A) \subset \{z\} \cup \text{dom}(\Delta)}{\{\forall z. (A \supset C[z/!x])\} x := U ; F :_u \{C'\}} \\
 \frac{\{T\} U^{\Gamma; \Delta; \alpha} :_u \{A\} \quad \text{dom}(\Delta) = \vec{r}}{\{!\vec{r} = \vec{i}\} U^{\Gamma; \Delta; \alpha} :_u \{A \wedge !\vec{r} = \vec{i}\}}
 \end{array}$$

3. 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 $\{T\}U :_z \{B\}$ 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).
4. The rule for dereference starts from a CAP (C, C') for F , and adjoin 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).
5. 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, which does not lose 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 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 on the precondition may be obtained by realising that, while not explicitly present, the consequence of $\models \{T\}U :_z \{A\}$ (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 in fact have $\exists z.(A \wedge C[z/!x])$. Note its second conjunct stipulates the original precondition 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 $\vdash_{\text{char}} \{C\}F :_u \{C'\}$ when $\{C\}F :_u \{C'\}$ is derivable from the rules in Figure 8 except, when F is a value, we take the result of applying the weakening. We now observe (with $\sigma' \sqsubseteq \sigma'_0$ denoting the pointwise extension of \sqsubseteq):

Proposition 6.8 *If $\vdash_{\text{char}} \{C\}F :_u \{C'\}$, then (C, C') satisfies the following conditions.*

1. (*soundness*) $\models \{C\}F :_u \{C'\}$ with B being a TA 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 The arguments closely follow those for Proposition 6.7, see Appendix B.2. \square

6.6 Observational Completeness

We conclude this section with the establishment of observational completeness. We first define the standard logical equivalence, cf. [26].

Definition 6.9 (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 we do not restrict the class of formulae to TCAs. The main result of this section follows.

Theorem 6.10 *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 6.5, there exist semi-closed FCF values F and \vec{U} such that, say,

$$(FM_1, \vec{r} \mapsto \vec{U}) \Downarrow \quad \text{and} \quad (FM_2, \vec{r} \mapsto \vec{U}) \Uparrow. \quad (6.6)$$

By Proposition 6.8, there are assertions which characterise F and \vec{U} (in the sense of Definition 6.3). Let the characteristic formula for F at f be written $\llbracket F \rrbracket(f)$. We now

reason:

$$\begin{aligned}
(FM_1, \vec{r} \mapsto \vec{U}) \Downarrow & \\
\Rightarrow (f : [F] \cdot m : [M_1] \models \{\wedge_i \llbracket U_i \rrbracket (!r_i)\} f \bullet m = z \{T\}) & \\
\Rightarrow \forall V. (f : V \models \llbracket [F] \rrbracket_f \supset (f : V \cdot m : [M_1] \models \{\wedge_i \llbracket U_i \rrbracket !r_i\} f \bullet m = z \{T\})) & \\
\Rightarrow \models \{T\} M_1 :_m \{\forall f. \{\llbracket [F] \rrbracket (f) \wedge (\wedge_i \llbracket U_i \rrbracket (!r_i))\} f \bullet m = z \{T\}\} &
\end{aligned}$$

But by (6.6) we have

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

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 6.10 which says that the strongest post condition always gives a CAP for a semi-closed value, after a definition.

Definition 6.11 *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 $\text{sp}(C, M, u)$, 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 6.12 *Let $A \in \text{sp}(T, V^{\Delta; \alpha}, u)$. 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 $W \not\sqsubseteq V$ but $\{T\}W :_u \{A\}$. By Theorem 6.10, there is B such that $\models \{T\}V :_u \{B\}$ and $\not\models \{T\}W :_u \{B\}$. By assumption we have $A \supset B$. Hence $\{T\}W :_u \{B\}$, a contradiction. \square

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

7 Reasoning Examples

7.1 Deriving Hoare Logic for Total Correctness

We first embed the standard proof rules of Hoare logic for total correctness with recursive procedures [41] in the present logic, thus establishing a precise link between the proposed logic and traditional Hoare logics for total correctness. Then we show the generalisation of these rules at the end.

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

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

In $\text{proc } p = P \text{ in } Q$, a procedure body P is named p , where we allow calls to p to occur in P . The reduction rules are standard [73] 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 PCF. In correspondence with the logic for imperative PCFv, we consider a program logic for this language which guarantees total correctness. Assertions, still ranged over by C, C', \dots , a proper subset of those of Section 3, having only natural numbers and references to them as data types and missing evaluation formulae are given below. Let $\star \in \{\wedge, \vee, \supset\}$ and $Q \in \{\forall, \exists\}$.

$$\begin{aligned} e ::= & \quad i^{\text{Nat}} \mid !x \mid n \mid \text{op}(e_1, \dots, e_n) \\ C ::= & \quad e_1 = e_2 \mid C_1 \star C_2 \mid \neg C \mid Q i^{\text{Nat}}. C \end{aligned}$$

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

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.

Figure 9 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 list Kleymann's strengthened consequence rule [41] (discussed in Section 4.2), 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}).

Fig. 9 Hoare Logic with Recursive Procedure (total correctness)

$$\begin{array}{c}
\frac{}{[Skip] \Sigma \vdash \{C\} \text{skip} \{C\}} \quad \frac{}{[AssignH] \Sigma \vdash \{C[e/!x]\} x := e \{C\}} \quad \frac{\Sigma \vdash \{C\} P \{C_0\} \quad \Sigma \vdash \{C_0\} Q \{C'\}}{\Sigma \vdash \{C\} P; Q \{C'\}} [Seq] \\
\frac{\Sigma \vdash \{C \wedge e\} P_1 \{C'\} \quad \Sigma \vdash \{C \wedge \neg e\} P_2 \{C'\}}{\Sigma \vdash \{C\} \text{if } e \text{ then } P_1 \text{ else } P_2 \{C'\}} [IfH] \quad \frac{C \wedge e \supset e' \geq 0 \quad \Sigma \vdash \{C \wedge e \wedge e' = i\} P \{C \wedge e' \leq i\}}{\Sigma \vdash \{C\} \text{while } e \text{ do } P \{C \wedge \neg i\}} [While] \\
\frac{\{C\} p \{C'\} \in \Sigma}{\Sigma \vdash \{C\} \text{call } p \{C'\}} [Call] \quad \frac{\Sigma, \{\exists j \leq i. C(j)\} p \{C_0\} \vdash \{C(i)\} P \{C_0\} \quad \Sigma, \{\exists i. C(i)\} p \{C_0\} \vdash \{C\} Q \{C'\}}{\Sigma \vdash \{C\} \text{proc } p = P \text{ in } Q \{C'\}} [RecProc] \\
\frac{\{C_0\} P \{C'_0\} \quad C \supset \exists \vec{j}. (C_0[\vec{j}/\vec{i}] \wedge (C'_0[\vec{y}/\vec{x}][\vec{j}/\vec{i}] \supset C'[\vec{y}/\vec{x}]))}{\{C\} M :_u \{C'\}} [Consequence-Aux]
\end{array}$$

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

$$\begin{array}{l}
\llbracket \text{skip} \rrbracket \stackrel{\text{def}}{=} () \quad \llbracket x := e \rrbracket \stackrel{\text{def}}{=} x := e \quad \llbracket P; Q \rrbracket \stackrel{\text{def}}{=} \llbracket P \rrbracket; \llbracket Q \rrbracket \quad (\stackrel{\text{def}}{=} (\lambda(). Q) P) \\
\llbracket \text{if } e \text{ then } P \text{ else } Q \rrbracket \stackrel{\text{def}}{=} \text{if } e \text{ then } \llbracket P \rrbracket \text{ else } \llbracket Q \rrbracket \\
\llbracket \text{while } e \text{ do } P \rrbracket \stackrel{\text{def}}{=} (\mu w. \lambda(). \text{if } e \text{ then } \llbracket P \rrbracket; (w()) \text{ else } ()) () \\
\llbracket \text{call } p \rrbracket \stackrel{\text{def}}{=} p() \quad \llbracket \text{proc } p = P \text{ in } Q \rrbracket \stackrel{\text{def}}{=} (\lambda p. \llbracket Q \rrbracket)(\mu p. \lambda(). \llbracket P \rrbracket)
\end{array}$$

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

$$\{\llbracket \Sigma \rrbracket \wedge C\} \llbracket P \rrbracket \{C'\} \quad \text{with} \quad \llbracket \emptyset \rrbracket \stackrel{\text{def}}{=} \top \text{ and } \llbracket \Sigma, \{C\} p \{C'\} \rrbracket \stackrel{\text{def}}{=} \llbracket \Sigma \rrbracket \wedge \{C\} p \bullet () \{C'\}$$

Formulae are simply the subset of those for the imperative PCF. If we use the standard model of number theory [49, §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 9 has a clean encoding into the logic for imperative PCF.

For `skip`, observing $u \notin \text{fv}(C)$ and without a premise:

$$1. \quad \{C[() / u] (\stackrel{\text{def}}{=} C)\} \llbracket \text{skip} \rrbracket \{C\} \quad (\text{Const})$$

`[AssignH]` and other rules are embedded using the following rule (easily derivable in the proof rules in Section 4).

$$\frac{}{[Simple] \{C[e/u]\} e :_u \{C\}}$$

`[AssignH]` is cleanly decomposed into `[Simple]` and `[Assign]`.

$$\begin{array}{l}
1. \quad \{C[m/!x][e/m] (\stackrel{\text{def}}{=} C[e/!x])\} e :_m \{C[m/!x][() / u]\} \quad (\text{Simple}) \\
\hline
2. \quad \{C[e/!x]\} x := e :_u \{C\} \quad (\text{Assign})
\end{array}$$

Above by definition u is not in C . The derivation of $[Seq]$, $[IfH]$, $[Call]$ and $[RecProc]$ can be found in Appendix C. For $[While]$, we translate the induction on loop to the induction on recursion. The following rule (immediately derivable from $[App]$) is convenient.

$$[Run] \frac{\{C\} V :_u \{\{C\} u \bullet () \{C'\}\}}{\{C\} V () \{C'\}}$$

Further we let:

$$B(x, i) \stackrel{\text{def}}{=} \{C \wedge e' = i\} x \bullet () \{C \wedge \neg e\} \quad B'(x, i) \stackrel{\text{def}}{=} \forall j \preceq i. B(x, j)$$

The inference follows. Below and henceforth (e3) etc. are axioms from §3.5, indicating their use in the consequence rule and (Conseq) etc. are shorthand for (Consequence) etc. Line 4 below uses $\exists j \preceq i. C \stackrel{\text{def}}{=} \exists j. (j \preceq i \wedge C)$.

1. $\{C \wedge e \wedge e' = n\} \llbracket P \rrbracket \{C \wedge e' \preceq n\}$ (with $C \wedge e \supset e' \succeq 0$)	(premise)
2. $\{B'(w, n) \wedge C \wedge e' \preceq n\} w :_m \{B'(m, n) \wedge C \wedge e' \preceq n\}$	(Var)
3. $\{B'(w, n) \wedge C \wedge e' \preceq n\} w :_m \{B'(m, n)\}$	(Conseq)
4. $\{B'(w, n) \wedge C \wedge e' \preceq n\} w :_m \{\{\exists j \preceq n. (C \wedge e' = j)\} w \bullet () \{C \wedge \neg e\}\}$	(e3)
5. $\{B'(w, n) \wedge C \wedge e' \preceq n\} w :_m \{\{C \wedge e' \preceq n\} w \bullet () \{C \wedge \neg e\}\}$	(e8)
6. $\{B'(w, n) \wedge C \wedge e' \preceq n\} w :_m \{\{B'(w, n) \wedge C \wedge e' \preceq n\} w \bullet () \{B'(w, n) \wedge C \wedge \neg e\}\}$	(e7)
7. $\{B'(w, n) \wedge C\} w () \{C \wedge \neg e\}$	(Run)
8. $\{B'(w, n) \wedge C \wedge e \wedge e' = n\} \llbracket P \rrbracket ; w () \{C \wedge \neg e\}$	(1, 7, Seq)
9. $\{C \wedge \neg e \wedge e' = n\} () \{C \wedge \neg e \wedge e' = n\}$	(Unit)
10. $\{B'(w, n) \wedge C \wedge \neg e \wedge e' = n\} () \{C \wedge \neg e\}$	(Conseq)
11. $\{B'(w, n) \wedge C \wedge e' = n\} \text{if } e \text{ then } \llbracket P \rrbracket ; (w ()) \text{ else } () \{C \wedge \neg e\}$	(8, 10, IfH)
12. $\{B'(w, n)\} \lambda(). \text{if } e \text{ then } \llbracket P \rrbracket ; (w ()) \text{ else } () :_u \{B(u, n)\}$	(Abs)
13. $\{\top\} \mu w. \lambda(). \text{if } e \text{ then } \llbracket P \rrbracket ; (w ()) \text{ else } () :_u \{\forall i. B(u, i)\}$	(Rec)
14. $\{\top\} \mu w. \lambda(). \text{if } e \text{ then } \llbracket P \rrbracket ; (w ()) \text{ else } () :_u \{\{\exists i. (C \wedge e' = i)\} x \bullet () \{C \wedge \neg e\}\}$	(e3)
15. $\{\top\} \mu w. \lambda(). \text{if } e \text{ then } \llbracket P \rrbracket ; (w ()) \text{ else } () :_u \{\{C\} x \bullet () \{C \wedge \neg e\}\}$	(e8)
16. $\{C\} \llbracket \text{while } e \text{ do } P \rrbracket \{C \wedge \neg e\}$	(Run)

The inference above (which may be read from the bottom to the top) shows how naturally the proof rule for total correctness of the while loop arises from the induction principle of $[Rec]$ rule. One may also observe that the derivation immediately extends to the proof rule for the while loop based on well-founded induction, using $[Rec-Wf]$.

Since $[Consequence-Aux]$ in Figure 9 is the precise mirror of $[Consequence-Aux]$ in Figure 5, we can conclude as follows, assuming the standard model of natural numbers for assertions of Hoare logic.

Proposition 7.1 (embedding of Hoare logic for total correctness) $\Sigma \vdash \{C\}P\{C'\}$ implies $\vdash \{[\Sigma] \wedge \{C\}\} [P] \{C'\}$.

The analytical nature of the proposed logic is evident through the preceding embedding result (one may say that this analytical nature comes from the origin of the present program logic, a logic for typed processes [32]). We are thus motivated to explore a further use of the proposed logic for developing and analysing program logics of imperative constructs. Below we briefly outline two such instances.

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 imperative PCFv.

$$[While'] \frac{\{C\}M ;_b \{B^b \wedge C\} \quad C \wedge B[t/b] \supset e' \geq 0 \quad \{C \wedge B[t/b] \wedge e' = n\} N \{C \wedge e' \leq n\}}{\{C\} \text{while } M \text{ do } N \{C \wedge B[f/b]\}}$$

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.

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

A more radical extension is incorporation of higher-order procedures. In this case, it may be easier to use essentially the whole of the proof rules for the imperative PCFv (with, say, block rules and associated variable binding names), with possible distinction between expressions (which can be higher-order) and commands (which are always first-order and never return values). Note we can still use the presented rules for commands (with generalisation such as $[While']$). Thus we do not lose the original reasoning principle while gaining precise compositional reasoning for higher-order expressions.

7.2 Reasoning for Imperative Higher-Order Functions

We illustrate derivations of assertions using simple higher-order imperative programs. We start from first-order imperative commands, using $[AssignH]$ of Figure 9 in § 7.1.

1. $\frac{\{!x \times 2 = 6\} y := !x \times 2 \quad \{!y = 6\}}{\text{(AssignH)}}$
2. $\{!x = 3\} y := !x \times 2 \quad \{!y = 6\} \quad (!x = 3 \supset !x \times 2 = 6, \text{Conseq})$

If we wish to reason about assignment of a higher-order program, we can no longer use $[AssignH]$. Below we let $Double(u) \stackrel{\text{def}}{=} \forall i. (u \bullet i = i \times 2)$.

1. $\frac{\{z = i\} z \times 2 ;_m \{m = i \times 2\}}{\text{(Var, Num, } \times \text{)}}$
2. $\frac{\{T\} \lambda z. (z \times 2) ;_v \{Double(v)\}}{\text{(Abs)}}$
3. $\{T\} x := \lambda z. (z \times 2) \{Double(!x)\} \quad \text{(Assign)}$

where we set $A = \top$, $C_0 = \text{Double}(v)$ and $C = \text{Double}(!x)$ in $[\text{Assignment}]$. Next we infer a function with dereference with type $\text{Nat} \Rightarrow \text{Nat}$ from Example 3.6 (5) as follows.

1. $\frac{\{ \text{Double}(!x) \} !x ;_m \{ \text{Double}(m) \}}{\text{Deref}}$
2. $\frac{\{ y = 3 \} y ;_n \{ n = 3 \}}{\text{Var}}$
3. $\frac{\{ \text{Double}(!x) \wedge y = 3 \} (!x)y ;_u \{ \text{Double}(!x) \wedge u = 6 \}}{\text{App}}$
4. $\frac{\{ \top \} \lambda y. (!x)y ;_u \{ \{ \text{Double}(!x) \} u \bullet 3 = 6 \{ \text{Double}(!x) \} \}}{\text{Abs}}$
5. $\frac{\{ \text{Double}(!x) \} (\lambda y. (!x)y) 3 ;_u \{ u = 6 \wedge \text{Double}(!x) \}}{\text{App}}$

Next we use the following slightly enriched version of $[\text{Seq}]$ of Figure 9 in § 7.1.

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

Using $[\text{Seq}]$, we can plug-in the post and pre-conditions of the conclusions of the last two derivations.

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

By a similar reasoning, we obtain (using C in § 3.2, page 14, for the post-condition):

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

Combining this and $\{ \top \} \lambda y. (!x)y ;_u \{ \{ C \wedge !w = 0 \} u \bullet 3 = 6 \{ C \wedge !w = 1 \} \}$ (which uses D in § 3.2) by $[\text{Seq}]$ 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 reason for M in (2.14), Section 2, Page 10, using $A(fg)$ given there.

1. $\frac{\{ A(fg) \wedge \text{Even}(x) \} y := x \{ A(fg) \wedge \text{Even}(!y) \}}{\text{AssignH, Conseq}}$
2. $\frac{\{ A(fg) \wedge \text{Even}(!y) \} g(f) \{ A(fg) \wedge \text{Odd}(!y) \}}{\text{Var, Var, App}}$
3. $\frac{\{ A(fg) \wedge \text{Odd}(!y) \} g(f) \{ A(fg) \wedge \text{Even}(!y) \}}{\text{Var, Var, App}}$
4. $\frac{\{ A(fg) \wedge \text{Even}(!y) \} !y + 1 ;_z \{ \text{Odd}(z) \wedge \text{Even}(!y) \}}{\text{Simple, Conseq}}$
5. $\frac{\{ A(fg) \} M ;_u \{ \text{Even_then_Odd}(u, x) \}}{\text{Seq', Seq', Seq', Abs}}$

Similarly we can derive $\{ A(fg) \} M ;_u \{ \text{Odd_then_Even}(u, x) \}$, and, through $[\text{Pre-}\wedge]$, finally $\{ A(fg) \} M ;_u \{ \text{Even_then_Odd}(u, x) \wedge \text{Odd_then_Even}(u, x) \}$.

7.3 Three Programming Examples Revisited

First recall `closureFact`

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

Its specification can be given as follows.

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

The derivation that `closureFact` satisfies this specification is a straightforward application of the proof rules we have seen so far. We use the following fact about the factorial:

$$0! = 1 \quad \wedge \quad \forall i^{\text{Nat}}. (i+1)! = (i+1) \times i!. \quad (7.1)$$

Let $A(g, i) \stackrel{\text{def}}{=} \{\mathbb{T}\} g \bullet () = z \{z = i!\}$, $B(f, i) \stackrel{\text{def}}{=} \{\mathbb{T}\} f \bullet i \{ \{\mathbb{T}\} !y \bullet () = z \{z = i!\} \}$, and $B'(f, i) \stackrel{\text{def}}{=} \forall j^{\text{Nat}} \lesssim i. B(f, j)$. 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 **[Add]** in Figure 5.

$\{1 = x!\} 1 ;_m \{m = x!\}$	(Const)
2. $\{B'(f, i) \wedge x = i \wedge x = 0\} 1 ;_m \{m = x!\}$	((7.1), 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, (7.1), 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. $\{B'(f, i) \wedge x = i\} \text{if } x = 0 \text{ then } M_1 \text{ else } M_2 ;_u \{A(!y, x)\}$	(4, 13, If-Hoare)
15. $\{B'(f, i)\} \lambda x^{\text{Nat}}. \text{if } x = 0 \text{ then } M_1 \text{ else } M_2 ;_u$ $\{ \forall x^{\text{Nat}}. \{x = i\} u \bullet i \{ \{\mathbb{T}\} !y \bullet () = z \{z = i!\} \} \}$	(Abs)
16. $\{B'(f, i)\} \lambda x^{\text{Nat}}. \text{if } x = 0 \text{ then } M_1 \text{ else } M_2 ;_u$ $\{ \forall x^{\text{Nat}}. (x = i \supset B(u, x)) \}$	(e5, Conseq)
17. $\{B'(f, i)\} \lambda x^{\text{Nat}}. \text{if } x = 0 \text{ then } M_1 \text{ else } M_2 ;_u \{B(u, i)\}$	(Conseq)
18. $\{\mathbb{T}\} \mu f^{\text{Nat} \Rightarrow \text{Unit}}. \lambda x^{\text{Nat}}. \text{if } x = 0 \text{ then } M_1 \text{ else } M_2 ;_u \{ \forall i^{\text{Nat}}. B(u, i) \}$	(Rec)

Next we consider `circFact`:

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

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

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

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. §5.2, Page 32). 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:

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

We also set, for brevity:

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

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. $\{C(!x, g, y)\} \text{if } y = 0 \text{ then } 1 \text{ else } y \times (!x)(y - 1) :_m \{m = y! \wedge C(!x, g, y)\}$ (IfH)
4. $\{\text{T}\} M :_u \{ \forall y. \{C(!x, g, y)\} u \bullet y = y! \{C(!x, g, y)\} \}$ (Abs, \forall)
5. $\{\text{T}\} x := M \{ \forall y. \{C(!x, g, y)\} !x \bullet y = y! \{C(!x, g, y)\} \}$ (Assign)
6. $\{\text{T}\} \text{circFact} \{ \exists g. (\forall i. A(!x, g, i) \wedge !x = g) \}$ (Conseq)

The application of (Consequence) in Line 6 uses the following entailment:

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

which is inferred as follows.

$$\begin{aligned}
& \forall y g. \{C(!x, g, y)\} !x \bullet y = y! \{C(!x, g, y)\} \\
& \equiv \forall y. \forall g. \{\forall j \preceq y. A(g, g, j) \wedge !x = g\} !x \bullet y = y! \{\forall j \preceq y. A(g, g, j) \wedge !x = g\} \\
& \equiv \forall g. \forall y. \{\forall j \preceq y. A(g, g, j) \wedge !x = g\} !x \bullet y = y! \{\forall j \preceq y. A(g, g, j) \wedge !x = g\} \\
& \equiv \forall g. \forall y. ((\forall j \preceq y. A(g, g, j) \supset \{!x = g\} !x \bullet y = y! \{!x = g\})) \quad (\text{e5, e8}) \\
& \supset \exists g. (\forall y. (\forall j \preceq y. A(g, g, j) \supset \{!x = g\} g \bullet y = y! \{!x = g\}) \wedge !x = g) \quad (\star) \\
& \stackrel{\text{def}}{=} \exists g. (\forall y. (\forall j \preceq y. A(g, g, j) \supset A(g, g, y)) \wedge !x = g) \\
& \supset \exists g. (\forall y. A(g, g, y) \wedge !x = g) \\
& \supset \exists g. (\forall y. A(!x, g, y) \wedge !x = g),
\end{aligned}$$

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

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

which holds for an arbitrary y .

Finally we treat scheduler's behaviour.

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

where map is the standard higher-order map function:

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

Below a parameterised formula $C(i)$ is used for representing a sequence of states; we assume the operators for lists, $::$ and Nil, are part of the assertion language.

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

where i is implicitly universally quantified. The assertion says: given e.g. a list $l \stackrel{\text{def}}{=} [(f, a), (g, b)]$, if both $\{C_0\} fa \{C_1\}$ and $\{C_1\} gb \{C_2\}$ hold, then $\{C_0\} \text{scheduler } l \{C_2\}$ also holds. Since it is inductively given, the assertion indicates the analogous property for a job list of an arbitrary length. The main judgement can then be written:

$$\{T\} \text{scheduler} :_u \{\text{Sched}(u)\} \quad (7.2)$$

We now show how (7.2) can be derived (for the map, we omit the result values, only considering the cases when the target of the function is Unit).

$$\begin{aligned}
\text{Map}'(m, f) & \stackrel{\text{def}}{=} \{T\} m \bullet f = u \{\text{Map}(u, f)\} \\
\text{Map}(u, f) & \stackrel{\text{def}}{=} \\
& \{C_0\} u \bullet \text{Nil} \{C_0\} \wedge \\
& \forall x^{\beta}, y^{List(\beta)}. \\
& (\{C(i+1)\} f \bullet x \{C(i)\} \wedge \{C(i)\} u \bullet y \{C(0)\} \supset \{C(i+1)\} u \bullet (x :: y) \{C(0)\}) \\
\text{App}(n, z) & \stackrel{\text{def}}{=} (\{C(i+1)\} \pi_1(z) \bullet \pi_2(z) \{C(i)\} \supset \{C(i+1)\} n \bullet z \{C(i)\}).
\end{aligned}$$

We can then derive:

$$\{\mathsf{T}\} \text{map} :_m \{\forall f^{\beta \Rightarrow \text{Unit}}. \text{Map}'(m, f)\} \quad (7.3)$$

$$\{\mathsf{T}\} \text{app} :_n \{\forall z^{(\alpha \Rightarrow \text{Unit}) \times \alpha}. \text{App}(n, z)\} \quad (7.4)$$

The derivation of the judgement for `map` in (7.3) needs an inductive reasoning (closely following the one given in [35, § 5]), which we omit from the present version. The inference for the judgement (7.4) for `app` is straightforward, which we list below. For brevity below we set

$$A(x, y) \stackrel{\text{def}}{=} \{C(i+1)\} x \bullet y \{C(i)\} \wedge C(i)$$

Further, in the initial two lines we extend stateless expressions used in [Simple] (i.e. the grammar of e) to include data types, here in particular pairs.

1. $\{C(i+1) \wedge \pi_1(z) = \pi_1(z)\} \pi_1(z) :_l \{C(i+1) \wedge l = \pi_1(z)\}$ (Simple)

2. $\{C(i+1)\} \pi_1(z) :_l \{C(i+1) \wedge l = \pi_1(z)\}$ (1, Conseq)

3. $\{C(i+1) \wedge \pi_2(z) = \pi_2(z)\} \pi_2(z) :_m \{C(i+1) \wedge m = \pi_2(z)\}$ (Simple)

4. $\{C(i+1)\} \pi_2(z) :_m \{C(i+1) \wedge m = \pi_2(z)\}$ (3, Conseq)

5. $\{C(i+1) \wedge A(\pi_1(z), \pi_2(z)) \wedge l = \pi_1(z)\} \pi_2(z) :_m \{C(i+1) \wedge A(l, m)\}$
(4, Constancy, Conseq)

6. $\{C(i+1) \wedge A(\pi_1(z), \pi_2(z))\} \pi_1(z) \pi_2(z) \{C(i)\}$ (2, Constancy, 5, App)

7. $\{\mathsf{T}\} \lambda z. \pi_1(z) \pi_2(z) :_n \{\forall z. (\{A(\pi_1(z), \pi_2(z)) \wedge C(i+1)\} n \bullet z \{C(i)\})\}$ (Abs)

8. $\{\mathsf{T}\} \lambda z. \pi_1(z) \pi_2(z) :_n \{\forall z. \text{App}(n, z)\}$ (e5)

Assuming the judgement for `map` is also deduced, we now derive (7.2) from (7.3) and (7.4). We use the following inference on validity.

$$\begin{aligned} & \forall f. \text{Map}'(m, f) \wedge \forall z. \text{App}(n, z) \\ & \supset \text{Map}'(m, n) \wedge \text{App}(n, z) \quad (\forall\text{-inst}) \\ & \supset \{\mathsf{T}\} m \bullet n = u \{ \text{Map}(u, n) \wedge \text{App}(n, z) \} \quad (\text{e7}) \\ & \supset \{\mathsf{T}\} m \bullet n = u \{ \text{Map}(u, \langle g, a \rangle) \wedge \text{App}(n, \langle g, a \rangle) \} \quad (\forall\text{-inst}, (\text{e8})) \\ & \supset \{\mathsf{T}\} m \bullet n = u \{ \text{Sched}(u) \} \quad (\text{modus ponens}, (\text{e8})) \end{aligned}$$

We can now deduce:

1. $\{\mathsf{T}\} \text{map} :_m \{\forall f. \text{Map}'(m, f)\}$ (7.3)

2. $\{\mathsf{T}\} \text{app} :_n \{\forall z. \text{App}(n, z)\}$ (7.4)

3. $\{\forall f. \text{Map}'(m, f)\} \text{app} :_n \{\forall f. \text{Map}'(m, f) \wedge \forall z. \text{App}(n, z)\}$ (Constancy)

4. $\{\forall f. \text{Map}'(m, f)\} \text{app} :_n \{\{\mathsf{T}\} m \bullet n = u \{ \text{Sched}(u) \} \}$ (Conseq)

5. $\{\mathsf{T}\} (\text{map app}) :_u \{ \text{Sched}(u) \}$ (1, 4, App)

where, in Line 4, we use the above inference on validity for the entailment. We have reached (7.2).

8 Discussion

In this section we first briefly outline two further topics we could not touch in the main sections; then we present discussions on related work, focussing on past investigations of logics for Hoare-like higher-order imperative programming languages such as Algol.

8.1 Further Topics

Other Completeness Properties As observed in Section 6, our proof system can derive characteristic assertions for semi-closed FCFs w.r.t. total correctness, and is (relatively) complete for them, again w.r.t. total correctness. Does this extend to the whole language? We strongly believe so.

Conjecture. (1) For each well-typed M , $\vdash \{C\}M :_u \{C'\}$ such that (C, C') characterises M in the sense of Section 6.

(2) For each TCA C' at u , $\models \{C\}V :_u \{C'\}$ implies $\vdash \{C\}V :_u \{C'\}$.

(2) is a corollary of (1) (through Kleymann's consequence rule). The property, if it holds, indicates that the assertion language can pinpoint, and the proof rules can relatively justify, any total correctness property a program can have. We also believe the same result extends to partial correctness, using an admissibility condition.

Aliasing, Local State, Polymorphism and Recursive Types In Section 3, it is observed that allowing reference types to be carried by other types (including arrow types 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 phenomenon results in significant increase of complexity in behaviour, hence in its semantic and logical treatment. A clean and tractable logical treatment of this generalised class of behaviour is possible on the basis of the logic studied here, using ideas from the π -calculus. This extension is reported in [9].

On the basis of these stratifications, local state, where a new local reference is dynamically created, stored and communicated, can also be captured by a simple enrichment of the assertion language, with further, non-trivial added complexity in both behaviours and their logics. The full exploration of local state will be reported elsewhere. Another natural extension is polymorphism and recursive types. While we have omitted them in the present paper, their integration is straightforward following [35]. Some of these and other extensions will be reported in coming reports.

8.2 Related Work

In the following we discuss related work focussing on logics for higher-order imperative languages. For comparisons in different contexts (for example, process logics and general aliasing), see [9, 32, 33, 35].

Equational Logics for Higher-Order Functions. Equational logics for the λ -calculi have been studied since the classical work by Curry and Church. LCF [19] 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, 63]. 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 [35]. This suggests fruitful interplay between the present logical method, on the one hand, and the reasoning principles as developed in, and extending, [4, 63] on the other.

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 focussing on Algol and its derivatives. One of the basic works in this period is Clarke’s work [11] (see also [13, §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. [12].
- 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 $\{T\}P\{F\}$, 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.

Clarke’s construction of general computing device under finite models relates to Jones and Muchnick’s preceding work in [39, 40], 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. Among others example they showed [39] that undecidability can come from differences in the calling mechanisms (their work is also related with Olderog’s work discussed).

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

focusses 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 PCFv” we mean that Nat is interpreted as a (non-trivial) finite domain, e.g. consider Nat consists of 0 and 1 with standard modular arithmetics. By “static local variable” we mean a local variable declaration never exported outside of its scope.

Proposition 8.1 *The termination problem of the imperative PCFv in §3.1 extended with static local variables is undecidable under finite base types.*

Proof By Clarke’s result (see also an alternative construction of Turing machine in Cousot’s survey [13]). \square

Note static local variables in the above sense can be easily captured in the present logic through the standard method in Hoare logic, cf.[7]. At this point we do not know whether finite imperative PCFv without static local variables has the same properties or not.

Proposition 8.1 indicates that Hoare-like logic in the traditional sense cannot be complete under finite models. We next consider the other side of the coin, suggesting inherent complexity of the assertion language of the present logic. The discussion starts from the following conjecture, following Loader’s significant result on call-by-name PCF.

Conjecture 8.2 The observational congruence \cong in finite PCFv is undecidable.

Here by “finite PCFv” we mean PCFv whose base types are restricted to finite ones (in particular we stipulate natural numbers consist of only 0 and 1: we note this makes PCFv’s termination decidable since in that case a program’s termination is reducible to that of while programs under finite models). Loader showed, in [46], the contextual congruence of finitary call-by-name PCF (with bottom, which is essential, but without recursion) is undecidable⁶. We believe his combinatorial argument extends to finite PCFv (so that Conjecture 8.2 holds), though this has not been verified as of April 2006.

Under the above Conjecture, we easily obtain the following. 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 8.3 *If Conjecture 8.2 holds, then the validity of assertions in the logic for the functional sublanguage is undecidable even under finite base types.*

Proof Because, for this language, we can derive characteristic formulae are derivable following Section 6. By the extensionality axiom noted in Section 3, 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. \square

⁶ The congruence in finite PCF becomes decidable if we have a decidable enumeration procedure for all representative elements of congruence classes for each higher-order type. However, while we do know the upper bound of their number for each type (which are all extensional functions of that type), we cannot figure out their exact number due to sequentiality constraint. This is essentially why a natural (and in fact any) enumeration procedure does not work.

Corollary 8.4 *If Conjecture 8.2 holds, then the validity of assertions in the present logic for finite imperative PCFvis undecidable.*

Proof By taking the empty reference basis. □

Thus having finite domains does not make the logic substantially easier, at least for its higher-order part. We may note that the above result does *not* limit the use of finite models for model checking. One may also note, as Harel did [24], it is standard to consider only strong models for total correctness (a model of arithmetics is strong if it can represent all arithmetical relations). The point of the above discussion (which easily extends to imperative call-by-name PCF) is to elucidate an inherent complexity of logical description of higher-order procedures under a the above discussion crucially relies on direct description of higher-order behaviours in the logical language. The partial correctness counterpart of the present logic also allows the same argument.

Program Logics for Sublanguages of Algol (1): Olderog’s Analysis. In [59]. 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. [71] independently presented a sound and complete logic for an Algol-like language with copy rule. Later Olderog [60] 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}_{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}_{pas} (here “admissibility” indicates closure under natural syntactic transformations respecting semantics inside \mathcal{L}_{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 [60], follows (we use `letrec/let` for readability).

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

which is, modulo translation of `let/letrec`, easily a PCFv-program, strengthening our intuition behind Proposition 8.3.

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 non-regular 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 [59, 60] 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 [14], 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 [59], we have:

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

which is now of type Prg , so that the judgement is an ordinary Hoare triple. There are two 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 `Map` and `App` as we did in Section 7 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 [17] and Halpern [23] studied completeness for Clarke’s sub-language of Algol. Halpern [23] 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:

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

where $C\mathcal{A}$ 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 5. 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 [62].

Reynolds’s Specification Logic. Specification logic by Reynolds [67] is a program logic for Idealised Algol which combines 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 filled 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 [67] are done for formalising rules for noninterference. Subsequent studies on semantics of specification logics by O’Hearn [66], Tennent [70] and Ghica [18] 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 asserts on and compositionally verifies higher-order expressions, whereas Reynolds’s logic does neither. As a possible counterpart, Reynolds’s logic uses 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 a 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 (intended to be) 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 framework presented here can cleanly accommodate partial correctness.

Reynolds’s logic precedes the presented logic in that it aims 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 reflecting his choice of the language, 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 [29]. 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, we can obtain a clean compositional logic following the present framework and its ramifications.

Other Related Work. Mason [47] 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 [25], introduced by Pratt [65] and studied by Harel and others [24], 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 [42], and are found in the work by von Oheimb [72], Leavens and Baker [45], Abadi and Leino [5] and Bierman and Parkinson [10], 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 come 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 [58, 68] study extensions of Hoare logic in which new logical connectives are used for reasoning about aliasing and 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. Since the comparison with the present series of works is detailed in the sequel to the present work which treats aliasing [9], here we only briefly mention a basic difference in philosophy. The present family of logics present program logics whose aim is to precisely capture the observational semantics of high-level programming languages as stratified logical theories. Separation logic aims to offer direct reasoning principles for the resource-oriented, or low-level, aspects of programming languages using their novel connectives. Rather than aiming to capture resources in program logics (which is an important topic for study), the present work focusses on the non-trivial high-level behaviour of programs, higher-order procedures, and tries to precisely capture its standard semantics through assertions and proof rules, as demonstrated by our emphasis on observational completeness (cf. Theorem 6.10).

The characterisation of observational semantics by logical formulae are well-known in process logics [26] and is also discussed in Hoare logics [29, 50]. Nevertheless, none of the related work discussed above reports observational completeness in the sense of Theorem 6.10. 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. [7, 41, 59].

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 §7 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 [3], 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. [15, 27, 29] (this line of study includes integrated software development frameworks such as VDM [38] and more recent Z notation[74]). One of the

ideas strongly advocated in [29] 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 [56]), no tractable solutions have been known (Hoare and Jifeng [27, 29] 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 [74]? The semantic analysis of the proposed logic and its extensions, as partly discussed in Section 6, 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 [31, 33] 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. [30, 31, 33] discuss process logics and their relationship to the program logics in detail.

References

1. C- home page. <http://www.cminusminus.org>.
2. Haskell home page. <http://haskell.org>.
3. The Java Modeling Language (JML) home page. <http://www.jmlspecs.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. K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.
8. Krzysztof R. Apt. Ten years of hoare’s logic: A survey – part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.
9. Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *ICFP’05*, pages 280–293, 2005. www.dcs.qmul.ac.uk/~kohei/logics.
10. Gavin M. Bierman and Matthew J. Parkinson. Separation logic and abstractions. In *POPL’05*, 2005.
11. Edmund Clarke. Programming language constructs for which it is impossible to obtain good hoare axiom systems. *J. ACM*, 26(1):129–147, 1979.
12. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
13. Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.
14. Werner Damm and Bernhard Josko. A sound and relatively* complete hoare-logic for a language with higher type procedures. *Acta Inf.*, 1983.
15. 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.
16. Robert W. Floyd. Assigning meaning to programs. In *Symp. in Applied Mathematics*, volume 19, 1967.
17. 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.
18. Dan R. Ghica. Semantical Analysis of Specification Logic, 3: An Operational Approach. In *Proc. ESOP*, volume 2986 of *LNCS*, pages 264–278, 2004.
19. Mike Gordon, Robin Milner, and Christopher Wadsworth. Edinburgh LCF. *LNCS 78*, Springer Verlag, 78.
20. Irene Greif and Albert R. Meyer. Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer. *ACM Trans. Program. Lang. Syst.*, 3(4), 1981.
21. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI’02*. ACM, 2002.
22. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
23. Joseph Y. Halpern. A good hoare axiom system for an algol-like language. In *11th POPL*, pages 262–271. ACM Press, 1984.
24. David Harel. Proving the correctness of regular deterministic programs. *TCS*, 12(16), 1980.
25. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
26. Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.

27. C. A. R. Hoare. Proof of correctness of data representations. pages 385–396, 2002.
28. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
29. Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
30. Kohei Honda. Sequential process logics: Soundness proofs. Available at: www.dcs.qmul.ac.uk/~kohei/logics, November 2003. Typescript, 50 pages.
31. Kohei Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.
32. Kohei Honda. From process logic to program logic (full version of [31]). Available at: www.dcs.qmul.ac.uk/~kohei/logics, November 2004. Typescript, 52 pages.
33. Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/~kohei/logics, March 2004. Typescript, 234 pages.
34. Kohei Honda and Nobuko Yoshida. Game-Theoretic Analysis of Call-by-Value Computation. *TCS*, 221:393–456, 1999.
35. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202, 2004.
36. Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*. www.dcs.qmul.ac.uk/~kohei/logics.
37. J. Martin E. Hyland and C. H. Luke Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
38. C B Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., 1986.
39. Niel Jones and S Muchnick. Complexity of finite memory programs with recursion. *Journal of ACM*, 25(2):312–321, 1977.
40. Niel Jones and S Muchnick. Even simple programs are hard to analyze. *Journal of ACM*, 24(2):338–350, 1977.
41. Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.
42. Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.
43. Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
44. Peter Landin. A correspondence between algol 60 and church's lambda-notation. *Comm. ACM*, 8:2, 1965.
45. 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.
46. Ralph Loader. Finitary PCF is not decidable. *Theor. Comput. Sci.*, 266(1-2):341–364, 2001.
47. I A. Mason. A first order logic of effects. *Theoretical Computer Science*, 185(2):277–318, 1997.
48. Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *JFP*, 1(3):287–327, 1991.
49. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
50. Albert R. Meyer. Floyd-Hoare logic defines semantics (preface version). In *LICS'86*, 1986.
51. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *POPL'88*, 1988.
52. Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
53. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1):1–77, 1992.
54. Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

55. 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.
56. David A. Naumann. Soundness of data refinement for a higher-order imperative language. *Theor. Comput. Sci.*, 278(1-2):271–301, 2002.
57. P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.
58. Peter O’Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL’04*, 2004.
59. Ernst-Rüdiger Olderog. Sound and complete Hoare-like calculi based on copy rules. *Acta Inf.*, 16:161–197, 1981.
60. Ernst-Rüdiger Olderog. A characterization of hoare’s logic for programs with pascal-like procedures. In *15th Theory of Computing*, pages 320–329, 1983.
61. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
62. Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *HOOTS’98*, CUP, pages 227–273, 1998.
63. Gordin Plotkin and Martín Abadi. A logic for parametric polymorphism. In *LICS’98*, 1998.
64. Gordon D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI, Aarhus University, 1981.
65. 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.
66. John C. Reynolds. *The Craft of Programming*. Prentice-Hall International, 1981.
67. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
68. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS’02*, 2002.
69. 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.
70. Robert D. Tennent. Semantical analysis of specification logic. *I & C*, 85(2):135–162, 1990.
71. 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.
72. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001.
73. Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.
74. Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.

A Typed Congruence and Soundness

A.1 Typed Congruence

We give the compatibility rules in Figure 10. As in the typing rules in Figure 1, the congruence rules do not include the weakening. Since we have no hiding or abstraction of reference names, the reference basis remains constant throughout the congruent closure. This leads to a refined notion of congruence that in fact subsumes the one that allows weakening.

Fig. 10 Compatibility Rules for Imperative PCFv

$$\begin{array}{c}
\text{[Var]} \frac{\Gamma(x) = \alpha}{\Gamma; \Delta \vdash x \mathcal{R} x : \alpha} \quad \text{[Op]} \frac{\Gamma; \Delta \vdash M_i \mathcal{R} N_i : \alpha_i \ (1 \leq i \leq n) \quad \text{op} : \alpha_1 \times \dots \times \alpha_n \Rightarrow \beta}{\Gamma; \Delta \vdash \text{op}(M_1..M_n) \mathcal{R} \text{op}(N_1..N_n) : \beta} \\
\text{[Abs]} \frac{\Gamma, x : \alpha; \Delta \vdash M \mathcal{R} N : \beta}{\Gamma; \Delta \vdash \lambda x^\alpha. M \mathcal{R} \lambda x^\alpha. N : \alpha \Rightarrow \beta} \quad \text{[Rec]} \frac{\Gamma, x : \alpha \Rightarrow \beta; \Delta \vdash \lambda y^\alpha. M \mathcal{R} \lambda y^\alpha. N : \alpha \Rightarrow \beta}{\Gamma; \Delta \vdash \mu x^{\alpha \Rightarrow \beta}. \lambda y^\alpha. M \mathcal{R} \mu x^{\alpha \Rightarrow \beta}. \lambda y^\alpha. N : \alpha \Rightarrow \beta} \\
\text{[App]} \frac{\Gamma; \Delta \vdash M_1 \mathcal{R} N_1 : \alpha \Rightarrow \beta \quad \Gamma; \Delta \vdash M_2 \mathcal{R} N_2 : \alpha}{\Gamma; \Delta \vdash M_1 M_2 \mathcal{R} N_1 N_2 : \beta} \\
\text{[If]} \frac{\Gamma; \Delta \vdash M_0 \mathcal{R} N_0 : \text{Bool} \quad \Gamma; \Delta \vdash M_i \mathcal{R} N_i : \alpha_i \ (i = 1, 2)}{\Gamma; \Delta \vdash \text{if } M_0 \text{ then } M_1 \text{ else } M_2 \mathcal{R} \text{if } N_0 \text{ then } N_1 \text{ else } N_2 : \alpha} \\
\text{[Inj]} \frac{\Gamma; \Delta \vdash M \mathcal{R} N : \alpha_i}{\Gamma; \Delta \vdash \text{in}_i(M) \mathcal{R} \text{in}_i(N) : \alpha_1 + \alpha_2} \\
\text{[Case]} \frac{\Gamma; \Delta \vdash M_0 \mathcal{R} N_0 : \alpha_1 + \alpha_2 \quad \Gamma, x_i : \alpha_i; \Delta \vdash M_i \mathcal{R} N_i : \beta \ (i = 1, 2)}{\Gamma; \Delta \vdash \text{case } M_0 \text{ of } \{\text{in}_i(x_i^{\alpha_i}). M_i\}_{i \in \{1, 2\}} \mathcal{R} \text{case } N_0 \text{ of } \{\text{in}_i(x_i^{\alpha_i}). N_i\}_{i \in \{1, 2\}} : \beta} \\
\text{[Pair]} \frac{\Gamma; \Delta \vdash M_i \mathcal{R} N_i : \alpha_i \ (i = 1, 2)}{\Gamma; \Delta \vdash \langle M_1, M_2 \rangle \mathcal{R} \langle N_1, N_2 \rangle : \alpha_1 \times \alpha_2} \quad \text{[Proj]} \frac{\Gamma; \Delta \vdash M \mathcal{R} N : \alpha_1 \times \alpha_2}{\Gamma; \Delta \vdash \pi_i(M) \mathcal{R} \pi_i(N) : \alpha_i \ (i = 1, 2)} \\
\text{[Deref]} \frac{\Delta(x) = \text{Ref}(\alpha)}{\Gamma; \Delta \vdash !x \mathcal{R} !x : \alpha} \quad \text{[Assign]} \frac{\Gamma; \Delta \vdash M \mathcal{R} N : \alpha \quad \Delta(x) = \text{Ref}(\alpha)}{\Gamma; \Delta \vdash (x := M) \mathcal{R} (x := N) : \text{Unit}}
\end{array}$$

Below we summarise the basic properties of \cong . As before, when we write $\Gamma \cdot \Gamma'$ etc., it is understood that $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$.

Proposition A.1 (properties of \cong)

1. (renaming) Let ϕ be an injective name substitution. Then $\Gamma; \Delta \vdash M \cong N : \alpha$ iff $\Gamma\phi; \Delta\phi \vdash M\phi \cong N\phi : \alpha$.
2. (weakening) $\Gamma; \Delta \vdash M \cong N : \alpha$ implies $\Gamma \cdot \Gamma'; \Delta \vdash M \cong N : \alpha$.
3. (thinning) $\Gamma \cdot \Gamma'; \Delta \vdash M \cong N : \alpha$ and $\text{fv}(M, N) \cap \text{dom}(\Gamma') = \emptyset$ imply $\Gamma; \Delta \vdash M \cong N : \alpha$.
4. (divergence) Assume $\Delta \vdash M_{1,2} : \alpha$ and, for each $\Delta \vdash \sigma$, we have $(M_i, \sigma) \uparrow (i = 1, 2)$. Then $\Delta \vdash M_1 \cong M_2 : \alpha$.

5. (substitution) $\Gamma; \Delta \vdash M \cong N : \alpha$ iff, with $\text{dom}(\Gamma) = \{\vec{x}\}$, we have $\Delta \vdash M\sigma \cong N\sigma : \alpha$ for each $\sigma \stackrel{\text{def}}{=} [\vec{V}/\vec{x}]$ with $\Delta \vdash V_i : \Gamma(x_i)$ for each $x_i \in \text{dom}(\Gamma)$.
6. (invariance) Let $\Delta \vdash \sigma_{1,2}$ and write $\Delta \vdash \sigma_1 \cong \sigma_2$ when $\Delta \vdash \sigma_1(x) \cong \sigma_2(x) : \alpha$ for each $x : \text{Ref}(\alpha) \in \Delta$. Then if $\Delta \vdash M_1 \cong M_2 : \alpha$ and $\Delta \vdash \sigma_1 \cong \sigma_2$, and $(M_1, \sigma_1) \Downarrow (V_1, \sigma'_1)$ then $(M_2, \sigma_2) \Downarrow (V_2, \sigma'_2)$ such that $\Delta \vdash V_1 \cong V_2 : \alpha$ and $\Delta \vdash \sigma'_1 \cong \sigma'_2$.
7. (unit) Let $\Delta \vdash V : \text{Unit}$. Then $V \cong ()$.
8. (boolean) Let $\Delta \vdash V : \text{Bool}$. Then either $V \cong \text{t}$ or $V \cong \text{f}$. Further $\text{t} \not\cong \text{f}$.
9. (numerals) Let $\Delta \vdash V : \text{Nat}$. Then $V \cong n$ for some numeral n . Further $m \not\cong n$ whenever m and n are distinct numerals.
10. (first-order operators) Let $\Delta \vdash \text{op}(\vec{V}) : \alpha$. Then there exists W such that $(\text{op}(\vec{V}), \sigma) \Downarrow (W, \sigma)$ such that $\text{op}(\vec{V}) \cong W$ for each $\Delta \vdash \sigma$.
11. (abstraction) Let $\Delta \vdash (\lambda x.M)V : \alpha$. Then $\Delta \vdash (\lambda x.M)V \cong M[V/x] : \alpha$.
12. (recursion) Assume $\Gamma \cdot x : \alpha ; \Delta \vdash \lambda y.M : \alpha$ and let $W \stackrel{\text{def}}{=} (\lambda y.M)[\mu x.\lambda y.M/x]$. Then we always have $\Gamma; \Delta \vdash (\lambda y.M)[W/x] \cong (\mu x.\lambda y.M)$.
13. (product) Let $\Delta \vdash W : \alpha \times \beta$. Then $W \cong \langle \pi_1(W), \pi_2(W) \rangle$. Further $(\pi_i(W), \sigma) \Downarrow (V, \sigma)$ implies $\pi_i(W) \cong V$ ($i = 1, 2$).
14. (sum) Let $\Delta \vdash W : \alpha + \beta$. Then $W \cong \text{in}_i(V)$ for some $i \in \{1, 2\}$ and V . Further $\Delta \vdash \text{in}_i(V) \cong \text{in}_i(W)$ implies $\Delta \vdash V \cong W$.

Proof All are standard. (7–14) use the fact that $\Delta \vdash M : \alpha$ and $\Delta \vdash \sigma$ imply either $(M, \sigma) \Uparrow$ or $(M, \sigma) \Downarrow$. \square

In Proposition A.1, (2) does not weaken the reference basis: we cannot, as Example 5.3 has shown. Similarly (5) considers values typed under the identical reference basis.

The properties corresponding to those stated in Proposition A.1 for \sqsubseteq also hold for \sqsubseteq . In particular we observe:

- Proposition A.2** 1. (renaming) Let ϕ be an injective name substitution. Then $\Gamma; \Delta \vdash M \sqsubseteq N : \alpha$ iff $\Gamma\phi; \Delta\phi \vdash M\phi \sqsubseteq N\phi : \alpha$.
2. (weakening) $\Gamma; \Delta \vdash M \sqsubseteq N : \alpha$ implies $\Gamma \cdot \Gamma'; \Delta \vdash M \sqsubseteq N : \alpha$.
 3. (thinning) $\Gamma \cdot \Gamma'; \Delta \vdash M \sqsubseteq N : \alpha$ and $\text{fv}(M, N) \cap \text{dom}(\Gamma') = \emptyset$ imply $\Gamma; \Delta \vdash M \sqsubseteq N : \alpha$.
 4. (divergence) Assume $\Delta \vdash M_{1,2} : \alpha$ and, for each $\Delta \vdash \sigma$, $(M_1, \sigma) \Uparrow$. Then $\Delta \vdash M_1 \sqsubseteq M_2 : \alpha$.
 5. (substitution) $\Gamma; \Delta \vdash M \sqsubseteq N : \alpha$ iff, with $\text{dom}(\Gamma) = \{\vec{x}\}$, we have $\Delta \vdash M\sigma \sqsubseteq N\sigma : \alpha$ for each $\sigma \stackrel{\text{def}}{=} [\vec{V}/\vec{x}]$ with $\Delta \vdash V_i : \Gamma(x_i)$ for each $x_i \in \text{dom}(\Gamma)$.
 6. (invariance) Let $\Delta \vdash \sigma_{1,2}$ and write $\Delta \vdash \sigma_1 \sqsubseteq \sigma_2$ when $\Delta \vdash \sigma_1(x) \sqsubseteq \sigma_2(x) : \alpha$ for each $x : \text{Ref}(\alpha) \in \Delta$. Then if $\Delta \vdash M_1 \sqsubseteq M_2 : \alpha$ and $\Delta \vdash \sigma_1 \sqsubseteq \sigma_2$, and $(M_1, \sigma_1) \Downarrow (V_1, \sigma'_1)$ then $(M_2, \sigma_2) \Downarrow (V_2, \sigma'_2)$ such that $\Delta \vdash V_1 \sqsubseteq V_2 : \alpha$ and $\Delta \vdash \sigma'_1 \sqsubseteq \sigma'_2$.

Proof Standard. For example, in (substitution), “only if” direction is immediate, while “if” direction is by closing $M_{1,2}$ by the λ -abstractions over \vec{x} and applying \vec{V} , and by noting β_V -equality is a subset of \cong . \square

A.2 Proof of Proposition 5.13

Below we lists the proof of the soundness of axioms stated in Section 3 and the proof rules in Figures 5 and 6.

Among the axioms in Figure 3, (t1)–(t8) are easy using Proposition A.1. For (t1),

$$\begin{aligned} & \llbracket x^{\text{Unit}} \rrbracket_{\mathcal{M}} \cong () && \text{(Proposition A.1-7)} \\ \supset \mathcal{M} \models x^{\text{Unit}} = () && \text{(Definition 5.8)} \end{aligned}$$

For (t2),

$$\begin{aligned} & \llbracket x^{\text{Bool}} \rrbracket_{\mathcal{M}} \cong t \vee \llbracket x^{\text{Bool}} \rrbracket_{\mathcal{M}} \cong f && \text{(Proposition A.1-8)} \\ \supset \mathcal{M} \models x^{\text{Bool}} = t \vee x^{\text{Bool}} = f && \text{(Definition 5.8)} \end{aligned}$$

For (t3),

$$\begin{aligned} & t \neq f && \text{(Proposition A.1-8)} \\ \supset \mathcal{M} \models t \neq f && \text{(Definition 5.8)} \end{aligned}$$

(t4–t8) are similar, using the corresponding clauses in Proposition A.1.

Each of (e1–e8) in Figure 4 is essentially direct from the definition of evaluation formulae and the standard logical tautologies. For example, we infer for (e1) as follows. We omit the typing below.

$$\begin{aligned} & (\xi, \sigma_0) \models \{C_1\}x \bullet y = z\{C'\} \wedge (\xi, \sigma_0) \models \{C_2\}x \bullet y = z\{C'\} \\ \equiv & \forall \sigma. ((\xi, \sigma) \models C_1 \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C') \wedge \\ & \forall \sigma. ((\xi, \sigma) \models C_2 \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C') \\ \equiv & \forall \sigma. ((\xi, \sigma) \models C_1 \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C') \wedge \\ & (\xi, \sigma) \models C_2 \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C') \\ \equiv & \forall \sigma. ((\xi, \sigma) \models C_1 \vee (\xi, \sigma) \models C_2) \\ & \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C') \\ \equiv & \forall \sigma. ((\xi, \sigma) \models (C_1 \vee C_2)) \\ & \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C') \end{aligned}$$

(e2–e5) are similar. For (e6):

$$\begin{aligned} & (\xi, \sigma_0) \models \{C\}x \bullet y = z\{A^z \supset C'\} \\ \equiv & \forall \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models A \supset C') \\ \equiv & \forall \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. ((\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge \\ & (\xi \models A \supset (z : V \cdot \xi, \sigma') \models C')))) \\ \supset & \forall \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. ((\xi \models A \supset (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma')) \wedge \\ & (\xi \models A \supset (z : V \cdot \xi, \sigma') \models C')))) \\ \equiv & \xi \models A \supset \\ & \forall \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. ((\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C')))) \end{aligned}$$

Note the entailment cannot be reversed, since adding an assumption A to a conjunct strictly weakens the assertion in general.

For (e7) we infer:

$$\begin{aligned} & (\xi, \sigma_0) \models \{C\}x \bullet y = z\{C'\} \\ \supset & \forall \sigma. (((\xi, \sigma) \models C \wedge \xi \models A) \\ & \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge (z : V \cdot \xi, \sigma') \models C') \\ \equiv & \forall \sigma. (((\xi, \sigma) \models C \wedge \xi \models A) \\ & \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge ((z : V \cdot \xi, \sigma') \models C' \wedge \xi \models A)) \\ \equiv & \forall \sigma. (((\xi, \sigma) \models C \wedge A) \\ & \supset \exists V, \sigma'. (\xi(x) \bullet \xi(y), \sigma) \Downarrow (V, \sigma') \wedge ((z : V \cdot \xi, \sigma') \models C' \wedge A)) \end{aligned}$$

Again the entailment is in general irreversible. Finally (e8) is immediate. We have proved Proposition 5.13.

A.3 Proof of Theorem 5.14

The proofs of the soundness of the rules in Figures 5 and 6 closely follow those of the soundness of the traditional Hoare logic [73]. We present the reasoning for each rule one by one, starting from the structural rules.

Convention A.3 Since each statement used in the proof is lengthy, we use numbered sequences for proofs, meaning implications from the top to the bottom, starting from premises (if any).

For $[\wedge\text{-}\supset]$, we first observe:

Lemma A.4 *Let V be well-typed and ξ is a well-typed substitution of semi-closed programs for free variables in V . Then $V\xi$ is a semi-closed value.*

Proof By easy structural induction. □

We now reason:

1. $\forall \xi, \sigma. ((\xi, \sigma) \models C \wedge A \supset (\xi \cdot u : (V\xi), \sigma) \models C')$ (premise)

2. $\forall \xi, \sigma. (((\xi, \sigma) \models C \wedge (\xi, \sigma) \models A) \supset (\xi \cdot u : (V\xi), \sigma) \models C')$ (def)

3. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset (\xi, \sigma) \models A \supset (\xi \cdot u : (V\xi), \sigma) \models C')$

4. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset (\xi \cdot u : (V\xi), \sigma) \models A \supset (\xi \cdot u : (V\xi), \sigma) \models C')$ (Lem. 5.11-3)

5. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset (\xi \cdot u : (V\xi), \sigma) \models A \supset C')$

The reasoning for $[\supset\text{-}\wedge]$ is essentially identical with that of (e6). $[\vee\text{-}Pre]$ corresponds to (e1), while $[\wedge\text{-}Post]$ is its dual. The rules $[Aux\exists]$, $[Aux\forall]$, $[Aux_{inst}]$ and $[Aux_{abst}]$ are all instances of $[Consequence\text{-}Aux]$, while $[Consequence\text{-}Aux]$ is the precise mirror of Kleymann's corresponding rule [41] so that Kleymann's argument is directly adaptable.

Thus we are left with *[Invariance]*, which we justify below.

1. $\forall(\xi, \sigma)^{\Gamma; \Delta}. ((\xi, \sigma) \models C \supset \exists V, \sigma'. (M\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot u : V, \sigma') \models C')$
(premise)

2. $\forall(\xi, \sigma)^{\Gamma; \Delta}. \forall \sigma_0^{\Delta_0}. ((\xi, \sigma \cdot \sigma_0) \models C \supset \exists V, \sigma'. (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0) \wedge (\xi \cdot u : V, \sigma' \cdot \sigma_0) \models C')$ (Pro.A.1-3)

3. $\forall(\xi, \sigma)^{\Gamma; \Delta}. \forall \sigma_0^{\Delta_0}. (((\xi, \sigma \cdot \sigma_0) \models C \wedge (\xi, \sigma_0) \models A) \supset \exists V, \sigma'. (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0) \wedge (\xi \cdot u : V, \sigma' \cdot \sigma_0) \models C' \wedge (\xi, \sigma_0) \models A)$

4. $\forall(\xi, \sigma)^{\Gamma; \Delta}. \forall \sigma_0^{\Delta_0}. (((\xi, \sigma \cdot \sigma_0) \models C \wedge (\xi, \sigma \cdot \sigma_0) \models A) \supset \exists V, \sigma'. (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0) \wedge (\xi \cdot u : V, \sigma' \cdot \sigma_0) \models C' \wedge (\xi \cdot u : V, \sigma' \cdot \sigma_0) \models A)$ (Pro.A.1-3)

5. $\forall(\xi, \sigma)^{\Gamma; \Delta}. \forall \sigma_0^{\Delta_0}. (((\xi, \sigma \cdot \sigma_0) \models C \wedge A) \supset \exists V, \sigma'. (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0) \wedge (\xi \cdot u : V, \sigma' \cdot \sigma_0) \models C' \wedge A)$

(note that $V\xi = V$ since V is semi-closed). This concludes the soundness proof for structural rules.

We now turn to the rules in Figure 5. For *[Var]*, let the typing be $\Gamma; \Delta \vdash x : \alpha$ (note this means x is not a reference name) and let $\Gamma; \Delta \vdash C$. Below we remember $u \notin \text{dom}(\Gamma, \Delta)$ (since u is an anchor).

1. $\frac{}{} \text{—} \text{ (no premise)}$

2. $\forall \mathcal{M}^{\Gamma; \Delta}. (\mathcal{M} \models C[x/u] \supset \mathcal{M} \cdot u : (\llbracket x \rrbracket_{\mathcal{M}}) \models C)$ (Lemma 5.12-1(a))

For *[Const]*, let the typing be $\Gamma; \Delta \vdash c : \alpha$ and $\Gamma; \Delta \vdash C$. Again noting u is fresh,

1. $\frac{}{} \text{—} \text{ (no premise)}$

2. $\forall \mathcal{M}^{\Gamma; \Delta}. (\mathcal{M} \models C[c/u] \supset \mathcal{M} \cdot u : (\llbracket c \rrbracket_{\mathcal{M}}) \models C)$ (Lemma 5.12-1(a))

For *[Op]*, we infer as follows, assuming $\Gamma; \Delta \vdash \text{op}(\vec{M}) : \alpha$ and $\Gamma; \Delta \vdash C, C'$ where, by Convention 4.1 (2), we assume $\{\vec{m}\} \cap \text{dom}(C, C') = \emptyset$. Below we let ξ' be typed by

$\Gamma, \vec{m} : \vec{\beta}; \Delta$ and ξ by $\Gamma; \Delta$. We write (prem) to indicate it is a premise.

$$\begin{array}{l}
1. \forall \xi', \sigma. ((\xi', \sigma) \models C \supset \exists \sigma', V. ((M_0 \xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi'[m_0 : V], \sigma') \models C_1)) \quad (\text{prem}) \\
\hline
2. \forall \xi', \sigma. ((\xi', \sigma) \models C_1 \supset \exists \sigma', V. ((M_1 \xi', \sigma) \Downarrow (V, \sigma') \wedge (\xi'[m_1 : V], \sigma') \models C_2)) \quad (\text{prem}) \\
\hline
\vdots \\
\hline
n. \forall \xi', \sigma. ((\xi', \sigma) \models C_{n-1} \supset \exists \sigma', V. ((M_{n-1} \xi', \sigma) \Downarrow (V, \sigma') \wedge (\xi'[m_n : V], \sigma') \models C'[\text{op}(\vec{m})/u])) \quad (\text{prem}) \\
\hline
n+1. \forall \xi', \sigma. ((\xi', \sigma) \models C \supset \\
\quad \exists \sigma', \vec{V}. ((\text{op}(M_0..M_{n-1}) \xi', \sigma) \longrightarrow^* (\text{op}(V_1..V_{n-1}), \sigma') \wedge \\
\quad (\xi'[\vec{m} : \vec{V}], \sigma') \models C'[\text{op}(\vec{m})/u])) \quad (1..n) \\
\hline
n+2. \forall \xi', \sigma. ((\xi', \sigma) \models C \supset \\
\quad \exists \sigma', \vec{V}, W. ((\text{op}(M_0..M_{n-1}) \xi', \sigma) \longrightarrow^* (\text{op}(V_1..V_{n-1}), \sigma') \wedge \\
\quad (\xi'[\vec{m} : \vec{V}] \cdot u : \text{op}(\vec{V}), \sigma') \models C')) \quad (\text{Lemma 5.12-1(b)}) \\
\hline
n+3. \forall \xi, \sigma. ((\xi, \sigma) \models C \supset \\
\quad \exists \sigma', W. ((\text{op}(\vec{M}) \xi, \sigma) \Downarrow (W, \sigma') \wedge (\xi \cdot u : W, \sigma') \models C')) \quad (\text{Prop. A.1-10 and Lem. 5.11-2})
\end{array}$$

For [Abs], let $\Gamma \cdot x : \alpha; \Delta \vdash M : \beta$ and $\Gamma \cdot x : \alpha; \Delta \vdash \xi'$ and $\Gamma; \Delta \vdash \xi$.

$$\begin{array}{l}
1. \forall \xi', \sigma. ((\xi', \sigma) \models C \wedge A^{\text{rx}} \supset \exists \sigma', V. ((M \xi', \sigma) \Downarrow (V, \sigma') \wedge (\xi' \cdot m : V, \sigma') \models C')) \quad (\text{prem}) \\
\hline
2. \forall \xi. (\xi \models A \supset \\
\quad \forall W, \sigma. (\xi \cdot x : W, \sigma) \models C \supset \\
\quad \exists \sigma', V. ((M(\xi \cdot x : W), \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot x : W \cdot m : V, \sigma') \models C')) \quad (\text{Lemma 5.11-2}) \\
\hline
3. \forall \xi. (\xi \models A \supset \\
\quad \forall W, \sigma. (\xi \cdot x : W, \sigma) \models C \supset \\
\quad \exists \sigma', V. (((\lambda x. M \xi) W, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot x : W \cdot m : V, \sigma') \models C')) \quad (\text{Prop. A.1-11}) \\
\hline
4. \forall \xi. (\xi \models A \supset \\
\quad \forall W, \sigma. (\xi \cdot x : W, \sigma) \models C \supset \\
\quad \exists \sigma', V. ((([u]_{(\xi u : (\lambda x. M \xi))}([x]_{(x:W)})) \cdot \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot x : W \cdot m : V, \sigma') \models C')) \quad (\text{Definition 5.7}) \\
\hline
5. \forall \xi. (\xi \models A \supset (\xi \cdot u : (\lambda x. M \xi), \sigma) \models \forall x. \{C\} u \bullet x = m \{C'\}) \quad (\text{Definition 5.8})
\end{array}$$

For [App], let $\Gamma; \Delta \vdash M : \alpha \Rightarrow \beta$ and $\Gamma; \Delta \vdash N : \alpha$. Further let $\Gamma \cdot m : \alpha \Rightarrow \beta; \Delta \vdash \xi_0$ and $\Gamma; \Delta \vdash \xi$.

$$\begin{array}{l}
1. \forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. ((M \xi, \sigma) \Downarrow (V, \sigma') \wedge \xi \cdot m : V \models C_0)) \quad (\text{prem}) \\
\hline
2. \forall \xi_0, \sigma. ((\xi_0, \sigma) \models C_0 \supset \\
\quad \exists W, \sigma'. ((N \xi_0, \sigma) \Downarrow (W, \sigma') \wedge \xi_0 \cdot n : W \models C_1 \wedge \{C_1\} m \bullet n = u \{C'\})) \quad (\text{prem}) \\
\hline
3. \forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists U, \sigma'. (((MN) \xi, \sigma) \Downarrow (U, \sigma') \wedge \xi \cdot u : U \models \{C'\})) \quad (1, 2, \text{Definition 5.8})
\end{array}$$

Next for $[If]$, assume $\Gamma; \Delta \vdash \text{if } M \text{ then } N_1 \text{ else } N_2$ and $\Gamma; \Delta \vdash \xi$.

1. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. ((M\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot b : V, \sigma') \models C_0))$ (prem)

2. $\forall \xi, \sigma. ((\xi, \sigma) \models C_0[t/b] \supset \exists V, \sigma'. ((N_1\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot u : V, \sigma') \models C'))$ (prem)

3. $\forall \xi, \sigma. ((\xi, \sigma) \models C_0[f/b] \supset \exists V, \sigma'. ((N_2\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot u : V, \sigma') \models C'))$ (prem)

4. $\forall \xi, \sigma. ((\xi \cdot b : t, \sigma) \models C_0 \supset \exists V, \sigma'. ((N_1\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot u : V, \sigma') \models C'))$
(2, Lemma 5.12-1(a))

5. $\forall \xi, \sigma. ((\xi \cdot b : f, \sigma) \models C_0 \supset \exists V, \sigma'. ((N_2\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot u : V, \sigma') \models C'))$
(3, Lemma 5.12-1(a))

6. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. ((\text{if } M \text{ then } N_1 \text{ else } N_2)\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot u : V, \sigma') \models C'))$
(1, 4, 5, Proposition A.1-8)

For $[In_1]$, let $\Gamma; \Delta \vdash \text{in}_i(M) : \alpha_1 + \alpha_2$ and $\Gamma; \Delta \vdash (\xi, \sigma)$. By Convention 4.1 (2) we have $v \notin \text{fv}(C')$.

1. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. (M\xi, \sigma) \Downarrow V \wedge (\xi \cdot v : V, \sigma') \models C'[\text{in}_1(v)/u])$ (prem)

2. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. (\text{in}_1(M)\xi, \sigma) \Downarrow \text{in}_1(V) \wedge (\xi \cdot v : V \cdot u : (\text{in}_1(V)), \sigma') \models C')$
(Lemma 5.12-1(a))

3. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. (\text{in}_1(M)\xi, \sigma) \Downarrow \text{in}_1(V) \wedge (\xi \cdot u : (\text{in}_1(V))\sigma') \models C')$
(Lemma 5.11-2)

$[Case]$ is similar to $[If]$, while $[Pair]$ and $[Proj]$ are reasoned just as for $[In_1]$ above. These cases are omitted.

For $[Deref]$, assume $\Gamma; \Delta \vdash !x : \alpha$, and let $\Gamma; \Delta \vdash \mathcal{M}$. note $u \notin \text{dom}(\Gamma, \Delta)$. We infer:

1. — (no premise)

2. $\mathcal{M} \models C[!x/u] \supset \mathcal{M} \cdot u : ([!x]\mathcal{M}) \models C$ (Lemma 5.12-1(a))

For $[Assign]$, let $\Gamma; \Delta \vdash x := M : \text{Unit}$ and $\Gamma; \Delta \vdash (\xi, \sigma)$. Noting $m \notin \text{fv}(C')$, we infer:

1. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V. ((M\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot m : V, \sigma') \models C'[m/!x][()/u]))$ (prem)

2. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists V, \sigma'. ((M\xi, \sigma) \Downarrow (V, \sigma') \wedge (\xi \cdot m : V \cdot u : (), \sigma'[x \mapsto V]) \models C'))$
(Lemma 5.12-1(a)/2)

3. $\forall \xi, \sigma. ((\xi, \sigma) \models C \supset \exists \sigma''. ((x := (M\xi), \sigma) \Downarrow ((), \sigma'') \wedge (\xi \cdot u : (), \sigma'' \models C')))$ (Lemma 5.11-2)

For $[Rec]$, let $\Gamma \cdot x : \alpha ; \Delta \vdash \lambda y.M : \alpha$ and let W be as in the above Lemma. Since the state part of the model does not play any role, we only mention the environment part in

the following.

1. $\models \{A^{xi} \wedge \forall j \lesssim i.B(j)[x/u]\} \lambda x.M \{B(i)^{x}\}$ (prem)

2. $\models \{\top\} \lambda x.M :_u \{A^{xi} \supset \forall i^{\text{Nat}}.((\forall j \lesssim i.B(j)[x/u]) \supset B(i)^{x})\}$ ($\wedge\text{-}\supset$, Aux \forall)

3. $\forall \xi^{\Gamma;\Delta}. \forall V^{\alpha}. (\xi \cdot x : V \cdot u : ((\lambda y.M)\xi \cdot x : V) \models A^{xi} \supset \forall i^{\text{Nat}}.((\forall j \lesssim i.B(j)[x/u]) \supset B(i)^{x}))$
(2 expanded)

4. $\forall \xi^{\Gamma;\Delta}. (\xi \cdot x : (W\xi) \cdot u : ((\lambda y.M)\xi \cdot x : (W\xi)) \models A^{xi} \supset \forall i^{\text{Nat}}.((\forall j \lesssim i.B(j)[x/u]) \supset B(i)^{x}))$
(\forall -inst)

5. $\forall \xi^{\Gamma;\Delta}. (\xi \cdot x : (\mu x.\lambda.M\xi) \cdot u : (\mu x.\lambda.M\xi) \models A^{xi} \supset \forall i^{\text{Nat}}.((\forall j \lesssim i.B(j)[x/u]) \supset B(i)^{x}))$
(Proposition A.1-12)

6. $\forall \xi^{\Gamma;\Delta}. (\xi \cdot x : (\mu x.\lambda.M\xi) \cdot u : (\mu x.\lambda.M\xi) \models A^{xi} \supset \forall i^{\text{Nat}}.(\forall j \lesssim i.B(j) \supset B(i)^{x}))$
(Lemma 5.12-1(a))

7. $\forall \xi^{\Gamma;\Delta}. (\xi \cdot u : (\mu x.\lambda.M\xi) \models A^{xi} \supset \forall i^{\text{Nat}}.(\forall j \lesssim i.B(j) \supset B(i)^{x}))$ (Lemma 5.11-2)

8. $\forall \xi^{\Gamma;\Delta}. (\xi \cdot u : (\mu x.\lambda.M\xi) \models A^{xi} \supset \forall i^{\text{Nat}}.B(i)^{x})$ (mathematical induction)

This concludes the proof of Theorem 5.14.

B Observational Completeness: Detailed Proofs

B.1 Supplement to Proof of Lemma 6.5

Assume $\Delta \vdash M_1 \not\cong M_2 : \alpha$ and let $C[\cdot]$ and \vec{V} be such that, for example:

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

which means, through the β_V -equality:

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

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. Leaving details to Appendix B.1, 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 5.2), 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:

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

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. $\eta^{\text{Nat} \Rightarrow \text{Nat}}(y) \stackrel{\text{def}}{=} \lambda x. \text{let } z = yx \text{ in } z$). If $\alpha = \text{Nat}$ then it is identity.
2. $\text{case}^\omega x$ of $\langle i : M_i \rangle$ 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, $\text{succ}(M)$, 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.

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

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

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

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 $\lambda x^{\text{Nat}}.M$. This decides a finite number of numerals ever fed to each abstraction of the form $\lambda x^{\text{Nat}}.M$ in W^\dagger . 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. We are now in the final stage. First observe that, in each subterm of the form $\text{succ}(N)$, N is either a numeral or again of the form $\text{succ}(N')$ (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.

B.2 Proofs for Proposition 6.8

We can verify the three conditions above reduce to the original conditions in Proposition 6.7 when σ is empty. In the subsequent proof of Proposition 6.8, we use the following notations for brevity.

Notation B.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 $C' \stackrel{\text{def}}{=} C \wedge u = n$. MTC is trivial. For closure, assume $\{C_0\}M :_u \{C'\}$ with $C_0 \supset C$ and let $(\xi, \sigma) \models C_0$. Then

$$(\xi \cdot u : M\xi, \sigma) \Downarrow (\xi \cdot u : V, \sigma'_0) \models C' \quad \Rightarrow \quad V \cong n \wedge \sigma \cong \sigma'$$

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

(Case-n) Let $F' \stackrel{\text{def}}{=} \text{case } x \text{ of } \langle n_i : F_i \rangle_i$, $C \stackrel{\text{def}}{=} \bigvee_i (x = n_i \wedge C_i)$ and $C' \stackrel{\text{def}}{=} \bigvee_i (x = n_i \wedge C'_i)$. 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:

$$\begin{aligned} (F'\xi', \sigma) \Downarrow &\Leftrightarrow \bigvee_i (\xi'(x) = n_i \wedge (F_i\xi, \sigma) \Downarrow) \\ &\Leftrightarrow \bigvee_i (\xi'(x) = n_i \wedge (\xi, \sigma) \models A_i) \\ &\Leftrightarrow \xi \models A. \end{aligned}$$

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

$$\models \{E \wedge x = n_i\}M :_u \{C_i\} \tag{B.1}$$

We can now reason:

$$\begin{aligned} (\xi', \sigma) \models E &\Rightarrow \exists i. (\xi' \models E \wedge x = n_i) \\ &\Rightarrow \exists i. (\xi(x) = n_i \wedge F_i\xi \sqsubseteq M\xi) \quad \text{(IH)} \\ &\Rightarrow F'\xi' \cong F_i\xi \sqsubseteq M\xi. \end{aligned}$$

(Omega) Same as in Proposition 6.7.

(Abstraction) Same as in Proposition 6.7, combined with the invariance reasoning given in (Numeral) above.

(Dereference) Let $F' \stackrel{\text{def}}{=} \text{let } x = !y \text{ in } F$ 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.

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

Next we show the closure property.

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

$$\begin{aligned} \models \{C_0\}M\{C'\} \wedge (\xi, \sigma) \models C_0 &\Rightarrow \models \{C \wedge C_0\}M\{C'\} \wedge (\xi', \sigma) \models C \wedge C_0 \\ &\Rightarrow (F'\xi', \sigma) \sqsubseteq (M\xi', \sigma) \\ &\Rightarrow (F', \sigma) \sqsubseteq (M\xi', \sigma) \end{aligned}$$

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

(Assignment) Let

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

Further by induction we stipulate:

(IH1) (C, C') satisfies (MTC) and (closure) w.r.t. F at u ;

(IH2) (T, 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:

$$\forall \xi, \sigma. (\xi \cdot z : U\xi, \sigma) \models A \quad (\text{B.3})$$

Hence also:

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

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 .

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

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

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 :

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

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

$$\{C_0\} M :_u \{C'\}. \quad (\text{B.5})$$

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

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

Fig. 11 Asserted Programs for the Proof of Closure in Assignment Rule

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

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

Immediately:

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

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

$$\{C_0[y/!x]\} x := y \{C_0\} \quad (\text{B.7})$$

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

$$\{C_0[y/!x]\} x := y; M :_u \{C'\} \quad (\text{B.8})$$

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

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

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

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

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

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

(Let-Application) Let $F' \stackrel{\text{def}}{=} \text{let } x = fU \text{ in } F$ $\xi_0 = \vec{y} : \vec{V}$ 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 $\models \{\top\}U :_z \{A\}$ hence for any ξ_0 (omitting auxiliary I):

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

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

$$\begin{aligned} & (F'\xi, \sigma) \Downarrow \\ \Leftrightarrow & (\xi_0 \cdot x : WU, \sigma) \Downarrow (\xi \cdot x : S, \sigma) \models C && \text{(IH1, (B.11))} \\ \Leftrightarrow & \forall U_1 \supseteq U\xi_0 \supset (\xi \cdot x : WU_1, \sigma) \Downarrow (\xi \cdot x : S_1, \sigma'_1) \models C && \text{(C TCA at } x) \\ \Leftrightarrow & z : U_1 \cdot \xi_0 \models A \supset (\xi \cdot x : WU_1, \sigma) \Downarrow (\xi \cdot x : S_1, \sigma'_1) \models C && \text{(IH2)} \\ \Leftrightarrow & z : U_1 \cdot \xi_0 \models A \supset (z : U_1 \cdot \xi \cdot \vec{j} : \vec{V}, \sigma) \models \{!\vec{r} = \vec{j}\} f \bullet z = x \{C\} && \text{(Def-eval)} \\ \Leftrightarrow & (\xi \cdot \vec{j} : \vec{V}, \sigma) \models \{A \wedge !\vec{r} = \vec{j}\} f \bullet z = x \{C\} && \text{(e5)} \\ \Leftrightarrow & (\xi, \sigma) \models \forall \vec{j}. (!\vec{r} = \vec{j} \supset \{A \wedge !\vec{r} = \vec{j}\} f \bullet z = x \{C\}) \end{aligned}$$

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:

$$\{C_0\}M :_u \{C'\}. \tag{B.12}$$

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

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

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 (B.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:

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

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 (B.13) we know: $(\xi \cdot u : V'', \sigma'') \sqsubseteq (\xi \cdot u : V', \sigma')$, as required.

C Derivations of Hoare Rules

This appendix lists the derivations of $[Seq]$, $[IfH]$, $[Call]$ and $[RecProc]$ in Figure 9 in S 7.1 First we derive $[Seq]$ from $[App]$ and $[Abs]$.

1. $\{C_0\} \llbracket Q \rrbracket \{C'\}$	(premise)
2. $\{T\} \lambda z. \llbracket Q \rrbracket :_m \{\{C_0\} m \bullet ()\} \{C'\}$	(1, Abs)
3. $\{C_0\} \lambda z. \llbracket Q \rrbracket :_m \{C_0 \wedge \{C_0\} m \bullet ()\} \{C'\}$	(2, Promote)
4. $\{C\} \llbracket P \rrbracket :_n \{n = () \wedge C_0\}$	(premise)
5. $\{C \wedge \{C_0\} m \bullet ()\} \{C'\} \llbracket P \rrbracket :_n \{C_0 \wedge n = () \wedge \{C_0\} m \bullet ()\} \{C'\}$	(4, Constancy)
6. $\{C \wedge \{C_0\} m \bullet ()\} \{C'\} \llbracket P \rrbracket :_n \{C_0 \wedge \{C_0\} m \bullet n\} \{C'\}$	(5, Consequence)
7. $\{C\} \llbracket P; Q \rrbracket \{C'\}$	(3, 6, App)

$[IfH]$ is derived as follows.

1. $\{C\} e :_b \{b = e \wedge C\}$	(Simple)
2. $\{C \wedge e\} \llbracket P \rrbracket \{C'\}$	(premise)
3. $\{C \wedge e = \mathbf{t}\} \llbracket P \rrbracket \{C'\}$	(2, Consequence)
4. $\{C \wedge \neg e\} \llbracket Q \rrbracket \{C'\}$	(premise)
5. $\{C \wedge e = \mathbf{f}\} \llbracket Q \rrbracket \{C'\}$	(4, Consequence)
6. $\{C\} \text{if } e \text{ then } \llbracket P \rrbracket \text{ else } \llbracket Q \rrbracket \{C'\}$	(3, 5, If)

We next turn to the rules for procedure. $[Call]$ is simple:

1. $\llbracket \Sigma \rrbracket \supset \{C\} p \bullet () \{C'\}$	(premise)
2. $\{\llbracket \Sigma \rrbracket \wedge C\} p :_m \{\llbracket \Sigma \rrbracket [m/p] \wedge C\}$	(Var)
3. $\{\llbracket \Sigma \rrbracket \wedge C\} p :_m \{\{C\} m \bullet () \{C'\} \wedge C\}$	(1, 2, Consequence)
4. $\{\llbracket \Sigma \rrbracket \wedge C\} p :_m \{\{\llbracket \Sigma \rrbracket \wedge C\} m \bullet () \{C'\}\}$	(e7)
5. $\{\llbracket \Sigma \rrbracket \wedge C\} \llbracket \text{call } p \rrbracket \{C'\}$	(Run)

For $[RecProc]$, the recursion rule $[Rec]$ offers a precise account of the induction principle for recursive procedures.

- | | |
|--|--------------|
| 1. $\{[\Sigma] \wedge \{\exists j \lesssim i.C(j)\}p \bullet ()\{C'\} \wedge C(i)\} \llbracket P \rrbracket \{C_0\}$ | (Assumption) |
| 2. $\{[\Sigma] \wedge \forall j^{\text{Nat}}. \{j \lesssim i \wedge C(j)\}p \bullet ()\{C'\} \wedge C(i)\} \llbracket P \rrbracket \{C_0\}$ | (e3) |
| 3. $\{[\Sigma] \wedge \forall j^{\text{Nat}} \lesssim i. \{C(j)\}p \bullet ()\{C'\} \wedge C(i)\} \llbracket P \rrbracket \{C_0\}$ | (e5) |
| 4. $\{[\Sigma] \wedge \forall j^{\text{Nat}} \lesssim i. \{C(j)\}p \bullet ()\{C'\}\} \lambda(). \llbracket P \rrbracket :_m \{\{C(i)\}m \bullet ()\{C_0\}\}$ | (Abs) |
| 5. $\{[\Sigma]\} \mu p. \lambda(). \llbracket P \rrbracket :_m \{\forall i. \{C(i)\}m \bullet ()\{C_0\}\}$ | (Rec) |
| 6. $\{[\Sigma]\} \mu p. \lambda(). \llbracket P \rrbracket :_m \{\{\exists i.C(i)\}m \bullet ()\{C_0\}\}$ | (e3) |
| 7. $\{[\Sigma] \wedge \{\exists i.C(i)\}p \bullet ()\{C_0\} \wedge C\} \llbracket Q \rrbracket \{C'\}$ | (Assumption) |
| 8. $\{[\Sigma]\} \lambda p. \llbracket Q \rrbracket :_n \{\forall p. \{\exists i.C(i)\}p \bullet ()\{C_0\} \supset \{C\}n \bullet p\{C'\}\}$ | (Abs) |
| 9. $\{[\Sigma] \wedge C\} (\lambda p. \llbracket Q \rrbracket) (\mu p. \lambda(). \llbracket P \rrbracket) \stackrel{\text{def}}{=} \llbracket \text{proc } p = P \text{ in } Q \rrbracket \{C'\}$ | (6, 8, App) |

Lines 2 and 3 above use $\exists j \lesssim i.C \stackrel{\text{def}}{=} \exists j. (j \lesssim i \wedge C)$ and $\forall j \lesssim i.C \stackrel{\text{def}}{=} \forall j. (j \lesssim i \supset C)$. Note the induction principle which uses the existential is now understood as the universal quantification in the assumption. Arguably this offers a more direct, and intuitive, view on induction.