Logical Reasoning for Higher-Order Functions with Local State

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Martin Berger
Queen Mary, University of London
martinb@dcs.qmul.ac.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 can be fully captured by a predicate which asserts reachability of a reference name from a possibly higher-order datum. The logic enjoys a strong match with the semantics of programs, in the sense that valid assertions characterise the standard contextual congruence. 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 [13] for call-by-value higher-order functions with ML-like new reference generation [4, 5], 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:

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

We use the standard notation [35]: 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 [39, § 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 [35]. Difficulties in formal reasoning about shared (possibly higher-order) local store, both axiomatic and otherwise, have been well-known since [14, 27, 29].

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 [39, § 1].

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

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* [39].

> *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 [19, 37, 39, 43]. 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 [21]:

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

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 [19, 23, 24, 27, 37, 39], 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, 35, 36] 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, 16–18]. 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 [32]). 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 proof rules for the remaining constructs from our foregoing program logics. At the level of assertions, we can find a set of useful axioms for (un)reachability, which are effectively combined with logical primitives and associated axioms for higher-order functions and aliasing, which were studied in our preceding works [6, 18]. These axioms for reachability are closely related with reasoning principles studied in existing semantic studies on local state, such as the principle of local invariant. 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 outlines the semantics of the logic and its properties. 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. Online Appendix [3] lists auxiliary definitions and omitted derivations. Detailed definitions and proofs are found in the full version [2].

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.

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

We use the standard notation [35]. 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 $\mathtt{let}\ x = M\ \mathtt{in}\ N$. Typing is standard: we take the equi-isomorphic approach [35] for recursive types. Nat, Bool and Unit *atomic types*. We leave illustration of each construct to standard textbooks [35], except for the focus of the present study, $\mathtt{ref}(M)$, 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:

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

Above σ is a store, a finite map from locations to closed values, denoting the initial state; whereas $\sigma \uplus [l \mapsto V]$ is the result of disjointly adding a pair (l, V) to σ. The resulting configuration uses a binder (the use of the ν-binding simplifies the correspondence with models 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 [35] except for the above rule and for closing it under ν-bindings.

A *basis* $\Gamma; \Delta$ is a pair of finite maps, one from variables to non-reference types ($\Gamma, \Gamma', \ldots$), the other from locations and variables to reference types ($\Delta, \Delta', \ldots$). $\Theta, \Theta', \ldots$ combine two kinds of bases. The typing rules are standard [35]. 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 [26, § 2.8], extended with assertions for evaluation [17, 18] (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. The grammar follows, letting $\star \in \{\wedge, \vee, \supset\}$ and $\mathcal{Q} \in \{\forall, \exists\}$.

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

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

Formulae include the standard logical connectives and quantification [26]. We include, following [6, 17], quantifications over type variables ($\mathrm{X}, \mathrm{Y}, \ldots$). 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 [18], 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. $y\#x$ [12, 40] 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'. We often write $!\tilde{x}$ to mean $!x_1..!x_n$ with $\tilde{x} = x_1..x_n$. $C_1 \equiv C_2$ 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, hence z from x. So the final state satisfies $x \hookrightarrow y \wedge y \hookrightarrow z \wedge x \hookrightarrow z$.
2. Next, assuming w is newly generated, we may wish to say w is *unreachable* from x, to ensure freshness of w. For this we assert $w\#x$, which, as noted, stands for $\neg(x \hookrightarrow w)$. $x\#y$ always implies $x \neq y$. Note that $x \hookrightarrow x \equiv x \hookrightarrow\ !x \equiv \mathsf{T}$ 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.

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

   Above "$@x$", called *write set*, indicates that the evaluation alters at most x, leaving content of other references unchanged. Intuitively, this formula stands for the following assertion with r and h fresh.

   $$\forall \mathrm{X}, r^{\mathsf{Ref}(\mathrm{X})}, h^{\mathrm{X}}, x, i.\ \{!x = i \wedge r \neq x \wedge !r = h\}u \bullet () = z\{!x = i + 1 \wedge !x = z \wedge !r = h\}$$

   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 [3, B.2].
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. The program $\lambda n^{\mathsf{Nat}}.\mathtt{ref}(n)$, named u, meets the following specification. Let i be fresh.

   $$\forall \mathrm{X}.\forall i^{\mathrm{X}}.\forall n^{\mathsf{Nat}}.\{\mathsf{T}\}u \bullet n = z\{z\#i \wedge !z = n\}@\emptyset$$

   Since i is universally quantified from the outside, it represents an arbitrary datum in the *initial* state. The assertion says a reference z, which is created by applying u to n, is disjoint from any such i, i.e. z is fresh unreachable from any other datum.

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

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

In the first line, νx says x is 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 [13], augmented with a fresh name called *anchor* [16–18].

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

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, 18], 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 $u \notin \mathsf{fv}(C')$.

- $\{C\}M :_u \{C'\}@\tilde{w}$ intuitively stands for $\{C \wedge y \neq \tilde{w} \wedge !y = i\}M :_u \{C' \wedge !y = i\}$ with y fresh. $\tilde{w}$ is a write set (cf. § 2.3).
- $\{C\}\ M :_m \{\nu x.C'\}$ stands for $\{C\}\ M :_m \{\exists x.(x \neq i \wedge C')\}$.
- $\{C\}\ M :_m \{\nu \# x.C'\}$ stands for $\{C\}\ M :_m \{\exists x.(x \# i \wedge C')\}$.
- $\{C\}\ M :_m \{\# m.C'\}$ stands for $\{C\}\ M :_m \{m \# i \wedge C'\}$.

The full compositional proof rules are given in Figure 1 in Appendix A. In spite of the semantic enrichment, all compositional proof rules stay as in the base logic [6] 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.

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 introduced in our preceding study [6] works in harmony with the (un)reachability predicate.

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

The side condition says that the assertion C_0 is invariant under all contents of the variables in the write set of M, thus ensuring that the writing by M does not alter C_0. We then have:

$$[Inv\text{-}Val]\ \frac{\{C\}\ V :_m \{C'\}}{\{C \wedge C_0\}\ V :_m \{C' \wedge C_0\}}$$

$$[Inv\text{-}\#]\ \frac{\{C\}\ M :_m \{C'\}@x \quad \text{no dereference occurs in } \tilde{e}}{\{C \wedge x \# \tilde{e}\}\ M :_m \{C' \wedge x \# \tilde{e}\}@x}$$

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 2, clause 3-(5) later).

p

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 and Soundness

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 [30]. 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:

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

which corresponds to the freshness assertion "$\nu x.C$".

Definition 1 (models) An *open model of type* $\Theta = \Gamma;\Delta$, with Δ closed, is a tuple (ξ, σ) where:

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

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, inducing the standard α-equality. $\mathcal{M}, \mathcal{M}', \ldots$ range over models.

An open model maps variables and locations to closed values: a model then specifies part of the locations as "hidden" (for treatment of type variables see Appendix B).

Models in the above sense are very concrete. Since assertions in the present logic are intended to capture observable behaviour of programs, 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 $\mathcal{M}_i \stackrel{\text{def}}{=} (\nu \tilde{l_i})(\tilde{x} : \tilde{V}_i, \sigma_i)$ under the same typing. Then we write $\mathcal{M}_1 \approx \mathcal{M}_2$ if the following clause holds for each well-typed, closed $C[\,\cdot\,]$ in which no labels from $\tilde{l}_{1,2}$ occcur:

$$(\nu \tilde{l_1})(C[\langle \tilde{V}_1 \rangle], \sigma_1) \Downarrow \quad \text{iff} \quad (\nu \tilde{l_2})(C[\langle \tilde{V}_2 \rangle], \sigma_2) \Downarrow$$

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 [35] (for reference Appendix B in [3] lists the definition of $\cong$).

3.2 Semantics of Reachability.

Let σ be a store and $S \subset \mathsf{dom}(\sigma)$. Then the *label closure of S in σ*, written $\mathsf{ncl}(S, \sigma)$, 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 1 *For all σ, we have:*

1. *$S \subset \mathsf{ncl}(S, \sigma)$; $S_1 \subset S_2$ implies $\mathsf{ncl}(S_1, \sigma) \subset \mathsf{ncl}(S_2, \sigma)$; and $\mathsf{ncl}(S, \sigma) = \mathsf{ncl}(\mathsf{ncl}(S, \sigma), \sigma)$*
2. *$\mathsf{ncl}(S_1, \sigma) \cup \mathsf{ncl}(S_2, \sigma) = \mathsf{ncl}(S_1 \cup S_2, \sigma)$*

We now set:

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

Above $[\![e_i]\!]_{\xi,\sigma}$ is the obvious interpretation of e_i (see Appendix B). 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.

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

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

The proof is easy by Lemma 1. 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.

3.3 Soundness and observational completeness.

The definitions of satisfiability $\mathcal{M} \models C$ other than reachability is given in Appendix B in [3] (logical connectives are interpreted classically: type variables are treated syntactically [17]). 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:

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

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.

Theorem 1 (soundness) $\vdash \{C\}M :_u \{C'\}$ *implies* $\models \{C\}M :_u \{C'\}$.

Another basic property of the logic is that its judgements distinguish programs just as the observational congruence does (observational completeness [6, 18]). Write $\cong$ for the standard contextual congruence [35] for the programs; and $M_1 \cong_{\mathcal{L}} M_2 : \alpha$ when we have $\models \{C\}M_1 :_u \{C'\}$ iff $\models \{C\}M_2 :_u \{C'\}$.

Theorem 2 (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 1. Note our types include recursive types (taken up to tree unfolding [35]).

Proposition 2 (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) $[!x]y \hookrightarrow x \ \equiv\ y \hookrightarrow x \ \equiv\ \langle !x\rangle y \hookrightarrow x$.

3-(5) says that altering the content of x does not affect reachability *to* x (because: for an update of x to invalidate $y \hookrightarrow x$, y should first reach 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 used for deriving [*Inv*-#] in §2.4 (we cannot substitute $!x$ for y in $[!x]x\#y$ to avoid name capture [6]).

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. The proof of the theorem below again relies on Proposition 2.

Theorem 3 (elimination) *Suppose all reachability predicates in C are finite. Then there exists C' such that $C \equiv C'$ and no reachability predicate occurs in C'.*

A straightforward coinductive extension of the above axioms (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.

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

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 [12, 38, 43]). Now define:

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

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

Proposition 3 $x \hookrightarrow y \ \equiv\ \exists n.(x \rhd^n y) \ \equiv\ (x = y \vee x \rhd y \vee \exists z.(x \rhd z \wedge z \neq y \wedge z \hookrightarrow y))$.

Proposition 3, combined with Theorem 3, 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 4 (undecidability of $\rhd$ and $\hookrightarrow$) (1) $\mathcal{M} \models f^{\alpha\Rightarrow\beta} \rhd x$ *is undecidable.* (2) $\mathcal{M} \models f^{\alpha\Rightarrow\beta} \hookrightarrow x$ *is undecidable.*

The proof of (1) reduces the satisfiability to the halting problem of PCFv-terms. We then reduce (2) to (1). The result holds even if we take call-by-value $\beta\eta$-equality as the underlying equality.

Proposition 4 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 5 (unreachable function) *The following is valid:* $\{C\}\, f \bullet y = z\{C'\}@\tilde{w} \supset \{C \wedge x\#\, fy\tilde{w}\}\, f \bullet y = z\{C' \wedge x\#z\tilde{w}\}@\tilde{w}$.

Proposition 5 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 local state, its behaviour often crucially relies on an invariant on its local store. Let us first consider a function from a base type to a base type which writes to local references of a base type. Even programs of this kind pose fundamental difficulties in reasoning [27]. Take:

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

The program behaves as a pure function $\lambda y.(y > 7)$. For this purpose it keeps the obvious local invariant, $!x = 7$. We demand this assertion to survive under arbitrary invocations of this function: thus (naming the function u) we arrive at the following invariant:

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

The assertion (4.3) says that: (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). We then observe:

$$C_1 \ =\ \ \forall y.\{!x = 7\}u \bullet y = z\{z = (y > 7)\}@\emptyset \tag{4.4}$$

The program `compHide` is easily given the following judgement:

$$\{\mathsf{T}\}\mathtt{compHide} :_u \{\mathsf{v}\#x.(C_0 \ \wedge\ C_1)\} \tag{4.5}$$

(for the notation $\mathsf{v}\#x$ see § 2.4.) 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:

$$\{\mathsf{T}\}\mathtt{compHide} :_u \{\forall y.\{\mathsf{T}\}u \bullet y = z\{z = (y > 7)\}\} \tag{4.6}$$

which describes a purely functional behaviour. We now show the underlying reasoning principle as an axiom. First we introduce a

notation for invariant. Below we assume z and $\tilde{w}$ are fresh and have atomic types (Unit, Bool or Nat) or their products/sums.

$$\mathsf{Inv}_A(u,C_0,\tilde{x},\tilde{w}) \;=\; C_0 \;\wedge\; \forall y.\{C_0\}u\bullet y = z\{C_0\}@\tilde{x}\tilde{w} \qquad (4.7)$$

$\mathsf{Inv}_A(u,C_0,\tilde{x},\tilde{w})$ says: (1) C_0 holds now; and (2) whenever C_0 holds then the application converges and again C_0 holds. We say C is *stateless except* $\tilde{x}$, iff: (1) each dereference $!y$ for $y \notin \{\tilde{x}\}$ occurs either in pre/post conditions of evaluation formulae or under $[!y]/\langle !y\rangle$; (2) (un)reachability predicates occur in pre/post conditions of evaluation formulae; and (3) evaluation formula never occur negatively nor under content quantifications. Above a formula C occurs *negatively* if it occurs in C_1 of $C_1 \supset C_2$ or in C of $\neg C$.

Proposition 6 (axiom for information hiding (1)) *Assume C_0 is stateless except $\tilde{x}$ and $x_i \notin \mathsf{fv}(C,C',E')$ for each $x_i \in \{\tilde{x}\}$. Then:*

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

is valid, where with m fresh and

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

The axiom (AIH_A) is used together with [*ConsEval*] in order to entail from E_1 to E_2 within the proof derivation. Its validity is proved using Proposition 1.

The axiom says that *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.* We note:

- C_0 being stateless except $\tilde{x}$, ensures that satisfiability of C_0 is not affected by state change except at that of x_i.
- $[!\tilde{x}]C$ says that whether C holds or not does not depend on $\tilde{x}$: literally it means that for each content of $\tilde{x}$ we have C holds.

Coming back to `compHide`, we can take C_0 to be $!x = 7$, $\tilde{w}$ to be the empty string, C and E' to be T and C' to be $z = (y > 7)$.

We now extend the axiom for general higher-order functions, after refining the invariant notation.

$$\mathsf{Inv}(u,C_0,\tilde{x},\tilde{r},\tilde{w}) = C_0 \wedge \forall y.\{C_0 \wedge \tilde{x}\#\tilde{r}y\}u\bullet y = z\{C_0 \wedge \tilde{x}\#z\tilde{w}\}@\tilde{w}\tilde{x}$$

where $\{\tilde{r}\tilde{w}yz\} \cap (\mathsf{fv}(C_0) \cup \{\tilde{x}\}) = \emptyset$ and y is of a base type. This time, since u may return an arbitrary (higher-order) function as its result z, we cannot guarantee that x is never exported. Thus we directly demand it, saying: *if $\tilde{w}$ is to be written, then the content of $\tilde{w}$ and a return value z should never reach $\tilde{x}$*, under the unreachability assumption $x\#\tilde{r}y$ (here $\tilde{r}$ indicate those values whose disjointness from $\tilde{x}$ is needed for the invariant).

Proposition 7 (axiom for information hiding (2)) *Let* (AIH) *be an axiom given by replacing $\mathsf{Inv}_A(u,C_0,\tilde{x},\tilde{w})$ with $\mathsf{Inv}(u,C_0,\tilde{x},\tilde{r},\tilde{w})$ in Proposition 6. Then* (AIH) *is valid.*

(AIH_A) and (AIH) assume that the invariant C_0 only talks about the content of x. This does not have to be so. We consider the extension of (AIH_A) in this regard: (AIH) is similarly generalised.

Proposition 8 (axiom for information hiding (3)) *Let* ($\mathsf{AIH}_{A\exists}$) *be the result of replacing, in* (AIH_A), *$\nu\#\tilde{x}$ with $\nu\#\tilde{x}.\exists\tilde{g}$ with $\tilde{g}$ only occurring in C_0. Then* ($\mathsf{AIH}_{A\exists}$) *is valid.*

Another simple extension, which we do not discuss here, allows a return value to be a composite one. Next, the following axiom stipulates how an invariant is *transferred* by functional applications.

Proposition 9 (invariant by application) *Suppose C_0 is stateless except $\tilde{x}$. Then the following is valid.*

$$(\mathsf{Inv}(f,C_0,\tilde{x},\tilde{r},\tilde{w}) \;\wedge\; \{\mathsf{T}\}g\bullet f = z\{\mathsf{T}\}) \supset \{C_0 \wedge \tilde{x}\#g\tilde{r}\}g\bullet f = z\{\tilde{x}\#z \wedge C_0\}.$$

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 keeps the local invariant.

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, which are found in Online Appendix [3].

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.

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

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 [27, 37, 41], $\mathtt{new}\ x := M\ \mathtt{in}\ N$, 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:

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

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.

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

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

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

To shed light on how the difference in sharing is captured in inferences, Appendix C in [3] 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 [39] from Introduction.

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

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 $!b = (!a)!$, C, E' be T, and C' be $y = x!$. Note C_0 is stateless except ab, cf. Prop.6.

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

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, 18]). $(\star)$ in Line 5 is by (AIH_A) in Proposition 6.

5.3 Information Hiding (2): Stored Circular Procedures

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

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

In [18], we have derived the following judgement.

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

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. $\{\mathsf{T}\}\mathtt{circFact}; !x :_u \{\mathit{CircFact}(u,x)\}@x$	
3. $\{\mathsf{T}\}\mathtt{circFact}; !x :_u \{CF_a \wedge CF_b\}@x$	(2, Conseq)
4. $\{x\#i\}\mathtt{circFact}; !x :_u \{x\#i \wedge CF_a \wedge CF_b\}@x$	(3, Inv-#)
5. $\{\mathsf{T}\}\mathtt{safeFact} :_u \{\nu\#x.(CF_a \wedge CF_b)\}@\emptyset$	(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. $\{\mathsf{T}\}\mathtt{safeFact} :_u \{\mathit{Fact}(u)\}@\emptyset$	(5, 6, ConsEval)

Line 1 is immediate. Line 2 is (5.1). Line 6, $(\star)$ is by (AIH_A), Proposition 6, setting $C_0 = !x = u$, $C, E' = \mathsf{T}$ and $C' = y = x!$. Note this example can again use (AIH_A) 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:

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

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.

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

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

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

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).

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

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 ($\mathsf{AIH}_{A\exists}$). 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).

5.4 Information Hiding (3): Higher-Order Invariant

We move to a local invariant for higher-order functions, taking a program which instruments an original program with a simple profiling, counting the number of times of invocation [43, p.104].

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

where for simplicity we assume y has a base type. Since x is never exposed, this program should behave precisely as f. Since f can return a higher-order value, we need to use (AIH). We use the

following assertions, assuming $x \notin \mathsf{fv}(C, C')$.

$$\begin{array}{rcl} E & = & \forall y.\{\mathsf{T}\}\, f \bullet y = z\{\mathsf{T}\}@\tilde{w} \ \wedge\ \forall y.\{C\}\, f \bullet y = z\{C'\}@\tilde{w} \\ E_0 & = & \forall y.\{C \ \wedge\ x\#\, fy\tilde{w}\}\, f \bullet y = z\{x\#z\tilde{w} \ \wedge\ C'\}@\tilde{w} \\ E_1 & = & \forall y.\{\mathsf{T}\}\, f \bullet y = z\{\mathsf{T}\}@\tilde{w} \ \wedge \\ & & \forall y.\{[!x]C \wedge x\#\, fy\tilde{w}\}u \bullet y = z\{C' \wedge x\#z\tilde{w}\}@x\tilde{w} \\ E_2 & = & \forall y.\{C\}u \bullet y = z\{C'\}@\tilde{w}. \end{array}$$

Our aim is to derive the following assertion.

$$\{E\}\ \texttt{profile} :_u \{E_2\} \tag{5.5}$$

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.* The derivation follows.

1. $\{\mathsf{T}\}x := !x + 1\{\mathsf{T}\}@x$	(Assign)
2. $\{[!x]C \wedge E \wedge x\#\, fy\tilde{w}\}\ x := !x + 1\ \{C \wedge E \wedge x\#\, fy\tilde{w}\}@x$	(Inv, Conseq)
3. $\{C \wedge E \wedge E_0 \wedge x\#\, fy\tilde{w}\}\ \ fy :_z \{C' \wedge x\#z\tilde{w}\}@\tilde{w}$	(App, Conseq)
4. $\{C \wedge E \wedge x\#\, fy\tilde{w}\}\ \ fy :_z \{C' \wedge x\#z\tilde{w}\}@\tilde{w}$	(3, Conseq)
5. $\{[!x]C \wedge E \wedge x\#\, fy\tilde{w}\}x := x + 1; fy :_z \{C' \wedge x\#z\tilde{w}\}@x\tilde{w}$	(2, 4, Seq)
6. $\{E\}\ \lambda y.(x := x + 1; fy) :_u \{E_1\}@\emptyset$	(5, Abs, Inv)
7. $\{E\}\texttt{profile}\{\nu\#x.E_1\}@\emptyset$	(LetRef)
8. $\{E\}m \bullet () = u\{\nu\#x.E_1\} \supset \{E\}m \bullet () = u\{E_2\}$	$(\star)$
9. $\{E\}\texttt{profile} :_u \{E_2\}@\emptyset$	(7, 8, ConsEval)

Line 2 uses: for any C, x we have $[!x][!x]C \equiv [!x]C$. Also by $[!x]E \equiv E$, and by $[!x]x\#\, fy\tilde{w} \equiv x\#\, fy\tilde{w}$ (by Proposition 2 (3)-5), [*Inv*] becomes applicable. Line 4 uses $E \supset E_0$ by the reachability axiom in Proposition 5 (setting $\tilde{r} = f$). Line 8 $(\star)$ uses (AIH), Proposition 6, setting C_0, E' to be T.

5.5 Information Hiding (4): Nested Local Invariant

The next example, which is from [19, 27], uses a function with local state as an argument to another function.

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

where $\Omega \stackrel{\text{def}}{=} \mu f.\lambda().(f())$ (note $\Omega()$ immediately diverges) and $\mathit{even}(n)$ tests evenness of n. 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` may as well satisfy the following judgement.

$$\{E \wedge C\}\ \texttt{MeyerSieber}\ \{C'\}@\tilde{w} \tag{5.6}$$

where we set, with $x, m \notin \mathsf{fv}(C, C')$:

$$\begin{array}{rcl} E & = & \forall f.(A \supset (\{\mathsf{T}\}g \bullet f\{\mathsf{T}\} \wedge \{C\}g \bullet f\{C'\}@\tilde{w})) \\ A & = & \{\mathsf{T}\}f \bullet () = z\{\mathsf{T}\}@\emptyset \end{array}$$

(Above the omission of an anchor of Unit type in E follows ***Convention*** in §2.2). (5.6) says that: *if feeding g with the observable behaviour of f as an argument always terminates and further satisfies $\{C\}g \bullet f\{C'\}@\tilde{w}$, then* `MeyerSieber` *starting from C also terminates with the final state C' and the write set $\tilde{w}$.*

For the derivation of (5.6) we use the following assertions (ε is the empty string).

$$\begin{array}{rcl} I & = & \mathsf{Inv}(f, \mathit{Even}(!x), x, \varepsilon, \varepsilon) \\ G_0 & = & \{\mathit{Even}(!x) \wedge x\#g\}g \bullet f\{\mathit{Even}(!x)\} \\ G_1 & = & \{\mathsf{T}\}g \bullet f\{\mathsf{T}\} \end{array}$$

The derivation of (5.6) requires

$$\{\mathit{Even}(!x) \wedge I \wedge G_1\}\ g \bullet f\ \{\mathit{Even}(!x)\} \supset \{\mathit{Even}(!x) \wedge G_0\}\ g \bullet f\ \{\mathit{Even}(!x)\}$$

for which we apply the axiom in Proposition 9.

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 [19] (cf.[10, 35, 36]). The following program generates a simple object each time it is invoked.

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

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`:

$$\{\mathsf{T}\}\ \texttt{cellGen} :_u \{\mathit{CellGen}(u)\} \tag{5.7}$$

where we set:

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

$\mathit{Cell}(o,x)$ 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 (5.7) 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 $\mathit{Cell}(o,x)$ (Appendix C in [3]).

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 [20, §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). The program then reads:

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

cyclesimple receives a node in a cyclic list. By its type, the content of the node is either $\texttt{in}_1(())$, a nil node, or $\texttt{in}_2(\langle y_1, y_2 \rangle)$, 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 data structures.

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

$\mathsf{path}(g,p,g')$ indicates that traversing p-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 on cyclesimple.

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

$\mathsf{isCycle}(g)$ says the node g is part of a cycle (its negation is linearity); 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 $\mathsf{cycleSimple}(u)$:

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

where we set:

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

(6.2) also uses an evaluation formula with a generalised write set defined by a predicate. The 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 all references that may be updated by this evaluation are within the set $\{w|E(w)\}$ (see [3, B.2] for precise semantics), allowing us to specify an unbounded number of references as a write set. Thus $\mathsf{cycleSimple}(u)$ says: *If the program u receives an argument z which is a node of a cyclic list, then it 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 distance above for induction for recursion [2].

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:

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

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. [41, §6]).

$$\begin{array}{rcl}\mathtt{treeCopy} \stackrel{\text{def}}{=} & \mu f.\lambda x^{\mathit{Tree}(\alpha)}.\mathtt{case}\ !x\ \mathtt{of}\\ & \quad \mathtt{in}_1(n) : \mathtt{ref}(\mathtt{inj}_1(n))\\ & \quad \mathtt{in}_2(\langle y_1,y_2\rangle) : \mathtt{ref}(\mathtt{inj}_2(\langle fy_1, fy_2\rangle))\end{array}$$

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's set, for brevity:

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

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

$$\begin{array}{l}\mathsf{path}(g,\varepsilon,g') \equiv g = g'\\ \mathsf{path}(g,l.p,g') \equiv \exists y_1y_2.(\mathsf{branch}(!g,y_1,y_2) \wedge \mathsf{path}(y_1,p,g'))\\ \mathsf{path}(g,r.p,g') \equiv \exists y_1y_2.(\mathsf{branch}(!g,y_1,y_2) \wedge \mathsf{path}(y_2,p,g'))\end{array}$$

The first clause says that the empty path leads from g to g; the second that $l.p$ 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.

$$\begin{array}{rcl}\mathsf{match}(g,p_1,p_2) & \equiv & \exists y.(\mathsf{path}(g,p_1,y) \wedge \mathsf{path}(g,p_2,y))\\ \mathsf{leaf}(g,p,x) & \equiv & \exists y.(\mathsf{path}(g,p,y) \wedge \mathsf{atom}(!y,x))\\ \mathsf{iso}(g,g') & \equiv & \forall p_1p_2.(\mathsf{match}(g,p_1,p_2) \equiv \mathsf{match}(g',p_1,p_2))\\ & \wedge & \forall px.(\mathsf{leaf}(g,p,x) \equiv \mathsf{leaf}(g',p,x))\end{array}$$

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:

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

$\mathsf{tree}\,(g)$ 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"). This gives a basis for inductive reasoning. Note if g is a tree then $\mathsf{distance}(g,p,g')$ is equivalent to $\mathsf{path}(g,p,g')$.

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

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

which stands for, with i fresh:

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

indicating the set $\{x \mid E(x)\}$ of references are newly generated. We can now assert for treeCopy, naming it u, with g typed as $\mathit{Tree}(\alpha)$:

$$\begin{array}{l}\mathsf{treecopy}[\alpha](u)\ =\\ \{\mathsf{tree}\,(g)\}u \bullet g = g'\{\nu\#\{h\,|\,\mathsf{reach}(g',h)\}.\,\mathsf{iso}(g,g')\}@\emptyset\end{array}$$

where $\mathsf{reach}(g',h)$ stand for $\exists p.\mathsf{path}(g',p,h)$ and g is of type $\mathit{Tree}(\alpha)$. The assertion reads:

> *Whenever u is invoked with a tree g of type $\mathit{Tree}(\alpha)$, 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, (6.2) 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 [41, § 6]:

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

Above we set $\mathsf{disjoint}(x,y) \equiv \neg\exists p.(\mathsf{path}(x,p,y) \vee \mathsf{path}(y,p,x))$. The derivation of the judgement $\{\mathsf{T}\}\ \mathtt{treeCopy} :_u \{\mathsf{treecopy}(u)\}$ uses $\mathsf{distance}$ for induction and is straightforward.

6.3 Dags and Graphs

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

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

$\mathsf{dag}\,(g)$ asserts 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 $\mathsf{tree}\,(g) \supset \mathsf{dag}\,(g)$. A simple extension of `treeCopy` to create a fresh duplicate of an original dag, called `dagCopy`, is given in Appendix D in [3]. The program satisfies $\{\mathsf{T}\}\mathtt{dagCopy} :_u \{\mathsf{dagcopy}[\alpha](u)\}$, where $\mathsf{dagcopy}[\alpha](u)$ is given as, with g typed $\mathit{Tree}(\alpha)$:

$$\{\mathsf{dag}\,(g)\}u \bullet g = g'\{\mathsf{v}\#\{h \mid \mathsf{reach}(g',h)\}.\,\mathsf{iso}(g,g')\}@\emptyset$$

The derivation of $\{\mathsf{T}\}\mathtt{dagCopy} :_u \{\mathsf{dagcopy}(u)\}$ is given in [2], which uses the same $\mathsf{distance}$ predicate in induction for recursion.

Finally a program `graphCopy` (given in Appendix D in [3]) makes a fresh duplicate of an arbitrary datum g of type $\mathit{Tree}(\alpha)$, including those with circularity. The program satisfies the judgement $\{\mathsf{T}\}\mathtt{graphCopy} :_u \{\mathsf{graphcopy}[\alpha](u)\}$ where $\mathsf{graphcopy}[\alpha](u)$ is:

$$\forall g^{\mathit{Tree}(\alpha)}.\{\mathsf{T}\}u \bullet g = g'\{\mathsf{v}\#\{h \mid \mathsf{reach}(g',h)\}.\,\mathsf{iso}(g,g')\}@\emptyset$$

The assertion $\mathsf{graphcopy}[\alpha](u)$ says: *When fed with any graph of type Tree*(α)*, u creates its fresh duplicate, and does nothing else.* This assertion 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 include an assertion for a polymorphic variant of $\mathsf{graphcopy}[\alpha](u)$ [17].

Proposition 10 *Fix α. Then each of the following implications is valid and strict.*

$$\begin{array}{rcccl} \forall \mathrm{X}.\mathsf{graphcopy}[\mathrm{X}](u) & \supset & \mathsf{graphcopy}[\alpha](u) & \supset & \mathsf{dagcopy}[\alpha](u) \\ & \supset & \mathsf{treecopy}[\alpha](u) & \supset & \mathsf{treesep}[\alpha](u). \end{array}$$

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 [4, 5]. Target programming languages of our preceding logics [6, 16–18] do not include local state. As is well-known [19, 23, 24, 27, 37, 39], 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 [27] presents many interesting examples and reasoning principles based on denotational semantics. Mason and Talcott [23, 24] 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 [37, 39, 43] 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 [39]. Sumii and Pierce [44] 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 [44], Koutavas and Wand [19] 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 2 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 [22].

Hoare Logics (1): Local Variables and ML-like Languages. To our knowledge, Hoare and Wirth [15] 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 [15] 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 [5, 31], *Extended ML* [42] 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* [45] 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 [45]. 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 [32] first presented the use of reachability predicates for reasoning about linked lists. Based on [32], Lahiri and Qadeer [20] 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 [20]) 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. An interesting question is whether we can apply their ideas on effective axiomatisation to a large class of mutable data structures treatable in our method.

Hoare Logics (3): Separation Reynolds, O'Hearn and others [9, 34, 41] 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 [41]), 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:

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

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 highly 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 10 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, 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 [41] criticises the use of reachability for describing data structure, taking the in-place reversal of a linear list as an example. As discussed in Section 6, a tractable reasoning is possible for such examples 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.

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 [12, 38] extend First-Order Logic is extended with constructs to reason about freshness of names based on the theory of permutations. The key syntactic additions are the (interdefinable) "fresh" quantifier $\mathcal{N}$ and the freshness predicate #, mediated by the swapping (finite permutation) predicate. Miller and Tiu [28] 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 3, 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, 25, 33]. 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*.

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] On-line appendix. www.doc.ic.ac.uk/~yoshida/local.

[4] Standard ML home page. http://www.smlnj.org.

[5] The Caml home page. http://caml.inria.fr.

[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.

[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.

[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] M. Gabbay and A. Pitts. A New Approach to Abstract Syntax Involving Binders. In *Proc. LICS '99*, pages 214–224, 1999.

[13] C. A. R. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.

[14] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[15] C. A. R. Hoare and N. Wirth. Axiomatic semantics of Pascal. *ACM TOPLAS*, 1(2):226–244, 1979.

[16] K. Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.

[17] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202. ACM, 2004.

[18] 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.

[19] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. POPL*, 2006.

[20] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, pages 115–126. ACM, 2006.

[21] P. Landin. A correspondence between algol 60 and church's lambda-notation. *Comm. ACM*, 8:2, 1965.

[22] 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.

[23] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theor. Comput. Sci.*, 105(2):167–215, 1992.

[24] I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *LICS*, pages 186–197, 1992.

[25] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 159:200–227, May 2005.

[26] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.

[27] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL'88*, 1988.

[28] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, to appear.

[29] R. Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971.

[30] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1):1–77, 1992.

[31] R. Milner, M. Tofte, and R. W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[32] G. Nelson. Verifying reachability invariants of linked structures. In *POPL '83*, pages 38–47. ACM Press, 1983.

[33] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, 2006.

[34] P. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, 2004.

[35] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[36] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for objecr-oriented programming. *JFP*, 4(2):207–247, 1993.

[37] 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.

[38] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[39] 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.

[40] J. C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.

[41] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, 2002.

[42] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *POPL'85*, pages 67–77. ACM, 1985.

[43] I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, Dec. 1994.

[44] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *POPL'04*, pages 161–172. ACM Press, 2004.

[45] 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.

Figure 1 Proof Rules

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

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

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

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

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

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

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

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

$$[Deref]\ \frac{\{C\}\ M :_m \{C'[!m/u]\}}{\{C\}\ !M :_u \{C'\}}$$

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

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

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

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

A. Appendix: Proof Rules

Figure 1 presents all compositional proof 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. $\mathsf{dom}(\Theta)\cap(\mathsf{dom}(\Gamma,\Delta)\cup\{u\})=\emptyset$. $C^{\frown\tilde{x}}$ indicates $\mathsf{fv}(C)\cap\{\tilde{x}\}=\emptyset$.

In [*Abs*,*Rec*], A,B denote *stateless* formulae except ε (empty string), as given in §4 (just before Proposition 6, page 6). As examples, $\{!y=i\}u\bullet()\{!y=i+1\}\wedge\langle!x\rangle!x=3$ is stateless: but neither $!x=1$ nor $\langle!x\rangle!y=1$ is.

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

$$C\{\!|e_2/!e_1|\!\} \stackrel{\text{def}}{=} \exists m.(\langle!e_1\rangle(C\wedge!e_1=m)\ \wedge\ m=e_2).$$

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.

We also use structural rules in the reasoning. For a summary of significant structural rules, see [2], Appendix B.

B. Appendix: Models

B.1 Observational Congruence

Programs Write $(\nu\tilde{l})(M,\sigma) \Downarrow (\nu\tilde{l'})(V,\sigma')$ for $(\nu\tilde{l})(M,\sigma) \rightarrow^* (\nu\tilde{l'})(V,\sigma')$; and $(\nu\tilde{l})(M,\sigma) \Downarrow$ for $(\nu\tilde{l})(M,\sigma) \Downarrow (\nu\tilde{l'})(V,\sigma')$ for some $(\nu\tilde{l'})(V,\sigma')$. Let $\Gamma;\Delta \vdash M_{1,2} : \alpha$. 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 closing context $C[\,\cdot\,]$ of type Unit which is typable under Δ and in which no labels from $\tilde{l}_{1,2}$ occur, we have:

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

We often write $(\nu\tilde{l_1})(M_1,\sigma_1) \cong (\nu\tilde{l_2})(M_2,\sigma_2)$, leaving type information implicit.

Models Given models $\mathcal{M}_i^{\Gamma;\Delta} = (\nu\tilde{l_i})(\{y_i : V_{i1},..,y_i : V_{in}\},\sigma_i)$ for $i = 1,2$, we set $\Gamma;\Delta \vdash \mathcal{M}_1 \approx \mathcal{M}_2$ iff

$$(\nu\tilde{l_1})(\langle V_{11},..,V_{1n}\rangle,\sigma_1) \cong (\nu\tilde{l_2})(\langle V_{21},..,V_{2n}\rangle,\sigma_2)$$

B.2 Semantics

Let $\Gamma;\Delta \vdash e : \alpha$, $\Gamma;\Delta \vdash \mathcal{M}$ and $\mathcal{M} = (\nu\tilde{l})(\xi,\sigma)$. Then the *interpretation of e under* (ξ,σ), denoted $[\![e]\!]_{\xi,\sigma}$ is inductively given by:

$$\begin{array}{ll} [\![x]\!]_{\xi,\sigma} = \xi(x) & [\![!e]\!]_{\xi,\sigma} = \sigma([\![e]\!]_{\xi,\sigma}) \\ [\![()]\!]_{\xi,\sigma} = () \quad [\![\mathsf{n}]\!]_{\xi,\sigma} = \mathsf{n} & [\![\mathsf{b}]\!]_{\xi,\sigma} = \mathsf{b} \quad [\![l]\!]_{\xi,\sigma} = l \\ [\![\mathsf{op}(\tilde{e})]\!]_{\xi,\sigma} = \mathsf{op}([\![\tilde{e}]\!]_{\xi,\sigma}) & [\![\langle e,e'\rangle]\!]_{\xi,\sigma} = \langle [\![e]\!]_{\xi,\sigma}, [\![e']\!]_{\xi,\sigma}\rangle \\ [\![\pi_i(e)]\!]_{\xi,\sigma} = \pi_i([\![e]\!]_{\xi,\sigma}) & [\![\mathtt{inj}_i(e)]\!]_{\xi,\sigma} = \mathsf{inj}_i([\![e]\!]_{\xi,\sigma}) \end{array}$$

Then we define semantics of the assertions as follows (the new notations are illustrated below): All omitted cases are by de Morgan duality. Let u,u',u'' be fresh.

- $\mathcal{M} \models e_1 = e_2$ if $\mathcal{M}[u : e_1] \approx \mathcal{M}[u : e_2]$.
- $\mathcal{M} \models C_1 \wedge C_2$ if $\mathcal{M} \models C_1$ and $\mathcal{M} \models C_2$.
- $\mathcal{M} \models \neg C$ if not $\mathcal{M} \models C$.
- $\mathcal{M} \models \forall x^{\alpha}.C$ if (1) $\forall e.(\mathcal{M}[x : e] \models C)$ and $\forall V.(\mathcal{M}[x : V] \models C)$ when α is any type; and (2) $\forall \mathcal{M}'.((\nu l)(\mathcal{M}'/x) \approx \mathcal{M} \supset \mathcal{M}' \models C)$ s.t. $\mathcal{M}'(x) = l$ when α is a reference type.
- $\mathcal{M} \models \forall \mathrm{X}.C$ if for all closed type α, $\mathcal{M}\cdot \mathrm{X} : \alpha \models C$.
- $\mathcal{M} \models [!x]C$ if $\forall \mathcal{M}'.(\mathcal{M} \overset{[!x]}{\approx} \mathcal{M}' \supset \mathcal{M}' \models C)$.
- $\mathcal{M} \models \{C\}e \bullet e' = x\{C'\}$ if, whenever $\mathcal{M}[u{:}N] \Downarrow \mathcal{M}_0$ and $\mathcal{M}_0/u \models C$ for some N, we have $\mathcal{M}[x : L] \Downarrow \mathcal{M}' \models C'$ where we set $L \overset{\text{def}}{=} \mathtt{let}\ u = e\ \mathtt{in}\ u' = e'\ \mathtt{in}\ \mathtt{let}\ u'' = N\ \mathtt{in}\ uu'$.
- $\mathcal{M} \models e_1 \hookrightarrow e_2$ if $(\nu\tilde{l})(\xi,\sigma) \approx \mathcal{M}$ implies $[\![e_2]\!]_{\xi,\sigma} \in \mathsf{ncl}(\mathsf{fl}([\![e_1]\!]_{\xi,\sigma}),\sigma)$

Above we use the following notations (assuming well-typedness): $\mathcal{M}[u : N] \Downarrow \mathcal{M}'$ appears in § 3.1. $\mathcal{M}[e \mapsto V]$ denotes the obvious substitution (with e of a reference type). $\mathcal{M}/u = (\nu\tilde{l})(\xi \cdot u : V,\sigma)$ if $\mathcal{M} = (\nu\tilde{l})(\xi \cdot u : V,\sigma)$; otherwise $\mathcal{M}/u = \mathcal{M}$. For $\mathcal{M}_{1,2}$ of the same type, $\mathcal{M}_1 \overset{[!x]}{\approx} \mathcal{M}_2$ iff $\forall V.(\mathcal{M}_1[x \mapsto V] \approx \mathcal{M}_2[x \mapsto V])$.

In the satisfaction of $\forall x^{\alpha}.C$ above, we consider the case the location is hidden. In $\forall \mathrm{X}.C$, we augment a model $\mathcal{M}$ with a map from type variables to closed types.

For evaluation formula, the defining clause says:

> *In any initial hypothetical state satisfying C evolvable from the current state, the application of e_1 to e_2 (both evaluated in the current state) terminates and the result z and the final state satisfy C'.*

Following [6, 18], we consider hypothetical initial state since a function can be invoked any time later, not only at the present state. The satisfaction of its generalised located assertion (which subsumes its finite counterpart):

$$\mathcal{M} \models \{C\}e \bullet e' = x\{C'\}@\{z\,|\,E(z)\}$$

iff it satisfies the clause of the evaluation formula above and the following, letting $\mathcal{M}_0 \overset{\text{def}}{=} (\nu\tilde{l})(\xi,\sigma_0)$ and $\mathcal{M}' \approx (\nu\tilde{l}\tilde{l'})(\xi,\sigma')$, $\forall\tilde{V}.((\nu\tilde{l})(\xi,\sigma_0[\tilde{l_1} \mapsto \tilde{V}]) \approx (\nu\tilde{l}\tilde{l'})(\xi,\sigma'[\tilde{l_1} \mapsto \tilde{V}]))$ where $l \in \{\tilde{l_1}\}$ iff $(\nu\tilde{l})(\xi \cdot z : l,\sigma_0) \models E$. This says:

> *The value stored at each location z satisfying $\neg E(z)$ in $\mathcal{M}_0$, is exactly preserved when the application at $\mathcal{M}_0$ results in $\mathcal{M}'$, taking $\mathcal{M}'$ up to $\approx$.*

For formal details, see [2, C.2].

C. Derivations for Examples in Section 5

This appendix lists the derivations omitted in Section 5.

C.1 Derivation for [*LetRef*]

We can derive [*LetRef*] as follows.

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

Lines 5 and 8 use the standard logical law (discussed below). Lines 4 and 7 use the following derived/admissible proof rules:

$$[\mathit{Subs}]\ \frac{\{C\}\ M :_u \{C'\} \quad u \notin \mathsf{fpn}(e)}{\{C[e/i]\}\ M :_u \{C'[e/i]\}}$$

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

[*LetOpen*] opens the "scope" of $\tilde{y}$ to N. The crucial step is Line 5, which turns freshness "#" into locality "ν" through the standard law of equality and existential, $C \equiv \exists y.(C \wedge x = y)$ with y fresh.

C.2 Derivation for `IncUnShared`

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

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

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.

Figure 2 `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. $\{n \geq 1 \supset \mathit{IsEven}'(!y,gh,n-1,xy)\}\ \mathtt{if}\ n=0\ \mathtt{then\ f\ else\ not}((!y)(n-1)) :_m \{z = \mathit{Odd}(n) \wedge !x = g \wedge !y = h\}@\emptyset$ (IfH)

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

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. $\{\mathsf{T}\}\ y := M_y\{\ \forall gh, n \geq 1.(\mathit{IsOdd}(g,gh,n-1,xy) \supset \mathit{IsEven}(!y,gh,n,xy)) \wedge !y = h\}@y$ (Similar with Line 6)

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

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

Let us define:

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

We also use:

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

We use the following derived rules and one standard structure rule appeared in [18].

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

$$[\mathit{IfH}] \qquad \frac{\{C \wedge e\}\, M_1 :_u \{C'\} \ \ \{C \wedge \neg e\}\, M_2 :_u \{C'\}}{\{C\}\ \mathtt{if}\ e\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u \{C'\}}$$

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

Figure 2 lists the derivation for `MutualParity`. In Line 5, we use the following axiom for the evaluation formula from [18]:

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

where A is stateless formula and we here set $A = \mathit{IsEven}(h,gh,n-1,xy)$. Line 9 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:

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

The derivation is similar to `safeFact`.

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

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

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

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

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

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

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

C.4 Derivation for Meyer-Seiber

For the derivation of (5.6) we use (ε is the empty string): $I = \mathsf{Inv}(f, \mathit{Even}(!x), x, \varepsilon, \varepsilon)$, $G_0 = \{\mathit{Even}(!x) \wedge x\#g\}g \bullet f\{\mathit{Even}(!x)\}$, and $G_1 = \{\mathsf{T}\}g \bullet f\{\mathsf{T}\}$. The derivation follows. Below $M_{1,2}$ is the

body of the first/second lets, respectively.

1. $\{Even(!x) \wedge G_0\}\ gf\ \{Even(!x)\}$	(App)
2. $\{Even(!x) \wedge I \wedge G_1\}\ gf\ \{Even(!x)\}$	(1, Conseq)
3. $\{E \wedge [!x]C \wedge I \wedge x\#g\}\ gf\ \{C'\}@\tilde{w}x$	(App)
4. $\{E \wedge [!x]C \wedge I \wedge x\#g\}\ gf\ \{Even(!x) \wedge C'\}@\tilde{w}x$	(2, 3, Conj)
5. $\{Even(!x) \wedge C'\}$ `if` $even(!x)$ `then` $()$ `else` $\Omega()$ $\{C'\}@\emptyset$	(If)
6. $\{E \wedge [!x]C \wedge I \wedge x\#g\} M_2 \{C'\}@\tilde{w}x$	(4, 5, Seq)
7. $\{Even(!x)\}\lambda().x :=!x+2 :_f \{I\}@\emptyset$	(Abs etc.)
8. $\{E \wedge [!x]C \wedge Even(!x) \wedge x\#g\}\ M_1\ \{C'\}@\tilde{w}x$	(7, 6, LetRef)
9. $\{E \wedge C\}\ 0 :_m\ \{E \wedge C \wedge Even(m)\}@\emptyset$	(Const)
10. $\{E \wedge C\}$ `MeyerSieber` $\{C'\}@\tilde{w}$	(9, LetRef)

Line 2 uses the axiom in Proposition 9. Line 4 uses the standard structural rule. Line 10 cancels $[!x]$ from $[!x]C$ which is possible since m does not occur in C.

C.5 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.

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

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:

$$\{\mathsf{T}\}\ \texttt{cell} :_o\ \{I_0 \wedge G_1 \wedge G_2 \wedge E'\} \qquad \text{(C.1)}$$

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

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

which will become, after taking off the invariant I_0:

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

Note I_0 is stateless except 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. We set $\texttt{cell}' \stackrel{\text{def}}{=} \texttt{let}\ y = \texttt{ref}(0)\ \texttt{in}\ \texttt{cell}$ below.

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. $\{\mathsf{T}\}\ \texttt{let}\ x_1 = z\ \texttt{in}\ \texttt{cell}' :_o \{\nu\#x_1.(I_0 \wedge G_1 \wedge G_2) \wedge E'\}$	(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. $\{\mathsf{T}\}\ \texttt{let}\ x_{0,1} = z\ \texttt{in}\ \texttt{cell}' :_o\ \{\nu\#x.(\ G_1' \wedge G_2' \wedge E'\ )\}$	(LetRef)
6. $\{\mathsf{T}\}\ \texttt{cellGen} :_u\ \{CellGen(u)\}$.	(Abs)

D. Algorithms for Dag and Graph

This appendix lists the programs for the dag copy and graph copy. The detailed derivation can be found in [2]. First we show the algorithm for the dag copy.

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

When the 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 assumes, for brevity, a predefined data type for a table (which in fact is 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, as well as the constant $\emptyset$ for the empty table.

Next we present a copying algorithm which works with any graph of *Tree*-type, including those with circular edges.

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

where $\texttt{tmp} = \texttt{inj}_1(0)$. $\texttt{graphCopy}^{\alpha}$ is essentially identical with $\texttt{dagCopy}^{\alpha}$ 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.

Finally the polymorphic version of $\texttt{graphCopy}^{\alpha}$ is simply given by $\Lambda X.\texttt{graphCopy}^{X}$, using the standard universal type abstraction.