A Compositional Program Logic for Imperative Higher-Order Functions

Kohei Honda[1], Nobuko Yoshida[2], and Martin Berger[1]

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

Abstract. Imperative higher-order functions have been one of the main topics in the study of semantics and types of programming languages for decades. They are not only a cornerstone of functional programming languages, but also serves as a central analytical tool for procedural, object-oriented and even low-level languages. In spite of this prominent status, compositional program logics for imperative extensions of the λ-calculus have not been known so far. This paper proposes 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 central feature of the logic is its systematic use of names and operations on them. This allows precise and general description of complex higher-order imperative behaviours involving, for example, stored higher-order procedures. Valid logical descriptions faithfully correspond to the observational congruence of the language. The proof rules of the logic can cleanly embed and justify the standard proof rules for total correctness of Hoare logic with recursive procedure. The use of the logic is demonstrated through non-trivial reasoning examples which have been hard to assert and infer in the existing program logics.

1 Introduction

Imperative higher-order functions have been one of the main 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 [28] and Haskell [2] and central to the analysis of procedural, object-oriented and even low-level languages [1, 11, 29, 32, 36]. In spite of their prominent status, compositional program logics for imperative extensions of the λ-calculus have not been known so far [5, 8].

This paper proposes a simple compositional program logic for an imperative extension of call-by-value PCF, built on Hoare logic [16] and our preceding work on program logics for pure higher-order functions [18, 20]. The program logic can precisely specify behaviours of imperative higher-order procedures as logical assertions up to the observational congruence, as well as enabling us to formally derive these assertions using proof rules which follow the syntactic structure of given programs. As far as we know, this is the first time a compositional assertional method for specifying and proving imperative higher-order functions has been developed. This is made possible by the central feature of the logic, a systematic use of names and operations on them (whose origin lies in the logics for the π-calculus [15, 19, 27]), allowing general and flexible description of complex higher-order imperative behaviours. The logic uses Floyd-Hoare's fundamental observation [9, 16] on logical description of sequential programs, the significance of the initial and final states of imperative variables for describing state change induced by execution of programs.

A syntactically simple extension to the Floyd-Hoare tradition is that assertions in the present logic not only talk about first-order data stored in imperative variables, as in Hoare's logic and its extensions, but also about arbitrary higher-order procedural behaviours, which may be fed as arguments, 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 $\mathtt{twice} \stackrel{\text{def}}{=} \lambda f^{\alpha\Rightarrow\alpha}.\lambda x^{\alpha}.(f(fx))$, with α possibly 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 induce side effects. We may also recall that storing higher-order procedures in imperative variables and using them is a fundamental idiom in system-level programming, which is one of the reasons why imperative λ-calculi are used for analysing low-level programs [11, 29, 36]. Thus incorporation of higher-order behaviour in the universe of discourse is essential, which also means inherent increase in semantic complexity in comparison with the traditional Hoare logic where the first-order universe, essentially the number theory, suffices. While in sheer calculability of validity the standard model for Hoare logic is already intractable, the new setting demands additional considerations including axioms for reasoning, as we shall discuss in subsequent sections.

Let us present two simple, but non-trivial programming examples. A clean and rigorous description of their behaviour by a logical means, as well as their derivations by compositional proof rules, is set to be a technical challenge in our present inquiry. We use the notations from the standard textbooks [12, 32].

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

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

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

Note that, inside the body of $\mathtt{closureFact}$, 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.

As another example, let us consider the following program.

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

where $\mathtt{map}$ is the standard higher-order map function, i.e.

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

where $x{+}{+}y$ 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 of its behaviour would be:

> *Given a list of jobs* $\langle f_1, x_1\rangle, \langle f_2, x_2\rangle, .., \langle f_{n-1}, x_{n-1}\rangle$ *(*$n \geq 1$*), suppose 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 that, 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 logical language to describe such complex behaviour with precision, combined with mechanical proof rules for deriving assertions compositionally. The logical description of programs' behaviour is semantically exact in the sense that valid assertions characterise programs' behaviour up to the observational congruence.

In the remainder, Section 2 illustrates central ideas of the logic informally. Section 3 introduces the target language and the syntax of assertions, and explores the basic axioms for assertions. Section 4 presents and analyses compositional proof rules of the logic, including structural rules. Section 5 defines semantics of the assertions and judgements in the present logic, and proves the soundness of the axioms presented in Section 3 and of the proof rules presented in Section 4. Section 6 explores the use of the logic through reasoning examples, including an embedding of Hoare logic for total correctness (which offers a clean analysis of the standard proof rules for imperative idioms such as assignment, while loop and recursive procedures), as well as illustration of how we can treat two programming examples presented above. Section 7 discusses related work 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 [18, 20].

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, $N \stackrel{\text{def}}{=} \lambda x^{\mathsf{Nat}}.x + x$ where Nat is the type of natural numbers. If we apply 5 to N, 10 is returned. More generally, the result of applying any natural number to N is always even. To represent these behavioural properties using logical formulae, we do not mention N itself in a formula, but rather describe its properties by *naming* it as, say, f. Thus we can write $f \bullet 5 = 10$ as a property of N, named as f. Similarly we can write

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

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

Using these formulae (which we let range over $C, C', D, \ldots$), the judgement of the logic has the following shape.

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

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

$$\{\mathsf{T}\}N :_f \{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\} \tag{2.2}$$

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

$$\{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\} f3 :_u \{Even(u)\}$$

from two small specifications (1) $\{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\} f :_m \{\forall x^{\mathsf{Nat}}.Even(m \bullet x)\}$ (which is an instance of the axiom for variables: f named as m satisfies the same predicate as the one assumed for f, replacing f by m); and (2) $\{\mathsf{T}\}3 :_m m = 3$ ("3" named as m satisfies a predicate $m = 3$) and by combining them together using the entailment $(\forall x^{\mathsf{Nat}}.Even(f \bullet x) \wedge m = 3) \supset Even(f \bullet m)$ (which is an instance of the standard axiom in logic with equality, $(A(x,x) \wedge x = y) \supset A(x,y)$ [24, §2.8]). The same framework works for higher-order programs where multiple higher-order variables share assumptions.

$$\{\forall x^{\mathsf{Nat}}.Even(f \bullet x)\} f3 + f(f5) + 1 :_u \{Odd(u)\} \tag{2.3}$$

where $Odd(n)$ says n is odd. Now by just combining two specifications N in (2.2) and $L \stackrel{\text{def}}{=} f3 + f(f5) + 1$ in (2.3), we can derive:

$$\{\mathsf{T}\}\mathtt{let}\ f = N\ \mathtt{in}\ L :_u \{Odd(u)\} \tag{2.4}$$

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

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

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

where the logical term $!x$ represents dereferencing x [28]. The resulting behaviour is located at the anchor name u. The assertion says: the program $2{+}\,!x$ returns 9 whenever x initially stores 7, and it does not change this content of x. As another example, this time with side-effects, 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 $W \stackrel{\text{def}}{=} \lambda x.(w := (1{+}!w)\,;x+x)$, slightly modified from $\lambda x.x+x$ in the previous paragraph. Recalling L from (2.3), we define $M \stackrel{\text{def}}{=} \mathtt{let}\ f = W\ \mathtt{in}\ L$. Now this function does not only satisfy $Odd(u)$, but also *changes a memory cell* w when invoked. Hence we would expect the following:

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

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

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

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

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

$EvenS(w, f)$ 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 $EvenS(w, f)$ we can derive:

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

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

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

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 ($EvenS(w,f)$) 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 (2.5) in a compositional way, using essentially the same `let`-rule as for stateless functions.

When working with higher-order functions, assertions with pre/post-conditions can be nested 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:

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

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 is that ours can directly assert formulae about the combined behaviour of (possibly higher-order) procedures. Consider the following program:

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

where f is of type $\mathsf{Unit} \Rightarrow \mathsf{Unit}$ and g of $(\mathsf{Unit} \Rightarrow \mathsf{Unit}) \Rightarrow \mathsf{Unit}$. 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:

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

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

3 Logic for Imperative Call-by-Value PCF

3.1 Imperative PCF

Below we briefly review the programming language we shall use in the present study, the call-by-value PCF with unit, sums and products, augmented with imperative variables (henceforth also called *references*). The grammar of programs is standard [32], given below. Assume given an infinite set of *variables* ($x, y, z, \ldots$, also called *names*).

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

The grammar uses types ($\alpha, \beta, \ldots$), which are given later. Constants ($\mathbf{c}, \mathbf{c}', \ldots$) include the unit $()$, natural numbers and booleans. $\mathsf{op}(\vec{M})$ (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 $\vee$. $!M$ dereferences M while $x := N$ is assignment.

The dynamics of the language is given by the standard call-by-value reduction using store [12, 32]. A *store* ($\sigma, \sigma', \ldots$) 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 [12, 32]. We use the left-to-right evaluation. First, the following rules are from call-by-value PCF (omitting obvious symmetric rules and the rules for first-order operators).

$$
\begin{array}{rcl}
(\lambda x.M)V & \rightarrow & M[V/x] \\
\pi_1(\langle V_1, V_2 \rangle) & \rightarrow & V_1 \\
\mathtt{case}\ \mathtt{in}_1(W)\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i \in \{1,2\}} & \rightarrow & M_1[W/x_1] \\
\mathtt{if}\ \mathtt{t}\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 & \rightarrow & M_1 \\
(\mu f.\lambda g.N)W & \rightarrow & N[W/g][\mu f.\lambda g.N/f]
\end{array}
$$

The rules for assignment and dereference are given next. Below $\sigma[x \mapsto V]$ denotes the store which maps x to V and otherwise agrees with σ. In both rules we let $x \in \mathsf{dom}(\sigma)$.

$$
\begin{array}{rcl}
(!x,\ \sigma) & \rightarrow & (\sigma(x),\ \sigma) \\
(x := V,\ \sigma) & \rightarrow & ((),\ \sigma[x \mapsto V])
\end{array}
$$

Finally the contextual rules are given as follows.

$$
\frac{M \rightarrow M'}{(M, \sigma) \rightarrow (M', \sigma)} \qquad \frac{(M, \sigma) \rightarrow (M', \sigma')}{(\mathcal{E}[M], \sigma) \rightarrow (\mathcal{E}[M'], \sigma')}
$$

where $\mathcal{E}[\cdot]$ is the left-to-right evaluation context with eager evaluation for pairs, projection and injection, inductively of the shape (omitting the first-order operators):

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

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 each natural number n.

Fig. 1 Typing Rules for the Core Language

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

$$[\mathit{Num}]\ \frac{-}{\Gamma;\Delta\vdash \mathtt{n}:\mathsf{Nat}}\qquad [\mathit{Eq}]\ \frac{\Gamma;\Delta\vdash M_{1,2}:\mathsf{Nat}}{\Gamma;\Delta\vdash M_1=M_2:\mathsf{Bool}}$$

$$[\mathit{Abs}]\ \frac{\Gamma,x\!:\!\alpha;\Delta\vdash M:\beta}{\Gamma;\Delta\vdash \lambda x^{\alpha}.M:\alpha\Rightarrow\beta}\qquad [\mathit{Rec}]\ \frac{\Gamma,x\!:\!\alpha\Rightarrow\beta;\Delta\vdash \lambda y^{\alpha}.M:\alpha\Rightarrow\beta}{\Gamma;\Delta\vdash \mu x^{\alpha\Rightarrow\beta}.\lambda y^{\alpha}.M:\alpha\Rightarrow\beta}$$

$$[\mathit{App}]\ \frac{\Gamma;\Delta\vdash M:\alpha\Rightarrow\beta\quad \Gamma;\Delta\vdash N:\alpha}{\Gamma;\Delta\vdash MN:\beta}$$

$$[\mathit{If}]\ \frac{\Gamma;\Delta\vdash M:\mathsf{Bool}\quad \Gamma;\Delta\vdash N_i:\alpha_i\ (i=1,2)}{\Gamma;\Delta\vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2:\alpha}\qquad [\mathit{Inj}]\ \frac{\Gamma;\Delta\vdash M:\alpha_i}{\Gamma;\Delta\vdash \mathtt{in}_i(M):\alpha_1+\alpha_2}$$

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

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

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

The grammar of types is also standard [12, 32].

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

We call $\alpha,\beta,\ldots$ *value types*, and $\mathsf{Ref}(\alpha),\ldots$ *reference types*. Note reference types are never carried by other types. See Remark 1 below for a discussion on this point.

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

The typing rules are standard [32], listed in Figure 1. We write $\Gamma;\Delta\vdash M:\rho$ when M has type α under Γ and Δ, always assuming $\mathsf{dom}(\Gamma)\cap\mathsf{dom}(\Delta)=\emptyset$. We extend the typing to stores, writing $\Gamma;\Delta\vdash\sigma$ if $\mathsf{dom}(\Delta)=\mathsf{dom}(\sigma)$ and, moreover, for each $x\in\sigma$, we have $\Gamma;\Delta\vdash\sigma(x):\alpha$ iff $\Delta(x)=\mathsf{Ref}(\alpha)$. Accordingly, we restrict the reduction relation so that, in $(M,\sigma)\longrightarrow(M',\sigma')$, we have $\Gamma;\Delta\vdash M:\rho$ and $\Gamma;\Delta\vdash\sigma$. From now on we only consider well-typed programs and reduction.

Remark 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 leads to a distinct class of behaviours which deserve treatment on their own right, as discussed in the sequel to the present paper.

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

Definition 2. (semi-closed programs [25]) $\Gamma;\Delta \vdash M : \alpha$ is *semi-closed* (resp. *closed*) when $\mathsf{dom}(\Gamma) = \emptyset$ (resp. $\mathsf{dom}(\Gamma) = \Delta = \emptyset$), often written $\Delta \vdash M : \alpha$ (resp. $\vdash M : \alpha$). The notation $\Delta \vdash \sigma$ is understood similarly.

Proposition 3. *Let* $\Delta \vdash M : \alpha$ *and* $\Delta \vdash \sigma$. *Then either* $(M, \sigma) \Uparrow$ *or* $(M, \sigma) \Downarrow$.

Proof. By mechanical structural induction. $\square$

We shall later use the following syntactic properties of semi-closed programs. Below $M \stackrel{\text{def}}{=} N$ indicates M and N are definitionally equal.

Proposition 4.

1. (unit) *Let* $\Delta \vdash V : \mathsf{Unit}$. *Then* $V \stackrel{def}{=} ()$.
2. (boolean) *Let* $\Delta \vdash V : \mathsf{Bool}$. *Then either* $V \stackrel{def}{=} \mathbf{t}$ *or* $V \stackrel{def}{=} \mathbf{f}$.
3. (numerals) *Let* $\Delta \vdash V : \mathsf{Nat}$. *Then* V *is a numeral.*
4. (first-order operators) *Let* $\Delta \vdash \mathsf{op}(\vec{V}) : \alpha$ *(in such a case the program is in fact closed). Then for an arbitrary* $\Delta \vdash \sigma$, *we have* $(\mathsf{op}(\vec{V}), \sigma) \Downarrow$.
5. (abstraction and recursion) *Let* $\Delta \vdash W : \alpha \Rightarrow \beta$. *Then either* $W \stackrel{def}{=} \lambda x^{\alpha}.M$ *for some* x *and* M *or* $W \stackrel{def}{=} \mu x^{\alpha \Rightarrow \beta}.V$ *for some* x *and* V.
6. (product) *Let* $\Delta \vdash W : \alpha \times \beta$. *Then* $W \stackrel{def}{=} \langle V_1, V_2 \rangle$ *for some* V_1 *and* V_2.
7. (sum) *Let* $\Delta \vdash W : \alpha + \beta$. *Then* $W \stackrel{def}{=} \mathtt{in}_i(V)$ *for some* $i \in \{1, 2\}$ *and* V.
8. (reference) *Let* $\Delta \vdash M : \mathsf{Ref}(\alpha)$. *Then* $M = x$ *such that* $\Delta(x) = \mathsf{Ref}(\alpha)$.

Proof. (1–3) are because values of atomic types are either constants or variables by definition. (4–7) are also direct from the definition of values and reduction rules. $\square$

3.2 Terms, Formulae and Judgement

Terms and Formulae. The logical language is that of first-order logic with equality [24, § 2.8] together with an assertion for the evaluation of stateful expressions. The grammar of terms and formulae follows.

$$
\begin{array}{lcl}
e & ::= & x^{\rho} \mid () \mid \mathbf{n} \mid \mathbf{b} \mid \mathbf{op}(\vec{e}) \mid \langle e, e' \rangle \mid \pi_i(e) \mid \mathsf{inj}_i^{\alpha+\beta}(e) \mid\ !e \\
C & ::= & e = e' \mid \neg C \mid C \wedge C' \mid C \vee C' \mid C \supset C' \mid \forall x^{\alpha}.C \mid \exists x^{\alpha}.C \mid \{C\}\ e \bullet e' \searrow x\ \{C'\}
\end{array}
$$

The first set of expressions (ranged over by $e, e', \ldots$) are *terms* while the second set are *formulae* (ranged over by $A, B, C, C' \ldots$). Terms, which are from the logics for pure functions studied in [18, 20] except for $!x$, include all the constants 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. $\mathsf{fv}(e)$ denotes the free variables occurring in e.

The predicate $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ is called *evaluation formula*, where the name x binds its free occurrences in C'. Intuitively, $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ asserts that an invocation of e with an argument e' under the initial state C terminates with a final state and a

Fig. 2 Typing Rules for Terms and Formulae

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

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

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

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

resulting value, named as u, both described by C'. $\bullet$ is non-commutative. Note we can assert divergence by negating evaluation formulae.[3]

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 $\Gamma;\Delta$. We shall often write $\Theta \vdash C$ if C is well-typed under Θ where $\Theta, \Theta', \ldots$ range over finite maps combining two kinds of bases.

Convention 5. Hereafter we only consider well-typed terms and formulae and often omit type annotations. Formulae are often called *assertions*.

Convention 6. 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 $((\neg A) \wedge B) \supset (((\forall x.C) \vee D) \supset E)$. $C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$, the logical equivalence of C_1 and C_2. We use truth T (definable as $1 = 1$) and falsity F (which is $\neg\mathsf{T}$). The standard binding convention is always assumed. $\mathsf{fv}(C)$ denotes the set of *free variables* in C. $\{C\}\ e_1 \bullet e_2 \searrow e'\ \{C'\}$ with e' not a variable, stands for $\{C\}\ e_1 \bullet e_2 \searrow x\ \{x = e' \wedge C'\}$ with x fresh; and $\{C\}\ e_1 \bullet e_2\ \{C'\}$ for $\{C\}\ e_1 \bullet e_2 \searrow ()\ \{C'\}$.

Example 7. (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 $\mathsf{Ref}(\mathsf{Nat})$. Thus we have $y{:}\mathsf{Ref}(\mathsf{Nat}) \vdash\ !y = 6$.
2. The assertion $\mathit{Double}(u) \stackrel{\text{def}}{=} \forall n^{\mathsf{Nat}}.\{\mathsf{T}\}\, u \bullet n \searrow 2 \times n\, \{\mathsf{T}\}$ says that a procedure named u with type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ always returns double of the argument. This is satisfied by, for example, $\lambda z.2 \times z$. The predicate is typed as: $u{:}\mathsf{Nat} \Rightarrow \mathsf{Nat} \vdash \mathit{Double}(u)$.

[3] In the logics for pure functions studied in [18, 20], terms include $\perp^{\alpha}$ 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 is sometimes convenient.

3. Using the notation in (2) above, $Double(!x) \stackrel{\text{def}}{=} \forall n^{\mathsf{Nat}}.\forall n^{\mathsf{Nat}}.\{\mathsf{T}\}\, !x \bullet n \searrow 2 \times n\, \{\mathsf{T}\}$ says that x stores a doubling function. Note we have $x{:}\mathsf{Ref}(\mathsf{Nat}{\Rightarrow}\mathsf{Nat}) \vdash Double(!x)$. This assertion is satisfied, for example, after $x := \lambda z.2 \times z$ is executed.
4. The assertion $\{Double(!x)\}\, u \bullet 3 \searrow 6\, \{Double(!x)\}$ says that, if 3 is fed to the function named by u under the initial state $Double(!x)$, then the result is 6, with the same final state. We have a typing

$$u{:}\mathsf{Nat}{\Rightarrow}\mathsf{Nat}\ ;\ x{:}\mathsf{Ref}(\mathsf{Nat})\ \vdash\ \{Double(!x)\}\, u \bullet 3 \searrow 6\, \{Double(!x)\}$$

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.
5. The assertion

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

says that an imperative variable x stores a function which, when invoked, would increment w as well as returning the double of the argument. For typing, we have

$$\emptyset\ ;\ x{:}\mathsf{Ref}(\mathsf{Nat}{\Rightarrow}\mathsf{Nat}), w{:}\mathsf{Ref}(\mathsf{Nat}) \vdash C.$$

This is satisfied when, for example, $f(w) \stackrel{\text{def}}{=} \lambda z.(w := !w + 1; z \times 2)$ is stored in x.
6. Using C in (4) above, let:

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

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

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

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

Substitution and Avoidance of Name Capture. In first-order logic as well as the predicate calculus with equality, capture-avoiding syntactic substitutions play a fundamental role in inferences. Due to the existence of evaluation formulae, some care is needed for their usage in the present theory. For example consider the following assertion:

$$C\ \stackrel{\text{def}}{=}\ m = 0\ \wedge\ \forall i.\{m = 0 \wedge\ !y = i\}f \bullet ()\{!y = i+1\} \tag{3.1}$$

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 (the condition $m = 0$ in the precondition of the evaluation formula is already satisfied so is equivalent to T). Now suppose we wish to substitute $!y$ for m. Then we obtain:

$$C[!y/m]\ \stackrel{\text{def}}{=}\ !y = 0\ \wedge\ \forall i.\{!y = 0 \wedge\ !y = i\}f \bullet ()\{!y = i+1\} \tag{3.2}$$

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 (3.1) is insignificant, so we in fact have:

$$C \;\equiv\; m = 0 \wedge \forall i.\{!y = i\} f \bullet ()\{!y = i+1\} \tag{3.3}$$

which shows (3.2) does not give us what we naturally expect to have. This is because of the use of universal quantification in the semantics of evaluation formulae, as formally discussed in Section 5. For this reason we strengthen the notion of a term being "free for x in C" (cf.[24, §2.1]) in the standard quantification theory in the following way.

Definition 8. A term e^{α} is *free for x^{α} in C* when:

1. x does not occur freely in the scope of a quantification in C which binds a name that occurs in e.
2. x does not occur freely in the pre/post conditions of evaluation formulae in C whenever e contains a dereference.

In the standard quantification theory, we can always rename bound variables to avoid capture of names. In the present case, we can in addition use existential quantification to "flush out" all names in dangerous positions. As an example, we can transform C as:

$$C' \;\stackrel{\text{def}}{=}\; \exists z.(m = 0 \;\wedge\; \forall i.\{z = 0 \wedge !y = i\} f \bullet ()\{!y = i+1\} \;\wedge\; z = m) \tag{3.4}$$

on which we can safely perform the previous substitution:

$$C'[!y/m] \;\stackrel{\text{def}}{=}\; \exists z.(!y = 0 \;\wedge\; \forall i.\{z = 0 \wedge !y = i\} f \bullet ()\{!y = i+1\} \;\wedge\; z = !y) \tag{3.5}$$

without causing a semantic hazard. Except for using the notion of freedom from capture in Definition 8 and, in correspondence, the implicit transformation as above, the substitution operation itself is defined in the standard way.

To treat assignment, we also use a substitution for a dereferenced name, written $C[e/!x]$. In this case, the substitution should not affect the pre/post conditions of evaluation formulae in C (detailed analysis of this point will be given in the sequel to the present paper). Thus we define, assuming $e'[e/!x]$ is a literal substitution and letting $\star \in \{\wedge, \vee, \supset\}$ and $Q \in \{\forall, \exists\}$:

$$\begin{array}{rcl}
(e_1 = e_2)[e/!x] & = & e_1[e/!x] = e_2[e/!x] \\
(C_1 \star C_2)[e/!x] & = & C_1[e/!x] \star C_2[e/!x] \\
(\neg C)[e/!x] & = & \neg(C[e/!x]) \\
(Qy^{\alpha}.C)[e/!x] & = & (Qy^{\alpha}.(C[e/!x]) \\
(\{C\} e_1 \bullet e_2 \searrow z\{C'\})[e/!x] & = & \{C\}(e_1[e/!x]) \bullet (e_2[e/!x]) \searrow z\{C'\}
\end{array}$$

Because $[e/!x]$ never affects pre/post conditions of evaluation formulae, the only capture of names we need to consider is the one induced by quantifiers. Thus we stipulate:

Definition 9. A term e^{α} is *free for* $!(x^{\mathsf{Ref}(\alpha)}$ *in C* when x does not occur freely in the scope of a quantification in C which binds a name that occurs in e.

Based on the above definitions, whenever we write $C[e/!x]$ we assume e is free for $!x$ in C by an appropriate alpha-conversion to be done beforehand.

Judgement. Following Hoare [16], a judgement in the present program logic consists of a pair of formulae and a program, which takes the following shape.

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

This sequent is used for both validity and provability. If we wish to be specific, we prefix it with either $\vdash$ (for provability) or $\models$ (for validity), In $\models\{C\}M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, $M^{\Gamma;\Delta;\alpha}$ is the *subject* of the judgement; u its *anchor*, which should not be in $\mathsf{dom}(\Gamma,\Delta)\cup\mathsf{fv}(C)$; C its *pre-condition*; and C' its *post-condition*. Henceforth we only treat judgements which are well-typed in the following sense.

Definition 10. We say $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is *well-typed* iff (1) $\Gamma;\Delta \vdash M : \alpha$ and (2) $\Gamma,\Delta,\Theta \vdash C$ and $u\!:\!\alpha,\Gamma,\Delta,\Theta \vdash C'$ for some Θ such that $\mathsf{dom}(\Theta)\cap(\mathsf{dom}(\Gamma,\Delta)\cup\{u\}) = \emptyset$.

Remark 11. The program in a judgement always has a value type i.e. it is never a reference name. Because of Proposition 4 (8) and because of the semantics of reference names formalised in Section 5, this does not let us lose expressiveness of the logic.

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

Definition 12. (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 $\mathsf{dom}(\Gamma,\Delta)\cup\{u\}$. The *auxiliary names* in this judgement are those free names in C and C' which are not primary.

Example 13. In a judgement "$\{x = i\}2\times x^{x:\mathsf{Nat};\mathsf{Nat}} :_u \{u = 2\times i\}$", x and u are primary while i is auxiliary. u is in addition its anchor.

For brevity, we often omit the typing of the subject program from a judgement, writing $\{C\}\ M :_u \{C'\}$. This convention is sound as far as there exists some typing which can type a given judgement, and will be later justified semantically.

Intuitively, $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ says that: *if M is closed by values satisfying C (for non-reference variables), and is evaluated starting from a store satisfying C (for reference variables), then it terminates so that the final state and the resulting value, named u, together satisfy C'.* This informal reading will be made precise in Section 5.

Remark 14. (pre/post conditions) Following Hoare Logic and its extensions, we call C and C' in $\{C\}\ M^{\Gamma;\Delta;\alpha;} :_u \{C'\}$ *pre/post-conditions*. However these formulae describe not only state change but also stateless properties, which differs from the standard Hoare logic. Another difference is direct description of behaviours inside assertions: Free non-reference variables as well as an anchor may denote a complex behaviour, similarly imperative variables may as well store complex behaviours, all described in C and C' (for a program which does not immediately change state, for example when a program is a value, then the part of assertion on state change carries no information, though is harmless).

Fig. 3 Axioms for Data Types and Evaluation Formulae

(t1)	T	$\supset$	$x^{\mathsf{Unit}} = ()$
(t2)	T	$\supset$	$x^{\mathsf{Bool}} = \mathbf{t} \vee x^{\mathsf{Bool}} = \mathbf{f}$
(t3)	T	$\supset$	$\mathbf{t} \neq \mathbf{f}$
(t4)	T	$\equiv$	$\pi_i(x_1, x_2) = x_i$
(t5)	T	$\equiv$	$(\pi_1(x), \pi_2(x)) = x$
(t6)	$\mathtt{in}_i(x) = \mathtt{in}_i(y)$	$\supset$	$x = y$
(t7)	T	$\supset$	$\mathtt{in}_1(x) \neq \mathtt{in}_2(y)$
(t8)	T	$\supset$	$x^{\mathsf{Ref}(\alpha)} \neq y^{\mathsf{Ref}(\alpha)}$
(e1)	$\{C_1\} x \bullet y \searrow z \{C\} \wedge \{C_2\} x \bullet y \searrow z \{C\}$	$\equiv$	$\{C_1 \vee C_2\}\, x \bullet y \searrow z\, \{C\}$
(e2)	$\{C\} x \bullet y \searrow z \{C_1\} \wedge \{C\} x \bullet y \searrow z \{C_2\}$	$\equiv$	$\{C\}\, x \bullet y \searrow z\, \{C_1 \wedge C_2\}$
(e3)	$\{\exists w^{\alpha}.C\}\, x \bullet y \searrow z\, \{C'^{\text{-}x}\}$	$\equiv$	$\forall w^{\alpha}.\{C\}\, x \bullet y \searrow z\, \{C'\}$
(e4)	$\{C^{\text{-}x}\}\, x \bullet y \searrow z\, \{\forall w^{\alpha}.C'\}$	$\equiv$	$\forall w^{\alpha}.\{C\}\, x \bullet y \searrow z\, \{C'\}$
(e5)	$\{A \wedge C\}\, x \bullet y \searrow z\, \{C'\}$	$\equiv$	$A \supset \{C\}\, x \bullet y \searrow z\, \{C'\}$
(e6)	$\{C\}\, x \bullet y \searrow z\, \{A^{\text{-}z} \supset C'\}$	$\supset$	$A \supset \{C\}\, x \bullet y \searrow z\, \{C'\}$
(e7)	$\{C\} x \bullet y \searrow z \{C'\}$	$\supset$	$\{C \wedge A\} x \bullet y \searrow z \{C' \wedge A\}$
(e8)	$\{C_0\} x \bullet y \searrow z \{C'_0\}$	$\supset$	$\{C\}\, x \bullet y \searrow z\, \{C'\}$ when $C \supset C_0$ and $C'_0 \supset C'$.
(ext)	$\mathsf{Ext}^{\Delta;\alpha \Rightarrow \beta}(x, y)$	$\supset$	$x = y$

3.3 Axioms and Rules for Validity

In the proof rules presented in the next section, one often needs to calculate validity of assertions. Formally, a formula C is *valid* if it is true under arbitrary models (the class of models we consider will be discussed in Section 5). Practically, we can often calculate validity syntactically. As is standard [16], we shall freely use axioms, rules and theorems from:

- The propositional calculus;
- First-order logic with equality [24, §2.8] (which includes, among others, the substitutivity rule, $x = y \wedge C(x,x) \supset C(x,y)$ (where $C(x,y)$ rewrites some of occurrences of x to y in $C(x,x)$ [24, §2.8]).
- The formal number theory (Peano arithmetic).

There are also natural axioms for data types (in addition to natural numbers), as well as those for evaluation formulae, the main ones out of which we shall discuss in this subsection. We use the following notion, which plays an important role from now on.

Definition 15. (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. $A, B, \ldots$ range over stateless formulae.

Example 16. (stateless formulae)

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

Note a stateless formula can use a stateful formula inside its evaluation formula. For distinction, we sometimes call well-typed formulae in general, *stateful formulae*. We also use the following notation.

Convention 17. $C^{\text{-}\vec{x}}$ is C in which no name from $\vec{x}$ freely occurs.

We list basic axioms for data types (except for natural numbers) and those for evaluation formulae in Figure 3, where variables of value types can be freely instantiated into arbitrary terms of the corresponding types through universal quantification/instantiation.

The rules from (t1) to (t7) are about data types. (t1) is equivalent to:

$$C[()/x^{\mathsf{Unit}}] \equiv C \tag{3.6}$$

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.

$$(C \equiv C[\mathbf{t}/x^{\mathsf{Bool}}]) \vee (C \equiv C[\mathbf{f}/x^{\mathsf{Bool}}]) \tag{3.7}$$

Further (t2) justifies the use of boolean terms as formulae, via the following "macros". Below let $\star \in \{\wedge, \vee, \supset\}$.

$$\underline{e^{\mathsf{Bool}}} \stackrel{\text{def}}{\equiv} e = \mathbf{t} \tag{3.8}$$

$$\underline{\neg e} \equiv \neg \underline{e} \equiv e = \mathbf{f} \tag{3.9}$$

$$\underline{e_1 \star e_2} \equiv \underline{e_1} \star \underline{e_2} \tag{3.10}$$

$$e_1 = e_2 \equiv (\underline{e_1} \equiv \underline{e_2}) \tag{3.11}$$

The axioms from (3.9) to (3.11) are easily derivable from (3.8). Note these translations are possible because all boolean terms denote total values (cf. footnote 3), page 10). The use of boolean terms as formulae is often convenient in practice, so we stipulate:

Convention 18. Henceforth we often treat boolean terms as formulae, implicitly assuming the conversion (3.8) above.

(t4) and (t5) are the standard extensional axioms for pairs and projections (which also shows projection is definable from pairing and vice versa), while (t6) and (t7) are also standard axioms for injections. (t8) says we treat reference names as constants (which we shall later substantiate semantically). Since, by the typing, reference-typed terms do not appear as proper part of any term except as dereferenced reference names (i.e. of the form $!x$), and because the axiom (t8) says $x = x$ is all we can say about non-dereferenced reference names. Thus, in effect, there is no need to have reference names in formulae except as their dereferencing.

The rules from (e1) to (e8) are about evaluation formulae. (e1) and (e2) should read naturally. From these rules we obtain:

$$\{C_1\}x\bullet y\searrow z\{C_1'\} \ \wedge\ \{C_2\}x\bullet y\searrow z\{C_2'\} \ \supset\ \{C_1\wedge C_2\}\, x\bullet y\searrow z\ \{C_1'\wedge C_2'\} \quad (3.12)$$

$$\{C_1\}x\bullet y\searrow z\{C_1'\} \ \wedge\ \{C_2\}x\bullet y\searrow z\{C_2'\} \ \supset\ \{C_1\vee C_2\}\, x\bullet y\searrow z\ \{C_1'\vee C_2'\} \quad (3.13)$$

(e3) and (e6) are also naturally read. In (e6), the converse (the logical equivalence) does not generally hold. As an example, consider the following inference assuming the converse of (e6) were stipulated (as is standard, the inference as below stands for a sequence of implications).

$$\begin{array}{lll} \mathsf{T} & \supset & \mathsf{F}\supset\{\mathsf{T}\}x^{\alpha\Rightarrow\beta}\bullet y^{\alpha}a\searrow z\{\mathsf{T}\} \\ & \supset & \{\mathsf{T}\}x^{\alpha\Rightarrow\beta}\bullet y^{\alpha}\searrow z\{\mathsf{F}\supset\mathsf{T}\} \qquad \text{(the converse of (e6))} \\ & \supset & \{\mathsf{T}\}x^{\alpha\Rightarrow\beta}\bullet y^{\alpha}\searrow z\{\mathsf{T}\} \end{array}$$

The hypothetical inference claims any programs of type $\alpha\Rightarrow\beta$ will converge for any argument of type α, an absurd statement. Note this statement is *not* absurd if $\{\mathsf{T}\}x^{\alpha\Rightarrow\beta}\bullet y^{\alpha}\searrow z\{\mathsf{T}\}$ does not demand $x\bullet y$ should converge.

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. If we restrict the formulae concerned to stateless ones, we obtain the following variant, internalising the conjunction.

$$(A\supset A_0)\wedge\{C\wedge A_0\}x\bullet y\searrow z\{C'\wedge B_0\}\wedge(B_0\supset B)\supset\{C\wedge A\}x\bullet y\searrow z\{C'\wedge B\} \quad (3.14)$$

The final axiom, (ext), is about extensionality. The rule uses the following formula. Below the vector notations are understood naturally.

Definition 19. (extensionality formulae) Let $\Delta=\vec{r}:\mathsf{Ref}(\vec{\gamma})$.

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

The predicate expresses an extensional equality of two procedures. For simplicity let $\Delta=r:\mathsf{Ref}(\gamma)$. The predicate says: 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 also the predicate entails coincidence in divergence since if x diverges at a certain argument and at a certain state, and if y converges at the same argument and state, then the latter means x should also converge by the predicate, a contradiction. Note the "sameness" in this context means the identity at the level of abstract behaviours, or, more concretely, equality of two behaviours up to contextual congruence.

These axioms will be justified through the semantics of assertions in Section 5.

Remark 20. (capture of derefences) Because stores in evaluation formulae are implicitly universally quantified, the substitution $C[e/x]$ avoids "capture of names" by assuming, when e contains a dereference, that x does not occur in evaluation formulae in C. If not, the substitution is undefined. This does not lead to loss of expressive power.

Fig. 4 Proof Rules

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

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

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

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

$$[In_1]\ \frac{\{C\}\ M :_v \{C'[\mathtt{in}_1(v)/u]\}}{\{C\}\ \mathtt{in}_1(M) :_u \{C'\}} \qquad [Case]\ \frac{\{C^{\text{-}\vec{x}}\}\ M :_m \{C_0^{\text{-}\vec{x}}\} \quad \{C_0[\mathtt{in}_i(x_i)/m]\}\ M_i :_u \{C'^{\,\text{-}\vec{x}}\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

$$[Pair]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\} \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1, m_2\rangle/u]\}}{\{C\}\ \langle M_1, M_2\rangle :_u \{C'\}} \qquad [Proj_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

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

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

$$[Promote]\ \frac{\{A\}\ V :_u \{B\}}{\{A \wedge C\}\ V :_u \{B \wedge C\}} \qquad [Consequence]\ \frac{C \supset C_0 \quad \{C_0\}\ M :_u \{C_0'\} \quad C_0' \supset C'}{\{C\}\ M :_u \{C'\}}$$

4 Proof Rules

4.1 Proof Rules (1): Main Rules

The main proof rules of the program logic are given in Figure 4. When we derive a judgement for a program using these proof rules, the derivation precisely follows the structure of the program, except for the intermittent use of structural rules (in which only formulae are manipulated). In addition to Conventions 6 and 17, we stipulate:

Convention 21. (convention on the use of names)

- $i, j, \ldots$ that freely occur in a formula exclusively range over auxiliary names (in a given judgement).

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

In the following we illustrate the rules in Figure 4 one by one.

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

$$\frac{-}{\{x=3\}\ x:_u \{u=3\}} \tag{4.1}$$

Another simple example is

$$\frac{-}{\{x=x\}\ x:_u \{u=x\}} \tag{4.2}$$

Since $x=x$ is equivalent to T, we can conclude $\{\mathsf{T}\}x:_u \{u=x\}$ from (4.1) and the consequence rule given later, which is reasonable.

[*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). For example, to say 2 is even is fine:

$$\frac{-}{\{Even(2)\}\ 2:_u\ \{Even(u)\}} \tag{4.3}$$

Again we can check $Even(2)$ is always true, so that we can conclude with the judgement $\{\mathsf{T}\}\ 2:_u\ \{Even(u)\}$, which does read as a reasonable assertion. If however we wish to say 2 is odd, we get:

$$\frac{-}{\{Odd(2)\}\ 2:_u \{Odd(u)\}} \tag{4.4}$$

Since $Odd(2)$ is always false, we are assuming something impossible in the pre-condition, so that concluding absurdly in the post-condition is permitted (semantic justification of these arguments will be discussed in Section 5).

[*Op*] applies to standard first-order operators (which are assumed to be effect-free except evaluation of arguments). In the nullary case, the premise becomes $C \stackrel{\text{def}}{=} C'[\mathsf{c}/u]$, so that the rule is equivalent to [*Const*]. Note also the rule assumes op appears as a term constructor in the logical language. A simple instance of this rule is the rule for the addition operator.

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

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

$$\frac{\{Odd(x)\}\ x:_m \{Odd(m)\}\quad \{Odd(m)\}\ 1:_n \{Even(u)[m+n/u]\}}{\{Odd(x)\}\ x{+}1\ :_u\ \{Even(u)\}} \tag{4.5}$$

[*Abs*] says that if, under the assumptions C and A (which is about stateless behaviours and does not mention x), M named m returns/reaches a result and a state which together satisfy C', then $\lambda x.M$ named as u satisfies the same property as M, in the sense that, 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).

$$\frac{\{\mathsf{T} \wedge z=1\}\ !y=x+z\ :_m \{m=() \wedge !y=x+1\}}{\{z=1\}\ \lambda x.!y=x+z\ :_u \{\ \forall x^{\mathsf{Nat}}.\{\mathsf{T}\}\, u \bullet x \{!y=x+1\}\ \}} \tag{4.6}$$

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:

$$\frac{\{\mathsf{T} \wedge z=1\}\ !y=x+z\ :_m \{m=() \wedge !y=x+1\}}{\{\mathsf{T}\}\ \lambda x.\ !y=x+z\ :_u \{\ \forall x^{\mathsf{Nat}}.\{z=1\}\, u \bullet x \{!y=x+1\}\ \}} \tag{4.7}$$

which is semantically equivalence to (4.6) (we later discuss how we can formally mediate (4.6) and (4.7)).

[*App*] says that: if (1) Starting from C, a program M is evaluated into a value (function) 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 $\{C_1\}m \bullet n \searrow z\{C'\}$ (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

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

Again we assume the premises are already inferred.

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

Note how stateless formulae in the premise are turned into stateful pre/postconditions.

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

$$\frac{\begin{array}{l}\{x=\mathbf{t}\}\ x :_b\ \{b=\mathbf{t}\}\\ \{\mathbf{t}=\mathbf{t}\}\ y:=0\ :_u\ \{u=() \wedge !y=0\}\\ \{\mathbf{f}=\mathbf{t}\}\ y:=1\ :_u\ \{u=() \wedge !y=0\}\end{array}}{\{x=\mathbf{t}\}\ \mathtt{if}\ x\ \mathtt{then}\ y:=0\ \mathtt{else}\ y:=1\ :_u\ \{u=() \wedge !y=0\}} \tag{4.8}$$

In (4.8), the last judgement in the antecedent, $\{\mathbf{f}=\mathbf{t}\}\ y:=1\ :_u\ \{u=() \wedge !y=0\}$, holds since, under the absurd assumption $\mathbf{f}=\mathbf{t}$, we can conclude anything.

From **[*Abs*]** and **[*App*]**, the proof rule for the let construct is derivable (the derivation is shown in Section 6.1 for a slightly simpler case), which we list in Figure 5. **[*Let*]** states that, if, starting from C, the evaluation of M, named x, terminates and reaches a state and

Fig. 5 Proof Rule for Non-Recursive Let

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

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 $\mathtt{let}\ x = M\ \mathtt{in}\ N$, 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:

$$\frac{\begin{array}{c}\{\,!y = 0\}\ !y := !y+1\ ;\ !y\ :_x \{!y = 1 \wedge x = 1\} \\ \{\,!y = 1 \wedge x = 1\}\ !y := !y+x\ ;\ !y+1\ :_u \{!y = 2 \wedge u = 3\}\end{array}}{\{!y = 0\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u\ \{!y = 2 \wedge u = 3\}} \tag{4.9}$$

[*Let*] has the close similarity with the standard rule for sequencing in Hoare Logic [16]. In fact, when x in the [*Let*] rule above is typed with Unit, the variable does not carry information, so that the rule is reduced to the rule for sequencing. Write $\{C\}M\{C'\}$ for $\{C\}M :_x \{C_0^{-x}\}$ when M has type Unit (which in fact loses no information by the logical equivalence between $C[()/x^{\mathsf{Unit}}]$ and C, cf. (3.6), §3.3, page 15). We then obtain the following proof rule, assuming M and N have the unit type.

$$[\mathit{Seq}]\frac{\{C\}\ M\ \{C_0\} \quad \{C_0\}\ N\ \{C'\}}{\{C\}\ M;N\ \{C'\}} \tag{4.10}$$

[*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).

$$\frac{\{\mathsf{T}\}\ () :_v\ \{\mathtt{in}_1(v) = \mathtt{in}_1(())\}}{\{\mathsf{T}\}\ \mathtt{in}(()) :_u\ \{u = \mathtt{in}_1(())\}} \tag{4.11}$$

[*Deref*] is understood as **[*Var, Const*]**, showing, dereferenced imperative variables and program variables work in similar ways. In the rule, we assume $!x$ is free for m in C (cf. Definition 8, page 12, §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:

$$\frac{-}{\{\mathit{Even}(!x)\}\ !x\ :_u\ \{\mathit{Even}(u)\}} \tag{4.12}$$

[*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 free for $!x$ in C (for precise definitions, see Definition 9, page 12, §3.2); whereas $[()/u]$ is the standard substitution of $()$ for the anchor 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 $()$ of type Unit). Below we show a simple example of the use of [*Assign*].

$$\frac{\{\ !y = 1 \wedge\ !x = 0\ \}\ \ !y+1\ \ :_m \{\ Even(m)[v/!x]\ \}}{\{\ !y = 1 \wedge\ !x = 0\}\ y\ :=\ !y+1\ \ :_u \{Even(!y) \wedge\ !x = 0\}} \tag{4.13}$$

Later we shall see how we can derive the standard assignment rule in Hoare Logic [16] from the above rule.

[*Rec*] is the proof rule for the total correctness of recursion (slightly strengthening the rule in [18, 20]). i in the premise should be auxiliary. We later show how this rule can cleanly and precisely encode known proof rules for total correctness of the while loop and recursive procedures. It is notable that the rule has precisely the same shape as in the logic for pure functions. Observe i is implicitly universally quantified: we can thus regard the rule as the conjunction of two judgements, one for $i = 0$ and another for $i \gneq 0$, as follows. Note below $\forall j \lneq 0.B(j)[x/u]\ \equiv\ \forall j.(j \lneq 0\ \supset\ B(j)[x/u])\ \equiv\ \mathbf{t}$.

$$\frac{\{A^{\text{-}x}\}\ \lambda y.M :_u \{B(0)\} \quad \{A \wedge \forall j^{\mathsf{Nat}} \leq i.B(j)[x/u]\}\ \lambda y.M :_u \{B(i+1)\}}{\{A\}\ \mu x.\lambda y.M :_u \{\forall i.B(i)\}}$$

Put in this form, the rule says:

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

Then we conclude the program satisfies $B(n)$ for each n. As an example of the use of the rule, we let M and $B(i)$ in the rule to be the following program and formula.

$$\begin{aligned} M &\stackrel{\text{def}}{=} \mathtt{if}\ y = 0\ \mathtt{then}\ !z = 1\ \mathtt{else}\ x(y-1); !z = x \times !z \\ B(i) &\stackrel{\text{def}}{=} \{\mathsf{T}\}\ u \bullet i\ \{\ !y = i!\}. \end{aligned}$$

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

$$\frac{\begin{array}{c}\{\mathsf{T}\}\ \lambda y.M :_u\ \{\{\mathsf{T}\}\ u \bullet 0\ \{\ !y = 1\ \}\} \\ \{\mathsf{T} \wedge \forall j \leq i.\{\{\mathsf{T}\} x \bullet j\ \{\ !y = j!\ \}\}\}\ \lambda y.M :_u\ \{\{\mathsf{T}\}\ u \bullet (i+1)\ \{\ !y = (i+1)!\}\}\end{array}}{\{\mathsf{T}\}\ \mu x.\lambda y.M :_u\ \{\forall n.\{\mathsf{T}\}\ u \bullet n\ \{\ !y = n!\ \}\}}$$

The shape of the rule [*Rec*] allows a simple extension to induction on an arbitrary well-ordering, which is listed in Figure 6. Here we regard $[\![\alpha]\!]$, which are the closed values

Fig. 6 Proof Rule for Recursion using Well-founded Induction

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

The rule assumes $([\![\alpha]\!], \leq)$ is well-founded.

of type α, is equipped with a well-founded ordering (for example $\mathsf{Nat} \times \mathsf{Nat}$ can be equipped with the lexicographic ordering: we may also refine so that a well-ordering is given to a subset of a domain). This rule can be read just as [*Rec*], replacing the mathematical induction with the well-founded induction.

We shall later show how several known proof rules for total correctness on the while loop and the recursive procedures can be derived using this recursion rule and these recursion rules. In the same way, yet another recursion rule listed in [19] can justify the standard proof rule for loop for partial correctness.

In **[*Promote*]**, we add a predicate on states (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 the standard Hoare Logic, but is consistent in the present setting, since the value has no effects. As a simple example, under the typing $x : \mathsf{Ref}(\mathsf{Nat})$, we have:

$$\frac{\{\mathsf{T}\}\ 3\ :_u\ \{u = 3\}}{\{\mathsf{T} \wedge\, !x = 2\}\ 3\ :_u\ \{u = 3 \wedge\, !x = 2\}} \tag{4.14}$$

which adds the same state at x in the pre/post conditions. Since the constant 3 cannot change state, the conclusion does look reasonable.

[*Consequence*] is the rule introduced by Hoare [16], which involves calculation of validity of entailment. As in other rules, each of the two conditions, $C \supset C_0$ and $C_0' \supset C'$, indicates that the entailment holds under arbitrary models (as defined in Section 5).

Entailment can be considered as a subset relation of inhabitants of fomrulae. For example, assume there are two formulae, C_1 and C_2, each having a single free name x of type $\Delta;\alpha$. Write $|C_i|$ for the set of all semi-closed values of type $\Delta;\alpha$ satisfying C_i, that is: $|C_i| \stackrel{\text{def}}{=} \{\ V \mid x{:}V^{\Delta;\alpha} \models C_i\}$. Then we have the following meta-logical statement.

$$C_1 \supset C_2 \quad \equiv \quad |C_1| \subset |C_2| \tag{4.15}$$

In other words, $C_1 \supset C_2$ is the same thing as saying the inhabitants of C_1 are a subset of those of C_2. If α is Nat, then it means the natural numbers satisfying C_1 are a subset of those satisfying C_2. If α is a higher-order type, then the entailment means the set of programs inhabiting C_1 is a subset of those inhabiting C_2. Calculation of such inclusion is already intractable in the number theory (which is a basis of the consequence rule in Hoare Logic). It surely does not become easier in the present setting, and is qualitatively harder, involving both number-theoretic (and other algebraic) facts on the one hand, and behaviours of programs on the other. However, in practice one can often appeal to syntactic reasoning, cf. §3.3. Kleymann's powerful extension of the consequence rule [21] may as well be used, which we discuss next together with other structural rules.

Fig. 7 Structural Rules

$$[\wedge\text{-}\supset]\ \frac{\{C \wedge A\} V :_u \{C'\}}{\{C\} V :_u \{A \supset C'\}} \qquad [\supset\text{-}\wedge]\ \frac{\{C\} M :_u \{A \supset C'\}}{\{C \wedge A\} M :_u \{C'\}}$$

$$[\vee\text{-}Pre]\ \frac{\{C_1\} M :_u \{C\} \quad \{C_2\} M :_u \{C\}}{\{C_1 \vee C_2\} M :_u \{C\}} \qquad [\wedge\text{-}Post]\ \frac{\{C\} M :_u \{C_1\} \quad \{C\} M :_u \{C_2\}}{\{C\} M :_u \{C_1 \wedge C_2\}}$$

$$[Aux_{\exists}]\ \frac{\{C\}\, M :_u \{C'^{\text{-}i}\}}{\{\exists i.C\}\, M :_u \{C'\}} \qquad [Aux_{\forall}]\ \frac{\{C^{\text{-}i}\}\, M :_u \{C'\}}{\{C\}\, M :_u \{\forall i.C'\}}$$

$$[Aux_{inst}]\ \frac{\{C(i^{\alpha})\} M :_u \{C'(i^{\alpha})\} \quad \alpha \text{ atomic}}{\{C(\mathsf{c}^{\alpha})\} M :_u \{C'(\mathsf{c}^{\alpha})\}} \qquad [Aux_{abst}]\ \frac{\forall \mathsf{c}^{\alpha}.\ \{C(\mathsf{c}^{\alpha})\} M :_u \{C'(\mathsf{c}^{\alpha})\}}{\{C(i^{\alpha})\} M :_u \{C'(i^{\alpha})\}}$$

$$[Weak\text{-}\wedge]\ \frac{\{C\}\, M^{\Gamma;\Delta;\alpha} :_m \{C'\} \quad \mathsf{dom}(\Delta) \cap \mathsf{fn}(C_0) = \emptyset}{\{C \wedge C_0\} M^{\Gamma.\Gamma';\Delta,\Delta';\alpha} :_m \{C' \wedge C_0\}}$$

$$[Constancy]\ \frac{\{C\}\, M^{\Gamma;\Delta;\alpha} :_m \{C'\}}{\{C \wedge A\}\, M^{\Gamma;\Delta;\alpha} :_m \{C' \wedge A\}}$$

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

In [*Consequence-Aux*], we set $\{\vec{x}\} = \mathsf{dom}(\Gamma,\Delta) \cup \{u\}$, $\{\vec{i}\} = \mathsf{fn}(C, C', C_0, C_0') \setminus \{\vec{x}\}$, and $\vec{j}$ (resp. $\vec{y}$) are fresh and of the same length as $\vec{i}$ (resp. $\vec{x}$). We assume no auxiliary reference names occur.

4.2 Proof Rules (2): Structural Rules

Structural rules are those rules which do not change the shape of the concerned program, manipulating only formulae, which are often useful in practice. We have already seen two structural rules, [*Promote*] and [*Consequence*]. In Figure 7, we present a few additional structural rules which are found to be useful in reasoning about programs, including the strengthening of [*Consequence*] due to Kleymann [21].

[$\wedge$-$\supset$] 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 $A = \mathsf{F}$, 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 hold for general programs which may not be values. As a simple application, we mediate the conclusions of the two inferences from the same premise, (4.6) and (4.7) in Page 19. First we infer:

1.	$\{z = 1\}\ \lambda x.!y = x + z\ :_u\ \{\ \forall x^{\mathsf{Nat}}.\{\mathsf{T}\}\, u \bullet x\, \{!y = x+1\}\ \}$	(4.6)
2.	$\{\mathsf{T}\}\ \lambda x.!y = x + z\ :_u\ \{\ z = 1\ \supset\ \forall x^{\mathsf{Nat}}.\{\mathsf{T}\}\, u \bullet x\, \{!y = x+1\}\ \}$	($\wedge$-$\supset$)
3.	$\{\mathsf{T}\}\ \lambda x.!y = x + z\ :_u\ \{\ \forall x^{\mathsf{Nat}}.\{z = 1\}\, u \bullet x\, \{!y = x+1\}\ \}$	(Consequence)

The other direction just does the converse:

$$\dfrac{\dfrac{1.\quad \{\mathsf{T}\}\ \lambda x.!y = x+z\ :_u\ \{\ \forall x^{\mathsf{Nat}}.\{z=1\}\,u\bullet x\{!y=x+1\}\ \}\qquad (4.7)}{2.\quad \{\mathsf{T}\}\ \lambda x.!y = x+z\ :_u\ \{\ z=1\ \supset\ \forall x^{\mathsf{Nat}}.\{\mathsf{T}\}\,u\bullet x\{!y=x+1\}\ \}\qquad (\text{Consequence})}}{3.\quad \{z=1\}\ \lambda x.!y = x+z\ :_u\ \{\ \forall x^{\mathsf{Nat}}.\{\mathsf{T}\}\,u\bullet x\{!y=x+1\}\ \}\qquad (\supset\text{-}\wedge)}$$

The applications of [*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 §3.3.

[$\vee$-*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. **[$\wedge$-*Post*]** is the dual of [$\vee$-*Pre*]. One of the uses of [$\wedge$-*Post*] is when we infer for a closure, i.e. for λ-abstraction. Consider we wish to derive the following judgement.

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

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, [$\wedge$-*Post*] is particularly useful. For example, we can reason as follows in the above case.

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

Note [*Abs*] forces us to have a conclusion made of a single evaluation formula, albeit up to the entailment relation. [$\wedge$-*Post*] allows us to combine several evaluation formulae derived in this way, leading to clearly structured reasoning even if multiple evaluation formulae can in fact be encoded into one (though at this point it is not known whether [$\vee$-*Pre*] and [$\wedge$-*Post*] strictly add provability).

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

$$[\wedge\text{-}\mathit{Pre/Post}]\ \dfrac{\{C_1\}M :_u \{C_1'\}\quad \{C_2\}M :_u \{C_2'\}}{\{C_1\wedge C_2\}M :_u \{C_1'\wedge C_2'\}}$$

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 21) 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.

1. $\{\, !x + !y = i \}\ !x = !x + !y;\ y = !x \{\, !x + !y = 2 \times i \}$	Premise
2. $\{\mathsf{T}\}\ \lambda().(!x = !x + !y;\ y = !x)\{\ \{\, !x + !y = i\}\, u \bullet ()\ \{\, !x + !y = 2 \times i\}\}$	(Abs)
3. $\{\mathsf{T}\}\ \lambda().(!x = !x + !y;\ y = !x)\{\ \forall i^{\mathsf{Nat}}.\{\, !x + !y = i\}\, u \bullet ()\{\, !x + !y = 2 \times i\}\}$	($\mathsf{Aux}_\forall$)

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, 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.

[*Weak*-$\wedge$] weakens, or adds, a formula which does not touch existing reference variables. Because of this condition, if the initial state satisfies (resp. contradicts) C_0, then the final state will also satisfy (resp. contradicts) C_0. [*Weak*-$\wedge$] is useful when we wish to reason about each part of a given program assuming only what is relevant, and can often replace the use of the so-called constancy rule in Hoare logic. As a very simple example, consider the following assertion.

$$\{\ !x = 0 \wedge !y = 1\ \}\ !x = !x + 1\ ;\ !y := !y + 1\ \{\ !x = 1 \wedge !y = 2\ \} \tag{4.16}$$

We wish to assert and reason only about x in the first command, and only about y in the second (this becomes substantial if each part is a long program). This can be done if we use [*Weak*-$\wedge$], as the following inference shows. Below we make explicit the typing of programs and let $\Delta \stackrel{\text{def}}{=} x : \mathsf{Ref(Nat)}, y : \mathsf{Ref(Nat)}$.

1. $\{\ !x = 0\ \}\ (!x = !x + 1)^{x:\mathsf{Ref(Nat)};\mathsf{Unit}}\ \{\ !x = 1\ \}$	(Assign)
2. $\{\ !y = 1\ \}\ (!y = !y + 1)^{y:\mathsf{Ref(Nat)};\mathsf{Unit}}\ \{\ !y = 2\ \}$	(Assign)
3. $\{\ !x = 0 \wedge !y = 1\ \}\ (!x = !x + 1)^{\Delta;\mathsf{Unit}}\{\ !x = 1 \wedge !y = 1\ \}$	(1, Weak-$\wedge$)
4. $\{\ !x = 1 \wedge !y = 1\ \}\ (!y = !y + 1)^{\Delta;\mathsf{Unit}}\{\ !x = 1 \wedge !y = 2\ \}$	(2, Weak-$\wedge$)
5. $\{\ !x = 0 \wedge !y = 1\ \}\ (!x = !x + 1\ ;\ !y = !y + 1)^{\Delta;\mathsf{Unit}}\{\ !x = 1 \wedge !y = 2\ \}$	(3, 4, Seq)

The rule [*Seq*] is from (4.10), page 20. If we use an effect typing, we can further facilitate the reasoning.

Finally **[*Consequence-Aux*]** is the rule which was originally introduced in the setting of the standard Hoare Logic by Kleymann [21], capturing elegantly and precisely the semantics of auxiliary names in judgements. [*Consequence*] (of Figure 4), [$Aux_\exists$] 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 4. This is mainly because we can freely assert and reason about behaviours of procedures using quantifications, which is one of the main cases when we need the full use of semantics of auxiliary variables becomes necessary.

5 Models and Soundness

5.1 Models

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: [19] constructs a more uniform universe of models from typed processes.

We use the standard contextual congruence (cf.[32]) for defining an abstract notion of behaviour used in the semantics of the logic. Below we recall $\Delta \vdash \sigma$ means $\mathsf{dom}(\Delta) = \mathsf{dom}(\sigma)$ and, moreover, $\Delta \vdash \sigma(x) : \alpha$ iff $\Delta(x) = \mathsf{Ref}(\alpha)$ for each $x \in \mathsf{dom}(\sigma)$.

Definition 22. (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 $\Delta \vdash M_{1,2} : \mathsf{Unit}$ and for each σ such that $\Delta \vdash \sigma$, we have $(M_1, \sigma) \Downarrow$ iff $(M_2, \sigma) \Downarrow$.

Proposition 23. (properties of $\cong$)

1. (renaming) *Let ϕ be an injective name substitution. Then* $\Gamma;\Delta \vdash M \cong N : \alpha$ *iff* $\Gamma\phi;\Delta\phi \vdash M\phi : \alpha$.
2. (weakening) $\Delta \vdash M \cong N : \alpha$ *and* $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \emptyset$ *imply* $\Delta \cdot \Delta' \vdash M \cong N : \alpha$.
3. (thinning) $\Delta \cdot \Delta' \vdash M \cong N : \alpha$ *and* $(\mathsf{fn}(M,N)) \cap \mathsf{dom}(\Delta') = \emptyset$ *imply* $\Delta \vdash M \cong N : \alpha$.
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) *Assume* $\Delta \vdash M_{1,2} : \alpha$ *and, for each* $\Delta \vdash \sigma$, *we have* $(M_i, \sigma) \Uparrow$ *($i = 1,2$). Then* $\Delta \vdash M_1 \cong M_2 : \alpha$.
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* $x : \mathsf{Ref}(\alpha) \in \Delta$. *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* $\Delta \vdash V : \mathsf{Unit}$. *Then* $V \cong ()$.
8. (boolean) *Let* $\Delta \vdash V : \mathsf{Bool}$. *Then either* $V \cong \mathbf{t}$ *or* $V \cong \mathbf{f}$. *Further* $\mathbf{t} \not\cong \mathbf{f}$.
9. (numerals) *Let* $\Delta \vdash V : \mathsf{Nat}$. *Then* $V \cong \mathsf{n}$ *for some numeral n. Further* $\mathsf{m} \not\cong \mathsf{n}$ *whenever* m *and* n *are distinct numerals.*
10. (first-order operators) *Let* $\Delta \vdash \mathsf{op}(\vec{V}) : \alpha$. *Then there exists W such that* $(\mathsf{op}(\vec{V}), \sigma) \Downarrow (W, \sigma)$ *such that* $\mathsf{op}(\vec{V}) \cong W$ *for each* $\Delta \vdash \sigma$.
11. (abstraction) *Let* $\Delta \vdash (\lambda x.M)V : \alpha$. *Then* $\Delta \vdash (\lambda x.M)V \cong M\mathbf{sub}V{=}x. : \alpha$.
12. (recursion) *Assume* $\Gamma \cdot x : \alpha\ ;\ \Delta \vdash \lambda y.M : \alpha$ *and let* $W \stackrel{def}{=} (\lambda y.M)[\mu x.\lambda y.M/x]$. *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* $W \cong \mathtt{in}_i(V)$ *for some* $i \in \{1,2\}$ *and* V. *Further if* $\Delta \vdash \mathtt{in}_i(V) \cong \mathtt{in}_i(W)$ *then* $\Delta \vdash \mathtt{in}_i(V) \cong \mathtt{in}_i(W)$ *implies* $\Delta \vdash V \cong W$.

Proof. All are standard. (7–16) use Proposition 3 as necessary. □

Definition 24. An *abstract value of type* $\Delta;\alpha$ is a $\cong$-congruence class of semi-closed values typed α under Δ. We write $[V]^{\Delta;\alpha}$ for an abstract value represented by $\Delta \vdash V : \alpha$.

A model is defined using abstract values.

Definition 25. (models) A *model of type* $\Gamma;\Delta$ is a pair (ξ,σ) such that ξ is a finite map from $\mathsf{dom}(\Gamma)$ to abstract values such that each $x \in \mathsf{dom}(\Gamma)$ is mapped to an abstract value typed as $[V]^{\Delta;\Gamma(x)}$; and σ is a finite map from $\mathsf{dom}(\Delta)$ to abstract values such that each $x \in \mathsf{dom}(\Delta)$ is mapped to an abstract value typed as $[V]^{\Delta;\alpha}$ with $\Delta(x) = \mathsf{Ref}(\alpha)$. We let $\mathcal{M},\ldots$ range over models.

We write $\Gamma;\Delta \vdash \mathcal{M}$ 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, taken modulo $\cong$. As needed, we distinguish concrete stores from abstract stores by writing $[\sigma]^{\Delta}$ (which denotes an abstract store whose representative is σ).

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

- Given $\Gamma;\Delta \vdash \mathcal{M}$ s.t. $\mathcal{M} = (\xi,\sigma)$, we write $x^{\alpha} : [V]_{\cong}^{\Delta;\alpha} \in \mathcal{M}$ when $\Gamma(x) = \alpha$ and $\xi(x) = [V]_{\cong}^{\Delta;\alpha}$; and $x^{\mathsf{Ref}(\alpha)} \mapsto [V]_{\cong}^{\Delta;\alpha} \in \mathcal{M}$ when $\Delta(x) = \mathsf{Ref}(\alpha)$ and $\sigma(x) = [V]_{\cong}^{\Delta;\alpha}$.
- Given $[V]_{\cong}^{\Delta;\alpha}$ and $[W]_{\cong}^{\Delta;\beta}$, we write $\langle [V]_{\cong}^{\Delta;\alpha},\ [W]_{\cong}^{\Delta;\beta} \rangle$ for $[\langle V,W\rangle]_{\cong}^{\Delta;\alpha\times\beta}$ (i.e. the congruence class including $\langle V,W\rangle$). Similarly for other operations.

Definition 26. (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 $[\![e]\!]\mathcal{M}$, is given by the following clauses.

$$
\begin{aligned}
[\![x^{\alpha}]\!]\mathcal{M} &\stackrel{\text{def}}{=} [V]_{\cong}^{\Delta;\alpha} && (x^{\alpha} : [V]_{\cong}^{\Delta;\alpha} \in \mathcal{M})\\
[\![!x^{\mathsf{Ref}(\alpha)}]\!]\mathcal{M} &\stackrel{\text{def}}{=} [V]_{\cong}^{\Delta;\alpha} && (x^{\mathsf{Ref}(\alpha)} \mapsto [V]_{\cong}^{\Delta;\alpha} \in \mathcal{M})\\
[\![x^{\mathsf{Ref}(\alpha)}]\!]\mathcal{M} &\stackrel{\text{def}}{=} x\\
[\![\mathbf{c}^{\alpha}]\!]\mathcal{M} &\stackrel{\text{def}}{=} [\mathbf{c}]_{\cong}^{\Delta;\alpha}\\
[\![\mathbf{op}(\vec{e})]\!]\mathcal{M} &\stackrel{\text{def}}{=} \mathbf{op}([\![\vec{e}]\!]\mathcal{M})\\
[\![\langle e,e'\rangle]\!]\mathcal{M} &\stackrel{\text{def}}{=} \langle [\![e]\!]\mathcal{M}, [\![e']\!]\mathcal{M}\rangle\\
[\![\pi_i(e)]\!]\mathcal{M} &\stackrel{\text{def}}{=} \pi_i([\![e]\!]\mathcal{M})\\
[\![\mathsf{inj}_i^{\alpha+\beta}(e)]\!]\mathcal{M} &\stackrel{\text{def}}{=} \mathtt{in}_i([\![e]\!]\mathcal{M})
\end{aligned}
$$

Remark 27. (interpretation of terms)

1. By construction of models, all terms are interpreted as abstract 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 an abstract 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.

2. We can check all terms of atomic types are in effect interpreted as the congruence classes of constants (note any semi-closed values of atomic types are always closed values, hence essentially constants). In particular, in the fifth line, $\mathbf{op}([\![\vec{e}]\!]\mathcal{M}\cdot I)$ always evaluates to a closed value, hence a constant.

The satisfaction relation is defined using the component-wise union of two disjoint models, one interpreting primary names and another auxiliary names. Writing $I, I', \ldots$ for models used for interpreting auxiliary names, the satisfaction relation is written: $\mathcal{M}^{\Gamma;\Delta} \models^{I} C$, which reads: *under an interpretation I, C is satisfied by a model $\mathcal{M}$*. The satisfaction relation is defined following the standard first-order logic with equality [24, §2.8], adding an additional clause for evaluation formulae. Below we list all clauses for completeness. In the last clause, we set, with $\Delta \vdash \sigma$:

$$([V_1]^{\Delta;\alpha\Rightarrow\beta}[V_2]^{\Delta;\alpha},\ [\sigma]^{\Delta}) \Downarrow ([W]^{\Delta;\beta}, [\sigma']^{\Delta}) \quad \stackrel{\text{def}}{\Leftrightarrow} \quad (V_1V_2,\ \sigma) \Downarrow (W', \sigma') \text{ s.t. } \Delta \vdash W' \cong W : \beta$$

which is well-defined by Proposition 23 (6).

Definition 28. (satisfaction) Given $\Gamma;\Delta \vdash \mathcal{M}$ and $\Gamma;\Delta \vdash C$, the relation $\mathcal{M}^{\Gamma;\Delta} \vdash^{I} C$ (read: $\mathcal{M}^{\Gamma;\Delta}$ *satisfies C under I*) is generated from the following clauses.

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

where, in each clause, a logical connective/quantifier on the right-hand side (if any) is used semantically. We often omit $\Gamma;\Delta$ from $\mathcal{M}^{\Gamma;\Delta} \models^{I} C$, writing simply $\mathcal{M} \models^{I} C$.

Remark 29. (satisfaction for invocation formulae) Intuitively, the satisfaction of an invocation formula, $\mathcal{M}^{\Gamma;\Delta} \models^{I} \{C\}e_1 \bullet e_2 \searrow x\{C'\}$, 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 that the state part of $\mathcal{M}$ is neglected — we take any state at Δ satisfying C, even if it contradicts the state part of $\mathcal{M}$. This is because we generally need to describe the behaviour of invocation under hypothetical states, not the present one (for example, if we wish to specify the behaviour of $\lambda().x := !x+1$, we wish to say it increments a value stored in x not only in the present model but also 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, confusing abstract values and their representatives.

Definition 30. (semantics of judgement) $\models\{C\}\,M^{\Gamma;\Delta;\alpha}:_u\{C'\}$ iff, for each I, ξ and σ, whenever $(\xi,\sigma)\models^I C$, we have $(M\xi,\sigma)\Downarrow(V,\sigma')$ such that $(\xi\cdot u:[V]^{\Delta;\alpha},\sigma')\models^I C'$.

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

Proposition 31. *If* $\models\{C\}\,M^{\Gamma;\Delta;\alpha}:_u\{C'\}$ *and* $\Gamma;\Delta\vdash M\cong N:\alpha$*, then* $\models\{C\}\,N^{\Gamma;\Delta;\alpha}:_u\{C'\}$.

The proof is immediate from the definition (formally by Proposition 23 6).

We shall later prove the following two basic results on the semantics of the logic. The first result says we can safely use axioms in Figure 3, Section 3.3, Page 14, as the basis of reasoning.

Proposition 32. (soundness of axioms) *All axioms in Figure 3 are true under arbitrary (well-typed) models.*

The second one says that we can safely use the proof rules in Figures 4 and 7, or other derivable rules from them, for deriving properties of programs.

Theorem 33. (soundness) (soundness of proof rules) *If* $\vdash\{C\}\,M^{\Gamma;\Delta;\alpha}:_u\{C'\}$ *by the proof rules in Figures 4 and 7, then* $\models\{C\}\,M^{\Gamma;\Delta;\alpha}:_u\{C'\}$.

5.2 Basic Properties of the Models

We use the following notations in the rest of the section.

Notation 34.

1. We often write $[V],[W],\ldots$ to denote abstract values, omitting types. We also let $\mathbf{v}^{\alpha},\mathbf{w}^{\beta},\ldots$ etc., or often $\mathbf{v},\mathbf{w},\ldots$ range over abstract values.
2. Let $\Gamma;\Delta\vdash\mathcal{M}$ such that $\mathcal{M}=(\xi,\sigma)$ below.
   (a) We write $\mathcal{M}[x:\mathbf{v}]$ for the result of replacing the target of x in ξ with $\mathbf{v}$, assuming $x\in\mathsf{dom}(\xi)$. Similarly we define $\mathcal{M}[x\mapsto\mathbf{v}]$, $\xi[x:\mathbf{v}]$ and $\sigma[x\mapsto\mathbf{v}]$.
   (b) We write $\mathcal{M}\cdot x:\mathbf{v}$ for $(\xi\cup\{x:\mathbf{v}\},\sigma)$, assuming simultaneously $x\notin\mathsf{dom}(\xi\cup\sigma)$. Similarly we define $\mathcal{M}\cdot x\mapsto\mathbf{v}$, $\xi\cdot x:\mathbf{v}$ and $\sigma\cdot x\mapsto\mathbf{v}$.
3. $\xi\backslash\vec{x}$ indicates the result of taking off the submap of ξ whose domain is $\{\vec{x}\}$. Similarly we write $\mathcal{M}\backslash\vec{x}$ for the result of taking off $\vec{x}$-elements from the components of $\mathcal{M}$.

We also assume, as before, that all judgement, terms, models, etc. are implicitly well-typed. In the proofs, we often write down a sequence of (meta-logical) implication or (meta-logical) equivalences, meaning conjunction of pairs of implications/equivalences.[4]

[4] In more detail, given statements $\Psi_0,\Psi_1,\Psi_2,\ldots$ about the logic, we write:

$$\begin{array}{ccc}\Psi_1 & \supset & \Psi_2\\ & \supset & \Psi_3\\ & \vdots & \\ & \supset & \Psi_n\end{array}$$

for the sequence of implications:

$$\Psi_1\supset\Psi_2,\quad\Psi_2\supset\Psi_3,\quad\ldots\quad\Psi_{n-1}\supset\Psi_n$$

each of which is claimed to be valid meta-logically (e.g. in set theory). Similarly when some of $\supset$ are replaced by $\equiv$.

The following properties are standard.

Lemma 35. *Let* $\Gamma,\Theta \vdash C : \alpha$ *and assume given* $\Gamma \vdash \mathcal{M}$, $\Theta \vdash I$.

1. (renaming) *For each injective renaming* ϕ, $\mathcal{M} \models^{I} C$ *iff* $\mathcal{M}\phi \models^{I\phi} C\phi$.
2. (thinning) *If* $x \notin \mathsf{dom}(\Gamma,\Theta)$. *Then* $\mathcal{M}\cdot x : V \models^{I} C$ *implies* $\mathcal{M} \models^{I} C$.
3. (weakening) *If* $\mathcal{M},\mathcal{M}'$, I *and* I' *are all mutually disjoint in their domains, then* $\mathcal{M} \models^{I} C$ *implies* $\mathcal{M}\cdot\mathcal{M}' \models^{I\cdot I'} C$.

Proof. Each using the corresponding clauses in Proposition 23 (1–3). □

Many of the proof rules in Figure 4 use substitutions. The following is their standard properties. Below if $\mathcal{M} = (\xi,\sigma)$, we write $\mathsf{dom}(\mathcal{M})$ for $\mathsf{dom}(\xi)\cup\mathsf{dom}(\sigma)$. 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 36. (substitution) *Let* $\mathcal{M} = (\xi,\sigma)$ *and let* $x \notin \mathsf{dom}(I)$.

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

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 $x \in \mathsf{fn}(C)$.

$$\begin{array}{rcll}
\mathcal{M} \models^{I} C & \equiv & \mathcal{M}\cdot x : [\![e]\!]\mathcal{M} \models^{I} C \wedge x = e & \text{(Lem 35-3)}\\
 & \equiv & \mathcal{M}\cdot x : [\![e]\!]\mathcal{M} \models^{I} C[e/x] \wedge x = e & \text{(substitutivity)}\\
 & \equiv & \mathcal{M}\cdot x : [\![e]\!]\mathcal{M} \models^{I} C[e/x] & (\mathcal{M}\cdot x : [\![e]\!]\mathcal{M} \models^{I} x = e \text{ always})
\end{array}$$

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

$$[\![e'[e/!x]]\!]\mathcal{M} \quad = \quad [\![e']\!]\mathcal{M}[x \mapsto [\![e]\!]\mathcal{M}] \tag{5.1}$$

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

$$\begin{array}{rcll}
\mathcal{M} \models^{I} (e_1 = e_2)\sigma & \equiv & [\![e_1\sigma]\!]\mathcal{M} = [\![e_2\sigma]\!]\mathcal{M} & \text{(Def. 28)}\\
 & \equiv & [\![e_1]\!]\mathcal{M}[x \mapsto [\![e]\!]\mathcal{M}] = [\![e_2]\!]\mathcal{M}[x \mapsto [\![e]\!]\mathcal{M}] & \text{5.1}\\
 & \equiv & \mathcal{M}[x \mapsto [\![e]\!]\mathcal{M}] \models^{I} e_1 = e_2 & \text{(Def. 28)}
\end{array}$$

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

We now prove Proposition 32, the soundness of the axioms. The proof of Theorem 33 is given in the next subsection. Among the axioms in Figure 3, (t1)–(t8) are easy using Proposition 23. For (t1),

$$\begin{array}{rcll}
\mathsf{T} & \supset & [\![x^{\mathsf{Unit}}]\!]\mathcal{M} = [()] & \text{(Proposition 23-7)}\\
 & \supset & \mathcal{M} \models^{I} x^{\mathsf{Unit}} = () & \text{(Definition 28)}
\end{array}$$

For (t2),

$$\begin{array}{lll} \mathsf{T} & \supset & [\![x^{\mathsf{Bool}}]\!]\mathcal{M} = [\mathbf{t}] \ \vee \ [\![x^{\mathsf{Bool}}]\!]\mathcal{M} = [\mathbf{f}] \qquad \text{(Proposition 23-8)} \\ & \supset & \mathcal{M} \models^{I} x^{\mathsf{Bool}} = \mathbf{t} \ \vee \ x^{\mathsf{Bool}} = \mathbf{f} \qquad \text{(Definition 28)} \end{array}$$

For (t3),

$$\begin{array}{lll} \mathsf{T} & \supset & [\mathbf{t}] \neq [\mathbf{f}] \qquad \text{(Proposition 23-8)} \\ & \supset & \mathcal{M} \models^{I} \mathbf{t} \neq \mathbf{f} \qquad \text{(Definition 28)} \end{array}$$

(t4–t8) are similar, using the corresponding clauses in Proposition 23.

Each of (e1–e8) 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.

$$\begin{array}{ll} \multicolumn{2}{l}{(\xi,\sigma_0) \models^{I} \{C_1\}x \bullet y \searrow z\{C'\} \ \wedge \ (\xi,\sigma_0) \models^{I} \{C_2\}x \bullet y \searrow z\{C'\}} \\ \equiv & \forall\sigma.\ ((\xi,\sigma) \models^{I} C_1 \supset \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} C'\ ) \ \wedge \\ & \forall\sigma.\ ((\xi,\sigma) \models^{I} C_2 \supset \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} C'\ ) \\ \equiv & \forall\sigma.\ ((\xi,\sigma) \models^{I} C_1 \supset \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} C' \ \wedge \\ & \quad (\xi,\sigma) \models^{I} C_2 \supset \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} C'\ ) \\ \equiv & \forall\sigma.\ (\ (\xi,\sigma) \models^{I} C_1 \vee (\xi,\sigma) \models^{I} C_2) \\ & \qquad \supset \quad \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} C'\ ) \\ \equiv & \forall\sigma.\ (\ (\xi,\sigma) \models^{I} (C_1 \vee C_2) \\ & \qquad \supset \quad \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} C'\ ) \end{array}$$

(e2–e5) are similar. For (e6):

$$\begin{array}{ll} \multicolumn{2}{l}{(\xi,\sigma_0) \models^{I} \{C\}x \bullet y \searrow z\{A^{\text{-}z} \supset C'\}} \\ \equiv & \forall\sigma.\ ((\xi,\sigma) \models^{I} C \supset \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} A \supset C'\ ) \\ \equiv & \forall\sigma.\ ((\xi,\sigma) \models^{I} C \supset \exists V,\sigma'.(\ (\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge \\ & \qquad\qquad (\xi \models^{I} A \supset (z:[V]\cdot\xi,\sigma') \models^{I} C')\ )\ ) \\ \supset & \forall\sigma.\ ((\xi,\sigma) \models^{I} C \supset \exists V,\sigma'.(\ (\xi \models^{I} A \supset (\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma')) \ \wedge \\ & \qquad\qquad (\xi \models^{I} A \supset (z:[V]\cdot\xi,\sigma') \models^{I} C')\ )\ ) \\ \equiv & \xi \models^{I} A \ \supset \\ & \forall\sigma.\ ((\xi,\sigma) \models^{I} C \supset \exists V,\sigma'.(\ \xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \ \wedge \ (z:[V]\cdot\xi,\sigma') \models^{I} C')\ ) \end{array}$$

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

For (e7) we infer:

$$\begin{array}{ll} \multicolumn{2}{l}{(\xi,\sigma_0) \models^{I} \{C\}x \bullet y \searrow z\{C'\}} \\ \supset & \forall\sigma.\ (((\xi,\sigma) \models^{I} C \wedge \xi \models^{I} A) \\ & \qquad \supset \ \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge (z:[V]\cdot\xi,\sigma') \models^{I} C'\ ) \\ \equiv & \forall\sigma.\ (((\xi,\sigma) \models^{I} C \wedge \xi \models^{I} A) \\ & \qquad \supset \ \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge ((z:[V]\cdot\xi,\sigma') \models^{I} C' \wedge \xi \models^{I} A)\ ) \\ \equiv & \forall\sigma.\ (((\xi,\sigma) \models^{I} C \wedge A) \\ & \qquad \supset \ \exists V,\sigma'.(\xi(x) \bullet \xi(y),\sigma) \Downarrow (V,\sigma') \wedge ((z:[V]\cdot\xi,\sigma') \models^{I} C' \wedge A)\ ) \end{array}$$

Again the entailment is in general irreversible. Finally (e8) is immediate. We have proved Proposition 32.

5.3 Proof of Theorem 33

The proofs of the soundness of the rules in Figures 4 and 7 closely follows those of the soundness of the traditional Hoare logic [38]. We present the reasoning for each rule one by one, starting from the structural rules.

Convention 37. 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 38. *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. $\forall \xi, \sigma.\ (\ (\xi,\sigma) \models^I C \wedge A\ \supset\ (\xi \cdot u : [V\xi], \sigma) \models^I C'\ )$ (premise)

---

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

---

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

---

4. $\forall \xi, \sigma.\ (\ (\xi,\sigma) \models^I C \supset (\xi \cdot u : [V\xi], \sigma) \models^I A \supset (\xi \cdot u : [V\xi], \sigma) \models^I C'\ )$ (Lem. 35-3)

---

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

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-Aux*], while [*Consequence-Aux*] is the precise mirror of Kleymann's corresponding rule [21] so that Kleymann's argument is directly adaptable. The proof of [*Constancy*] is essentially identical with (e7). Thus we are left with $[Weak\text{-}\wedge]$, which we justify below.

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

---

2. $\forall (\xi,\sigma)^{\Gamma;\Delta}.\ \forall \sigma_0^{\Delta_0}.$
   $(\ (\xi, \sigma \cdot \sigma_0) \models^I C\ \ \supset$
   $\exists V, \sigma'.(\ (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0)\ \wedge\ (\xi \cdot u : [V\xi], \sigma' \cdot \sigma_0) \models^I C'\ )$ (Pro.23-3)

---

3. $\forall (\xi,\sigma)^{\Gamma;\Delta}.\ \forall \sigma_0^{\Delta_0}.$
   $(\ ((\xi, \sigma \cdot \sigma_0) \models^I C\ \wedge\ (\xi,\sigma_0) \models^I C_0)\ \ \supset$
   $\exists V, \sigma'.(\ (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0)\ \wedge\ (\xi \cdot u : [V\xi], \sigma' \cdot \sigma_0) \models^I C'\ \wedge\ (\xi,\sigma_0) \models^I C_0\ )$

---

4. $\forall (\xi,\sigma)^{\Gamma;\Delta}.\ \forall \sigma_0^{\Delta_0}.\ (\ ((\xi, \sigma \cdot \sigma_0) \models^I C\ \wedge\ (\xi, \sigma \cdot \sigma_0) \models^I C_0)\ \supset$
   $\exists V, \sigma'.(\ (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0) \wedge (\xi \cdot u : [V\xi], \sigma' \cdot \sigma_0) \models^I C' \wedge$
   $(\xi \cdot u : [V\xi], \sigma' \cdot \sigma_0) \models^I C_0\ )$ (Pro.23-3)

---

5. $\forall (\xi,\sigma)^{\Gamma;\Delta}.\ \forall \sigma_0^{\Delta_0}.\ (\ ((\xi, \sigma \cdot \sigma_0) \models^I C \wedge C_0\ \supset$
   $\exists V, \sigma'.(\ (M\xi, \sigma \cdot \sigma_0) \Downarrow (V, \sigma' \cdot \sigma_0)\ \wedge\ (\xi \cdot u : [V\xi], \sigma' \cdot \sigma_0) \models^I C' \wedge C_0\ )$

This concludes the soundness proof for structural rules.

We now turn to the rules in Figure 4. For [*Var*], let the typing be $\Gamma;\Delta \vdash x : \alpha$ (note this means x is not a reference name) and let $\Gamma,\Delta,\Theta \vdash C$. Below we remember $u \notin \mathsf{dom}(\Gamma,\Delta,\Theta)$ (since u is an anchor).

$$\begin{array}{llr} 1. & - & \text{(no premise)} \\ \hline 2. & \forall \mathcal{M}^{\Gamma;\Delta}.\,(\mathcal{M} \models^{I} C[x/u] \quad \supset \quad \mathcal{M}\cdot u : ([\![x]\!]\mathcal{M}\cdot I) \models^{I} C) & \text{(Lemma 36-1(a))} \end{array}$$

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

$$\begin{array}{llr} 1. & - & \text{(no premise)} \\ \hline 2. & \forall \mathcal{M}^{\Gamma;\Delta}.\,(\mathcal{M} \models^{I} C[\mathsf{c}/u] \quad \supset \quad \mathcal{M}\cdot u : ([\![\mathsf{c}]\!]\mathcal{M}\cdot I) \models^{I} C) & \text{(Lemma 36-1(a)))} \end{array}$$

For [*Op*], we infer as follows, assuming $\Gamma;\Delta \vdash \mathsf{op}(\vec{M}) : \alpha$ and $\Gamma,\Delta,\Theta \vdash C, C'$ where, by Convention 21 (2), we assume $\{\vec{m}\} \cap \mathsf{dom}(C,C') = \emptyset$. 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.

$$\begin{array}{rlr}
1. & \forall \xi',\sigma.\,((\xi',\,\sigma) \models^{I} C \;\supset\; \exists \sigma',V.\,(\,(M_0\xi,\sigma) \Downarrow (V,\sigma') \;\wedge\; \xi'[m_0 : [V]],\,\sigma') \models^{I} C_1) & \text{(prem)} \\ \hline
2. & \forall \xi',\sigma.\,((\xi',\,\sigma) \models^{I} C_1 \;\supset\; \exists \sigma',V.\,(\,(M_1\xi',\sigma) \Downarrow (V,\sigma') \;\wedge\; \xi'[m_1 : [V]],\,\sigma') \models^{I} C_2 & \text{(prem)} \\ \hline
 & \vdots & \\ \hline
n. & \forall \xi',\sigma.((\xi',\,\sigma) \models^{I} C_{n-1} \supset \exists \sigma',V.\,((M_{n-1}\xi',\sigma) \Downarrow (V,\sigma') \wedge \xi'[m_n : [V]],\sigma') \models^{I} C'[\mathsf{op}(\vec{m})/u]) & \\
 & & \text{(prem)} \\ \hline
n+1. & \forall \xi',\sigma.\,((\xi',\,\sigma) \models^{I} C \;\supset & \\
 & \quad \exists \sigma',\vec{V}.\,(\,(\mathsf{op}(M_0..M_{n-1})\xi',\sigma) \longrightarrow^{*} (\mathsf{op}(V_1..V_{n-1}),\sigma') \;\wedge & \\
 & \qquad (\xi'[\vec{m} : [\vec{V}]],\,\sigma') \models^{I} C'[\mathsf{op}(\vec{m})/u])) & \text{(1..n)} \\ \hline
n+2. & \forall \xi',\sigma.\,((\xi',\,\sigma) \models^{I} C \;\supset & \\
 & \quad \exists \sigma',\vec{V},W.\,(\,(\mathsf{op}(M_0..M_{n-1})\xi',\sigma) \longrightarrow^{*} (\mathsf{op}(V_1..V_{n-1}),\sigma') \;\wedge & \\
 & \qquad (\xi'[\vec{m} : [\vec{V}]]\cdot u : [\mathsf{op}(\vec{V})],\,\sigma') \models^{I} C')) & \text{(Lemma 36-1(b))} \\ \hline
n+3. & \forall \xi,\sigma.\,((\xi,\,\sigma) \models^{I} C \;\supset & \\
 & \quad \exists \sigma',W.\,(\,(\mathsf{op}(\vec{M})\xi,\sigma) \Downarrow (W,\sigma') \;\wedge\; (\xi\cdot u : [W],\,\sigma') \models^{I} C')) & \text{(Lemmas 10 and 35-2)}
\end{array}$$

For $[Abs]$, let $\Gamma\cdot x:\alpha\,;\,\Delta\vdash M:\beta$ as well as $\Gamma\cdot x:\alpha\,;\,\Delta\vdash\xi'$ and $\Gamma\,;\,\Delta\vdash\xi$. Below in Line 3 we confuse abstract values and their representatives.

1. $\forall\xi',\sigma.(\,(\xi',\,\sigma)\models^{I} C\wedge A^{-x}\supset\exists\sigma',V.(\,(M\xi',\sigma)\Downarrow(V,\sigma')\wedge\xi'\cdot m:[V],\,\sigma')\models^{I} C'\,)$ (prem)

2. $\forall\xi.(\,\xi\models^{I} A\supset$
   $\forall\mathbf{w},\sigma.(\,\xi\cdot x:\mathbf{w},\,\sigma)\models^{I} C\supset$
   $\exists\sigma',V.(\,(M(\xi\cdot x:\mathbf{w}),\sigma)\Downarrow(V,\sigma')\,\wedge\,\xi\cdot m:[V],\,\sigma')\models^{I} C'\,)$ (Lemma 35-2)

3. $\forall\xi.(\,\xi\models^{I} A\supset$
   $\forall\mathbf{w},\sigma.(\,\xi\cdot x:\mathbf{w},\,\sigma)\models^{I} C\supset$
   $\exists\sigma',V.(\,((\lambda x.M\xi)\mathbf{w},\,\sigma)\Downarrow(V,\sigma')\,\wedge\,\xi\cdot m:[V],\,\sigma')\models^{I} C'\,)$ (Prop. 23-11)

4. $\forall\xi.(\,\xi\models^{I} A\supset$
   $\forall\mathbf{w},\sigma.(\,\xi\cdot x:\mathbf{w},\,\sigma)\models^{I} C\supset$
   $\exists\sigma',V.(\,(((\llbracket u\rrbracket\xi\cdot u:[\lambda x.M\xi])(\llbracket x\rrbracket x:\mathbf{w})).\,\sigma)\Downarrow(V,\sigma')\,\wedge\,\xi\cdot m:[V],\,\sigma')\models^{I} C'\,)$
   (Definition 26)

5. $\forall\xi.(\,\xi\models^{I} A\quad\supset\quad(\xi\cdot u:[\lambda x.M\xi],\,\sigma)\models^{I}\forall x.\{C\}\,u\bullet x:_{m}\{C'\}$ (Definition 28)

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

1. $\forall\xi,\sigma.(\,(\xi,\sigma)\models^{I} C\quad\supset\quad\exists V,\sigma'.(\,(M\xi,\sigma)\Downarrow(V,\sigma')\,\wedge\,\xi[m:[V]]\models^{I} C_0))$ (prem)

2. $\forall\xi,\sigma.(\,(\xi,\sigma)\models^{I} C_0\supset$
   $\exists V,\sigma'.(\,(N\xi,\sigma)\Downarrow(W,\sigma')\,\wedge\,\xi\cdot m:[W]\models^{I} C_1\,\wedge\,\{C_1\}m\bullet n\searrow u\{C'\}$ (prem)

3. $\forall\xi',\sigma.(\,(\xi',\sigma)\models^{I} C\supset\exists U,\sigma'.(\,((MN)\xi',\sigma)\Downarrow(U,\sigma')\,\wedge\,\xi'\cdot u:[U]\models^{I}\{C'\}$
   (1, 2, Definition 28)

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

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

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

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

4. $\forall\xi,\sigma.(\,(\xi\cdot b:[\mathbf{t}],\sigma)\models^{I} C_0\supset\exists V,\sigma'.(\,(N_1\xi,\sigma)\Downarrow(V,\sigma')\,\wedge\,(\xi\cdot u:[V],\sigma')\models^{I} C'\,)\,)$
   (2, Lemma 36-1(a))

5. $\forall\xi,\sigma.(\,(\xi\cdot b:[\mathbf{f}],\sigma)\models^{I} C_0\supset\exists V,\sigma'.(\,(N_2\xi,\sigma)\Downarrow(V,\sigma')\,\wedge\,(\xi\cdot u:[V],\sigma')\models^{I} C'\,)\,)$
   (3, Lemma 36-1(a))

6. $\forall\xi,\sigma.(\,(\xi,\sigma)\models^{I} C\supset$
   $\exists V,\sigma'.(\,((\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2)\xi,\sigma)\Downarrow(V,\sigma')\,\wedge\,(\xi\cdot u:[V],\sigma')\models^{I} C'\,)\,)$
   (1, 4, 5, Lemma 8)

For $[In_1]$, let $\Gamma;\Delta \vdash \mathtt{in}_i(M) : \alpha_1 + \alpha_2$ and $\Gamma;\Delta \vdash (\xi,\sigma)$. By Convention 21 (2) we have $v \notin \mathsf{fn}(C')$.

$$\frac{\dfrac{1.\ \forall\xi,\sigma.(\ (\xi,\sigma) \models^I C \supset \exists V,\sigma'.\ (M\xi,\sigma) \Downarrow V \ \wedge\ (\xi \cdot v : [V]\sigma') \models^I C'[\mathtt{in}_1(v)/u]) \quad \text{(prem)}}{2.\ \forall\xi,\sigma.(\ (\xi,\sigma) \models^I C \supset \exists V,\sigma'.\ (\mathtt{in}_1(M)\xi,\sigma) \Downarrow \mathtt{in}_1(V) \ \wedge\ (\xi \cdot v : [V] \cdot u : [\mathtt{in}_1(V)],\ \sigma') \models^I C' \quad \text{(Lemma 36-1(a))}}}{3.\ \forall\xi,\sigma.(\ (\xi,\sigma) \models^I C \supset \exists V,\sigma'.\ (\mathtt{in}_1(M)\xi,\sigma) \Downarrow \mathtt{in}_1(V) \ \wedge\ (\xi \cdot u : [\mathtt{in}_1(V)]\sigma') \models^I C' \quad \text{(Lemma 35-2)}}$$

$[Case]$ 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 $u \notin \mathsf{dom}(\Gamma,\Delta)$. We infer:

$$\frac{1. \quad - \quad \text{(no premise)}}{2.\quad \mathcal{M} \models^I C[!x/u] \quad \supset \quad \mathcal{M} \cdot u : ([\![!x]\!]\mathcal{M} \cdot I) \models^I C \quad \text{(Lemma 36-1(a))}}$$

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

$$\frac{\dfrac{1.\quad \forall\xi,\sigma.\ (\ (\xi,\sigma) \models^I C \supset \exists V.((M\xi,\sigma) \Downarrow (V,\sigma') \wedge (\xi \cdot m : [V],\sigma') \models^I C'[m/!x][()/u])\ ) \quad \text{(prem)}}{2.\quad \forall\xi,\sigma.\ (\ (\xi,\sigma) \models^I C \supset \exists V,\sigma'.((M\xi,\sigma) \Downarrow (V,\sigma') \ \wedge\ (\xi \cdot m : [V] \cdot u : (),\ \sigma'[x \mapsto [V]]) \models^I C')\ ) \quad \text{(Lemma 36-1(a)/2)}}}{3.\quad \forall\xi,\sigma.\ (\ (\xi,\sigma) \models^I C \supset \exists\sigma''.((x := (M\xi),\sigma) \Downarrow ((),\sigma'') \wedge (\xi \cdot u : (),\sigma'' \models^I C'))\ ) \quad \text{(Lemma 35-2)}}$$

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.	$\models \{A^{\text{-}xi} \wedge \forall j \lneq i.B(j)[x/u]\}\ \lambda x.M\ \{B(i)^{\text{-}x}\}$	(prem)
2.	$\models \{\mathsf{T}\}\ \lambda x.M :_u\ \{A^{\text{-}xi} \supset \forall i^{\mathsf{Nat}}.((\forall j \lneq i.B(j)[x/u]) \supset B(i)^{\text{-}x})\}$	($\wedge$-$\supset$, $\mathsf{Aux}_\forall$)
3.	$\forall\xi^{\Gamma;\Delta}.\ \forall\mathbf{v}^{\alpha}.\ (\ \xi \cdot x : \mathbf{v} \cdot u : [(\lambda y.M)\xi \cdot x : \mathbf{v}] \models^I A^{\text{-}xi} \supset \forall i^{\mathsf{Nat}}.((\forall j \lneq i.B(j)[x/u]) \supset B(i)^{\text{-}x})\ )$	(2 expanded)
4.	$\forall\xi^{\Gamma;\Delta}.(\ \xi \cdot x : [W\xi] \cdot u : [(\lambda y.M)\xi \cdot x : [W\xi]] \models^I A^{\text{-}xi} \supset \forall i^{\mathsf{Nat}}.((\forall j \lneq i.B(j)[x/u]) \supset B(i)^{\text{-}x})\ )$	($\forall$-inst)
5.	$\forall\xi^{\Gamma;\Delta}.\ (\ \xi \cdot \cdot x : [\mu x.\lambda.M\xi] \cdot u : [\mu x.\lambda.M\xi] \models^I A^{\text{-}xi} \supset \forall i^{\mathsf{Nat}}.((\forall j \lneq i.B(j)[x/u]) \supset B(i)^{\text{-}x})\ )$	(Lemma 12)
6.	$\forall\xi^{\Gamma;\Delta}.\ (\ \xi \cdot \cdot x : [\mu x.\lambda.M\xi] \cdot u : [\mu x.\lambda.M\xi] \models^I A^{\text{-}xi} \supset \forall i^{\mathsf{Nat}}.(\forall j \lneq i.B(j) \supset B(i)^{\text{-}x})\ )$	(Lemma 36-1(a))
7.	$\forall\xi^{\Gamma;\Delta}.\ (\ \xi \cdot \cdot u : [\mu x.\lambda.M\xi] \models^I A^{\text{-}xi} \supset \forall i^{\mathsf{Nat}}.(\forall j \lneq i.B(j) \supset B(i)^{\text{-}x})\ )$	(Lemma 35-2)
8.	$\forall\xi^{\Gamma;\Delta}.\ (\ \xi \cdot \cdot u : [\mu x.\lambda.M\xi] \models^I A^{\text{-}xi} \supset \forall i^{\mathsf{Nat}}.B(i)^{\text{-}x}\ )$	(mathematical induction)

This concludes the proof of Theorem 33.

6 Reasoning Examples

6.1 Deriving Hoare Logic for Total Correctness

We first embed the standard proof rules of Hoare logic for total correctness with recursive procedures [21] in the present logic. The main interest in the embedding is a lucid analysis it offers on the known proof rules for basic imperative idioms, notably the proof rules for assignment, loops and recursions. The syntax of programs is given as follows. Let $p, q, \ldots$ range over procedure labels.

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

In $\mathtt{proc}\ p = P\ \mathtt{in}\ Q$, a procedure body P is named p, where we allow calls to p to occur in P. The reduction is rules are standard [38] 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 the imperative PCF. As a simple programming example, the following gives an imperative version of the familiar recursive factorial.

$$\begin{array}{l} \mathtt{proc}\ p\ = (\mathtt{if}\ !x \gneq 0\ \mathtt{then}\ (y := !x \times !y\ ;\ x := !x - 1\ ;\ \mathtt{call}\ p))\ \mathtt{else\ skip} \\ \mathtt{in}\ (y := 1\ ;\ \mathtt{call}\ p) \end{array}$$

A program with the same behaviour can be written using a while loop:

$$y := 1\ ;\ \mathtt{while}\ !x \gneq 0\ \mathtt{do}\ (y := !x \times !y\ ;\ x := !x - 1)$$

To specify and reason about these and other programs written in this simple procedural language, we consider a program logic which guarantees total correctness. Formulae, still ranged over by $C, C', \ldots$, are a proper subset of those of Section 3, having only natural numbers and references to them as data types and missing evaluation formulae. For reference we present the grammar of terms and formulae which we shall be using in this subsection. Let $\star \in \{\wedge, \vee, \supset\}$ and $\mathcal{Q} \in \{\forall, \exists\}$ below.

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

Note terms have the same grammar as expressions used in the language except the former add auxiliary (function) variables, exclusively ranged over by $i, j, \ldots$. This allows us to use program expressions as terms in formulae, just as in the standard Hoare logic.

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. Since a procedure can call itself from inside, guaranteeing total correctness demands the use of a sound induction principle.

Fig. 8 Hoare Logic with Recursive Procedure (total correctness)

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

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

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

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

Figure 8 presents the standard proof rules [21]. For simplicity of presentation, we use a single recursion in [*RecPro*] and mathematical induction in [*While*] and [*RecPro*] (the treatment of multiple recursion and well-founded induction in the embedding does not pose any technical difficulty). In [*While*] and [*RecPro*], we assume i, j are auxiliary and only occur in mentioned formulae (in particular the holes in $C(i)$ exhaust i). Among possible structural rules, we list Kleymann's strengthened consequence rule [21] (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) names occurring in C_0 and C_0', while $\vec{j}$ (resp. $\vec{y}$) are the vector of fresh names of the same length as $\vec{i}$ (resp. $\vec{x}$).

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

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

Note all commands have the unit type. When M has the unit type in $\{C\}M:_u\{C'\}$, we can safely ignore the anchor u by $C'[()/u^{\mathsf{Unit}}]\equiv C'$ (cf. §3.3), so that we shall often write $\{C\}M\{C'\}$ for $\{C\}M:_u\{C'\}$ with $u\notin\mathsf{fn}(C')$, without loss of generality, just like a Hoare triple. The judgement $\Sigma\vdash\{C\}P\{C'\}$ is now translated as:

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

Since formulae are simply the subset of those for the imperative PCF, we have only to use injection. One may also observe, if we use the standard model of number theory [24, §3.1] for Hoare logic, then validity of formulae also coincide for this subset of formulae.Thus, under this model, semantics of judgement and that of its embedding also coincide.

Thus the remaining task is to embed the proof rules. A pleasant fact is that each rule in Figure 8 has a clean encoding into the logic for imperative PCF. We show the embedding for each rule one by one. For clarity we do not mention $[\![\Sigma]\!]$ when it is insignificant for a given rule. For `skip`, observing $u \notin \mathsf{fn}(C)$ and without a premise:

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

For [*AssignH*] and other rules, we shall use the following rule derivable from the proof rules in Figure 4.

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

For derivablity we use induction on e, The base cases (for constants (numbers) and dereferences) are direct from the axioms. For $e = \mathbf{op}(e_1,...,e_n)$, By the inductive hypothesis, we already have $\{C[e_i/m_i]\}e_i :_{m_i} \{C\}$. Accumulating, we obtain:

$$\{C[\mathbf{op}(m_1,...,m_n)/u][\vec{e}/\vec{m}]\}\ \ e_n :_{m_n}\ \{C[\mathbf{op}(m_1,...,m_n)/u]\} \tag{6.1}$$

By substitution (noting $\vec{m}$ are fresh and pairwise distinct) and by [*Op*] we are done.

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

1.	$\{\, C[m/!x][e/m]\,(\stackrel{\text{def}}{=} C[e/!x])\,\}\ e :_m\ \{\, C[m/!x][()/u]\,\}$	(Simple)
2.	$\{C[e/!x]\}\ x := e :_u\ \{C\}$	(Assign)

Above by definition u is not in C. Next we derive [*Seq*] from [*App*] and [*Abs*].

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

[*IfH*] is derived as follows.

1.	$\{C\}e :_b \{b = e \wedge C\}$	(Simple)
2.	$\{C \wedge e\}[\![P]\!]\{C'\}$	(premise)
3.	$\{C \wedge e = \mathbf{t}\}[\![P]\!]\{C'\}$	(2, (3.8) in page 15, Consequence)
4.	$\{C \wedge \neg e\}[\![Q]\!]\{C'\}$	(premise)
5.	$\{C \wedge e = \mathbf{f}\}[\![Q]\!]\{C'\}$	(4, (3.9) in page 15, Consequence)
6.	$\{C\}\texttt{if}\ e\ \texttt{then}\ [\![P]\!]\ \texttt{else}\ [\![Q]\!]\{C'\}$	(3, 5, If)

For $[\mathit{While}]$, we translate the induction on loop to the induction on recursion. The following rule (immediately derivable from $[\mathit{App}]$) is convenient.

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

Further we let:

$$\begin{aligned} B(x,i) &\stackrel{\text{def}}{=} \{C \wedge e' = i\}\ x \bullet ()\ \{C\ \wedge\ \neg e\} \\ B'(x,i) &\stackrel{\text{def}}{=} \forall j \lneq i.\ B(x,j) \end{aligned}$$

The inference follows. Below and henceforth (e3) etc. are axioms from §3.3, indicating their use in the consequence rule. Line 4 below uses $\exists j \lneq i.C \stackrel{\text{def}}{=} \exists j.(j \lneq i \wedge C)$.

1. $\{C \wedge e \wedge e' = n\}\,[\![P]\!]\,\{C \wedge e' \lneq n\}$ (with $C \wedge e \supset e' \gneq 0$)	(premise)
2. $\{B'(w,n) \wedge C \wedge e' \lneq n\}\ w :_m\ \{B'(m,n) \wedge C \wedge e' \lneq n\}$	(Var)
3. $\{B'(w,n) \wedge C \wedge e' \lneq n\}\ w :_m\ \{B'(m,n)\}$	(Consequence)
4. $\{B'(w,n) \wedge C \wedge e' \lneq n\}\ w :_m\ \{\{\exists j \lneq n.(C \wedge e' = j)\}\, w \bullet ()\,\{C\ \wedge\ \neg e\}\}$	(e3, e5)
5. $\{B'(w,n) \wedge C \wedge e' \lneq n\}\ w :_m\ \{\{C \wedge e' \lneq n\}\, w \bullet ()\,\{C\ \wedge\ \neg e\}\}$	(e8)
6. $\{B'(w,n) \wedge C \wedge e' \lneq n\}\, w :_m \{\{B'(w,n) \wedge C \wedge e' \lneq n\}\, w \bullet ()\,\{B'(w,n) \wedge C \wedge \neg e\}\}$	(e7)
7. $\{B'(w,n)\ \wedge\ C\}\ w()\ \{C\ \wedge \neg e\}$	(Run)
8. $\{B'(w,n) \wedge C \wedge e \wedge e' = n\}\ [\![P]\!]\ ;\ w()\ \{C\ \wedge \neg e\}$	(1, Constancy, 5, Seq)
9. $\{C \wedge \neg e \wedge e' = n\}\ ()\ \{C \wedge \neg e \wedge e' = n\}$	(Unit)
10. $\{B'(w,n) \wedge C \wedge \neg e \wedge e' = n\}\ ()\ \{C \wedge \neg e\}$	(Consequence)
11. $\{B'(w,n) \wedge C \wedge e' = n\}$ `if` e `then` $[\![P]\!];(w\,())$ `else` $()\ \{C \wedge \neg e\}$	(8, 10, IfH)
12. $\{B'(w,n)\}\ \lambda()$.`if` e `then` $[\![P]\!];(w\,())$ `else` $() :_u\ \{B(u,n)\}$	(Abs)
13. $\{\mathsf{T}\}\ \mu w.\lambda()$.`if` e `then` $[\![P]\!];(w\,())$ `else` $() :_u\ \{\forall i.B(u,i)\}$	(Rec)
14. $\{\mathsf{T}\}\mu w.\lambda()$.`if` e `then` $[\![P]\!];(w())$ `else` $() :_u \{\{\exists i.(C \wedge e' = i)\}x \bullet ()\{C \wedge \neg e\}\}$	(e3)
15. $\{\mathsf{T}\}\ \mu w.\lambda()$.`if` e `then` $[\![P]\!];(w\,())$ `else` $() :_u\ \{\ \{C\}\ x \bullet ()\ \{C\ \wedge\ \neg e\}\ \}$	(e8)
16. $\{C\}\ [\![$`while` e `do` $P]\!]\ \{C\ \wedge \neg e\}$	(Run)

The inference above (which may be read from the bottom to the top) shows how canonically the proof rule for the total correctness of the while loop arises from the induction principle of $[\mathit{Rec}]$ rule, adapted to the encoded while program. One may also observe the derivation immediately extends to the proof rule for the loop based on well-founded induction, using $[\mathit{Rec\text{-}Wf}]$ in Figure 6, Page 22. The embedding also suggests that the analytical nature of the proof rules of the logic for the imperative PCF.

We now turn to the rules for procedure. [*Call*] is simple:

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

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

1.	$\{[\![\Sigma]\!] \wedge \{\exists j \lneq i.C(j)\}p\bullet()\{C'\} \wedge C(i)\}\ [\![P]\!]\ \{C_0\}$	(Assumption)
2.	$\{[\![\Sigma]\!] \wedge \forall j^{\mathsf{Nat}}.\{j \lneq i \wedge C(j)\}\ p\bullet()\ \{C'\} \wedge C(i)\}\ [\![P]\!]\ \{C_0\}$	(e3)
3.	$\{[\![\Sigma]\!] \wedge \forall j^{\mathsf{Nat}} \lneq i.\{C(j)\}p\bullet()\{C'\} \wedge C(i)\}\ [\![P]\!]\ \{C_0\}$	(e5)
4.	$\{[\![\Sigma]\!] \wedge \forall j^{\mathsf{Nat}} \lneq i.\{C(j)\}p\bullet()\{C'\}\}\ \lambda().[\![P]\!] :_m\ \{\{C(i)\}\ m\bullet()\ \{C_0\}\}$	(Abs)
5.	$\{[\![\Sigma]\!]\}\ \mu p.\lambda().[\![P]\!] :_m\ \{\forall i.\{C(i)\}\ m\bullet()\ \{C_0\}\}$	(Rec)
6.	$\{[\![\Sigma]\!]\}\ \mu p.\lambda().[\![P]\!] :_m\ \{\{\exists i.C(i)\}\ m\bullet()\ \{C_0\}\}$	(e3)
7.	$\{[\![\Sigma]\!] \wedge \{\exists i.C(i)\}\ p\bullet()\ \{C_0\} \wedge C\}\ [\![Q]\!]\ \{C'\}$	(Assumption)
8.	$\{[\![\Sigma]\!]\}\ \lambda p.[\![Q]\!] :_n\ \{\forall p.\{\exists i.C(i)\}\ p\bullet()\ \{C_0\} \supset \{C\}n\bullet p\{C'\}\}$	(Abs)
9.	$\{[\![\Sigma]\!] \wedge C\}\ (\lambda p.[\![Q]\!])(\mu p.\lambda().[\![P]\!])\ \{C'\}$	(6, 8, App)

Lines 2 and 3 above use $\exists j \lneq i.C \stackrel{\text{def}}{=} \exists j.(j \lneq i \wedge C)$ and $\forall j \lneq i.C \stackrel{\text{def}}{=} \forall j.(j \lneq i \supset C)$. 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, with a direct suggestion to its extension to arbitrary well-founded induction.

Since [*Consequence-Aux*] in Figure 8 is the precise mirror of [*Consequence-Aux*] Figure 4, we can conclude as follows, assuming the standard model for Hoare logic.

Proposition 39. (embedding of Hoare logic for total correctness) $\Sigma \vdash \{C\}P\{C'\}$ *implies* $\vdash \{[\![\Sigma]\!] \wedge \{C\}\}[\![P]\!]\{C'\}$.

6.2 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 8 in § 6.1.

1.	$\{!x\times 2=6\}\ y :=\ !x \times 2\ \{!y=6\}$	(AssignH)
2.	$\{!x=3\}\ y :=\ !x \times 2\ \{!y=6\}$	($!x = 3 \supset !x\times 2 = 6$, 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.	$\{z = i\} z \times 2 :_m \{m = i \times 2\}$	(Var, Num, $\times$)
2.	$\{\top\} \lambda z.(z \times 2) :_v \{Double(v)\}$	(Abs)
3.	$\{\top\} x := \lambda z.(z \times 2) \{Double(!x)\}$	(Promote, Assign)

Next we infer a function with dereference (cf. Section 2).

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

Next we use the following slightly enriched version of [*Seq*] of Figure 8 in § 6.1.

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

Using [*Seq'*], we can plug-in the post and pre-conditions of the conclusions of the last two derivations.

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

By a similar reasoning, we obtain (using C in § 3.2, page 10, for the post-condition):

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

Combining this and $\{\top\}\lambda y.(!x)y :_u \{\{C \wedge\ !w = 0\} u \bullet 3 \searrow 6 \{C \wedge\ !w = 1\}\}$ (which uses D in § 3.2) by [*Seq'*] gives us:

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

Finally we reason for M in (2.6), Section 2, Page 6, using $A(fg)$ given there.

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

Similarly we can derive $\{A(fg)\}M :_u \{Odd\_then\_Even(u, x)\}$, and, through [*Pre*-$\wedge$], finally $\{A(fg)\}M :_u \{Even\_then\_Odd(u, x) \wedge Odd\_then\_Even(u, x)\}$.

6.3 Two Programming Examples Revisited

Now we come back to the two programming examples in Introduction. First, the specification for `closureFact` is now easy to make precise, for instance as follows.

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

This assertion concisely describes the program's observable behaviour: we invite the reader to compare it with its informal description in Introduction.

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:

$$0! = 1 \quad \wedge \quad \forall i^{\mathsf{Nat}}.(i+1)! = (i+1)\times i!. \tag{6.2}$$

We further let $A(g,i) \stackrel{\text{def}}{=} \{\mathsf{T}\}g\bullet()\searrow z\{z = i!\}$ and $B'(f,i) \stackrel{\text{def}}{=} \forall j^{\mathsf{Nat}} \lneq i.B(f,i)$. 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 4.

1.	$\{1 = x!\}\ 1 :_m \ \{m = x!\}$	(Const)
2.	$\{B'(f,i)\wedge x = i\wedge x = 0\}\ 1 :_m \ \{m = x!\}$	((6.2), 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.	$\{B'(f,i)\wedge x = i\wedge x \neq 0\}\ f :_m \ \{B'(m)\}$	(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, (6.2), 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.	$\{B'(f,i)\wedge x = i\}\ \texttt{if}\ x = 0\ \texttt{then}\ M_1\ \texttt{else}\ M_2 :_u \ \{A(!y,x)\}$	(4, 13, If-Hoare)
15.	$\{B'(f,i)\}\ \lambda x^{\mathsf{Nat}}.\texttt{if}\ x = 0\ \texttt{then}\ M_1\ \texttt{else}\ M_2 :_u$ $\{\ \forall x^{\mathsf{Nat}}.\ \{x = i\}\, u\bullet i\{\{\mathsf{T}\}\, !y\bullet()\searrow z\{z = i!\}\}\ \}$	(Abs)
16.	$\{B'(f,i)\}\ \lambda x^{\mathsf{Nat}}.\texttt{if}\ x = 0\ \texttt{then}\ M_1\ \texttt{else}\ M_2 :_u$ $\{\ \forall x^{\mathsf{Nat}}.(x = i\ \supset\ B(u,x))\}$	(e5, Consequence)
17.	$\{B'(f,i)\}\ \lambda x^{\mathsf{Nat}}.\texttt{if}\ x = 0\ \texttt{then}\ M_1\ \texttt{else}\ M_2 :_u \ \{B(u,i)\}$	(Consequence)
18.	$\{\mathsf{T}\}\ \mu f^{\mathsf{Nat}\Rightarrow\mathsf{Unit}}.\lambda x^{\mathsf{Nat}}.\texttt{if}\ x = 0\ \texttt{then}\ M_1\ \texttt{else}\ M_2 :_u \ \{\forall i^{\mathsf{Nat}}.B(u,i)\}$	(Rec)

Next is an assertion describing basic aspects of `scheduler`'s behaviour (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).

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

In the assertion, i is implicitly universally quantified. The assertion says: given, for example, 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 $\{C_0\}\, \texttt{scheduler}\; l\, \{C_2\}$ 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:

$$\{\mathsf{T}\}\ \texttt{scheduler} :_u \{\mathsf{Sched}(u)\} \tag{6.3}$$

We now show how (6.3) can be derived. For brevity let us set (the definition of `map` is given in Introduction):

$$
\begin{array}{r l}
\texttt{scheduler} & \stackrel{\text{def}}{=} \texttt{map app} \\
\texttt{app} & \stackrel{\text{def}}{=} \lambda z^{(\alpha \Rightarrow \mathsf{Unit}) \times \alpha}. (\pi_1(z) \pi_2(z)),
\end{array}
$$

These components have assertions of their own (for the map, we omit the result values, only considering the cases when the target of the function is Unit).

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

We can then derive:

$$\{\mathsf{T}\}\, \texttt{map} :_m \{\forall f^{\beta \Rightarrow \mathsf{Unit}}\, \mathsf{Map}'(m,f)\} \tag{6.4}$$

$$\{\mathsf{T}\}\, \texttt{app} :_n \{\forall z^{(\alpha \Rightarrow \mathsf{Unit}) \times \alpha}.\ \mathsf{App}(n,z)\} \tag{6.5}$$

The derivation of the judgement for `map` in (6.4) needs an inductive reasoning (closely following the one given in [20, § 5]), which we omit from the present version. The inference for the judgement (6.5) 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, Consequence)
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, Consequence)
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, Consequence)
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.	$\{\mathsf{T}\}\ \lambda z.\pi_1(z)\pi_2(z) :_n\ \{\forall z.\ \mathsf{App}(n,z)\}$	(e5)

Assuming the judgement for map is also deduced, we now derive (6.3) from (6.4) and (6.5). We uses the following inference on validity.

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

We can now deduce:

1.	$\{\mathsf{T}\}$ `map` $:_m\ \{\forall f.\, \mathsf{Map}'(m,f)\}$	(6.4)
2.	$\{\mathsf{T}\}$ app $:_n\ \{\forall z.\mathsf{App}(n,z)\}$	(6.5)
3.	$\{\forall f.\, \mathsf{Map}'(m,f)\}$ app $:_n\ \{\forall f.\, \mathsf{Map}'(m,f) \wedge \forall z.\mathsf{App}(n,z)\}$	(Constancy)
4.	$\{\forall f.\, \mathsf{Map}'(m,f)\}$ app $:_n\ \{\{\mathsf{T}\} m \bullet n \searrow u \{\mathsf{Sched}(u)\}\}$	(Consequence)
5.	$\{\forall f.\, \mathsf{Map}'(m,f)\}$ (`map app`) $:_u\ \{\mathsf{Sched}(u)\}$	(1, 4, App)

where, in Line 4, we use the above inference on validity for the entailment. We have reached (6.3).

7 Discussion

7.1 Related Work

The logic and proof rules presented in this work are built on many preceding works on semantics and logics of programming languages. For typed and untyped λ-calculi, equational logics have been studied since the classical work by Church and Curry. Among many studies, LCF [10] augments the standard equational theories of the λ-calculus with Scott's fixed-point induction. In this tradition, we find program logics which can treat higher-order features and various data types elegantly, but only as long as they are confined to stateless computation. This is due to the difficulties inherent in applying syntactic equations based on reductions, such as $\beta\eta$-conversions, to stateful languages.

Compositional program logics for imperative languages have been studied extensively since Hoare's work (which in turn is partially based on earlier work by Floyd [9] and Naur [30]). The program logics studied in this tradition directly treat a basic feature of imperative programming languages, state change. However, higher-order procedures and data structures have not been fully captured in Hoare's logic and its extensions, in spite of their prominent status in both theories and practice. A central reason for the difficulty lies in the notion of semantics of judgements in Hoare logic centring on a pair of first-order states, which is hard to adapt to higher-order behaviours. Relatedly, it is a standard custom to include finite models for assertions in Hoare logic and its extensions, as well as the standard models of (say) Peano arithmetic, which is impossible to maintain in the presence of higher-order procedure, as Clarke has shown in [7]. Thus, reflecting the richness of the universe to be modelled, we need to think about a different class of logical universes than what have been considered so far for Hoare Logic. We regard deep investigation into models of the present logic and its ramifications, addressing both the understanding of their mathematical properties and their practical use, as an important further topic.

Some of the central elements of the proposed program logic come from these two traditions. Direct treatment of operators for (higher-order) data types in assertions is inherited from logics for λ-calculi; whereas the use of pairs of assertions to describe state change is progeny of logics for imperative languages. The universe of discourse now contains higher-order behaviours themselves. This is essential for general description of higher-order programs, and is made possible by the systematic use of names in assertions, which comes from logics for the π-calculus [18, 19, 27]. Names have been used in Hoare Logic since an early work by Kowaltowski [22], and are found in the work by Von Oheimb [37], Leavens and Baker [23], Abadi and Leino [4], Bierman and Parkinson [6], and others, for treating parameter passing and return values. These authors however 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. It would be an interesting subject of study to relate (and possibly integrate) reasoning techniques developed in these works to those developed in the present one. Another merit of the proposed approach is its tight correspondence with semantics of the target programming language, as indicated by Proposition 31 in Section 5.

Reynolds, O'Hearn and others [31, 35] study extensions of Hoare logic in which new logical connectives are used for reasoning about low-level operations such as garbage collection. Because of a difference in levels of abstraction (for example, Reynolds [35] treats references to memory cells as numerals), direct technical comparisons are difficult. Different aspects of low-level computation may be elucidated through a precise logical articulation of imperative higher-order functions in the present logic (cf. [11, 29, 36]). Studies based on these and other methods will contribute to an integrated understanding of diverse programming constructs, including both low-level and high-level ones.

Dynamic Logic [14], introduced by Pratt [33] and studied by Harel and others [13], 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.

A logic for higher-order imperative procedures, called specification logic, is developed by Reynolds for Idealised Algol [34]. Apart from the difference of the shape of assertions and target languages, his logic [34] does not have compositional proof rules.

7.2 Extensions and Further Topics

The recursion rule given in Figure 4 is for total correctness, and is used for embedding and justifying the known proof rules for loop and recursive procedures that guarantee termination [13]. Another recursion rule, studied in [19], is for partial correctness and has the following shape.

$$[\mathit{Rec\text{-}Partial}]\,\frac{\{A^{\text{-}x} \wedge B[x/u]\}\,\lambda y.M :_u \{B^{\text{-}x}\} \quad B \text{ admissible at } u \quad A \supset \exists u.B}{\{A\}\,\mu x.\lambda y.M :_u \{B\}}$$

"Admissibility" in the above rule intuitively says that B is about partial correctness (see [19, §5.3]). This rule can encode and justify the existing rules for loops and recursive procedures for partial correctness. There is a valid judgement which can be proved with [*Rec-Partial*] but not without.

For simplicity, we have omitted polymorphisms and recursive types in the present paper. Their integration is straightforward following our previous study [20]. The type structure in § 3.1 is restricted so that a reference type is never carried by other types. Lifting this restriction leads to a consistent extension of the present logic. Another extension is the introduction of local state. As is well-known, a simple proof rule suffices for local references which are used only during a run of a command. However ML-like languages allow extrusion of a local datum [26] beyond its original scope. These extensions are treated in sequels to the present paper.

The use of side-effect-free expressions on assignment has been one of the basic principles in compositional program logics. Freedom from side effects is however hard to maintain in the higher-order setting because of complex interplay between higher-order procedures. The clean embedding of Hoare's assignment rule in Section 6 may suggest the presented framework effectively extends the traditional approach while retaining its virtues. Experiment of the possible extensions in the context of an integrated verification framework such as JML [3] would be an interesting subject for further study.

Finally, the origin of the assertions and judgements introduced in the present work is the logic for typed π-calculi [18, 19], 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. [17–19] discuss process logics and their relationship to the program logics in detail.

References

1. C– home page. http://www.cminusminus.org.
2. The Haskell home page. http://haskell.org.
3. The Java Modeling Language (JML) home page. http://www.jmlspecs.org/.
4. Martín Abadi and Rustan Leino. A logic for object-oriented programs. In *Verification: Theory and Practice*, pages 11–41. Springer-Verlag, 2004.
5. K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.
6. Gavin M. Bierman and Matthew J. Parkinson. Separation logic and abstractions. In *POPL'05*, 2005.
7. E Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. In *POPL'79*, pages 129–147, 1979.
8. Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.
9. Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
10. Mike Gordon, Robin Milner, and Christopher Wadsworth. Edinburgh LCF. LNCS 78, Springer Verlag, 78.
11. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI'02*. ACM, 2002.
12. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
13. David Harel. Proving the correctness of regular deterministic programs. *TCS*, 12(16), 1980.
14. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logigc*. The MIT Press, 2000.
15. Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
16. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
17. Kohei Honda. Sequential process logics: Soundness proofs. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, November 2003. Typescript, 50 pages.
18. Kohei Honda. From process logic to program logic. In *Proc. ICFP'04*. ACM Press, 2004. A short version of Sections 1–3 and 6 in [19].
19. Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2004. Typescript, 234 pages.
20. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP'04*. ACM Press, 2004.
21. Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.
22. Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.
23. 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.

24. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
25. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Proc. POPL'88*, 1988.
26. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1), 1992.
27. Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *TCS*, 114:149–171, 1993.
28. Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML.* MIT Press, 1990.
29. 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.
30. P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.
31. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, 2004.
32. Benjamin C. Pierce. *Type Systems and Programming Languages*. MIT Press, 2002.
33. 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.
34. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
35. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, 2002.
36. Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
37. David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13), 2001.
38. Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.