Logical Reasoning for Higher-Order Functions with Local State

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

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

Abstract. We introduce an extension of Hoare logic for call-by-value higher-order functions with ML-like local reference generation. Local references may be generated dynamically and exported outside their scope, may store higher-order functions and may be used to construct complex mutable data structures. This primitive is captured logically using a predicate asserting reachability of a reference name from a possibly higher-order datum and quantifiers over hidden references. The logic enjoys three completeness properties: relative completeness, a logical characterisation of the contextual congruence and derivability of characteristic formulae. The axioms for reachability and local invariants play a fundamental role in reasoning about non-trivial programs combining higher-order procedures and dynamically generated references.

1 Introduction

New reference generation, embodied for example in ML's `ref`-construct, is a highly expressive programming primitive. The key functionality of this construct is, firstly, to induce local state by generating a fresh reference inaccessible from the outside. Consider the following program:

$$\texttt{Inc} \stackrel{\text{def}}{=} \texttt{let}\, x = \texttt{ref}(0)\, \texttt{in}\, \lambda().(x :=\!x+1;\, !x) \tag{1}$$

where "$\texttt{ref}(M)$" returns a fresh reference whose content is the value which M evaluates to; "$!x$" means dereferencing the imperative variable x; and ";" is sequential composition. In (1), a reference with content 0 is newly created, but never exported to the outside. When the anonymous function in `Inc` is invoked, it increments the content of a local variable x, and returns the new content. The procedure returns a different result at each call, whose source is hidden from external observers. This is different from $\lambda().(x :=\!x+1;\, !x)$ where x is globally accessible.

Secondly, local references may be exported outside of their original scope and be shared. Consider the following program from [25, § 6]:

$$\texttt{incShared} \stackrel{\text{def}}{=} a := \texttt{Inc}; b :=\!a; z_1 := (!a)(); z_2 := (!b)(); (!z_1 + !z_2) \tag{2}$$

This program returns 3. To understand the behaviour of `incShared`, we must capture the sharing of x between the procedures assigned to a and b. The scope of x is originally restricted to $!a$ but gets extruded to $!b$. If we replace $b :=\!a$ by $b := \texttt{Inc}$, two separate

instances of Inc are assigned to a and b, and the final result is 2. Controlling sharing by local reference is essential to writing concise algorithms that manipulate mutable data structures, but complicates formal reasoning, even for relatively small programs [8, 18].

Thirdly, through information hiding, local references can be used for efficient implementation of highly regular observable behaviour. The following program, taken from [25, § 1], called memFact, is a simple memoised factorial.

$$\texttt{let}\, a = \texttt{ref}(0)\ b = \texttt{ref}(1)\, \texttt{in}\, \lambda x.\texttt{if}\ x = !a\ \texttt{then}\ !b\ \texttt{else}\ (a := x; b := \texttt{fact}(x); !b)$$

Here fact is the standard factorial function. To external observers, memFact behaves purely functionally. The program implements a simple case of memoisation: when memFact is called with a stored argument in a, it immediately returns the stored value $!b$ without calculation. If x differs from what is stored at a, the factorial fx is calculated and the new pair is stored. The reason why memFact is indistinguishable from the pure factorial function can be understood through the following *local invariant* [25]:

> *Throughout all possible invocations of* memFact*, the content of b is the factorial of the content of a.*

Such local invariants capture one of the basic patterns in programming with local state, and play a key role in preceding studies of operational reasoning about program equivalence in the presence of local state [15, 23, 25, 30].

As a further example of local invariants, this time involving mutually recursive stored functions, consider the following program:

$$\begin{aligned}\texttt{mutualParity} \stackrel{\text{def}}{=} x := \lambda n.\texttt{if}\ n = 0\ \texttt{then f else not}((!y)(n-1));\\ y := \lambda n.\texttt{if}\ n = 0\ \texttt{then t else not}((!x)(n-1))\end{aligned}$$

After running mutualParity, the application $(!x)n$, returns true if n is odd, false if not, and $(!y)n$ acts dually. But since x and y are free, another program may prevent mutualParity from functioning correctly by inappropriate assignment to x or y. With local state, we can avoid unexpected interference at x and y.

$$\texttt{safeOdd} \stackrel{\text{def}}{=} \texttt{let}\, x = \texttt{ref}(\lambda n.\texttt{t})\, y = \texttt{ref}(\lambda n.\texttt{t})\, \texttt{in}\, (\texttt{mutualParity}; !x) \quad (3)$$

$$\texttt{safeEven} \stackrel{\text{def}}{=} \texttt{let}\, x = \texttt{ref}(\lambda n.\texttt{t})\, y = \texttt{ref}(\lambda n.\texttt{t})\, \texttt{in}\, (\texttt{mutualParity}; !y) \quad (4)$$

(Here $\lambda n.\texttt{t}$ can be any initialising value.) Now that x, y are inaccessible, the programs behave like pure functions, e.g. $\texttt{safeOdd}(3)$ always returns true without any side effects. In this case, the invariant says that *throughout all possible invocations,* $!x$ *is a procedure which checks if its argument is odd, provided y stores a procedure which does the dual, whereas* $!y$ *is a procedure which checks if its argument is even, whenever x stores a dual procedure*. Later we present general reasoning principles for local invariants which can verify these two and many other non-trivial examples [14, 15, 18, 23, 25].

This paper studies a Hoare logic for imperative higher-order functions with dynamic reference generation, a core part of ML-like languages. Our aim is to identify basic logical primitives needed to capture precisely the semantics of local state, on the basis of a stratification of logics for sequential higher-order functions in our preceding

works [2, 10, 12, 13]. For this purpose we introduce two new logical primitives, one for reachability of references from an arbitrary datum and another for quantifying hidden references (§ 2.2). This leads to a simple proof system for reference generation, which can assert and derive desired properties for programs with significant use of local state from the literature [14, 15, 18, 23, 25] (§ 2.4). The status of these new logical primitives is clarified through soundness and three completeness results, including relative completeness (§ 3). Basic axioms for reachability, hiding and local invariants are studied in § 4. The local invariance axioms capture a common pattern in reasoning about local state, and enable us to verify the examples in [14, 15, 18, 23, 25], including programs discussed above (§5). Comparisons with related work are found in §6. Detailed derivations, large examples and proofs are found in the full version [1].

Acknowledgement We thank Andrew Pitts and Ian Stark for communications on an early version of this paper. The mutual recursion example is due to Bernhard Reus. This work is partially supported by EPSRC GR/T04236, GR/S55545, GR/S55538, GR/T04724, GR/T03208, GR/T03258 and IST-2005-015905 MOBIUS.

2 Assertions for Local State

2.1 A Programming Language

Our target programming language is call-by-value PCF with unit, sums, products and recursive types, augmented with imperative constructs. Let $x, y, \ldots$ range over an infinite set of variables, and $\mathrm{X}, \mathrm{Y}, \ldots$ over an infinite set of type variables. Then types, values and programs are given by:

$$\begin{array}{rcl}
\alpha, \beta & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \mid \mathsf{Ref}(\alpha) \mid \mathrm{X} \mid \mu \mathrm{X}.\alpha \\
V, W & ::= & \mathtt{c} \mid x^{\alpha} \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M \mid \langle V, W \rangle \mid \mathtt{inj}_i^{\alpha+\beta}(V) \\
M, N & ::= & V \mid MN \mid M := N \mid \mathtt{ref}(M) \mid !M \mid \mathtt{op}(\tilde{M}) \mid \pi_i(M) \mid \langle M, N \rangle \mid \mathtt{inj}_i^{\alpha+\beta}(M) \\
& & \mid \ \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \mid \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}}
\end{array}$$

We use standard notation [22] like constants $\mathtt{c}$ (unit $()$, booleans $\mathtt{t}$, $\mathtt{f}$, numbers $\mathtt{n}$ and locations $l, l', \ldots$) and first-order operations $\mathtt{op}$ ($+, -, \times, =, \neg, \wedge, \ldots$). Locations only appear at runtime when references are generated. $\tilde{M}$ etc. denotes a vector and ε the empty vector. A program is *closed* if it has no free variables. We freely use shorthands like $M; N$, $\lambda().M$, and $\mathtt{let}\ x = M\ \mathtt{in}\ N$. Typing is standard: we take the equi-isomorphic approach [22] for recursive types. Nat, Bool and Unit are *base types*. We leave illustration of each construct to standard textbooks [22], except for reference generation $\mathtt{ref}(M)$, the focus of the present study, which behaves as: first M of type α is evaluated and becomes a value V; then a *fresh* reference of type $\mathsf{Ref}(\alpha)$ with initial content V is generated. This behaviour is formalised by the following reduction rule:

$$(\mathtt{ref}(V),\ \sigma) \longrightarrow (\nu\, l)(l,\ \sigma \uplus [l \mapsto V]) \qquad (l \text{ fresh})$$

Above σ is a store, a finite map from locations to closed values, denoting the initial state, whereas $\sigma \uplus [l \mapsto V]$ is the result of disjointly adding a pair (l, V) to σ. The resulting configuration uses a ν-binder, which denotes l fresh. The general form $(\nu\, \tilde{l})(M, \sigma)$

means $\tilde{l}$ (a vector of distinct locations) occur in M and σ (the order is irrelevant). We write (M,σ) for $(\nu\,\varepsilon)(M,\sigma)$. The one-step reduction $\longrightarrow$ over configurations is defined using standard rules [22] except for closure under ν-bindings. A *basis* $\Gamma;\Delta$ is a pair of finite maps, one from variables to non-reference types $(\Gamma,\Gamma',\ldots)$, the other from locations and variables to reference types $(\Delta,\Delta',\ldots)$. $\Theta,\Theta',\ldots$ combine two kinds of bases. The typing rules are standard. Sequents have form $\Gamma;\Delta \vdash M : \alpha$, to be read: M has type α under $\Gamma;\Delta$. We omit empty Γ or Δ . A store σ is typed under Δ, written $\Delta \vdash \sigma$, when, for each l in its domain, $\sigma(l)$ is a closed value which is typed α under Δ, where we assume $\Delta(l) = \mathsf{Ref}(\alpha)$. A configuration (M,σ) is *well-typed* if for some $\Gamma;\Delta$ and α we have $\Gamma;\Delta \vdash M : \alpha$ and $\Delta \vdash \sigma$. Standard type safety holds for well-typed configurations. *Henceforth we only consider well-typed programs and configurations.*

2.2 A Logical Language

The logical language is based on standard first-order logic with equality [17, § 2.8]. It extends the logic [2] with two new primitives. The grammar follows, letting $\star \in \{\wedge,\vee,\supset\}$, $\mathcal{Q} \in \{\exists,\forall,\nu,\overline{\nu}\}$ and $\mathcal{Q}' \in \{\exists,\forall\}$.

$$\begin{array}{rcl} e & ::= & x \mid \mathsf{c} \mid \mathsf{op}(\tilde{e}) \mid \langle e,e'\rangle \mid \pi_i(e) \mid \mathsf{inj}_i(e) \mid\ !e \\ C & ::= & e=e' \mid \neg C \mid C \star C' \mid \mathcal{Q}x^{\alpha}.C \mid \mathcal{Q}'\mathrm{X}.C \mid \{C\}e \bullet e' = x\{C\} \mid [!e]C \mid e \hookrightarrow e' \end{array}$$

The first grammar $(e,e',\ldots)$ defines *terms*; the second *formulae* $(A,B,C,E,\ldots)$. Terms include variables, constants c (unit $()$, numbers n, booleans t, f and locations $l,l',\ldots$), pairing, projection, injection and standard first-order operations. $!e$ denotes the dereference of a reference e. Formulae include standard logical connectives and first-order quantifiers [17], and following [2, 12], quantification over type variables.

Introduced in [13], $\{C\}\ e \bullet e' = x\ \{C'\}$ is the *evaluation formula*, which intuitively says: *If we apply a function e to an argument e' starting from an initial state satisfying C, then it terminates with a resulting value (name it x) and a final state together satisfying C'*. We shall also use a refined form of evaluation formulae, introduced in §2.3. $[!e]C$ is *universal content quantification*, introduced in [2] for treating aliasing. $[!e]C$ (with e of a reference type) says: *Whatever value a program may store in a reference e, the assertion C continues to be valid.*

There are two new logical primitives. First, $\nu x.C$ (*for some hidden reference x, C holds*) and $\overline{\nu}x.C$ (*for each hidden reference x, C holds*) are *hiding-quantifiers* which quantify over reference variables, i.e. x above is of the form $\mathsf{Ref}(\beta)$. They range over hidden references, such as x generated by `Inc` in (1) in § 1. The need for adding these quantifiers is illustrated in §4.1, Proposition 12. The second new primitive is $e_1 \hookrightarrow e_2$ (with e_2 of a reference type), which is the *reachability predicate*. It says: *We can reach the reference denoted by e_2 from a datum denoted by e_1*. We then set its dual [6, 29] as $e\#e' \equiv \neg e' \hookrightarrow e$, which says: *One can never reach a reference e starting from a datum denoted by e'*. $\#$ is used for representing freshness of new references.

Convention. Logical connectives are used with standard precedence/association, using parentheses as necessary to resolve ambiguities. We use truth T (definable as $1=1$) and falsity F (which is $\neg\mathsf{T}$). $x \neq y$ stands for $\neg(x=y)$. $\mathsf{fv}(C)$ (resp. $\mathsf{fl}(C)$) denotes the set of

free variables (resp. locations) in C. Note that x in $[!x]C$ occurs free, while in $\{C\}e \bullet e' = x\{C'\}$ x occurs bound with scope C'. We often write $[!x_1..x_n]C$ for $[!x_1]..[!x_n]C$. $C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$. We write $\tilde{e}\#e$ for $\wedge_i e_i\#e$; $e\#\tilde{e}$ for $\wedge_i e\#e_i$; and $\tilde{e}\#\tilde{e}'$ for $\wedge_{ij} e_i\#e'_j$. Terms are typed starting from variables. A formula is well-typed if all occurring terms are well-typed. *Hereafter we assume all terms and formulae we use are well-typed.* Type annotations are often omitted.

2.3 Assertions for Local State

We explain assertions for programs with local state with examples.

1. The assertion $x = 6$ says x of type Nat is equal to 6. Assuming x has type $\mathsf{Ref}(\mathsf{Nat})$, $!x = 2$ means x stores 2. Consider $x := y; y := z; w := 1$. After its run, we can reach z by dereferencing y, and y by dereferencing x. Hence z is reachable from y, y from x, hence z from x. So the final state satisfies $x \hookrightarrow y \wedge y \hookrightarrow z \wedge x \hookrightarrow z$.
2. Next, assuming w is newly generated, we may wish to say w is *unreachable* from x, to ensure freshness of w. For this we assert $w\#x$, which, as noted, stands for $\neg(x \hookrightarrow w)$. $x\#y$ always implies $x \neq y$. Note that $x \hookrightarrow x$ and $x \hookrightarrow !x$ are tautologies whereas $x\#x \equiv \mathsf{F}$. But $!x \hookrightarrow x$ may or may not hold (since there may be a cycle between x's content and x in the presence of recursive types).
3. We consider reachability in procedures. Assume $\lambda().(x := 1)$ is named f_w and $\lambda().!x$, f_r. Since f_w can write to x, we have $f_w \hookrightarrow x$. Similarly $f_r \hookrightarrow x$. Next suppose $\mathtt{let}\ x = \mathtt{ref}(z)\ \mathtt{in}\ \lambda().x$ has name f_c and z's type is $\mathsf{Ref}(\mathsf{Nat})$. Then $f_c \hookrightarrow z$ (e.g. consider $!(f_c()) := 1$). However x is *not* reachable from $\lambda().((\lambda y.())(\lambda().x))$ since semantically it never touches x.
4. $\lambda().(x :=!x+1;!x)$ named u satisfies: $\forall i^{\mathsf{Nat}}.\{!x = i\}u \bullet () = z\{!x = z \wedge !x = i+1\}$ saying: *invoking the function u increments the content of x and returns that content.*
5. We often wish to say that the write effects of an application are restricted to specific locations. The *located assertion* introduced in [2] is used for this purpose: $\{C\}e \bullet e' = x\{C'\}@\tilde{e}$ where each e_i is of a reference type and does not contain a dereference. $\tilde{e}$ is called *write set*. As an example: $\mathsf{inc}(u,x) \stackrel{\text{def}}{=} \forall i.\{!x = i\}u \bullet () = z\{z =!x = i+1\}@x$ is satisfied by $\lambda().(x :=!x+1;!x)$ named u, saying this function, when invoked, only touches x.
6. Assuming u denotes the result of evaluating `Inc` in the Introduction, we can assert, using the existential hiding quantifier:

$$\nu x.(!x = 0 \ \wedge\ \forall i^{\mathsf{Nat}}.\{!x = i\}u \bullet () = z\{z =!x \wedge !x = i+1\}@x) \tag{5}$$

   which says: there is a hidden reference x storing 0 such that whenever u is invoked, it stores to x and returns the increment of the value in x at the time of invocation.
7. $\lambda n^{\mathsf{Nat}}.\mathtt{ref}(n)$, named u, meets the following specification. Let i, X be fresh.

$$\forall n^{\mathsf{Nat}}.\forall \mathrm{X}.\forall i^{\mathrm{X}}.\{\mathsf{T}\}u \bullet n = z\{\nu x.(!z = n \ \wedge\ z\#i \wedge z = x)\}@\emptyset. \tag{6}$$

   This says that u, when applied to n, will return a hidden reference z whose content is n and which is unreachable from any existing datum; and it has no writing effects to the existing state. Since i ranges over arbitrary data, unreachability of x from each such i indicates x is freshly generated and is not stored in any existing reference.

2.4 Proof Rules

Following Hoare [7], a judgement consists of a program and a pair of formulae, but augmented with a fresh name called an *anchor* [10, 12, 13].

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

The judgement is about total correctness and reads: *If we evaluate M in the initial state satisfying C, then it terminates with a value named by u and a final state, which together satisfy C′*. The same 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). Let $\Gamma;\Delta$ be the minimum basis of M. In $\{C\}\ M :_u \{C'\}$, the name u is the *anchor* of the judgement, which should *not* be in $\mathsf{dom}(\Gamma,\Delta)\cup\mathsf{fv}(C)$; and C is the *pre-condition* and C' is the *post-condition*. The *primary names* are $\mathsf{dom}(\Gamma,\Delta)\cup\{u\}$, while the *auxiliary names* (ranged over by $i,j,k,...$) are those free names in C and C' which are not primary. An anchor is used for naming the value from M and for specifying its behaviour.

The full compositional proof rules are given in Appendix A. Despite our semantic enrichment, all compositional proof rules in [2] syntactically stay as they are, except for adding the following rule for reference generation, with fresh i, X:

$$[Ref]\ \frac{\{C\}\ M :_m \{C'\}}{\{C\}\ \mathtt{ref}(M) :_u \{\nu x.(C'[!u/m]\wedge u\#i^{\mathrm{X}}\wedge u=x)\}}$$

In this rule, $u\#i$ indicates that the newly generated cell u is unreachable from any i of arbitrary type X in the initial state: then the result of evaluating M is stored in that cell.

Reachability is a stateful property: for this reason it is generally not invariant under state change. For example, suppose x is unreachable from y; after running $y := x$, x becomes reachable from y. Hence a rule such as "if $\{C\}\ M :_m \{C'\}$, then $\{C\wedge e\#e'\}\ M :_m \{C'\wedge e\#e'\}$" is unsound. However from the general invariance rule [*Inv*] from [2] below (on the left), which uses the located form of judgement $\{C\}\ M :_u \{C'\}@\tilde{e}$ (understood as located evaluation formulae), we can derive an invariance rule for $\#$, [*Inv*-$\#$].

$$[Inv]\ \frac{\{C\}\ M :_m \{C'\}@\tilde{w}}{\{C\wedge[!\tilde{w}]C_0\}\ M :_m \{C'\wedge C_0\}@\tilde{w}} \qquad [Inv\text{-}\#]\ \frac{\begin{array}{c}\{C\}\ M :_m \{C'\}@x\\ \text{no dereference occurs in } \tilde{e}\end{array}}{\{C\wedge x\#\tilde{e}\}\ M :_m \{C'\wedge x\#\tilde{e}\}@x}$$

In [*Inv*], unlike the existing invariance rules as found in [28], we need no side condition "M does not modify stores mentioned in C_0": C and C_0 may even overlap in their mentioned references, and C does not have to mention all references M may read or write. For [*Inv*-$\#$], we note $[!x]x\#\tilde{e}\equiv x\#\tilde{e}$ is always valid if $\tilde{e}$ contains no dereference $!e$, cf. Proposition 7 3-(5) later. The side condition is indispensable: consider $\{\mathsf{T}\}x := x\{\mathsf{T}\}@x$, which does not imply $\{x\#!x\}x := x\{x\#!x\}@x$.

3 Models, Soundness and Completeness

3.1 Models

We introduce the semantics of the logic based on term models. For capturing local state, models incorporate hidden locations using ν-binders [20]. For example, the Introduc-

tion's `Inc`, named u, is modelled as: $(\nu l)(\{u : \lambda().(l :=\!!l+1;\, !l)\}, \{l \mapsto 0\})$, which says that the appropriate behaviour at is at u, in addition to a hidden reference l storing 0.

Definition 1. An *open model of type* $\Theta = \Gamma;\Delta$, with $\mathsf{fv}(\Delta) = \emptyset$, is a tuple (ξ, σ) where:

- ξ, called *environment*, is a finite map from $\mathsf{dom}(\Theta)$ to closed values such that, for each $x \in \mathsf{dom}(\Gamma)$, $\xi(x)$ is typed as $\Theta(x)$ under Δ, i.e. $\Delta \vdash \xi(x) : \Theta(x)$.
- σ, called *store*, is a finite map from labels to closed values such that for each $l \in \mathsf{dom}(\sigma)$, if $\Delta(l)$ has type $\mathsf{Ref}(\alpha)$, then $\sigma(l)$ has type α under Δ, i.e. $\Delta \vdash \sigma(l) : \alpha$.

When Θ includes free type variables, ξ maps them to closed types, with the obvious corresponding typing constraints. A *model* of type $(\Gamma;\Delta)$ is a structure $(\nu\tilde{l})(\xi, \sigma)$ with (ξ, σ) being an open model of type $\Gamma;\Delta \cdot \Delta'$ with $\mathsf{dom}(\Delta') = \{\tilde{l}\}$. $(\nu\tilde{l})$ act as binders. $\mathcal{M}, \mathcal{M}', \ldots$ range over models.

An open model maps variables and locations to closed values: a model then specifies part of the locations as "hidden". Since assertions in the present logic are intended to capture observable program behaviour, the semantics of the logic uses models quotiented by an observationally sound equivalence. Below $(\nu\tilde{l})(M, \sigma) \Downarrow$ means $(\nu\tilde{l})(M, \sigma) \longrightarrow^n (\nu\tilde{l'})(V, \sigma')$ for some n.

Definition 2. Assume $\mathcal{M}_i \stackrel{\text{def}}{=} (\nu\tilde{l_i})(\tilde{x} : \tilde{V_i}, \sigma_i)$ typable under $\Gamma;\Delta$. Then we write $\mathcal{M}_1 \approx \mathcal{M}_2$ if the following clause holds for each closing typed context $C[\,\cdot\,]$ which is typable under Δ and in which no labels from $\tilde{l}_{1,2}$ occur: $(\nu\tilde{l_1})(C[\langle\tilde{V_1}\rangle], \sigma_1) \Downarrow$ iff $(\nu\tilde{l_2})(C[\langle\tilde{V_2}\rangle], \sigma_2) \Downarrow$ where $\langle\tilde{V}\rangle$ is the n-fold pairings of a vector of values.

3.2 Semantics of Reachability and Hiding.

Let σ be a store and $S \subset \mathsf{dom}(\sigma)$. Then the *label closure of S in σ*, written $\mathsf{lc}(S, \sigma)$, is the minimum set S' of locations such that: (1) $S \subset S'$ and (2) If $l \in S'$ then $\mathsf{fl}(\sigma(l)) \subset S'$.

Lemma 3. *For all σ, we have:*

1. *$S \subset \mathsf{lc}(S, \sigma)$; $S_1 \subset S_2$ implies $\mathsf{lc}(S_1, \sigma) \subset \mathsf{lc}(S_2, \sigma)$; and $\mathsf{lc}(S, \sigma) = \mathsf{lc}(\mathsf{lc}(S, \sigma), \sigma)$*
2. *$\mathsf{lc}(S_1, \sigma) \cup \mathsf{lc}(S_2, \sigma) = \mathsf{lc}(S_1 \cup S_2, \sigma)$*
3. *$S_1 \subset \mathsf{lc}(S_2, \sigma)$ and $S_2 \subset \mathsf{lc}(S_3, \sigma)$, then $S_1 \subset \mathsf{lc}(S_3, \sigma)$*
4. *there exists $\sigma' \subset \sigma$ such that $\mathsf{lc}(S, \sigma) = \mathsf{fl}(\sigma') = \mathsf{dom}(\sigma')$.*

(1,2) are direct from the definition, and (3,4) follow from (1,2). Now set $\Gamma;\Delta \vdash e : \alpha$, $\Gamma;\Delta \vdash \mathcal{M}$ and $\mathcal{M} = (\xi, \sigma)$. The *interpretation of e under* $\mathcal{M}$, denoted $[\![e]\!]_{\xi,\sigma}$ is given by:

$$[\![x]\!]_{\xi,\sigma} = \xi(x) \quad [\![!e]\!]_{\xi,\sigma} = \sigma([\![e]\!]_{\xi,\sigma}) \quad [\![\mathsf{c}]\!]_{\xi,\sigma} = \mathsf{c} \quad [\![\mathsf{op}(\tilde{e})]\!]_{\xi,\sigma} = \mathsf{op}([\![\tilde{e}]\!]_{\xi,\sigma})$$
$$[\![\langle e, e'\rangle]\!]_{\xi,\sigma} = \langle [\![e]\!]_{\xi,\sigma}, [\![e']\!]_{\xi,\sigma}\rangle \quad [\![\pi_i(e)]\!]_{\xi,\sigma} = \pi_i([\![e]\!]_{\xi,\sigma}) \quad [\![\mathtt{inj}_i(e)]\!]_{\xi,\sigma} = \mathsf{inj}_i([\![e]\!]_{\xi,\sigma})$$

We now set the satisfaction of the reachability which says that the set of hereditarily reachable names from e_1 includes e_2 up to $\approx$.

$$\mathcal{M} \models e_1 \hookrightarrow e_2 \quad \text{if } [\![e_2]\!]_{\xi,\sigma} \in \mathsf{lc}(\mathsf{fl}([\![e_1]\!]_{\xi,\sigma}), \sigma) \text{ for each } (\nu\tilde{l})(\xi, \sigma) \approx \mathcal{M}$$

For the programs in § 2.3 (3), we can check $f_w \hookrightarrow x$, $f_r \hookrightarrow x$ and $f_c \hookrightarrow z$ hold under $f_w : \lambda().(x := 1)$, $f_r : \lambda().!x$, $f_c : \mathtt{let}\ x = \mathtt{ref}(z)\ \mathtt{in}\ \lambda().x$ (regardless of the store part).

The following characterisation of # is often useful for justifying axioms. Below $\sigma = \sigma_1 \uplus \sigma_2$ indicates that σ is the union of σ_1 and σ_2, assuming $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset$.

Proposition 4 (partition). *$\mathcal{M} \models x\#u$ iff for some $\tilde{l}$, V, l and $\sigma_{1,2}$, we have $\mathcal{M} \approx (\nu\tilde{l})(\xi \cdot u : V \cdot x : l,\ \sigma_1 \uplus \sigma_2)$ such that $\mathsf{lc}(\mathsf{fl}(V), \sigma_1 \uplus \sigma_2) = \mathsf{fl}(\sigma_1) = \mathsf{dom}(\sigma_1)$ and $l \in \mathsf{dom}(\sigma_2)$.*

The characterisation says that if x is unreachable from u then, up to $\approx$, the store can be partitioned into one covering all reachable names from u and another containing x.

The existential hiding-quantifier has the following semantics.

$$\mathcal{M} \models \nu x.C \quad \text{if} \quad \exists \mathcal{M}'.((\nu l)\mathcal{M}' \approx \mathcal{M} \ \wedge\ \mathcal{M}'[x : l] \models C)$$

where l is fresh, i.e. $l \notin \mathsf{fl}(\mathcal{M})$ where $\mathsf{fl}(\mathcal{M})$ denotes free labels in $\mathcal{M}$. The notation $(\nu l)\mathcal{M}'$ denotes addition of the hiding of l to $\mathcal{M}'$, as well as indicating that l occurs free in $\mathcal{M}'$. $\mathcal{M}[x : l]$ adds $x : l$ to the environment part of $\mathcal{M}$. This says that x denotes a hidden reference, say l, and the result of taking it off from $\mathcal{M}$ satisfies C. $\overline{\nu} x.C$ is defined dually. As an example of satisfaction, let: $\mathcal{M} \stackrel{\text{def}}{=} (\nu l)(\{u : \lambda().(l :=!l+1;!l)\},\ \{l \mapsto 0\})$ then we have $\mathcal{M} \models \nu x.C$ with $C = (!x = 0 \ \wedge\ \forall i^{\mathsf{Nat}}.\{!x = i\}u \bullet () = z\{z =!x \ \wedge\ !x = i+1\})$ using the above definition. To see this, let $\mathcal{M}' \stackrel{\text{def}}{=} (\{u : \lambda().(l :=!l+1;!l)\},\ \{l \mapsto 0\})$ then we surely have $(\nu l)\mathcal{M}' = \mathcal{M}$ and $\mathcal{M}'[x : l] \models C$. Here $\mathcal{M}$ represents a situation where l is hidden and u denotes a function which increments and returns the content of l; whereas $\mathcal{M}'$ is the result of taking off this hiding, exposing the originally local state.

3.3 Soundness and Completeness

The definition of satisfiability $\mathcal{M} \models C$ for the remaining formulae is given in [1]. where logical connectives are interpreted classically and type variables are treated syntactically [12]. Let $\mathcal{M}$ be a model $(\nu\tilde{l})(\xi, \sigma)$ of type $\Gamma; \Delta$, and $\Gamma; \Delta \vdash M : \alpha$ with u fresh.

Then *validity* $\models \{C\}M :_u \{C'\}$ is given by $\forall \mathcal{M}.(\mathcal{M} \models C \supset (\mathcal{M}[u{:}M] \Downarrow \mathcal{M}' \wedge \mathcal{M}' \models C')$ with $\mathcal{M}$ including all variables in M, C and C' except u, where we write $\mathcal{M}[u{:}N] \Downarrow \mathcal{M}'$ when $(N\xi, \sigma) \Downarrow (\nu\tilde{l}')(V, \sigma')$ and $\mathcal{M}' = (\nu\tilde{l}\tilde{l}')(\xi \cdot u{:}V,\ \sigma')$.

Theorem 5 (soundness). *$\vdash \{C\}M :_u \{C'\}$ implies $\models \{C\}M :_u \{C'\}$.*

We next discuss the completeness properties of the logic. A strong completeness property is *descriptive completeness* studied in [11], which is provability of a characteristic assertion for each program (i.e. assertions characterising programs' behaviour). In [11], we have shown that, for our base logic, this property directly leads to two other completeness properties, *relative completeness* (which says that provability and validity of judgements coincide) and *observational completeness* (which says that validity precisely characterises the standard contextual equivalence).

The proof of descriptive completeness closely follows [11]. Relative and observational completeness are its direct consequences. Descriptive completeness is established for a refinement of the present logic where evaluation formulae and content quantification are decomposed into fine-grained operators [1]. For the space sake, we only state observational completeness, which we regard as a basic semantic property of the logic.

Write $\cong$ for the standard contextual congruence for programs [22]; further write $M_1 \cong_{\mathcal{L}} M_2$ to mean ($\models \{C\}M_1 :_u \{C'\}$ iff $\models \{C\}M_2 :_u \{C'\}$). We have:

Theorem 6 (observational completeness). *For each $\Gamma;\Delta \vdash M_i : \alpha$ $(i = 1,2)$, we have $M_1 \cong_{\mathcal{L}} M_2$ iff $M_1 \cong M_2$.*

4 Axioms for Reachability, Hiding and Local Invariant

4.1 Basic Axioms for Reachability and Hiding

We start from the axioms for reachability. Note that our types include recursive types.

Proposition 7 (axioms for reachability). *The following assertions are valid.*

1. (1) $x \hookrightarrow x$; (2) $x \hookrightarrow y \wedge y \hookrightarrow z \supset x \hookrightarrow z$;
2. (1) $y \# x^{\alpha}$ *with* $\alpha \in \{\mathsf{Unit}, \mathsf{Nat}, \mathsf{Bool}\}$; (2) $x\#y \Rightarrow x \neq y$; (3) $x\#w \wedge w \hookrightarrow u \supset x\#u$.
3. (1) $\langle x_1, x_2\rangle \hookrightarrow y \equiv x_1 \hookrightarrow y \vee x_2 \hookrightarrow y$; (2) $\mathtt{inj}_i(x) \hookrightarrow y \equiv x \hookrightarrow y$; (3) $x \hookrightarrow y^{\mathsf{Ref}(\alpha)} \supset x \hookrightarrow !y$; (4) $x^{\mathsf{Ref}(\alpha)} \hookrightarrow y \wedge x \neq y \supset !x \hookrightarrow y$; (5) $[!x]x\#y \equiv x\#y$.

The proofs use Lemma 3. 3-(5) says that altering the content of x does not affect reachability *to* x. Note $[!x]y\#x \equiv y\#x$ is not valid at all. 3-(5) was already used for deriving $[\mathit{Inv}\text{-}\#]$ in §2.4 (we cannot substitute $!x$ for y in $[!x]x\#y$ to avoid name capture).

Let us say α is *finite* if it does not contains an arrow type or a type variable. We say $e \hookrightarrow e'$ is *finite* if e has a finite type. Then by Proposition 7 2-(1) and 3:

Theorem 8 (elimination). *Suppose all reachability predicates in C are finite. Then there exists C' such that $C \equiv C'$ and no reachability predicate occurs in C'.*

A straightforward coinductive extension of the above axioms gives a complete axiomatisation with recursive types [1], but not function types. For analysing reachability, we define the following "one-step" reachability predicate. Below e_2 is of a reference type.

$$\mathcal{M} \models e_1 \rhd e_2 \quad \text{if } [\![e_2]\!]_{\xi,\sigma} \in \mathsf{fl}([\![e_1]\!]_{\xi,\sigma}) \text{ for each } (\nu\tilde{l})(\xi,\sigma) \approx \mathcal{M} \tag{7}$$

We can show $(\nu\tilde{l})(\xi,\sigma) \models x \rhd l'$ is equivalent to $l' \in \bigcap\{\mathsf{fl}(V) \mid V \cong \xi(x)\}$, (the latter says that l' is in the support $[6,24,30]$ of f). We set: $x \rhd^1 y \equiv x \rhd y$; $x \rhd^{n+1} y \equiv \exists z.(x \rhd z \wedge !z \rhd^n y)$ $(n \geq 1)$. We also set $x \rhd^0 y \equiv x = y$. By definition:

Proposition 9. $x \hookrightarrow y \equiv \exists n.(x \rhd^n y) \equiv (x = y \vee x \rhd y \vee \exists z.(x \rhd z \wedge z \neq y \wedge z \hookrightarrow y))$.

Proposition 9, combined with Theorem 8, suggests that if we can clarify one-step reachability at function types then we will be able to clarify the reachability relation as a whole. Unfortunately this relation is inherently intractable.

Proposition 10. (1) $\mathcal{M} \models f^{\alpha\Rightarrow\beta} \rhd x$ *is undecidable.* (2) $\mathcal{M} \models f^{\alpha\Rightarrow\beta} \hookrightarrow x$ *is undecidable.*

The same result holds for call-by-value $\beta\eta$-equality. The result also implies that the validity of $\forall xf.(A \supset f \rhd x)$ is undecidable, since we can represent any PCFv-term as a formula using the method [11]. However Proposition 10 does not imply that we cannot obtain useful axioms for (un)reachability for function types. We discuss a collection of basic axioms in the following.

Proposition 11. *For an arbitrary C, the following is valid with i, X fresh:* $\{C \wedge x\#fy\tilde{w}\}f \bullet y = z\{C'\}@\tilde{w} \supset \forall \mathrm{X}, i^{\mathrm{X}}.\{C \wedge x\#fiy\tilde{w}\}f \bullet y = z\{C' \wedge x\#fiyz\tilde{w}\}@\tilde{w}$.

The above axiom says that if x is unreachable from f, y and $\tilde{w}$, then the application of f to y with the write set $\tilde{w}$ never exports x. Next we list basic axioms for hiding.

Proposition 12. (1) $C \supset \nu x.C$ *if* $x \notin \mathsf{fv}(C)$; $\nu x.C \equiv C$ *if* $x \notin \mathsf{fv}(C)$ *and no evaluation formula occurs in* C; (2) $\nu x.(C \wedge u = x) \equiv C \wedge \nu x.u = x$ *where* $x \notin \mathsf{fv}(C)$; *and* (3) $\nu x.(C_1 \vee C_2) \equiv (\nu x.C_1) \vee (\nu x.C_2)$; $\nu x.(C_1 \wedge C_2) \supset (\nu x.C_1) \wedge (\nu x.C_2)$

For (1), it is notable that we do *not* generally have $C \supset \nu x.C$. Neither $\nu x.C \supset C$ with $x \notin \mathsf{fv}(C)$ holds generally, see [1]. This shows that integrating these quantifiers into the standard $\forall$ and $\exists$-quantifiers let the latter lose their standard axioms, motivating the introduction of ν-operator. (2,3) list some of useful axioms for moving the scope of x.

4.2 Local Invariant

We now introduce an axiom for local invariants. Let us first consider a function which writes to a local reference of a base type. Even programs of this kind pose fundamental difficulties in reasoning, as shown in [18]. Take the following program:

$$\texttt{compHide} \stackrel{\text{def}}{=} \texttt{let}\ x = \texttt{ref}(7)\ \texttt{in}\ \lambda y.(y > !x) \tag{8}$$

The program behaves as a pure function $\lambda y.(y > 7)$. Clearly, the obvious local invariant $!x = 7$ is preserved. We demand this assertion to hold under arbitrary invocations of `compHide`: thus (naming the function u) we arrive at the following invariant:

$$C_0 \ = \ !x = 7 \ \wedge \ \forall y.\{!x = 7\}u \bullet y = z\{!x = 7\}@\emptyset \tag{9}$$

Assertion (9) says: (1) the invariant $!x = 7$ holds now; and that (2) once the invariant holds, it continues to hold for ever (note x can never be exported due to the type of y and z, so that only u will touch x). `compHide` is easily given the following judgement:

$$\{\mathsf{T}\}\texttt{compHide} :_u \{\nu x.(x\#i^{\mathrm{X}} \ \wedge \ C_0 \ \wedge \ C_1)\} \qquad (i \text{ fresh}) \tag{10}$$

with $C_1 = \forall y.\{!x = 7\}u \bullet y = z\{z = (y > 7)\}@\emptyset$. Thus, noting C_0 is only about the content of x, we conclude C_0 continues to hold automatically. Hence we cancel C_0 together with x:

$$\{\mathsf{T}\}\texttt{compHide} :_u \{\forall y.\{\mathsf{T}\}u \bullet y = z\{z = (y > 7)\}\} \tag{11}$$

which describes a purely functional behaviour. Below we stipulate the underlying reasoning principle as an axiom. Let y, z be fresh. For simplicity of presentation, we assume y has a base type.[3]

$$\mathsf{Inv}(u, C_0, \tilde{x}) \ = \ C_0 \ \wedge \ (\forall y i.\{C_0\}u \bullet y = z\{\mathsf{T}\} \supset \forall y i.\{C_0\}u \bullet y = z\{C_0 \ \wedge \ \tilde{x}\#z\}) \tag{12}$$

where we assume $C_0 \supset \tilde{x}\#i$. $\mathsf{Inv}(u, C_0, x)$ says that, first, currently C_0 holds; and that second if C_0 holds, then applying u to y results in, if it ever converges, C_0 again and the returned z is disjoint from $\tilde{x}$. Below we say C is *stateless* if $\mathcal{M} \models C$ and $\mathcal{M}[u : N] \Downarrow \mathcal{M}'$ imply $\mathcal{M}' \models C$ (its syntactical characterisation can be found in Appendix A).

[3] That is sufficient for all examples in this paper: The refinement allows arbitrary types [1].

Proposition 13 (axiom for information hiding). *Assume $C_0 \supset \tilde{x}\#i$ and $[!\tilde{x}]C_0$ is stateless. Suppose i,m are fresh, $\{\tilde{x},\tilde{g}\} \cap (\mathsf{fv}(C,C') \cup \{\tilde{w}\}) = \emptyset$ and y has a base type. Let $E_1 = \mathsf{Inv}(u,C_0,\tilde{x}) \wedge \forall yi.\{C_0 \wedge [!\tilde{x}]C\}u \bullet y = z\{C'\}@\tilde{w}\tilde{x}$ and $E_2 = \forall y.\{C\}u \bullet y = z\{C'\}@\tilde{w}$. Then the following assertion is valid.*

$$(\mathsf{AIH}) \quad \{E\}m \bullet () = u\{\nu\tilde{x}.\exists\tilde{g}.(E_1 \wedge E')\} \supset \{E\}m \bullet () = u\{E_2 \wedge E'\}$$

(AIH) is used with the consequence rule (Appendix A) to simplify from E_1 to E_2. Its validity is proved using Proposition 4. The axiom says: *if a function u with a fresh reference x_i is generated, and if it has a local invariant C_0 on the content of x_i, then we can cancel C_0 together with x_i.* The statelessness of $[!\tilde{x}]C_0$ ensures that satisfiability of C_0 is not affected by state change except at $\tilde{x}$; and $[!\tilde{x}]C$ says that whether C holds does not depend on $\tilde{x}$. Finally $\exists\tilde{g}$ in E_1 allows the invariant to contain free variables, extending applicability as we shall use in §5 for `safeEven`.

Coming back to `compHide`, we take C_0 to be $!x = 7 \wedge x\#i$, $\tilde{w}$ empty, both C and E' to be T and C' to be $z = (y > 7)$ in (AIH), to reach the desired assertion. [1] lists the axioms of the higher-order version of Proposition 11 and apply to the examples in [18].

5 Reasoning Examples

Shared Stored Function This section presents concrete reasoning examples. We show the key ideas; the detailed derivations can be found in [1]. We first present a simple example of hiding-quantifiers and unreachability using `incShared` in (2) from § 1. We use a derived rule for the combination of "let" and new reference generation.

$$[\mathit{LetRef}]\ \frac{\{C\}\ M :_m \{C_0\} \quad \{C_0[!x/m] \wedge x\#\tilde{e}\}\ N :_u \{C'\} \quad x \notin \mathsf{fpn}(\tilde{e})}{\{C\}\ \mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N :_u \{\nu x.C'\}}$$

where $\mathsf{fpn}(e)$ denotes the set of *free plain names* of e which are reference names in e that do not occur dereferenced, defined as: $\mathsf{fpn}(x) = \{x\}$, $\mathsf{fpn}(\mathsf{c}) = \mathsf{fpn}(!e) = \emptyset$, $\mathsf{fpn}(\langle e,e'\rangle) = \mathsf{fpn}(e) \cup \mathsf{fpn}(e')$, and $\mathsf{fpn}(\pi_i(e)) = \mathsf{fpn}(\mathsf{inj}_i(e)) = \mathsf{fpn}(e)$. We also restrict C' above to a *thin formula* given in Appendix A (this does not limit the usability of this rule, at least for the reasoning examples we shall treat). The notation $x\#\tilde{e}$ appeared in § 2.3. The rule reads: *Assume* (1) *M with pre-condition C leads to post-condition C_0, with the resulting value named m; and* (2) *running N from C_0 with m as the content of x together with the assumption x is unreachable from each e_i, leads to C' with the resulting value named u. Then running* $\mathtt{let}\ x = \mathsf{Ref}(M)\ \mathtt{in}\ N$ *from C leads to C' whose x is fresh and hidden.* The side condition $x \notin \mathsf{fpn}(e_i)$ is essential for consistency (e.g. without it, we could assume $x\#x$, i.e. F). The rule directly gives a proof rule for new reference declaration [18, 23, 28], $\mathtt{new}\ x := M\ \mathtt{in}\ N$, which has the same operational behaviour as $\mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N$. Note also that the original Hoare and Wirth [9]'s rule for local variable declaration is a special case of this rule.

Let $\mathsf{inc}(x,u,n) = \forall j.\{!x = j\}u \bullet () = j+1\{!x = j+1\}@x \wedge !x = n$ and $\mathsf{inc}'(n,m) = \mathsf{inc}(!a,x,n) \wedge \mathsf{inc}(!b,y,m) \wedge x \neq y$. The left derivation is for `incShared`, while that on the right is for a program where "$b := !a$" has been replaced by "$b := \mathtt{Inc}$" in `incShared`.

We assume and use pairwise distinctness of a, b, z_1, z_2, and omit anchors of unit type.

1.$\{\mathsf{T}\}\ a := \texttt{Inc}\ \{\nu x.\mathsf{inc}(!a,x,0)\}$	$\{\mathsf{T}\}\ a := \texttt{Inc}\ \{\nu x.\mathsf{inc}(!a,x,0)\}$
2.$\{\mathsf{inc}(!a,x,0)\}\ b :=!a\ \{\mathsf{inc}(!a,x,0) \wedge \mathsf{inc}(!b,x,0)\}$	$\{\mathsf{inc}(!a,x,0)\}\ b := \texttt{Inc}\ \{\nu y.\mathsf{inc}'(0,0)\}$
3.$\{\mathsf{inc}(!a,x,0)\}\ z_1 := (!a)()\ \{\mathsf{inc}(!a,x,1) \wedge !z_1 = 1\}$	$\{\mathsf{inc}'(0,0)\}\ z_1 := ..\ \{\mathsf{inc}'(1,0) \wedge !z_1 = 1\}$
4.$\{\mathsf{inc}(!b,x,1)\}\ z_2 := (!b)()\ \{\mathsf{inc}(!b,x,2) \wedge !z_2 = 2\}$	$\{\mathsf{inc}'(1,0)\}\ z_2 := ..\ \{\mathsf{inc}'(1,1) \wedge !z_2 = 1\}$
5.$\{!z_1 = 1 \wedge !z_2 = 2\}\ (!z_1) + (!z_2) :_u \{u = 3\}$	$\{!z_1 = 1 \wedge !z_2 = 1\}\ (!z_1) + (!z_2) :_u \{u = 2\}$

Line 1 uses [*LetRef*]. In Line 2 on the left, x is automatically shared after "$b :=!a$" which leads to scope extrusion, while in the right, $x \neq y$ in $\mathsf{inc}'(0,0)$ is ensured by the ν-binding operator.

Memoised Factorial [25] (from `memFact` in § 1). Our target assertion specifies the behaviour of a pure factorial. The following inference starts from the `let`-body of `memFact`, which we name V. We set: $E_{1a} = C_0 \wedge \forall xi.\{C_0\}u \bullet x = y\{C_0 \wedge ab\#y\}@ab$, and $E_{1b} = \forall xi.\{C_0 \wedge C\}u \bullet x = y\{C'\}@ab$ where we set C_0 to be $ab\#i \wedge !b = (!a)!$, C to be T, and C' to be $y = x!$. Note that $[!ab]C_0$ is stateless by Prop. 7 5; and that, by the type of y being Nat and Prop. 7 2-(1), we have $ab\#y \equiv \mathsf{T}$. We can now reason:

1.$\{\mathsf{T}\}\ V :_u \{\forall xi.\{C_0\}u \bullet x = y\{C_0 \wedge C'\}\}@\emptyset$	
2.$\{ab\#i\}\ V :_u \{E_{1a} \wedge E_{1b}\}$	(1, Conseq, Inv-#)
3.$\{\mathsf{T}\}\ \texttt{memFact} :_u \{\nu ab.(E_{1a} \wedge E_{1b})\}$	(2, LetRef)
4.$\{\mathsf{T}\}\ \texttt{memFact} :_u \{\forall x.\{\mathsf{T}\}u \bullet x = y\{y = x!\}@\emptyset.\}$	(3,(AIH),Conseq)

Line 2 uses the axiom $\{C\}f \bullet x = y\{C_1 \wedge C_2\}@\tilde{w} \supset \wedge_{i=1,2}\{C\}f \bullet x = y\{C_i\}@\tilde{w}$.

5.2 Mutually Recursive Stored Functions (from (3) in § 1). We first verify [1]:

$$\{\mathsf{T}\}\texttt{mutualParity} :_u \{\exists gh.IsOddEven(gh, !x!y, xy, n)\} \tag{13}$$

where, with $Even(n) \equiv \exists x.(n = 2 \times x)$ and $Odd(n) \equiv Even(n+1)$:

$$IsOddEven(gh, wu, xy, n) = (IsOdd(w, gh, n, xy) \wedge IsEven(u, gh, n, xy) \wedge !x = g \wedge !y = h)$$
$$IsOdd(u, gh, n, xy) = \forall n.\{!x = g \wedge !y = h\}u \bullet n = z\{z = Odd(n) \wedge !x = g \wedge !y = h\}@xy$$

where $IsOdd(u, gh, n, xy)$ says that *x stores a procedure which checks if its argument is odd if y stores a procedure which does the dual, and x does store the behaviour.* $IsEven(u, gh, n, xy)$ is defined dually. Our aim is to derive the following judgement for `safeOdd` starting from (13) (the case for `safeEven` is symmetric).

$$\{\mathsf{T}\}\texttt{safeOdd} :_u \{\forall n.\{\mathsf{T}\}u \bullet n = z\{z = Odd(n)\}@\emptyset\} \tag{14}$$

We first identify the local invariant: $C_0 = !x = g \wedge !y = h \wedge IsEven(h, gh, n, xy) \wedge xy\#ij$. Since C_0 only talks about g, h and the content of x and y, we know $[!xy]C_0$ is stateless. We now observe $IsOddEven(gh, !x!y, xy, n)$ is the conjunction of:

$$Odd_a = C_0 \wedge \forall n.\{C_0\}u \bullet n = z\{C_0\}@xy \quad Odd_b = \forall n.\{C_0\}u \bullet n = z\{z = Odd(n)\}@xy$$

As Line 3 in `memFact`, we can apply (AIH) to obtain (14).

Higher-Order Invariant [30, p.104]. We move to a program whose invariant behaviour depends on another function. The program instruments an original program with a simple profiling (counting the number of invocations), with α a base type.

$$\texttt{profile} \quad \stackrel{\text{def}}{=} \quad \texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}\ \lambda y^{\alpha}.(x :=\!x+1; fy)$$

Since x is never exposed, this program should behave precisely as f. We shall derive:

$$\{\forall y.\{C\}f \bullet y = z\{C'\}@\tilde{w}\}\ \texttt{profile} :_u \{\forall y.\{C\}u \bullet y = z\{C'\}@\tilde{w}\} \tag{15}$$

with $x \notin \mathsf{fv}(C,C')$ (by the bound name condition). This judgement says: *if f satisfies the specification $E = \forall y.\{C\}f \bullet y = z\{C'\}@\tilde{w}$, then* `profile` *satisfies the same specification E*. Note C and C' are arbitrary. To derive (15), we first set C_0, the invariant, to be $x\#fi\tilde{w}$. As with the previous derivations, we use two subderivations. First, by the axiom in Proposition 11, we can derive:

$$\{\mathsf{T}\}\lambda y.(x :=\!x+1; fy) :_u \{\forall yi.\{C_0\}u \bullet y = z\{C_0 \wedge x\#z\}@x\tilde{w}\} \tag{16}$$

Secondly, again by Prop. 11 we obtain $E \supset \forall y.\{C \wedge x\#f\tilde{w}\}f \bullet y = z\{x\#z\tilde{w}\}@\tilde{w}$. By this, E being stateless, Prop.7 3-(5) and [*Inv*-#], we obtain:

$$\{E\}\lambda y.(x :=\!x+1; fy) :_u \{\forall yi.\{C_0 \wedge [!x]C\}u \bullet y = z\{C' \wedge x\#z\}@x\tilde{w}\}. \tag{17}$$

By combining (16) and (17), we can use (AIH), hence done.

6 Related Work and Future Topics

For the sake of space, detailed comparisons with existing program logics and reasoning methods, in particular with Clarke's impossibility result, Caires-Cardelli's spatial logic, recent mechanisations of reachability predicates [16], as well as other logics such as LCF, Dynamic logic, higher-order logic, specification logic, Larch/ML, and Extended ML are left to the long version [1] and our past papers [2, 10, 12, 13]. Below we focus on work that treats locality and recent work on Hoare logics.

Reasoning Principles for Functions with Local State. There is a long tradition of studying equivalences over higher-order programs with local state. Meyer and Sieber [18] present examples and reasoning principles based on denotational semantics. Mason, Talcott and others [14] investigate equational axioms for an untyped version of the language treated in the present paper, including local invariance. Pitts and Stark [23, 25, 30] present powerful operational reasoning principles for the same ML-fragment considered here, including reasoning principle for local invariance at higher-order types [25]. Our axioms for information hiding in § 4, which capture a basic pattern of programming with local state, are closely related with these reasoning principles. Our logic differs in that its aim is to offer a method for describing and validating properties of programs beyond program equivalence. Equational and logical approaches are complimentary: Theorem 6 offers a basis for integration. For example, we may consider deriving a property of the optimised version M' of M: if we can easily verify $\{C\}M :_u \{C'\}$ and if we know $M \cong M'$, we can conclude $\{C\}M' :_u \{C'\}$, which is useful if M is better structured than M'.

Program Logics for Aliasing and Higher-Order Functions. Reynolds et al. [28] present a program logic for aliasing where fresh data generation is represented by a special conjunction denoting spatial disjointness from the original datum. Their method can reason many programs with aliasing. The logic studied in the present paper captures freshness through generic unreachability from arbitrary data in the initial state. Apart from completeness properties discussed in §3.3, the approach enables uniform treatment of known data types, including product, sum, reference, closure, etc. Reasoning examples using the present method include those in the present paper as well as higher-order invariants from [18], objects from [15], circular lists from [16], tree-, dag- and graph-copy from [5], as presented in [1, § 6]. Birkedal et al. [4] present a typing system for a variant of Idealised Algol where types are constructed from formulae of the logic in [28]. Their typing system uses subtyping calculated via categorical semantics, the focus of their study. [3] extends the logic in [28] with higher-order frame rules, and demonstrates reasoning about priority queues. Both works consider neither exportable fresh reference generation nor higher-order procedures in full generality. In particular, it would be difficult to validate the examples in § 5.

Nanevski et al [21] studies Hoare Type Theory (HTT) which combines dependent types and Hoare triples with anchors based on monadic understanding of computation. HTT aims to provide an effective general framework which unifies standard static checking techniques and logical verifications. Local store is not treated and left as an open problem in [21]. Reus and Streicher [27] present a Hoare logic for a simple language with higher-order stored procedures, extended in [26]. Soundness is proved with denotational methods. Completeness is not considered in [26, 27]. Their assertions contain quoted programs, which is necessary to handle recursion via stored functions. Their language does not allow procedure parameters and general reference creation.

The logic studied in the present work aims to capture the behaviour of sequential higher-order programs with local state in the framework of compositional program logics *à* la Hoare, stratified on the basis of simpler program logics [2, 10, 12, 13]. The semantic precision of the logic (cf. Theorem 6), axiomatisation of local invariance, and uniform extensibility to diverse data types are among those features not found in the preceding program logics mentioned above.

Meta-Logical Study on Freshness. Freshness of names has recently been studied from the viewpoint of formalising binding relations in programming languages and computational calculi. Pitts and Gabbay [6, 24] extend first-order logic with constructs to reason about freshness of names based on permutations. The key syntactic additions are the (interdefinable) "fresh" quantifier $\mathcal{N}$ and the freshness predicate #, mediated by a swapping (finite permutation) predicate. Miller and Tiu [19] are motivated by the significance of generic (or eigen-) variables and quantifiers at the level of both formulae and sequents, and split universal quantification in two, introduce a self-dual freshness quantifier ∇ and develop the corresponding sequent calculus of Generic Judgements. While these logics are not program logics, their logical machinery may be usable in the present context. As noted in Proposition 9, reasoning about $\hookrightarrow$ or # is tantamount to reasoning about $\triangleright$, which denotes the support (i.e. semantically free locations) of a datum. A characterisation of support by the swapping operation may be interesting from the viewpoint of axiomatisation of reachability.

References

1. A full version of this paper. http://www.doc.ic.ac.uk/˜yoshida/local.
2. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *ICFP'05*, pages 280–293, 2005.
3. B. Biering, L. Birkedal, and N. Torp-Smith. Bi hyperdoctrines and higher-order separation logic. In *ESOP'05*, LNCS, pages 233–247, 2005.
4. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *LICS'05*, pages 260–269, 2005.
5. R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *Workshop SPACE*, 2004.
6. M. Gabbay and A. Pitts. A New Approach to Abstract Syntax Involving Binders. In *Proc. LICS '99*, pages 214–224, 1999.
7. C. A. R. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
8. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
9. C. A. R. Hoare and N. Wirth. Axiomatic semantics of Pascal. *Toplas*, 1(2):226–244, 1979.
10. K. Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM, 2004.
11. K. Honda, M. Berger, and N. Yoshida. Descriptive and relative completeness for logics for higher-order functions. In *ICALP'06*, volume 4052 of *LNCS*, pages 360–371, 2006.
12. K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202. ACM, 2004.
13. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS'05*, pages 270–279, 2005. Full version is at [1].
14. F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Inf. Comput.*, 119(1):55–90, 1995.
15. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. POPL*, 2006.
16. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, pages 115–126. ACM, 2006.
17. E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
18. A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL'88*, pages 191 – 203, 1988.
19. D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.
20. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1):1–77, 1992.
21. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP06*, pages 62–73. ACM Press, 2006.
22. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
23. A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *Algol-Like Languages*, volume 2, chapter 17, pages 173–193. Birkhauser, 1997.
24. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
25. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. CUP, 1998.
26. B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *Proc. CSL*, volume 4207 of *LNCS*, pages 575–590, 2006.
27. B. Reus and T. Streicher. About Hoare logics for higher-order store. In *ICALP*, volume 3580 of *LNCS*, pages 1337–1348, 2005.
28. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*.

29. J. C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
30. I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, 1994.

A Appendix: Proof Rules

The following presents compositional proof rules. We omit the rules for the sum and products. The rule for the reference can be found in the main section.

$$[Var]\ \frac{-}{\{C[x/u]\}\ x :_u \{C\}} \quad [Const]\ \frac{-}{\{C[\mathsf{c}/u]\}\ \mathsf{c} :_u \{C\}} \quad [Succ]\ \frac{\{C\}\ M :_m \{C'[m+1/u]\}}{\{C\}\ Succ(M) :_u \{C'\}}$$

$$[Abs]\ \frac{\{C \wedge A^{\tilde{xi}}\}\ M :_m \{C'\}}{\{A\}\ \lambda x.M :_u \{\forall x\tilde{i}.\{C\}\ u \bullet x = m\ \{C'\}\}} \quad [App]\ \frac{\begin{array}{l}\{C\}\ M :_m \{C_0\} \\ \{C_0\}\ N :_n \{\ C_1 \wedge \{C_1\}\ m \bullet n = u\ \{C'\}\ \}\end{array}}{\{C\}\ MN :_u \{C'\}}$$

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

$$[Deref]\ \frac{\{C\}\ M :_m \{C'[!m/u]\}}{\{C\}\ !M :_u \{C'\}} \quad [Assign]\ \frac{\{C\}\ M :_m \{C_0\} \quad \{C_0\}\ N :_n \{C'\{\!|n/!m|\!\}\}}{\{C\}\ M := N\ \{C'\}}$$

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

$$[Cons\text{-}Eval]\ \frac{\{C_0\}\ M :_m \{C_0'\} \quad \forall \tilde{i}.\{C_0\}x \bullet () = m\{C_0'\} \supset \forall \tilde{i}.\{C\}x \bullet () = m\{C'\}\ x \text{ fresh, } \tilde{i} \text{ auxiliary}}{\{C\}\ M :_m \{C'\}}$$

We assume that judgements are well-typed in the sense that, in $\{C\}\ M :_u \{C'\}$ with $\Gamma;\Delta \vdash M : \alpha$, $\Gamma,\Delta,\Theta \vdash C$ and $u : \alpha, \Gamma,\Delta,\Theta \vdash C'$ for some Θ s.t. $\mathsf{dom}(\Theta) \cap (\mathsf{dom}(\Gamma,\Delta) \cup \{u\}) = \emptyset$. In the rules, $C^{\tilde{x}}$ indicates $\mathsf{fv}(C) \cap \{\tilde{x}\} = \emptyset$. Symbols $i, j, \ldots$ range over auxiliary names. We demand the postconditions of the proof rules [*App, If*] to be *thin*, where we say C is thin iff for each $\mathcal{M}$ and for each $y \in \mathsf{fv}(\mathcal{M}) \backslash \mathsf{fv}(C)$, $\mathcal{M} \models C$ implies $\mathcal{M}/y \models C$ (a syntactic characterisation of thinness is discussed in [1]).

In [*Abs, Rec*], A, B denote *stateless* formulae given in §4.2.Syntactically C is stateless when: (1) each dereference $!y$ only occurs either in pre/post conditions of evaluation formulae or under $[!y]$; (2) (un)reachability predicates occur in pre/post conditions of evaluation formulae; and (3) evaluation formulae and content quantifications never occur negatively (using the standard notion of negative/positive occurrences).

[*Assign*] uses *logical substitution* $C\{\!|e_2/!e_1|\!\}$ which is built with content quantification to represent substitution of content of a possibly aliased reference [2]. This is defined as: $C\{\!|e_2/!e_1|\!\} \stackrel{\text{def}}{=} \forall m.(m = e_2 \supset [!e_1](!e_1 = m \supset C))$. with m fresh. Intuitively $C\{\!|e_2/!e_1|\!\}$ describes the situation where a model satisfying C is updated at a memory cell referred to by e_1 (of a reference type) with a value e_2 (of its content type), with $e_{1,2}$ interpreted in the current model. [*Cons-Eval*] is a strengthened version of the standard consequence rule [*Conseq*].

The proof rules for the located judgement is given just as [2], adding the following rule for the reference, with i, X fresh.

$$[Ref]\ \frac{\{C\}\ M :_m \{C'\}@\tilde{e} \quad x \notin \mathsf{fpn}(\tilde{e}) \cup \mathsf{fv}(\tilde{e})}{\{C\}\ \mathtt{ref}(M) :_u \{\forall x.(u\#i^{\mathrm{X}} \wedge u = x \wedge C')\}@\tilde{e}}$$