Local State in Hoare Logic for Imperative Higher-Order Functions

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

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

Abstract. We introduce an extension of Hoare logic for imperative higher-order functions with local state. Local state may be generated dynamically and exported outside its scope, may store higher-order functions, and may be used to construct complex shared mutable data structures. The induced behaviour is captured with a first order predicate which asserts reachability of reference names. 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 manipulating dynamically generated local state. Axioms for reachability play a central role for reasoning about the examples.

Table of Contents

1 Introduction

Local State in Imperative Higher-Order Programming. New reference generation in ML-like languages [1, 2] is a powerful programming primitive. First, a newly created reference is hidden from the outside, enhancing modularity through localisation of read/write effects. Consider the following program:

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

We use standard notation [31]: in particular, $\mathtt{ref}(M)$ returns a fresh reference whose content is the value M evaluates to. $!x$ is dereferencing of an imperative variable x. When the anonymous function in `Inc` is invoked, it increments the content of its local variable x, and returns the new content. Thus 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.

The use of local state is also a source of representation independence. As an example,

$$\mathtt{Inc2} \stackrel{\text{def}}{=} \mathtt{let}\ x,y = \mathtt{ref}(0)\ \mathtt{in}\ \lambda().(x := !x + 1; y := !y + 1;\ (!x + !y)/2) \tag{1.2}$$

realises the same observable behaviour as `Inc`. But if x or y is not local, they clearly have distinct visible behaviours.

Freshness of names of imperative variables generated by programs is a fundamental element of the semantics of local state. Consider the following program:

$$\lambda x.\mathtt{let}\ y = \mathtt{ref}(1)\ \mathtt{in}\ \mathtt{if}\ x = y\ \mathtt{then}\ 0\ \mathtt{else}\ 100 \tag{1.3}$$

This function always returns 100, since a name fed by the environment (x) and the name of a newly generated location (y) cannot be identical. This freshness guarantees locality of state by prohibiting (direct) access to that state from the outside of its scope. This facility can be used for insulating dynamically generated data structures from undesirable interference by other programs.

As another example, consider the following program with stored higher-order procedures [33, § 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 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 $!b$. Note that if we replace $b :=!a$ at $l.2$ by $b := \mathtt{Inc}$, two separate instances of `Inc` are assigned to a and b, and the final result is 2. Controlling sharing by combining scope extrusion and local state is a foundation of many programming disciplines, including manipulation of dynamically generated mutable data structures (as shown in § 7), but it severely complicates reasoning even for relatively simple commands.

A further example demonstrates the power of combining stored functions and local references. We consider a factorial program which realises a recursion by circular references, an idea due to Landin [21].

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

This program calculates the factorial of n. But since x is still 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. In the presence of local state, we can hide x to avoid unexpected interference.

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

(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 the program behaves as the pure factorial function. The potential distance between the extensional and internal behaviour of a program with local state can be exploited for modular programming. But this distance also causes difficulties in reasoning, since it makes correspondence between programs' syntactic structures and their behaviours subtle to establish [20, 24, 32, 33].

Program Logic and Local State. This paper proposes a simple extension of Hoare logic for treating higher-order imperative programs with local state. Hoare logic has been highly successful in software engineering, including verification, rigorous software development and testing. However there have been three open issues which make it difficult to extend Hoare logic to imperative higher-order programming languages such as ML.

- Higher-order functions, including stored ones.
- General forms of aliasing induced by arbitrary nested reference types.
- Treatment of dynamically generated local state and its scope exclusion.

The first is a primary source of the expressive power of higher-order imperative programs (as seen in `circFact`). For the second point, since reference types can occur in other types, we can use references as parameters of function calls, return values and content of references and other data structures, causing potential aliasing. These three are fundamental elements of practical typed higher-order programming, but have defied clean logical treatment.

In preceding studies, the present authors have proposed Hoare logics which capture the first two features [3, 17–19]. The resulting logics enjoy a tight link with standard observational semantics in that assertions distinguish programs' behaviour just as the contextual behavioural equivalence does. As already stressed in the context of Hoare logics [10, 16], this property, observational completeness, is important when, for example, we wish to use compositional program logics together with other mathematical tools based on a firm semantic basis.

On the basis of our preceding works [3, 17–19], this paper introduces a compositional program logic for higher-order functions with dynamically generated local state. The logic enjoys observational completeness and offers a uniform basis for asserting and reasoning about the general class of dynamically generated mutable data structures such as graphs storing higher-order functions at their nodes. Its proof system, combined with axioms for reachability, enables precise compositional verification of subtle programming examples involving higher-order functions and local state, including those discussed above. To our knowledge, this is the first time a Hoare-like program logic for imperative higher-order functions with ML-like dynamically generated references in full type hierarchy are developed.

Outline. In the rest of the paper, Section 2 reviews the target programming language and its contextual equivalence. Section 3 introduces the assertion language and illustrates basic ideas of the assertion language, especially in the way it expresses local state, through simple examples. Section 4 introduces semantics of assertions. Section 5 presents the proof rules, including both compositional proof rules and structural rules. Section 6 proves the validity of the axioms, establishes soundness and and observational completeness of the logic. Section 7 offers examples of reasoning about programs. Section 8 gives comparisons with related work and concludes with further topics. Some auxiliary definitions and proofs are relegated to Appendix.

2 A Programming Language

2.1 Syntax and Reduction

As our target programming language, we use call-by-value PCF with unit, sums and products, augmented with imperative constructs [12,31]. Let $x,y,\ldots$ range over an infinite set of variables, often called *names*. Then types $(\alpha,\beta,\ldots)$, values $(V,W,\ldots)$ and programs $(M,N,\ldots)$ are given by the following grammar.

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

Above we use the standard notation [12,31,43]. The binding is induced in the standard way. Programs are considered up to the corresponding α-equality.

The language is identical with the one used in [3], except for the inclusion of a construct for reference generation. Constants $(\mathtt{c},\mathtt{c}',\ldots)$ include the unit $()$, natural numbers $\mathtt{n}$, booleans $\mathtt{b}$ (either truth $\mathtt{t}$ or false $\mathtt{f}$), and *locations* $(l,l',\ldots)$. Locations appear only at run-time. $\mathtt{op}(\tilde{M})$ (where $\tilde{M}$ is a vector of programs) is a standard n-ary arithmetic or boolean operation, such as $+$, $-$, $\times$, $=$ (equality of two numbers), $\neg$ (negation), $\wedge$ and $\vee$. We freely use obvious shorthands like $M;N$ and $\mathtt{let}\ x=M\ \mathtt{in}\ N$. Type annotations are often omitted from programs, writing e.g. $\lambda x.M$.

Since all constructs are standard, we leave their illustration to well-known textbooks [12,31,43], except for the focus of the present study. The reference generation, $\mathtt{ref}(M)$, behaves as:

> First M of type α is evaluated and becomes a value V; then a *fresh* local reference l of type $\mathsf{Ref}(\alpha)$ with initial content V is generated.

Then another form of new name generation [24,36], $\mathtt{new}\ x := M\ \mathtt{in}\ N$, behaves as follows:

> First, M of type α is evaluated; Then, assuming it terminates and becomes a value V, it generates a *fresh* local reference of type $\mathsf{Ref}(\alpha)$ with initial content V; finally N (which may possibly use x) is evaluated.

Note that

$$\mathtt{new}\ x := M\ \mathtt{in}\ N \stackrel{\text{def}}{=} \mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N \tag{2.1}$$

$$\mathtt{ref}(M) \stackrel{\text{def}}{=} \mathtt{new}\ x := M\ \mathtt{in}\ x \tag{2.2}$$

In this full version, we shall use both constructs. We formalise this and other behaviour of programs using the standard (one-step, call-by-value) reduction [12,31,43].

A *store* $(\sigma,\sigma',\ldots)$ is a finite map from locations to values. We write $\mathsf{dom}(\sigma)$ for the domain of σ and $\mathsf{fl}(\sigma)$ for locations occurring in both the domain and co-domain of σ.

$\sigma[l \mapsto V]$ denotes the store which maps localtion l to V and otherwise agrees with σ, for each $l \in \mathsf{dom}(\sigma)$. An *open configuration* is a pair of a closed program and a store, written $(M,\ \sigma)$. A *configuration* is an open configuration (M,σ) combined with a set of locations of $\mathsf{dom}(\sigma)$, written $(\nu\,\tilde{l})(M,\sigma)$ (the order of $\tilde{l}$ does not matter). We call $(\nu\,\tilde{l})$ in $(\nu\,\tilde{l})(M,\sigma)$, the latter's ν*-binder*, and consider locations in $\tilde{l}$ occur bound in $(\nu\,\tilde{l})(M,\sigma)$. Configurations are always considered up to the induced α-equality, including that on programs and stores. We assume the standard bound name convention for configurations. Open configurations are considered as configurations with the empty names in their ν-binders, i.e. we write (M,σ) for $(\nu\,\varepsilon)(M,\sigma)$ with ε denoting the empty string.

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

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

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

$$\begin{array}{rcl}
(\lambda x.M)V & \rightarrow & M[V/x] \\
\pi_1(\langle V_1,V_2\rangle) & \rightarrow & V_1 \\
\mathtt{if}\ \mathtt{t}\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 & \rightarrow & M_1 \\
(\mu f.\lambda g.N)W & \rightarrow & N[W/g][\mu f.\lambda g.N/f]
\end{array}$$

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

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

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

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

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

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

$$\frac{(\nu\tilde{l}_1)(M,\sigma) \rightarrow (\nu\tilde{l}_2)(M',\sigma')}{(\nu\tilde{l}\tilde{l}_1)(\mathcal{E}[M],\sigma) \rightarrow (\nu\tilde{l}\tilde{l}_2)(\mathcal{E}[M'],\sigma')}$$

where $\tilde{l}$ are disjoint from both $\tilde{l}_1$ and $\tilde{l}_2$, $\mathcal{E}[\cdot]$ is the left-to-right evaluation context (with eager evaluation), inductively given by:

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

Some notations:

– $(\nu\tilde{l})(M,\sigma)\Downarrow$ stands for $(M,\sigma) \longrightarrow^* (\nu\tilde{l}')(V,\sigma')$, for some $\tilde{l}'$, V and σ'.
– $(\nu\tilde{l})(M,\sigma)\Uparrow$ iff $(\nu\tilde{l})(M,\sigma) \longrightarrow^n$ for each natural number n, that is iff $(\nu\tilde{l})(M,\sigma)\Downarrow$ does not hold.
– If $(\nu\tilde{l})(M,\sigma)$ and l_i does not appear in M and $l_i \in \mathsf{dom}(\sigma)$, then we write M as $(\nu\tilde{l})(M,\sigma)$.

Example 2.1 (reduction, 1) Let us follow the reduction of `IncShared`. We set:

– $\texttt{IncShared}' \overset{\text{def}}{=} b :=!a; l_1 := (!a)(); l_2 := (!b)(); (!l_1) + (!l_2)$,
– $\texttt{IncShared}'' \overset{\text{def}}{=} l_1 := (!a)(); l_2 := (!b)(); (!l_1) + (!l_2)$, and
– $\texttt{IncShared}''' \overset{\text{def}}{=} l_2 := (!b)(); (!l_1) + (!l_2)$.

Further we set $W \overset{\text{def}}{=} \lambda().l :=!l+1; !l$. Omitting irrelevant values in a store, the reduction of `IncShared` follows (we recall $M;N$ stands for $(\lambda().N)M$ which reduces as $(\lambda().N)() \longrightarrow N$: for legibility this reduction is not counted below).

$$\begin{array}{l}
(\texttt{IncShared},\ \emptyset)\\
\longrightarrow (\nu l)(a := W; \texttt{IncShared}',\ \{l \mapsto 0,\ a,b,l_1,l_2 \mapsto \ldots\})\\
\longrightarrow (\nu l)(\texttt{IncShared}',\ \{x \mapsto 0,\ a \mapsto W,\ b,l_1,l_2 \mapsto \ldots\})\\
\longrightarrow (\nu l)(b := W; \texttt{IncShared}'',\ \{l \mapsto 0,\ a \mapsto W,\ b,l_1,l_2 \mapsto \ldots\})\\
\longrightarrow (\nu l)(\texttt{IncShared}'',\ \{l \mapsto 0,\ a,b \mapsto W,\ l_1,l_2 \mapsto \ldots\})\\
\longrightarrow (\nu l)(z_1 := 1; \texttt{IncShared}''',\ \{l \mapsto 1,\ a,b \mapsto W,\ l_1,l_2 \mapsto \ldots\})\\
\longrightarrow (\nu l)(\texttt{IncShared}''',\ \{l \mapsto 1,\ a,b \mapsto W,\ l_1 \mapsto 1,\ l_2 \mapsto \ldots\})\\
\longrightarrow (\nu l)(l_2 := 2; (!l_1)+(!l_2),\ \{l \mapsto 1,\ a,b \mapsto W,\ l_1 \mapsto 1,\ l_2 \mapsto \ldots\})\\
\longrightarrow (\nu l)((!l_1)+(!l_2),\ \{l \mapsto 1,\ a,b \mapsto W,\ l_1 \mapsto 1,\ l_2 \mapsto 2\})\\
\longrightarrow (\nu l)(3,\ \{l \mapsto 1,\ a,b \mapsto W,\ l_1 \mapsto 1,\ l_2 \mapsto 2\})
\end{array}$$

Observe l is shared throughout the reduction.

Example 2.2 (reduction, 2) The following example indicates the expressive power of new reference generation. We first introduce the following notation:

$$\mathsf{cons}(x,y) \overset{\text{def}}{=} \texttt{new}\ h := x\ \texttt{in}\ \texttt{new}\ t := y\ \texttt{in}\ \langle h,t\rangle$$

that is $\mathsf{cons}(x,l)$ generates two new references, h and t, with their respective content x (say a natural number) and y (which is either another pair or a terminator, the latter we write nil), and constructs a pair of these two fresh references. Since both components are references, we can modify their content: So, in a somewhat analytical way, $\mathsf{cons}(x,y)$ represents a mutable cons cell, as found in Lisp. We can further set

$$\mathsf{car}(\texttt{l}) \overset{\text{def}}{=} \pi_l(\texttt{l}) \qquad \mathsf{cdr}(\texttt{l}) \overset{\text{def}}{=} \pi_r(\texttt{l})$$

Note types match. The following program is similar to the one treated by Burstall [6].

$$L \quad \overset{\text{def}}{=} \quad x := \mathsf{cons}(0,\mathsf{nil})\ ;\ y := \mathsf{cons}(2,!x)\ ;\ \mathsf{car}(!x) := 1$$

We can then check, assuming x and y are not aliased:

$$\begin{array}{ll}
(L[l_x/x][l_y/y],\ \{l_x, l_y \mapsto \ldots\}) & \\
\quad \longrightarrow^* \ (\nu hlh'l')(\mathsf{car}(!l_x) := 1,\ \{h \mapsto 0,\ h' \mapsto 2,\ l \mapsto (h', \mathsf{nil}),\ l' \mapsto (h', l),\ l_x \mapsto l,\ l_y \mapsto l'\}) & \\
\quad \longrightarrow^* \ (\nu hlh'l')(h := 1,\ \{h \mapsto 0,\ h' \mapsto 2,\ l \mapsto (h', \mathsf{nil}),\ l' \mapsto (h', l),\ l_x \mapsto l,\ l_y \mapsto l'\}) & \\
\quad \longrightarrow \ \ (\nu hlh'l')((),\ \{h \mapsto 1,\ h' \mapsto 2,\ l \mapsto (h', \mathsf{nil}),\ l' \mapsto (h', l),\ l_x \mapsto l,\ l_y \mapsto l'\}) &
\end{array}$$

Analytical as it is, the final configuration precisely indicates the situation where x and y store two cells so that the tail of the pair stored in y coincides with the pair stored in x, as expected. This demonstrates (well-known) representability of procedural idioms in imperative higher-order functions.

Fig. 1 Typing Rules

$$[\mathit{Var}]\ \frac{-}{\Theta, x : \alpha \vdash x : \alpha} \qquad [\mathit{Constant}]\ \frac{-}{\Gamma;\Delta \vdash \mathsf{c}^C : C}$$

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

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

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

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

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

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

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

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

2.2 Typing

A *basis* $\Gamma;\Delta$ is a pair of finite maps, one from variables to non-reference types ($\Gamma, \ldots$, called *environment basis*) and the other from variables or labels to reference types ($\Delta, \ldots$, called *reference basis*). $\Theta, \Theta', \ldots$ combine two kinds of bases. The sequent for the typing is of the form:

$$\Gamma; \Delta \vdash M : \alpha$$

which reads: M has type α under $\Gamma; \Delta$. We often omit Γ or Δ if it is empty. We also use the typing sequent of the form:

$$\Theta \vdash M : \alpha$$

where Θ mixes these two kinds of maps, from which the original sequent can be immediately recovered (and vice versa). This latter form is convenient when, for example, a typing rule treats a variable regardless of its being a reference type or a non-reference type. The typing rules are standard [31], which we list in Figure 1 for reference (from first-order operations we only list two basic ones). We take the equi-isomorphic approach [31] for recursive types. In the first rule of Figure 1, c^C indicates a constant c has a base type C.

Notation 2.3 We often write $M^{\Gamma;\Delta;\alpha}$ for M such that $\Gamma; \Delta \vdash M : \alpha$.

Definition 2.4 *A program $M^{\Gamma;\Delta;\alpha}$ is* closed *if* $\mathsf{dom}(\Gamma) = \emptyset$.

One of the basic properties of typed formalisms is subject reduction. To state it, we need to type configurations. First we type a store by the following rule:

$$\frac{\forall l \in \mathsf{dom}(\sigma).(\ (\sigma(l) = V \wedge \Delta(l) = \mathsf{Ref}(\alpha)) \supset \Delta \vdash V : \alpha\ )}{\Delta \vdash \sigma}$$

That is, a store σ is typed under Δ when, for each l in its domain, $\sigma(l)$ is a closed value of type α under Δ, assuming $\Delta(l) = \mathsf{Ref}(\alpha)$. Note this means if $\Delta \vdash \sigma$ and V is stored (is in the co-domain of) σ, then any locations in V are already in $\mathsf{dom}(\Delta)$.

We then type a configuration by the following rule:

$$\frac{\Delta \cdot \tilde{l} : \tilde{\alpha} \vdash M : \alpha \quad \Delta \cdot \tilde{l} : \tilde{\alpha} \vdash \sigma}{\Delta \vdash (\nu \tilde{l})(M, \sigma)}$$

The following is standard [12, 31].

Proposition 2.5 (subject reduction) *Suppose* $\Gamma; \Delta_0 \vdash M : \alpha$ *and* $\Delta \vdash (\nu \tilde{l})(M, \sigma)$. *Then if we have a reduction* $(\nu \tilde{l})(M, \sigma) \longrightarrow (\nu \tilde{l}')(M', \sigma')$. *then we have (1)* $\Delta \vdash (\nu \tilde{l}')(M', \sigma')$ *and (2)* $\Delta' \vdash M' : \alpha$ *for some* $\Delta' \supset \Delta_0$.

Convention 2.6 *Henceforth we only consider well-typed programs and configurations.*

Write $C[\ ]^{\Gamma';\Delta';\alpha'}_{\Gamma;\Delta;\alpha}$ for a typed context which expects a program typed α under $\Gamma; \Delta$ to fill its hole and produces a program typed α' under $\Gamma'; \Delta'$. A typed context is *closing* if the resulting program is closed. We now define the standard contextual congruence on programs as follows.

Definition 2.7 (observational congruence) Let $\Gamma; \Delta \vdash M_{1,2} : \alpha$. Then we write $\Gamma; \Delta \vdash (\nu \tilde{l}_1)(M_1, \sigma_1) \cong (\nu \tilde{l}_2)(M_2, \sigma_2)$ often simply $(\nu \tilde{l}_1)(M_1, \sigma_1) \cong (\nu \tilde{l}_2)(M_2, \sigma_2)$ leaving type information implicit, if, for each typed context $C[\,\cdot\,]^{\mathsf{Unit}}_{\Gamma;\Delta;\alpha}$, the following condition holds:

$$(\nu \tilde{l}_1)(C[M_1],\ \sigma_1) \Downarrow \quad \equiv \quad (\nu \tilde{l}_2)(C[M_2],\ \sigma_2) \Downarrow$$

We also write $\Gamma; \Delta \vdash M_1 \cong M_2$, or simply $M_1 \cong M_2$ leaving type information implicit, if, $\tilde{l}_i = \sigma_i = \emptyset$ $(i = 1, 2)$.

3 Assertions for Local State

3.1 A Logical Language

The logical language we shall use is that of standard first-order logic with equality [23, § 2.8], extended with assertions for stateful evaluation [18, 19] (for imperative higher-order functions) and quantifications over store content [3] (for aliasing). On this basis we add a first-order predicate which asserts reachability of a reference name from a datum. The grammar follows, letting $\star \in \{\wedge, \vee, \supset\}$ and $Q \in \{\forall, \exists\}$.

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

The first set of expressions $(e, e', \ldots)$ are *terms* while the second set *formulae* $(A, B, C, C' \ldots)$. Terms include variables, constants (natural numbers, booleans and locations), pairing, projection, injection and standard first-order operations. $!e$ denotes the dereference (content) of a reference e. Note that e cannot contain actions with side effects. Using typed terms is not strictly necessary but contributes to clarity and understandability.

The logical language uses the standard logical connectives and quantification [23]. We include, following [3, 18], 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 programs' behaviour. Their use will be illustrated using concrete examples soon: here we informally outline their central ideas. First, $\{C\}\, e \bullet e' = x\, \{C'\}$ is called *evaluation formula*, introduced in [19], 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′.*

x binds free occurrences of x in C'. Having an explicit name x to denote the result in C' is crucial; evaluation formulae can be nested arbitrarily, by which we can describe specifications of arbitrary higher-order functions as assertions. This is a departure from other logics such as JML which uses specific variable `this` to denote the result. See examples of the assertions in [3, 19] and later in this paper.

$[!e]C$ and $\langle !e \rangle C$ are called *universal/existential content quantifications*, respectively, introduced and studied in [3]. The universal content quantification $[!e]C$ (with e of a reference type) says that:

> *Whatever (well-typed) value we may store in a reference denoted by e, the assertion C holds.*

$\langle !e \rangle C$ is interpreted dually:

> *For some possible (well-typed) content of a reference denoted by e, the assertion C holds.*

Note that, in both cases, we valuate validity of C under a hypothetical content of e, not necessarily its real one. From its meaning, we can see variables in e of $[!e]C$ and $\langle !e\rangle C$ occur free: in particular, x in $[!x]C$ and $\langle !x\rangle C$ is a free occurrence. In the presence of aliasing, quantifying over names is not the same thing as quantify over their content, demanding these quantifiers, enabling hypothetical statement over content of imperative variables. Their usage for structured reasoning for programs with aliasing, together with their logical status, is detailed in [3].

Finally, $e_1 \hookrightarrow e_2$ (which reads: "e_2 is reachable from e_1", with e_2 of a reference type and e_1 of an arbitrary type) is called *reachability predicate*. This predicate is newly introduced and plays an essential role in the present logic. $e_1 \hookrightarrow e_2$ says that:

> *One can reach (or better: some closed program can reach) the reference referred to by e_2 solely starting 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. In assertions, we often use its negation, written $y\#x$ [9, 35], which says one can never reach a reference y starting from x. Later we make precise its semantics and discuss methods for deriving this relation through syntactic axioms.

Note that e does not contain abstractions, applications and assignments which involve non-trivial dynamics (possibly infinite reductions) for its simplification. For example, pairing, projections and injections do not include such reduction and useful to represent data-structures.

Convention. Logical connectives are used with standard precedence/association, using parentheses as necessary to resolve ambiguities. $\mathsf{fv}(C)$ (resp. $\mathsf{ftv}(C)$) denotes the set of free variables (resp. free type variables) in C. Note that x in $[!x]C$ and $\langle !x\rangle C$ occurs free, while x in $\{C\}\ e \bullet e' = x\ \{C'\}$ occurs bound within its scope C'. $C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$. $C^{\text{-}\tilde{x}}$ indicates $\mathsf{fv}(C) \cap \{\tilde{x}\} = \emptyset$. We write $x\#\vec{y}$ for $\wedge_i x\#y_i$; similarly $\vec{x}\#y$ stands for $\wedge_i x_i\#y$.

Terms are naturally 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 examples.

3.2 Assertions for Higher-Order Functions and Aliasing

We start from a quick review of the assertion method introduced and studied in [3, 18, 19] which form the basis of the present work.

1. The assertion $x = 6$ says that x of type Nat is equal to 6. Assuming x has type $\mathsf{Ref}(\mathsf{Nat})$, $!x = 2$ means that x stores 2.
2. The assertion
$$\forall n.\{\mathsf{T}\}u \bullet n = z\{z = 2 \times n\} \tag{3.1}$$
says a program named u would double the number whenever it is invoked. The program $\lambda x.x + x$ named u satisfies this specification.

3. The assertion

$$\forall ni.\{!x = i\}u \bullet n = z\{z = () \wedge !x = i + n\} \tag{3.2}$$

describes a program of type $\mathsf{Nat} \Rightarrow \mathsf{Unit}$, which, upon receiving a natural number n, returns $()$ (the unique closed value of type Unit) and increments the content of x by n. The program $\lambda y.x := !x + y$, named u, has this behaviour.
4. As seen above, a program whose return type is Unit can have only one return value, $()$, if any, so that we naturally omit it. Thus we write, abbreviating (3.2) above:

$$\forall ni.\{!x = i\}u \bullet n\{!x = i + n\} \tag{3.3}$$

A similar example (assuming multi-ary arguments):

$$\forall x, y, i, j.\{!x = i \wedge !y = j\}u \bullet (x, y)\{!x = j \wedge !y = i\} \tag{3.4}$$

which is satisfied by the standard swapping function named u, given as follows: $\lambda(x, y).\mathtt{let}\ i = !y\ \mathtt{in}\ (y := !x; x := i)$.
5. For a fuller specification of the swapping function, we may refine the assertion (3.4) by saying what it does not do, that is it does not touch references except what it has received as arguments. Such a property is often essential when a program is used as part of a bigger program. The following notation, called *located assertion* [3], is used for this purpose.

$$\forall x, y, i, j.\{!x = i \wedge !y = j\}u \bullet (x, y)\{!x = j \wedge !y = i\} @ xy \tag{3.5}$$

Above "$@\tilde{x}$" indicates that the evaluation touches only $\tilde{x}$, leaving content of other references unchanged. The assertion (3.5) in fact stands for the following formula in our logical language, with $r \neq xy$ standing for $r \neq x \wedge r \neq y$ and r and h fresh.

$$\begin{aligned}&\forall \mathrm{X}, r^{\mathsf{Ref}(\mathrm{X})}, h^{\mathrm{X}}, x, y, i, j.\\ &\{!x = i \wedge !y = j \wedge r \neq xy \wedge !r = h\}u \bullet (x, y)\{!x = j \wedge !y = i \wedge !r = h\}\end{aligned} \tag{3.6}$$

The assertion says:

For any r of any reference type which is distinct from x and y, its content, denoted by h, stays invariant after the execution of the swapping function,

that is, only x and y are touched (these x and y are called *write effects*: in this work, as in [3], we assume write effects do not contain dereferences, which simplifies their semantics without losing generality). Translation from (3.5) to (3.6) is mechanical, so that located assertions can be treated just as standard formulae. The ability to (in)equate and quantify over reference names plays a crucial role in this translation.
6. The located assertion can also be used for refining (3.1):

$$\forall n.\{\mathsf{T}\}u \bullet n = z\{z = 2 \times n\} @ \emptyset \tag{3.7}$$

asserting that no store is touched by this function, i.e. u has purely functional behaviour. In (3.7), only the input/output relation matters: hence without loss of precision we can further abbreviate it to:

$$\forall n.u \bullet n \searrow 2 \times n \tag{3.8}$$

7. Consider the assertion $\langle !x \rangle !y = 5$ which stands for: "for some content of x, $!y = 5$ holds." Because $!y = 5$ is equivalent to $(x = y \wedge !x = 5) \vee (x \neq y \wedge !y = 5)$, and because $\langle !x \rangle$ cancels any constraint about the content of x (but *not* about x itself), we know $\langle !x \rangle !y = 5$ is equivalent to $x \neq y \supset !y = 5$. Next consider $[!x]!y = 5$. It says that whatever x may store, the number stored in y is 5. The assertion is logically equivalent to $x \neq y \wedge !y = 5$. In this way, $\langle !e \rangle C$ claims C holds for the content of a reference qualified in C *if* that reference is distinct from e; whereas $[!e]C$ claims C holds *and* any reference whose content is discussed in C is distinct from e.
8. Using content quantification, we can define *logical substitution* which is robust in the presence of aliasing, used for the proof rule for assignment.

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

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. Combination of content quantification and predicate for locality in the present logic offers a tractable tool for modular reasoning, as demonstrated in Section 5.

3.3 Assertions for Local State

Concrete examples of assertions for local state follow.

1. Consider a simple command $x := y; y := z; w := 1$. After its run, we can reach the reference z by dereferencing y, and y by dereferencing x. Hence z is reachable from y, y from x, and z from x. That is the final state satisfies

$$x \hookrightarrow y \wedge y \hookrightarrow z$$

which also implies $x \hookrightarrow z$, by transitivity.
2. Next, assuming w is newly generated, we may wish to say w is *unreachable* from x, to ensure the freshness of w. For this we assert

$$w\#x,$$

which, as noted, stands for $\neg(x \hookrightarrow w)$. Note that $w\#x \supset w \neq x$. Note also that $x \hookrightarrow x \equiv \mathsf{T}$ and $x\#x \equiv \mathsf{F}$, but $!x \hookrightarrow x$ may not be T and $x\#!x$ may not be F.
3. Consider $x_1 := y_1; y_1 := z_1; x_2 := y_2; y_2 := z_2$. Then we may wish to say that any content reachable from x_1 (here x_1, y_1, z_1) is unreachable from any content reachable from x_2 (here x_2, y_2, z_2), so that we can represent a deep separation of the two resources. To represent this specification, we asserts $x_1 \star x_2$, which we formally set:

$$e_1 \star e_2 \stackrel{\text{def}}{=} \forall \mathrm{X}, y^{\mathsf{Ref}(\mathrm{X})}.(e_1 \hookrightarrow y \supset y\#e_2)$$

which is logically equivalent to $\forall \mathrm{X}, y^{\mathsf{Ref}(\mathrm{X})}.(y\#e_1 \vee y\#e_2)$. This means any y reachable from e_1 is unreachable from e_2 and vice verse. That is, all reachable nodes from e_1 are disjoint with those from e_2: we have two mutually disjoint data structures.

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 `let` $x = \texttt{ref}(z)$ `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 it cannot touch x in any context.
5. Consider the reference $\texttt{ref}(M) \stackrel{\text{def}}{=} \texttt{new}\ x\ := M\ \texttt{in}\ x$. It returns a freshly generated memory cell, initialised to the value pass as an argument. Then the program $\lambda n.\texttt{ref}(n)$ meets

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

   where j is fresh. This means that it creates a new reference z whose stored value is n, and its new name is unreachable from any name (hence fresh). We abbreviate it as:

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

6. Finally we consider a factorial program which realise a recursion by circular references, the idea due to Landin [21] in Introduction. In [19], we have shown we can derive the following post-condition for the above program (with an initial state, say T).

   $$\mathsf{circfact}(x) \stackrel{\text{def}}{=} \exists g.(\forall n.\{!x = g\}g \bullet n = z\{z = n! \wedge !x = g\}@x \ \wedge \ !x = g)$$

   The assertion says:

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

   In `circFact`, x occurs free. In the present language, we can further hide x as shown in `safeFact` in Introduction. Since x is now invisible and inaccessible from the outside, so that the program as a whole behaves as a pure factorial function, satisfying:

   $$\forall n.u \bullet n \searrow n! \qquad (3.9)$$

   We derive (3.9) as the postcondition of `safeFact` from $\mathsf{circfact}(x)$ using the axioms for reachability in Section 7.7.

Further examples of assertions are found later.

3.4 Formulae for Freshness

Below we list three forms of formulae for freshness. First, the following appeared already: νx says x is distinct from any names existing in the initial state, giving the most general (weakest) form of freshness. Below x is of a reference type.

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

Note that z and x are distinct variables by the binding condition. We can equivalently use $\mathsf{Ref}(\mathrm{X})$ instead of X (the meaning does not change since a reference name cannot be equated with a variable of non-reference type).

Next we demand $\#$ instead of simple inequality. Below z again should be typed with a reference type.

$$\{C\}e\bullet e' = z\{\#z.C'\} \stackrel{\text{def}}{=} \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e\bullet e' = z\{z\#i \wedge C'\}$$

Note z is bound in the whole formula because it occurs as the result name in the evaluation formula. In contrast, $\#z.C'$ does*not* induce binding on z, as can be seen from the encoding. It is notable that $\{C\}e\bullet e' = z\{\#x.C'\}$ (with x and z distinct) is never valid since i can then denote x. Thus the binding on z induced by the evaluation formula is essential for its consistency, analogous to the existential binding in the first freshness formula.

Finally the strongest disjointness property uses $\star$. below z may *not* be of a reference type.

$$\{C\}e\bullet e' = z\{\star z.C'\} \stackrel{\text{def}}{=} \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e\bullet e' = z\{z\star i \wedge C'\} \tag{3.10}$$

The formula says the invocation of e with argument e' results in a chunk of connected data structure which as a whole is fully disjoint from what existed in the initial state.

The last two forms of freshness formulae have variations, where we combine $\sharp$ with ν. While we may not use them in the subsequent technical development nor in examples, they sometimes become useful in reasoning. Let x below be of a reference type.

$$\{C\}e\bullet e' = z\{\nu\sharp x.C'\} \quad \stackrel{\text{def}}{=} \quad \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e\bullet e' = z\{\exists x.(x\#i \ \wedge \ C')\} \tag{3.11}$$

which says the evaluation of $e\bullet e'$ leads to a creation of a reference x which is unreachable from anything extant in the initial state. Similarly, and this time x of an arbitrary type:

$$\{C\}e\bullet e' = z\{\nu\star x.C'\} \quad \stackrel{\text{def}}{=} \quad \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e\bullet e' = z\{\exists x.(x\star i \ \wedge \ C')\} \tag{3.12}$$

which says that the evaluation of $e\bullet e'$ leads to a creation of a chunk of data structure reachable from x which is disjoint from anything extant in the initial state. In these variants too, i can be typed $\mathsf{Ref}(\mathrm{X})$ instead of X with the same effect.

We may further extend these notations to a set of references, e.g. we may write, with $S \stackrel{\text{def}}{=} \{w|E(w)\}$,

$$\{C\}e\bullet e' = z\{\#S.C'\} \quad \stackrel{\text{def}}{=} \quad \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e\bullet e' = z\{\forall w.(E(w) \supset w\#i) \ \wedge \ C'\} \tag{3.13}$$

4 Models

This section introduces models for the logic and defines the semantics (satisfaction relation) of assertions. The class of models we introduce is based on, and extends, those for our preceding program logics for higher-order behaviour[3, 17–19]. For clear presentation, we take three steps to introduce them.

Step 1: We first build *open models*, which capture imperative higher-order functions [19] and aliasing [3]. Open models directly come from [3].
Step 2: We incorporate locality to open models using "ν" *binders*, which hide reference names.
Step 3: Resulting models are made semantically precise through a quotient construction, on the basis of which we introduce basic operators for interpretation.

Following [3, 17–19], we use programs themselves to build models, though other options are possible. The highlight of the section is interpretation of equality which precisely captures the observational meaning of assertions for behaviour with local state, answering to one of the issues raised in Introduction.

4.1 Open Models

We first define open models.

Definition 4.1 (open models) *An* open model of type $\Theta = \Gamma;\Delta$, *written* $\mathcal{M}^{\Gamma;\Delta}$ *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.* $\Delta \vdash \sigma(l) : \alpha$.

Example 4.2 (open model) As an example, an assertion:

$$!x = 0 \ \wedge\ \forall i.\{!x = i\}u \bullet () = z\{!x = z \wedge !x = i+1\} \tag{4.1}$$

may be interpreted using the following open model.

$$\{u : \lambda().(l := !l+1;\ !l)\}, \quad \{l \mapsto 0\} \tag{4.2}$$

We can the interpret identifiers, terms and predicates in (4.1) using (4.2).

4.2 Models with Locality

Open models are close to the standard notion of model in that they are maps interpreting identifiers in assertions. For capturing local state, we have to foresake this map-like nature of models and incorporating hidden names. We illustrate the key idea using the Introduction's (1.1), reproduced below.

$$\mathtt{Inc} \overset{\text{def}}{=} \mathtt{new}\ x := 0\ \mathtt{in}\ \lambda().(x := !x+1;\ !x)$$

When we run this program, the initial state may for example be given as the empty open model: then, after running `Inc`, we reach a state where a hidden name stores 0, to be used by the resulting procedure when invoked. We represent this state of affairs by adding a binder (as used in configurations for reduction, cf. §2.1) to (4.2), as follows.

$$(\nu\, l)(\{u : \lambda().(l := !l+1;\ !l)\},\quad \{l \mapsto 0\}) \tag{4.3}$$

(4.3) says that there is a behaviour named u and a reference named l which is not aliased with any other names that this reference stores 0, and that the name l is hidden. Based on (4.3), we may assert:

$$\exists x.(!x = 0\ \wedge\ \forall i.\{!x = i\}u \bullet i = z\{!x = z \wedge !x = i+1\}) \tag{4.4}$$

This gives us the following notion of models with hidden names.

Definition 4.3 (models) *A* model *of type* $\Gamma;\Delta$ *is a structure*

$$(\nu\,\tilde{l})(\xi, \sigma)$$

where (ξ,σ) *is an open model of type* $\Gamma;\Delta\cdot\Delta'$ *with* $\mathsf{dom}(\Delta') = \{l\}$. $\mathcal{M}, \mathcal{M}', \ldots$ *range over models.*

In $(\nu\vec{l})(\xi,\sigma)$, $\tilde{l}$ act as binders, which gives standard α-equality on models.

4.3 Abstract Models.

Observationally, models in Definition 4.3 are too concrete, consider:

$$(\nu\, l l')(\{u : \lambda().(l := !l+1; l' := !l'+1;\ (!l + !l')/2)\},\ \{l, l' \mapsto 0\}) \tag{4.5}$$

The behaviour located at u has the same observable behaviour as that located at u in (4.3), in spite of its difference in internal structure. Indeed, just as (4.3) originates in `Inc`, (4.5) originates in

$$\texttt{Inc2} \stackrel{\text{def}}{=} \texttt{new}\ x, y := 0\ \texttt{in}\ \lambda().(x := !x+1; y := !y+1;\ (!x + !y)/2)$$

which is contextually equivalent to `Inc`: and if two models originate in the same abstract behaviour, we wish them to be the same model. For this purpose we use the behavioural equivalence $\cong$.

Definition 4.4 *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*

$$\Gamma;\Delta \vdash \mathcal{M}_1 \approx \mathcal{M}_2 \qquad \textit{iff} \qquad (\nu\,\tilde{l}_1)(\langle V_{11}, .., V_{1n}\rangle, \sigma_1) \cong (\nu\,\tilde{l}_2)(\langle V_{21}, .., V_{2n}\rangle, \sigma_2)$$

The $\approx$*-congruence classes are called* abstract models.

4.4 Operations on Models

The following operations and relations on models are used for defining the satisfaction relation for our assertion language. In the next subsection we shall show they are invariant under $\approx$.

Definition 4.5 (expansion) *Let $M^{\Gamma;\Delta} \overset{def}{=} (\nu\tilde{l})(\xi,\sigma)$. Assuming $\Gamma;\Delta \vdash N : \alpha$ and u fresh we set*

$$\mathcal{M}[u\!:\!N] \overset{def}{=} (\nu\tilde{l})(\xi\cdot u\!:\!N\xi,\ \sigma).$$

Note $\mathcal{M}[u\!:\!N]$ is not necessarily a model, because $N\xi$ may not be a value. To obtain a model, we write

$$\mathcal{M}[u\!:\!N] \Downarrow (\nu\tilde{l}\tilde{l}')(\xi\cdot u : V, \sigma')$$

when $(N\xi,\sigma) \Downarrow (\nu\tilde{l}')(V,\sigma')$. By determinacy of the reduction, $\mathcal{M}'$ is uniquely determined, should $\mathcal{M}[u\!:\!N]$ converge. If not, we write $\mathcal{M}[u\!:\!N] \Uparrow$.

Definition 4.6 (update) *Let $M^{\Gamma;\Delta} \overset{def}{=} (\nu\tilde{l})(\xi,\sigma)$ and e and V w.r.t. $M^{\Gamma;\Delta}$ be respectively typed as $\mathsf{Ref}(\alpha)$ and α under $\Gamma;\Delta$*

$$\mathcal{M}[e \mapsto V] \ \overset{def}{=}\ (\nu\tilde{l})(\xi, \sigma[l \mapsto V\xi])$$

if $(\nu\tilde{l})(e\xi,\sigma) \Downarrow (\nu\tilde{l})(l,\sigma)$.

Notation 4.7 Let $\mathcal{M} = (\nu\tilde{l})(\xi\cdot u : V, \sigma)$. Write $\mathcal{M}/u$ for $(\nu\tilde{l})(\xi,\sigma)$.

4.5 Properties of Operations

This subsection shows the operations and relations on models introduced in the previous subsection are closed with respect to $\approx$, so that they can directly be considered as operations/relations on abstract models.

We start from one important notion in the present model, symmetry, coming from a process-theoretic nature of our models. We first define permutation concretely.

Definition 4.8 (permutation) *Let $\mathcal{M}^{\Gamma;\Delta} \overset{def}{=} (\nu\tilde{l})(\xi\cdot v\!:\!V\cdot w\!:\!W,\ \sigma)$. Then, for any $v,w \in$ $\mathsf{dom}(\Gamma)$, we set:*

$$\binom{vw}{wv}\mathcal{M} \ \overset{def}{=}\ (\nu\tilde{l})(\xi\cdot v\!:\!W\cdot w\!:\!V,\ \sigma).$$

which we call a (binary) permutation of $\mathcal{M}$ at u and w. *We extend this to an arbitrary bijection ρ on $\mathsf{dom}(\Gamma)$, writing $(\rho)\mathcal{M}$.*

Remark 4.9

1. By definition, given $\mathcal{M}^{\Gamma;\Delta}$, if $\Gamma(v) = \Gamma(w)$ then a permutation $\binom{uv}{vu}\mathcal{M}$ has the same type as $\mathcal{M}$.
2. As a simple example of a permutation, if u,v are fresh w.r.t. $\mathcal{M}$ and $\{u,v\}\cap(\mathsf{fv}(e)\cup \mathsf{fv}(e')) = \emptyset$, then we have:

$$\binom{uv}{vu}(\mathcal{M}[u\!:\!e][v\!:\!e']) \ \overset{\text{def}}{=}\ \mathcal{M}[u\!:\!e'][v\!:\!e]$$

We shall have further examples later.

3. A permutation in the sense of Definition 4.8 is related with, but different from, a bijective renaming on a model, which we write e.g. $\mathcal{M}[uv/vu]$. For example, we have:
$$(\mathcal{M}[u\!:\!e][v\!:\!e'])[uv/vu] \stackrel{\text{def}}{=} \mathcal{M}[v\!:\!e][u\!:\!e']$$
Note $\binom{uv}{vu}\mathcal{M}[v:e][u:e']$ is essentially the same model as $\mathcal{M}$ except they differ in typing (as a sequence).

Definition 4.10 (symmetry) *A permutation* ρ *on* $\mathcal{M}$ *is a* symmetry on $\mathcal{M}$ *when* $(\rho)\mathcal{M} \approx \mathcal{M}$.

Any model has the trivial symmetry, identity. To show more examples, we introduce a semantic presearving encoding from a model to a term. Let $\mathcal{M} = (\nu\tilde{l})(\{y_1 : V_1, y_2 : V_2, ..., y_n : V_n\},\ [l_1 \mapsto W_1]\cdots[l_m \mapsto W_m])$. Then we define:

$$[\![\mathcal{M}]\!] \stackrel{\text{def}}{=} \texttt{let}\, l_1 = \texttt{ref}(W_1)\ \texttt{in}\,\texttt{let}\, l_2 = \texttt{ref}(W_2)\ \texttt{in}\,\texttt{let}\, l_n = \texttt{ref}(W_n)\ \texttt{in}\ \langle V_1, ..., V_n\rangle$$

Obviously we have: $\mathcal{M}_1 \approx \mathcal{M}_2$ iff $[\![\mathcal{M}_1]\!] \cong [\![\mathcal{M}_2]\!]$. The following shows subtlety of symmetries.

Example 4.11 (symmetry)

1. The following two models correspond to `IncShared` and `IncUnShared`:
$$\mathcal{M}_1 \stackrel{\text{def}}{=} (\nu\, l)(\{u : \texttt{Inc}[l/x], v : \texttt{Inc}[l/x]\},\ \{l \mapsto 0\})$$
$$\mathcal{M}_2 \stackrel{\text{def}}{=} (\nu\, ll')(\{u\!:\!\texttt{Inc}[l/x], v\!:\!\texttt{Inc}[l'/x]\},\ \{l \mapsto 0, l' \mapsto 0\})$$
Both $\mathcal{M}_1$ and $\mathcal{M}_2$ have an obvious symmetry $\binom{uv}{vu}$.
2. If we expand these two models, however, we find one retains a symmetry while another doesn't.
$$\binom{wv}{vw}(\mathcal{M}_1[w\!:\!u]) \approx \mathcal{M}_1[w\!:\!u]$$
$$\binom{wv}{vw}(\mathcal{M}_2[w\!:\!u]) \not\approx \mathcal{M}_2[w\!:\!u]$$
To see why the latter is the case, let:
$$C[\,] \stackrel{\text{def}}{=} \lambda x.(\texttt{let}\, y = \pi_2(x)()\ \texttt{in}\,\texttt{let}\, z = \pi_3(x)\ \texttt{in}\, z)[\,]$$
Then $V[\![\mathcal{M}_2[w\!:\!u]]\!] \longrightarrow^* 1$ but $V[\![\binom{wv}{vw}\mathcal{M}_2[w\!:\!u]]\!] \longrightarrow^* 2$, because of sharing.

Definition 4.12 (witness) *Given a partial map* $\mathcal{F}$ *on models of specific types, we say a program* M witnesses $\mathcal{F}$ *if* $[\![\mathcal{F}\,\mathcal{M}]\!] \cong_{\mathsf{id}} M[\![\mathcal{M}]\!]$ *for each* $\mathcal{M}$ *in the domain of* $\mathcal{F}$.

Remark 4.13 (witness) By having witnesses for these operations, they can be considered as operations on encodings of concrete models, hence in effect those on abstract models, by the contextual closure of $\cong$. Concrete presentation of operations often elucidates the nature of operations: the notion of witness, as well as the subsequent results, indicates that we can safely work with concrete operations without violating abstract nature of models, as far as they can be given appropriate witnesses.

Lemma 4.14 *Let* $\mathsf{dom}(\Gamma;\Theta) = \{y_1,...,y_n\}$ *below.*

1. *Given* $\Gamma;\Delta \vdash N : \alpha$,

$$\lambda m^{\Pi\Gamma}.\langle m, (..(\lambda y_1...y_n.N)\pi_1(m)..\pi_n(m))\rangle$$

witnesses the expansion from $\mathcal{M}$ *of type* $\Gamma;\Delta$ *to* $\mathcal{M}[u\!:\!N]$.

2. *Given* $\Gamma;\Delta \vdash e : \mathsf{Ref}(\alpha)$ *and* $N^{\Gamma;\Delta;\alpha}$*, the program*

$$\lambda x^{\Pi\Gamma}.((..((\lambda y_1...y_n.e)\pi_1(m))..\pi_n(m)) := N; x)$$

witnesses the update $[\![\mathcal{M}[e \mapsto N]]\!]$ *of* $\mathcal{M}^{\Gamma;\Delta}$.

3. *For i such that* $1 \leq i \leq n$*, the program*[3]*.*

$$\lambda m^{\Pi\Gamma}.\langle \pi_1(m),..,\pi_{i-1}(m),\pi_{i+1}(m),..,\pi_n(m)\rangle$$

witnesses the projection $\mathcal{M}/y_i$.

PROOF: Immediate from the construction. □

Lemma 4.15 $\mathcal{M}_1 \approx \mathcal{M}_2$ *implies* $\mathcal{M}_1[u\!:\!N] \approx \mathcal{M}_2[u\!:\!N]$.

PROOF: Let u be fresh, $\mathcal{M}_i^{\Gamma;\Delta}$ and $\Gamma;\Delta \vdash N : \alpha$. Write V for the witness of the expansion in Definition 4.12. We infer:

$$\begin{aligned}\mathcal{M}_1 \approx \mathcal{M}_2 &\Rightarrow [\![\mathcal{M}_1]\!] \cong [\![\mathcal{M}_2]\!]\\ &\Rightarrow V[\![\mathcal{M}_1]\!] \cong V[\![\mathcal{M}_2]\!]\\ &\Rightarrow [\![\mathcal{M}_1[u\!:\!N]]\!] \cong [\![\mathcal{M}_2[u\!:\!N]]\!]\\ &\Rightarrow \mathcal{M}_1[u\!:\!N] \approx \mathcal{M}_2[u\!:\!N]\end{aligned}$$

where the third step uses Lemma 4.14 (1). □

Lemma 4.16 $\mathcal{M}_1 \cong \mathcal{M}_2$ *implies* $\mathcal{M}_1/u \cong \mathcal{M}_2/u$.

PROOF: As in Lemma 4.15, using Lemma 4.14 (3) instead of (1). □

By successively applying Lemma 4.16, the property extends to arbitrary projections of models.

Lemma 4.17 $\mathcal{M}_1 \approx \mathcal{M}_2$ *implies* $(\nu l)\mathcal{M}_1 \approx (\nu l)\mathcal{M}_2$.

PROOF: We prove that if $\Gamma;\Delta,x : \mathsf{Ref}(\alpha) \vdash M_1 \cong M_2 : \beta$ and $\Gamma;\Delta \vdash V : \alpha$, then

$$\Gamma;\Delta \vdash \mathtt{let}\ x = \mathtt{ref}(V)\ \mathtt{in}\ M_1 \ \cong \mathtt{let}\ x = \mathtt{ref}(V)\ \mathtt{in}\ M_2 : \beta$$

[3] In the third clause, if $i = 1$ (resp. $i = n$) then we take off $\pi_{i-1}(m)$ (resp. π_{i+1}) from the sequence $\langle \pi_1(m),..,\pi_{i-1}(m),\pi_{i+1}(m),..,\pi_n(m)\rangle$

Choose appropriately typed $C[\cdot]$.

$$\begin{aligned} C[\texttt{let}\ x = \texttt{ref}(V)\ \texttt{in}\ M_1] &\longrightarrow (\nu l)(C[M_1[l/x]], l \mapsto V) \Downarrow \\ &\Leftrightarrow (l := V; M_1) \Downarrow \\ &\Leftrightarrow (l := V; M_2) \Downarrow \\ &\Leftrightarrow C[\texttt{let}\ x = \texttt{ref}(V)\ \texttt{in}\ M_2] \longrightarrow (\nu l)(C[M_2[l/x]], l \mapsto V) \Downarrow \end{aligned}$$

as required. Then by definition, we obtain the result. $\Box$
Finally we mention basic properties of permutation and symmetries. First, a permutation has an obvious witness:

Lemma 4.18 (permutation witness) *Given $\mathcal{M}^{\Gamma;\Delta}$ and a bijection ρ on $\mathsf{dom}(\Gamma)$, the operation ρ is witnessed by the standard isomorphism on Π permuting the elements following ρ.*

Corollary 4.19 *Given $\mathcal{M}_{1,2}^{\Gamma;\Delta}$ and a bijection ρ on $\mathsf{dom}(\Gamma)$, we have $\mathcal{M}_1 \approx \mathcal{M}_2$ iff $(\rho)\mathcal{M}_1 \approx (\rho)\mathcal{M}_2$.*

Hence we know:

Proposition 4.20 *If $\mathcal{M}_1 \approx \mathcal{M}_2$ and if (ρ) is a symmetry of $\mathcal{M}_1$, then (ρ) is also a symmetry of $\mathcal{M}_2$.*

4.6 Semantics of Assertions

This subsection defines semantics of assertions. The interpretation of terms is straightforward, given as follows.

Definition 4.21 Let $\Gamma;\Delta \vdash e : \alpha$, $\Gamma;\Delta \vdash \mathcal{M}$ and $\mathcal{M} = (\xi, \sigma)$. Then the *interpretation of e under $\mathcal{M}$*, denoted $[\![e]\!]_{\xi,\sigma}$ is inductively given by the clauses below.

- $[\![x]\!]_{\xi,\sigma} = \xi(x)$.
- $[\![!e]\!]_{\xi,\sigma} = \sigma([\![e]\!]_{\xi,\sigma})$.
- $[\![()]\!]_{\xi,\sigma} = ()$, $[\![\mathsf{n}]\!]_{\xi,\sigma} = \mathsf{n}$, $[\![\mathsf{b}]\!]_{\xi,\sigma} = \mathsf{b}$, and $[\![l]\!]_{\xi,\sigma} = l$.
- $[\![\mathsf{op}(\tilde{e})]\!]_{\xi,\sigma} = \mathsf{op}([\![\tilde{e}]\!]_{\xi,\sigma})$.
- $[\![\langle e, e' \rangle]\!]_{\xi,\sigma} = \langle [\![e]\!]_{\xi,\sigma}, [\![e']\!]_{\xi,\sigma} \rangle$.
- $[\![\pi_i(e)]\!]_{\xi,\sigma} = \pi_i([\![e]\!]_{\xi,\sigma})$.
- $[\![\texttt{inj}_i(e)]\!]_{\xi,\sigma} = \mathsf{inj}_i([\![e]\!]_{\xi,\sigma})$.

where the operators π_i etc. are abused to denote the corresponding ones.

Definition 4.22 (name closure) Let σ be a store in some model and $S \subset \mathsf{dom}(\sigma)$. Then the *label closure of S in* σ, written $cl(S, \sigma)$, is the minimum S' such that: (1) $S \subset S'$ and (2) If $l \in S'$ then $\mathsf{fl}(\sigma(l)) \subset S'$.

Lemma 4.23 *For all σ, we have:*

1. *$S \subset cl(S, \sigma)$*

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

We are now ready to define semantics of assertions. To treat type variables, we augment a model $\mathcal{M}$ with a map from type variables to closed types. In the following, all omitted cases are by de Morgan duality.

– $\mathcal{M} \models e_1 = e_2$ 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 for all V, $\mathcal{M}'[x:V] \models C$; or for each $(\nu\, l)\mathcal{M}' \approx \mathcal{M}$ with l typed by α, $\mathcal{M}'[x:l] \models C$.
– $\mathcal{M} \models \forall \mathrm{X}.C$ if for all closed type α, $\mathcal{M}\cdot\mathrm{X}:\alpha \models C$.
– $\mathcal{M} \models [!e]C$ if for all V, $\mathcal{M}[e \mapsto V] \models C$.
– $\mathcal{M} \models \{C\}e \bullet e' = x\{C'\}$ if, whenever $\mathcal{M}[u:N] \Downarrow \mathcal{M}_0$ and $\mathcal{M}_0 \models C$ with u fresh, we have $\mathcal{M}_0[x:ee'] \Downarrow \mathcal{M}' \models C'$.
– $\mathcal{M} \models e_1 \hookrightarrow e_2$ if for each $(\nu\, \tilde{l})(\xi,\sigma) \approx \mathcal{M}$, $[\![e_2]\!]_{\xi,\sigma} \in cl(\mathsf{fl}([\![e_1]\!]_{\xi,\sigma}),\sigma)$

The reachability clause says the set of hereditarily reachable names from e_1 includes e_2 up to $\approx$. We can check, with f_w, f_r and f_c denoting $\lambda().x := 1$, $\lambda().!x$ and `let` $x =$ `ref`(z) `in` $\lambda().x$ respectively as in § 3.3 (4), we have $f_w \hookrightarrow x$, $f_r \hookrightarrow x$ and $f_c \hookrightarrow z$.

The following characterisation of # is often useful for justifying axioms for fresh names.

Proposition 4.24 (partition) *$\mathcal{M} \models x\#u$ if and only if $\mathcal{M} \approx (\nu\, \tilde{l})(\xi \cdot u:V \cdot x:l,\ \sigma_1 \uplus \sigma_2)$ such that $cl(\mathsf{fl}(V),\sigma_1 \uplus \sigma_2) = \mathsf{fl}(\sigma_1) = \mathsf{dom}(\sigma_1)$ and $l \in \mathsf{dom}(\sigma_2)$.*

PROOF: Let $\sigma = \sigma_1 \uplus \sigma_2$. Note $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset$. For the only-if direction, we note $l \notin cl(\mathsf{fl}(V),\sigma)$ by definition of reachability. Since $l \in \mathsf{dom}(\sigma_2)$, there exists σ_1 such that $l \notin \mathsf{dom}(\sigma_1)$ and $cl(\mathsf{fl}(V),\sigma) = cl(\mathsf{fl}(V),\sigma_1) = \mathsf{fl}(\sigma_1) = \mathsf{dom}(\sigma_1)$, hence done. The if-direction is obvious by definition of reachability. $\square$

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.

Remark 4.25 (equality with locality) The clause for the satisfaction of the equality $e_1 = e_2$ given above, does *not* use equality/equivalence of interpreted terms as elements of some set. In spite of this, Section 6 will show the induced relation satisfies all standard axioms for equality. We shall also show, in this section, that it captures intuitive meaning of equality on behaviour with locality. Here we illustrate how the given clause precisely captures the subtlety of equality of behaviours with shared local state, using simple examples. The defining clause for equality validate:

$$(\nu l)\begin{pmatrix} u:\lambda().(l := !l+1; !l), & \\ v:\lambda().(l := !l+1; !l), & l \mapsto 0 \end{pmatrix} \models u = v$$

On the other hand, it also says:

$$(\nu l l')\begin{pmatrix} u : \lambda().(l := !l+1; !l), & \\ v : \lambda().(l' := !l'+1; !l'), & l \mapsto 0, l' \mapsto 0 \end{pmatrix} \models u \neq v$$

Call the first model $\mathcal{M}_1$ and the second $\mathcal{M}_2$ ($\mathcal{M}_1$ and $\mathcal{M}_2$ are the same models as treated in Remark 4.11, page 18). We observe:

- In $\mathcal{M}_1$, u and v share state, while in $\mathcal{M}_2$ they don't, reminiscent of `IncShared` and `IncUnShared` in Introduction.
- Intuitively, saying u and v are equal in $\mathcal{M}_1$ makes sense because running u always has the same effect as running v, so behaviourally any description of u should be replaceable with those of v and vice versa.
- But, in $\mathcal{M}_2$, running u and v once each is quite different from running only u twice: u and v are far from being mutually substitutive.

More concretely, both for $\mathcal{M}_1$ and $\mathcal{M}_2$, we can obviously soundly assert:

$$\exists x.(!x = 0 \ \wedge \ \mathsf{inc}(u,x) \ \wedge \ \mathsf{inc}(u,x)) \tag{4.6}$$

where $\mathsf{inc}(u,x) \stackrel{\text{def}}{=} \forall i.\{!x = i\}u \bullet () = z\{!x = z \wedge !x = i+1\}@x$. In $\mathcal{M}_1$, since we have $u = v$, we can apply the law of equality to (4.6), obtaining:

$$\exists x.(!x = 0 \ \wedge \ \mathsf{inc}(u,x) \ \wedge \ \mathsf{inc}(v,x)) \tag{4.7}$$

which indicates that, if we invoke u and then invoke v, the latter returns 2 *rather than* 1. Since this is obviously not the case in $\mathcal{M}_2$, it is wrong to say $u = v$ in $\mathcal{M}_2$, justifying $\mathcal{M}_2 \models u \neq v$. Note that, in spite of this, $\mathcal{M}_2$ satisfies

$$\exists x.(!x = 0 \ \wedge \ \mathsf{inc}(u,x)) \ \wedge \ \exists x.(!x = 0 \ \wedge \ \mathsf{inc}(v,x)) \tag{4.8}$$

which indicates, if compared independently, u and v show precisely the same observable behaviour, witnessing the contextual congruence of the behaviour at u and the behaviour at v.

Now suppose $\mathcal{M}_2 \models u = v$. By definition this means $\mathcal{M}_2[w\!:\!u] \approx \mathcal{M}_2[w\!:\!v]$. Since $\binom{uw}{wu}$ is obviously a symmetry of $\mathcal{M}_2[w\!:\!u]$, by Proposition 4.20, it should also be a symmetry of $\mathcal{M}_2[w : v]$. But we have already seen it is not so in Example 4.11 (2) (page 18), a contradiction. Thus it cannot be the case $\mathcal{M}_2 \models u = v$, that is we have $\mathcal{M}_2 \models u \neq v$, as required.

Formal properties of the satisfaction relation (such as satisfiability of standard axioms) will be studied in Section 6.

5 Judgement and Proof Rules

5.1 Judgement and its Semantics

This section introduces judgements and basic proof rules. A judgement consists of a program and a pair of formulae following Hoare [14], augmented with a fresh name called *anchor* [17–19].

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

which intuitively 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.*

Note the judgement is about total correctness.[4] Sequents have identical shape as those in [3, 19]: the described computational situations is however quite different, where both C and C' may describe 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). In $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$:

1. u is the *anchor* of the judgement, which should *not* be in $\mathsf{dom}(\Gamma,\Delta)\cup\mathsf{fv}(C)$; and
2. C is the *pre-condition* and C' is the *post-condition*.

An anchor is used for naming the value from M and for specifying its behaviour. As in Hoare logic, the distinction between primary and auxiliary names plays an important role in both semantics and derivations.

Definition 5.1 (primary/auxiliary names) *In* $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, *the* primary names *are* $\mathsf{dom}(\Gamma,\Delta)\cup\{u\}$, *while the* auxiliary names *are those free names in C and C' which are not primary.*

Convention 5.2 *Henceforth we assume judgements are always well-typed (including those in the conclusions of proof rules), in the sense that, in* $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, $\Gamma,\Delta,\Theta\vdash C$ *and* $u:\alpha,\Gamma,\Delta,\Theta\vdash C'$, *for some* Θ *such that* $\mathsf{dom}(\Theta)\cap(\mathsf{dom}(\Gamma,\Delta)\cup\{u\}) = \emptyset$. *In spite of this, we often omit type information from a program in a judgement, writing* $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ *(under monomorphic typing, this in fact does not lose precision).*

We now make precise the semantics of judgement.

Definition 5.3 (semantics of judgement) Let $\mathcal{M} \stackrel{\text{def}}{=} (\nu\tilde{x})(\xi,\sigma)$ be of type $\Gamma;\Delta;\mathcal{D}$, and $\Gamma;\Delta\vdash N:\alpha$ with u fresh. The validity $\models\{C\}M :_u \{C'\}$ is given by:

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

(note free names stay Above we demand, for well-definedness, that $\mathcal{M}$ includes all variables in M, C and C' except u.

[4] Total correctness was chosen following [3, 18, 19]. The proof rules for partial correctness are essentially identical except for recursion.

Notation 5.4 (judgement for freshness) We use the following abbreviation for judgements similar to those with evaluation formulae in § 3.3. Below let i be fresh.

- $\{C\}M\{C'\}$ stands for $\{C\}M :_u \{u = () \wedge C'\}$.
- $\{C\}M :_u \{C'\}@\tilde{e}$ stands for $\{C \wedge y \neq \tilde{e} \wedge !y = i\}M :_u \{C' \wedge !y = i\}$ with y fresh (to be precise, y and i are respectively typed as $\mathsf{Ref}(\mathrm{X})$ and X for a fresh X).
- $\{C\}\ M :_m \{\nu x.C'\}$ stands for $\{C\}\ M :_m \{\exists x.(x \neq i \wedge C')\}$ (to be precise, i is typed as X for a fresh X, similarly in the following).
- $\{C\}\ M :_m \{\#m.C'\}$ stands for $\{C\}\ M :_m \{m \# i \wedge C'\}$.
- $\{C\}\ M :_m \{\star m.C'\}$ stands for $\{C\}\ M :_m \{m \star i \wedge C'\}$.

We may also combine these forms as in $\{C\}\ M :_m \{\nu x.C'\}@\tilde{y}$ and $\{C\}\ M :_m \{\nu \# x.C'\}@\tilde{y}$.

5.2 Proof Rules (1): Basic Proof System

The proof rules in a compositional program logic are given as a proof system which builds assertions following the syntactic structure of programs. Since we are working with a typed programming language, the proof rules precisely follow the typing rules presented in Section 2.2. The proof system is augmented with *structural rules* which only manipulate formulae without changing the program itself. These structural rules often play a fundamental role in deriving/validating required specifications for programs. In contrast, the proof rules which follow the syntactic structure of programs may be called compositional proof rules. Figure 2 presensts the full compositional proof rules in the present logic. There is one rule for each typing rule. Some of the major structural rules are presented in Figure 3.

For each rule, we stipulate:

- Free $i, j, \ldots$ range over auxiliary names; no primary names in the premise(s) occur as auxiliary names in the conclusion (this may be considered as a variant of the bound name convention).
- $A, A', B, B', \ldots$ range over *stateless formulae*, i.e. those formulae which do not contain active dereferences (a dereference $!e$ is *active* if it does not occur in pre/post conditions of evaluation formulae nor under the scope of content quantification of $!e$). We also exclude $x \# y$ and $\neg A$ from stateless formulae (they implicitly contain dereference: see Section 6.2).

In the following, we focus two new aspects of the presented proof rules: the proof rule for new reference generation, and the proof rule for universal quantification and the consequence rule with evaluation formula in Figure 3.

Proof Rule for New Reference Generation. The compositional rules in Figure 2 stay identical with those in the base logic [3] except for adding the rule for new name generation, in spite of the significant semantic enrichment. [*Ref*] says that the newly generated cell is unreachable from any datum in the initial state, and stores what M evaluates to in the reference named by u .

Fig. 2 Compositional Rules

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

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

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

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

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

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

$$[Proj_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

$$[In_1]\ \frac{\{C\}\ M :_v \{C'[\mathtt{inj}_1(v)/u]\}}{\{C\}\ \mathtt{inj}_1(M) :_u \{C'\}}$$

$$[Case]\ \frac{\{C^{\text{-}x_1x_2}\}\ M :_m \{C_0^{\text{-}x_1x_2}\} \quad \{C_0[\mathtt{inj}_i(x_i)/m]\}\ M_i :_u \{C'^{\text{-}x_1x_2}\} \quad i \in \{1,2\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{inj}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

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

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

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

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

Variants for New Name Generation. In the side condition of [*New*], $\mathsf{fpn}(e)$ denotes the set of *free plain names* of e which are reference names in e that occur without being dereferenced. Formally $\mathsf{fpn}(e)$ is inductively defined as: $\mathsf{fpn}(x) \overset{\text{def}}{=} \{x\}$, $\mathsf{fpn}(\mathsf{c}) = \mathsf{fpn}(!e) \overset{\text{def}}{=} \emptyset$, for other constructs it acts homomorphically. $\mathsf{fpn}(\tilde{e})$ is short for $\cup_i \mathsf{fpn}(e_i)$, $x \# \tilde{e}$ for $\wedge_i\ x \# e_i$. This new name generation rule reads, in direct correspondence with the reduction of the "new" construct (cf. §2.1):

> *Assume:* (1) *starting from C, the evaluation of M reaches C_0, with the resulting value named m; and* (2) *starting from C_0 with m as content of x ($C_0[!x/m]$) together with the assumption x is unreachable from any existing datum ($x \# \tilde{e}$), the evaluation of N reaches C with the resulting value named u. Then starting from C, $\mathtt{new}\ x := M\ \mathtt{in}\ N$ with its result named u reaches C'.*

Fig. 3 Structural rules

$$[\mathit{Promote}]\ \frac{\{C\}\ V :_u \{C'\}}{\{C \wedge C_0\}\ V :_u \{C' \wedge C_0\}} \qquad [\mathit{Consequence}]\ \frac{C \supset C_0 \quad \{C_0\}\ M :_u \{C'_0\} \quad C'_0 \supset C'}{\{C\}\ M :_u \{C'\}}$$

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

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

$$[\mathit{Aux}_{\exists}]\ \frac{\{C\}\ M :_u \{C'^{\,\text{-}i}\}}{\{\exists i.C\}\ M :_u \{C'\}} \qquad [\mathit{Aux}_{\forall}]\ \frac{\{C^{\text{-}i}\}\ M :_u \{C'\} \quad i \text{ is of a base type}}{\{C\}\ M :_u \{\forall i.C'\}}$$

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

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

$$[\mathit{Consequence\text{-}Aux}]\ \frac{\{C_0\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'_0\} \quad C\ \supset\ \exists \tilde{j}.(\ C_0[\tilde{j}/\tilde{i}]\ \wedge\ [!\tilde{e}](C'_0[\tilde{j}/\tilde{i}] \supset C')\ )}{\{C\}\ M :_u \{C'\}}$$

$$[\mathit{ConsEval}]\ \frac{\begin{array}{c}\{C_0\}\ M :_m \{C'_0\} \quad x \text{ fresh; } \tilde{i} \text{ auxiliary} \\ \forall \tilde{i}.\{C_0\} x \bullet () = m\{C'_0\} \supset \forall \tilde{i}.\{C\} x \bullet () = m\{C'\}\end{array}}{\{C\}\ M :_m \{C'\}}$$

In [*Consequence-Aux*], we let $!\tilde{e}$ (resp. $\tilde{i}$) exhaust active dereferences (resp. auxiliary names) in C, C', C_0, C'_0, while $\tilde{j}$ are fresh and of the same length as $\tilde{i}$.

Fig. 4 Variants of [*Ref*] and Other Related Rules.

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

$$[\mathit{NewVar}]\ \frac{\{C\}\ M :_m \{C_0\} \quad \{C_0[!x/m] \wedge x \# \tilde{e}\}\ N :_u \{C'\} \quad x \notin \mathsf{fpn}(\tilde{e}) \cup \mathsf{fv}(C')}{\{C\}\ \mathtt{new}\ x\ := M\ \mathtt{in}\ N :_u \{C'\}}$$

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

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

$$[\mathit{Ref}']\ \frac{\{C\}\ M :_m \{C'\} \quad u \notin \mathsf{fpn}(\tilde{e})}{\{C\}\ \mathtt{ref}(M) :_u \{C\ \wedge\ u \# \tilde{e}\}}$$

Some remarks:

– The side condition $x \notin \mathsf{fpn}(\tilde{e})$ is essential for the consistency of the rule: as an extreme case, $x\#x$ is obviously false, similarly $x\#\langle x,x\rangle$ (cf. § 3.3). However $x\#!x$ is *not* falsity, and in fact *should* hold in the pre-condition for N: immediately after a new reference is generated, it cannot be stored anywhere.
– The intermediate condition, $x\#\tilde{e}$, gives a much stronger notion of freshness in comparison with the postcondition in the conclusion, $\nu x.C'$, i.e. $\exists x.(C' \wedge x \neq i)$ for fresh i. This is because, during the run of N, x may have been stored somewhere: so all that we can say is x is distinct from any reference name in the initial configuration (C' may assert stronger freshness conditions for x depending on N's behaviour).

We may also observe that the proof rule has an equipotent variant, listed as the first rule ([*NewVar*]) in Figure 4. [*NewVar*] is equi-potent with the original rule when combined with the consequence rule. Indeed, if we have [*NewVar*] and if we assume the premise of [*New*], we simply add $x \neq i$ to the pre/post condition of the second judgement by [*StatelessInv*], then use [*Consequence*] to existentially abstract x, to reach the premise, hence the conclusion, of [*Newvar*]. Conversely, if we have [*New*] and if we assume the premise of [*NewVar*], since the premise directly fits as a premise for [*New*], we obtain $\{C\}\ \mathtt{new}\ x\ := M\ \mathtt{in}\ N :_u \{\nu x.C'\}$: but $\nu x.C' \equiv C'$ since x does not occur free in C'. In inference, [*New*] is more convenient when we derive the conclusion from the premise ("*forward reasoning*"): whereas [*NewVar*] follows the conventional style of compositional proof rules, starting from the conclusion to find the premise ("*backward reasoning*"). Note also [*NewVar*] is essentially identical with the original proof rule for new variable declaration by Hoare and Wirth [13] except adding the condition for unreacability.

In Section 2, we discussed interplay between freshness and locality. In particular, we asked how $\mathtt{ref}(M)$, combined with $\mathtt{let}$, can commute with $\mathtt{new}\ x := M\ \mathtt{in}\ N$. This question is best answered by mutual derivations of their proof rules. Below we show derivation in one direction, deriving [*New*] from the rule for $\mathtt{ref}(M)$, listed as the second rule in Figure 4, leaving the other direction to the reader. [*Ref*] says that the newly generated cell is unreachable from any initially existing datum. We further use the next two rules in Figure 4. [*Subs*] is the standard substitution rule (the rule implicitly assumes, by our convention of typability of judgements, that $u \notin \mathsf{fn}(e)$ when i occurs in C: when i does *not* occur in C, the side-condition demands u not to occur in $\mathsf{fpn}(e)$). [*LetOpen*] is a refinement of the standard let rule:

$$[\mathit{Let}]\ \frac{\{C\}\ M :_x \{C_1\} \quad \{C_1\}\ N :_u \{C'\}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{C'\}}$$

(There is some interest in deriving [*LetOpen*] from [*Let*], which we shall discuss later.) Using the encoding in (2.1) and the proof rule above, as well as others, we can derive

[***New***] 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. $\{C\}\ \mathtt{ref}(M) :_x \{\nu y.(C_0[!x/m] \wedge x\#\tilde{e} \wedge x = y)\}$	(Consequence)
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. $\{C\}\ \mathtt{let}\ x = \mathtt{ref}(M)\ \mathtt{in}\ N :_u \{\nu x.C'\}$	(Consequence)

As may be observed, the crucial step in the above derivation is Line 5, which *turns freshness into locality through the standard law of equality and existential*,

$$C \quad \equiv \quad \exists y.(C \wedge x = y)$$

with y fresh. This also indicates that allowing ν-bound name (here y) to be equated with even free names (here x) as in the post-condition of Line 5 is inevitable if we wish to have locality (new) and freshness (ref) interact consistently. This may also offer a case for the use of existential quantifier in the present setting.

We also record a variant of the [*Ref*] as the last rule in Figure 4, suggested in the derivation above. Note the rule is close to the premise of [*New*] and [*NewVar*].

The proof rule [*LetOpen*] used above, which opens the scope of $\tilde{y}$ and which is often useful, is derivable from [*Let*] and a couple of structural rules, including [*Consequence*]. We list below its derivation which elucidates how smoothly the representation of freshness in the present logic integrates with compositional logic for sequential languages. The derivation follows (the premises are those of [*LetOpen*]; for simplicity we treat a single name ν-binder, whose generalisation is obvious; and we assume i is fresh).

1. $\{C\}\ M :_m \{\exists y.(y \neq i\ \wedge\ C_0)\}$	(premise)
2. $\{C_0\}\ N :_u \{C'\}$	(premise)
3. $\{y \neq i\ \wedge\ C_0\}\ N :_u \{y \neq i\ \wedge\ C'\}$	(2, Invariance)
4. $\{y \neq i\ \wedge\ C_0\}\ N :_u \{\exists y.(y \neq i\ \wedge\ C')\}$	(3, Consequence)
5. $\{\exists y.(y \neq i\ \wedge\ C_0)\}\ N :_u \{\exists y.(y \neq i\ \wedge\ C')\}$	(4, Aux$_\exists$)
6. $\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{\nu y.C'\}$	(1, 5, Let)

Observe how $y \neq i$ slides from the post-condition of a first judgement to later ones. Similar scope opening rules are derivable from other rules with non-trivial premises.

Let us list a small example from the Introduction, (1.3), a program which always returns 100 because of a freshness of local reference, reproduced below.

$$M \stackrel{\text{def}}{=} \mathtt{new}\ y := 1\ \mathtt{in}\ \mathtt{if}\ x = y\ \mathtt{then}\ 0\ \mathtt{else}\ 100 \tag{5.1}$$

The desired judgement is $\{\mathsf{T}\}M :_m \{m = 100\}$. For deriving this judgement, we use:

$$[\mathit{Erase}]\ \frac{\{C\}\ M :_m \{\nu x.C'\} \quad x \notin \mathsf{fn}(C')}{\{C\}\ M :_m \{C'\}}$$

The derivation follows, omitting trivial reasoning.

1. $\{\mathsf{T}\}\ 1 :_m \{\mathsf{T}\}$
2. $\{x \neq y\}$ $\texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ 100 :_u \{u = 100\}$
3. $\{y \# x\}$ $\texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ 100 :_u \{u = 100\}$ (2, Consequence)
4. $\{\mathsf{T}\}$ $\texttt{new}\ y := 1\ \texttt{in}\ \texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ 100 :_u \{\nu x.u = 100\}$ (1,3,New)
5. $\{\mathsf{T}\}$ $\texttt{new}\ y := 1\ \texttt{in}\ \texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ 100 :_u \{u = 100\}$ (4, Erase)

Above Line 2 uses the entailment

$$y \# x \quad \supset \quad x \neq y$$

which is immediate from $x = y \supset x \hookrightarrow y$. Final lines could also have used [*NewVar*] instead of [*New*].

Proof Rule for Universal Abstraction. The structural rules in Figure 3 stay identical with those in the preceding logic [3] except for a slight change in $[Aux_{\forall}]$. Without this side condition, the rule is not sound. As a simple example, [*Ref'*] immediately gives us:

$$\{\mathsf{T}\}\ \texttt{ref}(0) :_u \{u \neq i\} \tag{5.2}$$

from which we can*not* conclude

$$\{\mathsf{T}\}\ \texttt{ref}(0) :_u \{\forall i.(u \neq i)\} \tag{5.3}$$

since if so we can substitute u for i by the standard law of universal quantification, reaching an inconsistent judgement $\{\mathsf{T}\}\ \texttt{ref}(0) :_u \{\mathsf{F}\}$.

Let us further clarify the semantic stauts of (5.2) and its difference from (5.3). Semantically, (5.2) says that, for each $\mathcal{M}$ (implicitly assuming typability), we have, by referring to the definition of expansion (Definition 4.5, page 17) and semantics of judgement (Definition 5.3, page 23):

$$\mathcal{M}[u:\texttt{ref}(0)] \Downarrow (\nu l)(\mathcal{M} \cdot u : l,\ l \mapsto 0) \models u \neq i$$

with l fresh. Note i should be interpreted in $\mathcal{M}$, i.e. it is in the domain of ξ. This means either i is a non-reference in which case $u \neq i$ is immediate, or, if i is a reference, it cannot be mapped to the same label as u, hence again we have $u \neq i$. This is how the inequality $u \neq i$ for a fresh i can represent freshness of u.

In contrast, (5.3) says that, again for each $\mathcal{M}$, we have:

$$\mathcal{M} \cdot u : l, \quad l \mapsto 0 \models \forall i.(u \neq i)$$

from which we can deduce, by the semantics of universal abstraction:

$$\mathcal{M} \cdot u, i : l, \quad l \mapsto 0 \models u \neq i$$

which is a straightforward contradiction. This contradiction is caued simply because i can be coalesced with i: the quantification in the postcondition is effective *after* the post-state is obtained, so i can denote anything in the poststate. In (5.2), i can only be interpreted in $\mathcal{M}$ outside of u simply because we know i should already exist in $\mathcal{M}$ and because u is added as a singleton (or unaliased) name in the postcondition, so that i cannot be equated with u. This is essentially why [*New*] rule is sound, whose formal proof is given in Section 6.

5.3 Proof Rules (2): Derived Located Rules

We find several rules derivable or admissible in the basic proof system to be of substantial help in reasoning about programs. The first class of these rules are for located judgements, listed in Figure 5 and Figure 6. All rules come from [3] except for the new name generation rule and the universal quantification rule, both corresponding to the new rules in the basic proof system. In [*New*] rule, x should be taken off from the write effects (by the freshness condition, other write effects are guaranteed to be distinct from x). In the structural rules for located assertions in Figure 6, [*Invariance*] generalises the rule of the same name in Figure 3, and is used for modular and extensional reasoning for programs with write effects.

These rules stay as they originally are in [3] in spite of a major change in semantics of assertions. A central reason for this is that the operators on models used for interpreting the present logic discussed in Section 4 work precisely in the same way as the set-theoretic operators used for interpreting the logic in [3] (and [19]), up to a certain abstraction level (for this purpose the behavioural aspect of these operators, or more directly their $\approx$-closure, cf. Section 4.5, is essential). Thus the soundness of each proof rule follows precisely the same reasoning as the one given in the preceding models, the latter based on set-theoretic operations. Let us see how this is so taking the invariance rule, [*Invariance*], as an example. Recall that we say C is $!e$*-free* if $[!e]C \equiv C$. By its premise we assume:

$$\models \{C\} M :_u \{C'\} @ e \tag{5.4}$$

where for simplicity we consider a single write effect, which is easily extended. By the $!e$-freeness of C_0, we have:

$$\forall \mathcal{M}_0, \forall V. (\mathcal{M}_0 \models C_0 \equiv \mathcal{M}_0[e \mapsto V] \models C_0). \tag{5.5}$$

Now suppose $\mathcal{M} \models C \wedge C_0$, that is we have (a) $\mathcal{M} \models C$ and (b) $\mathcal{M} \models C_0$. By (a) and (5.4) we know:

$$\mathcal{M}[u : M] \Downarrow \mathcal{M}' \models C' \quad \text{and} \quad \exists W. (\mathcal{M}'/u = \mathcal{M}[e \mapsto W]) \tag{5.6}$$

The second half in (5.6) is by the write effect. (5.6), (b) and (5.5) indicate: $\mathcal{M}' \models C_0$, Hence we reach $\mathcal{M}[u : M] \Downarrow \mathcal{M}' \models C' \wedge C_0$.

As we have already explored in our previous work [3, 19], programs of specific shape allow efficient reasoning rules, some of which are listed in Figure **??**. More specific rules for freshness will be introduecd in § 7.

Fig. 5 Derived compositional rules for located assertions

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

$$[\mathit{Add}]\ \frac{\{C\}\,M_1 :_{m_1} \{C_0\}@\tilde{e}_1 \quad \{C_0\}\,M_2 :_{m_2} \{C'[m_1+m_2/u]\}@\tilde{e}_2}{\{C\}\,M_1+M_2 :_u \{C'\}@\tilde{e}_1\tilde{e}_2}$$

$$[\mathit{Abs}]\ \frac{\{C \wedge A^{\text{-}x}\}\ M :_m \{C'\}@\tilde{e}}{\{A\}\ \lambda x.M :_u \{\{C\}u \bullet x = m\{C'\}@\tilde{e}\}@\emptyset}$$

$$[\mathit{App}]\ \frac{\{C\}\,M :_m \{C_0\}@\tilde{e}\ \ \{C_0\}\,N :_n \{C_1 \wedge \{C_1\}m \bullet n = u\{C'\}@\tilde{e}_2\}@\tilde{e}_1}{\{C\}\ MN :_u \{C'\}@\tilde{e}\tilde{e}_1\tilde{e}_2}$$

$$[\mathit{If}]\ \frac{\{C\}\ M :_b \{C_0\}@\tilde{e}_1 \quad \{C_0[\mathsf{t}/b]\}\ M_1 :_u \{C'\}@\tilde{e}_2 \quad \{C_0[\mathsf{f}/b]\}\ M_2 :_u \{C'\}@\tilde{e}_2}{\{C\}\ \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 :_u \{C'\}@\tilde{e}_1\tilde{e}_2}$$

$$[\mathit{Pair}]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\}@\tilde{e}_1 \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1, m_2\rangle/u]\}@\tilde{e}_2}{\{C\}\ \langle M_1, M_2\rangle :_u \{C'\}@\tilde{e}_1\tilde{e}_2}$$

$$[\mathit{Proj}_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}@\tilde{e}}{\{C\}\ \pi_1(M) :_u \{C'\}@\tilde{e}}$$

$$[\mathit{In}_1]\ \frac{\{C\}\ M :_v \{C'[\mathtt{inj}_1(v)/u]\}@\tilde{e}}{\{C\}\ \mathtt{inj}_1(M) :_u \{C'\}@\tilde{e}}$$

$$[\mathit{Case}]\ \frac{\{C^{\text{-}x_1x_2}\}\ M :_m \{C_0^{\text{-}x_1x_2}\}@\tilde{e}_1 \quad \{C_0[\mathtt{inj}_i(x_i)/m]\}\ M_i :_u \{C'^{\text{-}x_1x_2}\}@\tilde{e}_2 \quad i \in \{1,2\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{inj}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}@\tilde{e}_1\tilde{e}_2}$$

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

$$[\mathit{Assign}]\ \frac{\{C\}\ M :_m \{C_0\}@\tilde{e}_1 \quad \{C_0\}\ N :_n \{C'\{\!|n/\,!m|\!\}\}@\tilde{e}_2 \quad C_0 \supset m = e'}{\{C\}\ M := N\ \{C'\}@\tilde{e}_1\tilde{e}_2e'}$$

$$[\mathit{New}]\ \frac{\{C\}\ M :_m \{C_0\}@\tilde{g}_1 \quad \{C_0[!x/m] \wedge x\#\tilde{e}\}\ N :_u \{C'\}@\tilde{g}_2x \quad x \notin \mathsf{fpn}(\tilde{e}) \cup \mathsf{fv}(\tilde{g}_1\tilde{g}_2)}{\{C\}\ \mathtt{new}\ x := M\ \mathtt{in}\ N :_u \{\nu x.C'\}@\tilde{g}_1\tilde{g}_2}$$

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

6 Axioms, Soundness and Observational Completeness

6.1 Axioms for Reachability

This subsection shows axioms of reachability predicate and its negation. There are three non-standard logical primitives in the present assertion language.

1. Evaluation formulae (for imperative higher-order functions).
2. Content quantification (for aliasing).
3. Reachability (for local state).

Fig. 6 Derivable structural rules for located judgements.

$$[\mathit{Promote}]\ \frac{\{C\}\ V :_u \{C'\}@\emptyset}{\{C \wedge C_0\}\ V :_u \{C' \wedge C_0\}@\emptyset} \qquad [\mathit{Consequence}]\ \frac{C \supset C_0 \quad \{C_0\}\ M :_u \{C_0'\}@\tilde{e} \quad C_0' \supset C'}{\{C\}\ M :_u \{C'\}@\tilde{e}}$$

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

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

$$[\mathit{Aux}_\exists]\ \frac{\{C\}\ M :_u \{C'^{\,\text{-}i}\}@\tilde{e}}{\{\exists i.C\}\ M :_u \{C'\}@\tilde{e}} \qquad [\mathit{Aux}_\forall]\ \frac{\{C^{\text{-}i}\}\ M :_u \{C'\}@\tilde{e} \quad i \text{ is of a base type.}}{\{C\}\ M :_u \{\forall i.C'\}@\tilde{e}}$$

$$[\mathit{Invariance}]\ \frac{\{C\}\ M :_u \{C'\}@\tilde{e} \quad C_0 \text{ is } !\tilde{e}\text{-free}}{\{C\ \wedge\ C_0\}\ M :_u \{C' \wedge C_0\}@\tilde{e}}$$

$$[\mathit{Weak}]\ \frac{\{C\}\ M :_m \{C'\}@\tilde{e}}{\{C\}\ M :_m \{C'\}@\tilde{e}e'} \qquad [\mathit{Thinning}]\ \frac{\{C \wedge !e' = i\}\ M :_m \{C' \wedge !e' = i\}@\tilde{e}e' \quad i \text{ fresh}}{\{C\}\ M :_m \{C'\}@\tilde{e}}$$

$$[\mathit{Consequence\text{-}Aux}]\ \frac{\{C_0\}\ M^{\Gamma;\Delta;\alpha} :_u \{C_0'\}@\tilde{e} \quad C\ \supset\ \exists \tilde{j}.(\ C_0[\tilde{j}/\tilde{i}]\ \wedge\ [!\tilde{e}](C_0'[\tilde{j}/\tilde{i}] \supset C')\ )}{\{C\}\ M :_u \{C'\}@\tilde{e}}$$

In [*Consequence-Aux*], we let $!\tilde{e}$ (resp. $\tilde{i}$) exhaust active dereferences (resp. auxiliary names) in C, C', C_0, C_0', while $\tilde{j}$ are fresh and of the same length as $\tilde{i}$.

The axioms for the first two constructs are respectively treated in [19] and [3], including their interplay with each other. We list basic axioms of reachability, which are useful for reasoning examples later.

Lemma 6.1 *Assume $\mathcal{M} \models x\#\tilde{y}\tilde{l}$ and $\mathsf{fv}(N) \cup \mathsf{fl}(N) \subseteq \tilde{y}\tilde{l}$. Then for all N with u fresh, $\mathcal{M}[u\!:\!N] \Downarrow \mathcal{M}'$ implies $\mathcal{M}' \models x\#u\tilde{y}\tilde{l}$.*

Proposition 6.2 (axioms for reachability) . *The following assertions are valid (we assume appropriate typing).*

1. (1) $x \hookrightarrow x$; (2) $x \hookrightarrow y \wedge y \hookrightarrow z\ \supset\ x \hookrightarrow z$; (3) $x\#w \wedge w \hookrightarrow u\ \supset\ x\#u$.
2. (1) $y\#x^{\alpha}$ *with* $\alpha \in \{\mathsf{Unit}, \mathsf{Nat}, \mathsf{Bool}\}$; (2) $\langle x_1, x_2 \rangle \hookrightarrow y\ \equiv\ x_1 \hookrightarrow y \vee x_2 \hookrightarrow y$;
   (3) $\mathtt{inj}_i(x) \hookrightarrow y\ \equiv\ x \hookrightarrow y$; (4) $x \hookrightarrow y^{\mathsf{Ref}(\alpha)}\ \supset\ x \hookrightarrow !y$;
   (5) $x^{\mathsf{Ref}(\alpha)} \hookrightarrow y \wedge x \neq y\ \supset !x \hookrightarrow y$.
3. (1) $\{C \wedge x\#fy\}\, f\bullet y\!=\!z\{C'\} \supset \{C \wedge x\#fy\}\, f\bullet y\!=\!z\{C' \wedge x\sharp z\}$;
   (2) $\{C \wedge x\#fy\tilde{w}\}\, f\bullet y\!=\!z\{C'\}@\tilde{w} \supset \{x\#fy\tilde{w}\}\, f\bullet y\!=\!z\{C' \wedge x\sharp zw\}@\tilde{w}$;
   (3) $\{x\#fy\tilde{w} \wedge C\}\, f\bullet y\!=\!z\{C'\}@\tilde{w} \supset \{x\#fy\tilde{w} \wedge C\}\, f\bullet y\!=\!z\{x\#fyz\tilde{w} \wedge C'\}@\tilde{w}$.

PROOF: Axioms in 1 and 2 use Lemma 4.23. Axiom 1-(1) is direct by Lemma 4.23 (1), while Axiom 1-(2) is by Lemma 4.23 (5). Axiom 1-(3) is proved by a contradiction.

Fig. 7 Other derived located proof rules.

$$[\mathit{AssignVar}]\ \frac{C\{\!|e/!x|\!\} \supset x = g}{\{C\{\!|e/!x|\!\}\}\ x := e\ \{C\}@g} \qquad [\mathit{AssignSimple}]\ \frac{C\{\!|e'/!e|\!\} \supset e = g}{\{C\{\!|e'/!e|\!\}\}\ e := e'\ \{C\}@g}$$

$$[\mathit{IfSimple}]\ \frac{\{C \wedge e\}\ M_1\ \{C'\}@\tilde{g} \quad \{C \wedge \neg e\}\ M_2\ \{C'\}@\tilde{g}}{\{C\}\ \mathtt{if}\ e\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2\ \{C'\}@\tilde{g}}$$

$$[\mathit{AppSimple}]\frac{C \supset \{C\}\, e \bullet (e_1..e_n) = u\, \{C'\}@\tilde{g}}{\{C\}\ e(e_1..e_n) :_u \{C'\}@\tilde{g}}$$

$$[\mathit{NewSimple}]\ \frac{\{C \wedge !x = e \wedge x\#\tilde{e'}\}\ N :_u \{C'\}@\tilde{g}x \quad x \notin \mathsf{fpn}(\tilde{e'}) \cup \mathsf{fv}(\tilde{g})}{\{C\}\ \mathtt{new}\, x\ := e\ \mathtt{in}\, N :_u \{\nu x.C'\}@\tilde{g}}$$

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

$$[\mathit{Seq}]\ \frac{\{C\}\ M\ \{C_0\}@\tilde{g} \quad \{C_0\}\ N\ \{C'\}@\tilde{g'}}{\{C\}\ M;N\ \{C'\}@\tilde{g}\tilde{g'}} \qquad [\mathit{Seq\text{-}Inv}]\ \frac{\{C_1\}\ M\ \{C_1'\}@\tilde{e_1} \quad \{C_2\}\ N\ \{C_2'\}@\tilde{e_2}}{\{C_1 \wedge [!\tilde{e_1}]C_2\}\ M;N\ \{C_2' \wedge \langle !\tilde{e_2}\rangle C_1'\}@\tilde{e_1}\tilde{e_2}}$$

Assume $\neg w \hookrightarrow x \wedge w \hookrightarrow u$ but $u \hookrightarrow x$. Then by transitivity, we have $w \hookrightarrow x$, which contradicts $x\#w$. Axiom 2 (1) is trivial by $\mathsf{fv}(\sigma(x)) = \emptyset$. Axiom 2 (2) is by Lemma 4.23 (4), while Axiom 2 (3) is by definition of the name closure. Axiom 2 (4) is by Lemma 4.23 (6). Axiom 2 (5) is by Lemma 4.23 (2) and (3). The proof of Axiom 3 (1,2) are subsumed by that of Axiom 3 (2) below. Axiom 3 (2) is proved by Proposition 4.24 and the definition of the model of the evaluation formula. Suppose $\mathcal{M} \models \{x\#fyw \wedge C\}\, f \bullet y = z\{C'\}@w$. The definition of the evaluational formula says, with u fresh,

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

We prove such $\mathcal{M}'$ always satisfies $\mathcal{M}' \models x\#zw$. Assume $\mathcal{M}_0 \approx (\nu\vec{l})(\xi, \sigma_0 \uplus \sigma_x)$ with $\xi(x) = l$, $\xi(y) = V_y$, $\xi(f) = V_f$ and $\xi(w) = l_w$ such that $cl(\mathsf{fl}(V_f, V_y, l_w), \sigma_0 \uplus \sigma_x) = \mathsf{fl}(\sigma_0) = \mathsf{dom}(\sigma_0)$ and $l_x \in \mathsf{dom}(\sigma_x)$. By this partition, during evaluation of $z : fy$, σ_x is unchanged, i.e. $(\nu\vec{l})(\xi \cdot z : fy, \sigma_0 \uplus \sigma_x) \longrightarrow\!\!\!\!\rightarrow (\nu\vec{l})(\xi \cdot z : V_f V_y, \sigma_0 \uplus \sigma_x) \longrightarrow\!\!\!\!\rightarrow (\nu\vec{l'})(\xi \cdot z : V_z, \sigma_0' \uplus \sigma_x)$. Then obviously there exists σ_1 such that $\sigma_1 \subset \sigma_0'$ and $cl(\mathsf{fl}(V_z, l_w), \sigma_0' \uplus \sigma_x) = \mathsf{fl}(\sigma_1) = \mathsf{dom}(\sigma_1)$. Hence by Proposition 4.24, we have $\mathcal{M}_0 \models x\#wz$, completing the proof. □

3 above says that if x is unreachable from a function f, its argument y and its write variable w, then the same is true for its return value z and w. Note that $f = \lambda().x$ saitsifies $\{y = ()\}\, f \bullet y = z\{y = ()\}@\emptyset$ but does not $\{y = ()\}\, f \bullet y = z\{y = () \wedge x\#wz\}@\emptyset$ hence $x\#f$ in the precondition is necessary to derive $x\#z$ in the postcondition. Similarly for $x\#f$ and $x\#y$.

6.2 Elimination Results

Finite Types 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. We show, for finite and recursively finite types, the (non-)reachability predicates can be eliminated from the assertion language, in the sense that each assertion containing such predicates has an equivalent assertion which does not use them.

Theorem 6.3 *Suppose any reachability predicates in C are finite. Then there exists C' such that $C \equiv C'$ and no reachability predicate occurs in C'.*

As the first step, we define a simple inductive method for defining reachability from a datum of a finite type.

Definition 6.4 (i-step reachability) *Let α be a finite type. Then the i-step reachability predicate* $\mathsf{reach}(x^{\alpha},\ y^{\mathsf{Ref}(\beta)},\ i^{\mathsf{Nat}})$ *(read:*"a reference y is reachable from x in at most i-steps"*) is inductively given as follows (below we assume y is typed* $\mathsf{Ref}(\beta)$, $C \in \{\mathsf{Unit}, \mathsf{Bool}, \mathsf{Nat}\}$*, and omit types when evident).*

$$\begin{aligned}
\mathsf{reach}(x^{\alpha},\ y,\ 0) &\equiv x = y \\
\mathsf{reach}(x^{C},\ y,\ n+1) &\equiv \mathsf{F} \\
\mathsf{reach}(x^{\alpha_1 \times \alpha_2},\ y,\ n+1) &\equiv \vee_i \mathsf{reach}(\pi_i(x),\ y,\ n)\ \vee\ \mathsf{reach}(x,\ y,\ n) \\
\mathsf{reach}(x^{\alpha_1 + \alpha_2},\ y,\ n+1) &\equiv \exists x'.(x' = \mathtt{inl}(x)\ \ \wedge\ \ \mathsf{reach}(x',\ y,\ n))\ \vee \\
&\quad\ \exists x'.(x' = \mathtt{inr}(x)\ \ \wedge\ \ \mathsf{reach}(x',\ y,\ n))\ \vee \\
&\quad\ \mathsf{reach}(x,\ y,\ n) \\
\mathsf{reach}(x^{\mathsf{Ref}(\alpha)},\ y,\ n+1) &\equiv \mathsf{reach}(!x,\ y,\ n)\ \vee\ \mathsf{reach}(x,\ y,\ n)
\end{aligned}$$

Remark 6.5 With C being a base type, $\mathsf{reach}(x^{C},\ y,\ 0) \equiv x = y \equiv \mathsf{F}$ (since a reference y cannot be equal to a datum of a base type).

A key lemma follows.

Proposition 6.6 *If α is finite, then the following logical equivalence is valid, i.e. is true in any model.*

$$x^{\alpha} \hookrightarrow y \equiv \exists i.\mathsf{reach}(x^{\alpha},\ y,\ i)$$

PROOF: For the "if" direction, we show, by induction on i, $\mathsf{reach}(x^{\alpha},\ y,\ i) \supset x^{\alpha} \hookrightarrow y$. For the base case, we have $i = 0$, in which case:

$$\begin{aligned}
\mathsf{reach}(x^{\alpha},\ y,\ 0)\ &\Rightarrow\ x = y \\
&\Rightarrow\ x \hookrightarrow y
\end{aligned}$$

For induction, let the statement holds up to n. We only show the case of a product. Other cases are similar.

$$\begin{aligned}
\mathsf{reach}(x^{\alpha_1 \times \alpha_2},\ y,\ n+1)\ &\Rightarrow\ \vee_i \mathsf{reach}(\pi_i(x),\ y,\ n)\ \vee\ \mathsf{reach}(x,\ y,\ n) \\
&\Rightarrow\ \vee_i \pi_i(x) \hookrightarrow y\ \vee\ x \hookrightarrow y
\end{aligned}$$

But if $\pi_1(x) \hookrightarrow y$ then $x \hookrightarrow y$ by the definition of reachability. Similarly when $\pi_2(x) \hookrightarrow y$, hence done.

For the converse, we show the contrapositive, showing:

$$\mathcal{M} \models \neg\exists i.\mathsf{reach}(x^\alpha,\ y,\ i) \quad \Rightarrow \quad \mathcal{M} \models \neg x^\alpha \hookrightarrow y$$

If we have $\mathcal{M} \models \neg\exists i.\mathsf{reach}(x^\alpha,\ y,\ i)$ with α finite, then the reference y is not among references hereditarily reachable from x (if it is, then either $x = y$ or y is the content of a reference reachable from x because of the finiteness of α, so that we can find some i such that $\mathcal{M} \models \mathsf{reach}(x^\alpha,\ y,\ i)$), hence done. □

Now let us define the predicate $x^\alpha \hookrightarrow' y^{\mathsf{Ref}(\beta)}$ with α finite, by the axioms given in Axiom 2 in § 6.1 which we reproduce below.

$$\begin{aligned}
x^{\mathsf{Unit}} \hookrightarrow' y^{\mathsf{Ref}(\beta)} &\equiv \mathsf{F}\\
x^{\mathsf{Bool}} \hookrightarrow' y^{\mathsf{Ref}(\beta)} &\equiv \mathsf{F}\\
x^{\mathsf{Nat}} \hookrightarrow' y^{\mathsf{Ref}(\beta)} &\equiv \mathsf{F}\\
x^{\alpha_1\times\alpha_2} \hookrightarrow' y^{\mathsf{Ref}(\beta)} &\equiv \exists x_{1,2}.(x = \langle x_1, x_2\rangle \wedge \textstyle\bigvee_{i=1,2} x_i \hookrightarrow' y)\\
x^{\alpha_1+\alpha_2} \hookrightarrow' y^{\mathsf{Ref}(\beta)} &\equiv \exists x'.((\textstyle\bigvee_{i=1,2} x = \mathtt{inj}_i(x'))\ \wedge\ x' \hookrightarrow' y)\\
x^{\mathsf{Ref}(\alpha)} \hookrightarrow' y^{\mathsf{Ref}(\beta)} &\equiv x = y \vee !x \hookrightarrow' y
\end{aligned}$$

The inductive definition is possible due to finiteness. We now show:

Proposition 6.7 *If α is finite, then the following logical equivalence is valid.*

$$x^\alpha \hookrightarrow' y^{\mathsf{Ref}(\beta)} \quad \equiv \quad \exists i.\mathsf{reach}(x^\alpha,\ y^{\mathsf{Ref}(\beta)},\ i)$$

PROOF: $\mathsf{reach}(x^\alpha,\ y^{\mathsf{Ref}(\beta)},\ i) \supset x^\alpha \hookrightarrow' y^{\mathsf{Ref}(\beta)}$ is by induction on i. The converse is by induction on α. Both are mechanical and omitted. □

Corollary 6.8 *If α is finite, then the logical equivalence*

$$x^\alpha \hookrightarrow y^{\mathsf{Ref}(\beta)} \quad \equiv \quad x^\alpha \hookrightarrow' y^{\mathsf{Ref}(\beta)}$$

is valid, i.e. $\hookrightarrow$ is completely characterised by the axioms for $\hookrightarrow'$ given above.

PROOF: Immediate from Propositions 6.6 and 6.7. □

Recursively Types We say a type is *recursively finite* when it is closed and contains neither an arrow type nor type quantifiers. When α can be recursively finite, we define $\mathsf{reach}(x^\alpha,\ y^{\mathsf{Ref}(\beta)},\ i^{\mathsf{Nat}})$ by precisely the same clauses (taking the iso-recursive approach [31]). Then we again obtain, by precisely identical arguments:

Proposition 6.9 *If α is recursively finite, then the logical equivalence*

$$x^\alpha \hookrightarrow y \equiv \exists i.\mathsf{reach}(x^\alpha,\ y,\ i)$$

is valid, i.e. is true in any model.

Remark 6.10 The literally same argument as for Proposition 6.6 can be used for Proposition 6.9 because, regardless of α being finite or recursively finite, if V^{α} can reach y, then y is indeed found (possibly hereditarily going through the store) at a place where the type of y occurs in α. This property does not generally hold when α contains a function type, precluding simple operational reasoning based on disjointness.

A convenient axiomatisation of $\hookrightarrow$ and its negation in the presence of recursive finite types uses coinduction for the obvious reason: for example, to say y is not reachable from x, we can show the existence of a finite set of pairs including $\langle x, y\rangle$ which is closed under one-step reachability and which shows no datum reachable from x does not coincide with y. For such assertions, we need a small increment in the grammar of assertions. First, terms are incremented with finite relations:

$$e \quad ::= \quad \ldots \quad | \quad \emptyset \quad | \quad \mathcal{R}$$

(We may type these expressions with appropriate types.) We also extend the assertion with:

$$C \quad ::= \quad \ldots \quad | \quad \langle x, y\rangle \in \mathcal{R}$$

For effectively asserting on, and reasoning about, finite relations, we may use simple axioms for set membership as well as basic operations on sets.

Co-inductive axiomatisation specifies a property of a finite relation, saying if $\langle x, y\rangle$ is related by $\mathcal{R}$, and $\mathcal{R}$ satisfies a certain property, then $x \hookrightarrow y$. The axiomatisation focusses on a certain closure property of $\mathcal{R}$ and is given in Figure 8.

Fig. 8 Axioms for (Non-)Reachability (recursive finite case: α, β are recursively finite)

$$x^{\alpha} \hookrightarrow y^{\mathsf{Ref}(\beta)} \quad \equiv \quad \exists \mathcal{R}.(\langle x, y\rangle \in \mathcal{R} \ \wedge \ \Psi(\mathcal{R}))$$

where $\Psi(\mathcal{R})$ is given by the conjunction of:

$$\begin{array}{rcl}
\forall \mathrm{X}_{1,2}, z.\ \langle z^{\mathrm{X}_1 \times \mathrm{X}_2}, y\rangle \in \mathcal{R} & \supset & \vee_{i=1,2} \langle \pi_i(z), y\rangle \in \mathcal{R} \\
\forall \mathrm{X}_{1,2}, z.\ \langle z^{\mathrm{X}_1 + \mathrm{X}_2}, y\rangle \in \mathcal{R} & \supset & \vee_{i=1,2} \exists z'.(z = \mathtt{inj}_i(z') \ \wedge \ \langle z', y\rangle \in \mathcal{R}) \\
\forall \mathrm{X}, z.\ \langle z^{\mathsf{Ref}(\mathrm{X})}, y\rangle \in \mathcal{R} & \supset & z = y \ \vee \ \langle !z, y\rangle \in \mathcal{R}
\end{array}$$

Symmetrically

$$y^{\mathsf{Ref}(\beta)} \# x^{\alpha} \quad \equiv \quad \exists \mathcal{R}.(\langle x, y\rangle \in \mathcal{R} \ \wedge \ \Phi(\mathcal{R}))$$

where $\Phi(\mathcal{R})$ is given by the conjunction of:

$$\begin{array}{rcl}
\forall \mathrm{X}_{1,2}, z.\ \langle z^{\mathrm{X}_1 \times \mathrm{X}_2}, y\rangle \in \mathcal{R} & \supset & \wedge_{i=1,2} \langle \pi_i(z), y\rangle \in \mathcal{R} \\
\forall \mathrm{X}_{1,2}, z.\ \langle z^{\mathrm{X}_1 + \mathrm{X}_2}, y\rangle \in \mathcal{R} & \supset & \wedge_{i=1,2} \exists z'.(z = \mathtt{inj}_i(z') \ \wedge \ \langle z', y\rangle \in \mathcal{R}) \\
\forall \mathrm{X}, z.\ \langle z^{\mathsf{Ref}(\mathrm{X})}, y\rangle \in \mathcal{R} & \supset & z \neq y \ \wedge \ \langle !z, y\rangle \in \mathcal{R}
\end{array}$$

Remark 6.11 (content quantification and reachability) Given $e_1^\alpha \hookrightarrow e_2$ or $e_1 \# e_2$, if α is finite or recursively finite, then the axiomatisations discussed above offer a simple way to calculate content-quantified (un)reachability, such as $\langle !x\rangle e_1^\alpha \hookrightarrow e_2$: we simply decompose $\hookrightarrow$ (or $\#$) into its witness, and check how content quantification interacts with each (in)equation, using the standard axioms for content quantifiers [3]. If α contains a function type or a type variable, this method does not work. In some cases, however, we can indeed reason about such interplay easily. For example, the following logical equivalence is valid in any model under the assumption that β is recursively finite and that $\mathsf{Ref}(\beta)$ does not occur in β up to the type isomorphism.

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x. j\# x) \quad \supset \quad (\langle !x\rangle i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)} \equiv [!x] i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)} \equiv i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)}) \quad (6.1)$$

Equivalently, under the same assumption:

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x. j\# x) \quad \supset \quad (\langle !x\rangle x^{\mathsf{Ref}(\beta)} \# i^\alpha \equiv [!x] x^{\mathsf{Ref}(\beta)} \# i^\alpha \equiv i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)} \quad (6.2)$$

Note the axioms do not depend on α. Without going into a rigorous notion of models we shall discuss in the next section, the reason why (6.1) and (6.2) hold can be easily understood. By the assumption $\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x. j\# x$, any datum is unreachable to x except x itself. Since, by the type of β, we can never reach x from the content of x, this means changing the content of x cannot influence the reachability (in fact even under hypothetical content of x we can only have $i \hookrightarrow x \equiv i = x$) (indeed we believe the condition on β can be taken away). The proof of the validity of (6.1) and (6.2) can be found in Appendix.

A usable axiomatisation of $\hookrightarrow$ for finite recursive types needs coinduction. For this purpose we assume a finite relation (whose variables we write $\mathcal{R},\ldots$) and a set membership predicate ($(x,y) \in \mathcal{R}$) as part of the logic, with the corresponding axioms (we do not need such axioms as well-ordering and construction of powerset). Then we set:

$$x^\alpha \hookrightarrow' y^{\mathsf{Ref}(\beta)} \quad \equiv \quad \exists \mathcal{R}.(\langle x,y\rangle \in \mathcal{R} \wedge \Psi(\mathcal{R}))$$

where $\Psi(\mathcal{R})$ is given by the conjunction of:

$$\begin{aligned}
&\langle z^{\mathrm{X}_1\times\mathrm{X}_2}, y\rangle \in \mathcal{R} \supset \vee_{i=1,2}\langle \pi_i(z), y\rangle \in \mathcal{R}\\
&\langle z^{\mathrm{X}_1+\mathrm{X}_2}, y\rangle \in \mathcal{R} \supset \exists z'.(z = \mathtt{inl}(z') \wedge \langle z', y\rangle \in \mathcal{R}) \vee\\
&\qquad\qquad\qquad\qquad \exists z'.(z = \mathtt{inr}(z') \wedge \langle z', y\rangle \in \mathcal{R})\\
&\langle z^{\mathsf{Ref}(\mathrm{X})}, y\rangle \in \mathcal{R} \supset z = y \vee \langle !z, y\rangle \in \mathcal{R}
\end{aligned}$$

where we omit the outermost (type and variable) universal quantification over $\mathrm{X}_{1,2}$ and z.

Proposition 6.12 *If α is recursively finite, then*

$$x^\alpha \hookrightarrow' y^{\mathsf{Ref}(\beta)} \quad \equiv \quad \exists i.\mathsf{reach}(x^\alpha, y^{\mathsf{Ref}(\beta)}, i)$$

is valid.

PROOF: $\mathsf{reach}(x^{\alpha},\ y^{\mathsf{Ref}(\beta)},\ i) \supset x^{\alpha} \hookrightarrow' y^{\mathsf{Ref}(\beta)}$ is by induction on i. For the converse, simply go through the chain from x to y inside a(ny) witnessing $\mathcal{R}$ for $\hookrightarrow'$. □

Thus we obtain, by Propositions 6.9 and 6.12:

Corollary 6.13 *If α is recursively finite, then*

$$x^{\alpha} \hookrightarrow y^{\mathsf{Ref}(\beta)} \quad \equiv \quad x^{\alpha} \hookrightarrow' y^{\mathsf{Ref}(\beta)}$$

is valid, i.e. $\hookrightarrow$ is completely characterised by the axioms for $\hookrightarrow'$ given above.

Theorem 6.14 *Let C be such that each reachability predicate occurring in C is recursively finite. Then there exists C' such that $C' \equiv C$ such that C' does not contain any occurrences of reachability predicates.*

6.3 Consistency of Logic

Before establishing soundness of the logic, it is necessary to check the consistency of the logical constructs in the sense that equality, connectives and quantifications satisfy the standard axioms. For logical connectives, this is direct from the definition. For equality and quantification, however, this is not immediate, due to the non-standard definition of their semantics.

First we check the equality indeed satisfies the standard axioms for equality. We start from the following lemmas.

Lemma 6.15 *Let $\mathcal{M}$ has a type $\Gamma;\Delta;\mathcal{D}$ below.*

1. *(injective renaming) Let $u,v \in \mathsf{dom}(\Gamma)$. Then $\mathcal{M} \models C$ iff $\mathcal{M}[uv/vu] \models C[uv/vu]$.*
2. *(permutation) Let $u,v \in \mathsf{dom}(\Gamma)$. Then we have $\mathcal{M} \models C$ iff $\binom{uv}{vu}\mathcal{M} \models C[uv/vu]$.*
3. *(exchange) Let $u,v \notin \mathsf{fv}(e,e')$. Then we have $\mathcal{M}[u\!:\!e][v\!:\!e'] \models C$ iff $\mathcal{M}[v\!:\!e'][u\!:\!e] \models C$.*
4. *(monotonicity) $\mathcal{M} \models C$ implies $\mathcal{M}[u\!:\!e] \models C$. Further if $u \notin \mathsf{fv}(C)$ then $\mathcal{M}[u\!:\!e] \models C$ implies $\mathcal{M} \models C$.*
5. *(symmetry) $\mathcal{M} \models e_1 = e_2$ iff for fresh and distinct u,v: $\mathcal{M}[u\!:\!e_1][v\!:\!e_2] \approx \mathcal{M}[u\!:\!e_2][v\!:\!e_1]$.*
6. *(substitution, 1) $\mathcal{M}[u\!:\!x][v\!:\!e] \approx \mathcal{M}[u\!:\!x][v\!:\!e[u/x]]$.*
7. *(substitution, 2) $\mathcal{M}[u\!:\!e][v\!:\!e'] \approx \mathcal{M}[u\!:\!e][v\!:\!e'[e/u]]$.*

PROOF: For (1), both directions are simultaneously established by induction on C, proving for both C and its negation. If C is $e_1 = e_2$, we have, letting $\mathcal{M} \stackrel{\text{def}}{=} (\nu\tilde{y})(\xi,\sigma)$, $\delta \stackrel{\text{def}}{=} [uv/vu]$ and $\xi' \stackrel{\text{def}}{=} \xi\delta$:

$$\begin{array}{l}
\mathcal{M} \models e_1 = e_2 \\
\ \Rightarrow \mathcal{M}[x:e_1] \approx \mathcal{M}[x:e_2] \\
\ \Rightarrow (\nu\tilde{y})(\xi\cdot x:[\![e_1]\!]_{(\xi,\sigma)},\sigma) \cong_{\mathsf{id}} (\nu\tilde{y})(\xi\cdot x:[\![e_2]\!]_{(\xi,\sigma)},\sigma) \\
\ \Rightarrow (\nu\tilde{y})(\xi'\cdot x:[\![e_1\delta]\!]_{(\xi',\sigma)},\sigma) \cong_{\mathsf{id}} (\nu\tilde{y})(\xi'\cdot x:[\![e_2\delta]\!]_{(\xi',\sigma)},\sigma)\ (*) \\
\ \Rightarrow \mathcal{M}\delta[x:e_1\delta] \approx \mathcal{M}\rho[x:e_2\delta] \\
\ \Rightarrow \mathcal{M}\delta \models (e_1 = e_2)\delta
\end{array}$$

Above $(*)$ used $[\![e_i]\!]_{(\xi,\sigma)} \overset{\text{def}}{=} [\![e_i\delta]\!]_{(\xi',\sigma)}$. Dually for its negation. The rest is easy by induction. (2) is by precisely the same reasoning. (3) is immediate from (1) and (2). (4) is similar, for which we again show a base case.

$$\begin{array}{lr}
\mathcal{M}' \models e_1 = e_2 & \\
\Leftrightarrow \mathcal{M}[x:e_1] \approx \mathcal{M}[x:e_2] & \text{(Def)} \\
\Leftrightarrow \mathcal{M}[x:e_1][u:e] \approx \mathcal{M}[x:e_2][u:e] & \text{(Lem.4.15)} \\
\Leftrightarrow \mathcal{M}[u:e][x:e_1] \approx \mathcal{M}[u:e][x:e_2] & \text{((3) above)}
\end{array}$$

Dually for the negation. For (5), the "only if" direction:

$$\begin{array}{llr}
\mathcal{M} \models e_1 = e_2 & & \\
\Leftrightarrow & \mathcal{M}[u:e_1] \approx \mathcal{M}[u:e_2] & \text{(Def)} \\
\Leftrightarrow & \mathcal{M}[u:e_1][v:e_2] \approx \mathcal{M}[u:e_2][v:e_2] \wedge & \\
 & \mathcal{M}[u:e_2][v:e_2] \approx \mathcal{M}[u:e_2][v:e_1] & \text{(Lem.4.15; (3) above)} \\
\Rightarrow & \mathcal{M}[u:e_1][v:e_2] \approx \mathcal{M}[u:e_2][v:e_1]. &
\end{array}$$

Operationally, the encoding of models simply removes all references to u, v and replaces them by positional information: hence all relevant difference is induced, if ever, by behavioural differences between e_1 and e_2, which however cannot exist by assumption. The "if" direction is immediate by projection.

(6) and (7) are best argued using concrete models. For (6), Let $\mathcal{M} = (\nu\tilde{y})(\xi,\sigma)$ and let $\xi(x) = W$. We infer:

$$\begin{aligned}
\mathcal{M}[u{:}x][v{:}e] &\overset{\text{def}}{=} (\nu\tilde{y})(\xi\cdot u:W\cdot v:e\xi,\ \sigma) \\
&\overset{\text{def}}{=} (\nu\tilde{y})(\xi\cdot u:W\cdot v:(e[u/x])\xi,\ \sigma)
\end{aligned}$$

For (7), let $\mathcal{M} = (\nu\tilde{y})(\xi,\sigma)$ and $W = [\![e]\!]_{\xi,\sigma}$ (the standard interpretation of e by ξ and σ). We then have

$$\begin{aligned}
\mathcal{M}[u{:}e][v{:}e'] &\approx (\nu\tilde{y})(\xi\cdot u:W\cdot v:[\![e']\!]_{\xi,\sigma},\ \sigma) \\
&\overset{\text{def}}{=} (\nu\tilde{y})(\xi\cdot u:W\cdot v:[\![e'[e/u]]\!]_{\xi,\sigma},\ \sigma)
\end{aligned}$$

The last line is because the interpretation is homomorphic. □

We are now ready to establish the standard axioms for equality.

Lemma 6.16 (axioms for equality) *For any model $\mathcal{M}$ and x, y, z and C:*

1. $\mathcal{M} \models x = x$, $\mathcal{M} \models x = y \supset y = x$ *and* $\mathcal{M} \models (x = y \wedge y = z) \supset x = z$.
2. $\mathcal{M} \models (C(x,y) \wedge x = y) \supset C(x,x)$.

where $C(x,y)$ and $C(x,x)$ is understood as in [23, §2.4] (i.e. $C(x,y)$ indicates C together with some of the occurrences of x and y, while $C(x,x)$ is the result of substituting x for the latter).

PROOF: For the first clause, reflexivity is because $\mathcal{M}[u\!:\!x] \approx \mathcal{M}[u\!:\!x]$, while symmetry and transitivity are from those of $\approx$. For the second clause, we proceed by induction on C. We show the case where C is $e_1 = e_2$. It suffices to prove $\mathcal{M} \models x = y$ and $\mathcal{M} \models C$ imply $\mathcal{M} \models C[x/y]$.

$$\begin{aligned} \mathcal{M} \models x = y &\Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y] \approx \mathcal{M}[u\!:\!y][v\!:\!x] && (6.3)\\ &\Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_i] \approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i] && (6.4) \end{aligned}$$

Here (6.3) is by Lemma 6.15.5 and (6.4) follows from Lemma 4.15.

$$\begin{aligned} \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_i] &\approx \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_i[v/y]] && \text{(Lem. 6.15.6)}\\ &\approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i[v/y]] && \text{(Lem. 4.15, 6.3)}\\ &\approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i[vv/xy]] && \text{(Lem. 6.15.6)}\\ &\approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i[xx/xy]] && \text{(Lem. 6.15.7)}\\ &\approx \mathcal{M}[w\!:\!e_i[xx/xy]][u\!:\!y][v\!:\!x] && \text{(Lem. 6.15.3)} \end{aligned}$$

$$\begin{aligned} \mathcal{M} \models e_1 = e_2 &\Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y] \models e_1 = e_2 && \text{(Lem. 6.15.4)}\\ &\Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_1] \approx \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_2] \end{aligned}$$

Thus we get

$$\begin{aligned} \mathcal{M}[w\!:\!e_1[xx/xy]][u\!:\!y][v\!:\!x] &\approx \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_1]\\ &\approx \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_2]\\ &\approx \mathcal{M}[w\!:\!e_2[xx/xy]][u\!:\!y][v\!:\!x] \end{aligned}$$

This allows to conclude to:

$$\mathcal{M}[w\!:\!e_1[xx/xy]] \approx \mathcal{M}[w\!:\!e_2[xx/xy]]$$

which is equivalent to:

$$\mathcal{M} \models C(x,x).$$

as required. □

Lemma 6.17 *(axioms for $\forall$) For any model $\mathcal{M}$:*

1. *$\mathcal{M} \models (\forall x^{\alpha}.C) \supset C[e/x]$ for all e of type α.*
2. *$\mathcal{M} \models (\forall x^{\alpha}.(C_1 \supset C_2)) \supset (C_1 \supset \forall x^{\alpha}.C_2)$, provided x in C_1.*

PROOF: We only show (1) when α is a value type.

$$\mathcal{M} \models \forall x^{\alpha}.C \Leftrightarrow \forall N.\mathcal{M}[x\!:\!N] \models C \quad \Rightarrow \forall e.\mathcal{M}[x\!:\!e] \models C$$

(2) is standard. □

Lemma 6.18 *(content quantification) For any $\mathcal{M}$.*

1. $\mathcal{M} \models [!x]C \supset C$,
2. $\mathcal{M} \models [!x](!x = m \supset C) \equiv \langle !x \rangle (!x = m \wedge C)$,
3. $\mathcal{M} \models ([!x](C_1 \supset C_2)) \supset C_1 \supset [!x]C_2$ *when* $[!x]C \equiv C_1$.

PROOF: For (1)

$$\mathcal{M} \models [!x]C \text{ implies } \mathcal{M}[x \mapsto !x] \models C \text{ implies } \mathcal{M} \models C.$$

For (2)

$$\begin{aligned}\mathcal{M} \models [!x](!x = m \supset C) &\Leftrightarrow \mathcal{M}[x \mapsto m] \models C \\ &\Leftrightarrow \mathcal{M}[x \mapsto m] \models C \wedge !x = m \\ &\Leftrightarrow \mathcal{M} \models \langle !x \rangle (C \wedge !x = m)\end{aligned}$$

Finally, for (3), if $[!x]C_1 \equiv C_1$ then: $\mathcal{M} \models C$ iff for all V, $\mathcal{M}[e \mapsto V] \models C_1$. Consequently:

$$\begin{aligned}&\mathcal{M} \models [!x](\neg C_1 \vee C_2) \\ &\equiv \forall V.\ \mathcal{M}[e \mapsto V] \not\models C_1 \text{ or } \mathcal{M}[e \mapsto V] \models C_2) \\ &\equiv \mathcal{M} \not\models C_1 \text{ or } \forall V.\ \mathcal{M}[e \mapsto V] \models C_2 \\ &\equiv \mathcal{M} \models \neg C_1 \ \vee [!x]C_2\end{aligned}$$

as required. □

Lemma 6.19 $\mathcal{M} \models \langle !x \rangle (C \wedge !x = m)$ *iff* $\mathcal{M}[x \mapsto [\![e]\!]_{\xi,\sigma}] \models C$

PROOF: Straightforward and standard. Below let us set $\mathcal{M}$ to be the model $(\nu \tilde{l})(\xi \cdot x : l \cdot m : V, \sigma \cdot [l \mapsto W])$ (the case $x \notin \mathsf{fv}(C)$ is obvious).

$$\begin{aligned}&\mathcal{M} \models \langle !x \rangle (C \wedge !x = m) \\ &\Leftrightarrow \exists \mathcal{M}' = \mathcal{M}[x \mapsto V] \text{ s.t. } (\mathcal{M}' \models C \wedge \mathcal{M}' \models !x = m) \\ &\Leftrightarrow (\nu \tilde{l})(\xi \cdot x : l \cdot m : V, \sigma \cdot [l \mapsto V]) \\ &\Leftrightarrow \mathcal{M}[x \mapsto m] \models C\end{aligned}$$

as required (note the reasoning is identical with the corresponding proof in [3], or with the corresponding one in Hoare logic). □

Notation 6.20 We write $C\{\!|e'/!e|\!\}$ for $\exists m.(\langle !x \rangle (C \wedge !e = m) \wedge m = e')$ with m fresh.

Lemma 6.21 *(plain free reference names) Let $u \notin \mathtt{fpn}(e)$. Then, with u fresh, for all M:*

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

PROOF: $\mathcal{M}'$ has shape:

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

with $(\nu \tilde{l}_0)(M\xi, \sigma_0) \Downarrow (\nu \tilde{l}_0)(V, \sigma)$. Then one can check $[\![e]\!]_{\xi \cdot u : l, \sigma \cdot [l \mapsto V]} = [\![e]\!]_{\xi,\sigma} \notin cl(l, \sigma \cdot [l \mapsto V]) = cl(l, [l \mapsto V])$. □

Lemma 6.22 *Suppose A is stateless and $\mathcal{M} \models A$. Then:*

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

PROOF: By induction on A. □

6.4 Soundness of Proof Rules

we are ready to prove:

Theorem 6.23 $\vdash \{C\}\, M :_u \{C'\}$ *implies* $\models \{C\}\, M :_u \{C'\}$.

PROOF: We start with [*Var*].

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

Similarly [*Const*] is reasoned:

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

Next, [*Inj*$_1$] is reasoned:

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

For [*Proj*] we reason as follows.

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

For [*Case*], we reason:

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

Now we reason for [*Abs*]. We assume $x, \tilde{i}$ have functional types.

$$\begin{array}{l} \quad \mathcal{M} \models A \supset \mathcal{M}[u\,{:}\,\lambda x.M] \models \forall x\tilde{i}.\{C\}u \bullet x = m\{C'\} \\ \equiv \mathcal{M} \models A \supset \mathcal{M}[u\,{:}\,\lambda x.M][x\,{:}\,V][\tilde{i}\,{:}\,\tilde{W}] \models \{C\}u \bullet x = m\{C'\} \\ \equiv \mathcal{M} \models A \supset (\mathcal{M}[u\,{:}\,\lambda x.M][x\,{:}\,V][\tilde{i}\,{:}\,\tilde{W}][k\,{:}\,N] \Downarrow \mathcal{M}_0 \wedge \mathcal{M}_0 \models C) \\ \qquad\qquad\qquad\qquad\qquad\qquad \supset (\mathcal{M}_0[m\,{:}\,ux] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') \\ \equiv (\mathcal{M} \models A \wedge \mathcal{M}[u\,{:}\,\lambda x.M][x\,{:}\,V][\tilde{i}\,{:}\,\tilde{W}][k\,{:}\,N] \Downarrow \mathcal{M}_0 \wedge \mathcal{M}_0 \models C) \\ \qquad\qquad\qquad\qquad\qquad\qquad \supset (\mathcal{M}_0[m\,{:}\,ux] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') \\ \equiv (\mathcal{M} \models A \wedge \mathcal{M}[u\,{:}\,\lambda x.M][x\,{:}\,V][\tilde{i}\,{:}\,\tilde{W}][k\,{:}\,N] \Downarrow \mathcal{M}_0 \wedge \mathcal{M}_0 \models A \wedge C) \quad \text{(Lemma 6.22 (1))} \\ \qquad\qquad\qquad\qquad\qquad\qquad \supset (\mathcal{M}_0[m\,{:}\,ux] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') \\ \subset \mathcal{M}_0 \models A \wedge C \supset (\mathcal{M}_0[m\,{:}\,M] \Downarrow \mathcal{M}_0' \wedge \mathcal{M}_0' \models C') \end{array}$$

If x has a reference type, we use Lemma 6.22 (2) instead of (1). Then reasoning is identical.

[*App*] is reasoned as follows.

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

For [*Deref*], we infer:

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

For [*Assign*] we reason as follows, assuming u to be fresh.

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

For [*Rec*], we establish the result for a variant:

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

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

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

[*Rec*] is easily derivable with [*Rec-Ren*] using mathematical induction at the level of assertions. Proving the converse derivability (or rather equi-potence) needs a different method.

For [*Ref*], we reason:

$$\begin{array}{rcl}\mathcal{M} \models C & \Rightarrow & \mathcal{M}[m\!:\!M] \Downarrow \mathcal{M}' \models C' \\ & \Rightarrow & \mathcal{M}[m\!:\!M][u\!:\!\mathtt{ref}(M)] \Downarrow \mathcal{M}_0 \approx \mathcal{M}'[u \mapsto m] \\ & \text{and} & \mathcal{M}'[u \mapsto m] \models C' \wedge !u = m \wedge u \# i \quad \text{(Lemma 6.21)} \\ & \Rightarrow & \mathcal{M}[u\!:\!\mathtt{ref}(M)] \Downarrow \mathcal{M}''/m \models \# u.C[!u/m]\end{array}$$

□

Remark 6.24 On the soundness of other proof rules from [3, 19], we observe:

– All proof rules listed in the previous logic [3, 19] are sound in the present logic except for a minor adjustment in [*Aux*-$\forall$] (which introduces $\forall i$ to the postcondition if an auxiliary name i does not occur in the pre-condition). Among others this allows us to have all the derived rules for modular verification in [3] (one of them is the invariance rule which is already treated above).
– As we have seen in Section 5, the lack of validity of [*Aux*-$\forall$] in [3, 19] in its general form stems from the existence of new name generation rather than a particular choice of assertion language in the present context. For this reason we believe this is inevitable in any logic which allows description of dynamic new name generation as treated in the present work. The rule is sound if we restrict the type of an auxiliary name to be quantified so that it does not include a reference type.

6.5 Observational Completeness (1): FCFs

Basic Ideas. In this subsection and next, we establish observational completeness, which says the logic differentiates programs precisely as the standard observational completeness does. The employed arguments are close to what have been given in [3, 19], taking the following steps.

1. We first reduce the contexts needed to witness semantic difference to finite canonical forms. In particular, we reduce the "new" construct to generators of relative fresh names. The resulting restricted programs are called *FCFs*. Following [3], we extend the logical language with so-called *vector variables*.
2. Next we derive a characteristic formula for each FCF, by induction of the structure of FCFs. In particular, the preconditions of characteristic formulae we derive are always *open*, in the sense that whenever $(\nu\tilde{x})(\xi,\sigma) \models C$ then $(\xi,\sigma) \models C$ holds (this property does not hold for general formulae).
3. Finally, using 1 and 2, we show any differentiating context between M_1 and M_2 can be reduced to a formula which can only be satisfied by one of $M_{1,2}$ but not the other, reaching observational completeness.

Notice if two programs are observationally equivalent then surely they satisfy the same set of pre/post-conditions, because satisfiability is closed under $\cong$.

Context Reduction. For establishing this result, we reduce arbitrary differentiating contexts to programs of specific shape, which we call FCFs. The grammar of FCFs are from [3, §8.4]. Formally, *finite canonical forms*, or *FCFs* for short, ranged over by $F, F', \ldots$, are a subset of typable programs given by the following grammar (which are read as programs in imperative PCFv in the obvious way). $U, U', \ldots$ range over FCFs which are values.

$$\begin{array}{rcl} F & ::= & \mathtt{n} \mid x^{\mathsf{Ref}(\alpha)} \mid \omega^{\alpha} \mid \lambda x.F \mid \mathtt{let}\, x = yU\, \mathtt{in}\, F \\ & \mid & \mathtt{case}\, x\, \mathtt{of}\, \langle \mathtt{n_i} : F_i \rangle_{i\in X} \mid \mathtt{case}\, x\, \mathtt{of}\, \langle \mathtt{y_i} : F_i \rangle_{i\in X} \\ & \mid & \mathtt{let}\, x = !y\, \mathtt{in}\, F \mid x := U; F \end{array}$$

where:

– In the second case construct, x and y_i should be typed by the same reference type.
– In each of the case constructs, X should be a finite non-empty subset of natural numbers (it diverges for other values); and
– ω^α stands for a diverging closed term of type α (e.g. $\omega^\alpha \stackrel{\text{def}}{=} (\mu x^{\alpha\Rightarrow\alpha}.\lambda y.xy)V$ with V any closed value typed α).

Note no "new" construct is used. Reference names are treated in the case construct. We omit the obvious translation to imperative PCFv-terms and typing rules.

We now outline the proof of the following result.

Lemma 6.25 *Let V_1 and V_2 be closed and $V_1 \not\cong V_2$. Then there exists an FCF U in the above sense s.t. $UV_i \Downarrow$ and $UV_j \Uparrow$ with $i \neq j$.*

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

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

which means, through the β_V-equality:

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

where we set $W \stackrel{\text{def}}{=} \lambda x.C[x]$. Note the convergence in $(WM_1,\ \tilde{r} \mapsto \tilde{V}) \Downarrow$ takes, by the very definition, only a finite number of reductions. Let it be n. Then (occurrences of) λ-abstractions in W and $\tilde{V}$ can only be applied up to n-times, similarly for other destructors. Also all occurrences of "new" construct can only be used at most n-times. So, taking some fresh n names, we can replace them with generators of a sequence of these relatively fresh n names for each type, each set up as a procedure, called at each place where "new" is used, using a finite list and a counter as a way to counting it.

Using these decompositions, we transform these programs into FCF values maintaining the convergence property while being made less defined (we leave the details to [19, Appendix A]). Once this is done, we obtain (semi-closed) FCF values, which we set to be F and $\tilde{U}$. The transformation maintains convergence behaviour of $(WM_1, \tilde{r} \mapsto \tilde{V})$. Further $(FM_2, \tilde{r} \mapsto \tilde{U})$ is more prone to divergence than $(WM_2, \tilde{r} \mapsto \tilde{V})$, so it still diverges. Thus we obtain

$$(FM_1,\ \tilde{r} \mapsto \tilde{U}) \Downarrow \qquad \text{and} \qquad (FM_2,\ \tilde{r} \mapsto \tilde{U}) \Uparrow .$$

For detailed illustration of context reduction, see [19, §6 and Appendix A]. □

For brevity we only consider programs typable by the set of types inductively generated from Nat, arrow types and reference types. Accordingly, we assume the "if" construct branches on numerals, judging if it is (say) zero or non-zero, and the syntax of the assignment has the form $M_1 := M_2\,; N$, with the obvious operational semantics. The technical development is easily extendible to other constructs.

Vector Variables. Another preparation needed for observational completeness is a small extension of the logical language. The added construct is not generally used for assertions, and may not be necessary for observational completeness *per se*, but we do use it in our present proof. The construct can be used when a program uses a behaviour with generic (unknown) side effects in the environment: however, when we use external programs, an assertion may as well constrain side effects of external behaviours in some way, so its practical use would be limited. The extension is given at the level of logical terms, as follows:

$$e \quad ::= \quad \dots \quad | \quad \mathfrak{a} \quad | \quad !\mathfrak{a}$$

$\mathfrak{a}$ is called *vector variable*, and represents a vector of values. For our present purpose, we only need to allow constructors on vector variables except for dereferences, as given above. Vector variables and their dereferences are only used for equations and quantifications, though other constructions are possible (for example injection of a vector variable makes sense).

For typing vector variables, we need to introduce vector types, which are used only for typing vector variables.

$$\vec{\alpha} ::= \vec{X} \mid \mathsf{Ref}(\vec{X})$$

$\vec{X}$ denotes a vector of generic types (which can be distinct from each other). We can consider other sorts of vector types (for example a vector of standard types is surely useful), but this is all we need in the present context.

There are several natural predicates usable for vector variables and values. Among them is a membership relation, written $x \in \mathfrak{a}$, which says x is one of the values constituting the vector $\mathfrak{a}$. We write this operation $x \in \mathfrak{a}$. We then define, for a $\mathsf{Ref}(\vec{X})$-typed $\mathfrak{a}$:

$$Max(\mathfrak{a}^{\mathsf{Ref}(\vec{X})}) \quad \stackrel{\text{def}}{=} \quad \forall Y.\, \forall x^{Y}, \mathfrak{b}^{\mathsf{Ref}(\vec{X})}.(x \in \mathfrak{b} \supset x \in \mathfrak{a})$$

This makes sense since reference names in a model is always finite.

The interpretation of a vector variable and its dereference is given by extending a model to interpret a vector variable as a sequence of values. As data, we add, in (ξ, σ):

– The environment map, ξ, now also maps vector variables to their values: each vector variable is mapped to a vector of values of the corresponding types. If $\mathfrak{a}$ is of type $\mathsf{Ref}(\vec{Y})$, then it is mapped to a vector of identicals in $\mathsf{dom}(\sigma)$.
– The store map, σ, does not change, still mapping identicals to stored values.

A vector variable of a vector type is then interpreted simply as a vector of values mapped in the model. We interpret terms as follows:

$$\begin{array}{rll} [\![\mathfrak{a}^{\vec{X}}]\!]_{(\xi,\sigma)} \stackrel{\text{def}}{=} \xi(\mathfrak{a}^{\vec{X}}) & & \\ [\![!\mathfrak{a}^{\mathsf{Ref}(\vec{X})}]\!]_{(\xi,\sigma)} \stackrel{\text{def}}{=} !\mathbf{i}_1 \dots !\mathbf{i}_n & \qquad & ([\![\mathfrak{a}]\!]_{(\xi,\sigma)} = \mathbf{i}_1 \dots \mathbf{i}_n) \end{array}$$

6.6 Observational Completeness (2): Characteristic Formulae

TCA and Characteristic Assertions Pairs. We first make the class of formulae we shall use for defining characteristic formulae. Below $\sqsubseteq$, the *contextual preorder*, is the standard pre-order counterpart of $\cong$.

Definition 6.26 (TCAs) An assertion C is a *total correctness assertion (TCA) at u* if whenever $(\xi \cdot u : \kappa, \sigma) \models C$ and $\kappa \sqsubseteq \kappa'$, we have $(\xi \cdot u : \kappa', \sigma) \models C$. Similarly C is a *total correctness assertion (TCA) at* $!x$ if whenever $(\xi, \sigma \cdot x \mapsto \kappa) \models C$ and $\kappa \sqsubseteq \kappa'$, we have $(\xi, \sigma \cdot x \mapsto \kappa') \models C$.

Intuitively, total correctness is a property which is closed upwards — if a program M satisfies it and there is a more defined program N then N also satisfies it (there are assertions which describe partial correctness rather than total correctness. For example,

$$\forall x.(u \bullet x \searrow x! \ \vee \ u \bullet x \Uparrow)$$

is a partial correctness assertion for a factorial). Practically all natural total correctness specification (which does not mention, essentially, non-termination) would be straightforwardly describable as a TCA. The present logic, including its proof system, is geared towards total correctness: from this viewpoint, we may as well restrict our attention to total correctness assertions. Below we introduce three notions which are about characteristic formulae for total correctness.

Convention 6.27 (TCA pair) We say a pair (C, C') is a *TCA-pair for* $\Gamma; \Delta \vdash M : \alpha$ *at u*, or simply a *TCA-pair* with the concerned typed program implicit, when: (1) C is a TCA at $\mathsf{dom}(\Gamma; \Delta)$ and (2) C' is a TCA at $\{u\} \cup \mathsf{dom}(\Delta)$ and a co-TCA at $\{\tilde{\Gamma}\}$ (a *co-TCA* is given by the same clauses as in Definition 6.26 except changing $\sqsubseteq$ with $\sqsupseteq$).

Definition 6.28 (characteristic assertion pair) We say a TCA pair (C, C') is *a characteristic assertion pair (CAP) for* $\Gamma; \Delta \vdash M : \alpha$ *at u* iff we have: (1) $\models \{C\} M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ and (2) $\models \{C\} N^{\Gamma;\Delta;\alpha} :_u \{C'\}$ implies $\Gamma; \Delta \vdash M \sqsubseteq N : \alpha$. We also say (C, C') *characterise* $\Gamma; \Delta \vdash M : \alpha$ *at u* when (C, C') is a CAP for $\Gamma; \Delta \vdash M : \alpha$ at u.

Definition 6.29 (minimal terminating condition) Let $\Gamma \vdash M : \alpha$. Then we say C is an *minimal terminating condition*, or an *MTC*, for $\Gamma \vdash M : \alpha$ iff the following condition holds: $(\xi, \sigma) \models C$ if and only $(M\xi, \sigma) \Downarrow$; and (2) $(M\xi, \sigma) \Downarrow$ implies, if $(\xi, \sigma) \models \exists \tilde{i}.C$ where $\tilde{i}$ are auxiliary names in C (i.e. $\mathsf{fv}(C) \backslash \mathsf{dom}(\Gamma)$).

An MTC is a condition by which termination or divergence of a program is determined. In the purely functional sublanguage, this is solely about the class of closing substitutions, while in imperative PCFv, the notion also includes paired stores. We can now introduce a strengthened version of CAP.

Definition 6.30 (strong CAP) Let $\Gamma; \Delta \vdash M : \alpha$. Then we say a TCA pair (C, C') is a *strong characteristic assertion pair, or strong CAP for* $\Gamma; \Delta \vdash M : \alpha$ *at u*, iff we have:

1. (soundness) $\models \{C\} M^{\Gamma;\Delta;\alpha} :_u \{C'\}$.
2. (MTC) C is an MTC for M.
3. (closure) If $\models \{C \wedge E\}\ N :_u \{B\}$ and $\mathcal{M} \models C \wedge E$, then $\mathcal{M}[u : M] \sqsubseteq \mathcal{M}[u : N]$ (with the latter preorder defined in the obvious way).

Proposition 6.31 *If (C, C') is a strong CAP of M, then (C, C') is a CAP of M.*

PROOF: If (C, C') is a strong CAP for M, then, by definition, for any ξ and σ, we have $(M\xi, \sigma) \sqsubseteq (N\xi, \sigma)$. Since $M \sqsubseteq N$ iff $\forall \xi, \sigma.((M\xi, \sigma) \sqsubseteq (N\xi, \sigma))$ we are done. $\square$

A strong CAP says that a pair (C, C') defines a program in a way stronger than a CAP in one point: it demands, in addition to being a CAP, that giving a more focus/restriction on the precondition (an initial environment and store) leads to a more focus/restriction on the postcondition (the resulting value and state). Because of this closure property, the use of strong CAP, instead of CAP, is fundamental for the subsequent technical development.

Fig. 9 Proof rules for characteristic assertions of FCFs.

$$\frac{-}{\{\mathsf{T}\}\ \mathtt{n}^{\Gamma;\mathsf{Nat}} :_u \{u = \mathbf{n}\}\ @\ \emptyset} \qquad \frac{\{C_i\}\ F_i^{\Gamma\cdot x:\mathsf{Nat};\alpha} :_u \{C_i'\}}{\{\vee_i(x = \mathbf{n}_i \wedge C_i)\}\ \mathtt{case}\, x\, \mathtt{of}\ \langle \mathtt{n_i} : F_i \rangle_i^{\Gamma;\alpha} :_u \{\vee_i(x = \mathbf{n}_i \wedge C_i')\}}$$

$$\frac{-}{\{\mathsf{T}\}\ x^{\Gamma;\mathsf{Ref}(\alpha)} :_u \{u = x\}\ @\ \emptyset} \qquad \frac{\{C_i\}\ F_i^{\Gamma\cdot x:\mathsf{Ref}(\beta);\alpha} :_u \{C_i'\}}{\{\vee_i(x = y_i \wedge C_i)\}\ \mathtt{case}\, x\, \mathtt{of}\ \langle \mathtt{y_i} : F_i \rangle_i^{\Gamma;\alpha} :_u \{\vee_i(x = y_i \wedge C_i')\}}$$

$$\frac{\{C\}\ F^{\Gamma,x:\alpha;\beta} :_m \{C'\}}{\{\mathsf{T}\}\ \lambda x.F^{\Gamma;\alpha\Rightarrow\beta} :_u \{\forall x.\{C\} u \bullet x \searrow m\{C'\}\}\ @\ \emptyset}$$

$$\frac{\{\mathsf{T}\}\ U^{\Gamma;\alpha} :_z \{A\}\ @\ \emptyset \quad \{C\} F^{\Gamma\cdot x:\beta;\gamma} :_u \{C'\}}{\{\forall \mathfrak{a}, \mathfrak{b}.\ ((!\mathfrak{a} = \mathfrak{b} \wedge Max(\mathfrak{a})) \supset \forall z.\{A \wedge !\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\})\}\ \mathtt{let}\ x = yU\ \mathtt{in}\ F^{\Gamma;\gamma} :_u \{C'\}}$$

$$\frac{\{C\}\ F :_u \{C'\}}{\{C[!x/y]\}\ \mathtt{let}\ y = !x\ \mathtt{in}\ F :_u \{C'\}}$$

$$\frac{\{\mathsf{T}\}\ U^{\Delta;\alpha} :_z \{A\} \quad \{C\}\ F :_u \{C'\} \quad \mathsf{fv}(A) \subset \{z\} \cup \mathsf{dom}(\Delta)}{\{\forall z.(A \supset C\{\!|z/!x|\!\})\}\ x := U\, ;\, F :_u \{C'\}}$$

$$\frac{-}{\{\mathsf{F}\}\ \omega^{\Gamma;\alpha} :_u \{\mathsf{F}\}}$$

From FCFs to Characteristic Formulae. In Figure 9, we present the proof rules for deriving strong CAPs for FCFs. To be explicit with involved typing, we annotate each program with its typing of the form $M^{\Theta;\alpha}$ where Θ is the union of the environment basis and the reference basis. As in [3, 19], the rules use equality instead of termination preorder for legibility. We observe:

1. In the rules for values (reference names, numerals and abstraction), we use located judgements for precise description of its behaviour, which are regarded as their translations into non-located judgements (we assume fresh names are chosen at each rule for implicit reference names used in located judgements).

2. In the rule for let-application (the sixth rule), by assuming $\mathfrak{a}$ and $\mathfrak{b}$ being fresh and typed by a generic reference vector type (say $\mathsf{Ref}(\mathrm{X})$) and the corresponding generic vector type (say X), they can stand for arbitrary reference name and its content, which is essential for stipulating the (assumed) property of f, even in the setting of local state..
3. In the rule for assignment, we use the logical substitution rather than the syntactic one, to deal with arbitrary aliasing.

We write:

$$\vdash_{\mathsf{char}} \{C\}F :_u \{C'\}$$

if $\{C\}\, F :_u \{C'\}$ is provable from these proof rules. In each rule, we assume each premise is derived in this proof system, not others. We leave the illustration of these rules to [3, 19]. These formulae, especially preconditions, have very restricted shape, which plays an important role in the technical discussions.

Definition 6.32 *A formula C is* open *if whenever $(\nu\tilde{x})(\xi,\sigma) \models C$ we have $(\xi,\sigma) \models C$.*

Proposition 6.33 *Whenever $\vdash_{\mathsf{char}} \{C\}F :_u \{C'\}$, the formula C is open.*

PROOF: By induction of the rules in Figure 9, noting a reference is never existentially quantified (distinction in case constructs does not differ by taking of the ν-binder). □

Proposition 6.34 *If $\vdash_{\mathsf{char}} \{C\}\ F :_u \{C'\}$, then (C,C') is a strong CAP of F at u.*

PROOF: All arguments follow [3, 19]. Below we only show the case of let, which is most non-trivial.

(Let-Application) We need to say:

> *If a function denoted by the variable is such that it converges under the present store, then it converges.*

For asserting this and related situation, we use vector variables. For focussing on the central point of the argument, we stipulate:

Convention 6.35 *In the following proof, we deliberately confuse reference names and identicals for simplicity, treating only the former. This does not change the arguments since the coalescing of reference names does not play any role in the proof.*

Let U and F be typed as $U^{\Gamma;\alpha}$, $F^{\Gamma\cdot x:\beta;\gamma}$, and $\tilde{y}$ be non-reference names in the basis. We set, with $\tilde{r} = r_1..r_i..r_n$ ($n \geq 0$):

$$F' \stackrel{\text{def}}{=} \mathtt{let}\, x = fU\ \mathtt{in}\, F \quad (6.5)$$

$$\xi_0 = \tilde{y}{:}\tilde{S} \quad (6.6)$$

$$\xi_1 = \mathfrak{a}{:}\tilde{r}, \mathfrak{b}{:}\tilde{V} \quad (6.7)$$

$$\xi = \xi_0 \cdot f{:}W \quad (6.8)$$

$$\sigma = \tilde{r} \mapsto \tilde{V} \quad (6.9)$$

$$C_0 \stackrel{\text{def}}{=} \forall \mathfrak{a}, \mathfrak{b}.\ ((!\mathfrak{a} = \mathfrak{b} \wedge Max(\mathfrak{a})) \supset \forall z.\{A \wedge !\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\}) \quad (6.10)$$

such that (ξ,σ) is a model which conforms to Γ, and assume we have:

(IH1) C, C' is a strong CAP at u for F (hence in particular C is an MTC for F).
(IH2) T, A is a strong CAP at z for U.

Note, by (IH2), we have $\models \{\mathsf{T}\} U :_z \{A\}$ hence for any ξ_0 and I:

$$\xi_0 \cdot z : U\xi_0 \models A \tag{6.11}$$

Further we observe $(\xi, \sigma) \models Max(\mathfrak{a}^{\mathsf{Ref}(\vec{X})}) \quad \Rightarrow \quad \{[\![\mathfrak{a}]\!]\} = \mathsf{dom}(\sigma)$. In the present case, we can safely set $[\![\mathfrak{a}]\!] = \tilde{r}$, using Convention above. We now show C_0 is an MTC for F', starting from one direction, $(\nu\tilde{x})(\xi, \sigma) \models C_0$ implies $(F'\xi, \sigma) \Downarrow$.

$$
\begin{array}{lll}
\multicolumn{2}{l}{(\nu\tilde{x})(\xi,\ \sigma) \models C_0} & \\
\Rightarrow & (\xi,\ \sigma) \models C_0 & (C_0 \text{ open}) \\
\Rightarrow & (\xi \cdot \xi_1,\ \sigma) \models \forall z.\{A \wedge !\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\}\,) & (\forall, \text{Max}) \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \ \supset\ (z : U_1 \cdot \xi_0 \cdot \xi_1 \cdot \xi,\ \sigma) \models \{!\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\} & \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \ \supset\ (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma'_1) \models C & \\
\Rightarrow & \forall U_1 \sqsupseteq U\xi_0 \ \supset\ (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma'_1) \models C & \text{(IH2)} \\
\Rightarrow & (\xi_0 \cdot x : WU,\ \sigma) \Downarrow (\xi \cdot x : S,\ \sigma) \models C & \\
\Rightarrow & (F'\xi,\ \sigma) \Downarrow & \text{(IH1)}
\end{array}
$$

as required. Next we show $(F'\xi,\ \sigma) \Downarrow$ and $\{\tilde{x}\} \cap \mathsf{fv}(C_0) = \emptyset$ imply $(\nu\tilde{x})(\xi, \sigma) \models C_0$.

$$
\begin{array}{lll}
\multicolumn{2}{l}{(F'\xi, \sigma) \Downarrow} & \\
\Rightarrow & (\xi_0 \cdot x : WU,\ \sigma) \Downarrow (\xi \cdot x : S,\ \sigma) \models C & \text{(IH1)} \\
\Rightarrow & \forall U_1 \sqsupseteq U\xi_0 \ \supset\ (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma'_1) \models C & \text{(IH2)} \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \ \supset\ (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma'_1) \models C & \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \ \supset\ (z : U_1 \cdot \xi \cdot \xi_1,\ \sigma) \models \{!\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\} & \\
\Rightarrow & (\xi \cdot \xi_1,\ \sigma) \models \{A \wedge !\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\} & \text{(e5)} \\
\Rightarrow & \forall \xi'_1.\ (\ (\xi \cdot \xi'_1,\ \sigma) \models !\mathfrak{a} = \mathfrak{b} \wedge Max(\mathfrak{a}) \ \supset\ \{A \wedge !\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\}) & \\
\Rightarrow & (\xi,\ \sigma) \models C_0 & \\
\Rightarrow & (\nu\tilde{x})(\xi,\ \sigma) \models C_0 &
\end{array}
$$

Above, in the second to the last line, ξ' is an arbitrary interpretation of $\mathfrak{a}$ and $\mathfrak{b}$. This step is because, as far as $!\mathfrak{a} = \mathfrak{b}$ and $Max(\mathfrak{a})$ hold, (apart from how elements are permuted and duplicated) the content of $\mathfrak{a}$ and $\mathfrak{b}$ are invariant. Finally the last line is because for any $\tilde{x}$ such that $\{\tilde{x}\} \cap \mathsf{fv}(C_0) = \emptyset$, if we have $(\xi, \sigma) \models C_0$ then we have $(\nu\tilde{c})(\xi, \sigma) \models C_0$.

For the closure condition, let $\Gamma; \Delta \vdash M : \alpha$ and assume

$$\{C_0 \wedge E\} M :_u \{C'\}. \tag{6.12}$$

By Proposition 6.33, we safely assume $C_0 \wedge E$ is open. Let $\mathsf{dom}(\Delta) = \mathsf{dom}(\sigma) = \{\tilde{r}\}$ (the effect of aliasing can be ignored in the following arguments). Following [19, §6.6, p.58], we reason using the following programs. Let a vector of names $\tilde{z}$ be fresh below. We write $\mathtt{let}\ \tilde{z} = !\tilde{r}\ \mathtt{in}\ F$ for a sequence of let-derefs and $\tilde{r} := \tilde{V}$ for a sequence of assignments.

$$M_0 \quad \stackrel{\text{def}}{=} \quad \mathtt{let}\ \tilde{z} = !\tilde{r}\ \mathtt{in}\ \mathtt{let}\ x = yU\ \mathtt{in}\ (\tilde{r} := \tilde{z}\ ;\ M) \tag{6.13}$$

By checking the reduction we have: $M \cong_\Delta M_0$, so that we hereafter safely use M_0 instead of M without any affect on semantics. Now assume $(\nu\tilde{x})(\xi,\ \sigma) \models C_0\ \wedge\ E$ with $C_0 \wedge E$ open. By open-ness we have $(\xi,\ \sigma) \models C_0\ \wedge\ E$. By (6.13) we have, with $\mathsf{dom}(\sigma'_0) = \tilde{w}$ and with newly introduced entities existentially quantified:

$$\begin{aligned}(\xi{\cdot}u : M_0\xi,\ \sigma) &\longrightarrow^* (\nu\,\tilde{w})(\xi{\cdot}u : (\tilde{r} := \sigma(\tilde{r}); M)\xi,\ \sigma_0 \cdot \sigma'_0) \models C\\ &\longrightarrow^* (\nu\,\tilde{w})(\xi{\cdot}u : M\xi,\ \sigma \cdot \sigma'_0) \models C_0\ \wedge\ E\\ &\longrightarrow^* (\nu\,\tilde{v}\tilde{w})(\xi{\cdot}u : V',\ \sigma') \models C'\end{aligned}$$

where the mixing of reduction and satisfiability should be easily understood. Since the update of $\tilde{r}$ satisfies the assertion $C_0 \wedge E$ (because other parts of the store do not affect its validity), we know:

$$\{C\}\,(\tilde{r} := \sigma(\tilde{r}); M)\xi,\ \sigma_0)\ :_u\ \{C'\} \tag{6.14}$$

By $\models \{C\}F\ :_u\ \{C'\}$ we have (again with appropriate existential quantifications)

$$(\xi{\cdot}u : F'\xi,\ \sigma) \longrightarrow^* (\nu\,\tilde{w})(\xi{\cdot}u : F\xi,\ \sigma_0 \cdot \sigma'_0) \models C \longrightarrow^* (\nu\,\tilde{v}\nu\,w)(\xi{\cdot}u : V'',\ \sigma'') \models C'$$

By (IH1) and (6.14), we have reached $(\nu\,\tilde{v}\tilde{w})(\xi{\cdot}u : V'',\ \sigma'') \sqsubseteq (\nu\,\tilde{v}\tilde{w})(\xi{\cdot}u : V',\ \sigma')$, as required. □

We are now ready to establish the main result of this section, after a definition.

Definition 6.36 (logical equivalence) Assuming $\Gamma;\Delta \vdash M_1 : \alpha$ and $\Gamma;\Delta \vdash M_2 : \alpha$, we write $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$ when $\models \{C\}M_1^{\Gamma;\Delta\alpha}\ :_u\ \{C'\}$ iff $\models \{C\}M_2^{\Gamma;\Delta;\alpha}\ :_u\ \{C'\}$.

Theorem 6.37 *Let* $\Gamma;\Delta \vdash M_{1,2} : \alpha$. *Then* $\Gamma;\Delta \vdash M_1 \cong M_2 : \alpha$ *if and only if* $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$.

PROOF: The "only if" direction is direct from the definition of the model. For the "if" direction, we prove the contrapositive. Suppose $M_1 \cong_{\mathcal{L}} M_2$ but $M_1 \not\cong M_2$. By abstraction, we can safely assume $M_1, 2$ are semi-closed values. By Lemma 6.25, there exist semi-closed FCF values F and $\tilde{U}$ such that, say,

$$(FM_1, \tilde{r} \mapsto \tilde{U}) \Downarrow \qquad \text{and} \qquad (FM_2, \tilde{r} \mapsto \tilde{U}) \Uparrow. \tag{6.15}$$

By Proposition 6.34, there are assertions which characterise F and $\tilde{U}$. Let the characteristic formula for F at f be written $[\![F]\!](f)$. We now reason:

$$\begin{aligned}&(FM_1,\ \tilde{r} \mapsto \tilde{U}) \Downarrow\\ &\ \Rightarrow (f : [F]{\cdot}m : [M_1]\ \models\ \{\wedge_i [\![U_i]\!](!r_i)\}\ f \bullet m \searrow_z \{\mathsf{T}\}\\ &\ \Rightarrow \forall\kappa.\ (f : \kappa \models [\![F]\!]_f\ \supset\ (f : \kappa{\cdot}m : [M_1]\ \models\ \{\wedge_i [\![U_i]\!]_{!r_i}\}\ f \bullet m\ \searrow_z \{\mathsf{T}\}\\ &\ \Rightarrow\ \models \{\mathsf{T}\}\ M_1 :_m \{\forall f.\{[\![F]\!](f)\ \wedge\ (\wedge_i [\![U_i]\!](!r_i))\}\ f \bullet m\ \searrow_z \{\mathsf{T}\}\}\end{aligned}$$

But by (6.15) we have

$$\not\models \{\mathsf{T}\}\ M_2 :_m \{\forall f.\{[\![F]\!](f)\ \wedge\ (\wedge_i [\![U_i]\!](!r_i))\}\, f \bullet m \searrow_z \{\mathsf{T}\}\}$$

that is $M_1 \not\cong_{\mathcal{L}} M_2$, a contradiction. Thus we conclude $M_1 \cong M_2$, as required. □

7 Reasoning Examples

This section demonstrates how the proposed logical machinery enables accurate description and reasoning about a variety of programs which combine functions with local state and which are hard to treat with existing methods. We extensively use the notations for freshness from § 3.4.

7.1 Stored Procedures with Shared Local State

We first show how the logic can precisely reason about a hidden state shared by stored procedures, taking `IncShared` in Introduction as an example. We use:

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

We now reason for `IncShared`, using [*New*], showing the key inference steps. For brevity we work with the implicit global assumption that a, b, c_1, c_2 are pairwise distinct and omit an anchor from the judgement when the return value is a unit type.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{1.\{\top\}\ \texttt{Inc} :_u \{\nu x.\mathsf{inc}'(u,x,0)\}}{2.\{\top\}\ a := \texttt{Inc}\ \{\nu x.\mathsf{inc}'(!a,x,0)\}}}{3.\{\mathsf{inc}'(!a,x,0)\}\ b :=!a\ \{\mathsf{inc}'(!a,x,0) \wedge \mathsf{inc}'(!b,x,0)\}}}{4.\{\mathsf{inc}'(!a,x,0)\}\ c_1 := (!a)()\ \{\mathsf{inc}'(!a,x,1) \wedge !c_1 = 1\}}}{5.\{\mathsf{inc}'(!b,x,1)\}\ c_2 := (!b)()\ \{\mathsf{inc}'(!b,x,2) \wedge !c_2 = 2\}}}{6.\{!c_1 = 1 \wedge !c_2 = 2\}\ (!c_1) + (!c_2) :_u \{u = 3\}}}{7.\{\top\}\ \texttt{IncShared} :_u \{\nu x.u = 3\}}}{8.\{\top\}\ \texttt{IncShared} :_u \{u = 3\}}$$

Line 1 is by the application of [*New*].[5] In Line 7, we used the following derived "scope-opening" rule.

$$[\mathit{SeqOpen}]\ \dfrac{\{C_0\}M\{\nu x.C\} \quad \{C\}N :_u \{C_1\}}{\{C_0\}\ M;N :_u \{\nu x.C_1\}}$$

We contrast the above inference with that of:

$$a := \texttt{Inc}; b := \texttt{Inc}; c_1 := (!a)(); c_2 := (!b)(); (!c_1 + !c_2)$$

[5] In the short version, we use [*LetRef*] which is identical with [*New*] in this long version. Its mapping and derivation using $\texttt{ref}(M)$ are found in (2.1) in § 2.1 and page 28 in § 5.2, respectively.

Call this program `IncUnShared`, which assigns to a and b two separate instances of `Inc`. The lack of sharing in `IncUnShared` is captured by the following derivation:

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

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

Simple as they look, we do not know preceding Hoare-like logics which can derive these specifications.

7.2 Dynamic Mutable Data Structure (1): Trees

Imperative higher-order functions with local state offer a surprisingly versatile medium for clean, rigorous description of algorithms which manipulate dynamically generated mutable data structures. In the following we explore how we can reason about these algorithms tractably and efficiently. One of the aims of our experiments is to see whether the general nature of the proposed logic leads to remarkably effective reasoning principles for mutable dynamic data structures with different degrees of sharing, a hard topic since the inception of program logic.

Let us start from a data structure which has the least sharing, *trees*. In this long version, to compare with the method by Separation Logic in [37, §6], we verify essentially the same algorithm. The following program creates a new isomorphic copy of a given tree.

$$\begin{array}{ll}
\texttt{treeCopy} \overset{\text{def}}{=} \mu f.\lambda x.\texttt{case}\ !x\ \texttt{of} & \\
\qquad\qquad \texttt{inj}_1(n) : \texttt{ref}(\texttt{inj}_1(n)) & \\
\qquad\qquad \texttt{inj}_2(\langle y_1, y_2\rangle) : \texttt{ref}(\texttt{inj}_2(\langle f y_1, f y_2\rangle)) &
\end{array}$$

To type this program, we use recursive types, introduced in Section 6.1 (as noted there, no change in semantics and proof rules is necessary). We set the type of mutable trees as $\mathit{Tree} \overset{\text{def}}{=} \mu \mathrm{X}.(\mathsf{Ref}(\mathsf{Nat} + (\mathrm{X} \times \mathrm{X})))$, and can easily check that `treeCopy` has type $\mathit{Tree} \to \mathit{Tree}$.

Before asserting for `treeCopy`, we first specify trees. To compare with with the method by Separation Logic in [37, §6] precisely, we first use the *S*-structures to represent the assertions. Subsection **??** shows the verification without *S*-structures.

$$\begin{aligned}
\mathsf{tree}\, e^{\mathsf{Nat}}\,(u) &\equiv \mathsf{atom}(!u, e^{\mathsf{Nat}})\\
\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(u) &\equiv \exists m_1 m_2.(\mathsf{branch}(!u, m_1, m_2) \wedge m_1 \star m_2 \wedge \\
&\qquad\qquad \textstyle\bigwedge_{i=1,2}(\mathsf{tree}\,\tau_i\,(m_i) \wedge u \# m_i))\\
\mathsf{atom}(u, n) &\equiv u = \mathtt{inj}_1(n)\\
\mathsf{branch}(u, y_1, y_2) &\equiv u = \mathtt{inj}_2(\langle y_1, y_2\rangle)
\end{aligned}$$

So a tree is either an atom (which is a leaf with a numeral) or a branch with two mutually disjoint subtrees ($m_1 \star m_2$) where the top is also unreachable from the subtrees ($u\# m_i$). τ in $\mathsf{tree}\,\tau\,(x)$ is an *S-expression*, such as $((1,2),3)$, which uniquely determines the shape of a tree. Its use follows Reynolds [37] and is often convenient. We include *S*-expressions among standard terms in our logical language. For reference, the grammar is given as:

$$\tau \ ::= \ x \mid e^{\mathsf{Nat}} \mid (\tau_1, \tau_2).$$

Symbols $\tau, \tau', \ldots$ are also used as variables for *S*-expressions for readability. Thus $\mathsf{tree}\,\tau\,(u)$ is in fact a binary predicate.

We can now assert for `treeCopy`, naming it *u*.

$$\mathsf{treecopy}(u) \ \stackrel{\mathrm{def}}{=}\ \forall x, \tau. \{\mathsf{tree}\,\tau\,(x)\} u \bullet x = y \{\star y. \mathsf{tree}\,\tau\,(y)\} @ \emptyset \qquad (7.1)$$

$\mathsf{treecopy}(u)$ reads:

> *If u is fed with a tree of shape τ, then it will return, without observable write effects, a tree of the same shape whose nodes are all fresh and unreachable from existing data structures.*

As far as its argument is restricted to trees, this is a full specification of `treeCopy`. As a result, it entails, often through easy syntactic calculation, other assertions the program satisfies. For example it immediately implies:

$$\mathsf{treecopyS}\,(u) \ \stackrel{\mathrm{def}}{=}\ \forall x, \tau. \{\mathsf{tree}\,\tau\,(x)\} u \bullet x = y \{\mathsf{tree}\,\tau\,(x) \wedge \mathsf{tree}\,\tau\,(y) \wedge x \star y\}$$

which is essentially equivalent to Reynolds's assertion in [37] (the converse implication may not hold even we add "$@\emptyset$" to $\mathsf{treecopyS}\,(u)$). The difference between $\mathsf{treecopy}(u)$ and $\mathsf{treecopyS}\,(u)$ is that the former says the program creates a fresh tree, while the latter says its execution results in the original tree and another which are mutually disjoint (the latter in fact leaves a possibility for sharing between the new tree and some old data structure).

Apart from the assertion itself, a highlight of this subsection is how efficient reasoning principles for a specific but significant class of data structures arise as derived proof rules through concrete derivation. Our aim is to prove the following judgement:

$$\{\mathsf{T}\}\ \mathtt{treeCopy} :_u \{\mathsf{treecopy}(u)\} @ \emptyset \qquad (7.2)$$

We use derived rules for $\star$-freshness in Figure 10. These rules use inductive nature of $\star$-freshness (e.g. if *x* and *y* are $\star$-fresh so is $\langle x, y\rangle$, etc.: In fact there is a rule to treat a

Fig. 10 Proof Rules for $\star$

$$[\mathit{Simple}\star]\ \frac{\text{no reference occurs in } e}{\{C\}\ e :_u \{\star u.(C \wedge u = e)\}@\emptyset} \qquad [\mathit{Ref}\star]\ \frac{\{C\}\ M :_m \{\star u.C'\}@\tilde{e}}{\{C\}\ \mathtt{ref}(M) :_u \{\star u.C'[!u/m]\}@\tilde{e}}$$

$$[\mathit{Pair}\star]\ \frac{C \equiv C_1 \wedge [!\tilde{e}_1]C_2 \quad \{C_i\}\ M_i :_{m_i} \{\star m_i.C_i'\}@\tilde{e}_i \quad C' \equiv \langle !\tilde{e}_2\rangle C_1' \wedge C_2'}{\{C\}\ \langle M_1, M_2\rangle :_u \{\star u.\exists\tilde{m}.(u = \langle m_1, m_2\rangle \wedge m_1 \star m_2 \wedge C')\}@\tilde{e}_1\tilde{e}_2}$$

$$[\mathit{Inj}\star]\ \frac{\{C\}\ M :_m \{\star m.C'\}@\tilde{e}}{\{C\}\ \mathtt{inj}_j(M) :_u \{\star u.\exists m.(C' \wedge u = \mathtt{inj}_j(m))\}@\tilde{e}}$$

chunk of "fresh" constructors in one go, saving further inference steps) and are derivable from the original proof rules: these specialised rules make the best of the special nature of a class of data structures, here the lack of sharing.
We also use the following rule for typed pattern matching:

$$[\mathit{CaseMatch}]\ \frac{\{C \wedge e = \mathtt{inj}_i(e_i)\}\ M_i :_m \{C'\}@\tilde{e}_i \quad i = 1,2}{\{C\}\ \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inj}_i(e_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}@\tilde{e}_1\tilde{e}_2}$$

We first set $\mathsf{treecopy}'\,\tau\,(u)$ so that $\mathsf{treecopy}(u) \equiv \forall\tau.\mathsf{treecopy}'\,\tau\,(u)$, and:

$$C \stackrel{\text{def}}{=} \mathsf{tree}\,\tau\,(x)\ \wedge\ \forall\tau' < \tau.\ \mathsf{treecopy}'\,\tau'\,(f)$$
$$A_i \stackrel{\text{def}}{=} \mathsf{tree}\,\tau_i\,(y_i)\ \wedge\ \tau_i < \tau\ \wedge\ \forall\tau' < \tau.\ \mathsf{treecopy}'\,\tau'\,(f)$$

where $\tau' < \tau$ is the lexical ordering on trees, used for induction for recursion (as we shall see in the next subsection, we can dispense with τ and use a different ordering). We start from the case branch for leaves.

$$\frac{1.\ \{C \wedge \mathsf{atom}(!x, n)\}\ \mathtt{inj}_2(n) :_m \{\star m.(\mathsf{atom}(m, n) \wedge \tau = n)\}@\emptyset \quad \text{(Simple)}}{2.\ \{C \wedge \mathsf{atom}(!x, n)\}\ \mathtt{ref}(\mathtt{inj}_1(n)) :_u \{\star u.\mathsf{tree}\,\tau\,(u)\}@\emptyset \qquad \text{(Ref)}}$$

Thus all leaves are $\star$-fresh. We now present the rest of the reasoning, including the induction. $\star$-freshness is built up starting from induction hypothesis. From Line 8, we set M so that $\texttt{treeCopy} \stackrel{\text{def}}{=} \mu f.\lambda x.M$ for brevity.

3. $A_i \supset \{A_i\} f \bullet y_i = z_i \{\star z_i.\mathsf{tree}\,\tau_i\,(z_i)\} @\emptyset$

4. $\{A_i\}\ f y_i :_{z_i} \{\star z_i.\mathsf{tree}\,\tau_i\,(z_i)\} @\emptyset$ (AppSimple)

5. $\{A_1 \wedge A_2\}$ $\texttt{inj}_2(\langle f y_1, f y_2\rangle)_u$ $\{\star u.\exists \tilde{m}.(\mathsf{branch}(u, m_1, m_2) \bigwedge_{i=1,2} \mathsf{tree}\,\tau_i\,(m_i) \wedge m_1 \star m_2)\} @\emptyset$ (Pair, Inj)

6. $\{A_1 \wedge A_2\}\ \texttt{ref}(\texttt{inj}_2(\langle f y_1, f y_2\rangle)) :_u \{\star u.\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(u)\} @\emptyset$ (Ref)

7. $\{C \wedge \mathsf{branch}(!x, y_1, y_2)\}\ \texttt{ref}(\texttt{inj}_2(\langle f y_1, f y_2\rangle)) :_u \{\star u.\mathsf{tree}\,\tau\,(u)\} @\emptyset$ Consequence

8. $\{\mathsf{tree}\,\tau\,(x) \wedge \forall \tau' < \tau.\mathsf{treecopy}'\,\tau'\,(f)\}\ M :_u \{\star u.\mathsf{tree}\,\tau\,(u)\} @\emptyset$ (2, 7, CaseMatch)

9. $\{\forall \tau' < \tau.\mathsf{treecopy}'\,\tau'\,(u)\}\ \lambda x.M :_u \{\mathsf{treecopy}'\,\tau\,(u)\} @\emptyset$ (Lam)

10. $\{\mathsf{T}\}\ \texttt{treeCopy} :_u \{\mathsf{treecopy}(u)\} @\emptyset$ (Rec)

We have arrived at (7.2).

7.3 Dynamic Mutable Data Structures (2): DAGs

We continue our experiments, inspecting whether our observation on trees scale to more complex data structures. We treat directed acyclic graphs, or *dags*, which allow more sharing than trees. This example is treated as one of the benchmark examples by Bornat and others [5].

A dag has the same type as *Tree*, but its specification is more liberal. Again using S-expressions, $\mathsf{dag}\,\tau\,(x)$ asserts x is a dag whose leaves are labelled as τ. The base case $\mathsf{dag}\,n\,(x)$ is the same as trees:

$$\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(u) \equiv \exists m_1 m_2.(\mathsf{branch}(!u, m_1, m_2) \wedge \bigwedge_{i=1,2}(\mathsf{dag}\,\tau_i\,(m_i) \wedge u \# m_i))$$

Thus a dag is the same as a tree except, at each branch, two subgraphs can share each other's nodes. It still has a rigid hierarchical structure: the top of a dag is unreachable from subgraphs. τ in $\mathsf{dag}\,\tau\,(x)$ no longer uniquely determines its shape, for example a leaf labelled 2 in $((1,2),(2,1))$ may or may not be shared.

Suppose that we wish to create a new dag from a τ-labelled existing dag. If we use $\texttt{treeCopy}$, we lose the original sharing structure — it produces a fresh τ-labelled tree (the verification that $\{\mathsf{dag}\,\tau\,(x)\} u \bullet x = y \{\star y.\mathsf{tree}\,\tau\,(y)\} @\emptyset$ is satisfied by $\texttt{treeCopy}$ is literally identical with §7.2). So if we are to preserve sharing, we need to slightly change the algorithm. One such change follows.

```
dagCopy  $\stackrel{\text{def}}{=}$ $\lambda g$.new $x$ := $\emptyset$ in Main $g$
   Main  $\stackrel{\text{def}}{=}$ $\mu f.\lambda g$.if dom($!x, g$) then get($!x, g$) else
                  case $!g$ of
                  inj$_1(n)$ : newEntry(inj$_1(n), g$)
                  inj$_2(y_1, y_2)$ : newEntry(inj$_2(\langle f y_1, f y_2\rangle), g$)
newEntry $\stackrel{\text{def}}{=}$ $\lambda(y, g)$.let $g'$ = ref($y$) in ($x$ := put($!x, \langle g, g'\rangle$); $g'$)
```

When the program is called with the root of a dag, it first creates an empty table and stores it to a local variable x. The table will remember those nodes in the original dag which have already been processed, associating them with the corresponding nodes in the fresh dag. The rest works as `treeCopy` except, 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 clarity, a pre-defined data type[6] for a table inducing a finite function with a pre-defined "API", as will be discussed later.

Main Assertion. A key property of `dagCopy` is that it creates a fresh dag preserving the original's sharing structure. To discuss such matters with precision, a simple way is to use *path expressions*:

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

A path expression (hereafter simply *path*) represents a way to traverse a dag from one node to another in the forward direction of directed edges. Taking in paths as part of terms, reachability from g to g' through p is easily defined as:

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

The first clause says that the empty path leads us from g to g; the second that $l.p$ leads from g to g' iff we go left from g (which should be a branch node) and, from there, p leads us to g'. The third is the symmetric case.

Next $\mathsf{match}(g, p_1, p_2)$ asserts two paths $p_{1,2}$ from g lead to the same node, whereas $\mathsf{leaf}(g, p, n)$ says we reach a leaf of label n from g following p, defined as:

$$\begin{aligned}
\mathsf{match}(g, p_1, p_2) &\equiv \exists y.(\mathsf{path}(g, p_1, y) \wedge \mathsf{path}(g, p_2, y)) \\
\mathsf{leaf}(g, p, n) &\equiv \exists y.(\mathsf{path}(g, p, y) \wedge \mathsf{atom}(y, n))
\end{aligned}$$

The isomorphism between two collections of nodes, respectively reachable from g and g', as labelled directed graphs, is defined as follows.

$$\begin{aligned}
\mathsf{iso}(g, g') \equiv\; &\forall p_1 p_2.(\mathsf{match}(g, p_1, p_2) \equiv \mathsf{match}(g', p_1, p_2)) \\
&\wedge \forall pn.(\mathsf{leaf}(g, p, n) \equiv \mathsf{leaf}(g', p, n))
\end{aligned}$$

We assert for `dagCopy`, named u.

$$\mathsf{dagcopy}(u) \;\equiv\; \forall \tau, g^{Tree}.\{\mathsf{dag}\,\tau\,(g)\} u \bullet g = g' \{\star g'.\mathsf{iso}(g, g')\} @ \emptyset$$

The assertion says:

> *Whenever* `dagCopy` *is invoked with a dag g, it creates a fresh dag isomorphic to g, without any write effects.*

[6] The data type is in fact realisable as, say, lists.

As may be expected, $\mathsf{dagcopy}(u)$ (strictly) entails $\mathtt{treecopy}(u)$. Again disjointness of a created dag from the existing dag is entailed without being stated, cf. [5]. The judgement we wish to establish is:

$$\{\mathsf{T}\}\mathtt{dagCopy} :_u \{\mathsf{dagcopy}(u)\}@\emptyset \tag{7.3}$$

The derivation is essentially mechanical, which we list below.

Intermediate Assertion The intermediate assertion specifies for a "scratch pad" x which stores a table associating already created new nodes with their originals.

$$\begin{array}{l} \mathsf{dc}\,\tau(u) \\ \overset{\text{def}}{=}\ \forall g, org. \\ \quad \{\mathsf{dag}\,\tau(g) \wedge !x = org \wedge \mathsf{con}(org)\} \\ \quad\quad u \bullet g = g' \\ \quad \{\#^{\text{-}x}\{z \,|\, g' \hookrightarrow z \wedge z \notin \mathsf{cod}(org)\}.(\mathsf{con}(!x) \wedge !x = org \cup \langle g, g' \rangle^*)\}\,@x \end{array}$$

which says:

Suppose g is a dag and x contains a table org which is consistent (i.e. only relates isomorphic nodes). Then invocation of u with g terminates with the return value g' and, moreover: (1) references names reachable from g' minus those in the codomain of org are freshly generated; and (2) x stores a table which is consistent and which adds to org the set of co-reachable nodes from $\langle g, g' \rangle$. Further the invocation only modifies x.

Notations used in the assertion are illustrated in the following.

Consistency. Both the pre/post conditions of (7.4) use the invariant $\mathsf{con}(t)$ ("t is consistent") where t is a table (finite map) mapping original graph nodes to the corresponding newly created graph nodes (detailed later). The predicate is given as:

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

$\mathsf{con}(t)$ says t only associates isomorphic graphs, and that its domain (hence co-domain, by isomorphism) is closed under reachability (the notations such as $\langle g, g' \rangle \in t$ are illustrated later).

$\#$-Freshness. The post-condition uses the predicate of the form $\#^{\text{-}x}\{z|C_0(z)\}.C$ which refines the predicate given at the end of Section 5.3. The predicate is defined as, with i and X fresh:

$$\#^{\text{-}x}\{z|C_0(z)\}.C \overset{\text{def}}{=} \forall \mathrm{X}.\forall \mathrm{z}^{\mathsf{Ref}(\mathrm{X})}.((C_0(z) \wedge x \# i) \supset z \# i)\ \wedge\ C \tag{7.4}$$

which says:

Each reference name z satisfying $C_0(z)$ is #-fresh w.r.t. any datum in the pre-state except those which can reach x (which includes x itself by definition).

The reservation on x is needed since newly generated nodes are immediately stored in x, so that ♯-freshness of a newly generated node does not hold w.r.t. x (and any datum from which we can reach x: in fact, in the present case, x is only reachable from x, so $x\#i$ is the same thing $x \neq i$).

Fig. 11 Proof Rules for ♯

$$[Ref\#]\ \frac{\{C\}\ M :_m \{\#\{z|E\}.C'\}@\tilde{e}}{\{C\}\ \mathtt{ref}(M) :_u \{\#\{z|E[!u/m] \vee z = u\}.C'[!u/m]\}@\tilde{e}}$$

$$[Pair\#]\ \frac{\begin{array}{l}\{C\}\ M_1 :_{m_1} \{\#\{z|G_1\}.C_1'\}@\tilde{e}\\ \{C_1'\}\ M_2 :_{m_2} \{\#\{z|G_2\}.C'[\langle m_1, m_2\rangle/u]\}@\tilde{e}\\ C \supset x \in \tilde{e}\end{array}}{\{C\}\ \langle M_1, M_2\rangle :_u \{\#\{z|G_1 \vee G_2\}.C'\}@\tilde{e}}$$

$$[Inj\#]\ \frac{\{C\}\ M :_m \{\#\{z|E[\mathtt{inj}_j(m)/u]\}.C'[\mathtt{inj}_j(m)/u]\}@\tilde{e}}{\{C\}\ \mathtt{inj}_j(M) :_u \{\#\{z|E\}.C'@\tilde{e}\}}$$

$$[App\#]\ \frac{\begin{array}{l}\{C\}\ M_1 :_{m_1} \{\#\{z|G_1\}.C_1\}@\tilde{e}\\ \{C_1\}\ M_2 :_{m_2} \{\#\{z|G_2\}.(C_2 \wedge \{C_2\}m_1 \bullet m_2 = u\{\#\{z|G_3\}.C'\}@\tilde{e})\}@\tilde{e}\\ C \supset x \in \tilde{e}\end{array}}{\{C\}\ M_1M_2 :_u \{\#\{z| \vee_{i=1,2,3} G_i\}.C'\}@\tilde{e}}$$

$$[AssVar\#]\ \frac{\{C\}M :_m \{C'\{\!|m/!x|\!\} \wedge r\#i\}@\tilde{e}}{\{C\}\ x := M\ \{C' \wedge (x\#i \supset r\#i)\}@\tilde{e}x}$$

We use the proof rules for #-freshness in Figure 11. For the notation for #-freshness see (7.4), §7.3. Among the presented rules, the rule $[AssVar\#]$ says:

If r is unreachable from i, then writing some value to x may change reachability to i from x, hence from any datum from which we can reach x, but nothing else.

The rule is in fact an instance of $[Assign]$.

Axioms for Table APIs. Tables are finite maps on *Tree*-typed references, equipped with the following three procedures.

- $\mathtt{get}(t, g)$ to get the image of g in t.
- $\mathtt{put}(t, \langle g, g'\rangle)$ to add a new tuple $\langle g, g'\rangle$ when g is not in the domain of t.
- $\mathtt{dom}(t, g)$ (resp. $\mathtt{cod}(t, g)$) judges if g is in the pre-image (resp. image) of t.

We also use $\emptyset$ for the empty table. By the following axioms we can treat a table as a set-theoretic finite map. Below $f(x)\searrow e$ stands for $\{\mathsf{T}\}f \bullet x = y\{y = e\}@\emptyset$.

$$\begin{array}{l}
\forall g.\mathtt{dom}(\emptyset,g)\searrow\mathsf{f}\\
\forall t,g.(\mathtt{dom}(t,g)\searrow\mathsf{t} \vee \mathtt{dom}(t,g)\searrow\mathsf{f})\\
\mathtt{dom}(t,g)\searrow\mathsf{t} \;\equiv\; \exists g'.\mathtt{get}(t,g)\searrow g'\\
\mathtt{get}(t,g)\searrow g' \wedge \mathsf{extend}(t,t') \;\supset\; \mathtt{get}(t',g)\searrow g'\\
\mathtt{dom}(t_0,g_0)\searrow\mathsf{f} \;\supset\; \exists t_1.(\mathtt{put}(t_0,\langle g,g'\rangle)\searrow t_1 \;\wedge\; \mathtt{get}(t_1,g)\searrow g')\\
\mathtt{dom}(t_0,g_0)\searrow\mathsf{f} \;\wedge\; \mathtt{put}(t_0,\langle g_1,g_1'\rangle)\searrow t_1 \;\wedge\; g_0 \neq g_1 \;\supset\; \mathtt{dom}(t_0,g_0)\searrow\mathsf{f}
\end{array}$$

We omit the axioms for $\mathtt{cod}(t,g')$ which are symmetric to $\mathtt{dom}(t,g)$. $\mathsf{extend}(t,t')$ further says t' adds zero or more tuples to t, i.e.:

$$\mathsf{extend}(t,t') \;\equiv\; t = t' \vee \exists t_0,g,g'.(\mathsf{extend}(t,t_0) \wedge \mathtt{dom}(t_0,g)\searrow\mathsf{f} \wedge \mathtt{put}(t_0,\langle g,g'\rangle)\searrow t')$$

The axioms above allow us to use the following set-theoretic notations without loss of precision.

- $\langle g,g'\rangle \in t$ stands for $\mathtt{get}(t,g)\searrow g'$.
- $g \in \mathtt{dom}(t)$ stands for $\mathtt{dom}(t,g)\searrow\mathsf{t}$.
- $t = t_1 \cup t_2$ stands for $\forall g,g'.(\langle g,g'\rangle \in t \;\equiv\; \bigvee_{i=1,2}\langle g,g'\rangle \in t_i)$.

$\langle g,g'\rangle^*$ is all pairs of nodes co-reachable from g and g', i.e.

$$\langle g,g'\rangle^* \;\stackrel{\text{def}}{=}\; \{\langle z,z'\rangle \mid \exists p.(\mathsf{path}(g,p,z) \wedge \mathsf{path}(g',p,z'))\} \qquad (7.5)$$

Derivation (1): The Whole Program We first look at the derivation for the whole program.

```
$u : \{\mathsf{T}\}$  (abs)
lambda g.
    $g' : \{\mathsf{dag}\,\tau(g)\}$  (new)
    new x:= $\emptyset$ in
        $g' : \{!x = \emptyset\}$  (app)
            $m : \{\mathsf{T}\}$
            Main
            $\{\forall\tau.\mathsf{dc}\,\tau(m)\}@\emptyset$
            g
            $\{(!x = \emptyset \wedge \mathsf{dag}\,\tau(g)) \;\wedge$
             $\{!x = \emptyset \wedge \mathsf{dag}\,\tau(g)\}m \bullet g = g'\{\#^{-x}S.\mathsf{iso}(g,g')\}@x\}@\emptyset$
        $\{\#^{-x}S.\mathsf{iso}(g,g')\}@x$
    $\{\star g'.\mathsf{iso}(g,g')\}@\emptyset$
$\{\forall g.\{\mathsf{dag}\,\tau(g)\}u \bullet g = g'\{\star g'.\mathsf{iso}(g,g')\}@\emptyset\}@\emptyset$
```

Above we set, for brevity:

$$S(g,org) = \{z \mid g \hookrightarrow z \;\wedge\; \mathtt{cod}(org,z)\searrow\mathsf{f}\}. \qquad (7.6)$$

Further we write S for $S(g',org)$. Some illustrations:

– l.1 introduces the anchor u, without any pre-condition, while l.3 introduces the assumption on the argument g'.
– In l.5, we introduce the anchor g' and note the subsequent sub-derivation are for the application.
– l.6 introduces an anchor for the function part (the “main” program). The pre-condition T means it does not add any further assumption (the subsequent reasoning may use what has been assumed so far, for example l.3).
– In l.8, we conclude `Main` named m satisfies $\forall\tau.\mathsf{dc}\,\tau(m)$, to be inferred later.
– In l.12, the application is inferred using [*App*#] (though the standard application rule suffices in this case).
– From l.12 to l.13, we use the [*New*] rule. This part may deserve some illustration. Let, for brevity: $M \overset{\text{def}}{=} \texttt{Main}\,g$ and $C \overset{\text{def}}{=} !x = \emptyset \wedge \mathsf{dag}\,\tau(g)$. Note the sub-derivation l.5–12 means $\{C\}M\{\#^{-x}S.\mathsf{iso}(g,g')\}$, that is, with i fresh (cf.(7.4):

$$\{C\}M\{\forall z.((g' \hookrightarrow z \wedge x\#i) \supset z\#i) \wedge \mathsf{iso}(g,g')\}@x$$

By [*New*], we can strengthen C with $x\sharp i$. Since $\forall j \neq x \supset j\#x$ holds, we can apply (6.2), page 37, Section 6.1, so that the predicate $x\#i$ is $!x$-free. By [*Inv*] (the invariance rule for located assertions), we obtain:

$$\{C \wedge x\#i\}M\{\forall z.(g' \hookrightarrow z \supset z\#i) \wedge \mathsf{iso}(g,g')\}@x$$

that is

$$\{C \wedge x\#i\}M\{\star g'.\mathsf{iso}(g,g')\}@x.$$

Since [*New*] allows us to cancel x in the pre-condition, we obtain l.13.

Derivation (2): NewEntry The derivation for `NewEntry` is given below.

```
1  m : {T}  (abs)
2  lambda (y,g).
3      g′ : {!x = org  ∧  dom(org,g) ↘ f}  (let)
4      let h=
5          h : {T}  (ref-simple)
6          ref(y) in
7          {!x = org  ∧  dom(org,g) ↘ f  ∧  !h = y  ∧  h#i}@∅
8      g′ : {···}  (seq)
9          {T}  (assvar#)
10         x:=
11             m : {T}
12             put(!x, <g,h>);
13             {m = org∪{⟨g,h⟩}  ∧  !x=org  ∧  !h=y  ∧  h#i}@∅
14         {!x = org∪{⟨g,h⟩  ∧  !h = y  ∧  (x#i ⊃ h#i)}}@x
15         h
16     {!x = org∪{⟨g,g′⟩}  ∧  !g′ = y  ∧  (x#i ⊃ g′#i)}@x
17 {NE(m)}@∅
```

Above:

1. $\{\cdots\}$ indicates the immediately preceding assertion is repeated.
2. The program uses h instead of g' in for `let` to avoid the collision of names (though h is in effect named g' in the derivation).
3. In the last line, $NE(m)$ is given as:

$$\begin{array}{l} NE(m) \\ \overset{\text{def}}{=}\ \forall org, y, g. \\ \quad \{!x = org \wedge \mathtt{dom}(org, g) \searrow \mathsf{f}\}\ m \bullet (y, g) = g' \{\#^{\text{-}x} g'.(!x = org \cup \{\langle g, g' \rangle\} \wedge !g' = y)\}@x \end{array}$$

We illustrate the derivation line-by-line.

– l.1 introduces the anchor for this program. In l.2, we recall y is the content of a fresh reference to be created, and g is the original node.
– l.3 introduces g' as the anchor for the `let`-block, which continues up to l.16. The assertion says that the table org stored in x does not contain g (i.e. g has not been processed yet).
– l.4 introduces the variable h for the `let`.
– l.5–7 reason for the argument $\mathtt{ref}(y)$ of the `let` command.
– l.8–16 reason for the `let`-body. Since the `let` command as a whole is named g' (in l.3), its body should also be named g' in l.8. In the same line, the pre-condition repeats the post-condition obtained in l.7. l.8 also mentions the `let`-body itself is a sequential composition.
– l.9–14 reason for the first part of the sequential composition, the assignment to a variable x, using $[AssVar\#]$. l.9 does *not* introduce an anchor since the assignment is of the Unit-type. The precondition repeats the previous one.
– l.11–13 reasons for the argument (named m), with the repeated precondition. The reasoning uses the following valid assertion:

$$\begin{array}{l} g_0 \notin \mathtt{dom}(t_0) \wedge \mathtt{put}(t_0, \langle g_0, g_0' \rangle) \searrow t_1 \\ \qquad \forall g, g'.(\langle g, g' \rangle \in t_1 \ \equiv\ (\langle g, g' \rangle \in t_0 \vee (g = g_0 \wedge g' = g_0'))) \end{array}$$

which is easily derived from the axioms in §7.3. The post-condition says m is the result of adding $\langle g, h \rangle$ to org.
– Since x is not aliased (by its type), l.14 simply replaces $!x$ for m in l.13. Similarly, since the variable h is returned and this is named g' in the assertion, l.16 replaces g' for h in l.14.

Derivation (3): `Main` We list the derivation for the main program below.

```
1  u : {T}  (rec)
2  mu f.
3       u : {∀τ' < τ.  dcτ'(f)}  (abs)
4       lambda g.
5            g' : {dag τ(g)  ∧  !x = org  ∧  con(org)}  (if)
```

```
6          if dom(!x,g) then
7              $g' : \{\mathtt{dom}(!x,g) \searrow \mathtt{t}\}$
8              get(!x,g)
9              $\{!x = org \ \wedge \ \mathtt{get}(t,g) \searrow g'\}@\emptyset$
10             $\{DG\}@\emptyset$
11         else
12             $g' : \{\mathtt{dom}(!x,g) \searrow \mathtt{f}\} \ \ (\mathtt{case})$
13             case !g of
14                inl(n):
15                   $g' : \{\mathsf{dag}\, n\, (g)\}$
16                   NewEntry(inl(n), g)
17                   $\{\#^{-x} g'.(\mathsf{con}(!x) \ \wedge \ !x = org \cup \langle g, g' \rangle)\}@x$
18                   $\{DG\}@x$
19                inr(g1, g2):
20                   $g' : \exists \tau_{1,2}.(\tau = \langle \tau_1, \tau_2 \rangle \bigwedge_{i=1,2} \mathsf{dag}\, \tau_i\, (g_i))$
21                   NewEntry(inr(<fg1, fg2>),g)
22                   $\{DG\}@x$
23             $\{DG\}@x$
24         $\{DG\}@x$
25     $\{\mathsf{dc}\, \tau\, (u)\}@\emptyset$
26 $\{\forall \tau.\mathsf{dc}\, \tau\, (u)\}@\emptyset$
```

Above we use the predicate DG (for Dag Generated) which is set to be:

$$DG \overset{\mathrm{def}}{=} \#^{-x} S(g, org).(\mathsf{con}(!x) \wedge !x = org \cup \langle g, g' \rangle^*) \tag{7.7}$$

where we set $S(g, org) = \{z \mid g \hookrightarrow z \ \wedge \ \mathtt{cod}(org, z) \searrow \mathtt{f}\}$. The set-based $\sharp$-notation used above is easily decoded into universal quantification, but offers transparent reasoning

The reasoning is simple except the second branch of the case construct, whose details are presented later. Some illustration:

– l.1 introduces the main anchor, u.
– l.3 introduces the induction hypothesis for the recursion variable f (using the lexicographic ordering on S-expressions).
– l.5 names the abstraction body as g', places the assumption on g, and stipulates that the content of x, org, is consistent.
– The lambda-body is an if-branch. Since its guard is effect-less $\mathtt{dom}(!x,g)$, no state change needs be considered. Hence l.7 simply adds to the previous condition the "true" condition $\mathtt{dom}(!x,g) \searrow \mathtt{t}$, as a new precondition.
– l.9 is transformed to l.10 by the following deduction, after adding $\mathsf{con}(org)$ by [Inv], noting the write effect is empty.

$$\begin{aligned} & !x = org \ \wedge \ \mathtt{get}(!x,g) \searrow g' \ \wedge \ \mathsf{con}(org) \\ & \equiv \ !x = org \ \wedge \ \mathtt{get}(!x,g) \searrow g' \ \wedge \ \mathsf{con}(!x) \\ & \equiv \ !x = org \cup \langle g, g' \rangle^* \ \wedge \ \mathsf{con}(!x) \end{aligned}$$

The last line uses $\langle g, g' \rangle \in !x$ and $\mathsf{con}(!x)$ imply $\langle g, g' \rangle^* \subseteq !x$. Since $S(g, org) = \emptyset$ we arrive at DG.

- From l.11, the else-part is reasoned. l.12 introduces g' as its anchor (which is what names the whole if-command), and asserts that the guard does not hold, in addition to what has been assumed before the if-branch. This else-part is a case construct, reasoned in the next few lines.
- l.13–18 is the case when g is a leaf. For the entailment from l.17 to l.18, we infer:

$$\begin{array}{l} \mathsf{con}(org) \wedge\, !x = org \cup \langle g, g' \rangle \wedge\, !g' = !g = \mathtt{inj}_1(n) \\ \supset \mathsf{con}(org) \wedge\, !x = org \cup \langle g, g' \rangle \wedge \mathsf{iso}(g, g') \wedge \forall z.(g \hookrightarrow z \supset z = g) \\ \supset \mathsf{con}(!x) \wedge !x = org \cup \langle g, g' \rangle^* \end{array}$$

  Above we used:

$$(z \hookrightarrow g' \ \wedge\ \mathsf{atom}(!g')) \ \supset\ z = g'$$

  which is easily inferred from the definition and the axiom for reachability.
- l.19–22 is the reasoning for the inductive case, which is going to be detailed in the next paragraph.
- The remaining lines concludes the reasoning using the proof rules for case, if, λ and recursion, each directly applied (with nominal use of consequence rule in each case).

Inductive Case. In the following we present the reasoning for the inductive case. By calling the procedure twice, it adds $\sharp$-fresh names one by one, reaching the eventual $\sharp$-fresh set. The asserted program fragment is given below, starting from the preconditions from the preceding inferences.

```
1   ${\mathsf{dom}(!x, g) \searrow \mathsf{f} \ \wedge\ !x = org \ \wedge\ \forall \tau' < \tau.\ \mathsf{dc}\,\tau'\,(f)}$
2   $g' : \{\exists \tau_{1,2}.(\tau = \langle \tau_1, \tau_2 \rangle \ \wedge\ \bigwedge_{i=1,2} \mathsf{dag}\,\tau_i\,(g_i))\}$  $(\mathsf{app})$
3       $m : \{\mathsf{T}\}$
4       NewEntry
5       $\{NE(m)\}@\emptyset$
6       $(n, g) : \{\cdots\}$
7           $n : \{\mathsf{T}\}$  $(\mathtt{inr})$
8               $m : \{\mathsf{T}\}$  $(\mathtt{pair\#})$
9                   $g'_1 : \{\mathsf{T}\}$
10                  f g1
11                  $\{\#^{-x} S(g'_1, org).(\mathsf{con}(!x) \wedge !x = org \cup \langle g_1, g'_1 \rangle^*)\}@x$
12                  $g'_2 : \{\mathsf{con}(!x) \ \wedge\ !x = org \cup \langle g_1, g'_1 \rangle^*\}$
13                  f g2
14                  $\{\#^{-x} S(g'_2, org').C_0\}@x$
15              $\{\#^{-x} S(m, org).\exists g'_{1,2}.(C_0 \wedge m = \langle g'_1, g'_2 \rangle)\}@x$
16          $\{\#^{-x} S(n, org).\exists g'_{1,2}.(C_0 \ \wedge\ n = \mathtt{inr}(\langle g'_1, g'_2 \rangle))\}@x$
17      $\{\#^{-x} S(n, org).(C_1 \ \wedge\ \{C_1\} m \bullet (n, g) = g' \{\#^{-x} g'.C_2\}@x)\}@x$
18  $\{\#^{-x} S(g', org).(\mathsf{con}(!x) \ \wedge\ !x = org \cup \langle g, g' \rangle^*)\}@x$
```

Above we set:

1. Two applications of f leads to:

$$C_0 \stackrel{\text{def}}{=} \mathsf{con}(!x) \wedge !x = org \cup \bigcup_{i=1,2} \langle g_i, g_i' \rangle^*$$

   which says that the table in x is consistent and it is the result of adding the isomorphic pairs reachable from $\langle g_i, g_i' \rangle$ $(i = 1,2)$.
2. After enclosing $\langle g_1', g_2' \rangle$ with injection, we reach:

$$C_1 \stackrel{\text{def}}{=} \exists g_{1,2}'.(C_0 \ \wedge \ n = \mathtt{inr}(\langle g_1', g_2' \rangle))$$

3. After further enclosing with a fresh reference, we finally reach:

$$C_2 \stackrel{\text{def}}{=} \mathsf{con}(!x) \wedge !x = org \cup \langle g, g' \rangle^*$$

All inferences are mechanical. Some illustrations:

– l.1 records all assumptions from the preceding inferences.
– l.2 introduces the anchor g' for the subprogram in this case branch (which is the name given to the whole case branch, as well as the encoding if-command). In the following inferences we consider the same assumption without $\exists \tau_{1,2}$.[7]
– l.3–5 is the inference for `NewEntry`, for which we records the result of the inference in §7.3. Since $NE(m)$ is stateless, once it is inferred, it can be used for all later inferences.
– l.6–8 introduces anchors, but no new assumptions.
– l.8–15 reasons for the pair of two applications using $[Pair\#]$.
– In l.8, the pairing is named as m.
– In l.9–11, we reason for fy_1, starting from $!x = org$, as well as using two stateless assertions, $\forall \tau' < \tau. \mathsf{dc}\tau'(f)$ and $\tau_1 \lneq \tau$. The reasoning is direct from the assumption.
– In l.12–14, we reason for fy_2. We use, following $[Pair\#]$,

$$!x = org \cup \langle g_1, g_1' \rangle^*,$$

  in addition to stateless $\forall \tau' < \tau. \mathsf{dc}\tau'(f)$ and $\tau_2 \lneq \tau$. Again the reasoning is mechanical from the assumption.
– l.15 summarises l.9–14 by applying $[Pair\#]$. Note $m = \langle g_1', g_2' \rangle$, for which we have:

$$m = \langle g_1', g_2' \rangle \ \supset \ (m \hookrightarrow z \ \equiv \ g_{1,2}' \hookrightarrow z)$$

  so that, setting:

$$G(g, org) \ \stackrel{\text{def}}{=} \ g \hookrightarrow z \ \wedge \ \mathtt{cod}(org, z) \searrow \mathsf{f}$$

  we obtain:

$$m = \langle g_1', g_2' \rangle \ \supset \ ((G(g_1', org) \vee G(g_2', org \cup \langle g_1, g_1' \rangle^*)) \equiv G(m, org))$$

  which is used to determine the $\#$-fresh names in l.15

[7] Since the final post-condition does not mention $\tau_{1,2}$, we can quantify the pre-condition again at the end, so that this does not lose generality.

– $l.16$ simply encloses m with the injection.
– $l.17$ combines $l.16$ and $NE(m)$, in the form usable for $[App\#]$.
– $l.18$ applies $[App\#]$ to $l.2$–17, obtaining DG. As in $l.15$, we use:

$$!g' = n \supset ((G(n, org) \vee g' = z) \equiv G(g', org))$$

by which we know $S(g', org)$ characterises the #-fresh names of the application.

This concludes the proof of the inductive case, hence the whole derivation for the required judgement for `dagCopy`.

7.4 Trees and Dages without *S*-Structures

We explain how we can verify the trees and dages without using *S*-structures. First we can simply define the assertions deleting τ from the assertions such as:

$$\begin{aligned}
\mathsf{tree}(u) &\equiv \exists y.\mathsf{atom}(!u, y) \quad \vee \\
&\qquad \exists m_1 m_2.(\mathsf{branch}(!u, m_1, m_2) \wedge m_1 \star m_2 \wedge \\
&\qquad\qquad \textstyle\bigwedge_{i=1,2}(\mathsf{tree}(m_i) \wedge u \# m_i)) \\
\mathsf{atom}(u, n) &\equiv u = \mathtt{inj}_1(n) \\
\mathsf{branch}(u, y_1, y_2) &\equiv u = \mathtt{inj}_2(\langle y_1, y_2\rangle)
\end{aligned}$$

Similarly for other assertions and predicates. For the derivation, we define the predicate $\mathsf{size}(x, n)$ which denotes *the tree named by x has a size n*. This is inductively defined as follows:

$$\begin{aligned}
\mathsf{size}(u, 1) &\equiv \exists n.\mathsf{atom}(u, n) \\
\mathsf{size}(u, n_1 + n_2 + 1) &\quad \exists m_1, m_2.(\mathsf{branch}(u, m_1, m_2) \wedge \mathsf{size}(m_1, n_1) \wedge \mathsf{size}(m_2, n_2))
\end{aligned}$$

The rest of reasoning is identical using the above ordering instead of that between *S*-structures $\tau < \tau'$.

The next example, verification of a graph copy, treats a general reasoning method which does not require any explicit syntax in the assertions.

7.5 Dynamic Mutable Data Structures (3): Graphs

To test how structured reasoning for dynamically created data structures can be reasoned about in the present logical theory, we have experimented with a further refinement of the copying algorithm, again found in [5], which works with any argument of *Tree*-type, including one with circular edges (note *Tree* allows circular linkage).

```
graphCopy $\stackrel{\text{def}}{=}$ $\lambda g$.new $x$ := $\emptyset$ in Main $g$
     Main $\stackrel{\text{def}}{=}$ $\mu f.\lambda g$.if dom($!x, g$) then get($!x, g$) else
                        case $!g$ of
                           inj$_1$($n$) : newEntry(inj$_1$($n$), $g$)
                           inj$_2$($y_1, y_2$) :
                              let $g'$ = newEntry(tmp, $g$)
                              in $g'$ := inj$_2$($\langle f y_1, f y_2\rangle$); $g'$
```

where $\mathtt{tmp} = \mathtt{inj}_1(0)$. The program is essentially identical with `dagCopy` 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. The assertion for `graphCopy` (named u) is even simpler than the one for `dagCopy`:

$$\mathsf{graphcopy}(u) \;\equiv\; \forall g^{Tree}.\{\mathsf{T}\}u \bullet g = g'\{\star g'.\mathsf{iso}(g,g')\}@\emptyset$$

Note we no longer need any requirement on g except for its type (convergence is guaranteed because our models cannot hold infinite graphs). $\mathsf{graphcopy}(u)$ entails $\mathsf{dagcopy}(u)$ hence also $\mathsf{treecopy}(u)$, so a program satisfying $\mathsf{graphcopy}(u)$ copies a tree just as `treeCopy` does and copies a dag just as `dagCopy` does.

Deriving $\{\mathsf{T}\}\,\mathtt{graphCopy} :_u \{\mathsf{graphcopy}(u)\}@\emptyset$ is almost identical with deriving the judgement for `dagCopy` (hence `treeCopy` in many places), except for the two points: $\mathsf{con}(!x)$ is not invariance any more (note the table can contain fresh nodes with temporary content) and we can no longer use S-expressions for well-founded ordering. For the latter, we can use, for example, the number of g-reachable nodes minus those in the domain of $!x$, which strictly decreases when the induction moves to subgraphs.For the former, a weaker consistency condition works, which roughly says:

(1) Before processing g, a certain isomorphism holds between nodes in the domain of $!x$ “before” g and the corresponding ones in the co-domain.
(2) After processing g, the isomorphism also holds between all nodes “under” g and the corresponding nodes in the co-domain.

Above “before” and “under” are calculated through the lexicographic ordering of the minimum paths from the original root (the use of the root is solely for defining these relations and does not violate modular reasoning). In the following we first list these intermediate assertions, then proceed to the reasoning. We leave the detailed derivations to Appendix B.

7.6 Higher-Order Mutable Data Structures

Finally we consider replication of trees/dags/graphs which may store values of arbitrary types, such as references and higher-order functions. In fact, each of the three algorithms above already works, as it is, even when we replace Nat with an arbitrary type. The result is the so-called *shallow copy*, where, while the nodes of two graphs are still disjoint, data at leaves can share references (a simplest case is when stored data are reference names). The assertion follows.

$$\mathsf{gcopy}(\alpha)(u) \;\equiv\; \forall g^{tree(\alpha)}.\{\mathsf{T}\}u \bullet g = g'\{\sharp S.\,\mathsf{iso}(g,g')\}@\emptyset$$

where $tree(\alpha) = \mu \mathrm{X}.(\mathsf{Ref}(\alpha + (\mathrm{X}\times\mathrm{X})))$; and $S \overset{\text{def}}{=} \{h \,|\, \exists p.\mathsf{path}(g',p,h)\}$ (for the notation $\sharp S$, see § sub:fresh:abbrev). Note the assertion uses the same isomorphism predicate. We conclude:

Proposition 7.1 *The following assertions are valid and implications are strict:* $\mathsf{gcopy}(\mathsf{Nat})(u) \equiv \mathsf{graphcopy}(u)$, $\mathsf{graphcopy}(u) \supset \mathsf{dagcopy}(u)$, $\mathsf{dagcopy}(u) \supset \mathsf{treecopy}(u)$ *and* $\mathsf{treecopy}(u) \supset \mathsf{treecopyS}\,(u)$.

7.7 Local State and Information Hiding

As final examples, we show how to reason about programs which use new reference generation for information hiding, including Landin's factorial discussed in the Introduction. The expected properties of such a program often crucially depend on maintaining certain invariants of hidden store throughout invocations. As simple examples, consider the following two short programs.

$$\begin{aligned}\texttt{profile} &\overset{\text{def}}{=} \texttt{let}\ x = \texttt{ref}(0)\ \texttt{in}\ \lambda y.(x := !x + 1; fy)\\ \texttt{compHide} &\overset{\text{def}}{=} \texttt{let}\ x = \texttt{ref}(7)\ \texttt{in}\ \lambda y.(y > !x)\end{aligned}$$

The first is from [40, p.104], the second a variant from [24]. In either program, an external program can never access x. For `profile`, the invariant on x's content is trivial since its visible behaviour does not depend on it: thus `profile` behaves precisely as f does. For `compHide`, the invariant is that the content of x stays 7, hence it behaves as $\lambda y.y > 7$.

To reason about programs with invariants of this kind, the following axiom, combined with [*ConsEval*] in Figure 3, is often useful. Below $\nu\#x.C$ means $\exists x.(x\#i \wedge C)$ with i fresh if x has a reference type or $\exists x.C$ otherwise.

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

where m is fresh and:

- $C \overset{\text{def}}{=} C_0 \wedge \{C_0 \wedge C_1\}u \bullet y = z\{C_2\} \wedge \forall y.\{C_0 \wedge \tilde{x}\#y\tilde{r}\tilde{w}\}u \bullet y = z\{C_0 \wedge \tilde{x}\#z\tilde{w}\}@\tilde{w}\tilde{x}$ with $x_i \notin \mathsf{fv}(C_1) \cup \mathsf{fv}(C_2)$ and $\langle !\tilde{x}\rangle C_0 \equiv \mathsf{T}$.
- $C' \overset{\text{def}}{=} \{C_1\}u \bullet y = z\{C_2\}@\tilde{w}$.

(AIH) intuitively says:

> *Assume C_0 is an invarinat of the content of $\tilde{x}$; if C_0 holds and if $\tilde{x}$ are not reachable from the argument y or from the initial state and read data r_i and write data w_j ($\tilde{x}\#y\tilde{r}\tilde{w}$),[8] u applied to y never exports $\tilde{x}$ to the outside ($\tilde{x}\#z\tilde{w}$). Then if $\tilde{x}$ are fresh, we can ignore C_0 from the specification.*

Freshness and hiding of $\tilde{x}$ are essential: if they are not hidden, their content may be modified by external programs, destroying the invariant C_0.

The proof of validity of (AIH) uses Proposition 4.24 to show that, in any possible model, $\tilde{x}$ can never be touched by external programs except by invoking the function (say V) denoted by u, thanks to a partition of its store into two disjoint parts, one for the name closure of V, and another containing x. Thus C_0, which solely relies on the content of $\tilde{x}$, stays invariant in any future state.

$\langle !x\rangle C_0 \equiv \mathsf{T}$ means C_0 asserts about only the content of x; e.g. $\langle !x\rangle !x = 1 \equiv \langle !x\rangle(x = y \supset !y = 1) \equiv \mathsf{T}$, but $\langle !x\rangle(x = y \wedge !y = 1) \not\equiv \mathsf{T}$ and $\langle !x\rangle y \hookrightarrow x \not\equiv \mathsf{T}$.

[8] We can prove the same conclusion without $\tilde{x}\#\tilde{r}$ in the precondition of this evaluation formula, but this widens the applicability of this axiom (hence having $\tilde{x}\#\tilde{r}$ is more general).

Proof of Validity of (AIH)**.** We first show the following useful lemma.

Lemma 7.2 1. (narrowing) $\mathcal{M} \models C$ and $l \notin \mathsf{fl}(C)$ imply $(\nu l)\mathcal{M} \models C$

2. (scope opening) $((\nu l)\mathcal{M})[u\!:\!N] \equiv (\nu l)(\mathcal{M}[u\!:\!N])$ with $l \notin \mathsf{fl}(N)$.
3. Assume $\mathcal{M}_0 \models E$ and $\mathcal{M}_0[u\!:\!m()] \Downarrow \mathcal{M}$ with $u, m \notin \mathsf{fv}(E)$. Suppose $\mathcal{M} \models \exists x.(x\#i \wedge C)$ with i fresh. Then there exist ξ, σ, l, V such that $\mathcal{M} \approx (\nu l)\mathcal{M}'$ with $\mathcal{M}' = (\nu \tilde{l})(\xi, \sigma \cdot [l \mapsto V])$ and $l \notin \mathsf{fl}(\xi, \sigma)$; and $\mathcal{M}'[x\!:\!l] \models C$.

PROOF: The proofs of (1,2) are by definition, while (3) is direct from Proposition 4.24. □

Now let us define, with the conditions in (AHI):

$$\begin{array}{ll} G \equiv \nu\#x.G_0 & G_0 \equiv C_0 \wedge G_1 \wedge G_2 \\ G_1 \equiv \{C_0 \wedge C_1\}u \bullet y = z\{C_2\} & G_2 \equiv \forall y.\{C_0 \wedge \tilde{x}\#y\tilde{r}\tilde{w}\}u \bullet y = z\{C_0 \wedge \tilde{x}\#z\tilde{w}\}@\tilde{w}\tilde{x} \end{array}$$

W.o.l.g. we assume all vectors are unary, i.e. $\tilde{r} = r$, $\tilde{w} = w$ and $\tilde{x} = x$ (with x reference). By Lemma 7.2 (3), we know there exists $\mathcal{M}$ such that $\mathcal{M} = (\nu \tilde{l})(\xi \cdot [x:l], \sigma \cdot [l \mapsto V]) \models G$ with $l \notin \mathsf{fl}(\xi, \sigma)$. Then our aim is to prove:

$$\mathcal{M} \models G \quad \text{implies} \quad \mathcal{M} \models C' \tag{7.8}$$

Then $\mathcal{M} \models \{C_1\}u \bullet y = z\{C_2\}@\tilde{w}$ means, by definition,

$$\forall N.(\mathcal{M}[f\!:\!N] \Downarrow \mathcal{M}_a \models C_1 \quad \text{with } f \text{ fresh} \quad \supset \quad \mathcal{M}_a[z:uy] \Downarrow \mathcal{M}_b \vdash C_2)$$

Then by Lemma 7.2 (1,2), let $\mathcal{M}_0 = (\nu \tilde{l})(\xi, \sigma \cdot [l \mapsto V])$. Then we have:

$$\mathcal{M}_0[f:L] \Downarrow \mathcal{M}_{a0} \models C_1 \quad \text{with} \quad \mathcal{M}_a \approx (\nu l)\mathcal{M}_{a0}$$

We now show such $\mathcal{M}_{a0}$ always satisfy C_0, i.e.,

$$\mathcal{M}_{a0} \models C_0 \tag{7.9}$$

If we can prove (7.9) above, then we have $\mathcal{M}_{a0} \models C_0 \wedge C_1$. Then by $A \wedge B \supset A$, we have $\mathcal{M}_0 \models G_1$. Hence this concludes our aim to prove (7.8).

To prove (7.9), we use the assumption $\mathcal{M}_0 \models C_0 \wedge G_2$. By $x\#yrw$ and $x\#zw$, we know only U can touch l; and even so, the result of the application of uy in L does not affect the result z and effect variable w. Set $L = uy$. Then for all y, if $\mathcal{M}_0[x:l][f:uy] \Rightarrow \mathcal{M}_{0c}$, we have $\mathcal{M}_{0c} \models C_0$ and

$$\mathcal{M}_{0c} \;\equiv\; (\nu \tilde{l}\tilde{l}')(\xi \cdot u : U \cdot f : V_f \cdot x : l,\ \sigma_1' \uplus [l \mapsto V_x'] \cdot \sigma_2')$$

for some V_f, V_x' with $\mathsf{dom}(\sigma_2') \subset \{\tilde{l}'\}$ and $l \notin \mathsf{fv}(\sigma_1')$. Hence we have $\mathcal{M}_{0c} \models x\#y$ by definition of reachability. Similarly, by $\mathcal{M}_0 \models G_2$, for any L which contains an application of the form of uy, $\mathcal{M}_{0c}$ can satisfy C_0, by using $\mathcal{M}_{0c} \models x\#y \wedge C_0$. Hence we can derive (7.9), as required.

Profile. `profile` and `compHide` can be easily reasoned using (AIH), together with [*ConsEval*] in Figure 3. We start from `profile`. For simplicity, we assume `profile` $f\,y$ terminates. Our aim is to obtain:

$$\{\{C\}\,f\bullet y = z\{C'\}@\tilde{w} \wedge \{\mathsf{T}\}\,f\bullet y\{\mathsf{T}\}@\tilde{w}\}\ \texttt{profile} :_u \{\{C\}u\bullet y = z\{C'\}@\tilde{w}\}, \quad (7.10)$$

The above Hoare triple said that `profile` behaves just as f behaves, as expected.

First we derive:

$$\begin{array}{lll} E & = \{\mathsf{T}\}\,f\bullet y = z\{\mathsf{T}\}@\tilde{w} & \\ \supset E_0 & = \{x\#\,fy\tilde{r}\tilde{w}\}\,f\bullet y = z\{\mathsf{T}\}@\tilde{w}x & \text{Axiom (e8) in [19]} \\ \supset E' & = \{x\#\,fy\tilde{r}\tilde{w}\}\,f\bullet y = z\{x\#z\tilde{w}\}@\tilde{w}x & \text{Axiom 3 of Proposition 6.2} \end{array}$$

Let $G = \{C\}\,f\bullet y = z\{C'\}@\tilde{w}$. Using the above entailment, we derive:

$$\begin{array}{ll} 1.\{E'\wedge x\#\,fy\tilde{r}\tilde{w}\}\,f\,y :_z \{x\#z\tilde{w}\}@\tilde{w} & \text{AppSimple} \\ \hline 2.\{E\wedge x\#\,fy\tilde{r}\tilde{w}\}\,f\,y :_z \{x\#y\tilde{w}\}@\tilde{w} & \text{Conseq} \\ \hline 3.\{E\wedge x\#\,fy\tilde{r}\tilde{w}\}x := !x+1\{E\wedge x\#\,fy\tilde{r}\tilde{w}\}@x & \text{Inv\#} \\ \hline 4.\{E\wedge x\#\,fy\tilde{r}\tilde{w}\}x := !x+1;\ f\,y :_z \{E\wedge x\#y\tilde{w}\}@x\tilde{w} & \text{Seq} \\ \hline 5.\{E\}\lambda y.(x := !x+1;\ f\,y) :_z \{\forall y.\{x\#\,fy\tilde{r}\}u\bullet y = z\{x\#z\tilde{w}\}@x\tilde{w}\} & \text{Abs} \\ \hline 6.\{x\#i\wedge G\wedge E\}\lambda y.(x := !x+1;\ f\,y) :_u \{x\#i\wedge G\wedge\forall y.\{x\#\,fyr\}u\bullet y = z\{x\#z\tilde{w}\}@x\tilde{w}\}@\emptyset & \text{Invariance} \\ \hline 7.\{E\}\texttt{profile} :_u \{\nu\,\#x.(G\wedge E')@x\tilde{w}\}@\emptyset & \text{New} \end{array}$$

At Line 1, we use [*AppSimple*] in Figure 7.

$$[AppSimple]\,\frac{C\supset\{C\}\,e\bullet e' = u\{C'\}@\tilde{e}}{\{C\}\ ee' :_u \{C'\}@\tilde{e}}$$

From Line 1 to Line 2, we use the standard consequence rule. Line 3 is derived by the following invariant rule for the assignment.

$$[Inv\#]\,\frac{\{C\}M :_u \{C'\}@\tilde{x} \quad j_k \text{ fresh}}{\{C\wedge\tilde{x}\#\tilde{j}\}M :_u \{C'\wedge\tilde{x}\#\tilde{j}\}@\tilde{x}}$$

The proof of the validity of this rule is easy by Proposition 4.24 in § 4.5. Note that a value has an empty write effect set so that we can always apply the above rule when $M = V$. Also note that the condition of the write set is essential; for example, if $M = y := x$, then we can not apply the above rule since after running this assignment, x is reachable from y. From Line 3 to Line 4, we use [*Seq*] rule, the standard sequential composition rule. From Line 5 to Line 6, we use [*Promote*] appeared in Figure 6; and finally from Line 6 to Line 7, we use [*New*].

$$[Promote]\,\frac{\{C\}\ V :_u \{C'\}@\emptyset}{\{C\wedge C_0\}\ V :_u \{C'\wedge C_0\}@\emptyset}$$

Now we can apply [*ConsEval*] via (AIH) (taking the trivial invariant $C_0 \stackrel{\text{def}}{=} !x\geq 0 \equiv \mathsf{T}$), obtaining (7.10) as desired.

CompHide. For `compHide`, a direct compositional reasoning leads to $\{\mathsf{T}\}\texttt{compHide} :_u \{G\}$ where G is the assertion:

$$\nu\#x.(!x = 7 \wedge \forall n.\{!x = 7\}u \bullet n = z\{z = (n \geq 7) \wedge !x = 7\}@\emptyset)$$

The detailed derivation is given as follows:

$$1.\{\mathsf{T}\}\lambda y.y \geq 7\{\forall n.\{!x = 7\}u \bullet n = z\{z = (n \geq 7) \wedge !x = 7\}@\emptyset\}@\emptyset$$

$$2.\{\mathsf{T}\}7 :_m \{m = 7\}@\emptyset$$

$$3.\{x\#i \wedge !x = 7\}\lambda y.y \geq 7\{\forall n.\{!x = 7\}u \bullet n = z\{z = (n \geq 7) \wedge !x = 7\}@\emptyset \wedge x\#i \wedge !x = 7\}@\emptyset$$

$$4.\{\mathsf{T}\}\texttt{compHide} :_u \{\nu\#x.\forall n.\{!x = 7\}u \bullet n = z\{z = (n \geq 7) \wedge !x = 7\}@\emptyset\}@\emptyset$$

From Line 1 to 3, we again use [*Promote*] (this step is identical with Line 6 in `profile`). We can now apply [*ConsEval*] with (AIH), setting $C_0 \equiv !x = 7$ as the invariant, and noting $x\#n \equiv x\#z \equiv \mathsf{T}$ by Axiom (2-1) in Prop. 6.2, finally reaching

$$\forall n.u \bullet n \searrow (n \geq 7)$$

as the postcondition.

Safe Factorial. We conclude this section with a more substantial use of (AIH), taking Landin's factorial from the Introduction.

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

In [19], we have derived the following judgement.

$$\{\mathsf{T}\}\texttt{circFact} :_u \{\exists g.(\forall n.Fact(g, !x, n, x) \wedge\ !x = g)\} \tag{7.11}$$

where

$$Fact(g, u, n, x) \stackrel{\text{def}}{=} \{!x = g\}u \bullet n = z\{z = n! \wedge\ !x = g\}@\emptyset \tag{7.12}$$

The postcondition is logical equivalent to:

$$Fact(u, n, x) \stackrel{\text{def}}{=} \exists g.(Fact(g, u, n, x)\ \wedge\ u = g) \tag{7.13}$$
$$\equiv \{!x = u\}u \bullet n = z\{z = n! \wedge\ !x = u\}@\emptyset \wedge\ !x = u \tag{7.14}$$

The judgement (7.11) 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.*

Our purpose is to show that `safeFact` named u behaves as a pure factorial function, i.e. it satisfies the assertion $\forall n.\{\mathsf{T}\}u \bullet n = z\{z = n!\}@\emptyset$.

We first derive $Fact(u,n,x)$ for `circFact`, which is much simpler than one in (7.11). For the derivation, let:

$$C(g,!x,j) \quad \stackrel{\text{def}}{=} Fact(g,!x,j,x) \ \wedge \ !x = g.$$

We also set, for brevity:

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

We infer:

1. $\{(y \geq 1 \supset C(g,!x,y-1)) \ \wedge \ y=0\}\ 1 :_z \{z = y! \wedge \ !x = g\}@\emptyset$ (Simple)

---

2. $\{(y \geq 1 \supset C(g,!x,y-1) \ \wedge \ y \geq 1\}\ y \times (!x)(y-1) :_z \{z = y! \wedge \ !x = g\}@\emptyset$ (Simple, AppSimple)

---

3. $\{y \geq 1 \supset C(g,!x,y-1)\}$
   $\mathtt{if}\ y=0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1) :_z \{z = y! \wedge \ !x = g\}@\emptyset$ (IfH)

---

4. $\{\top\}\ \lambda y.\mathtt{if}\ y=0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1) :_u$
   $\{\ \forall g, y \geq 1.\{C(g,!x,y-1)\}u \bullet y = z\ \{z = y! \wedge \ !x = g\}\ \}@\emptyset$ (Abs, $\forall$)

---

5. $\{\top\}\ M :_u \{\ \forall g, y \geq 1.(Fact(g,!x,y-1,x) \supset Fact(g,u,y,x))\ \}@\emptyset$ (Conseq)

---

6. $\{\top\}\ x := M\{\ \forall g, y \geq 1(Fact(g,!x,y-1,x) \supset Fact(g,!x,y,x)\ \}@x$ (Assign)

---

7. $\{\top\}\ x := M\{\ \forall g, y \geq 1.((Fact(g,!x,y-1,x) \wedge !x = g) \supset (Fact(g,!x,y,x) \wedge !x = g))\ \}@x$ (Conseq)

---

8. $\{\top\}\ x := M\{\ \forall y \geq 1.(\exists g(Fact(g,!x,y-1,x) \wedge !x = g) \supset \exists g(Fact(g,!x,y,x) \wedge !x = g))\ \}@x$ (Conseq)

---

9. $\{\top\}\ x := M\{\ \forall y \geq 1.Fact(!x,y,x)\ \}@x$ (Conseq)

From Line 4 to Line 5, we used the following axiom for evaluation formulae, (e5) in [19] with $A = Fact(g,!x,y-1,x)$, $B = !x = g$ and $C = z = y! \wedge !x = g$.

$$\{A \wedge B\}e \bullet e' = z\{C\} \ \equiv \ (A \supset \{B\}e \bullet e' = z\{C\})$$

From Line 8 to Line 9, we use the following standard entailment.

$$\forall x.(A \supset B) \ \supset \ \exists x.A \supset \exists x.B$$

Finally we show the main derivation for `safeFact`. A derivation follows.

1.$\{\top\}\lambda y.y :_m \{\top\}@\emptyset$

2.$\{\top\}\mathtt{circFact};\,!x :_u \{Fact(u,n,x)\}@x$

3.$\{x\#i\}\mathtt{circFact};\,!x :_u \{x\#i \ \wedge \ Fact(u,n,x)\}@x$

4.$\{\top\}\mathtt{safeFact} :_u \{\nu\#x.Fact(u,n,x)\}@\emptyset$

5.$\{\top\}\mathtt{safeFact} :_u \{\forall n.\{\top\}u \bullet n = z\{z = n!\}@\emptyset\}@\emptyset$

Line 1 is immediate. Line 2 is (7.11) with [*Deref*] and [*Seq*]. From Line 2 to Line 3, we use [*Inv#*]. Line 4 is direct from Lines 1, 3 and [*New*]. Finally we arrive at Line 5

$$\forall n.u \bullet n \searrow n!$$

by [*ConsEval*] together with (AIH), letting: $E = C_1 = \top$, $\tilde{r} = \tilde{w} = \emptyset$, $C_0 = !x = g$ and $C_2 = z = n!$, and noting $x\#n \equiv x\#z \equiv \top$ by Axiom (2-1) in Prop. 6.2.

Safe Mutual Recursion. Finally we consider the advanced recursion by mutual circular references. The program demonstrates the power of the higher-order functions. Consider the following two assignments.

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

It is easy to see that, after these two assignments, $(!x)n$ returns the truth if n is odd, while $(!y)m$ returns the truth if n is even. This situation may be informally described thus:

> *x stores a procedure which computes whether its argument is odd or not using a procedure stored in y; y stores a procedure which computes whether its argument is even or not using a procedure stored in x*

Note an inherent circularity of this description. Apart from the local state, the first question is how we can logically describe such a program specification, and how can we derive it compositionally.

In the presence of local state, we can hide x, y to avoid unexpected interference as we have done for `safeFact`;

$$\begin{aligned}
\texttt{safeMutualOdd} &\stackrel{\text{def}}{=} \texttt{let}\ x, y = \texttt{ref}(\lambda n.\texttt{t})\ \texttt{in}\ (\texttt{mutualCheck}; !x) \\
\texttt{safeMutualEven} &\stackrel{\text{def}}{=} \texttt{let}\ x, y = \texttt{ref}(\lambda n.\texttt{t})\ \texttt{in}\ (\texttt{mutualCheck}; !y)
\end{aligned}$$

(above $\lambda n.\texttt{t}$ can be any initialising value). The program evaluates to a function which checks the even of the number: x, y are now hidden and inaccessible from the outside, so that the program behaves as the pure functions, e.g., $\texttt{safeMutualOdd}(3)$ returns the truth, while $\texttt{safeMutualOdd}(4)$ returns the false.

We first challenge the derivation of `mutualCheck`

$$\{\top\}\texttt{mutualCheck} :_u \{\forall n.\mathit{IsOddEven}(n, xy)\} \tag{7.15}$$

where:

$$\begin{aligned}
\mathit{IsOddEven}(n, xy) &\stackrel{\text{def}}{=} \exists gh.(\mathit{IsOdd}(!x, gh, n, xy) \wedge \mathit{IsEven}(!y, gh, n, xy) \wedge !x = g \wedge !y = h) \\
\mathit{IsOdd}(u, gh, n, xy) &\stackrel{\text{def}}{=} \{!x = g \wedge !y = h\}u \bullet n = z\{z = \mathit{Odd}(n) \wedge !x = g \wedge !y = h\}@xy \\
&\qquad \wedge !x = g \wedge !y = h \\
\mathit{IsEven}(u, gh, n, xy) &\stackrel{\text{def}}{=} \{!x = g \wedge !y = h\}u \bullet n = z\{z = \mathit{Even}(n) \wedge !x = g \wedge !y = h\}@xy \\
&\qquad \wedge !x = g \wedge !y = h \\
\mathit{Odd}(n) &\stackrel{\text{def}}{=} \exists x.(n = 2 \times x + 1) \qquad \mathit{Even}(n) \stackrel{\text{def}}{=} \exists x.(n = 2 \times x)
\end{aligned}$$

Our final aim is to derive the following "pure" assertion by (AIH).

$$\{\top\}\texttt{safeMutualOdd} :_u \{\forall n.\{\top\}u \bullet n = z\{z = \mathit{Odd}(n)\}@\emptyset\} \tag{7.16}$$

$$\{\top\}\texttt{safeMutualEven} :_u \{\forall n.\{\top\}u \bullet n = z\{z = \mathit{Even}(n)\}@\emptyset\} \tag{7.17}$$

Let us define:

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

The derivation of `mutualCheck` is similar with `circFact`.

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

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

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

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

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

6. $\{\top\}\ x := M_x\{\ \forall gh, n \geq 1.(\mathit{IsEven}(!y, gh, n-1, xy) \supset \mathit{IsOdd}(!x, gh, n, xy))\}@x$ (Assign)

7. $\{\top\}\ y := M_y\{\ \forall gh, n \geq 1.(\mathit{IsOdd}(!x, gh, n-1, xy) \supset \mathit{IsEven}(!y, gh, n, xy))\}@y$ (Conseq)

8. $\{\top\}$ `mutualCheck`
   $\{\forall gh.n \geq 1.((\mathit{IsEven}(!y, gh, n-1, xy) \wedge \mathit{IsOdd}(!x, gh, n-1, xy)) \supset$
   $(\mathit{IsEven}(!y, gh, n, xy) \wedge \mathit{IsOdd}(!x, gh, n, xy))\ \}@xy$ ($\wedge$-Post)

9. $\{\top\}$ `mutualCheck`
   $\{\forall n \geq 1.(\exists gh.(\mathit{IsEven}(!y, gh, n-1, xy) \wedge \mathit{IsOdd}(!x, gh, n-1, xy)) \supset$
   $\exists gh.(\mathit{IsEven}(!y, gh, n, xy) \wedge \mathit{IsOdd}(!x, gh, n, xy))\}@xy$ (Conseq)

10. $\{\top\}$ `mutualCheck`$\{\forall n.\mathit{IsOddEven}(n, xy)\}@xy$

Now we derive (7.16):

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

2.$\{\top\}$`mutualCheck;` $!y :_u$
$\{\exists gh.(!x = g \wedge !x = h\ \wedge\ \forall n.\mathit{IsOdd}(g, gh, n, xy)\ \wedge\ \forall n.\mathit{IsEven}(u, gh, n, xy))\}@xy$

3.$\{\top\}$`mutualCheck;` $!y :_u$
$\{\exists gh.(C_0\ \wedge\ \forall n.\{C_0\}u \bullet n = z\{z = \mathit{Even}(n) \wedge C_0\}@xy)\}@xy$

4.$\{xy\#ij\}$`mutualCheck;` $!y :_u \{\exists gh.(xy\#ij\ \wedge\ C_0\ \wedge\ \forall n.\{C_0\}u \bullet n = z\{z = \mathit{Even}(n)\ \wedge\ C_0\}@xy)\}@xy$

5.$\{\top\}$`safeMutualEven` $:_u \{\nu\#xygh.(C_0\ \wedge\ \forall n.\{C_0\}u \bullet n = z\{z = \mathit{Even}(n)\ \wedge\ C_0\}@xy)\}@\emptyset$

6.$\{\top\}$`safeMutualEven` $:_u \{\forall n.\{\top\}u \bullet n = z\{z = \mathit{Even}(n)\}@\emptyset)\}@\emptyset$ (AIH)

In Line 3, we let $C_0 = !x = g \wedge !y = h\ \wedge\ \forall n.\mathit{IsOdd}(g, gh, n, xy)$ and use the axiom

$$\{C\}e \bullet e' = z\{C'\}@\tilde{e}\ \supset\ \{C\ \wedge\ A\}e \bullet e' = z\{C'\ \wedge\ A\}@\tilde{e}$$

where A is stateless formula and $A = \forall n.\mathit{IsOdd}(g, n, gh, xy)$. Line 4 is again by $[\mathit{Inv} - \#]$. The final line is by an application of (AIH) noting $\langle !xy \rangle C_0 \equiv \top$ and $xy\#n \equiv \top$, $\tilde{w} = \emptyset$ and $xy\#z \equiv \top$. Hence we achieve the main result.

8 Discussions

8.1 Summary

This paper presented a program logic for imperative higher-order functions with new reference generation. The target languages of our preceding logics [3, 17–19] do not include local state: none of the examples treated in the present paper can be asserted and inferred in these logics. The new axioms on reachability involving general data types as well as higher-order functions are introduced and are shown to be effective for reasoning about programming examples which are known to be hard in the literature [24, 32, 33, 40]

8.2 Related Work.

Below we discuss related works, mainly focussing on local state and freshness. Comparisons w.r.t. other elements (e.g. higher-order functions, aliasing, polymorphism) are relegated to [3, 17–19].

Original Local Variable Rule by Hoare and Wirth. To our knowledge, the first work which introduced the proof rule for local variable is Hoare and Wirth's [13]. The rule is so-called for stack variables, i.e. those local variables which are never exported beyond their original scope. Since aliasing is not considered, the rule has the following simple shape.

$$\frac{\{C[e/x]\}P\{C'\}}{\{C\}\mathtt{newvar}\ x := e \text{ in } P\{C'\}}$$

This can be translated into the following rule in the context of the present logic:

$$\frac{\{C\}\ M :_m \{C_0\} \quad \{C_0[!x/m]\}\ N :_u \{C'\} \quad x \notin \mathsf{fv}(C')}{\{C\}\ \mathtt{new}\ x\ := M\ \mathtt{in}\ N :_u \{C'\}}$$

which is close to, and weaker than, [*NewVar*] in Figure 4, Page 26 (hence is sound). It is pleasant to see that, at the level of proof rules, the only essential difference between the original rule for local variables and the present one lies in the addition of the disjointness condition through the #-predicate. At the semantic level, incorporation of local state in the present logic leads to a very different model of assertions and judgement, as we have seen in Section 4.

Development Framework. The present work proposes a compositional program logic for a core part of ML [2, 26]. *Extended ML* [38] 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* [42] 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 [42]. 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.

Observational Congruence and Completeness. Stark and Pitts [32, 33, 40] develop powerful reasoning principles for behavioural equivalences on higher-order functions using operationally based techniques. Koutavas and Wand [20] recently showed a fully abstract bisimulation technique for the untyped version of the language treated in the present work, and applied the techniques to several non-trivial reasoning examples. Though in quite different settings from program logics, these works elucidate subtleties in reasoning on higher-order functions with local state, demonstrated through many subtle examples (such as those due to Meyer-Sieber [24]). Observational completeness theorem precisely relates their reasoning method to our logic, suggesting a rich technical interplay.

Higher-Order Logic. Several recent works present mechanisation of Hoare logics in higher-order logics, cf. [8, 22, 29, 41]. While these works do discuss some aspects of local state such as pointer-based data structures, they do not (aim to) offer a direct logical treatment of either ML-like general references or their combination with higher-order functions.

Reachability Predicate and Logics for Dynamic Data Structures. Assertion-based reasoning methods for dynamically generated mutable data structures have been studied from early days of program logics [15], cf. [3, §10]. Nelson would be the first to use a notion of reachability in this context [28], for reasoning about linearly linked lists via predicate transformers. His predicate is tailored for this particular data structure, and can be represented by the first-order part of our reachability predicate. Neither general mutable data types, higher-order functions nor ML-like new reference generation, which are the central elements of the present work, is treated in his work.

A basic difference between these preceding works and the present logic may be that we take an analytical approach in which central elements of sequential (higher-order) behaviour are distilled and stratified, to be respectively given their logical articulations. For example, aliasing and new reference generation are given separate treatments: the former is treated using content quantification while the latter is captured by freshness predicates. This leads to a uniform understanding of involved semantic structures through a logical means. Let us discuss this point taking a concrete example. The following is Burstall's example for mutable list cells, already given in Section 2.1, Example 2.2, Page 6.

$$L \quad \stackrel{\text{def}}{=} \quad x := \mathsf{cons}(0, \mathsf{nil})\ ;\ y := \mathsf{cons}(2, !x)\ ;\ \mathsf{car}(!x) := 1$$

Assuming the precondition $\{x \neq y\}$, the first command leads to:

$$\nu h, l.(!x = \langle h, l\rangle \wedge\ !h = 0 \wedge\ !l = \mathsf{nil}) \tag{8.1}$$

Then we obtain:

$$\nu h, l, h', l'.(!y = \langle h', l'\rangle \wedge\ !h' = 2 \wedge\ !l' = x \wedge\ !x = \langle h, l\rangle \wedge\ !h = 0 \wedge\ !l = \mathsf{nil}) \tag{8.2}$$

Finally, after the third line, we get:

$$\nu h, l, h', l'.(!y = \langle h', l'\rangle \wedge\ !h' = 2 \wedge\ !l' = x \wedge\ !x = \langle h, l\rangle \wedge\ !h = 1 \wedge\ !l = \mathsf{nil}) \tag{8.3}$$

Note $\nu hlh'l'.C$ entails, by definition, these new references are mutually distinct. The reasoning above is analytical, but it has the merit, in comparison with the methods used in the above cited preceding works in that we only have to apply a general proof rule to obtain these assertions.

The analytical presentation given above can be rendered into more efficient and readable format (both for assertions and for reasoning) by introducing short-hand notations without changing semantics. For example, we may write $ref(e)$ for a new reference which contains e as its content (so, for example, $x = \langle ref(0), ref(1)\rangle$ stands for $\nu ij.(x = \langle i, j\rangle \wedge !i = 0 \wedge !j = 1)$). Then the first assertion becomes:

$$!x = \langle ref(0), ref(\mathsf{nil})\rangle \tag{8.4}$$

The second one:

$$!y = \langle ref(2), ref(x)\rangle \wedge !x = \langle ref(0), ref(\mathsf{nil})\rangle \tag{8.5}$$

And the third assertion:

$$!y = \langle ref(2), ref(x)\rangle \wedge !x = \langle ref(1), ref(\mathsf{nil})\rangle \tag{8.6}$$

which are much more tractable than the original ones (we may further simplify the structures using the constructor for a mutable list). Derivation of these simplified assertions can use corresponding proof rules. We believe studying various reasoning methods proposed in this domain in the light of the present logic would lead to enrichment of reasoning methodologies for mutable data structures on a uniform basis. Among the existing threads of work, we later discuss how the present framework compares and interacts with the reasoning method based on separating connectives by Reynolds, O'Hearn, Bornat and others taking concrete examples.

Separation Logic. Reynolds, O'Hearn and others [5, 30, 37] propose, and experiment with, *separating conjunction* for Hoare logics of aliasing and dynamically generated data structures. As Reynolds shows [37], their conjunction is effective when data structures do not have non-trivial sharing, as in trees. When sharing is non-trivial, both assertions and reasoning tend to become highly complex in their approach: practical treatment of data with complex sharing is left open in [5]. One of the main issues lies in the need to encode the whole of a (concrete) target data structure as a formula, demanded by the use of separating conjunction itself [5]. Not only does this mean the size of concrete formulae grows in proportion to that of treated data structures, but also the construction of the encoding itself becomes onerous for e.g. dags and graphs. In contrast, our approach allows concise description of such notions as isomorphism without such encodings, as shown in §7. Proposition 7.1 also shows that assertions based on reachability offer accurate specifications entailing separation. The present logic also differs in that it can treat (stored) higher-order functions and general data types such as products, sums and polymorphism. Results similar to observational completeness may not have been reported for their logics.

Recent work by Birkedal et al. [4] presents a type system for Algol whose types are constructed from formulae of Separation Logic and whose typing is performed by logical entailment, formalised by categorical semantics. Their type system does not allow compositional logical reasoning for higher-order constructs, nor does it offer axioms for calculating entailment, which is at the heart of Hoare logic's practical use. Their higher-order frame rule captures only static compositionality, hence cannot reason about dynamically allocated data structures we studied in Section 7.

Logics for Fresh Names. Freshness of names is recently studied from the viewpoint of formalising binding relations by Pitts and Gabbay [9, 34]; and Miller and Tiu [25]. In the work by Pitts and Gabbay, 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 И and the freshness predicate #. The latter work by Miler and Tiu [25] is 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 new quantifier ∇ and develop the corresponding sequent calculus of Generic Judgements. While these works are not done in the context of Hoare logic, their logical machinery may well be usable in the present context, for example in refinement of axiomatisation of reachability including function types (which is one of the important future topics).

8.3 Future Work.

While equational reasoning for higher-order functions with local state have been studied in the literature (as discussed above), ours would be one of the initial trials to articulate this realm logically in Hoare-like assertion methods. In § 6 and 7, we have shown how axioms for reachability play a central role in non-trivial reasoning with local state. Clearly logical transformations needed to reach desired judgement in the present logic (cf. § 7) demand syntactic axioms which go much beyond number theory: some of the useful axioms for higher-order functions and aliasing are studied in [3, 17, 19], while those involving fresh names and reachability predicate are discussed in the present paper. A further study on axiom systems, their logical status and their practical use combined with existing tools [7] would be an interesting future research topic.

Several recent proposals of safe low-level languages are inspired by ML, including [11, 27, 39]. Since higher-order functions and local state are their central elements, it is interesting to extend the present logic to these languages. Another related interest is validation of library functions such as C's `malloc` which implement new reference generation, where the properties of `new` should be derivable in a logic rather than stipulated.

References

1. Standard ML home page. http://www.smlnj.org.
2. The Caml home page. http://caml.inria.fr.
3. Martin Berger, Kohei Honda, and Nobuko 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.
4. Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. LICS*, pages 260–269, 2005.
5. Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation and aliasing. In *Workshop SPACE*, 2004.
6. R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7, 1972.
7. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. Jean-Christophe Filliatre. Verification of non-functional programs using interpretations in type theory. *JFP*, 13(4):709–745, 2003.
9. Murdoch Gabbay and Andrew Pitts. A New Approach to Abstract Syntax Involving Binders. In *Proc. LICS '99*, pages 214–224, 1999.
10. Irene Greif and Albert R. Meyer. Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer. *ACM Trans. Program. Lang. Syst.*, 3(4), 1981.
11. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Management in Cyclone. In *PLDI'02*. ACM, 2002.
12. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
13. Hoare and Wirth. Axiomatic semantics of pascal. *ACM Trans. Program. Lang. Syst.*, 1(2):226–244, 1979.
14. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
15. Tony Hoare. Notes on data structuring. *Structured Programming*, pages 83–174, 1972.
16. Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
17. Kohei Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.
18. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202, 2004.
19. Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*, pages 270–279, 2005. Full version is available at: www.dcs.qmul.ac.uk/˜kohei/logics.
20. Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. POPL*, 2006.
21. Peter Landin. A correspondence between algol 60 and church's lambda-notation. *Comm. ACM*, 8:2, 1965.
22. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 159:200–227, May 2005.
23. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
24. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *POPL'88*, 1988.
25. Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, to appear.
26. Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

27. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
28. Greg Nelson. Verifying reachability invariants of linked structures. In *POPL '83*, pages 38–47. ACM Press, 1983.
29. Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, 2006.
30. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL'04*, 2004.
31. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
32. 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.
33. 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.
34. Andy M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
35. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
36. John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.
37. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, 2002.
38. Donald Sannella and Andrzej Tarlecki. Program Specification and Developemnt in Standard ML. In *POPL'85*, pages 67–77, 1985.
39. Zhong Shao. An overview of the FLINT/ML compiler. In *1997 ACM Workshop on Types in Compilation (TIC'97)*, 1997.
40. Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994.
41. David von Oheimb. Hoare logic for mutual recursion and local variables. In *FSTTCS*, volume 1738 of *LNCS*, pages 168–180, 1999.
42. Jannette Wing, Eugene Rollins, and Amy 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.
43. Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.

A Proof of Axioms in Remark 6.11

In this appendix we prove:

Proposition A.1 *Assume* β *is recursively finite and that* $\mathsf{Ref}(\beta)$ *does not occur in* β *up to the type isomorphism. Then the following assertion is valid, i.e. is true in any model:*

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x.\, j\# x) \quad \supset \quad (\langle !x\rangle(i^{\alpha} \hookrightarrow x^{\mathsf{Ref}(\beta)}) \;\equiv\; [!x](i^{\alpha} \hookrightarrow x^{\mathsf{Ref}(\beta)}) \;\equiv\; i^{\alpha} \hookrightarrow x^{\mathsf{Ref}(\beta)}) \tag{A.1}$$

which is equivalent to, through dualisation:

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x.\, j\# x) \quad \supset \quad (\langle !x\rangle i^{\alpha}\# x^{\mathsf{Ref}(\beta)} \;\equiv\; [!x] i^{\alpha}\# x^{\mathsf{Ref}(\beta)} \;\equiv\; x^{\mathsf{Ref}(\beta)}) \tag{A.2}$$

PROOF: Assume β is recursively finite and that $\mathsf{Ref}(\beta)$ does not occur in β up to the type isomorphism. Further let:

$$\mathcal{M} \stackrel{\mathrm{def}}{=} (\tilde{y})(\xi \cdot x^{\mathsf{Ref}(\beta)} : \mathbf{j},\ \sigma \cdot \mathbf{j} \mapsto W)$$

Now suppose we have:

$$\mathcal{M} \models \forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x.\, j\# x.$$

That is we can set, without loss of generality, that x occurs neither in ξ nor in σ, and that $\mathbf{j} = \{x\}$. Note further W can never contain any datum from which one can reach x, by the shape of β. By trivial operational reasoning, there is no possibility to have $i \hookrightarrow x$ except when we have $i = x$. Thus $i \hookrightarrow x$ is equivalent to $i = x$ whose validity does not depend on the content of x, as required. $\square$

B Graph Copy

Assertion for `Main` The main body of `graphCopy`, `Main`, uses a "scratchpad" x, just as the dagcopy. Thus its specification needs to say how the program changes the original scratchpad to a new one. The assertion, written: $\mathsf{gc}(r,i)(u)$, is given as follows.

$$
\begin{array}{l}
\mathsf{gc}(r,i)(u) \;\stackrel{\mathrm{def}}{=} \\
\quad \forall g, org. \\
\quad \{!x = org \;\;\wedge\;\; \mathsf{conPre}(org, g, r) \;\;\wedge\;\; g \notin \mathsf{dom}(org) \supset \mathsf{sizeR}(g,r,i)\} \\
\quad u \bullet g = g' \\
\quad \{\, \sharp^{-x}\{z' \mid \exists z.\langle z, z'\rangle \in !x \;\wedge\; \langle z, z'\rangle \notin org\}. \\
\quad\quad (\; (g \in \mathsf{dom}(org) \supset !x = org) \;\wedge \\
\quad\quad\quad (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x, g, r) \wedge !x = org \cup \langle g, g'\rangle_r^{*})\;)\;)\}@x
\end{array}
$$

Roughly $\mathsf{gc}(r,i)(u)$ says:

u can create an isomorphic copy of an original graph as far as "its size relative to a root node r" (this notion is made precise later) is no more than i, together with a consistent change in the content of the scratchpad.

When the argument graph is already in the scratchpad, u does nothing. Note the the predicate $\mathsf{gc}(r,i)(u)$ has the same structure as $\mathtt{dc}\,\tau(u)$. The predicates used above intuitively mean:

- $\mathsf{sizeR}(g,r,i)$ says the size of g relative to r is i.
- $!x = org$ records the initial table in org.
- $\mathsf{conPre}(org,g,r)$ says org is consistent as far as those nodes "before" g go, relative to r.
- $\mathsf{underR}(z,g',r)$ says z is "under" g' relative to r. $z \notin \mathsf{cod}(org)$ says z is in the codomain of the table org.
- $\mathsf{conPost}(!x,g,r)$ says x contains a table which is consistent as far as those nodes "under" or "before" g go, relative to r.
- $\langle g,g'\rangle_r^*$ is the pairs of nodes "downwardly reachable" from g and g' with common paths.

The formal definitions of these predicates are given below.

Associated Predicates. Predicates used for defining $\mathsf{gc}(r,i)(u)$ are defined below. Throughout $g,g',h,h',r,\ldots$ are of type *Tree*, while $p,p',\ldots$ are paths.

1. Rooted Lexicographic Ordering. We use the standard lexicographic order on paths, with left smaller than right, a prefix smaller than the whole. To wit, letting $p \prec q$ denote p is a prefix of q:

$$\frac{p \prec q}{p \sqsubseteq q} \qquad \frac{p \sqsubseteq q}{l.p \sqsubseteq l.q} \qquad \frac{p \sqsubseteq q}{r.p \sqsubseteq r.q} \qquad \frac{-}{l.p \sqsubseteq r.q}$$

Since $\sqsubseteq$ is the total order, the following predicate is well-defined:

$$\mathsf{minpathR}(r,g,p) \;\equiv\; \mathsf{path}(r,g,p) \;\wedge\; \forall p'.(\mathsf{path}(r,g,p') \supset p \sqsubseteq p')$$

Thus $\mathsf{minpathR}(r,g,p)$ asserts the shortest path from r to g is p.

Assume $g_{1,2}$ are reachable from r. Then we can order $g_{1,2}$ relative to r using their minimal paths as follows:

$$g_1 \sqsubseteq_r g_2 \quad \overset{\text{def}}{\equiv} \quad \exists p_{1,2}.(\wedge_{i=1,2}\mathsf{minpathR}(r,g_i,p_i) \;\wedge\; p_1 \sqsubseteq p_2)$$

$g_1 \sqsubseteq_r g_2$ says the shortest path from the root r to g_2 goes via g_1. This ordering reflects the program's behaviour: if $g_1 \sqsubseteq_r g_2$ `graphCopy` (starting from r) will reach g_1 before g_2.

2. Rooted Prefix Ordering. The rooted prefix relation is definable as follows:

$$g_1 \prec_r g_2 \quad \overset{\text{def}}{\equiv} \quad \exists p_{1,2}.(\wedge_{i=1,2}\mathsf{minpathR}(r,g_i,p_i) \;\wedge\; p_1 \prec p_2)$$

3. Rooted Closure Operation. The closure operation is defined as:

$$\langle g_0, g_0' \rangle_r^* \stackrel{\text{def}}{=}$$
$$\{\langle g, g' \rangle \mid g_0 \prec_r g,\ \exists p.\mathsf{path}(g_0, p, g),\ \mathsf{path}(g_0', p, g')\}$$

That is, $\langle g_0, g_0' \rangle_r^*$ is the set of "downwardly reachable" nodes from g_0 and g_0', through common paths.

4. Rooted "Before" Predicate. We define the notion "g_1 is before g_2 relative to r".

$$\mathsf{beforeR}(g_1, g_2, r) \ \equiv\ (g_1 \sqsubseteq_r g_2 \ \wedge\ \neg g_1 \prec_r g_2)$$

Thus $\mathsf{beforeR}(g_1, g_2, r)$ when g_1 is less than g_2 but g_1 is not a prefix of g_2. In other words, it says that g_1 is less than g_2 but not "above" g_2, i.e. is not in the minimum path between r and g_2.

5. Rooted "Under" and "Above" Predicate. The notion "g_1 is under g_2 relative to r" is nothing but $g_2 \prec_r g_1$.

$$\mathsf{underR}(g_1, g_2, r) \quad \equiv \quad g_2 \prec_r g_1$$

Dually we define "above" as:

$$\mathsf{aboveR}(g_1, g_2, r) \quad \equiv \quad g_1 \prec_r g_2$$

Intuitively, $\mathsf{underR}(g_1, g_2, r)$ says g_2 is between r (uppermost) and g_1 (below) w.r.t. minimal paths, while $\mathsf{aboveR}(g_1, g_2, r)$ says g_1 is between r (uppermost) and g_2 (below) w.r.t. minimal paths.

6. Rooted Size. The predicate $\mathsf{sizeR}(g, r, i)$ says that the number of nodes downwardly reachable from g w.r.t r is i. It is defined by induction on i (we give natural language definitions, from which their formal counterparts easily follow).

– $\mathsf{sizeR}(g, r, 1)$ holds iff, as well as g is reachable from r, either g is a leaf or the two branches of g are above g.
– $\mathsf{sizeR}(g, r, n+1)$ ($n \geq 1$) holds iff $\mathsf{sizeR}(g', r, n)$ for some g' such that g' is an immediate child of g and, moreover, g' is strictly below g (w.r.t. r).

For completeness, we set the predicate $\mathsf{sizeR}(g, r, 0)$ holds iff g is not reachable from r.

7. Weak Consistency (1). The consistency conditions for the table relating original nodes and newly created nodes use the following "one-step isomorphism".

$$\begin{aligned} &\mathsf{iso1}(\langle g_1, g_1' \rangle,\ \langle g_2, g_2' \rangle) \\ &\equiv \wedge_{i=l,r}(\mathsf{path}(g_1, i, g_2) \equiv \mathsf{path}(g_1', i, g_2')) \ \wedge \\ &\quad \forall n.(\mathsf{atom}(!g_1, n) \equiv \mathsf{atom}(!g_2, n)) \end{aligned} \tag{B.1}$$

Note the directedness of the predicate. The consistency used in the pre-condition is given as:

$$\begin{array}{l}
\mathsf{conPre}(t,g,r) \\
\;\equiv \forall g_0.((\mathsf{beforeOrAboveR}'(g_0,g,r) \supset g_0 \in \mathsf{dom}(org)) \;\wedge \\
\quad \forall g_{1,2}, g'_{1,2}. \\
\quad (\;(\;\mathsf{beforeR}'(g_1,g,r) \;\wedge\; \mathsf{beforeOrAboveR}'(g_2,g,r) \;\wedge \\
\qquad \langle g_1, g'_1\rangle, \langle g_2, g'_2\rangle \in t\;) \\
\qquad \supset\;\; \mathsf{iso1}(\langle g_1, g'_1\rangle,\; \langle g_2, g'_2\rangle)\;)
\end{array}$$

where we set, for brevity:

$$\begin{array}{rcl}
\mathsf{beforeR}'(h,g,r) & \stackrel{\text{def}}{=} & h \neq g \wedge \mathsf{beforeR}(h,g,r) \\
\mathsf{beforeOrAboveR}(h,g,r) & \stackrel{\text{def}}{=} & \mathsf{beforeR}(h,g,r) \vee \mathsf{aboveR}(h,g,r) \\
\mathsf{beforeOrAboveR}'(h,g,r) & \stackrel{\text{def}}{=} & h \neq g \wedge \mathsf{beforeOrAboveR}(h,g,r)
\end{array}$$

Note the first and third predicates define strict versions of "before" and "before or above" relations. The consistency condition says:

1. All nodes **strictly** before or above g are in $\mathsf{dom}(org)$;
2. if g_1 (source) is strictly "before" g and if g_2 (target) is strictly "before" or "above" g, one step path from g_1 to g_2 coincides with the path between the corresponding fresh nodes.

As further observations:

– By using one-step path, we are avoiding a round-about path going through nodes "after" g.
– Those nodes "above" g have already been placed in $\mathsf{dom}(org)$, but their corresponding fresh nodes usually point to only a temporary datum, so that a path **from** such a node may not be shared between the domain and codomain of *org*.
– A path **to** a node above g from those which are strictly "before" g (i.e. those which are processed) should already be isomorphic.

8. Weak Consistency (2). The consistency for the post-condition is similarly given, adding those nodes under g as its source.

$$\begin{array}{l}
\mathsf{conPost}(t,g,r) \\
\;\equiv \forall g_0.((\mathsf{aroundR}(g_0,g,r) \supset g_0 \in \mathsf{dom}(t)) \;\wedge \\
\quad \forall g_{1,2}, g'_{1,2}. \\
\quad (\;(\;\mathsf{beforeOrUnderR}(g_1,g,r) \;\wedge\; \mathsf{aroundR}(g_2,g,r) \;\wedge \\
\qquad \langle g_1, g'_1\rangle, \langle g_2, g'_2\rangle \in t\;) \\
\qquad \supset\;\; \mathsf{iso1}(\langle g_1, g'_1\rangle,\; \langle g_2, g'_2\rangle)\;)
\end{array}$$

where, for brevity, we set:

$$\mathsf{beforeOrUnderR}(h,g,r) \stackrel{\text{def}}{=} \mathsf{beforeR}(h,g,r) \vee \mathsf{underR}(h,g,r).$$
$$\mathsf{aroundR}(h,g,r) \stackrel{\text{def}}{=} \mathsf{beforeR}(h,g,r) \vee \mathsf{underR}(h,g,r) \vee \mathsf{aboveR}(h,g,r)$$

The second consistency condition says:

1. If g_0 is "before", "under" or "above" g, then g_0 should be in $\mathsf{dom}(org)$.
2. If g_1 (source) is "before" or "under" g and if g_2 (target) is "before", "under" or "above" g, then one step path from g_1 to g_2 coincides with the corresponding newly created nodes.

The condition is essentially identical with the pre-consistency above except those nodes "under" g are incremented.

8. Main Intermediate Assertion Revisited. At this point it may be valuable to revisit the main intermediate assertion, which we reproduce below for readability:

$$\begin{aligned}
\mathsf{gc}(r,i)(u) \stackrel{\text{def}}{=} \forall g, org.&\\
&\{\ !x = org \ \wedge\ \mathsf{conPre}(org,g,r)\ \wedge\ g \notin \mathsf{dom}(org) \supset \mathsf{sizeR}(g,r,i)\ \}\\
&u \bullet g = g'\\
&\{\ \sharp^{\text{-}x}\{z' \mid \exists z.(\langle z,z'\rangle \in !x \ \wedge\ \langle z,z'\rangle \notin org)\}.\\
&\quad (\ (g \in \mathsf{dom}(org) \supset !x = org) \wedge\\
&\quad\ (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x,g,r) \wedge !x = org \cup \langle g,g'\rangle_r^*)\ )\ )\ \}@x
\end{aligned}$$

We may observe the following correspondence between the main assertion and the program's behaviour.

– $\mathsf{sizeR}(g,r,i)$ does decrease when the program visits one of the branches of g which is strictly under g: and if its branch is not strictly under g, then that branch has already been processed, i.e. has already been placed in $\mathsf{dom}(org)$.
– If g is not in $\mathsf{dom}(org)$ hence it has indeed been processed, then those nodes under g are newly included in the table, together with corresponding freshly created nodes, except for those which are not in the codomain of org.
– The domain of the table further adds those nodes downwardly reachable from g and g', whereas its codomain adds their corresponding nodes.

Thus the assertion is nothing but logical articulation of behaviour of the main program.

For brevity, we shall call *GP* and *GG* for the precondition and postcondition of $\mathsf{gc}(r,i)(u)$:

$$\begin{aligned}
GP &\stackrel{\text{def}}{=} !x = org \wedge \mathsf{conPre}(org,g,r) \wedge g \notin \mathsf{dom}(org) \supset \mathsf{sizeR}(g,r,i)\\
GG &\stackrel{\text{def}}{=} \sharp^{\text{-}x}\{z' \mid \exists z.(\langle z,z'\rangle \in !x \wedge \langle z,z'\rangle \notin org)\}.\\
&\qquad (\ (g \in \mathsf{dom}(org) \supset !x = org) \wedge\\
&\qquad\ (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x,g,r) \wedge !x = org \cup \langle g,g'\rangle_r^*))
\end{aligned}$$

When we apply `Main` to the root node r, we should be able to derive:

$$\sharp^{\text{-}x}\{z|g \hookrightarrow z\}.\ !x = \langle g, g'\rangle^*_r \tag{B.2}$$

after invoking u with argument r (the root node), under the assumptions (1) $org = \emptyset$; and (2) $\forall r, n.\mathsf{gc}(r,n)(u)$.

Let us check that this is indeed possible, i.e. the pre-condition *GP* is indeed satisfied and, from *GG* under the conditions (1) and (2), we can derive (B.2). For the pre-condition, it suffices to show $\mathsf{conPre}(org, g, r)$. However because $\mathsf{beforeR}(h,r,r) \equiv h = r$ and because $\mathsf{beforeOrAboveR}'(h,r,r)$ never holds for any h, we immediately know $\mathsf{conPre}(\emptyset, g, r)$, so that *GP* holds.

For the postcondition, we calculate *GG* under $org = \emptyset$, $g = r$ and $g' = r'$. First, the content of x becomes:

$$!x \ = \ \emptyset \cup \langle r, r'\rangle^*_r \tag{B.3}$$

This entails the first condition of $\mathsf{conPost}(!x, r, r)$. Further, by the second condition of $\mathsf{conPost}(!x, r, r)$, we know

$$\forall g_{1,2}, g'_{1,2}.(\ \wedge_{i=1,2}\langle g_i, g'_i\rangle \in !x \supset \mathsf{iso1}((\langle g_1, g'_1\rangle,\ \langle g_2, g'_2\rangle))\ )$$

Similarly for the remaining condition.

Second, the condition for each $\sharp^{\text{-}x}$-fresh node g' becomes:

$$\exists z.(\mathsf{underR}(z,r,r)\ \wedge\ \langle z, g'\rangle \in !x)$$

By (B.3), this is the same thing as: $g' \prec_{r'} r'$, that is $r' \hookrightarrow g'$, as required.

Derivation (1): The Whole Program For the derivation for the whole program, we use:

$$U' \ = \ \{z \mid r' \hookrightarrow z \ \wedge\ z \notin \mathsf{cod}(org)\}$$

The derivation follows.

```
$u : \{\mathsf{T}\}$  (abs)
lambda r.
    $g' : \{\mathsf{T}\}$  (new)
    new x:= ∅ in
        $r' : \{!x = \emptyset\}$  (app)
            $m : \{\mathsf{T}\}$
            Main
            $\{\forall r, n.\mathsf{gc}(r,n)(m)\}@\emptyset$
            r
            $\{!x = \emptyset \wedge \{!x = \emptyset\} m \bullet r = r' \{\#^{\text{-}x} U'.\mathsf{iso}(r,r')\}@x\}@\emptyset$
        $\{\#^{\text{-}x} U'.\mathsf{iso}(r,r')\}@x$
    $\{\star r'.\mathsf{iso}(r,r')\}@\emptyset$
$\{\forall g.\{\mathsf{T}\} u \bullet g = g' \{\star g'.\mathsf{iso}(g,g')\}@\emptyset\}@\emptyset$
```

The structure is identical with the derivation for the dagcopy. Since many points overlap with the reasoning in §7.3, we only discuss the main difference.

- In l.11, the application is inferred using [$App\#$].
- From l.11 to l.12, we use the [*New*] rule. Let, for brevity, $M \overset{\text{def}}{=}$ `Main` and $C \overset{\text{def}}{=} !x = \emptyset$. Note the sub-derivation l.5–12 means $\{C\}M\{\#^{-x}U'.\mathsf{iso}(r,r')\}$, that is, with i fresh (cf.(7.4)):

  $$\{C\}M\{\forall z.((r' \hookrightarrow z \wedge x\#i) \supset z\#i) \wedge \mathsf{iso}(r,r')\}@x$$

  By [*New*], we can strengthen C by taking its conjunction with $x\sharp i$. Just as in the corresponding part in dagcopy (cf.§7.3), $x\#i$ is $!x$-free via (6.2), page 37, Section 6.1. By [*Inv*] we obtain:

  $$\{C \wedge x\#i\}M\{\forall z.(r' \hookrightarrow z \supset z\#i) \wedge \mathsf{iso}(r,r')\}@x$$

  that is

  $$\{C \wedge x\#i\}M\{\star r'.\mathsf{iso}(r,r')\}@x.$$

  Since [*New*] allows us to cancel x in the pre-condition, we obtain l.12.

Derivation (2): NewEntry Since the derivation for `NewEntry` is already given in §7.3, we only list the outermost derivation.

```
$m : \{\mathsf{T}\}$
  NewEntry
$\{NE(m)\}@\emptyset$
```

Derivation (3): `Main` The main body of the program is derived just as in §7.3: the only notable difference is how the inductive case is inferred.

```
$u : \{\mathsf{T}\}$  (rec)
mu f.
  $u : \{\forall j < n.\ \mathsf{gc}(jf)\}$  (abs)
  lambda g.
    $g' : \{\mathsf{sizeR}(g,r,n) \ \wedge\ !x = org \ \wedge\ \mathsf{conPre}(org,g,r)\}$  (if)
    if dom(!x,g) then
      $g' : \{g \in \mathsf{dom}(!x)\}$
      get(!x,g)
      $\{!x = org \ \wedge\ \langle g,g'\rangle \in !x\}@\emptyset$
      $\{GG\}@\emptyset$
    else
      $g' : \{g \notin \mathsf{dom}(!x)\}$  (case)
      case !g of
        inl(n):
          $g' : \{\mathsf{T}\}$
          NewEntry(inl(n), g)
          $\{\#^{-x}g'.(!x = org \cup \langle g,g'\rangle \ \wedge\ \mathsf{conPre}(org,g,r) \ \wedge$
           $\mathsf{atom}(!g,n) \ \wedge\ \mathsf{atom}(!g',n))\}@x$
          $\{GG\}@x$
```

```
20          inr(pair<g1, g2>):
21            $g' : \{\mathsf{T}\}$
22            let h =
23              NewEntry(tmp, g) in
24              h := inr(pair<f y1, f y2>);
25              $\{GG[h/g']\}@x$
26              h
27            $\{GG\}@x$
28        $\{GG\}@x$
29      $\{GG\}@x$
30    $\{\mathsf{gc}(r,i)(u)\}@\emptyset$
31  $\{\forall r, i.\mathsf{gc}(r,i)(u)\}@\emptyset$
```

Above we make explicit the constructor **pair** for paring, for a later convenience. As before, when two assertions are repeated in consecutive lines (such as Line 9 and Line 10), it means the previous assertion implies the following one.

Note the structure of the inference is essentially identical with that for `dagCopy`. As in the main program of `dagCopy`, the program processes g, its argument, in three different ways, depending on three sub-cases of g.

(A) g is already in the table;
(B) g is not in the table and g is a leaf; and
(C) g is not in the table and g is a branch.

In the following pragraphs, we treat each case one by one.

(A) No Processing. This is treated in Lines 7–11. The only non-trivial inference is to show Line 9 implies Line 10, i.e.:

$$!x = org \ \wedge \ \langle g, g'\rangle \in !x \quad \supset \quad GG$$

Since $!x = org$, the $\sharp$-fresh names become, from $!x = org$:

$$\{g' \mid \exists z.(\langle z, g'\rangle \in !x \ \wedge \ \langle z, g'\rangle \notin org)\} \ = \ \emptyset.$$

Next, from $\langle g, g'\rangle \in !x$ we obtain $g \in \mathsf{dom}(org)$, in which case GG says we should have $!x = org$, which indeed holds. Thus we have:

$$\begin{aligned} !x = org \ \wedge \ \langle g, g'\rangle \in !x &\Rightarrow \sharp^{-x}\emptyset.\ (g \in \mathsf{dom}(org) \supset !x = org) \\ &\Rightarrow GG \end{aligned}$$

We have arrived at Line 10.

(B) Base Case. Lines 15 to 19 treat the base case, i.e. the case when (1) the argument g is not in the table (has not been processed) and (2) g is a leaf with value n. Lines 17/18 is inferred using $!x$-freedom of $\mathsf{conPre}(g, org, r)$, which is easily shown by noting all terms used in $\mathsf{conPre}(g, org, r)$ cannot reach x (we omit the formal reasoning). The only

non-trivial inference is derivation of Line 19 from Lines 17/18. It suffices to show, again using $!x$-freedom of $\mathsf{conPre}(g, org, r)$:

$$\mathsf{conPre}(g,org,r) \wedge !x = org \cup \langle g,g' \rangle \ \wedge \ \mathsf{atom}(!g,n) \ \wedge \ g \notin \mathsf{dom}(org) \tag{B.4}$$

entails GG.

First, for the $\sharp^{\text{-}x}$-free names, a trivial set-theoretic reasoning gives us, under (B.4):

$$\{z' \mid \exists z.(\langle z,z' \rangle \in !x \backslash org)\} \tag{B.5}$$

Second, we derive the body, i.e.

$$\begin{array}{l}(g \in \mathsf{dom}(org) \supset !x = org) \ \wedge \\ (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x,g,r) \wedge !x = org \cup \langle g,g' \rangle_r^*) \ )\end{array}$$

Since $g \notin \mathsf{dom}(org)$, it suffices to derive:

$$\mathsf{conPost}(!x,g,r) \wedge !x = org \cup \langle g,g' \rangle_r^*)$$

Since g and g' are atoms, $\langle g,g' \rangle_r^* = \langle g,g' \rangle$, hence we only have to derive $\mathsf{conPost}(!x,g,r)$ from (B.4). First we note, because g is an atom:

$$\mathsf{underR}(h,g,r) \ = \ \{g\} \tag{B.6}$$

We recall $\mathsf{conPost}(!x,g,r)$ has the shape:

$$\begin{array}{l}\forall g_0.(\mathsf{aroundR}(g_0,g,r) \supset g_0 \in \mathsf{dom}(!x)) \wedge \\ \forall g_{1,2}, g'_{1,2}.(\ (\ \mathsf{beforeOrUnderR}(g_1,g,r) \ \wedge \ \mathsf{aroundR}(g_2,g,r) \ \wedge \ \langle g_1,g'_1 \rangle, \langle g_2,g'_2 \rangle \in t\ ) \\ \quad \supset \ \mathsf{iso1}(\langle g_1,g'_1 \rangle, \ \langle g_2,g'_2 \rangle)\ )\end{array}$$

For the first component, under (B.4) hence (B.6):

$$\begin{array}{rl}\forall g_0.(\mathsf{aroundR}(g_0,g,r) \supset g_0 \in \mathsf{dom}(!x)) & \Leftrightarrow \forall g_0.(\mathsf{beforeOrAboveR}(g_0,g,r) \supset g_0 \in \mathsf{dom}(!x))) \\ & \Leftarrow \mathsf{conPre}(org,g,r).\end{array}$$

where

$$\mathsf{beforeOrAboveR}(h,g,r) \overset{\text{def}}{=} \mathsf{beforeR}(h,g,r) \vee \mathsf{aboveR}(h,g,r)$$

The second component is equivalent to:

$$\begin{array}{c}\forall g_{1,2}, g'_{1,2}. \\ (\ (\ \mathsf{beforeR}(g_1,g,r) \ \wedge \ \mathsf{beforeOrAboveR}(g_2,g,r) \ \wedge \ \langle g_1,g'_1 \rangle, \langle g_2,g'_2 \rangle \in t\ ) \\ \supset \ \mathsf{iso1}(\langle g_1,g'_1 \rangle, \ \langle g_2,g'_2 \rangle)\ )\end{array}$$

which is easily implied by (B.4) (to be precise by $\mathsf{conPre}(org,g,r)$ and $\mathsf{atom}(!g,n)$), as required.

(C) Inductive Case. We show the inductive case, the second `case` construct, when the argument is not in the table and it is a branch.

We first present the asserted program fragment excepting the inference for the two recursive calls (which is detailed next).

```
$\{!x = org \;\wedge\; \mathsf{conPre}(org, g, r) \;\wedge\; g \notin \mathsf{dom}(org)\}$
$\{\forall i < j.\ \mathsf{gc}(i, r)(f) \;\wedge\; \mathsf{sizeR}(g, r, j)\}$
$g' : \{\mathsf{branch}(!g, g_1, g_2)\}$  $(\mathtt{let})$
let h =
    $g' : \{\mathsf{T}\}$
    NewEntry(tmp, g) in
    $\{\#^{\text{-}x} g'.(!x = org \cup \{\langle g, g' \rangle\})\}$
$g' : \{\ldots\}$  $(\mathtt{seq})$
    $\{\mathsf{T}\}$  $(\mathtt{assvar}\#)$
    h:= inr(pair<f g1, f g2>);
    $\{GG[h/g']\}@x$
    h
$\{GG\}@x$
```

The inference above is mechanical except for the omitted part, the reasoning for the $\mathtt{inr}(\langle f g_1, f g_2 \rangle)$.

```
$\{g \notin \mathsf{dom}(org) \;\wedge\; !x = org \cup \langle g, g' \rangle \;\wedge\; \mathsf{conPre}(org, g, r)\}$
$\{\forall i < j.\ \mathsf{gc}(i, r)(f) \;\wedge\; \mathsf{sizeR}(g, r, j)\}$
$\{\mathsf{branch}(!g, g_1, g_2)\}$
$n : \{\mathsf{T}\}$  $(\mathtt{inr})$
inr
    $m : \{\mathsf{T}\}$  $(\mathtt{pair}\#)$
    pair
        $g'_1 : \{\mathsf{T}\}$
        f g1
        $\{GG(g_1, g'_1, org)\}@x$
        $g'_2 : \{\ldots\}$
        f g2
        $\{GG1(g_1, g_2, g'_1, g'_2, org')\}@x$
    $\{GG2(g_1, g_2, m, org')\}@x$
$\{GG3(g_1, g_2, n, org')\}@x$
```

Above we use a parametrised version of *GG* and its refinements:

$$\begin{array}{rcl}
GG(g,g',org) & \overset{\text{def}}{=} & \{\ \sharp^{\text{-}x}\{z' \mid \exists z.(\langle z,z'\rangle \in !x\ \wedge\ \langle z,z'\rangle \notin org)\}. \\
& & \quad (\ (g \in \mathsf{dom}(org) \supset !x = org) \wedge \\
& & \quad\ \ (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x,g,r) \wedge !x{=}org\cup\langle g,g'\rangle^*_r))) \\
GG1(g_1,g_2,g'_1,g'_2,org) & \overset{\text{def}}{=} & \{\ \sharp^{\text{-}x}\{z' \mid \exists z.(\langle z,z'\rangle \in !x\ \wedge\ \langle z,z'\rangle \notin org)\}. \\
& & ((g_{1,2} \in \mathsf{dom}(org) \supset !x = org)\ \ \wedge \\
& & ((g_{1,2} \in \mathsf{dom}(org) \supset !x = org) \supset \\
& & \qquad (\mathsf{conPost}(!x,g,r) \wedge !x{=}org\cup \bigcup_{i=1,2} \langle g_i,g'_i\rangle^*_r))\ \ \wedge \\
& & ((g_1 \in \mathsf{dom}(org) \wedge g_2 \notin \mathsf{dom}(org)) \supset \\
& & \qquad (\mathsf{conPost}(!x,g,r) \wedge !x{=}org\cup\langle g_2,g'_2\rangle^*_r))\ \ \wedge \\
& & ((g_2 \in \mathsf{dom}(org) \wedge g_1 \notin \mathsf{dom}(org)) \supset \\
& & \qquad (\mathsf{conPost}(!x,g,r) \wedge !x{=}org\cup\langle g_1,g'_1\rangle^*_r))) \\
GG2(g_1,g_2,m,org) & \overset{\text{def}}{=} & \exists g'_{1,2}.(m = \langle g'_1,g'_2\rangle \wedge GG1(g_1,g_2,g'_1,g'_2,org)) \\
GG3(g_1,g_2,n,org) & \overset{\text{def}}{=} & \exists g'_{1,2}.(n = \mathtt{inr}(\langle g'_1,g'_2\rangle) \wedge GG1(g_1,g_2,g'_1,g'_2,org))
\end{array}$$

We observe:

– From Line 13 to Line 14, i.e. from $GG1(g_1,g_2,g'_1,g'_2,org)$ to $GG2(g_1,g_2,m,org)$, is direct from the proof rule for pairing.
– From Line 14 to Line 15, i.e. from $GG2(g_1,g_2,m,org)$ to $GG3(g_1,g_2,n,org)$, is direct from the proof rule for injection.

Thus the only non-trivial inferences in the reasoning above are before and after each recursive application, which we discuss below (the following natural language inferences are for clarity and can be easily made into formal inferences).
First Application. We show the assumptions in Lines 1–3 together imply the following precondition for the first application. Once this holds, then by the induction hypothesis on f, we immediately obtain the postcondition, $GG(g_1,g'_1,org)$. Below and henceforth we set $org0 \overset{\text{def}}{=} org \cup \langle g,g'\rangle$.

$$\mathsf{conPre}(org,g_1,r)\ \ \wedge\ \ g_1 \notin \mathsf{dom}(org0) \supset \mathsf{sizeR}(g_1,r,n)$$

In other words, we should show Lines 1–3 entail:

(a) $\mathsf{conPre}(org0,g_1,r)$.
(b) If $g_1 \notin \mathsf{dom}(org0)$, then $\mathsf{sizeR}(g_1,r,k)$ such that $k \lneq n$.

We note g_1 can be located either:

(i) before g;

(ii) above g; or
(iii) strictly below g.

Condition (a). We obtain $\mathsf{conPre}(org0, g_1, r)$ by inspecting each of its component. Firstly we can easily derive

$$\forall h.((\mathsf{beforeOrAboveR}'(h, g_1, r) \supset h \in \mathsf{dom}(org0))$$

from

$$\forall h.((\mathsf{beforeOrAboveR}'(h, g, r) \supset h \in \mathsf{dom}(org))$$

through the assumption, in any of (i), (ii) and (iii). Secondly, we already know one-step isomorphism with the following source and target:

– $\mathsf{beforeR}(h_1, g, r)$ as a source.
– $\mathsf{beforeOrAboveR}(h_1, g, r)$ as a target

If (i) is the case, since "above" g_1 is already before or above g, the same holds for g_1. If (ii) is the case, those "above" g_1 are a subset of those "above" g, similarly for "before", hence done. If (iii) is the case, the "above" nodes add g, while "before" nodes are unchanged. However in this case no other edges can exist from the nodes strictly before g (since if so g_1 itself should be strictly before g), hence as required.

Secondly, for the second component:

$$\begin{array}{l}\forall h_{1,2}, h'_{1,2}.\\ (\ (\ \mathsf{beforeR}(h_1, g_1, r)\ \wedge\ \mathsf{beforeOrAboveR}'(h_2, g_1, r)\ \wedge\\ \quad \langle h_1, h'_1\rangle, \langle h_2, h'_2\rangle \in org0\ )\\ \quad \supset\ \ \mathsf{iso1}(\langle h_1, h'_1\rangle,\ \langle h_2, h'_2\rangle)\ )\end{array}$$

Again we derive this assertion from the pre-condition for g:

$$\begin{array}{l}\forall h_{1,2}, h'_{1,2}.\\ (\ (\ \mathsf{beforeR}(h_1, g, r)\ \wedge\ \mathsf{beforeOrAboveR}'(h_2, g, r)\ \wedge\\ \quad \langle h_1, h'_1\rangle, \langle h_2, h'_2\rangle \in org\ )\\ \quad \supset\ \ \mathsf{iso1}(\langle h_1, h'_1\rangle,\ \langle h_2, h'_2\rangle)\ )\end{array}$$

By the reasoning for the first component, we already know the only case when new edges (one-step paths) should be considered is when $g_1 \notin \mathsf{dom}(org)$, i.e. when (iii) holds. In this case, however, there is no path from "before" nodes to g_1 as noted, hence in fact there is no additional one step path, as required.

Condition (b). This is easy:

1. If (i) and (ii) hold, then by the pre-consistency we know $g \in \mathsf{dom}(org)$, a contradiction.
2. If (iii) holds, then by definition we know the minimum path from r to g_1 strictly includes the one from r to g, i.e. (b) holds.

hence as required.
Second Application. We have $GG(g_1, g'_1, org)$ together with assumptions in Lines 1–3 except for replacing $!x = org$ with $!x = org1$, where we set $org1$ to be $org0 \cup \langle g_1, g'_1 \rangle$ (which is, in detail, equal to $org0$ if $g_1 \in \mathsf{dom}(org)$; and to $org0 \cup \langle g_1, g'_1 \rangle$ if else).

We first show this is enough as the precondition for the second recursion. As before, we need to show:

(a) $\mathsf{conPre}(org', g_2, r)$.
(b) If $g_2 \notin \mathsf{dom}(org)$, then $\mathsf{sizeR}(g_2, r, k)$ such that $k \lneq n$.

Again g_2 can be located either:

(i) before g;
(ii) above g; or
(iii) strictly below g.

Condition (a). The first component of $\mathsf{conPre}(org', g_2, r)$ is:

$$\forall h.((\mathsf{beforeOrAboveR}'(h, g_2, r) \supset h \in \mathsf{dom}(org1))$$

The reasoning is essentially the same as for g_1 (except g_1 itself can be "before" g_2):

– If (i) or (ii) is the case, i.e. if g_2 is before or above g, then surely g_2 is already in $\mathsf{dom}(org)$.
– If (iii) holds, then $\mathsf{beforeOrAboveR}'(h, g_2, r)$ may entail $h = g$ or $h = g1$ in addition: however we already know g and g_1 are in $\mathsf{dom}(org)$, hence done.

For the second component, we wish to have:

$$\begin{array}{l} \forall h_{1,2}, h'_{1,2}. \\ (\ (\ \mathsf{beforeR}(h_1, g_2, r)\ \wedge\ \mathsf{beforeOrAboveR}'(h_2, g_2, r)\ \wedge \\ \quad \langle h_1, h'_1 \rangle, \langle h_2, h'_2 \rangle \in org1\ ) \\ \quad \supset\ \ \mathsf{iso1}(\langle h_1, h'_1 \rangle,\ \langle h_2, h'_2 \rangle)\ ) \end{array} \tag{B.7}$$

In order to derive the assertion (B.7) from the immediately preceding postcondition, we observe there are two cases.

(a-1) $g_1 \notin \mathsf{dom}(org0)$, in which case all under g_1 are in $org1$ and, moreover, $\mathsf{conPost}(g_1, org1, r)$ does hold.
(a-2) $g_1 \in \mathsf{dom}(org0)$, in which case nothing has happened in processing g_1, so that we have $org1 = org0$.

We further note:

(I) If g_2 is before or above g, then (1) $\mathsf{beforeR}'(h_1, g_2, r)$ implies $\mathsf{beforeR}(h_1, g, r)$; and (2) $\mathsf{beforeOrAboveR}'(h_2, g_2, r)$ implies $\mathsf{beforeOrAboveR}'(h_2, g, r)$.
(II) If g_2 is strictly below g, then we have (1) $\mathsf{beforeR}'(h_1, g_2, r)$ implies $\mathsf{beforeR}(h_1, g, r)$ or, if (a-1) above holds, h_1 is g_1 or a node strictly below g_1 (if any); and (2) $\mathsf{beforeOrAboveR}'(h_2, g_2, r)$ implies $\mathsf{beforeOrAboveR}'(h_2, g, r)$ or $h_2 = g$ or, if (a-1) above holds, h_1 is g_1 or one of the nodes strictly below g_1 (if any).

We first consider the case **(a-1)**. We then have:

$$\begin{array}{l}
\mathsf{conPost}(org1, g1, r) \\
\equiv \forall h.((\mathsf{aroundR}(h,g,r) \supset h \in \mathsf{dom}(org1)) \ \wedge \\
\quad \forall h_{1,2}, h'_{1,2}. \\
\quad (\ (\ \mathsf{beforeOrUnderR}(h_1, g_1, r) \ \wedge \ \mathsf{aroundR}(h_2, g_1, r) \ \wedge \\
\quad\quad \langle h_1, h'_1\rangle, \langle h_2, h'_2\rangle \in org1 \ ) \\
\quad\quad \supset \ \mathsf{iso1}(\langle h_1, h'_1\rangle, \ \langle h_2, h'_2\rangle) \ ))
\end{array} \qquad \text{(B.8)}$$

If (I) is the case, then the isomorphism in (B.8) subsumes that of (B.7) since (because g_1 is strictly below g) (1) $\mathsf{beforeR}(h_1, g, r)$ implies $\mathsf{beforeOrUnderR}(h_1, g_1, r)$ and (2) $\mathsf{beforeOrAboveR}'(h_2, g, r)$ implies $\mathsf{aroundR}(h_2, g_1, r)$. The case when (II) holds is immediate from the definition.

We next consider the case **(a-2)**. In this case we have, in addition to being g_1 being above or before g:

$$\begin{array}{l}
\forall h_{1,2}, h'_{1,2}. \\
(\ (\ \mathsf{beforeR}'(h_1, g, r) \ \wedge \ \mathsf{beforeOrAboveR}'(h_2, g, r) \ \wedge \\
\quad \langle h_1, h'_1\rangle, \langle h_2, h'_2\rangle \in org1 \ ) \\
\quad \supset \ \mathsf{iso1}(\langle h_1, h'_1\rangle, \ \langle h_2, h'_2\rangle) \ )
\end{array} \qquad \text{(B.9)}$$

Next assume g_2 is strictly below g (if not, we already know the required result is vacuous). Then we have:

– $\mathsf{beforeR}'(h_1, g_2, r)$ implies $\mathsf{beforeR}(h_1, g, r)$, since g_1 does not count any more.
– $\mathsf{beforeOrAboveR}'(h_2, g_2, r)$ implies $\mathsf{beforeOrAboveR}'(h_2, g, r)$ or $h_2 = g_2$.

Thus it suffices to consider the edges to g_2 from those nodes in $\mathsf{beforeR}(h_1, g, r)$. However such edges cannot exist since g_2 is strictly below g, hence done.

Condition (b). Again if (i) and (ii) hold, then by the pre-consistency we know $g_2 \in \mathsf{dom}(org1)$. Hence (iii) is the only possibility, in which case $g \prec_r g_2$ immediately holds.

Finally, the postcondition of fg_2 becomes $GG2(g_1, g_2, m, org)$ since the content of $!x$ becomes $org1$ itself if $g_2 \in \mathsf{dom}(org)$ and $org1 \cup \langle g_2, g'_2\rangle^*_r$ if $g_2 \notin \mathsf{dom}(org)$; while the added $\sharp^{-x}$-fresh names are simply incremented using the induction hypothesis, hence done.

This concludes the inductive case, hence the derivation of the main judgement for `graphCopy`.