Logical Reasoning for Higher-Order Functions with Local State

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

[1]Queen Mary, University of London, UK
[2] Imperial College London, UK

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. We explore the logic's descriptive and reasoning power with non-trivial programming examples combining higher-order procedures and dynamically generated local state. Axioms for reachability and local invariant play a central role for reasoning about the examples.

1 Introduction

Reference Generation in Higher-Order Programming. This paper proposes an extension of Hoare Logic [14] for call-by-value higher-order functions with ML-like new reference generation [3, 4], and demonstrates its use through non-trivial reasoning examples. The new reference generation, the `ref`-construct in ML, is a highly expressive programming primitive. The first and central significance of this construct is that it induces a local state by generating a fresh reference inaccessible from the outside. Consider the following program:

$$\texttt{Inc} \stackrel{\text{def}}{=} \texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}\ \lambda().(x :=\!x+1;\ !x) \tag{1.1}$$

We use the standard notation [40]: in particular, "$\texttt{ref}(M)$" returns a fresh reference whose content is the value to which M evaluates. "$!x$" is the dereferencing of an imperative variable x. ";" is a sequential composition. In (1.1), a reference with content 0 is newly created and is never exported to the outside, so that it is hidden from the outside (i.e. it can never be directly read/written from the outside). When the anonymous function in `Inc` is invoked, it increments the content of a local variable x, and returns the new content. From an outside observer, 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.

Second, local references thus generated may be exported outside of its original scope and shared, contributing to expressibility of significant imperative idioms. The

next example shows how stored procedures interact with new reference generation and its sharing. We consider the following program from [44, § 6]:

```
1  a := Inc;          (* !x = 0 *)
2  b := !a;           (* !x = 0 *)
3  z1 := (!a)();      (* !x = 1 *)
4  z2 := (!b)();      (* !x = 2 *)
5  (!z1)+(!z2)
```

This program, which we hereafter call `IncShared`, first assigns, in Line 1 ($l.1$), the program `Inc` to a; then, in $l.2$, assigns the content of a to b; and invokes, in $l.3$, the content of a; then does the same for that of b in $l.4$; and finally in $l.5$ adds up the two numbers returned from these two invocations. By tracing the reduction of this program, we can check that if the initial value of x is 0 (at $l.1$ and $l.2$), then the return value of this program is 3. To specify and understand the behaviour of `IncShared`, it is essential to capture the sharing of x between two procedures assigned to a and b, whose scope is originally (at $l.1$) restricted to $!a$ but gets (at $l.2$) extruded to and shared by $!b$. Controlling sharing by combining scope extrusion and local reference also allows us to write concise algorithms that dynamically manipulate mutable data structures such as linked lists and graphs which may possibly store higher-order values [40]. Difficulties in formal reasoning about shared (possibly higher-order) local store, both axiomatic and otherwise, have been well-known since [15, 31, 33].

Thirdly, and related to the previous two points, local references can be used for efficient implementation of highly regular observable behaviour, for example purely functional behaviour, through information hiding. The following program is a simplification of the standard memoised function, taken from [44, § 1].

$$\begin{aligned}\texttt{memFact} \stackrel{\text{def}}{=}\ & \texttt{let}\ a = \texttt{ref}(0),\ \ b = \texttt{ref}(1)\ \texttt{in} \\ & \lambda x.\texttt{if}\ x = !a\ \texttt{then}\ !b\ \texttt{else}\ (a := x; b := \texttt{fact}(x); !b)\end{aligned}$$

Above `fact` is the standard factorial function. The program shows a simple case of memoisation when `memFact` is called with a stored argument in a, it immediately returns the stored return value $!b$. If the argument differs from the stored argument, it calculates the factorial fx, and stores the new pair. The reason why `memFact` behaves indistinguishably from the pure factorial is tantamount to the following *local invariant property* [44].

> *Throughout all possible invocations of this procedure, 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 the preceding studies on operational reasoning of program equivalence with local state [21, 42, 44, 50]. Can we distill this principle axiomatically and use it for effectively validating properties of higher-order programs with local state, such as `memFact`?

As a further example of local invariant, but this time involving a higher-order store, the following is yet another implementation of the factorial function using local state.

We start from the following program which realises a recursion by circular references [24]:

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

This program calculates the factorial of n. But since x is free in `circFact`, if a program reads from x and stores it in another variable, say y, assigns a diverging function to x, and feeds the content of y with 3, then the program diverges rather than returning 6. With local reference, we can hide x to avoid unexpected interference.

$$\texttt{safeFact} \stackrel{\text{def}}{=} \texttt{let } x = \texttt{ref}(\lambda y.y) \texttt{ in } (\texttt{circFact}; !x)$$

(above $\lambda y.y$ can be any initialising value). The program evaluates to a function which also calculates the factorial: but x is now invisible and inaccessible from the outside, so that `safeFact` behaves as the pure factorial function. In this case, the invariant says that x always stores the factorial — but notice the reason this stored procedure can calculate the factorial is precisely because x stores this very behaviour. We shall show a general reasoning principle for local invariants which can verify properties of these two and many other examples [21, 27, 28, 31, 42, 44], including mutually recursive multiple stored functions.

Program Logic for Imperative Higher-Order Functions. Starting from their origins in the λ-calculus, typed higher-order functional programming languages such as Haskell and ML, has been extensively studied, making them an ideal target for formal validation of programs' properties on a rigorous semantic basis. Further, given expressive power of imperative higher-order functions (attested by encodability of objects [10, 40, 41] and of low-level idioms [1]), a study of logics for these languages may have wide repercussions on logics of programming languages in general.

These languages combine higher-order functions and imperative features including new reference generation. Extending Hoare logic to these languages leads to technical difficulties due to their three fundamental features:

- Higher-order functions, including stored ones.
- General forms of aliasing induced by nested reference types.
- Dynamically generated local references and scope exclusion.

In our preceding studies, we presented Hoare logics for the core parts of ML which capture the first two features [6, 17, 19, 20]. On the basis of these works, the present work introduces an extension of Hoare logic for ML-like local reference generation. As noted above, this construct radically enriches programs' behaviour, and has defied its clean axiomatic treatment so far. A central challenge is to identify a simple but expressive logical primitive, equipped with proof rules (for Hoare triples) and axioms (for assertions), enabling tractable assertions and verification.

The program logic proposed in the present paper introduces a predicate representing (un)reachability of a reference from an arbitrary datum in order to capture new reference generation. Since we are working with higher-order programs, a datum and a reference may as well be, or store, a higher-order function. We shall show that this predicate is fully axiomatisable using (in)equality when it only involves first-order data types (the

result is closely related with known axiomatisations of reachability [37]). However we shall also show that the predicate becomes undecidable in itself when higher-order types are involved, indicating its inherent intractability.

A good news is, however, this predicate enables us to obtain a simple compositional proof rule for new reference generation, preserving all the compositional proof rules for the remaining constructs from our foregoing program logics. We also introduce a pair of mutually dual hiding quantifiers (i.e. quantifiers ranging over variables denoting hidden references). At the level of assertions, we can find a set of useful axioms for (un)reachability and the hiding quantifiers, which are effectively combined with logical primitives and associated axioms for higher-order functions and aliasing studied in our preceding works [6, 20]. These axioms for reachability and hiding quantifiers are closely related with reasoning principles studied in existing semantic studies on local state, such as the principle of local invariant. The status of these new logical primitives is clarified through soundness and three completeness results, including relative completeness.

Some of the non-trivial reasoning examples are presented in later sections, which include those involving local invariants and those involving higher-order mutable data structures with circular pointers.

Outline. Section 2 presents the programming language, the assertion language and proof rules. Section 3 gives the semantics of the logic and its properties, and proves soundness and the three completeness results. Section 4 explores axioms of the assertion language. Sections 5 and 6 discuss the use of the logic through non-trivial reasoning examples. Section 7 gives comparisons with related works and concludes with further topics. Appendix lists auxiliary definitions and omitted derivations.

2 Assertions for Local State

2.1 A Programming Language

As our target programming language, we use call-by-value PCF with unit, sums and products, 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 the following grammar.

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

We use the standard notations [40]. We use constants $\mathtt{c}$ (unit $()$, booleans $\mathtt{t}$, $\mathtt{f}$, numbers $\mathtt{n}$ and locations $l, l', \ldots$) and fist-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 $\texttt{let}\ x = M\ \texttt{in}\ N$. Typing is standard: we take the equi-isomorphic approach [40] for recursive types. Nat, Bool and Unit *atomic types*. We leave illustration of each construct to standard textbooks [40], except for the focus of the present study, the reference generation $\texttt{ref}(M)$, 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:

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

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 binder (the use of the ν-binding simplifies the correspondence with models discussed in §3). Its general form is $(\nu\, \tilde{l})(M,\sigma)$ where $\tilde{l}$ is a vector of distinct locations occurring in σ (the order is irrelevant). We write (M,σ) for $(\nu\, \varepsilon)(M,\sigma)$. The one-step reduction $\longrightarrow$ over configurations is defined using the standard rules [40] except for the above rule and for closing it under ν-bindings. The full rules are listed in Appendix A.1.

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 [40], which is left to Appendix A.2. The sequent has the form $\Gamma;\Delta \vdash M : \alpha$ which reads: M has type α under $\Gamma;\Delta$. We omit Γ or Δ if it is empty. 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 $\Delta \vdash \sigma$. The 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 we shall use is that of standard first-order logic with equality [30, § 2.8], extended with assertions for evaluation [19, 20] (for imperative higher-order functions) and quantifications over store content [6] (for aliasing). On this basis we add a binary predicate which asserts reachability of a reference name from a datum and its dual; and the pair of mutually dual quantifiers over hidden references. The grammar follows, letting $\star \in \{\wedge,\vee,\supset\}$, $\mathcal{Q} \in \{\exists,\forall,\nu,\overline{\nu}\}$ and $\mathcal{Q}' \in \{\exists,\forall\}$.

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

The first set of expressions $(e,e',\ldots)$ are *terms*; the second set *formulae* $(A,B,C,C'\ldots)$. Terms include variables, constants c (unit $()$, numbers $\mathtt{n}$, booleans $\mathtt{t}$, $\mathtt{f}$ and locations $l,l',\ldots$), pairing, projection, injection and standard first-order operations. $!e$ denotes the dereference of a reference e.

Formulae include the standard logical connectives and quantification [30]. Quantifiers, $\exists x.C$ and $\forall x.C$, are standard. The hiding quantifiers, $\nu x.C$ (read: "for some hidden

reference x, C holds") and $\overline{\nu}x.C$ (read: "for each hidden reference x, C holds"), which are mutually dual, are new and quantify only variables of reference types (x's type is $\alpha = \mathsf{Ref}(\beta)$). Semantically, these quantifiers range over hidden references, such as what `Inc` in Introduction generates. We also include, following [6, 19], quantifications over type variables ($\mathrm{X}, \mathrm{Y}, \ldots$). Type variables can be used only in the typing of auxiliary variables. We also use truth T (definable as $1 = 1$) and falsity F (which is $\neg\mathsf{T}$). $x \neq y$ stands for $\neg(x = y)$.

The remaining formulae are those specifically introduced for describing program behaviour. Their use will be illustrated using concrete examples soon: here we informally outline their ideas. $\{C\}\ e \bullet e' = x\ \{C'\}$ is called *evaluation formula*, introduced in [20], 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'.*

$[!e]C$ and $\langle !e\rangle C$ are *universal/existential content quantifications*, introduced in [6] for treating general aliasing. $[!e]C$ (with e of a reference type) says: *Whatever value we may store in a reference denoted by e, the assertion C is valid.* $\langle !e\rangle C$ is interpreted dually.

Finally, $e_1 \hookrightarrow e_2$ (with e_2 of a reference type), called *reachability predicate*, plays an essential role in the present logic. It says that: *We can reach the reference named by e_2 from a datum denoted by e_1.* As an example, if x denotes a starting point of a linked list, $x \hookrightarrow y$ says a reference y occurs in one of the cells reachable from x. $y\#x$ [13, 47] is the negation of $x \hookrightarrow y$, which says: *One can never reach a reference y starting from a datum denoted by x.*

Convention. Logical connectives are used with standard precedence/association, using parentheses as necessary to resolve ambiguities. $\mathsf{fv}(C)$ (resp. $\mathsf{fl}(C)$) denotes the set of free variables (resp. locations) in C. Note that x in $[!x]C$ and $\langle !x\rangle C$ occurs free, while in $\{C\}\ e \bullet e' = x\ \{C'\}$ it occurs bound with scope C'. With $\tilde{e} = e_1..e_n$, we often write $!\tilde{e}$ for $!e_1..!e_n$; and $[!\tilde{e}]C$ for $[!e_1]..[!e_n]C$. $C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$. We write $\tilde{x}\#y$ for $\wedge_i x_i\#y$; similarly for $x\#\tilde{y}$. We write $\{C\}e_1 \bullet e_2\{C'\}$ for $\{C\}e_1 \bullet e_2 = z\{z = () \wedge C'\}$ with $z \notin \mathsf{fv}(C')$. 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 in concrete assertions.

2.3 Assertions for Local State

We explain assertions for local state with examples.

1. 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, thus 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 $x \hookrightarrow x \equiv x \hookrightarrow !x \equiv \mathsf{T}$ and $x\#x \equiv \mathsf{F}$. 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. 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. Then $\forall i.\{!x = i\}u \bullet () = z\{!x = z \wedge !x = i+1\}$ asserts that the function u, upon receiving unit $()$, increments the content of x and returns it. For example for $\lambda().(x := !x+1; !x)$ named u satisfies it. For a stronger specification, we may refine this assertion by also specifying which references a program may write to. The following *located assertion* [6] is used for this purpose.

$$\mathsf{inc}(u,x) \;=\; \forall i.\{!x = i\}u \bullet () = z\{!x = z \wedge !x = i+1\} \mathbin{@} x$$

Above "$@x$", called *write set*, indicates that the evaluation alters at most x, leaving content of other references unchanged. The assertion says: "for any r of any reference type distinct from x, its content h stays invariant after the run," that is at most x is modified during the run. The exact semantic account of located assertions is given in Appendix C.

4. We consider reachability in (higher-order) functions. Assume $\lambda().(x := 1)$ is named f_w and $\lambda().!x$ is named 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 $f_c \hookrightarrow z$ (for example, consider $!(f_c()) := 1$). However x is *not* reachable from $\lambda().((\lambda y.())(\lambda().x))$ since semantically it never touches/uses x.
5. Assuming u denotes the result of evaluating `Inc` in the Introduction, we can assert, using the existential hiding quantifier:

$$\nu x.(!x = 0 \;\wedge\; \forall i^{\mathsf{Nat}}.\{!x = i\}u \bullet () = z\{z = !x \wedge !x = i+1\}@x) \tag{2.1}$$

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 stored in x at the time of invocation.

6. $\lambda n^{\mathsf{Nat}}.\mathtt{ref}(n)$, named u, meets the following specification. Let i, X be fresh.

$$\forall n^{\mathsf{Nat}}.\forall \mathrm{X}.\forall i^{\mathrm{X}}.\{\mathsf{T}\}u \bullet n = z\{\nu x.(!z = n \;\wedge\; z\#i \wedge z = x)\}@\emptyset. \tag{2.2}$$

The above assertion 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 that x is freshly generated and is not stored in any existing reference.

We list convenient abbreviations for evaluation formulae for representing "freshness". Below let i be fresh.

- $\{C\}e \bullet e' = z\{\nu\#x.C'\} \;=\; \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e \bullet e' = z\{\nu x.(x\#i \wedge C')\}$
- $\{C\}e \bullet e' = z\{\#z.C'\} \;=\; \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e \bullet e' = z\{\nu y.(z = y \;\wedge\; z\#i \;\wedge\; C'\}$

In the first line, νx says x is a hidden reference distinct from any names in the initial state, giving the weakest form of freshness (x may be replaced by a vector). z and x are distinct by the binding condition. In the second, $\#$ is used instead of inequality. The third is when the return value is unreachably fresh. Its use for 5 above yields:

$$\forall n.\{\mathsf{T}\}u \bullet n = z\{\#z.!z = n\}@\emptyset$$

2.4 Proof Rules

This subsection summarises judgements and proof rules for local reference generation. The judgement consists of a program and a pair of formulae following Hoare [14], augmented with a fresh name called *anchor* [17, 19, 20].

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

which says:

> *If we evaluate M in the initial state satisfying C, then it terminates with a value, name it u, and a final state, which together satisfy C'.*

As this reading indicates, our judgements are about total correctness. They have identical shape as those in [6, 20], even though described computational situations can be quite different, with both C and C' possibly specifying behaviours and data structures with local state.

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 $\{C\}\ M :_u \{C'\}$, 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 $i, j, k, \ldots$) 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.

We also use the following abbreviation similar to those with evaluation formulae. Below let i be fresh.

- $\{C\}M\{C'\}$ stands for $\{C\}M :_u \{u = () \wedge C'\}$ with $u \notin \mathsf{fv}(C')$.
- $\{C\}M :_u \{C'\}@\tilde{x}$ means that $\tilde{x}$ is a write set as for located assertions (cf. § 2.3)
- $\{C\}\ M :_m \{\nu\#x.C'\}$ stands for $\{C\}\ M :_m \{\nu x.(x\#i \wedge C')\}$.
- $\{C\}\ M :_m \{\#m.C'\}$ stands for $\{C\}\ M :_m \{\nu x.(m = x \wedge m\#i \wedge C')\}$.

The full compositional proof rules are given in Figure 2 in Appendix B. In spite of the semantic enrichment, all compositional proof rules in the base logic [6] stay as they were, except for adding the following rule for reference generation.

$$[Ref]\ \frac{\{C\}\ M :_m \{C'\}}{\{C\}\ \mathtt{ref}(M) :_u \{\#u.C'[!u/m]\}}$$

The rule says that the newly generated cell is unreachable from any datum in the initial state: then the result of evaluating M is stored in that cell which is named u. All the so-called structural rules (i.e. those rules which only manipulate assertions) in [6] also remain valid except for an additional condition in one rule, see Appendix B.

Invariant rules are useful for modular reasoning. Their use with (un)reachability needs some care. Suppose x is unreachable from y; after running $y := x$, x becomes reachable from y. Hence the following simple invariant rule for unreachability is unsound.

$$[Unsound\_Inv\_with\_\#]\ \frac{\{C\}\ M :_m \{C'\}}{\{C \wedge e\#e'\}\ M :_m \{C' \wedge e\#e'\}}$$

However the general invariant rule strengthen from our preceding study [6] works in harmony with the (un)reachability predicate.

$$[Inv]\frac{\{C\}\ M :_m \{C'\}@\tilde{w}}{\{C \wedge [!\tilde{w}]C_0\}\ M :_m \{C' \wedge C_0\}@\tilde{w}}$$

The effect set $\tilde{w}$ gives the minimum information by which the assertion we wish to add, C_0, can be stated as an invariant. Since $[!\tilde{w}]C_0$ says C_0 holds regardless of the content of $\tilde{w}$, surely it can stay invariant after execution of M. Unlike the invariant rule in Separation Logic, we need no side condition "M does not modify stores mentioned in C_0": C and C_0 may overlap in their mentioned references, and C does not have to mention all references M may read and write. which are direct instances of [*Inv*] (for the former we observe $\{C\}\ V :_m \{C'\}$ implies $\{C\}\ V :_m \{C'\}@\emptyset$ for any V; for the latter we note $[!x]x\#\tilde{e} \equiv x\#\tilde{e}$ is always valid under the side condition,[1] cf. Proposition 7, clause 3-(5) later).

Another useful structural rule is the following variation of the standard consequence rule.

$$[ConsEval]\frac{\begin{array}{c}\{C_0\}\ M :_m \{C_0'\} \quad x \text{ fresh};\ \ \tilde{i} \text{ auxiliary} \\ \forall\tilde{i}.\{C_0\}x\bullet() = m\{C_0'\} \supset \forall\tilde{i}.\{C\}x\bullet() = m\{C'\}\end{array}}{\{C\}\ M :_m \{C'\}}$$

This rule subsumes the standard consequence rule. In the present logic, the rule further enables non-trivial reasoning on fresh references, as we shall discuss later.

3 Models, Soundness and Completeness

3.1 Models

We introduce operationally-based semantics of the logic, based on term models. For capturing local state, models incorporate hidden locations using a ν-binder [34]. We illustrate the key idea using the Introduction's `Inc` (in (1.1)). We model `Inc` named u as:

$$(\nu l)(\{u : \lambda().(l :=!l+1;!l)\}, \quad \{l \mapsto 0\}) \tag{3.1}$$

(3.1) says that there is a behaviour named u and a reference named l, that this reference stores 0, and that l is hidden. By augmenting (3.1) with fresh j mapped to any location/datum from the initial state (hence disjoint from l), we may assert:

$$\nu x.(!x = 0 \ \wedge\ \forall i.\{!x = i\}u\bullet() = z\{!x = z \wedge !x = i+1\}@x \ \wedge\ x \neq j)$$

which represents the freshness assertion.

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

[1] This side condition is indispensable: consider $\{\mathsf{T}\}x := x\{\mathsf{T}\}@x$, for which it is wrong to conclude $\{x\#!x\}x := x\{x\#!x\}@x$.

– 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:

$$(\nu\tilde{l_1})(C[\langle\tilde{V_1}\rangle],\sigma_1) \Downarrow \quad \text{iff} \quad (\nu\tilde{l_2})(C[\langle\tilde{V_2}\rangle],\sigma_2) \Downarrow$$

where $\langle\tilde{V}\rangle$ is the n-fold pairings of a vector of values.

Definition 2 in effect takes models up to the standard contextual congruence. We could have used a different program equivalence (for example call-by-value $\beta\eta$ convertibility), as far as it is observationally adequate. Note we have

$$(\nu\tilde{l})(\xi\cdot x{:}V_1, \sigma\cdot l \mapsto W_1) \approx (\nu\tilde{l})(\xi\cdot x{:}V_2, \sigma\cdot l \mapsto W_2)$$

whenever $V_1 \cong V_2$ and $W_1 \cong W_2$, where $\cong$ is the standard contextual congruence on programs [40] (for reference Appendix A.3 lists the definition of $\cong$).

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* $\mathsf{lc}(S,\sigma) = \mathsf{fl}(\sigma') = \mathsf{dom}(\sigma')$.

Proof. (1–4) are direct from the definition. (5, 6) immediately follow from (1–4). □

We now set:

$$\mathcal{M} \models e_1 \hookrightarrow e_2 \quad \text{if } [\![e_2]\!]_{\xi,\sigma} \in \mathsf{lc}(\mathsf{fl}([\![e_1]\!]_{\xi,\sigma}),\sigma) \text{ for each } (\nu\tilde{l})(\xi,\sigma) \approx \mathcal{M} \tag{3.2}$$

Above $[\![e_i]\!]_{\xi,\sigma}$ is the obvious interpretation of e_i in the open model (see Appendix C). The clause says that the set of hereditarily reachable names from e_1 includes e_2 up to $\approx$. For programs in § 2.3 (4), 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 for fresh names. Below $\sigma = \sigma_1 \uplus \sigma_2$ indicates σ is the union of σ_1 and σ_2 such that $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset$.

Proposition 4 (partition). *$\mathcal{M} \models x\#u$ if and only if 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 $l \in \mathsf{dom}(\sigma_2)$.*

Proof. For the only-if direction, assume $\mathcal{M} \models x\#u$. By the definition of (un)reachability, we can set (up to $\approx$) $\mathcal{M} \stackrel{\text{def}}{=} (\nu\tilde{l'})(\xi \cdot u : V \cdot x : l,\ \sigma)$ such that $l \notin \mathsf{lc}(\mathsf{fl}(V), \sigma)$. Now take σ_1 such that $\mathsf{lc}(\mathsf{fl}(V), \sigma) = \mathsf{lc}(\mathsf{fl}(V), \sigma_1) = \mathsf{fl}(\sigma_1) = \mathsf{dom}(\sigma_1)$ by Lemma 3. Note by definition $l \notin \mathsf{dom}(\sigma_1)$. Now let $\sigma_2 \stackrel{\text{def}}{=} \sigma \backslash \mathsf{dom}(\sigma_1)$. Since $l \in \mathsf{dom}(\sigma)$, we know $l \in \mathsf{dom}(\sigma_2)$, hence done. The if-direction is obvious by definition of reachability. □

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 universal hiding quantifier has the following semantics:

$$\mathcal{M} \models \overline{\nu} x.C \stackrel{\text{def}}{\equiv} \forall \mathcal{M}', l.((\nu l)(\mathcal{M}'/x) \approx \mathcal{M} \wedge \mathcal{M}'(x) = l \supset \mathcal{M}' \models C)$$

where l is fresh. Above the initial (νl) adds l to the hiding of $\mathcal{M}'$, $\mathcal{M}'/x$ takes off the x-component from $\mathcal{M}'$, and $\mathcal{M}'(x) = l$ says x is assigned a free label l in $\mathcal{M}'$. Dually:

$$\mathcal{M} \models \nu x.C \stackrel{\text{def}}{\equiv} \exists \mathcal{M}', l.((\nu l)(\mathcal{M}'/x) \approx \mathcal{M} \wedge \mathcal{M}'(x) = l \supset \mathcal{M}' \models C)$$

where l is again fresh. 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

$$\mathcal{M} \models \nu x^{\mathsf{Ref(Nat)}}.(!x = 0 \ \wedge\ \forall i^{\mathsf{Nat}}.\{!x = i\}u \bullet () = z\{z = !x = i + 1\})$$

because if we set

$$\mathcal{M}' \quad \stackrel{\text{def}}{=} \quad (\{u : \lambda().(l := !l + 1; !l), x : l\},\ \{l \mapsto 0\})$$

then we have

$$(\nu l)(\mathcal{M}'/x) \ = \ \mathcal{M} \qquad \text{and} \qquad \mathcal{M}' \models C.$$

Intuitively, $\mathcal{M}$ represents a situation where a reference l is hidden, x denotes l, and u denotes a function which increments and returns the content of l.

3.3 Soundness and observational completeness.

The definitions of satisfiability $\mathcal{M} \models C$ other than reachability is given in Appendix C (logical connectives are interpreted classically: type variables are treated syntactically [19]). 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 the validity $\models \{C\}M :_u \{C'\}$ is given by:

$$\models \{C\}M :_u \{C'\} \stackrel{\text{def}}{\equiv} \forall \mathcal{M}.(\mathcal{M} \models C \Rightarrow \mathcal{M}[u:M] \Downarrow \mathcal{M}' \models C')$$

where we write $\mathcal{M}[u:N] \Downarrow \mathcal{M}'$ when $(N\xi, \sigma) \Downarrow (\nu\tilde{l'})(V, \sigma')$ and $\mathcal{M}' = (\nu\tilde{l}\tilde{l'})(\xi \cdot u:V,\ \sigma')$. Above we demand, for well-definedness, that $\mathcal{M}$ includes all variables in M, C and C' except u. The soundness result follows.

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

Proof. See Appendix C.4. □

We next discuss the completeness properties of the logic. A strong completeness property is *descriptive completeness* studied in [18], which is provability of a characteristic assertion for each program (i.e. assertions characterising programs' behaviour). In [18], 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 [18]. Relative and observational completeness are its direct corollaries. Descriptive completeness is established for a refinement of the present logic, given in Appendix C.3; evaluation formulae and content quantification are decomposed into a pair of fine-grained operators, which can represent the original ones. This refinement is not necessary for many reasoning examples and reading the rest of this paper.

We leave the details to a separated full version [26], and we state only the observational completeness, which we regard as a basic semantic property of the logic.

Write $\cong$ for the standard contextual congruence for programs [40]; further write $M_1 \cong_{\mathcal{L}} M_2$ to mean ($\models \{C\}M_1 :_u \{C'\}$ iff $\models \{C\}M_2 :_u \{C'\}$), with $\models$ as refined in Appendix C.3. 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

This section studies axioms for assertions involving (un)reachability. We start from basic axioms. The proofs use Lemma 3. Note our types include recursive types (taken up to tree unfolding [40]).

Proposition 7 (axioms for reachability). *The following assertions are valid (we assume appropriate typing).*

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) $[!x]y \hookrightarrow x \equiv y \hookrightarrow x \equiv \langle !x\rangle y \hookrightarrow x$.

Proof. 1, 2 and 3.(1–4) are direct from the definition (e.g. for 3-(2) we observe $l \in \mathsf{fl}(\mathtt{inj}_i(V))$ iff $l \in \mathsf{fl}(V)$). For 3-(5), suppose $\mathcal{M} \models y \hookrightarrow x$, and take $\mathcal{M}'$ which only differ from $\mathcal{M}$ in the stored value at (the reference denoted by) x. Since $\mathcal{M} \models y \hookrightarrow x$ holds, there is a shortest sequence of connected references from y to x which, by definition, does not include x as its intermediate node. Hence this sequence also exists in $\mathcal{M}'$, i.e. $\mathcal{M}' \models y \hookrightarrow x$, proving $[!x]y \hookrightarrow x \equiv y \hookrightarrow x$. This also means $\langle !x\rangle[!x]y \hookrightarrow x \equiv \langle !x\rangle y \hookrightarrow x$. By the axiom of content quantification we have $\langle !x\rangle[!x]y \hookrightarrow x \equiv [!x]y \hookrightarrow x$, hence done. □

3-(5) says that altering the content of x does not affect reachability *to* x. Note $[!x]x \hookrightarrow y$ is not valid at all. The dual of 3-(5): $[!x]x\#y \equiv x\#y \equiv \langle !x\rangle x\#y$ was already used for deriving [*Inv*-#] in §2.4 (we cannot substitute $!x$ for y in $[!x]x\#y$ to avoid name capture, cf. [6, §5.2 (long version)]).

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.

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'.*

Proof. By Proposition 7 (3). □

A straightforward coinductive extension of the above axioms (see [2]) gives a complete axiomatisation when the types also contain recursive types, but not function types.

For analysing reachability, it is useful to define the following "one-step" reachability predicate. Below e_2 is of a reference type.

$$\mathcal{M} \models e_1 \rhd e_2 \quad \text{if } [\![e_2]\!]_{\xi,\sigma} \in \mathsf{fl}([\![e_1]\!]_{\xi,\sigma}) \text{ for each } (\nu\tilde{l})(\xi,\sigma) \approx \mathcal{M} \tag{4.1}$$

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 l' is in the support of f in the sense of [13, 43, 50]). Now define:

$$\begin{aligned} x \rhd^1 y &\equiv x \rhd y \\ x \rhd^{n+1} y &\equiv \exists z.(x \rhd z \wedge !z \rhd^n y) \quad (n \geq 1) \end{aligned}$$

We also set $x \rhd^0 y \equiv x = y$. By definition we have:

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 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 (undecidability of $\triangleright$ and $\hookrightarrow$). (1) $\mathcal{M} \models f^{\alpha \Rightarrow \beta} \triangleright x$ *is undecidable.* (2) $\mathcal{M} \models f^{\alpha \Rightarrow \beta} \hookrightarrow x$ *is undecidable.*

Proof. For (1), let $V \stackrel{\text{def}}{=} \lambda().\texttt{if}\ M = ()\ \texttt{then}\ l\ \texttt{else}\ \mathsf{Ref}(0)$ with a closed PCFv-term M of type Unit. Then $f : V, x : l \models f \triangleright x$ iff $M \Downarrow$, reducing the satisfiability to the halting problem of PCFv-terms. For (2), take the same V so that the type of l and x is $\mathsf{Ref}(\mathsf{Nat})$ in which case $\triangleright$ and $\hookrightarrow$ coincide. □

The proof above indicates that the same result holds even if we take call-by-value $\beta\eta$-equality as the underlying equality. Further the result also implies that the validity of $\forall f, x.(A \supset f \triangleright x)$ is undecidable, since we can represent any PCFv-term as a formula using the method [18].

Proposition 10 does not imply we cannot obtain useful axioms for (un)reachability involving function types. We discuss a collection of basic axioms in the following.

Proposition 11 (unreachable function). *The following assertion is valid:* $\{C\} f \bullet y = z \{C'\} @ \tilde{w} \supset \{C \wedge x \# f y \tilde{w}\} f \bullet y = z \{C' \wedge x \# z \tilde{w}\} @ \tilde{w}$.

Proof. See Appendix D. □

Proposition 11 says that if x is unreachable from a function f, its argument y and its write set $\tilde{w}$, then the execution of this function does not return or write x.

When we do need to reason about a function with a local state, its behaviour often crucially relies on an invariant on its local, or hidden, store. We first list basic axioms for hiding quantifiers, presented for the existential (which is mainly used in reasoning). The proofs are all easy and omitted.

Proposition 12 (axioms for ν).

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

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.[2] Note this shows that integrating these quantifiers into the standard universal/existential quantifiers let the latter lose their standard axioms, motivating the introduction of ν-operator. (4,5) list some of useful axioms for moving the scope of x.

[2] As a simple example of the former, let $\mathcal{M} \stackrel{\text{def}}{=} (\{x : l,\ x' : l\},\ \{l \mapsto 5\})$. Then $\mathcal{M} \models x = x'$ but we do *not* have $\mathcal{M} \models \nu y.y = x'$ since l is certainly not hidden (x is renamed to fresh y to avoid confusion). For the latter, let $\mathcal{M} \stackrel{\text{def}}{=} (\nu l)(\{u : \lambda().!l\},\ \{l \mapsto 5\})$. Then $\mathcal{M} \models \nu x.\forall i.\{!x = i\} u \bullet () = z \{z = !x \wedge !x = i\}$. From this, we have $\mathcal{M} \models \nu x.\exists y, i.\{!y = i\} u \bullet () = z \{z = 0\}$. Also by definition of $\mathcal{M}$, $\mathcal{M} \models \{\mathsf{T}\} u \bullet () = z \{z = 5\}$. Hence $\mathcal{M} \models \exists y, i.\{!y = i\} u \bullet () = z \{z = 0\}$ does not hold.

4.1 Local Invariant

We now introduce an axiom for local invariants. Let us first consider a function which writes to local reference of a base type. Even programs of this kind pose fundamental difficulties in reasoning, as show in [31]. Take the following program:

$$\texttt{compHide} \stackrel{\text{def}}{=} \texttt{let}\ x = \texttt{ref}(7)\ \texttt{in}\ \lambda y.(y > !x) \tag{4.2}$$

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 survive under arbitrary invocations of `compHide`: thus (naming the function u) we arrive at the following invariant:

$$C_0 \ = \ !x = 7 \ \wedge \ \forall y.\{!x = 7\}u \bullet y = z\{!x = 7\}@\emptyset \tag{4.3}$$

Assertion (4.3) 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 with i fresh:

$$\{\mathsf{T}\}\texttt{compHide} :_u \{\nu x.(x\#i^{\mathrm{X}} \ \wedge \ C_0 \ \wedge \ C_1)\} \tag{4.4}$$

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:

$$\{\mathsf{T}\}\texttt{compHide} :_u \{\forall y.\{\mathsf{T}\}u \bullet y = z\{z = (y > 7)\}\} \tag{4.5}$$

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]

$$\mathsf{Inv}(u, C_0, \tilde{x}) \ = \ C_0 \ \wedge \ (\forall y i.\{C_0\}u \bullet y = z\{\mathsf{T}\} \supset \forall y i.\{C_0\}u \bullet y = z\{C_0 \ \wedge \ \tilde{x}\#z\}) \tag{4.6}$$

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 $\tilde{x}$.

We say C is *stateless* if $\mathcal{M} \models C$ and $\mathcal{M}[u : N] \Downarrow \mathcal{M}'$ imply $\mathcal{M}' \models C$. This assertion can be syntactically defined as syntactically.

Definition 13 (Stateless Formulae). We say 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). $A, B, \ldots$ range over stateless formulae.

Above a formula C occurs *negatively* if it occurs in C_1 of $C_1 \supset C_2$ or in C of $\neg C$. The property of stateless formulae is studied in Appendix D.1.

[3] That is sufficient for all examples in this paper. The refinement of formulae in § 3.3 allows y to be of arbitrary type.

Proposition 14 (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. Then the following assertion is valid*

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

with $E_1 = \mathsf{Inv}(u, C_0, \tilde{x}) \wedge \forall yi.\{C_0 \wedge [!\tilde{x}]C\}u\bullet y = z\{C'\}@\tilde{w}\tilde{x}$ and $E_2 = \forall y.\{C\}u\bullet y = z\{C'\}@\tilde{w}$.

Proof. See Appendix D.3.

(AIH) is used with the consequence rule (Appendix B) 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.

(AIH) eliminates ν from the postcondition based on local invariants. The following axiom also eliminates νx, this time solely based on freshness and disjointness of x.

Proposition 15 (ν-elimination). *Let $x \notin \mathsf{fv}(C)$ and m,i,X be fresh. Then the following is valid:*

$$\forall \mathrm{X}, i^{\mathrm{X}}.\{E\}m\bullet() = u\{\nu x.([!x]C \wedge x\#ui^{\mathrm{X}})\} \ \supset\ \{E\}m\bullet() = u\{C\}$$

This proposition says that if a restricted x in the post-state is completely hidden and is disjoint from any visible datum, then we can safely neglect it. Note so-called *stack-allocated variables* (i.e. statically declared variables in a block in block-structured languages) are used in this way. The proof is similar to Proposition 14.

The following axiom stipulates how an invariant is *transferred* by functional applications. The proof is similar to Propositions 11 and 14.

Proposition 16 (invariant by application). *Assume $[!\tilde{x}]C_0$ is stateless, $y \notin \mathsf{fv}(C_0)$ and y has a base type. Then the following is valid.*

$$(\forall y.\{C_0\}f\bullet y = z\{C_0\}@\tilde{x} \wedge \{C\}g\bullet f = z\{C'\}) \ \supset\ \{C \wedge C_0 \wedge \tilde{x}\#g\}g\bullet f = z\{\tilde{x}\#z \wedge C_0 \wedge C'\}$$

The axiom says that the result of applying a function g disjoint from a local reference x_i, to the argument f which satisfies the local invariant, again preserves the local invariant. It may be considered as a higher-order version of Proposition 11.

5 Reasoning Examples (1): Functions and Local State

5.1 Shared Stored Function

This section demonstrates the usage of the proposed logic through concrete examples. Some of the lengthy derivations are omitted but key ideas are provided.

We first treat `IncShared` from Introduction, a simple example of shared local state with stored functions. We use a proof rule for the combination of "let" and new reference generation, easily derivable from the proof rules in Section 2 through the standard decomposition of "let" into application and abstraction (see the derivation in Appendix E.1).

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

Above $\mathsf{fpn}(e)$ denotes the set of *free plain names* of e which are reference names in e that does not occur in dereference, defined as: $\mathsf{fpn}(x) = \{x\}$, $\mathsf{fpn}(\mathsf{c}) = \mathsf{fpn}(!e) = \emptyset$, $\mathsf{fpn}(\langle e, e'\rangle) = \mathsf{fpn}(e) \cup \mathsf{fpn}(e')$, $\mathsf{fpn}(\pi_i(e)) = \mathsf{fpn}(e)$ and $\mathsf{fpn}(\mathsf{inj}_i(e)) = \mathsf{fpn}(e)$. The rule reads:

> *Assume* (1) *running M from C leads to 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 the letref command from C leads to C' whose x is fresh and hidden.*

We note:

- 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).
- $\nu x.C'$ cannot be strengthened to $\#x.C'$ since N may store x in an existing reference.

One may note the rule directly gives a proof rule for general new reference declaration [31, 42, 48], $\mathtt{new}\ x := M\ \mathtt{in}\ N$, which has the same operational behaviour as $\mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N$.

We can now treat `IncShared` from Introduction:

$$\mathtt{IncShared} \stackrel{\text{def}}{=} a := \mathtt{Inc}; b := !a; c_1 := (!a)(); c_2 := (!b)(); (!c_1 + !c_2)$$

Naming it u, the assertion $\mathsf{inc}'(u,x,n)$ below captures its behaviour:

$$\begin{aligned} \mathsf{inc}(x,u) &= \forall j.\{!x = j\}u \bullet () = j+1\{!x = j+1\}@x. \\ \mathsf{inc}'(u,x,n) &= !x = n \wedge \mathsf{inc}(x,u). \end{aligned}$$

The following derivation for `IncShared` sheds light on how shared higher-order local state can be transparently reasoned in the present logic. For brevity we work with the implicit global assumption that a, b, c_1, c_2 are pairwise distinct and safely omit an

anchor from the judgement when the return value is a unit type.

$$
\begin{array}{ll}
1.\{\mathsf{T}\}\ \mathtt{Inc} :_u \{\nu x.\mathsf{inc}'(u,x,0)\} & \\
\hline
2.\{\mathsf{T}\}\ a := \mathtt{Inc}\ \{\nu x.\mathsf{inc}'(!a,x,0)\} & (1,\ \text{Assign}) \\
\hline
3.\{\mathsf{inc}'(!a,x,0)\}\ b := !a\ \{\mathsf{inc}'(!a,x,0) \wedge \mathsf{inc}'(!b,x,0)\} & (\text{Assign}) \\
\hline
4.\{\mathsf{inc}'(!a,x,0)\}\ c_1 := (!a)()\ \{\mathsf{inc}'(!a,x,1) \wedge !c_1 = 1\} & (\text{Assign}) \\
\hline
5.\{\mathsf{inc}'(!b,x,1)\}\ c_2 := (!b)()\ \{\mathsf{inc}'(!b,x,2) \wedge !c_2 = 2\} & (\text{App etc.}) \\
\hline
6.\{!c_1 = 1 \wedge !c_2 = 2\}\ (!c_1) + (!c_2) :_u \{u = 3\} & (\text{Deref etc.}) \\
\hline
7.\{\mathsf{T}\}\ \mathtt{IncShared} :_u \{\nu x.u = 3\} & (2\text{–}6,\ \text{LetOpen}) \\
\hline
8.\{\mathsf{T}\}\ \mathtt{IncShared} :_u \{u = 3\} & (\text{Conseq})
\end{array}
$$

Line 1 is by [*LetRef*]. Line 7 uses the following derived rule (noting sequential composition is a special case of "let"):

$$[\mathit{LetOpen}]\ \frac{\{C\}\ M :_x \{\nu\tilde{y}.C_0\} \quad \{C_0\}\ N :_u \{C'\}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{\nu\tilde{y}.C'\}}$$

Line 8 uses Proposition 12-3 (note C does not contain evaluation formulae) and $\exists x.C \supset C$. To shed light on how the difference in sharing is captured in inferences, Appendix E.2 lists the inference for a program which assigns *distinct* copies of `Inc` to a and b.

5.2 Information Hiding (1): Memoisation

Next we treat a memoised factorial [44] from Introduction.

$$
\begin{array}{l}
\mathtt{memFact} \stackrel{\text{def}}{=} \mathtt{let}\ a = \mathtt{ref}(0),\ \ b = \mathtt{ref}(1)\ \mathtt{in} \\
\qquad \lambda x.\mathtt{if}\ x = !a\ \mathtt{then}\ !b\ \mathtt{else}\ (a := x; b := \mathtt{fact}(x); !b)
\end{array}
$$

Our target assertion specifies the behaviour of a pure factorial.

$$\mathit{Fact}(u) = \forall x.\{\mathsf{T}\}u \bullet x = y\{y = x!\}@\emptyset.$$

The following inference starts from the body of the "let", which we name V. We set: $E_{1a} = C_0 \wedge \forall x.\{C_0\}u \bullet x = y\{C_0\}@ab$, and $E_{1b} = \forall x.\{C_0 \wedge C\}u \bullet x = y\{C'\}@ab$ where we let C_0 be $ab\#i \wedge !b = (!a)!$, C be T and C' be $y = x!$. Note that $[!ab]C_0$ is stateless (in the sense of Definition 13, page 15).

$$
\begin{array}{ll}
1.\{\mathsf{T}\}\ V :_u \{\forall x.\{!b = (!a)!\}u \bullet x = y\{y = x! \wedge !b = (!a)!\}@ab\} & \\
\hline
2.\{\mathsf{T}\}\ V :_u \{E_{1a} \wedge E_{1b}\} & (1,\ \text{Conseq}) \\
\hline
3.\{ab\#i\}\ V :_u \{ab\#i \wedge E_{1a} \wedge E_{1b}\} & (2,\ \text{Inv-Val}) \\
\hline
4.\{\mathsf{T}\}\ \mathtt{memFact} :_u \{\nu\#ab.(E_{1a} \wedge E_{1b})\} & (3,\ \text{LetRef}) \\
\hline
5.\{\mathsf{T}\}m\bullet() = u\{\nu\#ab.(E_{1a} \wedge E_{1b})\} \supset \{\mathsf{T}\}m\bullet() = u\{\mathit{Fact}(u)\} & (\star) \\
\hline
6.\{\mathsf{T}\}\ \mathtt{memFact} :_u \{\mathit{Fact}(u)\} & (4,\ 5,\ \text{ConsEval})
\end{array}
$$

Line 2 used $\{C\}f\bullet x{=}y\{C_1\wedge C_2\}@\tilde{w}\supset\wedge_{i=1,2}\{C\}f\bullet x=y\{C_i\}@\tilde{w}$ (from [6, 20]). $(\star)$ in Line 5 is by (AIH) in Proposition 14.

5.3 Information Hiding (2): Stored Circular Procedures

We next consider `circFact` from Introduction, which uses a self-recursive higher-order local store.

$$\begin{array}{l}\texttt{circFact}\stackrel{\text{def}}{=}x:=\lambda z.\texttt{if}\ z=0\ \texttt{then}\ 1\ \texttt{else}\ z\times(!x)(z-1)\\ \texttt{safeFact}\stackrel{\text{def}}{=}\texttt{let}\ x=\texttt{ref}(\lambda y.y)\ \texttt{in}\ (\texttt{circFact};!x)\end{array}$$

In [20], we have derived the following judgement.

$$\{\mathsf{T}\}\texttt{circFact}:_u\{\mathit{CircFact}(u,x)\}@x \tag{5.1}$$

where

$$\mathit{CircFact}(u,x)\ =\ \forall n.\{!x=u\}!x\bullet n=z\{z=n!\wedge!x=u\}@\emptyset\wedge\ !x=u$$

which says:

> *After executing the program, x stores a procedure which would calculate a factorial if x stores that behaviour, and that x does store the behaviour.*

We now show `safeFact` named u satisfies $\mathit{Fact}(u)$. Below we use: $CF_a\ =\ !x=u\wedge\forall n.\{!x=u\}!x\bullet n=z\{!x=u\}@\emptyset$ as well as $CF_b\ =\ \forall n.\{!x=u\}!x\bullet n=z\{z=n!\}@\emptyset$.

1. $\{\mathsf{T}\}\lambda y.y:_m\{\mathsf{T}\}@\emptyset$	
2. $\{\mathsf{T}\}\texttt{circFact};!x:_u\{\mathit{CircFact}(u,x)\}@x$	
3. $\{\mathsf{T}\}\texttt{circFact};!x:_u\{CF_a\ \wedge\ CF_b\}@x$	(2, Conseq)
4. $\{x\#i\}\texttt{circFact};!x:_u\{x\#i\wedge CF_a\wedge CF_b\}@x$	(3, Inv-#)
5. $\{\mathsf{T}\}\texttt{safeFact}:_u\{\nu\#x.(CF_a\wedge CF_b)\}@\emptyset$	(4, LetRef)
6. $\{\mathsf{T}\}m\bullet()=u\{\nu\#x.(CF_a\wedge CF_b)\}\supset\{\mathsf{T}\}m\bullet()=u\{\mathit{Fact}(u)\}$	$(\star)$
7. $\{\mathsf{T}\}\texttt{safeFact}:_u\{\mathit{Fact}(u)\}@\emptyset$	(5, 6, ConsEval)

Line 1 is immediate. Line 2 is (5.1). Line 6, $(\star)$ is by (AIH), Proposition 14, setting $C_0={!x=u}$, $C=E'=\mathsf{T}$ and $C'=y=x!$. Note this example can again use (AIH) since the behaviour in question is indeed first-order.

The reasoning easily extends to programs which use multiple locally stored, and mutually recursive, procedures. Consider:

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

After these two assignments, the application $(!x)n$, with n a natural number, returns true if n is odd, false if not; while $(!y)n$ acts dually. Informally the state of affairs may be described thus:

x stores a procedure which checks if its argument is odd, if y stores a procedure which does the dual; whereas y stores a procedure which checks whether its argument is even or not if x stores a procedure which does the dual.

Observe mutual circularity of this description. As before, we can avoid unexpected interference at x and y using local references.

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

Above $\lambda n.\texttt{t}$ can be any initialising value. Now that x,y are inaccessible, the programs behave as pure functions, e.g. $\texttt{safeOdd}(3)$ always returns true without any side effects, similarly $\texttt{safeOdd}(16)$ always returns false, To formally validate these behaviours, we can first verify the body of the "let" satisfies the following assertions.

$$\{\mathsf{T}\}\texttt{mutualParity} :_u \{\exists gh.\mathit{IsOddEven}(gh, !x!y, xy, n)\} \tag{5.2}$$

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

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

Our aim is to derive the following judgements starting from (5.2).

$$\{\mathsf{T}\}\texttt{safeOdd} :_u \{\forall n.\{\mathsf{T}\}u \bullet n = z\{z = \mathit{Odd}(n)\}@\emptyset\} \tag{5.3}$$
$$\{\mathsf{T}\}\texttt{safeEven} :_u \{\forall n.\{\mathsf{T}\}u \bullet n = z\{z = \mathit{Even}(n)\}@\emptyset\} \tag{5.4}$$

We reason for `safeOdd` (the case for `safeEven` is symmetric). We first identify the local invariant:

$$C_0 \ = \ !x = g \wedge \ !y = h \ \wedge \ \mathit{IsEven}(h, gh, n, xy)$$

The free variable h suggests the use of (AIH). Since C_0 only talks about g, h and the content of x and y, we know C_0 is stateless except xy. We now observe $\mathit{IsOddEven}(gh, !x!y, xy, n)$ is the conjunction of:

$$\mathit{Odd}_a = C_0 \ \wedge \ \forall n.\{C_0\}u \bullet n = z\{C_0\}@xy$$
$$\mathit{Odd}_b = \forall n.\{C_0\}u \bullet n = z\{z = \mathit{Odd}(n)\}@xy$$

We can now apply $(\mathsf{AIH}_{A\exists})$ to obtain (5.3). Full inferences can be found in Appendix E.3.

5.4 Information Hiding (3): Higher-Order Invariant

We move to a program whose invariant behaviour depends on another function [50, p.104]. 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. Thus our aim is to derive:

$$\{\forall y.\{C\}f \bullet y = z\{C'\}@\tilde{w}\}\ \texttt{profile} :_u \{\forall y.\{C\}u \bullet y = z\{C'\}@\tilde{w}\} \tag{5.5}$$

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 (5.5), 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:

$$\{\mathsf{T}\}\lambda y.(x :=!x+1; fy) :_u \{\forall yi.\{C_0\}u \bullet y = z\{C_0 \wedge x\#z\}@x\tilde{w}\} \tag{5.6}$$

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:

$$\{E\}\lambda y.(x :=!x+1; fy) :_u \{\forall yi.\{C_0 \wedge [!x]C\}u \bullet y = z\{C' \wedge x\#z\}@x\tilde{w}\}. \tag{5.7}$$

By combining (5.6) and (5.7) by the standard structural rule, we can use (AIH), hence done. The detailed derivation can be found in Appendix E.4.

5.5 Information Hiding (4): Nested Local Invariant

The next example uses a function with local state as an argument to another function. Let $\Omega \stackrel{\text{def}}{=} \mu f.\lambda().(f())$. $even(n)$ tests for evenness of n.

$$\begin{array}{l}\texttt{MeyerSieber} \stackrel{\text{def}}{=} \texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}\ \texttt{let}\ f = \lambda().x :=!x+2 \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{in}\ (gf\ ;\ \texttt{if}\ even(!x)\ \texttt{then}\ ()\ \texttt{else}\ \Omega())\end{array}$$

Note $\Omega()$ immediately diverges. Since x is local, and because g will have no way to access x except by calling f, the local invariant that x stores an even number is maintained. Hence `MeyerSieber` satisfies the judgement:

$$\{E \wedge C\}\ \texttt{MeyerSieber}\ \{C'\} \tag{5.8}$$

where, with $x, m \notin \mathsf{fv}(C, C')$: $E = \forall f.(\{\mathsf{T}\}f \bullet ()\{\mathsf{T}\}@\emptyset \supset \{C\}g \bullet f\{C'\})$ (anchors of type Unit are omitted). The judgement (5.8) says that: *if feeding g with a total and effect-free f always satisfies $\{C\}g \bullet f\{C'\}$, then* `MeyerSieber` *starting from C also terminates with the final state C'*. Note such f behaves as `skip`. For the derivation of (5.8), from an axiom for reachability we can derive $E \supset E'$ where $E' = \forall f.(\{\mathsf{T}\}f \bullet ()\{\mathsf{T}\}@x \supset \{[!x]C \wedge x\#g\}g \bullet f\{[!x]C'\})$. Further $\lambda().x :=!x+2$ named f satisfies both $A_1 \stackrel{\text{def}}{=} \{\mathsf{T}\}f \bullet ()\{\mathsf{T}\}@x$ and $A_2 \stackrel{\text{def}}{=} \{Even(!x)\}f \bullet ()\{Even(!x)\}@x$. From A_1 and E' we obtain $A_1' \stackrel{\text{def}}{=} \{[!x]C \wedge x\#g\}g \bullet f\{[!x]C'\}$. Using Prop. 16, A_1' and A_2 we obtain:

$$\{Even(!x) \wedge [!x]C \wedge E \wedge x\#gi\}\texttt{let}\ f = \lambda().x :=!x+2\ \texttt{in}\ (gf\ ;\ ...)\{[!x]C' \wedge x\#i\}$$

We then apply a variant of [*LetRef*] (replacing $C_0[!x/m]$ in the premise of [*LetRef*] in §2.4 with $[!x]C_0 \wedge !x = m$) to obtain $\{E \wedge C\}\ \texttt{MeyerSieber}\ \{\nu x.([!x]C' \wedge x\#i)\}$. Finally by Prop. 15 (noting the returned value has a base type, cf. Prop.7 2-(1)), we reach $\{E \wedge C\}\ \texttt{MeyerSieber}\ \{C'\}$. The full derivation is given in Appendix E.5.

5.6 Information Hiding (5): Object

As a final example of this section, we treat information hiding for a program with state, a small object encoded in imperative higher-order functions, taken from [21] (cf.[10, 40, 41]). The following program generates a simple object each time it is invoked.

$$\texttt{cellGen} \stackrel{\text{def}}{=} \lambda z.\begin{pmatrix}\texttt{let}\ x_{0,1} = \texttt{ref}(z)\ \texttt{in}\ \texttt{let}\ y = \texttt{ref}(0)\ \texttt{in}\\ \begin{pmatrix}\lambda().\texttt{if}\ \mathit{even}(!y)\ \texttt{then}\ !x_0\ \texttt{else}\ !x_1,\\ \lambda w.(y :=!y+1\ ;\ x_{0,1} := w\end{pmatrix}\end{pmatrix}$$

The object has a getter and a setter. Instead of having one local variable, it uses two with the same content, of which one is read at each odd-turn of the "read" requests, another at each even-turn. When writing, it writes the same value to both. Since having two variables in this way does not differ from having only one observationally, we expect the following judgement to hold `cellGen`:

$$\{\mathsf{T}\}\ \texttt{cellGen} :_u \{\mathit{CellGen}(u)\} \tag{5.9}$$

where we set:

$$\mathit{CellGen}(u) = \forall z.\{\mathsf{T}\}u \bullet z = o\{\nu\#x.(\mathit{Cell}(o,x)\wedge !x = z)\}@\emptyset$$
$$\mathit{Cell}(o,x) = \forall v.\{!x = v\}\pi_1(o) \bullet () = z\{z = v =!x\}@\emptyset\ \wedge \forall w.\{\mathsf{T}\}\pi_2(o) \bullet w\{!x = w\}@x$$

$\mathit{Cell}(o,x)$ says that $\pi_1(o)$, the getter of o, returns the content of a local variable x; and $\pi_2(o)$, the setter of o, writes the received value to x. Then $\mathit{CellGen}(u)$ says that, when u is invoked with a value, say z, an object is returned with its initial fresh local state initialised to z. Note both specifications only mention a single local variable. A straightforward derivation of (5.9) uses $!x_0 =!x_1$ as the invariant to erase x_1: then we α-converts x_0 to x to obtain the required assertion $\mathit{Cell}(o,x)$. See Appendix E.6 for full inferences.

6 Reasoning Examples (2): Higher-Order Mutable Data Structures

6.1 Circular Lists

This section introduces a reasoning method applicable to a general class of higher-order mutable data types through examples. The method uses a predicate on navigating paths over a network of data nodes for asserting on such a network; and the (un)reachability for their dynamic generation. Types play a prominent role.

We first consider the following program, which stores the constant 0 function at all nodes of a cyclic list [23, §1]. Let:

$$\mathit{List}(\alpha) = \mu \mathrm{X}.(\mathsf{Unit} + (\mathsf{Ref}(\alpha) \times \mathsf{Ref}(\mathrm{X}))).$$

which describes a mutable list using a sum (nil or cons) and a product (two cons cells, the first storing a value of type α and the second the next node of the list). The program

then reads:

$$
\begin{array}{l}
\mathsf{cyclesimple} \stackrel{\mathrm{def}}{=} \\
\quad \mu f.\lambda x^{\mathsf{Ref}(\mathit{List}(\mathsf{Nat}\Rightarrow\mathsf{Nat}))}.\mathtt{case}\ !x\ \mathtt{of} \\
\qquad \mathtt{in}_1(()) : () \\
\qquad \mathtt{in}_2(\langle y_1, y_2\rangle) : (y_1 := \lambda x^{\mathsf{Nat}}.0\ ;\ \mathtt{if}\ y_2 \neq z\ \mathtt{then}\ f y_2\ \mathtt{else}\ ())
\end{array}
$$

cyclesimple receives a node in a cyclic list. By its type, the content of the node is either $\mathtt{in}_1(())$, a nil node, or $\mathtt{in}_2(\langle y_1, y_2\rangle)$, a cons cell. If the argument is the latter, the program stores the zero function in its first field, and via its second field moves to the next cell and processes it, until coming back to the initial cell z. We can check that, as far as z is part of a cycle, the evaluation of cyclesimple$z$ zeroes all the nodes reachable from z.

An assertion for this program should specify the expected shape of the argument (i.e. it is a cycle) and how it is transformed into exactly the same cycle except for all of its fields storing the zero functions. We start from defining easy-to-read notations for the data types of the two components of a list, the nil and the cons.

$$
\begin{array}{rcl}
\mathsf{nil}(u) & \equiv & u = \mathtt{inj}_1(()) \\
\mathsf{cons}(u, y_1, y_2) & \equiv & u = \mathtt{inj}_2(\langle y_1, y_2\rangle)
\end{array}
$$

Below we introduce the key building blocks of the proposed method, adaptable to a wide range of higher-order mutable data structures.

$$
\begin{array}{rcl}
\mathsf{path}(g, 0, g') & \equiv & g = g' \\
\mathsf{path}(g, p+1, g') & \equiv & \exists y, y'(\mathsf{cons}(!g, y, y') \wedge \mathsf{path}(y', p, g'))
\end{array}
$$

$\mathsf{path}(g, p, g')$ indicates that traversing pth-nodes from g leads to g'. Its semantics is transparently given from that of the original logical language. The following two predicates, defined from the path predicate, is useful for asserting for cyclesimple.

$$
\begin{array}{rcl}
\mathsf{isCycle}(g) & \equiv & \exists p \neq 0.\ \mathsf{path}(g, p, g) \\
\mathsf{distance}(g, p, g') & \equiv & \mathsf{path}(g, p, g') \wedge \forall q.(\mathsf{path}(g, q, g') \supset p \leq q)
\end{array}
$$

$\mathsf{isCycle}(g)$ says the node g is part of a cycle (its negation is linearity of a list); whereas $\mathsf{distance}(g, p, g')$ says the distance (minimum path) between g and g' is p-steps, which is useful when carrying out inductive reasoning on a cyclic list. We can now write down the expected judgement for cyclesimple:

$$
\{\mathsf{T}\}\ \mathsf{cyclesimple} :_u\ \{\mathsf{cycleSimple}(u)\} \tag{6.1}
$$

with the following main assertion $\mathsf{cycleSimple}(u)$:

$$
\forall z.\{\mathsf{isCycle}(z)\}\ u \bullet z\ \{\mathsf{allZeros}(z)\} @ \{w \mid \mathsf{valnode}(z, w)\} \tag{6.2}
$$

where we set:

$$
\begin{array}{rcl}
\mathsf{valnode}(z, y) & \equiv & \exists p g y'.(\mathsf{path}(z, p, g) \wedge \mathsf{cons}(!g, y, y')) \\
\mathsf{allZeros}(z) & \equiv & \forall y.(\mathsf{valnode}(z, y) \supset \mathsf{iszero}(!y)) \\
\mathsf{iszero}(f) & \equiv & \forall x.\{\mathsf{T}\} f \bullet x = y\{y = 0\} @ \emptyset
\end{array}
$$

(6.2) also uses an evaluation formula which uses a generalised write set, described by a predicate. This generalised located assertion $\{C\}x \bullet y = z\{C'\}@\{w|E(w)\}$ roughly corresponds to

$$\forall wi.\{C \wedge \neg E(w) \wedge !w = i\}x \bullet y = z\{C' \wedge !w = i\},$$

saying that all references that may be updated by this evaluation are within the set $\{w|E(w)\}$, allowing us to specify an unbounded number of references as a write set (for the precise semantics of the generalised located assertions, see C.1). Thus $\mathsf{cycleSimple}(u)$ says:

> *If the program u receives z as an argument and if it is a node of a cyclic list, then the program fills all the data fields of this list with the zero function, and does nothing else*,

precisely capturing the behaviour of cyclesimple.

The derivation of (6.1) uses the predicate distance given above for recursion. We first set, with x, y and z of type $List(\alpha)$:

$$x \lesssim_z y \quad \equiv \quad \exists p, p'.(\mathsf{distance}(z,p,x) \wedge \mathsf{distance}(y,p',x) \wedge p \gtrsim p') \tag{6.3}$$

That is, $x \lesssim_z y$ iff the distance from x up to, but not including, z is strictly smaller than that from y. Thus $\lesssim_z$ combined with equality is a well-founded partial order (which is enough for carrying out induction [12]). The reasoning uses the following judgement for induction, writing $\mathsf{cyclesimple}'$ for the program cyclesimple minus the initial recursion:

$$\{\forall m \lesssim_z l.B(f,m)\}\mathsf{cyclesimple}' :_u \{B(u,l)\} \tag{6.4}$$

where we set, with $\mathsf{reach}(z,l)$ standing for $\exists p.\mathsf{path}(z,p,l)$:

$$B(u,l) \equiv \{\mathsf{isCycle}(z) \wedge \mathsf{reach}(z,l)\}u \bullet l\{\mathsf{allZerosUpto}(l,z)\}@\{w|\mathsf{valnodeUpto}(l,w,z)\}.$$

Above $\mathsf{allZerosUpto}(x,z)$ is the variant of $\mathsf{allZeros}(x)$ which says all are zeroed from x up to, but not including, z (i.e. just reaching the last node in the cycle, taking z to the initial node). Similarly for $\mathsf{valnodeUpto}(x,w,z)$. The inference for (6.4) is easy, noting we have $(\mathsf{reach}(z,l) \wedge \mathsf{cons}(l,v,l') \wedge l \neq z) \supset l' \lesssim_z l$. Finally we apply $[Rec]$ (to be precise, its refinement to well-founded partial order [12], cf. Appendix B) to obtain:

$$\{\mathsf{T}\}\mathsf{cyclesimple} :_u \{\forall l.B(u,l)\} \tag{6.5}$$

By instantiating l to z via the consequence rule, we arrive at (6.1).

6.2 Trees

We now treat a program which dynamically generate data structures (note cyclesimple alters, but not generates, a list). We use a slightly more complex data type:

$$Tree(\alpha) \stackrel{\text{def}}{=} \mu \mathrm{X}.(\mathsf{Ref}(\alpha + (\mathrm{X} \times \mathrm{X})))$$

A network of nodes of this type can form a tree, a dag, or a graph. The following program is intended to work only for trees of this type, creating an isomorphic copy of an original tree (cf. [48, §6]).

$$\begin{aligned}\texttt{treeCopy} \quad \stackrel{\text{def}}{=} \mu f.\lambda x^{\mathit{Tree}(\alpha)}.\texttt{case}\ !x\ \texttt{of}\\ &\texttt{in}_1(n) : \texttt{ref}(\texttt{inj}_1(n))\\ &\texttt{in}_2(\langle y_1, y_2\rangle) : \texttt{ref}(\texttt{inj}_2(\langle f y_1, f y_2\rangle))\end{aligned}$$

Note `treeCopy` has type $\mathit{Tree}(\alpha) \Rightarrow \mathit{Tree}(\alpha)$. The program carries out an inductive copy for the tree structure, but does a direct copy at stored data, possibly inducing a sharing. To assert and validate for `treeCopy`, we again use the path predicate. Since a one-step traversal can take either the left branch or the right one, the notion of a path becomes slightly more complex, for which we use the following expressions (added as terms to our logical language).

$$p ::= \varepsilon \mid l.p \mid r.p$$

Above l and r mean left and right branches. Using these terms we can now define the path predicate. First let us set, for brevity:

$$\begin{aligned}\mathsf{atom}(u^{\beta}, x^{\alpha}) &\equiv u = \texttt{inj}_1(x)\\ \mathsf{branch}(u^{\beta}, y_1^{\alpha}, y_2^{\beta}) &\equiv u = \texttt{inj}_2(\langle y_1, y_2\rangle) \quad (\beta = \mathit{Tree}(\alpha))\end{aligned}$$

We can now define the path predicate. We use the same notation $\mathsf{path}(g, p, g')$ (which is henceforth exclusively about $\mathit{Tree}(\alpha)$-typed data, with g and g' of type $\mathit{Tree}(\alpha)$).

$$\begin{aligned}\mathsf{path}(g, \varepsilon, g') &\equiv g = g'\\ \mathsf{path}(g, l.p, g') &\equiv \exists y_1 y_2.(\mathsf{branch}(!g, y_1, y_2) \wedge \mathsf{path}(y_1, p, g'))\\ \mathsf{path}(g, r.p, g') &\equiv \exists y_1 y_2.(\mathsf{branch}(!g, y_1, y_2) \wedge \mathsf{path}(y_2, p, g'))\end{aligned}$$

The first clause says that the empty path leads from g to g; the second says that the path $l.p$ leads from g to g' iff we go left from g and, from there, p leads to g'. The third is the symmetric case.

As for linked lists, the path predicate allows us to shape the assertions useful for specifying the behaviour of `treeCopy`.

$$\begin{aligned}\mathsf{match}(g, p_1, p_2) &\equiv \exists y.(\mathsf{path}(g, p_1, y) \wedge \mathsf{path}(g, p_2, y))\\ \mathsf{leaf}(g, p, x) &\equiv \exists y.(\mathsf{path}(g, p, y) \wedge \mathsf{atom}(!y, x))\\ \mathsf{iso}(g, g') &\equiv \forall p_1 p_2.(\mathsf{match}(g, p_1, p_2) \equiv \mathsf{match}(g', p_1, p_2))\\ &\quad \wedge \quad \forall p x.(\mathsf{leaf}(g, p, x) \equiv \mathsf{leaf}(g', p, x))\end{aligned}$$

As before, $\mathsf{match}(g, p_1, p_2)$ asserts two paths $p_{1,2}$ from g lead to an identical node; $\mathsf{leaf}(g, p, x)$ says we reach a leaf storing x (of type α) from g following p. $\mathsf{iso}(g, g')$ asserts two collections of nodes, respectively reachable from g and g', form isomorphic labelled directed graphs. Further we set:

$$\begin{aligned}\mathsf{tree}(g) &\equiv \forall p_1, p_2.(p_1 \neq p_2 \supset \neg \mathsf{match}(g, p_1, p_2))\\ \mathsf{distance}(g, p, g') &\equiv \mathsf{path}(g, p, g') \wedge \forall q.(\mathsf{path}(g, q, g') \supset p \sqsubseteq_{\mathsf{lex}} q)\end{aligned}$$

$\mathsf{tree}\,(g)$ says g is a tree iff it has no sharing. $\mathsf{distance}(g,p,g')$ defines the shortest path from g to g', where paths are ordered by the lexicographic ordering $\sqsubseteq_{\mathsf{lex}}$ (with the "left" smaller than the "right", and the empty string being the least). This gives a basis for inductive reasoning. Note if g is a tree then $\mathsf{distance}(g,p,g')$ is equivalent to $\mathsf{path}(g,p,g')$. The predicate $\mathsf{tree}\,(g)$ also has an equivalent inductive formulation:

$$\begin{array}{rcl}
\mathsf{disjoint}(x,y) & \equiv & \neg\exists p.(\mathsf{path}(x,p,y)\ \vee\ \mathsf{path}(y,p,x)).\\
\mathsf{tree}\,(g) & \equiv & \exists x.\mathsf{atom}(u,x)\quad \vee\\
& & \exists g_{1,2}.(\mathsf{branch}(u,g_1,g_2)\ \wedge\ \mathsf{disjoint}(g_1,g_2)\ \wedge\\
& & \bigwedge_{i=1,2}(\mathsf{tree}\,(g_i)\wedge\mathsf{disjoint}(g_i,u))
\end{array}$$

which is close to Reynolds' "separation"-based inductive definition [48] (see Section 7 for further comparisons).

As a final preparation, we need a notation for a generation of an unbounded number of fresh references. We extend the notation $\{C\}e\bullet e'=z\{\nu\#x.C'\}$ in §2.3 as follows.

$$\{C\}e\bullet e'=z\{\nu\#\{x\,|\,E(x)\}.C'\} \tag{6.6}$$

which roughly means, with i fresh:

$$\forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e\bullet e'=z\{(\forall x.(E(x)\supset x\#i))\wedge C')\} \tag{6.7}$$

indicating the set $\{x\mid E(x)\}$ of references are newly generated (for the exact semantics, see Appendix C.1). We can now assert for `treeCopy` with the following judgement.

$$\{\mathsf{T}\}\ \texttt{treeCopy}:_u\ \{\mathsf{treecopy}(u)\} \tag{6.8}$$

where we set, with g typed $\mathit{Tree}(\alpha)$:

$$\begin{array}{l}
\mathsf{treecopy}[\alpha](u)\quad =\\
\qquad \forall g^{\mathit{Tree}(\alpha)}.\{\mathsf{tree}\,(g)\}u\bullet g=g'\{\nu\#\{h\,|\,\mathsf{reach}(g',h)\}.\,\mathsf{iso}(g,g')\}@\emptyset
\end{array} \tag{6.9}$$

Above $\mathsf{reach}(g',h)$ stand for $\exists p.\mathsf{path}(g',p,h)$. The assertion $\mathsf{treecopy}[\alpha](u)$ reads:

> *Whenever u is invoked with a tree g of type Tree(α), it creates a tree g' whose reachable nodes are fresh and are isomorphic to those of the original, with no write effects.*

Note α may as well be a higher-order type. Note also the newly generated nodes may share a $\hookrightarrow$-reachable references with the original tree at data when α is higher-order, so that we cannot use $g'\hookrightarrow h$ instead of $\mathsf{reach}(g',h)$ based on the path predicate. As far as its argument is restricted to proper trees, (6.9) is the full specification of `treeCopy`. As such, it entails other assertions the program satisfies. For example it implies the following assertion stating a relative disjointness between two trees, close to [48, § 6]:

$$\mathsf{treesep}[\alpha](u)\quad =\quad \forall x.\{\mathsf{tree}\,(x)\}u\bullet x=y\{\mathsf{iso}(x,y)\wedge\mathsf{disjoint}(x,y)\} \tag{6.10}$$

The derivation of (6.8) can be done in several ways depending on how a recursion is inferred. One of the methods is to use the size (the number of the nodes) of the tree. Another method uses, as in §6.1, the order induced by distance, which we discuss below. Define $x \lesssim_z y$ (all of type *Tree*) as:

$$x \lesssim_z y \quad \equiv \quad \exists p, p'.(\mathsf{distance}(z,p,x) \wedge \mathsf{distance}(y,p',x) \wedge p' \sqsubseteq_{\mathsf{lex}} p \wedge p \neq p') \quad (6.11)$$

Note if x is a proper subtree of y which in turn is a subtree of z then we have $x \lesssim_z y$. Writing $\mathtt{treeCopy}'$ for $\mathtt{treeCopy}$ without the initial recursion, we have:

$$\{\forall g' \lesssim_z g.B'(f,g')\} e\ \mathtt{treeCopy}' :_u \ \{B'(u,g)\} \quad (6.12)$$

where we set:

$$B'(u,g) \ \equiv \ \{\mathsf{tree}\,(z) \ \wedge \ \mathsf{reach}(z,g)\} u \bullet g = g' \{\nu \# \{m | \mathsf{reach}(g',m)\}.\ \mathsf{iso}(g,g')\}$$

For deriving (6.12), we use the inductive reformulation of $\mathsf{tree}\,(g)$ given before, as well as simple entailments such as $(\mathsf{tree}\,(z) \ \wedge \ \mathsf{reach}(z,g)) \supset \mathsf{tree}\,(g)$.

We can then apply the proof rule for recursion (with well-founded ordering, cf. Appendix B) to (6.12) to obtain:

$$\{\mathsf{T}\}\mathtt{treeCopy}' :_u \{\forall g.B'(u,g)\} \quad (6.13)$$

By noting $\mathsf{reach}(g,g) = \mathsf{T}$ we arrive at (6.8).

$$\{\mathsf{T}\}\ \mathtt{treeCopy} :_u \{\{\mathsf{tree}\,(x)\} u \bullet g = g' \{\nu \# \{h \,|\, \mathsf{reach}(g',h)\}.\, \mathsf{iso}(g,g')\} @ \emptyset\} \quad (6.14)$$

Remark. As has been observed, we can also use the ordering based on size of trees:

$$x \lesssim y \quad \equiv \quad \exists n, m.(\mathsf{size}(x,n) \ \wedge \ \mathsf{size}(x,m) \ \wedge \ n \lesssim m)$$

where $\mathsf{size}(x,n)$ says that x is a tree such that the number of its nodes is n (defined by the obvious inductive definition). Using $\lesssim$ instead of $\lesssim_z$, and

$$B''(u,g) \quad \equiv \quad \{\mathsf{tree}\,(g)\} u \bullet g = g' \{\nu \# \{m | \mathsf{reach}(g',m)\}.\ \mathsf{iso}(g,g')\}$$

instead of $B'(u,g)$, we can derive (6.8) directly from the recursion rule. The predicate $\lesssim_z$ has the merit in that it applies to data structures of type *Tree* which are not trees, as we shall discuss in the next subsection.

6.3 Dags and Graphs

When trees become dags, we allow sharing but not circularity.

$$\begin{array}{rcl} \mathsf{isCycle}(g) & \equiv & \exists p.(\mathsf{path}(g,p,g) \wedge p \neq \varepsilon) \\ \mathsf{dag}\,(g) & \equiv & \forall h.(\mathsf{reach}(g,h) \supset \neg \mathsf{isCycle}(h)) \end{array}$$

$\mathsf{dag}\,(g)$ says g is a dag iff it has no circularity. Since $\mathsf{isCycle}(g) \supset \exists p.\mathsf{match}(g,p,p \cdot p)$, we have $\mathsf{tree}\,(g) \supset \mathsf{dag}\,(g)$. As for trees, there is an inductive characterisation:

$$\begin{array}{rcl} \mathsf{dag}\,(g) & \equiv & \exists x.\mathsf{atom}(u,x) \quad \vee \\ & & \exists g_{1,2}.(\ \mathsf{branch}(u,g_1,g_2) \ \wedge \ \bigwedge_{i=1,2} (\mathsf{dag}\,(g_i) \wedge \neg \mathsf{reach}(g_i,u))\ ) \end{array}$$

which says a dag is either an atom or consists of a root with two sub-dags from which there is no upward link to the root node.

A simple extension of `treeCopy` to create a fresh duplicate of an original dag, called `dagCopy`, is given below:

$$\begin{array}{rcl}
\texttt{dagCopy}^{\alpha} & \stackrel{\text{def}}{=} & \lambda g^{\mathit{Tree}(\alpha)}\texttt{let } x = \texttt{ref}(\emptyset) \texttt{ in Main } g \\
\texttt{Main} & \stackrel{\text{def}}{=} & \mu f.\lambda g.\texttt{if dom}(!x,g) \texttt{ then get}(!x,g) \texttt{ else} \\
 & & \quad \texttt{case } !g \texttt{ of} \\
 & & \quad \texttt{in}_1(n) : \texttt{new}(\texttt{inj}_1(n),g) \\
 & & \quad \texttt{in}_2(y_1,y_2) : \texttt{new}(\texttt{inj}_2(\langle fy_1, fy_2\rangle),g) \\
\texttt{new} & \stackrel{\text{def}}{=} & \lambda(y,g).\texttt{let } g' = \texttt{ref}(y) \texttt{ in } (x := \texttt{put}(!x,\langle g,g'\rangle); g')
\end{array}$$

When this program is called with the root of a dag, it first creates an empty table stored in a local variable x. The table remembers those nodes in the original dag which have already been processed, associating them with the corresponding nodes in the fresh dag. Before creating a new node, the program checks if the original node (say g) already exists in the table. If not, a new node (say g') is created, and x now stores the new table which adds a tuple $\langle g,g'\rangle$ to the original.

The program above assumes, for brevity, a pre-defined data type for a table (realisable as, say, lists), with associated procedures: $\texttt{get}(t,g)$ to get the image of g in t; $\texttt{put}(t,\langle g,g'\rangle)$ to add a new tuple when g is not in the domain; $\texttt{dom}(t,g)$ and $\texttt{cod}(t,g)$ to judge if g is in the pre/post-image of t; and the constant $\emptyset$ for the empty table.

The program satisfies

$$\{\mathsf{T}\}\texttt{dagCopy} :_u \{\mathsf{dagcopy}[\alpha](u)\}, \tag{6.15}$$

where $\mathsf{dagcopy}[\alpha](u)$ is given as, with g typed as $\mathit{Tree}(\alpha)$:

$$\forall g^{\mathit{Tree}(\alpha)}.\{\mathsf{dag}\,(x)\}u \bullet g = g'\{\nu\#\{h \mid \mathsf{reach}(g',h)\}.\,\mathsf{iso}(g,g')\}@\emptyset$$

The derivation of (6.15) centres on its treatment of recursion, for which we use the same order $x \lesssim_z y$ as used in the previous subsection, based on lexicographic ordering. Write $\texttt{Main}'$ for `Main` without its initial recursion. Then we have:

$$\{\forall h' \leq_g h.DC(u,h')\}\ \texttt{Main}' :_u\ \{DC(u,h)\} \tag{6.16}$$

where we set:

$$\begin{array}{rcl}
DC(u,h) & \stackrel{\text{def}}{=} & \forall org. \\
 & & \{\mathsf{dag}\,(g)\ \wedge\ \mathsf{reach}(g,h) \wedge !x = org \wedge \mathsf{con}(org)\} \\
 & & \quad u \bullet h = h' \\
 & & \{\nu\{z \mid \mathsf{reach}(h',z) \wedge z \notin \mathsf{cod}(org)\}(\mathsf{con}(!x) \wedge !x = org \cup \langle h,h'\rangle^*)\}\,@\,x
\end{array}$$

which says, noting $\mathsf{dag}\,(g) \wedge \mathsf{reach}(g,h)$ entails $\mathsf{dag}\,(h)$:

> *Suppose h is a dag and x contains a table org which is consistent (i.e. only relates isomorphic nodes). Then invocation of u with h terminates with the*

return value h' and, moreover: (1) references names reachable from h' minus those in the codomain of org are freshly generated; and (2) x stores a table which is consistent and which adds to org the set of co-reachable nodes from $\langle h, h' \rangle$. Further the invocation only modifies x.

$\langle h, h' \rangle^*$ denotes an isomorphism from those nodes reachable from h and those from h'. The set-based ν-notation is understood as the corresponding $\nu\#$-notation (cf. (6.6, 6.7)).

The invariant $\mathsf{con}(t)$ ("table t is consistent") is given by:

$$\begin{array}{lcl} \mathsf{con}(t) & \equiv & \forall g, g'.(\langle g, g' \rangle \in t \supset \mathsf{iso}(g, g')) \wedge \\ & & \forall g_0, g_1.(g_0 \in \mathtt{dom}(t) \wedge \mathsf{reach}(g_0, g_1) \supset g_1 \in \mathtt{dom}(t)) \end{array}$$

$\mathsf{con}(t)$ says t only associates isomorphic graphs, and that its domain (hence co-domain, by isomorphism) is closed under reachability. The derivation of (6.16) uses the inductive characterisation of $\mathsf{dag}(g)$ as well as the above invariant for induction.

From (6.16) we obtain, by $[Rec]$ (cf. Appendix B):

$$\{\mathsf{T}\}\ \mathtt{Main} :_u\ \{\forall h.DC(u,h)\} \tag{6.17}$$

We instantiate h to g to obtain:

$$\{\mathsf{T}\}\ \mathtt{Main} :_u\ \{DC(u,g)\} \tag{6.18}$$

The application rule then give us:

$$\{!x = \emptyset \wedge \mathsf{dag}(g)\}\ (\mathtt{Main}\, g) :_{g'}\ \{\nu\{z \mid \mathsf{reach}(h', z)\}.\ \mathsf{iso}(g, g')\}\ @\, x \tag{6.19}$$

By hiding x:

$$\{\mathsf{dag}(g)\}\ \mathtt{let}\ x = \mathtt{ref}(\emptyset)\ \mathtt{in}\ (\mathtt{Main}\, g) :_{g'}\ \{\nu\#\{z \mid \mathsf{reach}(h', z)\}.\ \mathsf{iso}(g, g')\}\ @\, \emptyset \tag{6.20}$$

where the change from ν to $\nu\#$ is by the following structural rule:

$$[\nu\_to\_\nu\#]\ \frac{\{C\}\ M\ :_u \{\nu x.C'\}\ @\ \emptyset}{\{C\}\ M\ :_u \{\nu\# x.C'\}\ @\ \emptyset}$$

which is valid since if a newly created reference is not stored anywhere, then it cannot be reachable from any initial store, hence any initial datum. Finally by abstraction we arrive at (6.16).

If we further allow a datum of *Tree* to have circular edges, then we have arbitrary graphs. The program `graphCopy` given below operates on just such datum, creating a fresh duplicate of an arbitrary datum of type $Tree(\alpha)$.

$$\begin{array}{rcl} \mathtt{graphCopy}^{\alpha} & \stackrel{\text{def}}{=} & \lambda g^{Tree(\alpha)}.\mathtt{let}\ x = \mathtt{ref}(\emptyset)\ \mathtt{in}\ \mathtt{GMain}\ g \\ \mathtt{GMain} & \stackrel{\text{def}}{=} & \mu f.\lambda g.\mathtt{if}\ \mathtt{dom}(!x, g)\ \mathtt{then}\ \mathtt{get}(!x, g)\ \mathtt{else} \\ & & \quad \mathtt{case}\ !g\ \mathtt{of} \\ & & \qquad \mathtt{in}_1(n) : \mathtt{new}(\mathtt{inj}_1(n), g) \\ & & \qquad \mathtt{in}_2(y_1, y_2) : \\ & & \qquad\quad \mathtt{let}\ g' = \mathtt{new}(\mathtt{tmp}, g) \\ & & \qquad\quad \mathtt{in}\ g' := \mathtt{inj}_2(\langle f y_1, f y_2 \rangle); g' \end{array}$$

where $\mathtt{tmp} = \mathtt{inj}_1(0)$. $\mathtt{graphCopy}^{\alpha}$ is essentially identical with $\mathtt{dagCopy}^{\alpha}$ except when it processes a branch node, say g. Since its subgraphs can have a circular link to g or above, we should first register g and its corresponding fresh node, say g' (the latter with a temporary content), before processing two subgraphs. Registering the pair $\langle g, g' \rangle$ is necessary since two subgraphs may as well refer up to a node nearer to the root (more precisely, this registering becomes necessary when, setting g_0 to be the original root, the minimum path from g_0 to a subgraph, say h, happens to be a prefix of the minimum path from g_0 to g itself). The program satisfies the judgement

$$\{\mathsf{T}\}\mathtt{graphCopy} :_u \{\mathsf{graphcopy}[\alpha](u)\} \tag{6.21}$$

where we set, with g typed as $\mathit{Tree}(\alpha)$:

$$\begin{array}{l} \mathsf{graphcopy}[\alpha](u) \equiv \\ \quad \forall g^{\mathit{Tree}(\alpha)}.\{\mathsf{T}\}u \bullet g = g'\{\nu\#\{h \mid \mathsf{reach}(g', h)\}.\, \mathsf{iso}(g, g')\}@\emptyset \end{array} \tag{6.22}$$

The assertion $\mathsf{graphcopy}[\alpha](u)$ says:

> *When fed with any graph of type $\mathit{Tree}(\alpha)$, u creates its fresh duplicate, and does nothing else.*

This is the simplest of the three assertions for copy algorithms we have seen so far, and is also the strongest. In the following comparisons of assertions, we use, assuming a polymorphic extension of the programming language [40] (type abstracting only values) and that of the logic [19] (treating types syntactically):

$$\mathsf{graphcopyPoly}(u) \quad \equiv \quad \forall \mathrm{X}.\mathsf{graphcopy}[\mathrm{X}](u \diamond \mathrm{X}) \tag{6.23}$$

where $u \diamond \alpha$ (with u having a type of the shape $\forall \mathrm{X}.\beta$) is a term denoting the type application of u to α [19].

Proposition 17. *Fix α. Then each of the following equivalence and implications is valid, with implications strict.*

$$\begin{array}{lcl} \exists w.(\mathsf{graphcopyPoly}(w) \wedge u = w \diamond \alpha) & \equiv & \mathsf{graphcopy}[\alpha](u) \supset \mathsf{dagcopy}[\alpha](u) \\ & \supset & \mathsf{treecopy}[\alpha](u) \supset \mathsf{treesep}[\alpha](u). \end{array}$$

Proof. Direct from the definition. For the first logical equivalence, note the presented graphcopy program already works generically, i.e. duplicates graphs of type $\mathit{Tree}[\alpha]$ for any α, taken as an untyped program. Thus $\Lambda \mathrm{X}.\mathtt{graphCopy}^{\mathrm{X}}$ gives the desired polymorphic program,[4] witnessing $\mathsf{graphcopyPoly}(w)$. □

The derivation of (6.21) refines that of (6.15), especially in induction step. When we apply f to a node in a graph, dagCopy can assume that the table stored in x contains pairs of isomorphic nodes. This is no longer so for graphCopy since it now contains

[4] If we are to use the implicit (a la Curry) typing for programs as in [5], we would assert for the untyped version of graphCopy simply by $\forall \mathrm{X}.\mathsf{graphcopy}[\mathrm{X}](u)$, in which case we have the (strict) implication $\forall \mathrm{X}.\mathsf{graphcopy}[\mathrm{X}](u) \supset \mathsf{graphcopy}[\alpha](u)$.

fresh nodes with temporary content. Thus we need to identify the portion of the table which contain the pairs whose codomain have already been processed. For this purpose we use the following predicates.

$$\begin{array}{lcl} \mathsf{below}(g_1,g_2,z) & \equiv & \exists p_{1,2},q.(\bigwedge_{i=1,2}\mathsf{distance}(z,p_i,g_i)\ \wedge\ p_2 = p_1\cdot q\ \wedge\ p_1\neq p_2) \\ \mathsf{above}(g_1,g_2,z) & \equiv & \mathsf{below}(g_2,g_1,z) \\ \mathsf{before}(g_1,g_2,z) & \equiv & g_1 \lesssim_z g_2\ \wedge\ \neg\mathsf{below}(g_1,g_2,z) \end{array}$$

Thus $\mathsf{below}(g_1,g_2,z)$, or equivalently $\mathsf{above}(g_2,g_1,z)$, asserts that, starting from z, the minimum path to g_1 goes via g_2 (note that, if we look the graph with the root at the top, this does visually mean g_1 is below g_2). Using these predicates, the content of the table before and after processing a node, say g, is described as, with z being the root:

(before processing g) Each node h such that $\mathsf{before}(h,g,z)$ has already been processed, i.e. the paths between these nodes in the domain are faithfully reproduced in the corresponding nodes in the codomain. The table also contains those nodes *above* g (i.e. h such that $\mathsf{above}(h,g,z)$) in their domain.

(after processing g) The table inherits the same pairs of nodes without changing them, and, in addition, each node h below g with respect to the root, i.e. h such that $\mathsf{below}(h,g,z)$, has now been processed and added to the table, together with g itself whose corresponding fresh node is g'. Note this automatically entails that no change in the table content takes place if g is already in the domain of the table.

Note that these properties are "chained" through the two invocations of f in the program, hence through the main invocation. Respectively writing the first and second conditions $\mathsf{conPre}(t,g,z)$ and $\mathsf{conPost}(t,g,g',z)$ with table t and root z, (whose definitions using path predicates are easy), we arrive at the following assertion for induction:

$$\begin{array}{lcl} GC(f) & \equiv & \forall g.\{\mathsf{conPre}(!x,g,z)\} \\ & & f\bullet g = g' \\ & & \{\ \nu\{l \mid \mathsf{below}(l,g',z)\ \wedge\ l\notin \mathsf{dom}(org)\}.\ \mathsf{conPost}(!x,g,g',z)\ )\ )\}@x \end{array}$$

Note f still touches only x (except creating new names) since those temporary new nodes above g are not modified when g is processed.

Using this predicate, we can reach the following judgement for the main procedure before we apply recursion, writing $\texttt{GMain}'$ for the result of taking off the initial recursion from GMain.

$$\{\forall h' \lesssim_g h.GC(f,h')\}\ \texttt{GMain}'\ :_u\ \{GC(u,h)\} \tag{6.24}$$

Note each subgraph of h is either $\lesssim_g$-smaller than h; or, if not, it is above h hence, by $\mathsf{conPre}(!x,h,g)$, is in the table. Thus the inductive hypothesis $\forall h' \lesssim_g h.GC(f,h')$ is enough for inferring for the two invocations of f. From (6.24) we obtain, by the rule for recursion [*Rec*] (cf. Appendix B):

$$\{\mathsf{T}\}\ \texttt{GMain}\ :_u\ \{\forall h.GC(u,h)\} \tag{6.25}$$

The rest is as for dagCopy, first inferring:

$$\{!x = \emptyset\}\ (\texttt{GMain}\ g) :_u \{\ \nu\{l \mid \mathsf{below}(l, g', z)\}.\ \mathsf{conPost}(!x, g, g', g)\}\ @\ x \qquad (6.26)$$

Since all nodes reachable from g are also below g or g itself, $\mathsf{conPost}(!x, g, g)$ means the table only contains isomorphic nodes. In particular it entails $\mathsf{iso}(g, g')$. Thus we obtain, from (6.26) by way of the consequence rule:

$$\{!x = \emptyset\}\ (\texttt{GMain}\ g) :_u \{\ \nu\{l \mid \mathsf{below}(l, g', z)\}.\ \mathsf{iso}(g, g')\}\ @\ x \qquad (6.27)$$

From (6.27) we obtain, as we have reasoned for dagCopy:

$$\{\mathsf{T}\}\ \texttt{let}\ x = \texttt{ref}(\emptyset)\ \texttt{in}\ (\texttt{GMain}\ g) :_u \{\ \nu\#\{l \mid \mathsf{below}(l, g', z)\}.\ \mathsf{iso}(g, g')\}\ @\ \emptyset \quad (6.28)$$

By applying the rule for abstraction, to (6.28), we finally arrive at (6.21).

7 Related Work and Conclusion

This paper proposed a Hoare-like program logic for imperative higher-order functions with dynamic reference generation, a core part of ML-like languages [3, 4]. Target programming languages of our preceding logics [6, 17, 19, 20] do not include local state. As is well-known [21, 27, 28, 31, 42, 44], local state in higher-order functions radically adds semantic complexity. To our knowledge, the present work proposed the first Hoare-like program logic for this class of languages: nor do we know the preceding Hoare-like logics which can assert and verify the demonstrated reasoning examples. In the following we discuss related works and conclude with further topics.

7.1 Related Works

Reasoning Principles for Functions with Local State. There are many studies of equivalences over higher-order programs with local state. An early work by Meyer and Sieber [31] presents many interesting examples and reasoning principles based on denotational semantics. Mason and Talcott [27, 28] give a series of detailed studies on equational axioms for an untyped version of the language treated in the present paper, including those involving local invariants. Pitts and Stark [42, 44, 50] present powerful operationally-based reasoning principles for the same language as the present work treats, with the reasoning principle for local invariants for higher-order types [44]. Sumii and Pierce [51] present a fully abstract bisimulation technique for equational reasoning on higher-order functions with dynamic sealing and type abstraction. Their bisimulations are parameterised by related seals, which are close to parameterisation by related stores in Pitts-Stark's principle. Building on [51], Koutavas and Wand [21] propose a fully abstract bisimulation technique for the untyped version of the language we treat, and apply the techniques for reasoning about several non-trivial programs with local store. They use denotational technique in relaxing a condition for bisimulations.

Our axioms for information hiding in § 4, which capture the basic patterns of programming with local state, are closely related with these reasoning principles. The

proposed logic differs in that its aim is to offer a method for describing and validating diverse properties of programs beyond program equivalence, represented as logical assertions. The equivalence-based approach for program validation and the assertion-based one are complimentary, to which Theorem 6 would offer a basis of integrated usage. 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'. Such a link can be further substantiated through a mechanised logic for semantics of higher-order behaviour along the line of Longley and Pollack's recent work [25].

Hoare Logics (1): Local Variables and ML-like Languages. To our knowledge, Hoare and Wirth [16] is the first to present a rule for local variable declaration (given for Pascal). In our notation, a version of their rule may be written as follows.

$$[\mathit{Hoare\text{-}Wirth}]\ \frac{\{C \wedge x \neq \tilde{y}\}\, P\, \{C'\}\ \ x \notin \mathsf{fv}(C') \cup \{\tilde{y}\}}{\{C[e/!x]\}\ \mathtt{new}\ x := e\ \mathtt{in}\ P\ \{C'\}}$$

Because this rule assumes references are never exported outside of their original scope, there is no need to have x in C'. Since aliasing is not permitted in [16] either, we can further dispense with $x \neq \tilde{y}$ in the premise. [*LetRef*] in § 5.1 differs from this rule in that it can treat new references generation exported beyond their original scope; aliased references; and higher-order procedures (both as programs and as stored values). We can check [*Hoare-Wirth*] is derivable from [*LetRef*] and [*Assign*].

Among the studies on verification methods for ML-like languages [4, 35], *Extended ML* [49] is a formal development framework for Standard ML. A specification is given by combining a module' signature and algebraic axioms on them. Correctness of an implementation w.r.t. a specification is verified by incremental syntactic transformations. *Larch/ML* [52] is a design proposal of a Larch-based interface language for ML. Integration of typing and interface specification is the main focus of the proposal in [52]. These two works do not (aim to) offer a program logic with compositional proof rules; nor do either of these works treat specifications for functions with dynamically generated references.

Hoare Logics (2): Reachability. A seminal work by Nelson [37] first presented the use of reachability predicates for reasoning about linked lists. Based on [37], Lahiri and Qadeer [23] study a tractable axiomatisation of cyclic lists and apply the resulting axiomatisation to the development of a VC generator/checker for a first-order procedural language. The key idea in their axiomatisation is to identify a head cell (or cells) of a cycle and use it for a straightforward inductive definition of reachability and associated invariant. For example, an invariant for the example program in §6.1 (which is from [23]) can be written as follows:

$$I(x,h) \ \ = \ \ B(x,h) \wedge \forall g.(R(h,g) \supset ((x \neq h \wedge R(x,g)) \vee \mathsf{iszero}(x)))$$

where $B(x,h)$ says x reaches (is blocked by) a head h; $R(x,y)$ says we can reach y from x; and $\mathsf{iszero}(x)$ says the datum in the cons x is zero. Thus $I(x,h)$ says x reaches a head h; and all cells starting from h reaching x are zeroed. We can then show $I(x,z)$ is an

invariant of the body command of csbody. This can be used for validating cyclesimple zeroes all fields in a cyclic list w.r.t. partial correctness.

As noted, the interest and significance of their method lies in simple inductive axiomatisations amenable to mechanical validation. Assertions and reasoning for higher-order behaviour with dynamic reference generation, including a general class of data structures and their dynamic generation, are not among their concerns and are not considered in their work.

Yorsh et al. [53] introduce a logic for expressing invariance for pointer manipulating programs with dynamic allocation. Their logic is undecidable, but contains a decidable sublogic. In our logic, reachability is undecidable, but useful axioms are proposed. They supply neither proof rules nor axioms, so a direct comparison is difficult.

An interesting question is whether we can apply their ideas on effective axiomatisation or mechanisation to a large class of mutable data structures treatable in our method.

Hoare Logics (3): Separation Reynolds, O'Hearn and others [9, 39, 48] study a reasoning method for dynamically generated and deallocated mutable data structures using a spacial conjunction, $C * C'$. Taking the tree copy in § 6.2 (which is from [48]), they start from a predicate $r \mapsto x$ which is roughly equivalent to $alloc(r) \wedge\ !r = x$ in our notation, with $alloc(r)$ indicating a reference r is allocated. To compare with their logic, consider τ which is the *structural description of a tree*: for example, $\tau = ((1,(2,3)))$ indicates a tree whose leaves store $1,2,3$ from left to right. Then $Tree(\tau)(u)$ asserts allocation of a τ-tree with the root u, in the way:

$$Tree((1,(2,3)))(u) \;=\; \exists xy.(u \mapsto xy * x \mapsto 1 * Tree((2,3)))(y)$$

where $C_1 * C_2$ indicates the conjunction of $C_{1,2}$ together with all the $alloc$-declared references of C_1 and those of C_2 are disjoint. We can then prove, writing $\mathtt{treeCopyImp}(x,y)$ for an imperative version of treeCopy which stores the result of copy in y:

$$\{Tree(\tau)(x)\}\,\mathtt{treeCopyImp}(x,y)\,\{Tree(\tau)(x) * Tree(\tau)(y)\} \tag{7.1}$$

In comparison with the proposed logic, we observe:

(1) The use of $*$ demands all concerned references are explicitly declared in assertions, made possible by the use of structural description (τ of $Tree(\tau)(u)$ in (7.1) above). The shape of the description usable for reasoning becomes complex [9] when data structures involve non-trivial sharing (as in dags and graphs). In contrast, § 6 has shown that our approach not only dispenses with the need for structural description but also allows concise and uniform assertions and reasoning for data structures with different degrees of sharing.

(2) As in (7.1), Reynolds's approach represents fresh data generation by relative spatial disjointness from the original datum, using the separating conjunction. This method does capture a significant part of the program's properties. The proposed logic represents freshness as temporal disjointness through the generic (un)reachability from arbitrary datum in the initial state. Proposition 17 demonstrates that this approach leads to strictly stronger (more informative) assertion, from which the assertion equivalent to the other approach can be derived.

(3) The presented approach enables uniform treatment of known data types in verification, including product, sum, reference, list, tree, closure, etc., through the use of anchors. This is a simple and general method which allows us to assert and compositionally verify trees, graphs, dags, lists, stored procedures, higher-order functions with local state and other data types on a uniform basis, with precise match with observational semantics.

See [6] for further comparisons. Reynolds [48] criticises the use of reachability for describing data structure, taking the in-place reversal of a linear list as an example. Following the method in Section 6, a tractable reasoning is possible for such an example using reachability combined with [*Inv*].

Birkedal et al. [8] present a "separation logic typing" for a variant of Idealised Algol where types are constructed from formulae of disjunction-free separation logic. The typing system uses the subtyping calculated via categorical semantics, on which their study focusses. In [7], they extend the original separation logic with higher-order predicates, and demonstrate how the extension helps modular reasoning on priority queues. Both of these works treat neither exportable fresh reference generation nor higher-order/stored procedures in full generality, so that it would be difficult to assert and validate examples treated in § 5 and § 6. It is an interesting future topic to examine the use of higher-order predicate abstraction in the present logic.

Other Hoare Logics. Nanevski et al [36] 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. Their system emphasises the clean separation between static validation and assertions. Local store is not treated and left as an open problem in [36]. Reus and Streicher [46] present a Hoare logic for a simple language with higher-order stored procedures, extended in [45], with primitives for the dynamic allocation and deallocation of references. Soundness is proved with denotational methods, but completeness is not proved. 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. Similarly with [36], but based on a monadic presentation, Krishnaswami [22] studies a specification logic for a high-level language which explicitly state effects and their order of execution. His logic does not handle recursive types or polymorphism. Completeness and invariance rules/axioms are not considered. In our case, aliasing is handled by content qualification in assertions, not at the programming language level. Content quantification enables formulating local invariance axioms. Completeness was not proved in this work.

No work mentioned in this section studies local invariance.

Meta-logical Study on Freshness. Freshness of names is recently studied from the viewpoint of formalising binding relations in programming languages and computational calculi. Pitts and Gabbay [13, 43] extend First-Order Logic with constructs to reason about freshness of names based on the theory of permutations. The key syntactic additions are the (interdefinable) "fresh" quantifier N and the freshness predicate #,

mediated by the swapping (finite permutation) predicate. Miller and Tiu [32] are motivated by the significance of generic (or eigen-) variables and quantifiers at the level of both formulae and sequents, and splits 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 well 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 (the semantic notion of freely occurring locations) of a datum/program. A characterisation of the support by the swapping operation may lead to deeper understanding of axiomatisations of reachability.

There are mechanisation of Hoare logics in higher-order logics, including [11, 29, 38]. While these works do discuss some aspects of imperative programs the proposed logic treats (such as pointer-based data structures), none so far may offer a general assertion method and compositional proof rules for ML-like reference generation or their combination with higher-order functions.

7.2 Further Topics

The present work is intended to be but a modest initial step in logically capturing the richness of the universe of behaviours of higher-order functions with local state. Many challenges remain before we reach a mature engineering basis for using the logical method studied in this paper. Some of the significant future topics include: *Further development of reasoning principles as axioms, including those on local invariants (are there a basic set of axioms capturing most of the reasoning principles?); Partial correctness logic; Coverage of the whole of SML/CAML; Extensions of the proposed method to higher-order languages with monadic encapsulation of imperative features such as Haskell and untyped higher-order languages such as Scheme (we strongly believe both are feasible and rewarding); Exploration of effective reasoning/validation methods for general mutable data structure, including semi-automatic verification;* and *integration with program development method.*

Acknowledgement We thank Andrew Pitts for his comments on an early version of this paper. The example of the mutual recursion in § 5 was given by Bernhard Reus. We thank him for his e-mail discussions on this example.

References

1. Flint project. http://flint.cs.yale.edu/flint/.
2. A full version of this paper. http://www.doc.ic.ac.uk/˜yoshida/local.
3. Standard ML home page. http://www.smlnj.org.
4. The Caml home page. http://caml.inria.fr.
5. H. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.
6. M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *ICFP'05*, pages 280–293, 2005. Full version is available at: www.dcs.qmul.ac.uk/˜kohei/logics.

7. B. Biering, L. Birkedal, and N. Torp-Smith. Bi hyperdoctrines and higher-order separation logic. In *ESOP'05*, LNCS, pages 233–247, 2005.
8. 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.
9. R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *Workshop SPACE*, 2004.
10. K. Bruce, L. Cardelli, and B. Pierce. Comparing object encodings. *Infomation and Computation*, 155(1/2):108–133, 1999.
11. J.-C. Filliatre. Verification of non-functional programs using interpretations in type theory. *JFP*, 13(4):709–745, 2003.
12. R. W. Floyd. Assigning meaning to programs. In *Symp. in Applied Mathematics*, volume 19, 1967.
13. M. Gabbay and A. Pitts. A New Approach to Abstract Syntax Involving Binders. In *Proc. LICS '99*, pages 214–224, 1999.
14. C. A. R. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
15. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
16. C. A. R. Hoare and N. Wirth. Axiomatic semantics of Pascal. *ACM TOPLAS*, 1(2):226–244, 1979.
17. K. Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.
18. 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.
19. K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202. ACM, 2004.
20. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*, pages 270–279, 2005. Full version is is available at: www.dcs.qmul.ac.uk/˜kohei/logics.
21. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. POPL*, 2006.
22. N. Krishnaswami. Separation logic for a higher-order typed language. In *Workshop Space06*, pages 73–82, 2006.
23. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, pages 115–126. ACM, 2006.
24. P. Landin. A correspondence between algol 60 and church's lambda-notation. *Comm. ACM*, 8:2, 1965.
25. J. Longley and R. Pollack. Reasoning about cbv functional programs in isabelle/hol. In *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2004.
26. N. Y. Martin Berger and K. Honda. Three completeness results for higher-order fucntions with alias and local state. http://www.doc.ic.ac.uk/˜yoshida/local.
27. I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theor. Comput. Sci.*, 105(2):167–215, 1992.
28. I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *LICS*, pages 186–197, 1992.
29. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 159:200–227, May 2005.
30. E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
31. A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL'88*, 1988.
32. D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, 6(4):749–783, 2005.

33. R. Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971.
34. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1):1–77, 1992.
35. R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML*. MIT Press, 1990.
36. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP06*, pages 62–73. ACM Press, 2006.
37. G. Nelson. Verifying reachability invariants of linked structures. In *POPL '83*, pages 38–47. ACM Press, 1983.
38. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, 2006.
39. P. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, 2004.
40. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
41. B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for objecr-oriented programming. *JFP*, 4(2):207–247, 1993.
42. 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. Reprinted from LICS'06.
43. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
44. 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.
45. B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *Proc. CSL*, volume 4207, pages 575–590, 2006.
46. B. Reus and T. Streicher. About Hoare logics for higher-order store. In *ICALP*, volume 3580, pages 1337–1348, 2005.
47. J. C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
48. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, 2002.
49. D. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *POPL'85*, pages 67–77. ACM, 1985.
50. I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, Dec. 1994.
51. E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *POPL'04*, pages 161–172. ACM Press, 2004.
52. J. Wing, E. Rollins, and A. Zaremski. Thoughts on a Larch/ML and a new Application for LP. In *First International Workshop on Larch, Dedham 1992*, pages 297–312. Springer-Verlag, 1992.
53. G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *FoSSaCS*, volume 3921, pages 94–110, 2006.

A Appendix: Reductions and Typing Rules

A.1 Reductions

A *reduction relation*, or often *reduction* for short, is a binary relation between configurations, written

$$(\nu\tilde{l})(M,\sigma_1) \longrightarrow (\nu\tilde{l'})(N,\sigma_2)$$

The relation is generated by the following rules. First, we have the standard rules for the call-by-value PCF:

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

The induced reduction becomes that for open configurations (hence for configurations with empty binder) by stipulating:

$$\frac{M \longrightarrow M'}{(M,\sigma) \longrightarrow (M',\sigma)}$$

Then we have the reduction rules for imperative constructs, i.e. assignment, dereference and new-name generation.

$$\begin{array}{rcll}(!l,\ \sigma) & \to & (\sigma(l),\ \sigma) & \\ (l := V,\ \sigma) & \to & ((),\ \sigma[l \mapsto V]) & \\ (\texttt{ref}(V),\ \sigma) & \to & (\nu\, l)(l,\ \sigma \uplus [l \mapsto V]) & \\ (\texttt{new } x := V \texttt{ in } N, \sigma) & \longrightarrow & (\nu\, l)(N[l/x], \sigma \uplus [l \mapsto V]) & (l \text{ fresh})\end{array}$$

Finally we close $\longrightarrow$ under evaluation contexts and ν-binders.

$$\frac{(\nu\tilde{l_1})(M,\sigma) \to (\nu\tilde{l_2})(M',\sigma')}{(\nu\tilde{l}\tilde{l_1})(\mathcal{E}[M],\sigma) \to (\nu\tilde{l}\tilde{l_2})(\mathcal{E}[M'],\sigma')}$$

where $\tilde{l}$ are disjoint from both $\tilde{l_1}$ and $\tilde{l_2}$, $\mathcal{E}[\cdot]$ is the left-to-right evaluation context (with eager evaluation), inductively given by:

$$\begin{array}{rcl}\mathcal{E}[\cdot] & ::= & (\mathcal{E}[\cdot]M) \mid (V\mathcal{E}[\cdot]) \mid \langle V,\mathcal{E}[\cdot]\rangle \mid \langle \mathcal{E}[\cdot],V\rangle \mid \pi_i(\mathcal{E}[\cdot]) \mid \texttt{in}_i(\mathcal{E}[\cdot]) \\ & \mid & \texttt{op}(\tilde{V},\mathcal{E}[\cdot],\tilde{M}) \mid \texttt{if } \mathcal{E}[\cdot] \texttt{ then } M \texttt{ else } N \mid \texttt{case } \mathcal{E}[\cdot] \texttt{ of } \{\texttt{in}_i(x_i).M_i\}_{i\in\{1,2\}} \\ & \mid & !\mathcal{E}[\cdot] \mid \mathcal{E}[\cdot] := M \mid V := \mathcal{E}[\cdot] \mid \texttt{ref}(\mathcal{E}[\cdot]) \mid \texttt{new } x := \mathcal{E}[\cdot] \texttt{ in } M\end{array}$$

A.2 Typing Rules

The typing rules are standard [40], which we list in Figure 1 for reference (from first-order operations we only list two basic ones). In the first rule of Figure 1, $\texttt{c}^C$ indicates a constant $\texttt{c}$ has a base type C. We also use the typing sequent of the form: $\Theta \vdash M : \alpha$ where Θ mixes these two kinds of maps.

Fig. 1 Typing Rules

$$[Var]\ \frac{}{\Theta, x:\alpha \vdash x:\alpha} \quad [Label]\ \frac{}{\Gamma;\Delta\cdot l:\alpha \vdash l:\alpha} \quad [Constant]\ \frac{}{\Gamma;\Delta \vdash \mathsf{c}^C : C}$$

$$[Add]\ \frac{\Gamma;\Delta \vdash M_{1,2} : \mathsf{Nat}}{\Gamma;\Delta \vdash M_1 + M_2 : \mathsf{Nat}} \quad [Eq]\ \frac{\Gamma;\Delta \vdash M_{1,2} : \mathsf{Nat}}{\Gamma;\Delta \vdash M_1 = M_2 : \mathsf{Bool}}$$

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

$$[Abs]\ \frac{\Theta, x:\alpha \vdash M:\beta}{\Theta \vdash \lambda x^{\alpha}.M : \alpha \Rightarrow \beta} \quad [App]\ \frac{\Gamma;\Delta \vdash M : \alpha \Rightarrow \beta \quad \Gamma;\Delta \vdash N : \alpha}{\Gamma;\Delta \vdash MN : \beta}$$

$$[Rec]\ \frac{\Gamma, x:\alpha\Rightarrow\beta\,;\Delta \vdash \lambda y^{\alpha}.M : \alpha\Rightarrow\beta}{\Gamma;\Delta \vdash \mu x^{\alpha\Rightarrow\beta}.\lambda y^{\alpha}.M : \alpha\Rightarrow\beta} \qquad [Iso]\ \frac{\Theta \vdash M:\alpha \quad \alpha \approx \beta}{\Theta \vdash M : \beta}$$

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

$$[Ref]\ \frac{\Gamma;\Delta \vdash V : \alpha}{\Gamma;\Delta \vdash \mathtt{ref}(V) : \mathsf{Ref}(\alpha)} \quad [New]\ \frac{\Gamma;\Delta \vdash M : \alpha \quad \Gamma;\Delta, x:\mathsf{Ref}(\alpha) \vdash N : \beta}{\Gamma;\Delta \vdash \mathtt{new}\ x\ :=M\ \mathtt{in}\ N : \beta}$$

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

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

A.3 Observational Congruence

Define:

$$(\nu\tilde{l})(M,\sigma) \Downarrow (\nu\tilde{l'})(V,\sigma') \quad \stackrel{\text{def}}{\equiv} \quad (\nu\tilde{l})(M,\sigma) \to^* (\nu\tilde{l'})(V,\sigma')$$

Further set:

$$(\nu\tilde{l})(M,\sigma) \Downarrow \quad \stackrel{\text{def}}{\equiv} \quad (\nu\tilde{l})(M,\sigma) \Downarrow (\nu\tilde{l'})(V,\sigma') \quad \text{for some } (\nu\tilde{l'})(V,\sigma').$$

Assume $\Gamma;\Delta,\tilde{l}_{1,2} : \tilde{\alpha}_{1,2} \vdash M_{1,2} : \alpha$. Then we write

$$\Gamma;\Delta \vdash (\nu\tilde{l_1})(M_1,\sigma_1) \cong (\nu\tilde{l_2})(M_2,\sigma_2)$$

if, for each typed context $C[\,\cdot\,]$ which produces a closed program which is typed as Unit under Δ and in which no labels from $\tilde{l}_{1,2}$ occur, the following holds:

$$(\nu\tilde{l_1})(C[M_1],\ \sigma_1) \Downarrow \quad \text{iff} \quad (\nu\tilde{l_2})(C[M_2],\ \sigma_2) \Downarrow$$

which we often write $(\nu\tilde{l_1})(M_1,\sigma_1) \cong (\nu\tilde{l_2})(M_2,\sigma_2)$ leaving type information implicit. We also write $\Gamma;\Delta \vdash M_1 \cong M_2$, or simply $M_1 \cong M_2$ leaving type information implicit, if, $\tilde{l_i} = \sigma_i = \emptyset$ $(i=1,2)$.

B Appendix: Proof Rules

Fig. 2 Proof Rules

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

$$[In_1]\ \frac{\{C\}\ M :_v \{C'[\mathsf{inj}_1(v)/u]\}}{\{C\}\ \mathtt{inj}_1(M) :_u \{C'\}} \qquad [Proj_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

$$[Case]\ \frac{\{C^{\frown\tilde{x}}\}\ M :_m \{C_0^{\frown\tilde{x}}\} \quad \{C_0[\mathsf{inj}_i(x_i)/m]\}\ M_i :_u \{C'^{\frown\tilde{x}}\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

$$[Add]\ \frac{\{C\}\,M_1 :_{m_1} \{C_0\} \quad \{C_0\}\,M_2 :_{m_2} \{C'[m_1+m_2/u]\}}{\{C\}\,M_1+M_2 :_u \{C'\}}$$

$$[Abs]\ \frac{\{C\wedge A^{\frown x\tilde{i}}\}\ M :_m \{C'\}}{\{A\}\ \lambda x.M :_u \{\forall x\tilde{i}.\{C\}\,u\bullet x=m\,\{C'\}\}}$$

$$[App]\ \frac{\{C\}\ M :_m \{C_0\} \quad \{C_0\}\ N :_n \{\ C_1\wedge\ \{C_1\}\ m\bullet n=u\ \{C'\}\ \}}{\{C\}\ MN :_u \{C'\}}$$

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

$$[Pair]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\} \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1,m_2\rangle/u]\}}{\{C\}\ \langle M_1,M_2\rangle :_u \{C'\}}$$

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

$$[Rec]\ \frac{\{A^{\frown xi}\wedge\forall j\lneq i.B(j)[x/u]\}\ \lambda y.M :_u \{B(i)^{\frown x}\}}{\{A\}\ \mu x.\lambda y.M :_u \{\forall i.B(i)\}} \qquad [Ref]\ \frac{\{C\}\ M :_m \{C'\}}{\{C\}\ \mathtt{ref}(M) :_u \{\#u.C'[!u/m]\}}$$

$$[Conseq]\ \frac{C\supset C_0 \quad \{C_0\}\ M :_u \{C_0'\} \quad C_0'\supset C'}{\{C\}\ M :_u \{C'\}}$$

Figure 2 presents all compositional proof rules (at the end we briefly discuss structural rules). 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, $C^{\frown\tilde{x}}$ indicates $\mathsf{fv}(C)\cap\{\tilde{x}\}=\emptyset$. Symbols $i,j,\ldots$ range over auxiliary names.

In [*Abs,Rec*], A,B denote *stateless* formulae, in the sense of Definition 13.

In [*Rec*], $\lneq$ in the precondition of the premise can be replaced by (or interpreted as) an arbitrary well-founded, and possibly partial, order on closed values of some type [12]. In this case, the universal abstraction on i follows the rule $[Aux_\forall]$ discussed later. Including this point, [*Rec*] is best considered as being derived from the following rule for recursion:

$$[Rec\text{-}Ren]\ \frac{\{A^{\frown x}\}\ \lambda y.M :_u \{B\}}{\{A^{\frown x}\}\ \mu x.\lambda y.M :_u \{B[u/x]\}}$$

and the following rule for moving a stateless formula from the precondition to the post-condition [6, §7.3]

$$[\wedge\text{-}\supset]\ \frac{\{A \wedge B\}\ V :_u \{C\}}{\{A\}\ V :_u \{B \supset C\}},$$

combined with an induction principle at the level of assertions. The use of less general [*Rec*] still gives a useful articulation in inferences.

[*Assign*] uses *logical substitution* which uses content quantification to represent a substitution of content of a possibly aliased reference [6].

$$C\{\!|e_2/!e_1|\!\} \stackrel{\text{def}}{=} \forall m.(m = e_2 \supset [!e_1](!e_1 = m \supset C)).$$

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. The proof rules for the located judgement is given just as [6], adding the following rule for the reference.

$$[\mathit{Ref}]\ \frac{\{C\}\ M :_m \{C'\}@\tilde{e}x \quad x \notin \mathsf{fpn}(\tilde{e}) \cup \mathsf{fv}(\tilde{e})}{\{C\}\ \mathtt{ref}(M) :_u \{\#x.C'\}@\tilde{e}}$$

For the structural rules (i.e. those proof rules which only manipulate assertions in pre/-post conditions), the structural rules given in [6, §7.3] for the base logic stay valid except that the universal abstraction rule $[\mathit{Aux}_\forall]$ in [6, §7.3] needs be weakened:

$$[\mathit{Aux}_\forall\text{-}\mathit{Val}]\ \frac{\{C\}\ V :_u \{C'\} \quad i \notin \mathsf{fv}(C) \cup \mathsf{fv}(V)}{\{C\}\ V :_u \{\forall i.C'\}}$$

The restriction to values can be taken off if we restrict the type of i:

$$[\mathit{Aux}_\forall]\ \frac{\{C\}\ M :_u \{C'\}@\tilde{e} \quad i \notin \mathsf{fv}(C) \cup \mathsf{fv}(M) \cup \mathsf{fv}(\tilde{e}) \quad i \text{ is of a base type}}{\{C\}\ M :_u \{\forall i.C'\}@\tilde{e}}$$

where the second side condition (i is of a base type) is not present in [6].

We observe the original structural rule, which does not have this condition, is not valid in the presence of new reference generation. For example we can take:

$$\{\mathsf{T}\}\ \mathtt{ref}(3) :_u \{u\#i \wedge !u = 3\}@\emptyset \tag{B.1}$$

which is surely valid. But without the side condition, we can infer the following from (B.1).

$$\{\mathsf{T}\}\ \mathtt{ref}(3) :_u \{\forall i.(u\#i \wedge !u = 3)\}@\emptyset$$

which does not make sense (just substitute u for i). This is because of a new name generation for which i cannot range over: such an interplay with new name generation is not possible if the target program is a value, or if i is of a base type.

C Appendix: Models

C.1 Models and Satisfaction

The semantics of the assertions follows. All omitted cases are by de Morgan duality.

1. $\mathcal{M} \models e_1 = e_2$ if $\mathcal{M}[u : e_1] \approx \mathcal{M}[u : e_2]$.
2. $\mathcal{M} \models C_1 \wedge C_2$ if $\mathcal{M} \models C_1$ and $\mathcal{M} \models C_2$.
3. $\mathcal{M} \models \neg C$ if not $\mathcal{M} \models C$.
4. $\mathcal{M} \models \forall x.C$ if $\forall \mathcal{M}'.(\mathcal{M}[x:N] \Downarrow \mathcal{M}' \ \wedge \ \mathcal{M} \approx \mathcal{M}'/x \supset \mathcal{M}' \models C)$
5. $\mathcal{M} \models \bar{\forall} x.C$ if $\forall \mathcal{M}', l.((\nu l)(\mathcal{M}'/x) \approx \mathcal{M} \ \wedge \ \mathcal{M}'(x) = l \supset \mathcal{M}' \models C)$
6. $\mathcal{M} \models \forall \mathrm{X}.C$ if for all closed type α, $\mathcal{M} \cdot \mathrm{X} : \alpha \models C$.
7. $\mathcal{M} \models [!x]C$ if $\forall \mathcal{M}'.((\mathcal{M}[u:N] \Downarrow \mathcal{M}' \ \wedge \ \forall V.\mathcal{M}[x \mapsto V] \approx (\mathcal{M}'/u)[x \mapsto V]) \supset \mathcal{M}' \models C)$.
8. $\mathcal{M} \models e_1 \hookrightarrow e_2$ if for each $(\nu \tilde{l})(\xi, \sigma) \approx \mathcal{M}$, $[\![e_2]\!]_{\xi,\sigma} \in \mathsf{lc}(\mathsf{fl}([\![e_1]\!]_{\xi,\sigma}), \sigma)$.
9. $\mathcal{M} \models \{C\} x \bullet y = z \{C'\}$ if $(\mathcal{M}[u:N] \Downarrow \mathcal{M}_0 \ \wedge \ \mathcal{M}_0 \models C) \supset (\mathcal{M}_0[z : xy] \Downarrow \mathcal{M}' \ \wedge \ \mathcal{M}' \models C')$.
10. $\mathcal{M} \models \{C\} x \bullet y = z \{C'\} @ \tilde{w}$ is defined as $(\mathcal{M}[u:N] \Downarrow \mathcal{M}_0 \ \wedge \ \mathcal{M}_0 \models C) \supset (\mathcal{M}_0[z : xy] \Downarrow \mathcal{M}' \ \wedge \ \mathcal{M}' \models C' \wedge \mathcal{M}'[\tilde{w} \mapsto \tilde{V}] \approx \mathcal{M}_0[\tilde{w} \mapsto \tilde{V}])$, where the last condition means at most $\tilde{w}$ are updated.
11. $\mathcal{M} \models \{C\} e \bullet e' = x \{C'\} @ \{z | E(z)\}$ iff it satisfies the defining clause of (9) above as well as the following, letting $\mathcal{M}_0 \stackrel{\text{def}}{=} (\nu \tilde{l})(\xi, \sigma_0)$ and $\mathcal{M}' \ \approx \ (\nu \tilde{l} \tilde{l}')(\xi, \sigma')$:

$$\forall \tilde{V}.((\nu \tilde{l})(\xi, \sigma_0[\tilde{l}_1 \mapsto \tilde{V}]) \approx (\nu \tilde{l} \tilde{l}')(\xi, \sigma'[\tilde{l}_1 \mapsto \tilde{V}])) \tag{C.1}$$

where $l \in \{\tilde{l}_1\}$ iff $(\nu \tilde{l})(\xi \cdot z : l, \sigma_0) \models E$.

In the defining clauses above, we use a following notations. In each item below, we assume $\mathcal{M} \stackrel{\text{def}}{=} (\nu \tilde{l})(\xi, \sigma)$, $\mathsf{fv}(e) \subset \mathsf{fv}(\mathcal{M})$, $\mathsf{fl}(e) \subset \mathsf{fl}(\mathcal{M})$, $\mathsf{fv}(N) \subset \mathsf{fv}(\mathcal{M})$, $\mathsf{fl}(N) \subset \mathsf{fl}(\mathcal{M})$, V closed, $\mathsf{fl}(V) \subset \mathsf{fl}(\mathcal{M})$, and leave the appropriate typability implicit.

(a) $\mathcal{M}[u:e]$ with u fresh and the variables and labels in e free in $\mathcal{M}$, denotes $(\nu \tilde{l})(\xi \cdot u : [\![e]\!]_{\xi,\sigma}, \sigma)$.
(b) $\mathcal{M}/u = (\nu \tilde{l})(\xi, \sigma)$ if $\mathcal{M} = (\nu \tilde{l})(\xi \cdot u : V, \sigma)$; otherwise $\mathcal{M}/u = \mathcal{M}$ (when $u \notin \mathsf{fv}(\mathcal{M})$)
(c) $\mathcal{M}[u:N] \Downarrow \mathcal{M}'$ when $(N\xi, \sigma) \Downarrow (\nu \tilde{l}')(V, \sigma')$ and $\mathcal{M}' = (\nu \tilde{l} \tilde{l}')(\xi \cdot u : V, \ \sigma')$ with $\mathcal{M} = (\nu \tilde{l})(\xi, \ \sigma)$.
(d) We write $\mathcal{M}[e \mapsto V]$ for $(\nu \tilde{l})(\xi, \sigma[l \mapsto V])$ with $\mathcal{M} = (\nu \tilde{l})(\xi, \sigma)$ and $[\![e]\!]_{\xi,\sigma} = l$.

In (1), the equality defined satisfies all standard axioms. (2) and (3) are standard. (4) takes any N as far as it does not change the state. (6) is from [19]. (7) is an extension from [6] where we evaluate N in a given context. (9) says that in any $\mathcal{M}$-initial hypothetical state satisfying C, the application of x to y returns z with final state satisfying C' (we need to consider hypothetical state since a function can be invoked any time later, not only at the present state).

We illustrate some of the defining clauses for satisfaction.

Quantification and Hiding Quantification. In the satisfaction of universal quantification over reference variables, it is notable to illustrate the difference between the standard quantification and the hiding quantification.

- $\forall x^{\alpha}.C$ says that for any reference x, which can be either (1) "existing" (free) reference or (2) a "fresh" reference with an arbitrary content, satisifes C (this second part allows natural assertions and reasoning and is needed for satisfying the standard quantification laws).
- $\bar{\forall} x^{\alpha}.C$ says that for any reference x which is hidden in the present model, C should hold. This generally mean x cannot be a free reference name.

Evaluation Formulae. In the satisfaction of evaluation formulae, we use the convergence $\mathcal{M}[u:N] \Downarrow \mathcal{M}'$. which intuitively means that $\mathcal{M}$ *can* reduce to $\mathcal{M}'$ through an arbitrary effects on $\mathcal{M}$ by an external program: in other words, $\mathcal{M}'$ is a hypothetical future state (or "possible world") of $\mathcal{M}$. Hence the defining clause for the satisfaction of evaluation formulae says:

> *In any initial hypothetical state which is reachable from the present state and which satisfies C, the application of e_1 to e_2 terminates and both the result z and the final state satisfy C'.*

The subsequent two clauses for satisfaction of located evaluation formulae in addition delimit the set of locations which may be modified by the evaluation. The last clause (11), which subsumes the preceding (10) by setting $E(z) \stackrel{\text{def}}{=} \vee_i z = w_i$, says:

> *The value stored in a model $\mathcal{M}_0$ evolved from $\mathcal{M}$ at a location other than those satisfying $E(z)$ (in $\mathcal{M}_0$), does not differ in $\mathcal{M}'$.*

Otherwise the condition for its satisfaction is the same as the original evaluation formulae. Note the predicate $E(z)$ is evaluated in $\mathcal{M}_0$, the hypothetical initial state (this is important for e.g. invariance rule, and also allows a natural reading). In the presence of local state, precisely encoding located assertions into unlocated ones seems difficult. As discussed in [6, 20], we need to consider hypothetical state in evaluation formulae since a function can be invoked any time later, not only at the present state.

We further illustrate how the satisfaction for evaluation formulae, especially its notion of hypothetical initial state, capture operational behaviour of applications. We first observe:

- In [6, 20], the notion of "hypothetical state" means an arbitrary store under the same typing (since the state is global and, in future when the function is invoked, the store may have been changed in any way by other programs).
- In the presence of local state, we cannot change the content of references arbitrarily since some locations may be inaccessible and, hence, remain constant. Thus we only consider a store which can result from the current one by some external effects as a hypothetical state.

Note the second "hypothetical state" subsumes the original notion, since when there is no hiding, a program can arbitrarily update the state of the exposed references.

We now illustrate this idea by examples. First, consider the following model:

$$\mathcal{M} \quad \stackrel{\text{def}}{=} \quad (\nu l)(u : \lambda().!l,\ \ l \mapsto 2) \tag{C.2}$$

A hypothetical state starting from this state can*not* include:

$$\mathcal{M}' \quad \stackrel{\text{def}}{=} \quad (\nu l)(u : \lambda().!l,\ \ l \mapsto 3) \tag{C.3}$$

since if (C.3) is allowed, we have

$$\mathcal{M} \models \neg\{\mathsf{T}\}u \bullet () = z\{z = 2\} \tag{C.4}$$

which is absurd. Note (C.2) means $\mathcal{M} \approx (u : \lambda().2,\ \ \emptyset)$, i.e. this model is stateless: so it is pointless to consider changing the state of this model.

As a more elaborate example, if we set:

$$\mathcal{M} \quad \stackrel{\text{def}}{=} \quad (\nu l)(u := \lambda().!l,\ w : \lambda().l :=!l + 1,\ \ l \mapsto 5) \tag{C.5}$$

We can check the set of all legitimate hypothetical states from this state (i.e. $\mathcal{M}'$ such that $\mathcal{M}[z : N] \Downarrow \mathcal{M}'$, without insignificant z portion in $\mathcal{M}'$) can be enumerated by:

$$\mathcal{M}' \quad \stackrel{\text{def}}{=} \quad (\nu l)(u := \lambda().!l,\ w : \lambda().l :=!l + 1,\ \ l \mapsto m) \tag{C.6}$$

for each $m \geq 5$ (since the only way an outside program can affect this model is to increment the content of l). Thus we have, for $\mathcal{M}$ in (C.5):

$$\mathcal{M} \models \{\mathsf{T}\}w \bullet () = z\{z \geq 5\} \tag{C.7}$$

which says in any *future* state where w is invoked, it always returns something no less than 5, which is operationally reasonable.

We can use this fact for semantically justifying:

$$\{\forall g.(g \bullet () \Downarrow \supset f \bullet g \Downarrow)\}\ L :_z\ \ \{z = \mathtt{t}\} \tag{C.8}$$

where we write $g \bullet () \Downarrow$ to denote $\{\mathsf{T}\}g \bullet ()\{\mathsf{T}\}$, and we set:

$$\begin{array}{rcl} L & \stackrel{\text{def}}{=} & \mathtt{let}\ x = \mathtt{ref}(5)\ \mathtt{in} \\ & & \quad \mathtt{let}\ u = \lambda().!x\ \mathtt{in} \\ & & \quad\quad \mathtt{let}\ w = \lambda().x :=!x + 1\ \mathtt{in} \\ & & \quad\quad\quad (fw)\ ;\ \mathtt{if}\ z \geq 5\ \mathtt{then}\ \mathtt{t}\ \mathtt{else}\ \mathtt{f} \end{array} \tag{C.9}$$

When the application fw takes place, some unknown computation occurs which may change the state: but we know, by assumption on f, that fw terminates; and that, after the two let's, which corresponds to the model (C.5) above, we have (C.7). Thus we can conclude that the program returns t, justifying (C.9).

Syntactically the judgement (C.9) is readily inferred using the axiom in Proposition 16, which says the invariant survives applications as far as certain condition (in particular termination) is satisfied.

C.2 Semantics of Judgement

Below we fix typings appropriately (as we discuss soon, it suffices to take the minimum typing covering the assertions and the program which always exists). First, the non-located judgement:

$$\models \{C\}\ M :_u \{C'\} \tag{C.10}$$

means

$$\forall \mathcal{M}.(\mathcal{M} \models C \supset \mathcal{M}[u : M] \Downarrow \mathcal{M}' \models C') \tag{C.11}$$

Let us write $\mathcal{M} \rightsquigarrow \mathcal{M}'$ for $\mathcal{M}[u:e] \Downarrow \mathcal{M}'_0$ such that $\mathcal{M}' \stackrel{\text{def}}{=} \mathcal{M}'_0/u.$ with u fresh. Then the above is equivalent to saying:

$$\forall \mathcal{M}, \mathcal{M}_0.(\mathcal{M} \rightsquigarrow \mathcal{M}_0 \models C \supset \mathcal{M}[u : M] \Downarrow \mathcal{M}' \models C') \tag{C.12}$$

This in turn is equivalent to, with $V \stackrel{\text{def}}{=} \lambda().M$:

$$\forall \mathcal{M}.(\mathcal{M}[m : V] \models \{C\}\, m \bullet () = u\{C'\}) \tag{C.13}$$

Second, the semantics of the located judgement:

$$\models \{C\}\ M :_u \{C'\}@\tilde{x} \tag{C.14}$$

may most easily be given using the corresponding located assertion following (C.13) above, with $V \stackrel{\text{def}}{=} \lambda().M$:

$$\forall \mathcal{M}.(\mathcal{M}[m : V] \models \{C\}\, m \bullet () = u\{C'\}@\tilde{x} \tag{C.15}$$

. We can further generalise (C.14) to denote the set of references:

$$\models \{C\}\ M :_u \{C'\}@\{z|E(z)\} \tag{C.16}$$

where we assume z is fresh and $(\exists z.E(z)) \equiv \mathsf{T}$. [5] Then the meaning of (C.16) is given as:

$$\forall \mathcal{M}.(\mathcal{M}[m : V] \models \{C\}\, m \bullet () = u\{C'\}@\{z|E(z)\}) \tag{C.17}$$

Note this subsumes (C.11) since if we set $E \stackrel{\text{def}}{=} \mathsf{T}$ in (C.17) the assertion. This concludes the definition of the semantics of judgements.

C.3 Refined Assertion Language for Completeness

We use the same notion of models and refine evaluation formula and content quantification. Each is decomposed into a pair, consisting of a modal operator and a more fine-grained evaluation formula/content quantifier. We only list their satisfaction and associated changes in proof rules.

We generate $\mathcal{M} \rightsquigarrow \mathcal{M}'$ inductively by: (1) $\mathcal{M} \rightsquigarrow \mathcal{M}$; and (2) if $\mathcal{M} \rightsquigarrow \mathcal{M}_0$ and $\mathcal{M}_0[u : N] \Downarrow \mathcal{M}'$ then $\mathcal{M} \rightsquigarrow \mathcal{M}'$. We write $\mathcal{M} \stackrel{\tilde{u}}{\rightsquigarrow} \mathcal{M}'$ when $\mathcal{M} \rightsquigarrow \mathcal{M}'$ and $\{\tilde{u}\} = \mathsf{dom}(\mathcal{M}') \setminus \mathsf{dom}(\mathcal{M})$. We now set:

[5] The condition $(\exists z.E(z)) \equiv \mathsf{T}$ prevents pathological cases: for example, if $E(z) \equiv \mathsf{F}$ then the statement becomes vacuous (under the given translation).

1. $\mathcal{M} \models \Box C$ if $\forall \mathcal{M}'.(\mathcal{M} \rightsquigarrow \mathcal{M}' \supset \mathcal{M}' \models C)$.
2. $\mathcal{M} \mod els \boxdot C$ if $\forall \mathcal{M}'.(\mathcal{M} \overset{\tilde{u}}{\rightsquigarrow} \mathcal{M}' \wedge \mathcal{M} \approx \mathcal{M}'/\tilde{u} \supset \mathcal{M}' \models C)$.
3. $\mathcal{M} \models e \bullet e' = x\{C\}$ if $\exists \mathcal{M}'.(\mathcal{M}[x : ee'] \Downarrow \mathcal{M}' \wedge \mathcal{M}' \models C)$.
4. $\mathcal{M} \models [!x]^{\circ} C$ if $\forall \mathcal{M}', N.(\mathcal{M}[x \mapsto N] \Downarrow \mathcal{M}' \wedge \forall V.(\mathcal{M}[x \mapsto V] \approx \mathcal{M}'[x \mapsto V]) \supset \mathcal{M}' \models C)$.

$\Box C$ says that C holds now and in any possible future; $\boxdot C$ says that C holds now and in any possible future which does not change the current state (it may be expanded). One-sided evaluation formula $e \bullet e' = x\{C\}$ says that if we apply e to e' now, the returned value and state satisfy C. Finally $[!x]^{\circ} C$ says that for any content of x, C holds (without considering expansion). We recover the original evaluation formula and universal content quantifiers by the following translations: $\{C\}x \bullet y = z\{C'\} \stackrel{\text{def}}{\equiv} \Box(C \supset x \bullet y = z\{C'\})$ and $[!x]C \stackrel{\text{def}}{\equiv} \boxdot [!x]^{\circ} C$. Accordingly, in the proof system, [*Assign*] now represents logical substitution using $[!x]^{\circ} C$ as follows: $C\{\!|n/!m|\!\} \stackrel{\text{def}}{=} [!m]^{\circ}(!m = n \supset C)$ (we can equivalently use the existential counterpart). [*Abs*] and [*App*] also use the decomposed formulae:

$$[Abs]\ \frac{\{A^{\frown x\tilde{i}} \wedge C\}\ M :_m \{C'\}}{\{A\}\ \lambda x.M :_u \{\Box \forall x\tilde{i}.(C \supset u \bullet x = m\{C'\})\}}$$

$$[App]\ \frac{\{C\}\ M :_m \{C_0\}\ \{C_0\}\ N :_n \{m \bullet n = u\{C'\}\}}{\{C\}\ MN :_u \{C'\}}$$

These decompositions are suggested by the proof of descriptive completeness. For the reasoning about the examples presented above, their use is not practically significant.

C.4 Proofs of Soundness

We prove the soundness theorem. We start with [*Var*].

$$\mathcal{M} \models C[x/u] \text{ implies } \mathcal{M}[u : x] \models C.$$

Similarly [*Const*] is reasoned:

$$\mathcal{M} \models C[\mathtt{c}/u] \text{ implies } \mathcal{M}[u : \mathtt{c}] \models C.$$

Next, [Inj_1] is reasoned:

$$\begin{array}{rcl} \mathcal{M} \models C & \Rightarrow & \mathcal{M}[m : M] \Downarrow \mathcal{M}' \models C'[\mathtt{inj}_1(m)/u] \\ & \Rightarrow & \mathcal{M}[m : M] \Downarrow \mathcal{M}' s.t.\ \mathcal{M}'[u : \mathtt{inj}_1(m)] \models C'. \\ & \Rightarrow & \mathcal{M}[m : M][u : \mathtt{inj}_1(m)] \models C'. \\ & \Rightarrow & \mathcal{M}[u : \mathtt{inj}_1(M)] \models C'. \end{array}$$

For [*Proj*] we reason as follows.

$$\mathcal{M} \models C \quad \Rightarrow \quad \mathcal{M}[m : M] \Downarrow \mathcal{M}' \models C'[\pi_1(m)/u], i.e.\ \mathcal{M}'[u : \pi_1(m)] \models C'$$

For [*Case*], we reason:

$$\begin{array}{lcl}
\mathcal{M} \models C & \Rightarrow & \mathcal{M}[m:M^{\alpha+\beta}] \Downarrow \mathcal{M}_0 \models C_0, \text{ if } \mathcal{M} = (\nu \tilde{l})(\xi,\sigma) \text{ and } (\nu \tilde{l})(M\xi,\sigma) \Downarrow (\nu \tilde{l'})(\mathtt{inj}_i(x_i)\xi,\sigma') \\
& \Rightarrow & \mathcal{M}_0[m:\mathtt{inj}_i(x_i)] \models C_0 \wedge m = \mathtt{inj}_i(x_i) \\
& \Rightarrow & \mathcal{M}_0[m:\mathtt{inj}_i(x_i)][u:M_1] \Downarrow \mathcal{M}' \models C' \\
& \Rightarrow & \mathcal{M}[u:\mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}}] \Downarrow \mathcal{M}'/m \models C'
\end{array}$$

Now we reason for [*Abs*]. We note, if A is stateless (cf. Definition 13) and $\mathcal{M} \models A$, then:

1. $\mathcal{M}[u:M] \Downarrow \mathcal{M}'$ with u fresh implies $\mathcal{M}' \models A$.
2. $\mathcal{M} \approx (\nu l)\mathcal{M}' \wedge \mathcal{M}'[x:l] \models A$.

Now assume $x, \tilde{i}$ have functional types.

$$\begin{array}{lll}
& \mathcal{M} \models A \supset \mathcal{M}[u:\lambda x.M] \models \forall x\tilde{i}.\{C\}u \bullet x = m\{C'\} & \\
\equiv & \mathcal{M} \models A \supset \mathcal{M}[u:\lambda x.M][x:V][\tilde{i}:\tilde{W}] \models \{C\}u \bullet x = m\{C'\} & \\
\equiv & \mathcal{M} \models A \supset (\mathcal{M}[u:\lambda x.M][x:V][\tilde{i}:\tilde{W}][k:N] \Downarrow \mathcal{M}_0 \wedge \mathcal{M}_0 \models C) & \\
& \qquad\qquad \supset (\mathcal{M}_0[m:ux] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') & \\
\equiv & (\mathcal{M} \models A \wedge \mathcal{M}[u:\lambda x.M][x:V][\tilde{i}:\tilde{W}][k:N] \Downarrow \mathcal{M}_0 \wedge \mathcal{M}_0 \models C) & \\
& \qquad\qquad \supset (\mathcal{M}_0[m:ux] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') & \\
\equiv & (\mathcal{M} \models A \wedge \mathcal{M}[u:\lambda x.M][x:V][\tilde{i}:\tilde{W}][k:N] \Downarrow \mathcal{M}_0 \wedge \mathcal{M}_0 \models A \wedge C) & \text{(1) above} \\
& \qquad\qquad \supset (\mathcal{M}_0[m:ux] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') & \\
\subset & \mathcal{M}_0 \models A \wedge C \supset (\mathcal{M}_0[m:M] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') &
\end{array}$$

If x has a reference type, we use (2) instead of (1). Then reasoning is identical.
[*App*] is reasoned as follows.

$$\begin{array}{lcl}
\mathcal{M} \models C & \Rightarrow & \mathcal{M}[m:M] \Downarrow \mathcal{M}_0 \models C_0 \\
& \Rightarrow & \mathcal{M}[n:N] \Downarrow \mathcal{M}_1 \models C_1 \wedge \{C_1\}m \bullet n = n\{C'\} \\
& \Rightarrow & \mathcal{M}[m:M][n:N][u:m \bullet n] \Downarrow \mathcal{M}' \models C_1' \\
& \Rightarrow & \mathcal{M}[u:MN] \Downarrow \mathcal{M}'/mn \models C'
\end{array}$$

For [*Deref*], we infer:

$$\begin{array}{lcl}
\mathcal{M} \models C & \Rightarrow & \mathcal{M}[m:M] \Downarrow \mathcal{M}' \models C'[!m/u] \\
& \Rightarrow & \mathcal{M}[m:!M] \Downarrow \mathcal{M}'/m \models C'
\end{array}$$

For [*Assign*] we first note that

$$\mathcal{M} \models \langle !x \rangle (C \wedge !x = m) \text{ iff } \mathcal{M}[x \mapsto [\![e]\!]_{\xi,\sigma}] \models C$$

Assume u is fresh.

$$\begin{array}{lcl}
\mathcal{M} \models C & \Rightarrow & \mathcal{M}[m:M] \Downarrow \mathcal{M}_0 \models C_0, \mathcal{M}_0[n:N] \Downarrow \mathcal{M}' \models C'\{\!|n/!m|\!\} \\
& \Rightarrow & \mathcal{M}'[m \mapsto n] \Downarrow \mathcal{M}'' \models C' \\
& \Rightarrow & \mathcal{M}[u:M := N] \Downarrow \mathcal{M}''/mn[u:()] \models C'
\end{array}$$

For [*Rec*], we establish the result for the following variant (already mentioned in Appendix B):

$$[\textit{Rec-Ren}]\ \frac{\{\mathsf{T}\}\ \lambda x.M :_u \{A\}}{\{\mathsf{T}\}\ \mu f.\lambda x.M :_u \{A[u/f]\}}$$

This variant and its relation with [*Rec*] is discussed below. Choose arbitrary $\mathcal{M}^{\Theta\cdot f:\alpha\Rightarrow\beta}$. Then $\mathcal{M} \models \mathsf{T}$ and

$$\begin{array}{lll}
\text{(IH)} & \Rightarrow & \forall \mathcal{M}.\mathcal{M}[u:\lambda x.M] \models A \\
& \Rightarrow & \forall \mathcal{M}.\mathcal{M}[f:\mu f.\lambda x.M][u:\lambda x.M] \models A \\
& \Rightarrow & \forall \mathcal{M}.\mathcal{M}[u,f:\mu f.\lambda x.M] \models A \\
& \Rightarrow & \forall \mathcal{M}.\mathcal{M}[u:\mu f.\lambda x.M] \models \forall f.(f = a \supset A) \\
& \Rightarrow & \forall \mathcal{M}.\mathcal{M}[u:\mu f.\lambda x.M] \models A[u/f]
\end{array}$$

[*Rec*] is easily derivable with [*Rec-Ren*] using mathematical induction at the level of assertions.

For [*Ref*], which shows the role of a fresh variable representing an arbitrary pre-state datum, let $u \notin \mathsf{fpn}(e)$. Then, with u fresh, for all M, we have:

$$\mathcal{M}[u:\mathtt{ref}(M)] \Downarrow \mathcal{M}' \text{ implies } \mathcal{M}' \models u\#e \qquad (\star)$$

[because: $\mathcal{M}'$ has shape:

$$(\nu\tilde{l})(\xi^{u} \cdot u : l, \sigma^{l} \cdot [l \mapsto V])$$

with $(\nu\tilde{l_0})(M\xi,\sigma_0) \Downarrow (\nu\tilde{l_0})(V,\sigma)$. Then one can check $[\![e]\!]_{\xi\cdot u:l,\sigma\cdot[l\mapsto V]} = [\![e]\!]_{\xi,\sigma} \notin \mathsf{lc}(l,\sigma\cdot[l\mapsto V]) = \mathsf{lc}(l,[l\mapsto V])$.] We can now reason, using $(\star)$:

$$\begin{array}{lll}
\mathcal{M} \models C & \Rightarrow & \mathcal{M}[m:M] \Downarrow \mathcal{M}' \models C' \\
& \Rightarrow & \mathcal{M}[m:M][u:\mathtt{ref}(M)] \Downarrow \mathcal{M}_0 \approx \mathcal{M}'[u\mapsto m] \\
& \text{and} & \mathcal{M}'[u\mapsto m] \models C' \wedge !u = m \wedge u\#i \quad (\star) \\
& \Rightarrow & \mathcal{M}[u:\mathtt{ref}(M)] \Downarrow \mathcal{M}''/m \models \#u.C[!u/m]
\end{array}$$

We complete all cases. □

D Appendix: Soundness of the Axioms

This appendix lists omitted proofs from Section 4. In D.1 we establish basic lemmas. In D.3, we prove (AIH)-axioms.

D.1 Basic Lemmas

We introduce a small notation: for $\mathcal{M}_{1,2}$ of the same type, we write $\mathcal{M}_1 \overset{[!\tilde{w}]}{\approx} \mathcal{M}_2$ when $\forall \tilde{V}.(\mathcal{M}_1[\tilde{w}\mapsto\tilde{V}] \approx \mathcal{M}_2[\tilde{w}\mapsto\tilde{V}])$.

Lemma 18. Suppose C is stateless except $\tilde{x}$ and $\mathcal{M} \models C$. Suppose $\mathcal{M} \rightsquigarrow \mathcal{M}'$ such that $\mathcal{M}$ and $\mathcal{M}'$ coincide in their content at $\tilde{x}$, i.e.

1. $\mathcal{M} \stackrel{\text{def}}{=} (\nu \tilde{l_0})(\xi,\ \sigma)$.
2. $\mathcal{M}' \stackrel{\text{def}}{=} (\nu \tilde{l_0}\tilde{l_1})(\xi \cdot \tilde{x}{:}\tilde{l},\ \sigma')$.
3. $\sigma(\xi(x_i)) = \sigma'(\xi(x_i))$ for each $x_i \in \{\tilde{x}\}$,

Then we have $\mathcal{M}' \models C$.

Proof. We first generate the set S of stateless formulae and the set SS of strongly stateless formulae as follows.

1. If $C \in SS$ then $C \in S$.
2. Equations and inequations are in SS.
3. Evaluation formulae of the form $\{C\}e \bullet e' = z\{C'\}@\tilde{w}$ where e and e' do not contain $!x_i$, are in S.
4. If $C_{1,2} \in S$ (resp. in $C_{1,2} \in SS$) then $C_1 \star C_2 \in S$ (resp. $C_1 \star C_2 \in SS$) for $\star \in \{\wedge, \vee\}$.
5. If $C \in S$ (resp. in $C \in SS$) then $\mathcal{Q}y.C \in S$ (resp. $\mathcal{Q}y.C \in SS$).
6. If $C \in SS$ then $[!y]C \in SS$ and $\langle !y\rangle C \in SS$.

We use induction on this generation rules to show:

(a) If $C \in SS$ and $\mathcal{M}_{1,2}$ coincide at $\tilde{x}$ then $\mathcal{M}_1 \models C$ iff $\mathcal{M}_2 \models C$.
(b) If $C \in S$ and $\mathcal{M}$ and $\mathcal{M}'$ satisfy the condition stated in Lemma 18, then $\mathcal{M} \models C$ implies $\mathcal{M}' \models C$.

We start from **(a)**. Equations and inequations are triviality. Similarly for conjunction and disjunction. For quantification, Consider $\forall y.C \in SS$ and assume

$$\mathcal{M}_1 \models \forall y.C \tag{D.1}$$

and

$$\mathcal{M}_1 \text{ and } \mathcal{M}_2 \text{ coincide at } \tilde{x}. \tag{D.2}$$

Let us write P for either e or V. From (D.2) and by definition we have

$$\mathcal{M}_1[y : P] \text{ and } \mathcal{M}_2[y : P] \text{ coincide at } \tilde{x}. \tag{D.3}$$

Take a logical term or value P. Then from (D.1) we have

$$\mathcal{M}_1[y : P] \models C. \tag{D.4}$$

Since $C \in SS$, by (D.3) and (D.4) as well as by induction hypothesis, we obtain $\mathcal{M}_2[y : P] \models C$, that is $\mathcal{M}_2 \models \forall y.C$, as required.

For content quantification, suppose $[!y]C \in SS$ (note $y \notin \{x\}$) and assume $\mathcal{M}_1 \models [!y]C$ and $\mathcal{M}_1$ and $\mathcal{M}_2$ coincide at $\tilde{x}$. Now suppose: $\mathcal{M}'_2 \overset{[!y]}{\approx} \mathcal{M}_2$ where we write

We can safely take $\mathcal{M}'_2$ to be the result of putting a different value at y in $\mathcal{M}_2$ (possibly with additional hidden store). Let $\mathcal{M}'_1$ be the result of putting the same value at y in $\mathcal{M}_1$. Then $\mathcal{M}'_1 \overset{[!y]}{\approx} \mathcal{M}_1$ hence $\mathcal{M}'_1 \models C$. By noting $\mathcal{M}'_1$ and $\mathcal{M}'_2$ coincide at $\tilde{x}y$ hence by induction hypothesis we obtain $\mathcal{M}'_2 \models C$. Hence $\mathcal{M}_2 \models [!y]C$, as required.

For **(b)**, first suppose $C \in SS$. By **(a)** and we immediately know C satisfies **(b)**. For evaluation formulae, if

$$\mathcal{M} \models \{C\} f \bullet g = h\{C'\} \tag{D.5}$$

and suppose $\mathcal{M} \rightsquigarrow \mathcal{M}'$. Suppose $\mathcal{M}' \rightsquigarrow \mathcal{M}_0 \models C$. Then by definition $\mathcal{M} \rightsquigarrow \mathcal{M}_0 \models C$ too. Moreover e and e' are identically interpreted in $\mathcal{M}$ and $\mathcal{M}'$ because they coincide at $\tilde{x}$ (see the defining clause of interpretation of evaluation formulae in Appendix C.1), hence as required. Note the argument extends to a generalised located assertion $\{C\}e \bullet e' = z\{C'\}@\{w|E(w)\}$ since in this case w of $E(w)$ is interpreted in $\mathcal{M}_0$. Conjunction and disjunction are immediate. For quantifications, suppose $\mathcal{M}_1 \models \forall y.C$ and suppose $\mathcal{M}_1 \rightsquigarrow \mathcal{M}_2$ is witnessed by N, i.e.: $\mathcal{M}_1[u:N] \Downarrow \mathcal{M}_0$ and $\mathcal{M}_2 \stackrel{\text{def}}{=} \mathcal{M}_0/u$, as well as $\mathcal{M}_{1,2}$ coincide at $\tilde{x}$. Let $\mathcal{M}_2[y:P] \Downarrow \mathcal{M}_2'$ and $\mathcal{M}_1[y:P] \Downarrow \mathcal{M}_1'$. By assumption we have

$$\mathcal{M}_1' \models C. \tag{D.6}$$

Since y is not in N and P does not alter any parts of the model (note P does not contain neither writes nor applications) we obtain:

$$\mathcal{M}_1'[u:N] \Downarrow \mathcal{M}_2'' \quad \text{s.t.} \quad \mathcal{M}_2''/u = \mathcal{M}_2'. \tag{D.7}$$

Note $\mathcal{M}_{1,2}'$ coincide at $\tilde{x}$. By (D.6) and induction hypothesis, we obtain $\mathcal{M}_2' \models C$, as required. The existential quantification is similarly reasoned, hence done. □

Remark. Semantically speaking, the statelessness of C except $\tilde{w}$ may be characterised by the following axiom: $\forall m.(\{C\}m \bullet ()\{\mathsf{T}\}@\tilde{w} \supset \{C\}m \bullet ()\{C\}@\tilde{w})$.

Lemma 19.

1. (narrowing) $\mathcal{M} \models C$ and $l \notin \mathsf{fl}(C)$ imply $(\nu l)\mathcal{M} \models C$
2. (scope opening) $((\nu l)\mathcal{M})[u:N] \equiv (\nu l)(\mathcal{M}[u:N])$ with $l \notin \mathsf{fl}(N)$.

Proof. By definition. □

Lemma 20. (expose) *If we have*

1. $\models \{C\}M :_u \{\nu\#x.C'\}$,
2. $\mathcal{M} \stackrel{def}{=} (\nu\tilde{l})(\xi,\ \sigma)$ *and* $\mathcal{M} \models C$ *and*
3. $\mathcal{M}[u:M] \Downarrow \mathcal{M}'$,

then we have $\mathcal{M}' \approx (\nu\tilde{l}\tilde{l}'l'')(\xi \cdot u:V,\ \sigma')$ *such that* $(\nu\tilde{l}\tilde{l}')(\xi \cdot u:V \cdot x:l'',\ \sigma') \models C$ *and* $l'' \notin \mathsf{lc}(\xi,\sigma')$.

Proof. We first expand $\{C\}M :_u \{\nu\#x.C'\}$ into:

$$\{C\}M :_u \{\exists x.(x\#i \wedge C')\} \tag{D.8}$$

with i fresh. Now assume $\mathcal{M} \models C$ as well as $\mathcal{M}[u:M] \Downarrow \mathcal{M}'$, so that we have $\mathcal{M}' \models \exists x.(x\#i \wedge C')$. Let $\mathcal{M}' \stackrel{\text{def}}{=} (\nu\tilde{l}\tilde{l}')(\xi,\ \sigma')$. By the semantics of $\exists$, we have two cases.

$$(\nu\tilde{l}\tilde{l}')(\xi \cdot x:l'',\ \sigma) \models (x\#i \wedge C') \tag{D.9}$$

such that either

(A) $l \notin \{\tilde{l'}\}$ (i.e. l'' exists from the prestate) or
(B) $l \in \{\tilde{l'}\}$ (i.e. l'' is newly created).

We show (A) is impossible. Suppose by contradiction (A) is the case. Noting i is in $\mathcal{M}$, let $\mathcal{M}_{l''}$ be the result of assigning l'' to i in $\mathcal{M}$ (otherwise leaving all data as in $\mathcal{M}$). Since i does not occur in C, we still have:

$$\mathcal{M}_{l''} \models C \tag{D.10}$$

Hence we have

$$\mathcal{M}[u:M] \Downarrow \mathcal{M}'_{l''} \models \exists x.(x\#i \wedge C'). \tag{D.11}$$

where $\mathcal{M}'_{l''}$ is the result of assigning l'' to i in $\mathcal{M}'$, otherwise as $\mathcal{M}'$ (note this is possible since i is fresh). This immediately means (A) is impossible. Since this holds for each free label in $\mathcal{M}$, we conclude only (B) is the possibility. So we let:

$$\mathcal{M}' \stackrel{\text{def}}{=} (\nu l\tilde{l'})(\xi \cdot u : V \cdot x : l'',\ \sigma \cdot \sigma') \tag{D.12}$$

such that

$$(\nu l\tilde{l'})(\xi \cdot u : V \cdot x : l'',\ \sigma) \models x\#i \wedge C'. \tag{D.13}$$

and $\mathsf{dom}(\sigma') = \{l\tilde{l'}\}$. We now show l'' is not accessible from ξ and σ. Suppose l'' *is* reachable from say $y \in \mathsf{dom}(\xi)$. Then we take $\mathcal{M}_y$ which is the result of assigning the image of y to i. By the same argument as before, we obtain:

$$\mathcal{M}_y[u:M] \Downarrow \mathcal{M}'_y \models \exists x.(x\#i \wedge C'). \tag{D.14}$$

Hence $(\nu l\tilde{l'})(\xi_y \cdot u : V \cdot x : l'',\ \sigma) \models x\#i \wedge C'$ where ξ_y comes from $\mathcal{M}_y$, which is impossible. Since the same argument holds for any hidden l'' mapped to x and any variable/label in $\mathsf{dom}(\xi \cup \sigma)$, we conclude l'' is unreachable from any prestate datum.

□

D.2 Proof of Proposition 11

Suppose $\mathcal{M} \models \{x\#fyw \wedge C\}f \bullet y = z\{C'\}@w$. The definition of the evaluation formula says, with u fresh,

$$\forall N, (\mathcal{M}[u:N] \Downarrow \mathcal{M}_0 \wedge \mathcal{M}_0 \models x\#fyw \wedge C \supset \exists \mathcal{M}'.(\mathcal{M}_0[z:fy] \Downarrow \mathcal{M}' \wedge \mathcal{M}' \models C')).$$

We prove such $\mathcal{M}'$ always satisfies $\mathcal{M}' \models x\#zw$. Assume

$$\mathcal{M}_0 \approx (\nu \vec{l})(\xi, \sigma_0 \uplus \sigma_x)$$

with $\xi(x) = l$, $\xi(y) = V_y$, $\xi(f) = V_f$ and $\xi(w) = l_w$ such that

$$\mathsf{lc}(\mathsf{fl}(V_f, V_y, l_w), \sigma_0 \uplus \sigma_x) = \mathsf{fl}(\sigma_0) = \mathsf{dom}(\sigma_0)$$

and $l_x \in \mathsf{dom}(\sigma_x)$. By this partition, during evaluation of $z : fy$, σ_x is unchanged, i.e.

$$(\nu \vec{l})(\xi \cdot z : fy, \sigma_0 \uplus \sigma_x) \longrightarrow\!\!\!\!\rightarrow (\nu \vec{l})(\xi \cdot z : V_f V_y, \sigma_0 \uplus \sigma_x) \longrightarrow\!\!\!\!\rightarrow (\nu \vec{l'})(\xi \cdot z : V_z, \sigma'_0 \uplus \sigma_x)$$

Then obviously there exists σ_1 such that $\sigma_1 \subset \sigma'_0$ and

$$\mathsf{lc}(\mathsf{fl}(V_z, l_w), \sigma'_0 \uplus \sigma_x) = \mathsf{fl}(\sigma_1) = \mathsf{dom}(\sigma_1)$$

Hence by Proposition 4, we have $\mathcal{M}_0 \models x\#wz$, completing the proof. □

D.3 Proof of Propositions 14

Proof. We first set:

$$G \equiv \nu\#x.G_0 \tag{D.15}$$
$$G_0 \equiv C_0 \wedge G_1 \wedge G_2 \tag{D.16}$$
$$G_1 \equiv \forall y.\{C_0 \wedge [!\tilde{x}]C\}u \bullet y = z\{C'\}@x\tilde{w} \tag{D.17}$$
$$G_2 \equiv \forall y.\{C_0 \wedge \tilde{x}\#y\tilde{r}\tilde{w}\}u \bullet y = z\{C_0 \wedge \tilde{x}\#z\tilde{w}\}@\tilde{w}\tilde{x} \tag{D.18}$$

By Proposition 11, it is suffice to assume G_2. W.o.l.g. we assume all vectors are unary, setting $\tilde{r} = r$, $\tilde{w} = w$ and $\tilde{x} = x$. Since the argument does not differ, we also set $E \stackrel{\text{def}}{=} \mathsf{T}$. Also we set the existential quantification is empty, setting $\tilde{g} = \emptyset$ (see the end of the proof).

Thus we are to prove, for each $\mathcal{M}$ and M:

$$\mathcal{M}_0[u : M] \Downarrow \mathcal{M} \models G \quad \supset \quad \mathcal{M}_0[u : M] \Downarrow \mathcal{M} \models G_1 \wedge G_2 \tag{D.19}$$

By Lemma 20 we can set[6]

$$\mathcal{M} \;=\; (\nu l\tilde{l'})(\xi, \sigma \cdot [l \mapsto V]) \models G \tag{D.20}$$

such that

$$\mathcal{M}^* \;=\; (\nu\tilde{l'})(\xi \cdot [x : l], \sigma \cdot [l \mapsto V]) \models C_0 \wedge G_1 \wedge G_2 \tag{D.21}$$

as well as

$$l \notin \mathsf{fl}(\xi, \sigma). \tag{D.22}$$

Now assume, for an appropriately typed N and fresh f:

$$\mathcal{M}^*[f : N] \Downarrow \mathcal{M}_a \tag{D.23}$$

Then we have

$$\mathcal{M}^*[f : N] \Downarrow \mathcal{M}^*_a \quad \text{such that} \quad (\nu l)\mathcal{M}^*_a = \mathcal{M}_a. \tag{D.24}$$

We now infer $\mathcal{M}^*_a \models C_0$ from $\mathcal{M}^* \models G_2$ and C_0 being stateless except x. By (D.21) and by the satisfaction of conjunction, we obtain $\mathcal{M}^* \models G_2$, i.e.

$$\mathcal{M}^* \models \forall y.\{C_0 \wedge x\#yrw\}u \bullet y = z\{C_0 \wedge x\#zw\}@wx \tag{D.25}$$

Observe

$$\mathcal{M}^* \models C_0 \;\wedge\; x\#yrw \tag{D.26}$$

Hence taking N to be uV (an application of u to V, whose shape is fixed by the assumed type, i.e. the base type or its composite) for any appropriately typed V (which, by the definition of N, cannot contain l), we have, by (D.25):

$$\mathcal{M}^*_a \models C_0 \;\wedge\; x\#yrw. \tag{D.27}$$

[6] For simplicity we assume only l is newly added: the general case does not change the argument.

Note the condition that C_0 holds and that x is disjoint from all visible data is still invariant in the resulting state. Hence we can again invoke u from $\mathcal{M}_a$ to obtain the same invariant.

In fact, for any appropriate N such that $\mathcal{M}[f:N]$ makes sense, it can only touch x through invoking u. Moreover by assumption we know C_0 stateless except x so that, by Lemma 18, no state change other than x can change the satisfiability of C_0. Thus, in $\mathcal{M}^*[f:N]$, as far as zero or more invocations of u results in the above invariant, C_0 continues to hold, that is:

$$\mathcal{M}_a^* \models C \tag{D.28}$$

By the previous argument we know x (or its denotation, l) is disjoint from the other visible references. That is, noting $x \notin \mathsf{fv}(C)$, for each active dereference $!y$, we have $y \neq x$, hence we have:

$$\mathcal{M}_a^* \models [!x]C \tag{D.29}$$

Thus we have

$$\mathcal{M}_a^* \models C_0 \ \wedge \ [!x]C \tag{D.30}$$

Hence we know:

$$\mathcal{M}_a^*[z:uy] \Downarrow \mathcal{M}_b^* \models C' \tag{D.31}$$

Since x is not used in C', we conclude, setting $\mathcal{M}_b \stackrel{\text{def}}{=} (\nu l)\mathcal{M}_b^*$:

$$\mathcal{M}_a[z:uy] \Downarrow \mathcal{M}_b \models C' \tag{D.32}$$

with the write set $x\tilde{w}$. Thus we have

$$\mathcal{M} \models \nu \# x.\{C\}u \bullet y = z\{C'\}@wx \tag{D.33}$$

Since x in (D.32) denotes the fresh l, this writing (if any) does not count if we start from $\mathcal{M}_a$, hence we obtain:

$$\mathcal{M} \models \{C\}u \bullet y = z\{C'\}@w \tag{D.34}$$

which is the required assertion.

For the case of $\tilde{g}$ is non-empty, we only have to add the assignment of values to $\tilde{g}$ guaranteed by their existential quantification, hence done. □

E Derivations for Examples in Section 5

This appendix lists the derivations omitted in Section 5.

E.1 Derivation for [*LetRef*]

We can derive [*LetRef*] as follows. Below i is fresh.

1. $\{C\}\ M :_m \{C_0\}$	(premise)
2. $\{C_0[!x/m] \wedge x\#\tilde{e}\}\ N :_u \{C'\}$ with $x \notin \mathsf{fpn}(\tilde{e})$	(premise)
3. $\{C\}\ \mathtt{ref}(M) :_x \{\nu y.(C_0[!x/m] \wedge x\#i \wedge x = y)\}$	(1,Ref)
4. $\{C\}\ \mathtt{ref}(M) :_x \{\nu y.(C_0[!x/m] \wedge x\#\tilde{e} \wedge x = y)\}$	(Subs n-times)
5. $\{C_0[!x/m] \wedge x\#\tilde{e} \wedge x = y\}\ N :_u \{C' \wedge x = y\}$	(2, Invariance)
6. $\{C\}\ \mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N :_u \{\nu y.(C' \wedge x = y)\}$	(4,5,LetOpen)
7. $\{C\}\ \mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N :_u \{\nu x.C'\}$	(Conseq)

The last line uses a standard logical law (discussed below). Lines 4 and 6 use the following derived/admissible proof rules:

$$[\mathit{Subs}]\ \frac{\{C\}\ M :_u \{C'\} \quad u \notin \mathsf{fpn}(e)}{\{C[e/i]\}\ M :_u \{C'[e/i]\}} \qquad [\mathit{LetOpen}]\ \frac{\{C\}\ M :_x \{\nu\tilde{y}.C_0\} \quad \{C_0\}\ N :_u \{C'\}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{\nu\tilde{y}.C'\}}$$

[*LetOpen*] opens the "scope" of $\tilde{y}$ to N. The crucial step is Line 5, which turns stronger "#" into "ν" (by definition), using the consequence rule.

E.2 Derivation for `IncUnShared`

For illustration, we contrast the inference of `IncShared` with:

$$\mathtt{IncUnShared} \stackrel{\text{def}}{=} a := \mathtt{Inc}; b := \mathtt{Inc}; c_1 := (!a)(); c_2 := (!b)(); (!c_1 + !c_2)$$

This program assigns to a and b two separate instances of `Inc`. This lack of sharing between a and b in `IncUnShared` is captured by the following derivation:

1. $\{\mathsf{T}\}\ \mathtt{Inc} :_m \{\nu x.\mathsf{inc}'(u,x,0)\}$
2. $\{\mathsf{T}\}\ a := \mathtt{Inc}\ \{\nu x.\mathsf{inc}'(!a,x,0)\}$
3. $\{\mathsf{inc}'(!a,x,0)\}\ b := \mathtt{Inc}\ \{\nu y.\mathsf{inc}''(0,0)\}$
4. $\{\mathsf{inc}''(0,0)\}\ c_1 := (!a)()\ \{\mathsf{inc}''(1,0) \wedge !c_1 = 1\}$
5. $\{\mathsf{inc}''(1,0)\}\ c_2 := (!b)()\ \{\mathsf{inc}''(1,1) \wedge !c_2 = 1\}$
6. $\{!c_1 = 1 \wedge !c_2 = 1\}\ (!c_1) + (!c_2) :_u \{u = 2\}$
7. $\{\mathsf{T}\}\ \mathtt{IncUnShared} :_u \{\nu xy.u = 2\}$
8. $\{\mathsf{T}\}\ \mathtt{IncUnShared} :_u \{u = 2\}$

Above $\mathsf{inc}''(n,m) = \mathsf{inc}'(!a,x,n) \wedge \mathsf{inc}'(!b,y,m) \wedge x \neq y$. Note $x \neq y$ is guaranteed by [*LetRef*]. This is in contrast to the derivation for `IncShared`, where, in Line 3, x is automatically shared after "$b :=\!!a$" which leads to scope extrusion.

E.3 Derivation for `mutualParity` and `safeEven`

Let us define:

$$M_x \stackrel{\text{def}}{=} \lambda n.\texttt{if } y=0 \texttt{ then f else not}((!y)(n-1))$$
$$M_y \stackrel{\text{def}}{=} \lambda n.\texttt{if } y=0 \texttt{ then t else not}((!x)(n-1))$$

We also use:

$$\begin{array}{ll} \mathit{IsOdd}'(u,gh,n,xy) & = \mathit{IsOdd}(u,gh,n,xy) \wedge\ !x=g \wedge\ !y=h \\ \mathit{IsEven}'(u,gh,n,xy) & = \mathit{IsEven}(u,gh,n,xy) \wedge\ !x=g \wedge\ !y=h \end{array}$$

We use the following derived rules and one standard structure rule appeared in [20].

$$[\mathit{Simple}]\ \frac{-}{\{C[e/u]\}e :_u \{C\}} \qquad [\mathit{IfH}]\ \frac{\{C\wedge e\}M_1 :_u \{C'\}\ \ \{C\wedge\neg e\}M_2 :_u \{C'\}}{\{C\}\,\texttt{if } e \texttt{ then } M_1 \texttt{ else } M_2 :_u \{C'\}}$$

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

Figure 3 lists the derivation for `MutualParity`. In Line 4, h in the evaluation formula can be replaced by $!y$ and vice versa because of $!y=h$ and the universal quatification of h.

$$\forall h.(!y=h\wedge\{C\}h\bullet n=z\{C'\}) \quad\equiv\quad \forall h.(!y=h\wedge\{C\}(!y)\bullet n=z\{C'\})$$

In Line 5, we use the following axiom for the evaluation formula from [20]:

$$\{C\wedge A\}\ e_1\bullet e_2=z\{C'\} \quad\equiv\quad A\ \supset\ \{C\}e_1\bullet e_2=z\{C'\}$$

where A is stateless and we set $A=\mathit{IsEven}(h,gh,n-1,xy)$. Line 9 is derived as Line 4 by replacing h and g by $!y$ and $!x$, respectively. Line 11 is the standard logical implication $(\forall x.(C_1\supset C_2)\supset(\exists x.C_1\supset\exists x.C_2))$. Now we derive for `safeEven`. Let us define:

$$\begin{array}{rl} \mathit{ValEven}(u) & = \forall n.\{\mathsf{T}\}u\bullet n=z\{z=\mathit{Even}(n)\}@\emptyset \\ C_0 & = !x=g\wedge\ !y=h\ \wedge\ \mathit{IsOdd}(g,gh,n,xy)\ \wedge\ x\#i\ \wedge\ y\#j \\ \mathit{Even}_a & = C_0\ \wedge\ \forall n.\{C_0\}u\bullet n=z\{C_0\}@xy \\ \mathit{Even}_b & = \forall n.\{C_0\}u\bullet n=z\{z=\mathit{Even}(n)\}@xy \end{array}$$

The derivation is given as follows.

1. $\{\mathsf{T}\}\lambda n.\texttt{t} :_m \{\mathsf{T}\}@\emptyset$

---

2. $\{\mathsf{T}\}\texttt{mutualParity}; !y :_u \{\exists gh.\mathit{IsOddEven}(gh,gu,xy,n)\}@xy$

---

3. $\{\mathsf{T}\}\texttt{mutualParity}; !y :_u \{\exists gh.(\mathit{Even}_a\ \wedge\ \mathit{Even}_b)\}@xy$

---

4. $\{xy\#ij\}\texttt{mutualParity}; !y :_u \{\exists gh.(xy\#ij\ \wedge\ \mathit{Even}_a\ \wedge\ \mathit{Even}_b)\}@xy$

---

5. $\{\mathsf{T}\}\texttt{safeEven} :_u \{\nu xy.\exists gh.(xy\#ij\ \wedge\ \mathit{Even}_a\ \wedge\ \mathit{Even}_b)\}@\emptyset$

---

6. $\{\mathsf{T}\}m\bullet()=u\{\nu xy\exists gh.(xy\#ij\ \wedge\ \mathit{Even}_a\wedge\mathit{Even}_b)\}\supset\{\mathsf{T}\}m\bullet()=u\{\mathit{ValEven}(u)\}$ (by (AIH))

---

7. $\{\mathsf{T}\}\texttt{safeEven} :_u \{\mathit{ValEven}(u)\}@\emptyset$

Fig. 3 `mutualParity` derivations

1. $\{(n \geq 1 \supset \mathit{IsEven}'(!y, gh, n-1, xy)) \ \wedge \ n = 0\}\ \mathtt{f} :_z \{z = \mathit{Odd}(n) \wedge !x = g \wedge !y = h\}@\emptyset$ (Const)

2. $\{(n \geq 1 \supset \mathit{IsEven}'(!y, gh, n-1, xy)) \ \wedge \ n \geq 1\}$
   $\mathtt{not}((!y)(n-1)) :_z \{z = \mathit{Odd}(n) \wedge !x = g \wedge !y = h\}@\emptyset$ (Simple, App)

3. $\{n \geq 1 \supset \mathit{IsEven}'(!y, gh, n-1, xy)\}$
   $\mathtt{if}\ n = 0\ \mathtt{then}\ \mathtt{f}\ \mathtt{else}\ \mathtt{not}((!y)(n-1)) :_m \{z = \mathit{Odd}(n) \wedge !x = g \wedge !y = h\}@\emptyset$ (IfH)

4. $\{\mathsf{T}\}\ \lambda n.\mathtt{if}\ n = 0\ \mathtt{then}\ \mathtt{f}\ \mathtt{else}\ \mathtt{not}((!y)(n-1)) :_u$
   $\{\ \forall gh, n \geq 1.\{\mathit{IsEven}'(h, gh, n-1, xy)\}u \bullet n = z\{z = \mathit{Odd}(n) \wedge !x = g \wedge !y = h\}@\emptyset\}@\emptyset$ (Abs, $\forall$, Conseq)

5. $\{\mathsf{T}\}\ M_x :_u \{\ \forall gh, n \geq 1.(\mathit{IsEven}(h, gh, n-1, xy) \supset \mathit{IsOdd}(u, gh, n, xy))\}@\emptyset$ (Conseq)

6. $\{\mathsf{T}\}\ x := M_x\{\ \forall gh, n \geq 1.(\mathit{IsEven}(h, gh, n-1, xy) \supset \mathit{IsOdd}(!x, gh, n, xy)) \wedge !x = g\}@x$ (Assign)

7. $\{\mathsf{T}\}\ y := M_y\{\ \forall gh, n \geq 1.(\mathit{IsOdd}(g, gh, n-1, xy) \supset \mathit{IsEven}(!y, gh, n, xy)) \wedge !y = h\}@y$

8. $\{\mathsf{T}\}\ \mathtt{mutualParity}$
   $\{\forall gh.n \geq 1.((\mathit{IsEven}(h, gh, n-1, xy) \wedge \mathit{IsOdd}(g, gh, n-1, xy)) \supset$
   $(\mathit{IsEven}(!y, gh, n, xy) \wedge \mathit{IsOdd}(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\ \}@xy$ ($\wedge$-Post)

9. $\{\mathsf{T}\}\ \mathtt{mutualParity}$
   $\{\forall n \geq 1 gh.((\mathit{IsEven}(h, gh, n-1, xy) \wedge \mathit{IsOdd}(g, gh, n-1, xy) \wedge !x = g \wedge !y = h) \supset$
   $(\mathit{IsEven}(!y, gh, n, xy) \wedge \mathit{IsOdd}(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\}@xy$ (Conseq)

10. $\{\mathsf{T}\}\ \mathtt{mutualParity}$
    $\{\forall n \geq 1 gh.((\mathit{IsEven}(!y, gh, n-1, xy) \wedge \mathit{IsOdd}(!x, gh, n-1, xy) \wedge !x = g \wedge !y = h) \supset$
    $(\mathit{IsEven}(!y, gh, n, xy) \wedge \mathit{IsOdd}(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\}@xy$ (Conseq)

11. $\{\mathsf{T}\}\ \mathtt{mutualParity}$
    $\{\forall n \geq 1.(\exists gh.(\mathit{IsEven}(!x, gh, n-1, xy) \wedge \mathit{IsOdd}(!y, gh, n-1, xy) \wedge !x = g \wedge !y = h) \supset$
    $\exists gh.(\mathit{IsEven}(!y, gh, n, xy) \wedge \mathit{IsOdd}(!x, gh, n, xy) \wedge !x = g \wedge !y = h)\}@xy$ (Conseq)

12. $\{\mathsf{T}\}\ \mathtt{mutualParity}\{\exists gh.\mathit{IsOddEven}(gh, !x!y, xy, n)\}@xy$

E.4 Derivation for `profile`

We derive:

$$\{\forall y.\{C\}f \bullet y = z\{C'\}@\tilde{w}\}\ \mathtt{profile} :_u \{\forall y.\{C\}u \bullet y = z\{C'\}@\tilde{w}\} \qquad \text{(E.1)}$$

which says: *if f satisfies the specification $\forall y.\{C\}f \bullet y = z\{C'\}$ and moreover if it is total, then* `profile` *satisfies the same specification.* First we derive:

$$\begin{array}{rl}
E & = \forall y.\{C\}f \bullet y = z\{C'\}@\tilde{w} \\
\supset E_0 & = \forall yi.\{C \ \wedge \ x\#i\}f \bullet y = z\{C'\}@\tilde{w}x \qquad \text{Axiom (e8) in [20]} \\
\supset E_1 & = \forall yi.\{C \ \wedge \ x\#i\}f \bullet y = z\{x\#z\tilde{w}i\}@\tilde{w}x \qquad \text{Proposition 11} \\
\supset E_2 & = \forall yi.\{C \ \wedge \ x\#i\}f \bullet y = z\{C' \ \wedge \ x\#i\}@\tilde{w}x \quad \text{Axiom (e8) in [20]}
\end{array}$$

We also let $E_3 = \forall yi \neq x.\{[!x]C \wedge x\#i\}f \bullet y = z\{C' \wedge x\#i\}@\tilde{w}x$. The inference follows.

$1.\{\mathsf{T}\}x :=!x+1\{\mathsf{T}\}@x$	(Assign)
$2.\{[!x]C \wedge E \wedge x\#i \wedge x \neq y\}\ x :=!x+1\ \{C \wedge E \wedge x\#i \wedge x \neq y\}@x$	(Inv-#, Conseq)
$3.\{C \wedge E \wedge x\#i \wedge x \neq y\}\ fy :_z \{C' \wedge x\#i \wedge x \neq y\}@\tilde{w}x$	(App, Conseq)
$4.\{[!x]C \wedge E \wedge x\#i \wedge x \neq y\}x := x+1; fy :_z \{C' \wedge x\#i \wedge x \neq y\}@x\tilde{w}$	(2, 3, Seq)
$5.\{E\}\ \lambda y.(x := x+1; fy) :_u \{E_2\}@\emptyset$	(4, Abs, Inv)
$6.\{E\}\ \lambda y.(x := x+1; fy) :_u \{\mathsf{Inv}(u, x\#i, \tilde{x})\}@\emptyset$	(Abs, Inv)
$7.\{E\}\mathtt{profile}\{\nu x.(\mathsf{Inv}(u, x\#i, \tilde{x}) \wedge E_3)\}@\emptyset$	(LetRef)
$8.\{E\}m \bullet () = u\{\nu x.(\mathsf{Inv}(u, x\#i, \tilde{x}) \wedge E_3)\} \supset \{E\}m \bullet () = u\{E\}$	$(\star)$
$9.\{E\}\mathtt{profile} :_u \{E\}@\emptyset$	(7,8,ConsEval)

Above Line 2 uses: for any C, x we have $[!x][!x]C \equiv [!x]C$. Also by $[!x]E \equiv E$ and by $[!x]x\#i \equiv x\#i$ (by Proposition 7 (3)-5), [*Inv*] becomes applicable. Line 6 is inferred by Proposition 14.

E.5 Derivation for Meyer-Sieber

For the derivation of (5.8) we use:

$$E = \forall f.(\{\mathsf{T}\}f \bullet ()\{\mathsf{T}\}@\emptyset \ \supset\ \{C\}g \bullet f\{C'\})$$

We use the following [*LetRef*] which is derived by [*Ref*] where C' is replaced by $[!x]C'$.

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

The derivation follows. Below $M_{1,2}$ is the body of the first/second lets, respectively.

$1.\{\mathit{Even}(!x) \wedge [!x]C'\}\ \mathtt{if}\ \mathit{even}(!x)\ \mathtt{then}\ ()\ \mathtt{else}\ \Omega()\ \{[!x]C'\}@\emptyset$	(If)
$2.\{[!x]C\}\ gf\ \{[!x]C'\}$	(cf. § 5.5)
$3.\{\mathit{Even}(!x) \wedge [!x]C\}\ gf\ \{\mathit{Even}(!x) \wedge [!x]C'\}$	(2, Inv)
$4.\{E \wedge [!x]C \wedge \mathit{Even}(!x) \wedge x\#gi\}\mathtt{let}\ f = \ldots\ \mathtt{in}\ (gf;\ldots)\{[!x]C' \wedge x\#i\}$	(3, Seq, Let)
$5.\{E \wedge C\}\ \mathtt{MeyerSieber}\ \{\nu x.([!x]C' \wedge x\#i)\}$	(4, LetRef)
$6.\{E \wedge C\}\ \mathtt{MeyerSieber}\ \{C'\}$	(9, Prop. 15)

E.6 Derivation for Object

We need the following generalisation. The procedure u in (AIH) is of a function type $\alpha \Rightarrow \beta$: when values of other types such as $\alpha \times \beta$ or $\alpha + \beta$ are returned, we can make use of a generalisation. For simplicity we restrict our attention to the case when types do not contain recursive or reference types.

$$\mathsf{Inv}(u^{\alpha\times\beta}, C_0, \tilde{x}) \models \wedge_{i=1,2} \mathsf{Inv}(\pi_i(u), C_0, \tilde{x})$$
$$\mathsf{Inv}(u^{\alpha+\beta}, C_0, \tilde{x}) \models \wedge_{i=1,2} \forall y_i.(u = \mathsf{inj}_i(y_i) \supset \mathsf{Inv}(y_i, C_0, \tilde{x}))$$
$$\mathsf{Inv}(u^{\alpha}, C_0, \tilde{x}) \models \mathsf{T} \qquad (\alpha \in \{\mathsf{Unit}, \mathsf{Nat}, \mathsf{Bool}\})$$

Using this extension, we can generalise (AIH) so that the cancelling of C_0 is possible for all components of u. For example, if u is a pair of functions, those two functions need to satisfy the same condition as in (AIH). This is what we shall use for `cellGen`. We call the resulting generalised axiom ($\mathsf{AIH_c}$).

Let `cell` be the internal λ-abstraction of `cellGen`. First, it is easy to obtain:

$$\{\mathsf{T}\}\ \texttt{cell} :_o \ \{I_0 \wedge G_1 \wedge G_2 \wedge E'\} \tag{E.2}$$

where, with $I_0 = !x_0 = !x_1$ and $E' = !x_0 = z$.

$$G_1 = \{I_0\}\pi_1(o) \bullet () = v\{v = !x_0 \wedge I_0\}@\emptyset$$
$$G_2 = \forall w.\{I_0\}\pi_1(o) \bullet w\{!x_0 = w \wedge I_0\}@x_0x_1$$

which will become, after taking off the invariant I_0:

$$G_1' = \{\mathsf{T}\}\pi_1(o) \bullet () = v\{v = !x_1\}@\emptyset$$
$$G_2' = \forall w.\{\mathsf{T}\}\pi_1(o) \bullet w\{!x_0 = w\}@x_0.$$

Note I_0 is stateless except for x_0. In G_1, notice the empty write set means $!x_1$ does not change from the pre to the postcondition. We now present the inference. Below we set $\texttt{cell}' \stackrel{\text{def}}{=} \texttt{let}\ y = \texttt{ref}(0)\ \texttt{in}\ \texttt{cell}$ and i, k fresh.

1. $\{\mathsf{T}\}\ \texttt{cell} :_o \ \{I_0 \wedge G_1 \wedge G_2 \wedge E'\}$

2. $\{\mathsf{T}\}\ \texttt{cell}' :_o \ \{I_0 \wedge G_1 \wedge G_2 \wedge E'\}$ (LetRef)

3. $\{\mathsf{T}\}\texttt{let}\ x_1 = z\ \texttt{in}\ \texttt{cell}' :_o \{\nu x_1.(x_1 \# i \wedge I_0 \wedge G_1 \wedge G_2) \wedge E'\}$ (LetRef)

4. $\{\mathsf{T}\}\ \texttt{let}\ x_1 = z\ \texttt{in}\ \texttt{cell}' :_o \ \{G_1' \wedge G_2' \wedge E'\}$ ($\mathsf{AIH_c}$, ConsEval)

5. $\{\mathsf{T}\}\ \texttt{let}\ x_{0,1} = z\ \texttt{in}\ \texttt{cell}' :_o \ \{\nu x.(\ x \# k \wedge G_1' \wedge G_2' \wedge E'\ )\}$ (LetRef)

6. $\{\mathsf{T}\}\ \texttt{cellGen} :_u \ \{\mathit{CellGen}(u)\}$. (Abs)