Local State in Hoare Logic for Imperative Higher-Order Functions

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

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

Abstract. We introduce an extension of Hoare logic for imperative higher-order functions with local state and full aliasing. 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 logic is built on the authors' previous study on Hoare logics for imperative higher-order functions and aliasing [5, 18–20]. Complex behaviour induced by local state is captured with a minimal syntax extension: a first order predicate which asserts reachability of reference names. After establishing soundness and observational completeness, we explore the logic's descriptive and reasoning power with non-trivial examples, offering effective reasoning principles for a general class of higher-order behaviours in the presence of local state.

Table of Contents

1 Introduction

This work introduces a compositional program logic for imperative higher-order functions with local state, the subject which defies a clean theoretical treatment for decades. Based on recent accumulated semantic understanding on this class of computation [3, 8, 25, 33, 37, 42], as well as building on our preceding work [5, 18, 20], we show a simple syntactic extension of Hoare's logic suffices to fully capture semantics of local state integrated with higher-order functions and general forms of aliasing. As has been clarified by many researchers, imperative higher-order functions with local state are capable of representing diverse forms of behaviours including standard procedural idioms, stored procedures, dynamic mutable data structures and objects: thus logically capturing imperative higher-order functions enables us, in principle, to do the same for these diverse classes of behaviour. We demonstrate a glimpse of such applicability through simple but non-trivial assertion and reasoning examples.

In this introduction, we first illustrate the nature and use of local state in imperative higher-order programming: then discusses key technical challenges a logic for local state should treat. How, and whether, these challenges can be satisfactorily met is the theme of the subsequent sections.

Local State in Imperative Higher-Order Programming. Local state in programming languages is usually realised by an imperative store whose visibility is restricted to local configurations. Existence and state of a local cell is hidden from outside of its scope. This gives a means to enhance modularity through localisation of read/write effects, both for data structures and procedures. It also leads to non-functional behaviour of programs. Consider the following program

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

Above we use standard notation: in particular, $\mathtt{new}\ x := M\ \mathtt{in}\ N$ first evaluates M to a value V, generates a fresh reference (imperative variable) x whose initial content is V, then evaluates N which may have access to x; whereas $!x$ denotes dereferencing an imperative variable x. After `Inc` runs, we obtain an unnamed procedure with local state. When this procedure is invoked, it increments the content of its local variable x, and returns the new content. Thus each time it is called, the program returns a different result, whose source is hidden from external observers. This is different from $\lambda().(x := !x+1;\, !x)$ where the content of x is globally accessible, so an external observer can describe its behaviour as a function from initial state and an argument to final state and a return value. Local state combined with higher-order procedures gives us a rich repertoire of behaviour unrealisable without it (this expressive power is attested to by encodability of objects [3, 8, 37] and of low-level idioms [2] in imperative λ-calculi).

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

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

realises the same observable behaviour as `Inc`. But if x is not local in `Inc` and `Inc2`, they have clearly distinct visible behaviours. This potential distance between the extensional and internal behaviour of a program can be exploited as a basis of modular programming. But this distance also causes difficulties in reasoning, since it makes correspondence between programs' syntactic structures and their behaviours subtle to establish, as has been explored by many authors including [25, 33, 42].

Freshness of names of imperative variables generated by programs is a fundamental element of the semantics of local state. Consider the following program ($\mathsf{Ref}(\mathsf{Nat})$ is the type of a reference storing natural numbers):

$$\lambda x^{\mathsf{Ref}(\mathsf{Nat})}.\mathtt{new}\ y := 1\ \mathtt{in}\ \mathtt{if}\ x = y\ \mathtt{then}\ 0\ \mathtt{else}\ 100 \tag{1.3}$$

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.

A local reference may be exported to the outside of its original scope. An immediate example are ML references:

$$\mathtt{ref}(M) \stackrel{\text{def}}{=} \mathtt{new}\ x := M\ \mathtt{in}\ x \tag{1.4}$$

This program generates a new reference cell with content M (or rather the value M evaluates to) and exports it immediately. As another example, consider the following program with stored procedures:

```
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 $l.1$, 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 using the standard reduction semantic [13, 36, 43], we can check that the initial value of x is 0 (at $l.1$ and $l.2$), then it is incremented at $l.3$ and $l.4$, so x's final value is 2 and the return value of this program is 3. Note that x is incremented every time the function is called. To describe 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, or shared by, $!b$. Controlling visibility and sharing by combining local state and scope extrusion is a foundation of many programming practice, including manipulation of complex, dynamically generated mutable data structures.

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 many scenes of 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 $\mathsf{Ref}(\alpha)$-types.
– Treatment of (dynamically generated and extruded) local state.

The first is an essential feature of any higher-order languages and a primary source of their expressive power. 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 general forms of potential aliasing (as well as contributing to the expressive power). The third feature has been discussed already. These three are fundamental elements of practical typed higher-order programming, but have defied clean logical treatment for a long time.

In preceding studies, the present authors have proposed syntactically simple extensions of Hoare logic which capture the first two of these features. Higher-order functions are treated in [18–20], aliasing in [5]. The resulting logic enjoys a tight link with standard observational semantics in that assertions distinguish programs' behaviour just as the contextual behavioural equivalence does (called *observational completeness*). We believe the last property is important when, for example, we wish to use compositional program logics together with other tools as part of a mathematically well-founded software infrastructure.

In this paper we aim to extend a similar logical treatment to local state. To this end we need to have clear answers to at least the following three problems.

1. *Shape of assertions.* What logical constructs can we use for capturing the richness of local state?
2. *Semantics of logic.* What models and interpretations can we use for building the logic on a clear semantic basis?
3. *Effectiveness of reasoning.* Does logically capturing the richness of imperative functions with local state lead to improvements of existing reasoning techniques, for example those for complex mutable data structures?

In the following we briefly elaborate each of these points.

Assertions: Freshness and Hiding. To be able to assert precisely on programs with local state, an assertion language should capture freshness and hiding. There are several recent proposals to deal with them logically [10, 11, 26, 38], but none of them in the setting of compositional program logics. In the present context, since local state is an integral part of imperative higher-order behaviour, an added construct for it should work in harmony with those proposed in [5, 19, 20]. Further, freshness and hiding have subtle interplay. Take $\mathtt{ref}(M)$, a reference agent in ML discussed already. It creates a fresh reference and then exports it in full. So if we are to assert the post-condition of this program, we have freshness, not hiding. But we can realise the basic construct for local state, $\mathtt{new}\,x := M\ \mathtt{in}\,N$, which surely can induce hiding, using $\mathtt{ref}(M)$ and $\mathtt{let}$:

$$\mathtt{let}\,x = \mathtt{ref}(M)\ \mathtt{in}\,N \qquad (1.5)$$

Given `let` only connects a result from one behaviour to another through a function variable (with the corresponding proof rule [5, 20]), this seems to suggest the assertion language and proof rules should allow conversion of freshness to hiding – and vice versa, since $\mathtt{ref}(M)$ is realisable as $\mathtt{new}\ x := M\ \mathtt{in}\ x$. Is there a simple logical answer to capture this convertibility?

Semantics: Behaviours with Locality. Hoare logic, and all of our extensions to it so far [5, 18–20], are built on standard first-order logic with equality. The semantic universes we have used in our extensions directly include abstract notions of higher-order behaviour, which are more complex than number theory, the universe in Hoare's original setting. Does richness of local state add a new twist in this context? Above, we discussed `IncShared`, a program which treats two stored procedures. Now compare it with the following program:

$$a := \mathtt{Inc};\ b := \mathtt{Inc};\ z_1 := (!a)();\ z_2 := (!b)();\ (!z_1 + !z_2) \qquad (1.6)$$

We call this program `IncUnShared`, which assigns to a and b two separate instances of `Inc`. We now ask whether, after the second assignment of (1.6), we should be allowed to assert "$!a\ =\ !b$" (i.e. the content of a and b are equal). Let us recall the standard defining clause for the satisfaction of equality of two logical terms:

$$\mathcal{M} \models e_1 = e_2 \quad \stackrel{\text{def}}{\equiv} \quad \mathcal{M}[\![e_1]\!] = \mathcal{M}[\![e_2]\!]$$

(i.e. $e_1 = e_2$ is true under a model $\mathcal{M}$ iff their interpretations are equal in $\mathcal{M}$ as elements of a set — sometimes equality is replaced by equivalence). Since surely two instances of `Inc` have the identical behaviours, the equality $!a\ =\ !b$ should hold for `IncUnShared`. We however observe that, in `IncUnShared`, running $!a$ twice and running $!a$ and $!b$ consecutively lead to different observable behaviours, due to their distinct local states. This violates one of the fundamental properties of equality, substitutivity [24, §2.8] (which says one can always substitute an equal for an equal). On the other hand, running $!a$ and running $!b$ have always identical observable effects in `IncShared`, thanks to their shared local state. So may we conclude `IncShared` should allow us to assert $!a = !b$?

But this also poses a problem: if $!a$ and $!b$ share a local state, what are the interpretations of $!a$ and $!b$? How can we represent a shared hidden state in such interpretation, given its key semantic feature is precisely its hidden-ness, the source of such phenomena as representation independence?

Effective Reasoning for Mutable Data Structures. The program logic we wish to obtain should work with all standard data types in imperative higher-order functions in full type hierarchy and the standard, unconstrained and typed "new" construct. We also like it to have a precise correspondence with the semantics of the target programming language. With these features, we could assert on, and reason about, arbitrary programs typable in, say, the basic part of ML, covering a fertile and broad terrain.

One of the concrete potentials arising from such expressivity is general and effective logical treatment of data structures [17]. In their analytical presentation, ML, Haskell and other typed higher-order languages allow precise representation of many important idioms for manipulating mutable data structures which may be dynamically generated and shared, couched in a rigorous type discipline. Generality in type structure also leads to the introduction of stored procedures, significant in many system-level programming languages [12]. Does generality in expressiveness of the logic (which we aim at), reflecting that of the programming language, lead to efficient reasoning methods on these data structures? What precisely does it add in comparison with existing approaches?

1.1 Contributions.

In this paper we present a compositional program logic for imperative higher-order functions with local state in full type hierarchy, as a minimal syntactic extension of the logic we presented in [5, 20]. Built on a rigorous semantic basis, it enjoys soundness as well as semantic completeness. To our knowledge this is the first time such a program logic is presented. Several non-trivial reasoning examples are shown including those involving dynamic mutable data structures taken from "benchmark" examples discussed by Reynolds, Bornat and others in [7, 41].

1.2 Prospect.

One of the central aspects of (compositional) program logic is to be able to logically assert, and potentially validate, essentially arbitrary observable properties of programs. While the assertion-based method has been already successful in practice, we believe this feature can be exploited further as a basis of diverse forms of software infrastructure, if sufficiently broad theoretical and engineering foundations can be established.

One of the basic practical and theoretical challenges in this context is to be able to assert and reason about software behaviour independently from individual programming languages. For this purpose we need a theory which can represent behaviour of programs in essentially arbitrary programming languages, up to a certain abstraction level but preserving their central properties. It is with this in mind that we started to work on the π-calculus-based theory of program logics [5, 18–20], where we aim to give a theory of logics for typed processes that can accommodate and mediate program logics for different programming languages, exploiting the well-known expressive power of the π-calculus [27, 28] and its generic behavioural logic, Hennessy-Milner logic [14, 29].

While presented as a logic for a specific programming language, the study of the proposed logic, as its predecessors [5, 18–20], has been carried out in this broader technical and conceptual framework, offering a π-calculus foundation of program logics for sequential higher-order languages. Our hope is that deeper theoretical understanding leads to truly effective engineering principles: in future we plan to report how we may utilise such understanding practically, along with further theoretical inquiries.

1.3 Structure of the Paper.

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. Among others we show how "freshness" and "locality" commute at the level of both assertions and derivations. Section 6 establishes soundness 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 [13, 36]. 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.

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

Above we use the standard notation [13, 36, 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 [5], except for the inclusion of a construct for local references (highlighted in the grammar above). We use boolean/arithmetic constants $(\mathtt{c}, \mathtt{c}', \ldots)$ and operations on them. 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 [13, 36, 43], except for the focus of the present study. The new name generation, $\mathtt{new}\, x := M \,\mathtt{in}\, N$, 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 x of type $\mathsf{Ref}(\alpha)$ with initial content V; Finally N (which may possibly use x) is evaluated.

We formalise this and other behaviour of programs using the standard (one-step, call-by-value) reduction [13, 36, 43].

A *store* $(\sigma, \sigma', \ldots)$ is a finite map from names to semi-closed values. We write $\mathsf{dom}(\sigma)$ for the domain of σ and $\mathsf{fv}(\sigma)$ for names occurring in both the domain and co-domain of σ. $\sigma[x \mapsto V]$ denotes the store which maps x to V and otherwise agrees with σ, for each $x \in \mathsf{dom}(\sigma)$. An *open configuration* is a pair of a semi-closed program and a store, written $(M,\ \sigma)$. A *configuration* is an open configuration (M, σ) combined with a subset of names of $\mathsf{dom}(\sigma)$, written $(\nu \tilde{x})(M, \sigma)$ (the order of $\tilde{x}$ does not matter). We call $(\nu \tilde{x})$ in $(\nu \tilde{x})(M, \sigma)$, the latter's ν*-binder*, and consider names in $\tilde{x}$ occur bound in $(\nu \tilde{x})(M, \sigma)$. 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. The order of names in $(\nu \tilde{x})$ is irrelevant, i.e. we freely permute them. 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

$$(\nu \tilde{x})(M, \sigma_1) \longrightarrow (\nu \tilde{y})(N, \sigma_2)$$

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

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

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

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

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

$$\begin{array}{rcl}
(!x,\ \sigma) & \to & (\sigma(x),\ \sigma) \\
(x := V,\ \sigma) & \to & ((),\ \sigma[x \mapsto V]) \\
(\mathtt{new}\ x := V\ \mathtt{in}\ N, \sigma) & \longrightarrow & (\nu x)(N, \sigma \cup [x \mapsto V])
\end{array}$$

In the last rule, note V in $x := V$ is semi-closed by $x := V$ being semi-closed, by the definition of configurations. Also $x \notin \mathsf{dom}(\sigma)$ is assumed by the bound name convention. Finally we close $\longrightarrow$ under evaluation contexts and ν-binders.

$$\frac{(\nu\tilde{y})(M,\sigma) \to (\nu\tilde{z})(M',\sigma')}{(\nu\tilde{x}\tilde{y})(\mathcal{E}[M],\sigma) \to (\nu\tilde{x}\tilde{z})(\mathcal{E}[M'],\sigma')}$$

where $\tilde{x}$ are disjoint from both $\tilde{y}$ and $\tilde{z}$, $\mathcal{E}[\cdot]$ is the left-to-right evaluation context (with eager evaluation), inductively given by (omitting first-order operators):

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

Some notations:

- $(M,\sigma)\Downarrow$ stands for $(M,\sigma) \longrightarrow^* (\nu\tilde{x})(V,\sigma')$, for some $\tilde{x}$, V and σ'.
- $(M,\sigma)\Uparrow$ iff $(M,\sigma) \longrightarrow^n$ for each natural number n, that is iff $(M,\sigma)\Downarrow$ does not hold.

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

- $\mathtt{IncShared}' \stackrel{\text{def}}{=} b := !a; z_1 := (!a)(); z_2 := (!b)(); (!z_1) + (!z_2)$,
- $\mathtt{IncShared}'' \stackrel{\text{def}}{=} z_1 := (!a)(); z_2 := (!b)(); (!z_1) + (!z_2)$, and
- $\mathtt{IncShared}''' \stackrel{\text{def}}{=} z_2 := (!b)(); (!z_1) + (!z_2)$.

Further we set $W \stackrel{\text{def}}{=} \lambda().x := !x+1; !x$. 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).

$$
\begin{aligned}
& (\mathtt{IncShared},\ \emptyset) \\
& \longrightarrow (\nu x)(a := W; \mathtt{IncShared}',\ \{x \mapsto 0,\ a,b,z_1,z_2 \mapsto \ldots\}) \\
& \longrightarrow (\nu x)(\mathtt{IncShared}',\ \{x \mapsto 0,\ a \mapsto W,\ b,z_1,z_2 \mapsto \ldots\}) \\
& \longrightarrow (\nu x)(b := W; \mathtt{IncShared}'',\ \{x \mapsto 0,\ a \mapsto W,\ b,z_1,z_2 \mapsto \ldots\}) \\
& \longrightarrow (\nu x)(\mathtt{IncShared}'',\ \{x \mapsto 0,\ a,b \mapsto W,\ z_1,z_2 \mapsto \ldots\}) \\
& \longrightarrow (\nu x)(z_1 := 1; \mathtt{IncShared}''',\ \{x \mapsto 1,\ a,b \mapsto W,\ z_1,z_2 \mapsto \ldots\}) \\
& \longrightarrow (\nu x)(\mathtt{IncShared}''',\ \{x \mapsto 1,\ a,b \mapsto W,\ z_1 \mapsto 1,\ z_2 \mapsto \ldots\}) \\
& \longrightarrow (\nu x)(z_2 := 2; (!z_1)+(!z_2),\ \{x \mapsto 1,\ a,b \mapsto W,\ z_1 \mapsto 1,\ z_2 \mapsto \ldots\}) \\
& \longrightarrow (\nu x)((!z_1)+(!z_2),\ \{x \mapsto 1,\ a,b \mapsto W,\ z_1 \mapsto 1,\ z_2 \mapsto 2\}) \\
& \longrightarrow (\nu x)(3,\ \{x \mapsto 1,\ a,b \mapsto W,\ z_1 \mapsto 1,\ z_2 \mapsto 2\})
\end{aligned}
$$

Observe x 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:

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

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

$$
\begin{aligned}
\mathsf{car}(l) &\stackrel{\text{def}}{=} \pi_l(l) \\
\mathsf{cdr}(l) &\stackrel{\text{def}}{=} \pi_r(l)
\end{aligned}
$$

Note types match. The following program is similar to the one treated by Burstall [9].

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

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

$$
\begin{aligned}
& (L,\ \{x,y \mapsto \ldots\}) \\
& \longrightarrow^* \ (\nu\, hlh'l')(\mathsf{car}(!x) := 1,\ \{h \mapsto 0,\ h' \mapsto 2,\ l \mapsto (h',\mathsf{nil}),\ l' \mapsto (h',l),\ x \mapsto l,\ y \mapsto l'\}) \\
& \longrightarrow^* \ (\nu\, hlh'l')(h := 1,\ \{h \mapsto 0,\ h' \mapsto 2,\ l \mapsto (h',\mathsf{nil}),\ l' \mapsto (h',l),\ x \mapsto l,\ y \mapsto l'\}) \\
& \longrightarrow \ \ (\nu\, hlh'l')((),\ \{h \mapsto 1,\ h' \mapsto 2,\ l \mapsto (h',\mathsf{nil}),\ l' \mapsto (h',l),\ x \mapsto l,\ y \mapsto l'\})
\end{aligned}
$$

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

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

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

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

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

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

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

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

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

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

2.2 Typing

A *basis* $\Gamma;\Delta$ is a pair of finite maps, one from names to non-reference types ($\Gamma, \ldots$, called *environment basis*) and the other from names 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 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-referen type. The typing rules are standard [36], which we list in Figure 1 for reference (from first-order operations we only list two basic ones). In the first rule of Figure 1, 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* semi-closed *if* $\mathsf{dom}(\Gamma) = \emptyset$*. It is* closed *if* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Delta) = \emptyset$.

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:

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

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

We then type a configuration by the following rule:

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

The following is standard [13, 36].

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

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

2.3 Observational Congruence and Distinction

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 *semi-closing* if it does not λ or ν-abstract reference names and the resulting program is semi-closed. 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 $\Gamma;\Delta \vdash M_1 \cong M_2 : \alpha$, or often simply $M_1 \cong M_2$ leaving type information implicit, if, for each closing $C[\ \cdot\ ]^{\mathsf{Unit}}_{\Gamma;\Delta;\alpha}$, the following condition holds:

$$(C[M_1],\ \emptyset) \Downarrow \quad \equiv \quad (C[M_2],\ \emptyset) \Downarrow$$

The relation $\cong$ is an equivalence relation on typed programs and is closed under typing rules, including weakening. By considering arbitrary typed contexts, $\cong$ detects differences between two programs with respect to arbitrary aliasing.

Example 2.8 ($\cong$ and distinction) Take the following program:

$$M \stackrel{\text{def}}{=} \mathtt{if}\ x^{\mathsf{Ref(Nat)}} = y^{\mathsf{Ref(Nat)}}\ \mathtt{then}\ 0\ \mathtt{else}\ 1$$

Then M behaves as 0 under the assumption that x and y are aliased, and behaves as 1 if they are distinct. In $\cong$, it is not equal to either 0 or 1, since both possibilities are being considered.

When we assert properties of programs, this fine-grained nature of $\cong$ is often too restrictive. For example, we may wish to say that M above has the property that, *if* (hypothetically) x and y are aliased, then M is the same thing as 0. For enabling such assertions, it is necessary to consider a family of logical equivalences paramterised by arbitrary aliasing situations. A simple tool for doing so, introduced in [5], is *distinction*, originating in the semantics of the π-calculus [28].

Definition 2.9 (distinction and identicals)

1. A *distinction over* Δ, ranged over by $\mathcal{D}, \mathcal{D}', \ldots$, is a type-respecting equivalence over $\mathsf{dom}(\Delta)$. The equivalence classes of a distinction are called *identicals* $(\mathbf{i}, \mathbf{j}, \ldots)$.
2. Let $\Delta \vdash M : \alpha$ and let $\mathcal{D}$ be a distinction over Δ. Regarding identicals of $\mathcal{D}$ as names, we can substitute a $\mathcal{D}$-identical $\mathbf{i}$ for each name $x \in \mathbf{i}$ in M, which we write $M\mathcal{D}$.

Intuitively, a distinction $\mathcal{D}$ on Δ specifies, for each pair of names in $\mathsf{dom}(\mathcal{D})$, whether they are aliased or distinct. $M\mathcal{D}$ is a program whose names are coalesced following $\mathcal{D}$. $M\mathcal{D}$ is typed by $\Delta\mathcal{D}$, which is defined similarly.

For example, recall $M \stackrel{\text{def}}{=} \mathtt{if}\ x = y\ \mathtt{then}\ 0\ \mathtt{else}\ 1$ we considered above. If $\mathcal{D}$ only equates x and y then we have

$$M\mathcal{D} \stackrel{\text{def}}{=} \mathtt{if}\ \mathbf{i} = \mathbf{i}\ \mathtt{then}\ 0\ \mathtt{else}\ 1 \qquad (\text{assuming } \mathbf{i} \stackrel{\text{def}}{=} \{x, y\}),$$

Note that $M\mathcal{D}$ reduces as $(M\mathcal{D}, \sigma) \longrightarrow (0, \sigma)$ which is quite different from M itself, showing that distinctions affect observable behaviour. By adjusting distinctions, we can capture a programs' behaviour under arbitrarily chosen aliasing situation.

By incorporating distinction into semantics, we can discuss properties of programs under arbitrary hypothetical distinctions considered. This is done by introducting distinction-respecting contextual congruence [5, 28], as follows.

Definition 2.10 (distinction-sensitive observational congruence) Let $\mathcal{D}$ be a distinction over Δ and assume $\Gamma; \Delta \vdash M_{1,2} : \alpha$. Then we set:

$$\Gamma; \Delta \vdash M_1 \cong_{\mathcal{D}} M_2 : \alpha \quad \text{iff} \quad ((C[M_1\mathcal{D}], \sigma) \Downarrow \text{ iff } (C[M_2\mathcal{D}], \sigma) \Downarrow)$$

for all semi-closing $C[\cdot]^{\mathsf{Unit}}_{\Gamma;\Delta;\alpha}$ and well-typed stores σ.

Intuitively, $\cong_{\mathcal{D}}$ equates programs assuming all names equated in $\mathcal{D}$ are aliased, and no others. We can check:

Proposition 2.11 *$M_1 \cong_{\mathcal{D}} M_2$ iff $M_1\mathcal{D} \cong_{\mathbf{ID}_{\mathcal{D}}} M_2\mathcal{D}$ where $\mathbf{ID}_{\mathcal{D}}$ is the identity distinction on identicals from $\mathcal{D}$.*

PROOF: Direct from the definition. □

Further we observe:

Proposition 2.12 *$\Gamma; \Delta \vdash M_1 \cong M_2 : \alpha$ iff $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{D}} M_2 : \alpha$ for each distinction $\mathcal{D}$ over Δ.*

PROOF: (outline) If $\Gamma;\Delta \vdash M_1 \cong M_2 : \alpha$, then by congruent closure, we can make a context which coalesces Δ following an arbitrarily $\mathcal{D}$. Using the resulting programs, the condition for $\cong_{\mathcal{D}}$ is satisfied. For the other direction, we analyse the defining context in Definition 2.7. Without loss of generality we can consider $C[\cdot]$ λ-abstracts all reference names in Δ: each such name is either never instantiated or instantiated into a name which in effect comes from a newly generated name. This defines aliasing over Δ, which can be represented by a distinction combined with appropriate initialisation, representable in the context in Definition 2.10. $\square$

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 [24, § 2.8], extended with assertions for stateful evaluation [19, 20] (for imperative higher-order functions) and quantifications over store content [5] (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 $Q \in \{\forall, \exists\}$, highlighting the sole change from [5].

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

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

The logical language uses the standard logical connectives and quantification [24]. We include, following [5, 19], 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 [20], which intuitively says:

> *If we apply a function e to an argument e' starting from an initial state satisfying C, then it terminates with a resulting value (name it x) and a final state together satisfying C'.*

$[!e]C$ and $\langle !e \rangle C$ are called *universal/existential content quantifications*, respectively, introduced and studied in [5]. 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 [5].

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 $y \sharp x$ [11, 39], 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.

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 $\mathsf{fv}(C) \cap \{\tilde{x}\} = \emptyset$. 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 [5, 19, 20] 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 $\lambda x.x + x$ 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* [5], 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.

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

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 [5], 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 (1): Basic Ideas

We first illustrate key ideas informally. Recall $\mathtt{Inc} \stackrel{\text{def}}{=} \mathtt{new}\, x := 0\ \mathtt{in}\ \lambda().(x := !x+1;\, !x)$ from Introduction. When we run this program, a new reference x has been generated with content 0, accessible by an internal procedure. We observe:

1. This x should be "fresh", i.e. distinct from all existing names.
2. This x should be "hidden", i.e. invisible to the observer.

A possible idea to describe the situation would be to consider a quantifier which combines these two functionalities. Let's write this quantifier $\nu x.C$ [28], which is purported to indicate both of the properties noted above. Then we may assert:

$$\nu x.(\,!x = 0 \ \wedge \ \mathsf{inc}(u,x)) \tag{3.9}$$

where we set $\mathsf{inc}(u,x) \stackrel{\text{def}}{=} \forall i.\{!x = i\}u \bullet () = z\{!x = z = i+1\} @ x$, which is an assertion for $N \stackrel{\text{def}}{=} \lambda().(x = !x+1; !x)$, named u.

We then inspect the following alternative presentation of $\mathtt{Inc}$, mentioned in (1.4).

$$\mathtt{IncLet} \stackrel{\text{def}}{=} \mathtt{let}\, x = \mathtt{ref}(0)\ \mathtt{in}\, N \tag{3.10}$$

The state after evaluating $\mathtt{ref}(0)$ would be asserted as, with an informal understanding that $\mathsf{fresh}(x)$ says "x is fresh":

$$!x{=}0 \ \wedge \ \mathsf{fresh}(x) \tag{3.11}$$

Note that (3.11) has no binding, which is faithful to the behaviour of $\mathtt{ref}(0)$: no sooner than generating a reference, the program exports it as a return value, so that the receiver can fully access it.

Now we recover νx (locality) out of $\mathsf{fresh}(x)$ (freshness). By using the standard let rule [5, 19, 20], $\mathtt{IncLet}$ is given the following assertion as its post-condition:

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

which reads:

> *There exists a reference we name x whose content is* 0 *and which is fresh w.r.t. existing names, and that u behaves as described in* $\mathsf{inc}(u,x)$.

Since this description is precisely what we want the ν-quantifier in (3.9) to capture, we have arrived at the following decomposition:

$$\nu x.C \stackrel{\text{def}}{=} \exists x.(C \wedge \mathsf{fresh}(x)) \tag{3.12}$$

But one question still remains: what is $\mathsf{fresh}(x)$? Since to say it is distinct from anything which is not x is just tautology, it should mean it is distinct from anything in the initial state of a judgement $\{C\}\ M :_u \{C'\}$. Just as we have encoded located assertions into inequality of names, we translate $\mathsf{fresh}(x)$ as

$$x \neq i^{\mathsf{Ref}(\mathsf{X})}, \tag{3.13}$$

where i is a fresh auxiliary variable, saying, when asserted in the post-condition, x is distinct from any existing names in the initial state.[3]

The reading above is possible because auxiliary names are implicitly universally quantified. In spite of this implicit universality, x cannot be substituted for i by the existential binding: by this existential, x indicates the name of a newly added reference which has not existed in the pre-condition. As we shall see later when we discuss proof rules, this framework for representing locality and freshness of a reference name works smoothly in compositional logical reasoning.

We thus need neither the ν-binding nor $\mathsf{fresh}(x)$ as added constructs: the standard logical primitives can represent them. The need for the predicate $e \hookrightarrow e'$, or its negation, $e' \,\sharp\, e$, arises from a deeper degree of freshness we need to assert when, for example, a reference has just been freshly generated. We shall demonstrate their use with examples in the next subsection.

3.4 Assertions for Local State (2): Examples

Concrete examples of assertions for local state follow.

1. Recall `IncShared` from the Introduction. Starting from T as precondition, the post-condition at each step changes as follows, using the ν-notation (understood as (3.12)) for clarity:
   i. $\nu x.(!x{=}0 \wedge \mathsf{inc}(!a,x))$ (created).
   ii. $\nu x.(!x{=}0 \wedge \mathsf{inc}(!a,x)) \wedge \mathsf{inc}(!b,x))$ (gets shared).
   iii. $\nu x.(!z_1{=}1 \wedge z_1 \neq z_2 \wedge x \neq z_{1,2} \wedge !x{=}1 \wedge \mathsf{inc}(!a,x) \wedge \mathsf{inc}(!b,x))$.
   iv. $\nu x.(!z_2{=}2 \wedge z_1 \neq z_2 \wedge x \neq z_{1,2} \wedge !x{=}2 \wedge \mathsf{inc}(!a,x) \wedge \mathsf{inc}(!b,x))$.
   v. (returns $!z_1 + !z_2$ which is 3).

   The inequality between x and $z_{1,2}$ is guaranteed by ν-binder, as will be clarified in Section 7.1.

[3] Semantically speaking, i denotes "an(y) *existing* name in the *initial* state" because an initial state, or a model representing it, should already interpret all free variables in pre/post-conditions, except for the anchor in the post-condition: thus i in $x \neq i$ should be interpreted in the pre-condition, which means a reference name (or a datum) i should already exist in the initial state. The precise semantic account is given in Section 5.1.

2. 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.
3. 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

$$w \,\sharp\, x,$$

which, as noted, stands for $\neg(x \hookrightarrow w)$. Note that $x \hookrightarrow x \equiv \mathsf{T}$ and $x \sharp x \equiv \mathsf{F}$, but $!x \hookrightarrow x$ may not be T and $x \sharp\, !x$ may not be F.
4. 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:

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

which is logically equivalent to $\forall \mathrm{X}, y^{\mathsf{Ref}(\mathrm{X})}.(y \,\sharp\, e_1 \vee y \,\sharp\, e_2)$. 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.
5. Consider the reference $\mathtt{ref}(M) \stackrel{\text{def}}{=} \mathtt{new}\ x := M\ \mathtt{in}\ x$. Then the program $\lambda n.\mathtt{ref}(n)$ meets

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

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:

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

6. Finally we consider a factorial program which realise a recursion by circular references, the idea due to Landin [22].

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

This program returns the factorial of n. In [20], we have shown we can derive the following post-condition for the above program (with an initial state, say T).

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

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 $\mathtt{circFact}$, x occurs free. In the present language, we can further hide x.

$$\mathtt{new}\, x := W \;\mathtt{in}\; (\mathtt{circFact}; !x)$$

Above W is an arbitrary closed program of type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ (W is needed since we cannot initialise x with a value in which x is free). The program evaluates to a procedure which also calculates the factorial of n: but x is now invisible and inaccessible from the outside, so that the program as a whole behaves as a pure factorial function, satisfying:

$$\forall n. u \bullet n \searrow n! \tag{3.14}$$

We derive (3.14) from $\{\mathsf{T}\}\ \mathtt{circFact} :_u \{\mathsf{circfact}(x)\}$ in Section 7.5.

Further examples of assertions are found in Section 5.

3.5 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 $\sharp$ instead of simple inequality. Below z again should be typed with a reference type.

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

Note z is bound in the whole formula because it occurs as the result name in the evaluation formula. In contrast, $\sharp z. C'$ does*not* induce binding on z, as can be seen from the encoding. It is notable that $\{C\} e \bullet e' = z \{\sharp x. C'\}$ (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.

$$\{C\} e \bullet e' = z \{\star z. C'\} \stackrel{\text{def}}{=} \forall \mathrm{X}, i^{\mathrm{X}}. \{C\} e \bullet e' = z \{z \star i \wedge C'\} \tag{3.15}$$

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.

$$\{C\} e \bullet e' = z \{\nu \sharp x. C'\} \quad \stackrel{\text{def}}{=} \quad \forall \mathrm{X}, i^{\mathrm{X}}. \{C\} e \bullet e' = z \{\exists x. (x \sharp i \ \wedge \ C')\} \tag{3.16}$$

Fig. 2 Axioms for Reachability (finite case: all mentioned types are finite)

(Fin1)	$x^{\mathsf{Unit}} \hookrightarrow y^{\mathsf{Ref}(\beta)}$	$\equiv$	F
(Fin2)	$x^{\mathsf{Bool}} \hookrightarrow y^{\mathsf{Ref}(\beta)}$	$\equiv$	F
(Fin3)	$x^{\mathsf{Nat}} \hookrightarrow y^{\mathsf{Ref}(\beta)}$	$\equiv$	F
(Fin4)	$x^{\alpha_1 \times \alpha_2} \hookrightarrow y^{\mathsf{Ref}(\beta)}$	$\equiv$	$\exists x_{1,2}.(x = \langle x_1, x_2 \rangle \ \wedge\ \bigvee_{i=1,2} x_i \hookrightarrow y)$
(Fin5)	$x^{\alpha_1 + \alpha_2} \hookrightarrow y^{\mathsf{Ref}(\beta)}$	$\equiv$	$\exists x'.(\bigvee_{i=1,2} x = \mathtt{in}_i(x') \ \wedge\ x' \hookrightarrow y)$
(Fin6)	$x^{\mathsf{Ref}(\alpha)} \hookrightarrow y^{\mathsf{Ref}(\beta)}$	$\equiv$	$x = y \ \vee\ !x \hookrightarrow y$

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:

$$\{C\}e \bullet e' = z\{\nu \star x.C'\} \quad \stackrel{\text{def}}{=} \quad \forall \mathrm{X}, i^{\mathrm{X}}.\{C\}e \bullet e' = z\{\exists x.(x \star i \ \wedge\ C')\} \tag{3.17}$$

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.

3.6 Axioms for Reachability

In this subsection we explore axiomatisation 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).

The axioms for the first two constructs are respectively treated in [20] and [5], including their interplay with each other. This subsection offers a basic axiomatic treatment of reachability predicate and its negation, restricting our attention to reachability from a datum whose type does not include function types (however we do consider recursive types), which is already quite useful in practice. These two sets of axioms will be shown to be complete in Section 6. Axiomatisation of general cases is more subtle, which will be discussed in later versions.

First we treat reachability predicates in the system without recursive types (in fact the technical development for recursive types is essentially identical).

Definition 3.1 (finite types) α is *finite* if it contains neither an arrow type nor (free/bound) type variable.

In this case, $y \hookrightarrow x$ can be simply axiomtised as in Figure 2: further these axioms are sound even if $\alpha_{1,2}$ and β are arbitrary. However, if involved types are not finite, the

Fig. 3 Axioms for (Non-)Reachability (recursive finite case: α, β are recursively finite)

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

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

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

Symmetrically

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

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

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

given axioms may not completely characterise the reachability. Symmetrically, $x \sharp y$ is axiomtised by the negation of the right-hand side of the given logical equivalences, e.g.

$$y^{\mathsf{Ref}(\beta)} \sharp x^{\alpha_1 \times \alpha_2} \qquad \equiv \qquad \forall x_{1,2}.(x = \langle x_1, x_2 \rangle \ \supset \ \bigwedge_{i=1,2} y \sharp x_i)$$

Again these axioms are valid even when arbitrary types are involved.

When we have recursive types, we first extend the class of types considered, both for programs and logic, as follows.

$$\alpha \quad ::= \quad \ldots \quad | \quad \mu X.\alpha$$

Taking the equi-isomorphic approach [36], there is no need to change either proof rules nor the semantics of the logic [19]. With recursive types, finiteness is reformulated as follows.

Definition 3.2 (recursively finite types) α is *recursively finite* when it is closed, does not contain any quantification, and it does not contains any arrow type.

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 (x,y) 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:

$$C \quad ::= \quad \dots \quad | \quad (x,y) \in \mathcal{R}$$

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 (x,y) 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 3.

The axiomatisation can easily be extended to other data types as far as function types are not involved. For example, in Section 7, we shall use a user-defined data type, a table representing a finite function from a reference to a reference (whose types do not contain this table type). For this table data type to be incorporated into the axiomatisation of $\sharp$, we can simply add the following additional clause to the condition for $\Phi(\mathcal{R})$ in in Figure 3. Below we write TBL for the type of the table.

$$\langle z^{\mathsf{TBL}}, y\rangle \in \mathcal{R} \quad \supset \quad \forall g, g'.(\langle g, g'\rangle \in z \supset (\langle g, y\rangle \in \mathcal{R} \ \wedge \ \langle g', y\rangle \in \mathcal{R}))$$

In a later version we shall discuss axiomatisation for non (recursively) finite types. Also see Section 6 for an alternative axiomatisation of finite and recursively finite types (a simple example is treated in the next remark).

Remark 3.3 (content quantification and reachability) Given $e_1^\alpha \hookrightarrow e_2$ or $e_1 \sharp e_2$, 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 $\sharp$) into its witness, and check how content quantification interacts with each (in)equation, using the standard axioms for content quantifiers [5]. If however 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.

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x. j \sharp x) \quad \supset \quad (\langle !x\rangle i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)} \ \equiv \ [!x] i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)} \ \equiv \ i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)}) \tag{3.18}$$

Equivalently, under the same assumption:

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x. j \sharp x) \quad \supset \quad (\langle !x\rangle x^{\mathsf{Ref}(\beta)} \sharp i^\alpha \ \equiv \ [!x] x^{\mathsf{Ref}(\beta)} \sharp i^\alpha \ \equiv \ i^\alpha \hookrightarrow x^{\mathsf{Ref}(\beta)} \tag{3.19}$$

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 (3.18) and (3.19) hold can be easily understood. By the assumption $\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x. j \sharp x$, 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 (3.18) and (3.19) can be found in Appendix.

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[5, 18–20]. For clear presentation, we take three steps to introduce them.

Step 1: We first build *open models*, which capture imperative higher-order functions [20] and aliasing [5]. Open models directly come from [5].
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 [5, 18–20], 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. Recall below a typable program M is said to be *semi-closed* if it can be typed as $\emptyset;\Delta \vdash M : \alpha$, i.e. all free variables of M are references.

Definition 4.1 (open models) *An* open model of type $\Gamma;\Delta;\mathcal{D}$*, written* $\mathcal{M}^{\Gamma;\Delta;\mathcal{D}}$*, with* $\mathcal{D}$ *a distinction over* Δ*, is a tuple* (ξ,σ) *where:*

- ξ*, called* environment*, is a finite map of the shape* $\xi_0 \cup \underline{\mathcal{D}}$*, where* ξ_0 *is a map from* $\mathsf{dom}(\Gamma)$ *to semi-closed values such that, for each* $x \in \mathsf{dom}(\Gamma)$*,* $\xi_0(x)$ *is typed as* $\Gamma(x)$ *under* $\Delta\mathcal{D}$*.*
- σ*, called* store*, is a finite map from the identicals of* $\mathcal{D}$ *to semi-closed values such that, for each* $\mathbf{i} \in \mathcal{D}$ *with* $(\Delta\mathcal{D})(\mathbf{i}) = \mathsf{Ref}(\alpha)$*,* $\sigma(\mathbf{i})$ *is typed as* α *under* $\Delta\mathcal{D}$*.*

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

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

may be interpreted using the following open model.

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

Above we set $\mathbf{i} = \{x\}$. We can the interpret indentifiers, 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 identiers 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.

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

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.

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

(4.3) says that there is a behaviour named u and a reference named x which is not aliased with any other names (recall we have set $\mathbf{i} = \{x\}$), that this reference stores 0, and that the name x is hidden. Based on (4.3), we may assert:

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

to which (if this is the post condition) we may as well add $\mathsf{fresh}(x)$. This gives us the following concrete notion of models.

Definition 4.3 (models) *A* model *of type* $\Gamma;\Delta;\mathcal{D}$*, with* $\mathcal{D}$ *being a distinction over* Δ*, is a structure*

$$(\nu\Delta')(\xi,\sigma)$$

where (ξ,σ) *is an open model of type* $\Gamma;\Delta{\cdot}\Delta';\mathcal{D}'$ *such that* $\mathcal{D}$ *is the restriction of* $\mathcal{D}'$ *onto* Δ *(i.e. for each* $x,y \in \mathsf{dom}(\Delta)$*, we have* $x\mathcal{D}y$ *iff* $x\mathcal{D}'y$*).* $\mathcal{M}, \mathcal{M}', \ldots$ *range over models.*

In $(\nu\Delta')(\xi,\sigma)$, names in $\mathsf{dom}(\Delta')$ act as binders, which gives standard α-equality on models.

4.3 Abstract Models.

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

$$(\nu xy)(\{u : \lambda().(x := !x+1; y := !y+1;\, (!x + !y)/2),\;\; x : \mathbf{i},\;\; y : \mathbf{j}\},\;\; \{\mathbf{i},\mathbf{j} \mapsto 0\}) \tag{4.5}$$

where $\mathbf{i} = \{x\}$ and $\mathbf{j} = \{y\}$. 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

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

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 distinction-sensitive behavioural equivalence $\cong_{\mathcal{D}}$, introduced in Section 2.3, Page 11.

For applying $\cong_{\mathcal{D}}$ to models, we first need to turn *a model $\mathcal{M}$ into behaviour* via encoding. The underlying idea is that we translate "ν" in the model into "new", preserving the distinction given in $\mathcal{M}$. As an example, the model (4.3) is encoded into:

$$\mathtt{new}\ \mathbf{i} := 0\ \mathtt{in}\ \lambda().(\mathbf{i} := !\mathbf{i} + 1;\ !\mathbf{i})$$

Below we give a formal definition, after a convention.

Convention 4.4 Henceforth we regard a basis (both environment and reference) as an ordered sequence of name-type pairs as well as being a finite map.

Definition 4.5 (encoding) *Let $\mathcal{M}^{\Gamma;\Delta;\mathcal{D}} = (\nu\tilde{x})(\xi,\sigma)$ be a model such (ξ,σ) is typable by $\Gamma;\Delta,\Delta';\mathcal{D}$. The* behavioural encoding of *$\mathcal{M}$, written $[\![\mathcal{M}]\!]$, is given as:*

$$\mathtt{new}\ \tilde{\mathbf{i}}\ \mathtt{in}\ (\tilde{\mathbf{j}} := \sigma(\tilde{\mathbf{j}}); \langle(\xi\mathcal{D})\rangle^{\Gamma}$$

This expression uses the following of abbreviations.

- *Multiple assignment $\tilde{x} := \tilde{e}$ is short for $x_1 := e_2;...;x_n := e_n$. $\mathtt{new}\ \tilde{\mathbf{i}}\ \mathtt{in}\ \cdots$ stands for initialisation of $\tilde{\mathbf{i}}$ with arbitrary values of appropriate types, which is needed for the same reason as in Landin's factorial, cf. §3.4 (6) in the main section.*
- *$\tilde{\mathbf{j}}$ are all the identicals from $\mathcal{D}$. On the other hand, $\tilde{\mathbf{i}}$ are all those $\mathcal{D}$-identicals such that, for each $\mathbf{i} \in \{\tilde{\mathbf{i}}\}$, $x \in \mathbf{i}$ implies $x \in \mathsf{dom}(\Delta')$, i.e. each identical in $\tilde{\mathbf{i}}$ is* fully *hidden in $\mathcal{M}$.*
- *Finally, $\langle\xi\mathcal{D}\rangle^{\Gamma}$ stands for $\langle\xi(y_1)\mathcal{D},...,\xi(y_m)\mathcal{D}\rangle$ where $y_1..y_m = \mathsf{dom}(\Gamma)$ (regarding Γ as a sequence of typed names: note we disregard $\underline{\mathcal{D}}$-part of ξ, cf.Definition 4.1).*

Remark 4.6 We may regard this translation as replacing access to functional values via names (in concrete models) by position-based access (in their encoding). This is reflected in typing:

$$\emptyset;\Delta\mathcal{D} \vdash [\![M^{\Gamma;\Delta;\mathcal{D}}]\!] : \tau_1 \times ... \times \tau_m$$

where we assume $\Gamma \cdot \Delta = y_1 : \tau_1 \cdots y_m : \tau_m$. Conversely, given $\Gamma;\Delta;\mathcal{D}$ and any terminated configuration typed as above, we can always obtain, uniquely up to aliasing in $\mathcal{D}$, a model whose encoding converges to that configuration.

We now introduce the equivalence on models. Below id denotes the identity distinction.

Definition 4.7 *Given models $\mathcal{M}_i^{\Gamma;\Delta;\mathcal{D}}$ for $i = 1,2$ we set*

$$\Gamma;\Delta;\mathcal{D} \vdash \mathcal{M}_1 \approx \mathcal{M}_2 \qquad \textit{iff} \qquad [\![\mathcal{M}_1]\!] \cong_{\mathsf{id}_{\mathcal{D}}} [\![\mathcal{M}_2]\!]$$

where $\mathsf{id}_{\mathcal{D}}$ is the identity distinction on identicals from $\mathcal{D}$. The $\approx$-congruence classes are called abstract models.

Abstract models are the semantic basis of the present logic, while concrete models offer a convenient medium to work with.

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 4.8 (expansion) *Let* $M^{\Gamma;\Delta;\mathcal{D}} \stackrel{def}{=} (\nu\tilde{x})(\xi,\sigma)$. *Assuming* $\Gamma;\Delta \vdash N : \alpha$ *and* u *fresh we set*

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

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

$$\mathcal{M}[u\!:\!N] \Downarrow \mathcal{M}'$$

when $(N\xi,\sigma) \Downarrow (\nu\tilde{y})(V,\sigma')$ *and* $\mathcal{M}' = (\nu\tilde{x}\tilde{y})(\xi\cdot\tilde{y} : \tilde{\mathbf{i}}\cdot u : (V[\tilde{\mathbf{j}}/\tilde{y}]),\sigma')$ *where we set* $\mathbf{i}_j = \{y_j\}$ *for each* y_j *in* $\tilde{y}$ *except if* $V = y_j$ *then* $\mathbf{i}_j = \{y_j, u\}$. *By determinacy of* $\Downarrow$, $\mathcal{M}'$ *is uniquely determined, should* $\mathcal{M}[u\!:\!N]$ *converge. If not, we write* $\mathcal{M}[u\!:\!N]\Uparrow$.

If (the interpretation of) N in $\mathcal{M}[u\!:\!N]$ converges without affecting $\mathcal{M}$, then we can regard $\mathcal{M}[u\!:\!N]$ as $\mathcal{M}$ augmented with a new entry N at u. This motivates the following construction.

Definition 4.9 (gentle programs) *We call* N gentle in $\mathcal{M}$ *if* $\mathcal{M}[u\!:\!N] \Downarrow \mathcal{M}'$ *implies* $\mathcal{M} \approx \mathcal{M}'/u$. *Here* $\mathcal{M}/u$ *is the projection of* $\mathcal{M}$ *taking off only the* u*-component. We let* $\mathcal{G}(\mathcal{M})$ *stand for the set of gentle programs for* $\mathcal{M}$.

Notation 4.10 If $N \in \mathcal{G}(\mathcal{M})$ and u is fresh, then we often let $\mathcal{M}[u\!:\!N]$ denote the model resulting from $\mathcal{M}[u\!:\!N]$.

Example 4.11 (gentleness)

1. Logical terms e as programs are always gentle.
2. With $\mathcal{M}$ being the model in (4.3), $(\lambda x.\lambda y.!x+y)x$ is gentle:

$$\begin{array}{c}\mathcal{M}[v\!:\!(\lambda x.\lambda y.!x+y)x]\\ \Downarrow \\ \{u : \lambda().(x := !x+1;\, !x), v : \lambda y.!x+y, x : \mathbf{i}\}, \{\mathbf{i} \mapsto 0\}\end{array}$$

3. A gentle program can read hidden memory:

$$\begin{array}{c}\mathcal{M}[v\!:\!(\lambda x.\lambda y.x+y)!x]\\ \Downarrow \\ \{u : \lambda().(x := !x+1;\, !x),\ v : \lambda y.0+y,\ x : \mathbf{i}\},\ \{\mathbf{i} \mapsto 0\}\end{array}$$

   Even writing is allowed, provided there is no observable effect. Let $\mathcal{M}'$ be $\{u : \lambda y.(!x)y, x : \mathbf{i}\}$, $\{\mathbf{i} \mapsto \lambda n.n+1\}$. Now we update $\mathcal{M}'$ with something that overwrites the content of the hidden reference.

$$\begin{array}{c}\mathcal{M}'[v\!:\!x := \lambda m.(2m+2)/2]\\ \Downarrow \\ \{u : \lambda y.(!x)y,\ v : (),\ x : \mathbf{i}\},\ \{\mathbf{i} \mapsto \lambda m.(2m+2)/2\}\end{array}$$

   But $x := \lambda m.(2m+2)/2$ is gentle for $\mathcal{M}'$, as no context can distinguish $\lambda m.(2m+2)/2$ from $\lambda m.m+1$.

Definition 4.12 (update) *Let* $M^{\Gamma;\Delta;\mathcal{D}} \stackrel{def}{=} (\nu\tilde{x})(\xi,\sigma)$ *and e and N be respectively typed as* $\mathsf{Ref}(\alpha)$ *and* α *under* $\Gamma;\Delta$

$$\mathcal{M}[e \mapsto N] \stackrel{def}{=} (\nu\tilde{x})(\xi,\sigma[e\xi \mapsto N\xi])$$

The right-hand side of this definition is not a model, but induces a model (cf. Definition 4.8), as follows. We first evaluates $e\xi$ *to a reference, then evaluates* $N\xi$ *and, if it converges, assigns the latter to the former. If* $\mathcal{M}'$ *is the resulting model (which is unique up to* α*-conversion), we write* $\mathcal{M}[e \mapsto N] \Downarrow \mathcal{M}'$*. We often assume* $N \in \mathcal{G}(\mathcal{M})$*, in which case we write* $\mathcal{M}[e \mapsto N]$ *for denoting* $\mathcal{M}'$ *such that* $\mathcal{M}[e \mapsto N] \Downarrow \mathcal{M}'$ *Further We set:*

$$(!e)\mathcal{M} \stackrel{def}{=} \{\mathcal{M}[e \mapsto N] \mid N \in \mathcal{G}(\mathcal{M})\}$$

which are all variants of $\mathcal{M}$ *w.r.t. the content of a reference e.*

Remark 4.13 With e, ξ and σ typed as in Definition 4.12, we have $e\xi \in \mathsf{dom}(\sigma)$ by subject reduction.

Remark 4.14 The set $(!e)\mathcal{M}$ contains all variants of $\mathcal{M}$ that can be obtained from $\mathcal{M}$ by updating it's store at (the denotation of) e with any gentle N. It is used for defining the validity of content quantificatication later.

The following predicate is needed to interpret (standard) universal/existential quantifiers.

Definition 4.15 (extension) *Let* $M^{\Gamma;\Delta;\mathcal{D}} \stackrel{def}{=} (\nu\tilde{x})(\xi,\sigma)$*. Assume* $\mathcal{M}'$ *has a type expanding that of* $\mathcal{M}$ *with a fresh* $y : \alpha$*. Then* $\mathcal{M} \leq_{y:\alpha} \mathcal{M}'$ *(read:* $\mathcal{M}'$ *extends* $\mathcal{M}$ *with* $y : \alpha$*) is given as follows. First, when* α *is a value type:*

$$\mathcal{M} \leq_{y:\alpha} \mathcal{M}' \quad \stackrel{def}{\equiv} \quad \exists N \in \mathcal{G}(\mathcal{M}).\, \mathcal{M}' \cong \mathcal{M}[y:N].$$

Second, when α *is a reference type:*

$$\mathcal{M} \leq_{y:\alpha} \mathcal{M}' \quad \stackrel{def}{\equiv} \quad \mathcal{M} \approx (\nu y)\mathcal{M}'$$

where $(\nu y)\mathcal{M}'$ *adds a* ν*-hiding at y.*

Remark 4.16 Distinct treatment of reference/non-reference types in $\leq_{x:\alpha}$ is already found in [5]. The definition above simply generalises the construction in [5] to models with local state.

We also use the following notion, which intuitively says some model can be possibly reached starting from another model and a possibly side-effecting program.

Definition 4.17 *Let two models* $\mathcal{M}_{1,2}$ *be of type* $\Gamma;\Delta;\mathcal{D}$*, with card*$(\mathsf{fv}(\Gamma)) = n$*. Then* $\mathcal{M}_2$ *is a* derivative *of* $\mathcal{M}_1$*, written* $\mathcal{M}_1 \rightsquigarrow \mathcal{M}_2$*, if, for some* $\Gamma;\Delta \vdash N : ()$ *and fresh u, we have* $\mathcal{M}_1[u:N] \Downarrow \mathcal{M}'$ *such that* $\mathcal{M}_2 = \mathcal{M}_2'/u$*.*

Proposition 4.18 $\rightsquigarrow$ *is a preorder.*

PROOF: Reflexivity is immediate. For transitivity, if $\mathcal{M}_1[u:L] \Downarrow \mathcal{M}'$ with $\mathcal{M}_2 = \mathcal{M}'/u$ and $\mathcal{M}_2[u:N] \Downarrow \mathcal{M}''$ with $\mathcal{M}_3 = \mathcal{M}''/u$ then $\mathcal{M}_1[u:(\lambda().N)L] \Downarrow \mathcal{M}''$, hence done. $\square$

Remark 4.19 One may consider $\mathcal{M} \rightsquigarrow \mathcal{M}'$ as denoting: "*$\mathcal{M}'$ is a possible world reachable from $\mathcal{M}$*". Here "possible" means some program can change the content of the store part of the original model. The change of store content is highly constrained in the sense that only "accessible" memory cells can be updated (for example a memory cell may be used by some procedure only internally, never get exported: such a cell cannot be updated except (if ever) invoking that procedure, for example).

Finally we define three predicates needed for interpreting $\hookrightarrow$ (or dually $\sharp$), after a notation.

Notation 4.20 (content projection) Given $\mathcal{M}^{\Gamma;\Delta;\mathcal{D}}$ with $w \in \mathsf{dom}(\Delta)$, $\mathcal{M} \upharpoonright w$ is given by, with fresh u, $\pi_{n+1}([\![\mathcal{M}[u:!w]]\!])$ where $card(\mathsf{dom}(\Gamma)) = n$. By abuse of notation, we also let $\mathcal{M} \upharpoonright w$ denote a(ny) corresponding model (with a distinguished fresh u).

Definition 4.21 *Write* clp^{α} *for the set of closed programs of type* α*, let* e *be of a reference type, and* $\mathcal{M}' \stackrel{def}{=} \mathcal{M}[w:e]$ *for a fresh* w.

1. *The predicate* $\mathsf{readable}(\mathcal{M}, e', e)$ *is given as:*

$$\begin{array}{l}\exists \mathcal{M}_{1,2} \in (!w)\mathcal{M}'.\exists V \in \mathsf{clp}^{\alpha\Rightarrow\mathsf{Unit}}.\ (\mathcal{M}_1[u:Ve'] \Downarrow \ \wedge\ \mathcal{M}_2[u:Ve'] \Uparrow)\ \vee \\ \exists V \in \mathsf{clp}^{\alpha\Rightarrow\mathsf{Ref}(\beta)}.(\mathcal{M}[u:\texttt{if}\ e = Ve'\ \texttt{then}\ ()\ \texttt{else}\ \Omega] \Downarrow)\end{array}$$

2. *The predicate* $\mathsf{writable}(\mathcal{M}, e', e)$ *is given as:*

$$\exists \mathcal{M}''.\exists V \in \mathsf{clp}^{\alpha\Rightarrow\mathsf{Unit}}.\ (\mathcal{M}[u:Ve'] \Downarrow \mathcal{M}''\ \wedge\ \mathcal{M}' \upharpoonright w \not\approx \mathcal{M}'' \upharpoonright w)$$

We write $\mathsf{readableC}(\mathcal{M}, e', e)$ *and* $\mathsf{matchable}(\mathcal{M}, e', e)$ *respectively for the first/second disjuncts of the defining formula of* $\mathsf{readable}(\mathcal{M}, e', e)$.

The predicate $\mathsf{readable}(\mathcal{M}, e', e)$ says we can reach a reference e from e' and read from it. $\mathsf{writable}(\mathcal{M}, e', e)$ says we can reach a reference e from e' and write to it. In most of the cases $\mathsf{matchable}(\mathcal{M}, e', e)$ is not necessary:

Proposition 4.22 *Assume* e *is typed as* $\mathsf{Ref}(\alpha)$ *under the typing of* $\mathcal{M}$*. Below* $\Pi\mathsf{Unit}$ *denotes an* n*-fold product of* Unit.

1. *If* α *is not of the shape* $\Pi\mathsf{Unit}$*,* $\mathsf{matchable}(\mathcal{M}, e', e)$ *implies* $\mathsf{readableC}(\mathcal{M}, e', e)$.
2. *If* $\alpha = \Pi\mathsf{Unit}$*, neither* $\mathsf{readableC}(\mathcal{M}, e', e)$ *nor* $\mathsf{writable}(\mathcal{M}, e', e)$ *holds.*

PROOF: (2) is immediate since the content of a reference typed $\mathsf{Ref}(\Pi\mathsf{Unit})$ is unique (an n-ary pair of ()). For (1), Let $\mathcal{M} \stackrel{\text{def}}{=} (\nu\tilde{z})(\xi, \sigma)$ and assume $\mathsf{matchable}(\mathcal{M}, e', e)$ holds. Writing $\mathcal{M}[\![e]\!]$ for the result of evaluating e in $\mathcal{M}$, let $\mathcal{M}[\![e]\!] = \mathbf{i}$ (an identical) and $\mathcal{M}[\![e']\!] = V'$. By definition we have:

$$(\texttt{if}\ \mathbf{i} = VV'\ \texttt{then}\ ()\ \texttt{else}\ \Omega,\ \sigma) \Downarrow$$

Since there are at most two semantically distinct total closed inhabitants at α, let $W_{1,2}$ be arbitrary such two and let $\sigma_{1,2}$ be the result of storing $W_{1,2}$ at $\mathbf{i}$ of σ. Since $W_1 \not\cong W_2$ there is a closing context C such that (say) $C[W_1] \Downarrow$ and $C[W_2] \Uparrow$. We can then take, assuming $\sigma(\mathbf{i}) = W$:

$$U \stackrel{\text{def}}{=} \lambda x.C[\texttt{let}\ y = !\mathbf{i}\ \texttt{in}\ (\mathbf{i} := W; \texttt{let}\ y' = !(Vx)\ \texttt{in}\ \mathbf{i} := y; y')]$$

Since $(VV', \sigma) \Downarrow \mathbf{i}$, we have:

$$(UV', \sigma_1) \longrightarrow^* (C[W_1], \sigma_1) \Downarrow \qquad (UV', \sigma_2) \longrightarrow^* (C[W_1], \sigma_2) \Downarrow$$

Thus, by taking $(\nu\tilde{z})(\xi, \sigma_{1,2})$ and U to be $\mathcal{M}_{1,2}$ and U in the clause for $\mathsf{readableC}(\mathcal{M}, e', e)$ in Definition 4.21 (1), we are done. $\square$

Remark 4.23 (reachability predicates) $\mathsf{readable}(\mathcal{M}, e', e)$ and $\mathsf{writable}(\mathcal{M}, e', e)$ are used for defining reachability. Rouhgly they say:

> *e being reachable from e' means either we can extract information from e by having only e'; or we can affect information at e by having only e'.*

The use of $\mathsf{matchable}(\mathcal{M}, e', e)$ as part of $\mathsf{readable}(\mathcal{M}, e', e)$ may deserve some comments. Proposition 4.22 shows that, for *all* reference types except those of the shape $\mathsf{Ref}(\Pi\mathsf{Unit})$, we can dispense with $\mathsf{matchable}(\mathcal{M}, e', e)$ from the definition of $\mathsf{readable}(\mathcal{M}, e', e)$. For $\mathsf{Ref}(\Pi\mathsf{Unit})$-typed references, only having the predicate $\mathsf{readableC}(\mathcal{M}, e', e)$ is too weak to check whether e is reachable from e' or not. More concretely, suppose that, in some model $\mathcal{M}$, a reference x is typed as $\mathsf{Ref}(\mathsf{Unit})$ and another reference y stores x. Then a program $!y$ can surely return x in $\mathcal{M}$. Further, $\texttt{if}\ x = !y\ \texttt{then}\ 0\ \texttt{else}\ 100$ can directly use this reachability information, in the sense that, if the content of y is hypothetically another name which is distinct from x, then we get 100 instead of 0. This state of affairs is precisely what is captured by $\mathsf{matchable}(\mathcal{M}, y, x)$: we can reach x from y, and that makes a behavioural difference.

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

Definition 4.24 (permutation) *Let $\mathcal{M}^{\Gamma;\Delta;\mathcal{D}} \stackrel{def}{=} (\nu\tilde{x})(\xi_1 \cdot \xi_2 \cdot v : V \cdot w : W \cdot \xi_3,\ \sigma)$. Then, for any $v, w \in \mathsf{dom}(\Gamma)$, we set:*

$$\pi\binom{vw}{wv}\mathcal{M} \stackrel{def}{=} (\nu\tilde{x})(\xi_1 \cdot v : W\xi_2 \cdot \cdot w : V\xi_2,\ \sigma).$$

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 $\pi(\rho)\mathcal{M}$.*

Remark 4.25

1. By definition, given $\mathcal{M}^{\Gamma;\Delta;\mathcal{D}}$, if $\Gamma(v) = \Gamma(w)$ then a permutation $\pi\binom{uv}{vu}\mathcal{M}$ 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:

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

   We shall have further examples later.
3. A permutation in the sense of Definition 4.24 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 $\pi\binom{uv}{vu}\mathcal{M}[v\!:\!e][u\!:\!e']$ is essentially the same model as $\mathcal{M}$ except they differ in typing (as a sequence).

Definition 4.26 (symmetry) *A permutation* $\pi(\rho)$ *on* $\mathcal{M}$ *is a* symmetry on $\mathcal{M}$ *when* $\pi(\rho)\mathcal{M} \approx \mathcal{M}$.

Any model has the trivial symmetry, identity. The following shows subtlety of symmetries.

Example 4.27 (symmetry)

1. The following two models correspond to `IncShared` and `IncUnShared`:

$$\mathcal{M}_1 \stackrel{\text{def}}{=} (\nu x)(\{u : \mathtt{Inc}, v : \mathtt{Inc}, x : \mathbf{i}\},\ \{\mathbf{i} \mapsto 0\})$$
$$\mathcal{M}_2 \stackrel{\text{def}}{=} (\nu xy)(\{u\!:\!\mathtt{Inc}, v\!:\!\mathtt{Inc}[y/x], x\!:\!\mathbf{i}, y\!:\!\mathbf{j}\}\ \{\mathbf{i} \mapsto 0, \mathbf{j} \mapsto 0,\})$$

   Both $\mathcal{M}_1$ and $\mathcal{M}_2$ have an obvious symmetry $\pi\binom{uv}{vu}$.
2. If we expand these two models, however, we find one retains a symmetry while another doesn't.

$$\pi\binom{wv}{vw}(\mathcal{M}_1[w\!:\!u]) \approx \mathcal{M}_1[w\!:\!u]$$
$$\pi\binom{wv}{vw}(\mathcal{M}_2[w\!:\!u]) \not\approx \mathcal{M}_2[w\!:\!u]$$

   To see why the latter is the case, let:

$$V \stackrel{\text{def}}{=} \lambda x.(\mathtt{let}\ y = \pi_2(x)()\ \mathtt{in}\ \mathtt{let}\ z = \pi_3(x)\ \mathtt{in}\ z)$$

   Then $V[\![\mathcal{M}_2[w:u]]\!] \longrightarrow^* 1$ but $V[\![\pi\binom{wv}{vw}\mathcal{M}_2[w:u]]\!] \longrightarrow^* 2$, because of the lack of sharing.

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. A key idea we shall use for this purpose is the following notion.

Definition 4.28 (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 4.29 (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 4.30 *Let $\mathsf{dom}(\Gamma) = \{y_1, ..., y_n\}$ below.*

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

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

   witnesses the expansion from $\mathcal{M}$ of type $\Gamma;\Delta;\mathcal{D}$ to $\mathcal{M}[u\!:\!N]$.
2. *Given $\Gamma;\Delta \vdash e : \mathsf{Ref}(\alpha)$ and $N^{\Gamma;\Delta;\alpha}$, the program*

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

   witnesses the update $[\![\mathcal{M}[e \mapsto N]]\!]$ of $\mathcal{M}^{\Gamma;\Delta;\mathcal{D}}$.
3. *For i such that $1 \le i \le n$, the program*[4]*.*

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

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

PROOF: Immediate from the construction. □

Lemma 4.31 *$\mathcal{M}_1 \approx \mathcal{M}_2$ implies $\mathcal{M}_1[u\!:\!N] \approx \mathcal{M}_2[u\!:\!N]$.*

PROOF: Let u be fresh, $\mathcal{M}_i^{\Gamma;\Delta;\mathcal{D}}$ and $\Gamma;\Delta \vdash N : \alpha$. Write V for the witness of the expansion in Definition 4.28. We infer:

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

where the third step uses Lemma 4.30 (1). □

Lemma 4.32 *$\mathcal{M}_1 \cong \mathcal{M}_2$ implies $\mathcal{M}_1/u \cong \mathcal{M}_2/u$.*

PROOF: As in Lemma 4.31, using Lemma 4.30 (3) instead of (1). □

By successively applying Lemma 4.32, the property extends to arbitrary projections of models.

When N is gentle, Lemma 4.31 has the converse.

[4] 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 $\langle \pi_1(m), .., \pi_{i-1}(m), \pi_{i+1}(m), .., \pi_n(m)\rangle$

Lemma 4.33 *If N is gentle w.r.t. $\mathcal{M}_{1,2}^{\Gamma;\Delta;\mathcal{D}}$, then $\mathcal{M}_1[u\!:\!N] \approx \mathcal{M}_2[u\!:\!N]$ implies $\mathcal{M}_1 \approx \mathcal{M}_2$.*

PROOF: We infer:

$$\begin{array}{lcll} \mathcal{M}_1 & \approx & \mathcal{M}_1[u\!:\!N]/u & (\text{By } N \in \mathcal{G}(\mathcal{M}_1)) \\ & \approx & \mathcal{M}_2[u\!:\!N]/u & (\text{Lemmas 4.31 and 4.32}) \\ & \approx & \mathcal{M}_2 & (\text{By } N \in \mathcal{G}(\mathcal{M}_2)) \end{array}$$

as required. □

As expected, gentleness is also closed under $\approx$

Lemma 4.34 *If $N \in \mathcal{G}(\mathcal{M}_1)$ and $\mathcal{M}_1 \approx \mathcal{M}_2$ then $N \in \mathcal{G}(\mathcal{M}_2)$.*

PROOF: Assume $\mathcal{M}_1 \approx \mathcal{M}_2$.

$$\begin{aligned} N \in \mathcal{G}(\mathcal{M}_1) &\Leftrightarrow \mathcal{M}_1[u\!:\!N] \Downarrow \mathcal{M}_1' \wedge \mathcal{M}_1'/u \approx \mathcal{M}_1 \\ &\Leftrightarrow \mathcal{M}_2[u\!:\!N] \Downarrow \mathcal{M}_2' \wedge \mathcal{M}_2'/u \approx \mathcal{M}_2 \\ &\Leftrightarrow N \in \mathcal{G}(\mathcal{M}_2) \end{aligned}$$

Above the middle step uses Lemmas 4.31 and 4.32. □

Lemma 4.35 *If $N \in \mathcal{G}(\mathcal{M}_{1,2})$ and $\mathcal{M}_1 \approx \mathcal{M}_2$, then $\mathcal{M}_1[u \mapsto N] \approx \mathcal{M}_2[u \mapsto N]$.*

PROOF: As in Lemma 4.31, using Lemma 4.30 (2). □

Since the extension by a reference name uses a hiding operation on models (cf. Definition 4.15 (2)), its property w.r.t. $\approx$ depends on the following lemma.

Proposition 4.36 *If $\Gamma;\Delta,x:\mathsf{Ref}(\alpha) \vdash M_1 \cong_{\mathsf{id}} M_2 : \beta$ and $\Gamma;\Delta \vdash V : \alpha$, then*

$$\Gamma;\Delta,x:\mathsf{Ref}(\alpha) \vdash \mathtt{new}\,x := V\ \mathtt{in}\,M_1 \cong_{\mathsf{id}} \mathtt{new}\,x := V\ \mathtt{in}\,M_2 : \beta.$$

PROOF: Choose appropriatly typed semi-closing $C[\cdot]$.

$$\begin{aligned} C[\mathtt{new}\,x := V\ \mathtt{in}\,M_1] &\rightarrow (\nu x)(C[M_1], x \mapsto V) \Downarrow \\ &\Leftrightarrow (x := V; M_1) \Downarrow \\ &\Leftrightarrow (x := V; M_2) \Downarrow \\ &\Leftrightarrow C[\mathtt{new}\,x := V\ \mathtt{in}\,M_2] \rightarrow (\nu x)(C[M_2], x \mapsto V) \Downarrow \end{aligned}$$

as required. □

Lemma 4.37 *$\mathcal{M}_1 \approx \mathcal{M}_2$ implies $(\nu x)\mathcal{M}_1 \approx (\nu x)\mathcal{M}_2$.*

PROOF: Immediate by Proposition 4.36. □

Lemma 4.38 *Assume $\mathcal{M}_1 \approx \mathcal{M}_2$ and $\mathcal{M}_1' \approx \mathcal{M}_2'$. Then $\mathcal{M}_1 \leq_{y:\alpha} \mathcal{M}_2$ iff $\mathcal{M}_1' \leq_{y:\alpha} \mathcal{M}_2'$.*

PROOF: By Lemmas 4.31 and 4.37. □

For the remaining relations:

Lemma 4.39 *In (1) below, $\approx$ is extended point-wise to elements in$(!e)\mathcal{M}_{1,2}$.*

1. *$\mathcal{M}_1 \approx \mathcal{M}_2$ implies $(!e)\mathcal{M}_1 \approx (!e)\mathcal{M}_2$.*
2. *Assume $\mathcal{M}_1 \approx \mathcal{M}_2$. Then*
   (a) $\mathsf{readable}(\mathcal{M}_1, e', e)$ *iff* $\mathsf{readable}(\mathcal{M}_2, e', e)$.
   (b) $\mathsf{writable}(\mathcal{M}_1, e', e)$ *iff* $\mathsf{writable}(\mathcal{M}_2, e', e)$.

PROOF: (1) is by Lemmas 4.34 and 4.35. (2) is by construction. □

Finally we mention basic properties of permutation and symmetries. First, a permutation has an obvious witness:

Lemma 4.40 (permutation witness) *Given $\mathcal{M}^{\Gamma;\Delta;\mathcal{D}}$ and a bijection ρ on $\mathsf{dom}(\Gamma)$, the operation $\pi\rho$ is witnessed by the standard isomorphism on $\Pi\mathcal{D}$ permuting the elements following ρ.*

Corollary 4.41 *Given $\mathcal{M}_{1,2}^{\Gamma;\Delta;\mathcal{D}}$ and a bijection ρ on $\mathsf{dom}(\Gamma)$, we have $\mathcal{M}_1 \approx \mathcal{M}_2$ iff $\pi(\rho)\mathcal{M}_1 \approx \pi(\rho)\mathcal{M}_2$.*

Hence we know:

Proposition 4.42 *If $\mathcal{M}_1 \approx \mathcal{M}_2$ and if $\pi\rho$ is a symmetry of $\mathcal{M}_1$, then $\pi\rho$ is also a symmetry of $\mathcal{M}_2$.*

4.6 Semantics of Assertions

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 $\forall \mathcal{M}'.(\mathcal{M} \leq_{x:\alpha} \mathcal{M}' \Rightarrow \mathcal{M}' \models C)$.
- $\mathcal{M} \models \forall \mathrm{X}.C$ if $\forall \alpha \in \mathsf{clt}.\mathcal{M} \cdot \mathrm{X} : \alpha \models C$.
- $\mathcal{M} \models [!e]C$ if $\forall \mathcal{M}'.(\mathcal{M}' \in (!e)\mathcal{M} \Rightarrow \mathcal{M}' \models C)$.
- $\mathcal{M} \models \{C\}e \bullet e' = x\{C'\}$ if, whenever $\mathcal{M}_0$ is such that $\mathcal{M} \rightsquigarrow \mathcal{M}_0$ and $\mathcal{M}_0 \models C$, we have $\mathcal{M}_0[x : ee'] \Downarrow \mathcal{M}' \models C'$.
- $\mathcal{M} \models e' \hookrightarrow e$ if $\mathsf{readable}(\mathcal{M}, e', e)$ or $\mathsf{writable}(\mathcal{M}, e', e)$.

Remark 4.43 (satisfaction for evaluation formulae) There are several possible definitions of the satisfaction for evaluation formulae. One possible (most strict) way would be to set $\mathcal{M} \models \{C\}e \bullet e' = x\{C'\}$ to be, assuming $\mathcal{M}$ is of type $\Gamma;\Delta;\mathcal{D}$,

> Whenever we have $\mathcal{M}_0 \in (!\mathsf{dom}(\Delta))\mathcal{M}$ and $\mathcal{M}_0 \models C$, it holds that $\mathcal{M}_0[x : ee'] \Downarrow \mathcal{M}' \models C'$.

where $(!\tilde{e})\mathcal{M}$ is an obvious generalisation of $(!e)\mathcal{M}$. Another, and slightly more, general definition can be given as follows, assuming the same typing:

> Whenever we have $\Delta \vdash e_i : \mathsf{Ref}(\alpha_i)$, $\mathcal{M}_0 \in (!(e_0..e_n))\mathcal{M}$ and $\mathcal{M}_0 \models C$, it holds that $\mathcal{M}_0[x\!:\!ee'] \Downarrow \mathcal{M}' \models C'$.

These variants consider different notions of hypothetical states evaluation formulae can stipulate: the present one is the most general among them, though the other two are also of interest (and are useful) from the viewpoint of calculation of satisfiability.

Remark 4.44 (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:

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

On the other hand, it also says:

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

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 4.27, page 31). 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 replacable 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 $\mathsf{inc}(u,x) \stackrel{\text{def}}{=} \forall i.\{!x = i\}u \bullet () = z\{!x = z = i+1\} @ x$. 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 *rarther 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 $\pi\binom{uw}{wu}$ is obviously a symmetry of $\mathcal{M}_2[w:u]$, by Proposition 4.42, it should also be a symmetry of $\mathcal{M}_2[w:v]$. But we have already seen it is not so in Example 4.27 (2) (page 31), 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 [16], augmented with a fresh name called *anchor* [18–20].

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

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.[5] Sequents have identical shape as those in [5, 20]: 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* $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$*,*

1. $\Gamma,\Delta,\Theta \vdash C$ *and*
2. $u\!:\!\alpha,\Gamma,\Delta,\Theta \vdash C'$*,*

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) We say a well-typed judgement $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is *valid*, written $\models \{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, iff the following condition holds:

$$\forall \mathcal{M}^{\Gamma';\Delta';\mathcal{D}}.(\Gamma';\Delta' \vdash C \ \wedge\ \Gamma';\Delta' \vdash \exists u^{\alpha}.C' \ \wedge\ \mathcal{M} \models C \quad \supset \quad \mathcal{M}[u\!:\!M] \Downarrow \mathcal{M}' \models C'.)$$

where, by the previous conventions, models and judgements are assumed to be well-typed.

[5] Total correctness was chosen following [5, 19, 20]. The proof rules for partial correctness are essentially identical except for recursion.

Remark 5.4 (semantics of judgement and freshness) We observe how the simple defining clause for validity of judgement induces a sound interpretation of freshness of generated reference names, as informally discussed in Section 3.3. Recall the following judgement from Section 3.3:

$$\{\mathsf{T}\}\ \mathtt{Inc} :_u \{\nu x.(!x = 0\ \wedge\ \forall n.\{!x = n\}u \bullet () = z\{!x = z = n+1\})\}$$

which is the same thing as:

$$\{\mathsf{T}\}\ \mathtt{Inc} :_u \{\exists x.(x \neq i \wedge\ !x = 0\ \wedge\ \forall n.\{!x = n\}u \bullet () = z\{!x = z = n+1\})\}$$

We set the post condition as $\mathsf{IncSpec}$. Now suppose $\mathcal{M}^{\Gamma;\Delta} \models \mathsf{T}$, which always holds for any (well-typed) $\mathcal{M}$. We then want:

$$\mathcal{M}[u\!:\!\mathtt{Inc}] \Downarrow \mathcal{M}' \models \mathsf{IncSpec}$$

We show this is indeed the case. In the following we assume i is a reference name: the case when i is a non-reference is simpler. Since $\mathcal{M}$ should interpret i, we can safely set:

$$\mathcal{M} \stackrel{\text{def}}{=} (\nu\,\tilde{y})(\xi,\sigma) \text{ such that } \xi(i) = \mathbf{i} \text{ and } \sigma(\mathbf{i}) = V$$

where $V \stackrel{\text{def}}{=} \lambda().(x :=\, !x+1; !x)$. Then we observe:

$$\mathcal{M}[u\!:\!\mathtt{Inc}] \Downarrow (\nu\,\tilde{y}x)(\xi\cdot x\!:\!\{x\}\cdot u\!:\!V,\ \ \sigma\cdot\{x\} \mapsto 0) \stackrel{\text{def}}{=} \mathcal{M}'.$$

Note x can be aliased only to itself, because of the operational semantics of the "new" construct. We can now check:

$$\mathcal{M}' \ \models\ x \neq i\ \wedge\ \ !x = 0\ \ \wedge\ \ \forall n.\{!x = n\}u \bullet () = z\{!x = z = n+1\}$$

Note $x \neq i$ holds since x is aliased only to itself in $\mathcal{M}_0'$: whatever $\mathbf{i}$ would be, x cannot be included in it. The rest is direct application of the clauses for the satisfaction of conjunction and evaluation formulae. By the defining clause for the existential of a reference name, we obtain:

$$\mathcal{M} \models \exists x.(x \neq i\ \ \wedge\ \ !x = 0\ \ \wedge\ \ \forall n.\{!x = n\}u \bullet () = z\{!x = z = n+1\})$$

as required.

Notation 5.5 (judgement for freshness) We use the following abbreviation for judgements similar to those with evaluation formulae in § 3.4. 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).
- $\{C\}\ M :_m \{\sharp m.C'\}$ stands for $\{C\}\ M :_m \{m \sharp i \wedge C'\}$.
- $\{C\}\ M :_m \{\star m.C'\}$ 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 $\{C\}\ M :_m \{\nu\sharp x.C'\}@\tilde{y}$.

Fig. 4 Compositional Rules (the new rule is highlighted).

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

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

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

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

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

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

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

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

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

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

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

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

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

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 4 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 5.

Fig. 5 Structural rules (change from [5] is highlighted).

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

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

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

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

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

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

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

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 $\tilde{i}$.

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 however exclude $x \sharp y$ from stateless formulae (they implicitly contain dereference: see Section 6.1).

In the following, we focus two new aspects of the presented proof rules: the proof rule for new name generation (highlighted in Figure 4) and the proof rule for universal quantification (highlighted in Figure 5).

Proof Rule for New Name Generation. The compositional rules in Figure 4 stay identical with those in the base logic [5] except for adding the rule for new name generation, in spite of the significant semantic enrichment. 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

Fig. 6 Variants of [*New*] and Other Related Rules.

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

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

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

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

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

without being dereferenced. Formally $\mathsf{fpn}(e)$ is inductively defined as: $\mathsf{fpn}(x) \stackrel{\text{def}}{=} \{x\}$, $\mathsf{fpn}(\mathsf{c}) = \mathsf{fpn}(!e) \stackrel{\text{def}}{=} \emptyset$, for other constructs it acts homomorphically. $\mathsf{fpn}(\tilde{e})$ is short for $\cup_i \mathsf{fpn}(e_i)$, $x \sharp \tilde{e}$ for $\wedge_i\, x \sharp e_i$. 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 ($x \sharp \tilde{e}$), 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'.*

Some remarks:

– The side condition $x \notin \mathsf{fpn}(\tilde{e})$ is essential for the consistency of the rule: as an extreme case, $x \sharp x$ is obviously false, similarly $x \sharp \langle x, x \rangle$ (cf. § 3.4). However $x \sharp !x$ 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, $x \sharp \tilde{e}$, 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 6. [*NewVar*] is equi-potent with the original rule when combined with Consequence. 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 [*Invariant*], 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 [15] except adding the condition for unreacability.

In Introduction, 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 6, 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 6. [*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 (1.5) and the proof rule above, as well as others, we can derive [***New***] as follows.

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

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 6, 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, Inv)
4. $\{y \neq i \wedge C_0\}\, N :_u \{\exists y.(y \neq i \wedge C')\}$	(3, Cons)
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. $\{x \neq y\}\ \mathtt{if}\, x = y\, \mathtt{then}\, 0\, \mathtt{else}\, 100 :_u \{u = 100\}$	
3. $\{y \sharp x\}\ \mathtt{if}\, x = y\, \mathtt{then}\, 0\, \mathtt{else}\, 100 :_u \{u = 100\}$	(2, Cons)
4. $\{\mathsf{T}\}\ \mathtt{new}\, y := 1\, \mathtt{in}\, \mathtt{if}\, x = y\, \mathtt{then}\, 0\, \mathtt{else}\, 100 :_u \{\nu x.u = 100\}$	(1,3,New)
5. $\{\mathsf{T}\}\ \mathtt{new}\, y := 1\, \mathtt{in}\, \mathtt{if}\, x = y\, \mathtt{then}\, 0\, \mathtt{else}\, 100 :_u \{u = 100\}$	(4, Erase)

Above Line 2 uses the entailment

$$y \sharp x \quad \supset \quad x \neq y$$

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 5 stay identical with those in the preceding logic [5] except for a slight change in [$Aux_{\forall}$], which adds the highlighted side condition. Without this side condition, the rule is not sound. As a simple example, [*Ref'*] immediately gives us:

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

from which we can*not* conclude

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

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

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 4.8, page 27) and semantics of judgement (Definition 5.3, page 37):

$$\mathcal{M}[u\!:\!\mathsf{Ref}(0)] \Downarrow (\nu x)(\mathcal{M}\cdot x : \{u,x\} \mapsto 0) \ \models\ u \neq i$$

hence up to $\approx$:

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

Note i should be interpreted in $\mathcal{M}$, i.e. it is in the domain of ξ (note i is a distinct identifier from u, recalling i is chosen fresh). 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 identical 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:

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

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

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

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 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

Fig. 7 Derived compositional rules for located assertions

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

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

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

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

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

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

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

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

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

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

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

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

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

judgements, listed in Figure 7 and Figure 8. All rules come from [5] 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 8, [*Invariance*] generalises the rule of the same name in Figure 5, and is used for modular and extensional reasoning for programs with write effects.

These rules stay as they originally are in [5] in spite of a major change in semantics of assertions. A central reason for this is that the operators on models used for inter-

Fig. 8 Derivable structural rules for located judgements.

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

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

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

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

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

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

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

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 $\tilde{i}$.

preting the present logic discussed in Section 4 work precisely in the same way as the set-theoretic operators used for interpreting the logic in [5] (and [20]), 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. 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 $\mathcal{M}[u\!:\!M] \Downarrow \mathcal{M}' \models C' \wedge C_0$.

Fig. 9 Other Derived Located Proof rules.

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

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

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

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

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

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

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

$$[\mathit{Pair}\star]\ \frac{C \equiv C_1 \wedge [!\tilde{e}_1]C_2 \quad \{C_i\}\ M_i :_{m_i} \{\star m_i.C_i'\}@\tilde{e}_i \quad C' \equiv \langle !\tilde{e}_2\rangle C_1' \wedge C_2'}{\{C\}\ \langle M_1, M_2\rangle :_u \{\star u.\exists \tilde{m}.(u = \langle m_1, m_2\rangle \wedge m_1 \star m_2 \wedge C')\}@\tilde{e_1}\tilde{e_2}}$$

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

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

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

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

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

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

As we have already explored in our previous work [5, 20], programs of specific shape allow efficient reasoning rules, some of which are listed in Figure 9. There are three categories among these rules.

1. The rules for "simple" expressions (the first five rules), coming from [5, 20].
2. The rules for specific imperative constructs (the next three rules), again coming from [5, 20].
3. The rules which infer for judgements with freshnames.

In the following we briefly illustrate the rules in the third category. These rules can be subdevided into:

1. The first four rules, which derive (located) judgements for $\star$-freshness, of the form:

$$\{C\}\ M :_u \{\star u.C'\}@\tilde{e}$$

2. The next four rules, which derive (located) judgements for $\sharp$-freshness of a set of names, of the form:

$$\{C\}\ M :_u \{\sharp S.C'\}@\tilde{e}$$

   where S is the notation of the form:

$$\{z \mid G(z)\}$$

   understood as the standard set-theoretic notion, i.e. the set of names which satisfy $G(z)$ (hence z is a binder).

Finally the last rule is not part of these categories but is useful in practice.

The $\star$-freshness rules are easily understood. We discuss briefly on the second subcategory, i.e. the rules for set-based $\sharp$ names. These rules are useful when we accumulate $\sharp$-freshnames which are not $\star$-fresh (e.g. some of the reachable references are stored in existing references). The assertion:

$$\sharp S.C' \quad \text{with} \quad S = \{z \mid G(z)\}$$

stands for, for some fresh i:

$$\forall z.(G(z) \supset z \sharp i) \ \wedge \ C'$$

which is in accordance with intuitive understanding. Note there is no binding from the set to C: it is a simple conjunction. However having the fixed place for the $\sharp$-fresh names makes reasoning more transparent than without.

6 Elimination, Soundness and Observational Completeness

6.1 Elimination Result

In this subsection 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. As the first step, we define a simple inductive method for defining reachability from a datum of a finite type (recall a type is *finite* when it contains neither an arrow type nor free/bound type variable, cf. Definitiojn 3.1, page 21).

Definition 6.1 (i-step reachability) *Let α be a finite type. Then the i-step reachablility 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).*

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

Remark 6.2 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 6.3 *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 readability and writability (through obvious modification, replacing, in each, V in the defining clause with $\lambda x.(V\pi_1(x))$). Similarly when $\pi_2(x) \hookrightarrow y$, hence done.

For the converse, we show the contrapositive, showing:

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

The proof uses a simple operational reasoning. 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 $\mathcal{M} \models \mathsf{reach}(x^{\alpha},\ y,\ i)$). Thus, by a direct observation of reduction involved in readability and writability predicates, we can verify unreadability and unwritability, hence done. $\square$

Now let us define the predicate $x^{\alpha} \hookrightarrow' y^{\mathsf{Ref}(\beta)}$ with α finite, by the axioms given in Figure 2, Section 3.6, which we reproduce below.

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

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

Proposition 6.4 *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. $\square$

Corollary 6.5 *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 6.3 and 6.4. $\square$

When α can be recursively finite above (recall a type is recursively finite when it is closed and contains neither an arrow type nor type quantifiers, cf. Definitiojn 3.2), we define $\mathsf{reach}(x^{\alpha},\ y^{\mathsf{Ref}(\beta)},\ i^{\mathsf{Nat}})$ by precisely the same clauses (taking the iso-recursive approach [36]). Then we again obtain, by precisely identical arguments:

Proposition 6.6 *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 6.7 The literally same argument as for Proposition 6.3 can be used for Proposition 6.6 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 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:

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

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

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

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

Proposition 6.8 *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 $\hookrightarrow''$. $\square$

Thus we obtain, by Propositions 6.6 and 6.8:

Corollary 6.9 *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.

From these results we obtain the elimination result. Below we say a reachability predicate $x^{\alpha} \hookrightarrow y^{\mathsf{Ref}(\beta)}$ is *finite* if α is finite (β can be arbitrary).

Theorem 6.10 (elimination) *Let C be such that each reachability predicate occurring in C is finite. Then there exists C' such that $C' \equiv C$ such that C' does not contain any occurrences of reachability predicates.*

6.2 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 6.11 *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 $\pi\binom{uv}{vu}\mathcal{M} \models C[uv/vu]$.*
3. *(exchange) Let $u,v \notin \mathsf{fn}(e,e')$. 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 simulnaneously 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 $\xi' \stackrel{\text{def}}{=} \xi\delta$:

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

Above $(*)$ used $[\![e_i]\!]_{(\xi,\sigma)} \stackrel{\text{def}}{=} [\![e_i\delta]\!]_{(\xi',\sigma)}$. Dually for its negation. The rest is easy by indcution. (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.

$$\begin{aligned}
&\mathcal{M}' \models e_1 = e_2\\
&\Leftrightarrow \mathcal{M}[x:e_1] \approx \mathcal{M}[x:e_2] && \text{(Def)}\\
&\Leftrightarrow \mathcal{M}[x:e_1][u:e] \approx \mathcal{M}[x:e_2][u:e] && \text{(Lem.4.31)}\\
&\Leftrightarrow \mathcal{M}[u:e][x:e_1] \approx \mathcal{M}[u:e][x:e_2] && \text{((3) above)}
\end{aligned}$$

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

$$\begin{aligned}
&\mathcal{M} \models e_1 = e_2\\
&\Leftrightarrow\ \mathcal{M}[u:e_1] \approx \mathcal{M}[u:e_2] && \text{(Def)}\\
&\Leftrightarrow\ \mathcal{M}[u:e_1][v:e_2] \approx \mathcal{M}[u:e_2][v:e_2]\ \wedge\\
&\qquad \mathcal{M}[u:e_2][v:e_2] \approx \mathcal{M}[u:e_2][v:e_1] && \text{(Lem.4.31; (3) above)}\\
&\Rightarrow\ \mathcal{M}[u:e_1][v:e_2] \approx \mathcal{M}[u:e_2][v:e_1].
\end{aligned}$$

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:

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

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, by the gentleness of e:

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

The last line is because the interpretation is homomorphic. □

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

Lemma 6.12 (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 [24, §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 $\mathcal{M} \models C[x/y]$.

$$\begin{aligned}\mathcal{M} \models x = y &\Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y] \approx \mathcal{M}[u\!:\!y][v\!:\!x] && (6.1)\\ &\Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_i] \approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i] && (6.2)\end{aligned}$$

Here (6.1) is by Lemma 6.11.5 and (6.2) follows from Lemma 4.31.

$$\begin{aligned}\mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_i] &\approx \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_i[v/y]] && \text{(Lem. 6.11.6)}\\ &\approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i[v/y]] && \text{(Lem. 4.31, 6.1)}\\ &\approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i[vv/xy]] && \text{(Lem. 6.11.6)}\\ &\approx \mathcal{M}[u\!:\!y][v\!:\!x][w\!:\!e_i[xx/xy]] && \text{(Lem. 6.11.7)}\\ &\approx \mathcal{M}[w\!:\!e_i[xx/xy]][u\!:\!y][v\!:\!x] && \text{(Lem. 6.11.3)}\end{aligned}$$

$$\mathcal{M} \models e_1 = e_2 \Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y] \models e_1 = e_2 \qquad \text{(Lem. 6.11.4)}$$
$$\Rightarrow \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_1] \approx \mathcal{M}[u\!:\!x][v\!:\!y][w\!:\!e_2]$$

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, applying Lemma 4.33 twice, which is permissible as x and y are gentle, 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 6.13 *(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.

$$\begin{aligned}
\mathcal{M} \models \forall x^{\alpha}.C &\Leftrightarrow \forall N \in \mathcal{G}(\mathcal{M}).\mathcal{M}[x\!:\!N] \models C \\
&\Rightarrow \forall e.\mathcal{M}[x\!:\!e] \models C
\end{aligned}$$

For (2) we use Lemma 4.33. □

Lemma 6.14 *(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 $\mathcal{M}' \in (!x)\mathcal{M} : \mathcal{M}' \models C_1$. Consequently:

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

as required. □

Lemma 6.15 *$\mathcal{M} \models \langle !x\rangle(C \wedge !x = m)$ iff $\mathcal{M}[x \mapsto m] \models C$.*

PROOF: Straightforward and standard. Below let us set $\mathcal{M}$ to be the model $(\nu\tilde{y})(\xi \cdot x : \mathbf{i} \cdot m : V, \sigma \cdot \mathbf{i} \mapsto W)$.

$$\begin{aligned}
&\mathcal{M} \models \langle !x\rangle(C \wedge !x = m)\\
&\Leftrightarrow \exists \mathcal{M}' \in (!x)\mathcal{M}.(\mathcal{M}' \models C \wedge \mathcal{M}' \models !x = m)\\
&\Leftrightarrow (\nu\tilde{y})(\xi \cdot x : \mathbf{i} \cdot m : V, \sigma \cdot \mathbf{i} \mapsto V)\\
&\Leftrightarrow \mathcal{M}[x \mapsto m] \models C
\end{aligned}$$

as required (note the reasoning is identical with the corresponding proof in [5], or with the corresponding one in Hoare logic). □

Notation 6.16 We write $C\{\!|e'/!e|\!\}$ for $\exists m.(\langle !x\rangle(C \wedge !e = m) \wedge m = e')$ with m fresh.

Lemma 6.17 *(plain free reference names) Let $u \notin \mathsf{fpn}(e)$. Then, with u fresh:*

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

PROOF: In $\mathcal{M}'$ u is distinct from all other references, and not aliased. Hence $\mathcal{M}'$ has shape

$$(\nu\tilde{y})(\xi^{-u} \cdot u : \{u\}, \sigma^{-u} \cdot V^{-u}).$$

Consequently, e cannot reach u. □

Notation 6.18

1. We say $\mathcal{M}[\mathtt{inj}_1(x) : N]$ *makes sense* if the following condition holds:

   $$\mathcal{M} = (\nu\tilde{y})(\xi, \sigma) \quad \text{and} \quad (N\xi, \sigma) \Downarrow (\nu\vec{z})(\mathtt{inj}_i(V), \sigma').$$

   Similarly for other constructors.

2. Let $N \in \mathcal{G}(\mathcal{M})$ and x be fresh. Then we write $\mathcal{M} \uplus [x \mapsto N]$ for the addition of a fresh singleton identical x with value N to $\mathcal{M}$. Formally, given $\mathcal{M}$ such that $(\nu\tilde{y})(\xi, \sigma)$ and $(\mathcal{N}\xi, \sigma) \Downarrow (\nu\tilde{z})(V, \sigma')$, we set $\mathcal{M} \uplus [x \mapsto N]$ as:

   $$(\nu\tilde{y}\tilde{z})(\xi \cdot x : \{x\}, \sigma' \cdot \{x\} \mapsto V)$$

6.3 Soundness of Proof Rules

we are ready to prove:

Theorem 6.19 $\vdash \{C\}\, M :_u \{C'\}$ *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:

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

For [*Proj*] we reason as follows.

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

The [*Case*]-Rule uses the "makes sense" predicate in Notation 6.18

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

Now we reason for [*Abs*]. Let $\mathcal{M}$ have type $\Gamma;\Delta;\mathcal{D}$.

$$\begin{array}{rcl} \mathcal{M} \models A \wedge C & \Rightarrow & \mathcal{M}[m\!:\!M] \Downarrow \mathcal{M}' \models C' \\ & \Rightarrow & \mathcal{M}' \in (!\Delta)\mathcal{M}, \mathcal{M}' \models A \\ & \Rightarrow & \forall \mathcal{M}_0 \in (!\Delta)\mathcal{M}.\forall N \in \mathcal{G}(\mathcal{M}).\mathcal{M}_0[m\!:\!(\lambda x.M)N] \Downarrow \mathcal{M}' \models C' \\ & \Rightarrow & \mathcal{M}[u\!:\!\lambda x.M][x\!:\!N] \models \{C\}u \bullet x = m\{C'\} \\ & \Rightarrow & \mathcal{M}[u\!:\!\lambda x.M] \models \forall x\tilde{i}\{C\}u \bullet x = m\{C'\} \end{array}$$

[*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:

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

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

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

[*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 [*New*], assume $\mathcal{M} \models C$ and i is a free reference name in $\mathcal{M}$. The first premise says

$$\mathcal{M}[m\!:\!M] \Downarrow \mathcal{M}_0 \models C_0$$

hence, if $x \notin \mathsf{fpn}(\tilde{e})$, by Lemma 6.17, we have

$$\mathcal{M}_0 \uplus [x \mapsto m] \models C_0 \wedge !x = m \wedge x \sharp \tilde{e}$$

We then apply the second premise to get

$$(\mathcal{M}_0 \uplus [x \mapsto m])[u\!:\!N] \Downarrow \mathcal{M}' \models C' \wedge x \neq i$$

Summarising, we obtain

$$\mathcal{M}[u\!:\!N'] \Downarrow (\nu x)(\mathcal{M}'/m) \models \exists x.(C' \wedge x \neq i)$$

with $N' \stackrel{\text{def}}{=} \mathtt{new}\, x := M \;\mathtt{in}\; N$, as required.

For [*Ref*], we reason:

$$\begin{array}{rcl}
\mathcal{M} \models C & \Rightarrow & \mathcal{M}[m\!:\!M] \Downarrow \mathcal{M}' \models C' \\
& \Rightarrow & \mathcal{M}[m\!:\!M][u\!:\!\mathtt{ref}(M)] \Downarrow \mathcal{M}' \uplus [u \mapsto m] \\
& \text{and} & \mathcal{M}' \uplus [u \mapsto m] \Downarrow \mathcal{M}'' \models C' \wedge !n = m \wedge u \sharp i \quad \text{(Lemma 6.20)} \\
& \Rightarrow & \mathcal{M}[u\!:\!\mathtt{ref}(M)] \Downarrow \mathcal{M}''/m \models \sharp u.C[!u/m]
\end{array}$$

$\square$

Lemma 6.20 *If x is fresh, N is gentle w.r.t. $\mathcal{M}$ and $x \notin \mathsf{fpn}(e)$ holds, then $\mathcal{M} \uplus [x \mapsto N] \models x \sharp e$.*

Lemma 6.21 *The invariance rule [*Inv*] in Section 7 is sound.*

PROOF: By the premise:

$$\mathcal{M} \models C \wedge C_0 \qquad \Rightarrow \qquad \mathcal{M}[u\!:\!M] \Downarrow \mathcal{M}' \models C'$$

By $!e$-freeness

$$\forall \mathcal{M}. \models C_0 \supset \forall \mathcal{M}' \in (!e)\mathcal{M}.\mathcal{M}' \models C_0$$

Further, by write effects, we have $(!\tilde{e})M = (!\tilde{e})\mathcal{M}'$. Hence $\mathcal{M}' \models C_0$ since $\mathcal{M}' \in (!e)\mathcal{M}$, as required. $\square$

Remark 6.22 On the soundness of other proof rules from [5, 20], we observe:

- All proof rules listed in the previous logic [5, 20] are sound in the present logic except for a minor ajustment 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 [5] (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 [5, 20] in its general form stems from the existence of new name generation rather than a particular choise 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.

6.4 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 [5, 20], 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 [5], 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 [5, §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 (finite cases are understood as diverging when no macthable values exist). We omit the obvious translation to imperative PCFv-terms and typing rules.

We now outline the proof of the following result.

Lemma 6.23 *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 decompoisitions, we transform these programs into FCF values maintaining the convergence property while being made less defined (we leave the details to [20, 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 [20, §6 and Appendix A]. $\square$

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:

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

$\mathfrak{a}$ 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.

$$\overset{\Rightarrow}{\alpha} ::= \overset{\Rightarrow}{\mathsf{X}} \ | \ \mathsf{Ref}(\overset{\Rightarrow}{\mathsf{X}})$$

$\overset{\Rightarrow}{\mathsf{X}}$ 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 $\mathsf{Ref}(\vec{\vec{X}})$-typed $\mathfrak{a}$:

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

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 $\mathsf{Ref}(\vec{\vec{Y}})$, 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:

$$[\![\mathfrak{a}^{\vec{\vec{X}}}]\!]_{(\xi,\sigma)} \stackrel{\text{def}}{=} \xi(\mathfrak{a}^{\vec{\vec{X}}})$$

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

6.5 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 6.24 (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 6.25 (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 6.24 except changing $\sqsubseteq$ with $\sqsupseteq$).

Definition 6.26 (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 6.27 (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 6.28 (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 6.29 *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.

From FCFs to Characteristic Formulae. In Figure 10, 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 [5, 20], the rules use equality instead of termination preorder for legibility. We observe:

Fig. 10 Proof rules for characteristic assertions of FCFs.

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

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

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

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

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

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

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

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 [5, 20]. These formulae, especially preconditions, have very restricted shape, which plays an important role in the technical discussions.

Definition 6.30 *A formula C is* open *if whenever* $(\nu\tilde{x})(\xi,\sigma)\models C$ *we have* $(\xi,\sigma)\models C$.

Proposition 6.31 *Whenever* $\vdash_{\mathsf{char}} \{C\}F :_u \{C'\}$*, the formula C is open.*

PROOF: By induction of the rules in Figure 10, noting a reference is never existentially quantified (distinction in case constructs does not differ by taking of the ν-binder). $\square$

Proposition 6.32 *If* $\vdash_{\mathsf{char}} \{C\}\ F :_u \{C'\}$*, then* (C, C') *is a strong CAP of* F *at* u*.*

PROOF: All arguments follow [5, 20]. 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 6.33 *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 $\tilde{r} = r_1..r_i..r_n$ $(n \geq 0)$:

$$
\begin{aligned}
F' &\stackrel{\text{def}}{=} \mathtt{let}\ x = fU\ \mathtt{in}\ F && (6.3)\\
\xi_0 &= \tilde{y}:\tilde{S} && (6.4)\\
\xi_1 &= \mathfrak{a}:\tilde{r}, \mathfrak{b}:\tilde{V} && (6.5)\\
\xi &= \xi_0 \cdot f:W && (6.6)\\
\sigma &= \tilde{r} \mapsto \tilde{V} && (6.7)\\
C_0 &\stackrel{\text{def}}{=} \forall \mathfrak{a}, \mathfrak{b}.\,((!\mathfrak{a} = \mathfrak{b} \wedge Max(\mathfrak{a})) \supset \forall z.\{A \wedge !\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\}) && (6.8)
\end{aligned}
$$

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) $\top, A$ is a strong CAP at z for U.

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

$$\xi_0 \cdot z : U\xi_0 \models A \qquad (6.9)$$

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 $(F'\xi, \sigma) \Downarrow$.

$$
\begin{array}{lll}
(\nu\tilde{x})(\xi,\ \sigma) \models C_0 & & \\
\Rightarrow & (\xi,\ \sigma) \models C_0 & (C_0 \text{ open})\\
\Rightarrow & (\xi \cdot \xi_1,\ \sigma) \models \forall z.\{A \wedge !\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\}) & (\forall, \text{Max})\\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \supset (z : U_1 \cdot \xi_0 \cdot \xi_1 \cdot \xi,\ \sigma) \models \{!\mathfrak{a} = \mathfrak{b}\} f \bullet z \searrow x\{C\} & \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \supset (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma_1') \models C & \\
\Rightarrow & \forall U_1 \sqsupseteq U\xi_0 \supset (\xi \cdot x : WU_1,\ \sigma) \Downarrow (\xi \cdot x : S_1,\ \sigma_1') \models C & (\text{IH2})\\
\Rightarrow & (\xi_0 \cdot x : WU,\ \sigma) \Downarrow (\xi \cdot x : S,\ \sigma) \models C & \\
\Rightarrow & (F'\xi,\ \sigma) \Downarrow & (\text{IH1})
\end{array}
$$

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

$$
\begin{array}{lll}
(F'\xi,\sigma)\Downarrow & & \\
\Rightarrow & (\xi_0\cdot x : WU,\ \sigma)\Downarrow(\xi\cdot x : S,\ \sigma)\models C & \text{(IH1)}\\
\Rightarrow & \forall U_1 \sqsupseteq U\xi_0\ \supset\ (\xi\cdot x : WU_1,\ \sigma)\Downarrow(\xi\cdot x : S_1,\ \sigma_1')\models C & \text{(IH2)}\\
\Rightarrow & z : U_1\cdot\xi_0\models A\ \supset\ (\xi\cdot x : WU_1,\ \sigma)\Downarrow(\xi\cdot x : S_1,\ \sigma_1')\models C & \\
\Rightarrow & z : U_1\cdot\xi_0\models A\ \supset\ (z : U_1\cdot\xi\cdot\xi_1,\ \sigma)\models\{!\mathfrak{a}=\mathfrak{b}\}f\bullet z\searrow x\{C\} & \\
\Rightarrow & (\xi\cdot\xi_1,\ \sigma)\models\{A\wedge !\mathfrak{a}=\mathfrak{b}\}f\bullet z\searrow x\{C\} & \text{(e5)}\\
\Rightarrow & \forall\xi_1'.\ (\ (\xi\cdot\xi_1',\ \sigma)\models !\mathfrak{a}=\mathfrak{b}\wedge Max(\mathfrak{a})\ \supset\ \{A\wedge !\mathfrak{a}=\mathfrak{b}\}f\bullet z\searrow x\{C\}) & \\
\Rightarrow & (\xi,\ \sigma)\models C_0 & \\
\Rightarrow & (\nu\tilde{x})(\xi,\ \sigma)\models C_0 &
\end{array}
$$

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 becaues 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

$$\{C_0\wedge E\}M :_u \{C'\}. \tag{6.10}$$

By Proposition 6.31, 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 [20, §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.

$$M_0 \quad \stackrel{\text{def}}{=} \quad \mathtt{let}\ \tilde{z}=!\tilde{r}\ \mathtt{in}\ \mathtt{let}\ x=yU\ \mathtt{in}\ (\tilde{r}:=\tilde{z}\ ;\ M) \tag{6.11}$$

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 (6.11) we have, with $\mathsf{dom}(\sigma_0')=\tilde{w}$ and with newly introduced entities existentially quantified:

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

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:

$$\{C\}\,(\tilde{r}:=\sigma(\tilde{r});M)\xi,\ \sigma_0) :_u \{C'\} \tag{6.12}$$

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 (6.12), 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 6.34 (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 6.35 *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 $M_1 \ncong M_2$. By abstraction, we can safely assume $M_1, 2$ are semi-closed values. By Lemma 6.23, there exist semi-closed FCF values F and $\tilde{U}$ such that, say,

$$(FM_1, \tilde{r} \mapsto \tilde{U}) \Downarrow \qquad \text{and} \qquad (FM_2, \tilde{r} \mapsto \tilde{U}) \Uparrow . \tag{6.13}$$

By Proposition 6.32, there are assertions which characterise F and $\tilde{U}$. Let the characteristic formula for F at f be written $[\![F]\!](f)$. We now reason:

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

But by (6.13) 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 $M_1 \ncong_{\mathcal{L}} M_2$, a contradiction. Thus we conclude $M_1 \cong M_2$, as required. $\square$

7 Reasoning Examples

7.1 Stored Procedures with Shared Local State

This section presents, through examples, the use of the proposed logical theory for compositionally validating the behaviour of programs with local state. We start from derivations for `IncShared` and `IncUnShared` from Introduction, reproduced below.

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

Both `IncShared` and `IncShared` store two procedures at a and b. In `IncShared`, these procedures share a local counter, while in `IncUnShared` each owns a private one. As a result, the former returns 3 while the latter 2. We first infer for `IncShared`. We use:

$$\mathsf{inc}(x,u) \stackrel{\text{def}}{=} \forall j.\{!x = j\} u \bullet () = j+1 \{!x = j+1\} @x.$$
$$\mathsf{inc}'(u,x,n) \stackrel{\text{def}}{=} !x = n \wedge \mathsf{inc}(x,u).$$

Below we work with the implicit global assumption that a, b, c_1, c_2 are pairwise distinct.

1. $\{\mathsf{T}\}\ \texttt{Inc} :_u \{\nu x.\mathsf{inc}'(u,x,0)\} @\emptyset$

---

2. $\{\mathsf{T}\}\ a := \texttt{Inc}\ \{\nu x.\mathsf{inc}'(!a,x,0)\} @a$

---

3. $\{\mathsf{inc}'(!a,x,0) \wedge \mathsf{inc}'(!a,x,0)\}\ b := !a\ \{\mathsf{inc}'(!a,x,0) \wedge \mathsf{inc}'(!b,x,0)\} @bx$

---

4. $\{\mathsf{inc}'(!a,x,0)\}\ c_1 := (!a)()\ \{\mathsf{inc}'(!a,x,1) \wedge !c_1 = 1\} @c_1x$

---

5. $\{\mathsf{inc}'(!b,x,1)\}\ c_2 := (!b)()\ \{\mathsf{inc}'(!b,x,2) \wedge !c_2 = 2\} @c_2x$

---

6. $\{!c_1 = 1 \wedge !c_2 = 2\}\ (!c_1) + (!c_2) :_u \{u = 3\} @\emptyset$

---

7. $\{\mathsf{T}\}\ \texttt{IncShared} :_u \{\nu x.u = 3\} @abc_1c_2$

---

8. $\{\mathsf{T}\}\ \texttt{IncShared} :_u \{u = 3\} @abc_1c_2$

Line 3–7 adjust pre/post conditions by [*Inv*]. The result and Line 2 get composed using [*Seq-Inv*]. The last line is by [*Erase*]. Next we present the derivation of `IncUnShared`.

1. $\{\mathsf{T}\}\ M :_m \{\nu x.\mathsf{inc}'(u,x,0)\}$

---

2. $\{\mathsf{T}\}\ a := M\ \{\nu x.\mathsf{inc}'(!a,x,0)\} @a$

---

3. $\{\mathsf{T}\}\ b := M\ \{\nu y.\mathsf{inc}'(!b,y,0)\} @b$

---

4. $\{\mathsf{inc}'(!a,x,0)\}\ c_1 := (!a)()\ \{\mathsf{inc}'(!a,x,1) \wedge !c_1 = 1\} @c_1x$

---

5. $\{\mathsf{inc}'(!b,y,0)\}\ c_2 := (!b)()\ \{\mathsf{inc}'(!b,y,1) \wedge !c_2 = 1\} @c_2y$

---

6. $\{!c_1 = 1 \wedge !c_2 = 2\}\ (!c_1) + (!c_2) :_u \{u = 2\} @\emptyset$

---

7. $\{\mathsf{T}\}\ \texttt{IncUnShared} :_u \{\nu xy.u = 2\}$

---

8. $\{\mathsf{T}\}\ \texttt{IncUnShared} :_u \{u = 2\}$

In this derivation, $x \neq y$ is guaranteed by "ν" in Line 2 and 3. Thus we can compose Lines 4 and 5 by [*Sec-Inv*] (this is in contrast to the first inference, where, in Line 3, the state of x should be shared to infer for "$b :=!a$", where extrusion takes place). In more detail, to apply [*Sec-Inv*], we should have $[!c_1x]\mathsf{inc}'(!b,y,0) \equiv (c_1 \neq by \wedge x \neq by \wedge \mathsf{inc}'(!b,y,0))$ by [*Sec-Inv*], cf. § 3.2 (6), which is guaranteed by $c_1 \neq by$ and $x \neq by$).

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 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*. We verify essentially the algorithm Reynolds treated in [41, §6] in his tutorial. The following program creates a new isomorphic copy of a given tree.

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

To type this program, we use recursive types, introduced in Section 3.6 (as noted there, no change in semantics and proof rules is necessary). We set the type of mutable trees as $\mathit{Tree} \stackrel{\text{def}}{=} \mu \mathrm{X}.(\mathsf{Ref}(\mathsf{Nat} + (\mathrm{X} \times \mathrm{X})))$, and can easily check that `treeCopy` has type $\mathit{Tree} \rightarrow \mathit{Tree}$.

Before asserting for `treeCopy`, we first specify trees.

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

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 ($u \,\sharp\, m_i$). 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 [41] 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.

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

$\mathsf{treecopy}(u)$ 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:

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

which is essentially equivalent to Reynolds's assertion in [41] (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:

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

We use derived rules for $\star$-freshness in Figure 9. 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 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:

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

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.

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

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 `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. $\{A_1 \wedge A_2\}$ $\mathtt{inj}_2(\langle f y_1, f y_2\rangle)_u$ $\{\star u.\exists \tilde{m}.(\mathsf{branch}(u, m_1, m_2) \bigwedge_{i=1,2} \mathsf{tree}\,\tau_i\,(m_i) \wedge m_1 \star m_2)\} @\emptyset$	(Pair, Inj)
6. $\{A_1 \wedge A_2\}\ \mathtt{ref}(\mathtt{inj}_2(\langle f y_1, f y_2\rangle)) :_u \{\star u.\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(u)\} @\emptyset$	(Ref)
7. $\{C \wedge \mathsf{branch}(!x, y_1, y_2)\}\ \mathtt{ref}(\mathtt{inj}_2(\langle f y_1, f y_2\rangle)) :_u \{\star u.\mathsf{tree}\,\tau\,(u)\} @\emptyset$	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. $\{\top\}$ `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 [7].

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: for induction:

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

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.

Now suppose we wish to create a new dag from a τ-labelled existing dag. If we use `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 `treeCopy` is literally identical with §7.2). So if we are to preserve sharing, we need to refine the algorithm. One such refinement follows.

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

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.

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

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

We now 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.*

As may be expected, $\mathsf{dagcopy}(u)$ (strictly) entails $\mathsf{treecopy}(u)$. Again disjointness of a created dag from the existing dag is entailed without being stated, cf. [7]. 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.

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

which says:

> *Suppose g is a dag and x contains a table org which is consistent (i.e. only relates isormorphic 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 now 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).

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

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

which says:

> *Each reference name z satisfying $C_0(z)$ is $\sharp$-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 $\sharp$-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 $x \sharp i$ is the same thing $x \neq i$).

We use the proof rules for $\sharp$-freshness in Figure 9. For the notation for $\sharp$-freshness see (7.4), §7.3. Among the presented rules, the rule $[AssVar\sharp]$ 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 $f \bullet x \searrow y$ stands for $\{\mathsf{T}\} f \bullet x = y' \{y' = y\} @ \emptyset$.

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

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.

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

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

1 $u : \{\mathsf{T}\}$ (abs)
2 **lambda g.**
3 $\quad g' : \{\mathsf{dag}\,\tau(g)\}$ (new)
4 $\quad$ **new x:= $\emptyset$ in**
5 $\quad\quad g' : \{!x = \emptyset\}$ (app)
6 $\quad\quad\quad m : \{\mathsf{T}\}$
7 $\quad\quad\quad$ **Main**
8 $\quad\quad\quad \{\forall\tau.\mathsf{dc}\,\tau(m)\}@\emptyset$
9 $\quad\quad\quad$ **g**
10 $\quad\quad\quad \{(!x = \emptyset \wedge \mathsf{dag}\,\tau(g)) \ \wedge$
11 $\quad\quad\quad\ \ \{!x = \emptyset \wedge \mathsf{dag}\,\tau(g)\}m \bullet g = g'\{\sharp^{-x}S.\mathsf{iso}(g,g')\}@x\}@\emptyset$
12 $\quad\quad \{\sharp^{-x}S.\mathsf{iso}(g,g')\}@x$
13 $\quad \{\star g'.\mathsf{iso}(g,g')\}@\emptyset$
14 $\{\forall g.\{\mathsf{dag}\,\tau(g)\}u \bullet g = g'\{\star g'.\mathsf{iso}(g,g')\}@\emptyset\}@\emptyset$

Above we set, for brevity:

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

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 $[App\,\sharp]$ (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: $M \stackrel{\text{def}}{=} \mathtt{Main}\,g$ and $C \stackrel{\text{def}}{=} !x = \emptyset \wedge \mathsf{dag}\,\tau(g)$. Note the sub-derivation $l.5$–12 means $\{C\}M\{\sharp^{-x}S.\mathsf{iso}(g,g')\}$, that is, with i fresh (cf.(7.4)):

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

  By $[New]$, we can strengthen C with $x\sharp i$. Since $\forall j \neq x \supset j \,\sharp\, x$ holds, we can apply (3.19), page 23, Section 3.6, so that the predicate $x \,\sharp\, i$ is $!x$-free. By $[Inv]$ (the invariance rule for located assertions), we obtain:

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

  that is

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

  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 $m : \{\mathsf{T}\}$ $(\mathtt{abs})$
2 **`lambda (y,g).`**
3 $\quad g' : \{!x = org \ \wedge \ \mathtt{dom}(org, g) \searrow \mathsf{F}\}$ $(\mathtt{let})$
4 $\quad$ **`let h=`**
5 $\qquad h : \{\mathsf{T}\}$ $(\mathtt{ref\text{-}simple})$
6 $\qquad$ **`ref(y) in`**
7 $\qquad \{!x = org \ \wedge \ \mathtt{dom}(org, g) \searrow \mathsf{F} \ \wedge \ !h = y \ \wedge \ h \sharp i\}@\emptyset$
8 $\quad g' : \{\cdots\}$ $(\mathtt{seq})$
9 $\qquad \{\mathsf{T}\}$ $(\mathtt{assvar}\,\sharp)$
10 $\qquad$ **`x:=`**
11 $\qquad\quad m : \{\mathsf{T}\}$
12 $\qquad\quad$ **`put(!x, <g,h>);`**
13 $\qquad\quad \{m = org \cup \{\langle g, h\rangle\} \ \wedge \ !x = org \ \wedge \ !h = y \ \wedge \ h \sharp i\}@\emptyset$
14 $\qquad \{!x = org \cup \{\langle g, h\rangle \ \wedge \ !h = y \ \wedge \ (x \sharp i \supset h \sharp i)\}\}@x$
15 $\qquad$ **`h`**
16 $\quad \{!x = org \cup \{\langle g, g'\rangle\} \ \wedge \ !g' = y \ \wedge \ (x \sharp i \supset g' \sharp i)\}@x$
17 $\{NE(m)\}@\emptyset$

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:

$$
\begin{aligned}
& NE(m) \\
& \stackrel{\text{def}}{=} \ \forall org, y, g. \\
& \qquad \{!x = org \wedge \mathtt{dom}(org, g) \searrow \mathsf{F}\} \ m \bullet (y, g) = g' \{\sharp^{-x} g'.(!x = org \cup \{\langle g, g'\rangle\} \wedge !g' = y)\}@x
\end{aligned}
$$

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 repeates 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 $[AssVar\,\sharp]$. $l.9$ 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 rasoning uses the following valid assertion:

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

  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.

```
u : {T}  (rec)
mu f.
    u : {∀τ' < τ.  dc τ'(f)}  (abs)
    lambda g.
        g' : {dag τ(g)  ∧  !x = org  ∧  con(org)}  (if)
        if dom(!x,g) then
            g' : {dom(!x,g) ↘ T}
            get(!x,g)
            {!x = org  ∧  get(t,g) ↘ g'}@∅
            {DG}@∅
        else
            g' : {dom(!x,g) ↘ F}  (case)
            case !g of
              inl(n):
                g' : {dag n(g)}
                NewEntry(inl(n), g)
                {♯⁻ˣg'.(con(!x)  ∧  !x = org ∪ ⟨g,g'⟩)}@x
                {DG}@x
              inr(g1, g2):
                g' : ∃τ1,2.(τ = ⟨τ1,τ2⟩ ⋀i=1,2 dag τi(gi))
                NewEntry(inr(<fg1, fg2>),g)
                {DG}@x
            {DG}@x
        {DG}@x
    {dc τ(u)}@∅
{∀τ.dc τ(u)}@∅
```

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

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

where we set $S(g, org) = \{z \mid g \hookrightarrow z \;\wedge\; \mathtt{cod}(org, z) \searrow \mathsf{F}\}$. 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 $\mathtt{dom}(!x,g)\searrow\mathsf{T}$, 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:

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

  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.

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

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.

[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.6–8 introduces anchors, but no new assumptions.
– l.8–15 reasons for the pair of two applications using [*Pair* $\sharp$].
– 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* $\sharp$],

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

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 summaries l.9–14 by applying [*Pair* $\sharp$]. 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 $\sharp$-fresh names in l.15
– l.16 simply encloses m with the injection.
– l.17 combines l.16 and $NE(m)$, in the form usable for [*App* $\sharp$].
– l.18 applies [*App* $\sharp$] 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 $\sharp$-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 Dynamic Mutable Data Structures (3): Graphs

To test how structured reasoning for dynamically created data structures can be reasoned in the present logical theory, we have experimented with a further refinement of the copying algorithm, again found in [7], which now works with any argument of *Tree*-type, including one with circular edges (note *Tree* allows circular linkage).

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

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.

To derive $\{\mathsf{T}\}\,\mathtt{graphCopy} :_u \{\mathsf{graphcopy}(u)\}@\emptyset$ 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 now 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.

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.

$$\begin{aligned}
&\mathsf{gc}(r,i)(u) \stackrel{\text{def}}{=} \\
&\quad \forall g, org. \\
&\quad \{!x = org \;\wedge\; \mathsf{conPre}(org,g,r) \;\wedge\; g \notin \mathsf{dom}(org) \supset \mathsf{sizeR}(g,r,i)\} \\
&\quad u \bullet g = g' \\
&\quad \{\, \sharp^{\text{-}x}\{z' \mid \exists z.\langle z,z'\rangle \in !x \;\wedge\; \langle z,z'\rangle \notin org\}. \\
&\qquad (\,(g \in \mathsf{dom}(org) \supset !x = org) \wedge \\
&\qquad\;\; (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x,g,r) \wedge !x = org \cup \langle g,g'\rangle^*_r)\,)\,)\}@x
\end{aligned}$$

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 $\mathsf{dc}\,\tau\,(u)$. 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:

$$g_1 \sqsubseteq_r g_2 \quad \stackrel{\text{def}}{\equiv} \quad \exists p_{1,2}.(\wedge_{i=1,2}\mathsf{minpathR}(r,g_i,p_i) \;\wedge\; p_1 \sqsubseteq p_2)$$

$g_1 \sqsubseteq_r g_2$ 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:

$$g_1 \prec_r g_2 \quad \stackrel{\text{def}}{\equiv} \quad \exists p_{1,2}.(\wedge_{i=1,2}\mathsf{minpathR}(r,g_i,p_i) \;\wedge\; p_1 \prec p_2)$$

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".

$$\begin{aligned} &\mathsf{iso1}(\langle g_1, g_1' \rangle,\ \langle g_2, g_2' \rangle) \\ &\equiv \wedge_{i=l,r}(\mathsf{path}(g_1, i, g_2) \equiv \mathsf{path}(g_1', i, g_2')) \ \wedge \\ &\quad \forall n.(\mathsf{atom}(!g_1, n) \equiv \mathsf{atom}(!g_2, n)) \end{aligned} \tag{7.8}$$

Note the directedness of the predicate. The consistency used in the pre-condition is given as:

$$\begin{aligned} &\mathsf{conPre}(t, g, r) \\ &\equiv \forall g_0.((\mathsf{beforeOrAboveR}'(g_0, g, r) \supset g_0 \in \mathsf{dom}(org)) \ \wedge \\ &\quad \forall g_{1,2}, g_{1,2}'. \\ &\quad (\ (\ \mathsf{beforeR}'(g_1, g, r) \ \wedge \ \mathsf{beforeOrAboveR}'(g_2, g, r) \ \wedge \\ &\qquad \langle g_1, g_1' \rangle, \langle g_2, g_2' \rangle \in t \ ) \\ &\qquad \supset \ \mathsf{iso1}(\langle g_1, g_1' \rangle,\ \langle g_2, g_2' \rangle) \ ) \end{aligned}$$

where we set, for brevity:

$$\begin{aligned}
\mathsf{beforeR}'(h,g,r) &\stackrel{\text{def}}{=} h \neq g \wedge \mathsf{beforeR}(h,g,r)\\
\mathsf{beforeOrAboveR}(h,g,r) &\stackrel{\text{def}}{=} \mathsf{beforeR}(h,g,r) \vee \mathsf{aboveR}(h,g,r)\\
\mathsf{beforeOrAboveR}'(h,g,r) &\stackrel{\text{def}}{=} h \neq g \wedge \mathsf{beforeOrAboveR}(h,g,r)
\end{aligned}$$

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.

$$\begin{aligned}
&\mathsf{conPost}(t,g,r)\\
&\equiv \forall g_0.((\mathsf{aroundR}(g_0,g,r) \supset g_0 \in \mathsf{dom}(t)) \;\wedge\\
&\quad \forall g_{1,2}, g'_{1,2}.\\
&\quad (\;(\;\mathsf{beforeOrUnderR}(g_1,g,r) \;\wedge\; \mathsf{aroundR}(g_2,g,r) \;\wedge\\
&\qquad \langle g_1, g'_1\rangle, \langle g_2, g'_2\rangle \in t\;)\\
&\qquad \supset\; \mathsf{iso1}(\langle g_1, g'_1\rangle,\; \langle g_2, g'_2\rangle)\;)
\end{aligned}$$

where, for brevity, we set:

$$\begin{aligned}
&\mathsf{beforeOrUnderR}(h,g,r) \stackrel{\text{def}}{=} \mathsf{beforeR}(h,g,r) \vee \mathsf{underR}(h,g,r).\\
&\mathsf{aroundR}(h,g,r) \stackrel{\text{def}}{=} \mathsf{beforeR}(h,g,r) \vee \mathsf{underR}(h,g,r) \vee \mathsf{aboveR}(h,g,r)
\end{aligned}$$

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:

$$
\begin{array}{l}
\mathsf{gc}(r,i)(u) \stackrel{\text{def}}{=} \forall g, org. \\
\qquad\qquad\qquad \{\ !x = org \ \wedge\ \mathsf{conPre}(org,g,r) \ \wedge\ g \notin \mathsf{dom}(org) \supset \mathsf{sizeR}(g,r,i)\ \} \\
\qquad\qquad\qquad u \bullet g = g' \\
\qquad\qquad\qquad \{\ \sharp^{-x}\{z' \mid \exists z.(\langle z,z'\rangle \in !x \ \wedge\ \langle z,z'\rangle \notin org)\}. \\
\qquad\qquad\qquad\quad (\ (g \in \mathsf{dom}(org) \supset !x = org) \wedge \\
\qquad\qquad\qquad\quad\ \ (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x,g,r) \wedge !x{=}org \cup \langle g,g'\rangle_r^*)\ )\ )\ \}@x
\end{array}
$$

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 $\mathsf{gc}(r,i)(u)$:

$$
\begin{array}{lcl}
GP & \stackrel{\text{def}}{=} & !x = org \wedge \mathsf{conPre}(org,g,r) \wedge g \notin \mathsf{dom}(org) \supset \mathsf{sizeR}(g,r,i) \\
GG & \stackrel{\text{def}}{=} & \sharp^{-x}\{z' \mid \exists z.(\langle z,z'\rangle \in !x \ \wedge\ \langle z,z'\rangle \notin org)\}. \\
 & & \quad (\ (g \in \mathsf{dom}(org) \supset !x = org) \wedge \\
 & & \quad\ \ (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x,g,r) \wedge !x{=}org \cup \langle g,g'\rangle_r^*))
\end{array}
$$

When we apply `Main` to the root node r, we should be able to derive:

$$\sharp^{-x}\{z|g \hookrightarrow z\}.\ !x = \langle g,g'\rangle_r^* \tag{7.9}$$

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 (7.9). For the precondition, 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:

$$!x \ = \ \emptyset \cup \langle r,r'\rangle_r^* \tag{7.10}$$

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 (7.10), 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.

```
1  u : {T}  (abs)
2  lambda r.
3      g' : {T}  (new)
4      new x:= ∅ in
5          r' : {!x = ∅}  (app)
6              m : {T}
7              Main
8              {∀r,n.gc(r,n)(m)}@∅
9              r
10             {!x = ∅ ∧ {!x = ∅}m • r = r'{♯⁻ˣU'.iso(r,r')}@x}@∅
11         {♯⁻ˣU'.iso(r,r')}@x
12     {⋆r'.iso(r,r')}@∅
13 {∀g.{T}u • g = g'{⋆g'.iso(g,g')}@∅}@∅
```

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* $\sharp$].
- From l.11 to l.12, we use the [*New*] rule. Let, for brevity, $M \stackrel{\text{def}}{=} \texttt{Main}$ and $C \stackrel{\text{def}}{=} !x = \emptyset$. Note the sub-derivation l.5–12 means $\{C\}M\{\sharp^{\text{-}x}U'.\mathsf{iso}(r,r')\}$, that is, with i fresh (cf.(7.4)):

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

  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), $x \sharp i$ is $!x$-free via (3.19), page 23, Section 3.6. By [*Inv*] we obtain:

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

  that is

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

  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 : {T}
  NewEntry
{NE(m)}@∅
```

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.

```
1  u : {T}  (rec)
2  mu f.
3    u : {∀j < n. gc(jf)}  (abs)
4    lambda g.
5      g′ : {sizeR(g,r,n)  ∧  !x = org  ∧  conPre(org,g,r)}  (if)
6      if dom(!x,g) then
7        g′ : {g ∈ dom(!x)}
8        get(!x,g)
9        {!x = org  ∧  ⟨g,g′⟩ ∈!x}@∅
10       {GG}@∅
11     else
12       g′ : {g ∉ dom(!x)}  (case)
13       case !g of
14         inl(n):
15           g′ : {T}
16           NewEntry(inl(n), g)
17           {♯⁻ˣg′.(!x = org ∪ ⟨g,g′⟩  ∧  conPre(org,g,r)  ∧
18            atom(!g,n)  ∧  atom(!g′,n))}@x
19           {GG}@x
20         inr(pair<g1, g2>):
21           g′ : {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   {gc(r,i)(u)}@∅
31 {∀r,i.gc(r,i)(u)}@∅
```

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 paragraphs, 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:

$$\begin{aligned} !x = org \ \wedge\ \langle g, g' \rangle \in !x &\Rightarrow \sharp^{\text{-}x}\emptyset.\ (g \in \mathsf{dom}(org) \supset !x = org) \\ &\Rightarrow GG \end{aligned}$$

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 $\mathsf{conPre}(g, org, r)$:

$$\mathsf{conPre}(g, org, r) \wedge !x = org \cup \langle g, g' \rangle \ \wedge\ \mathsf{atom}(!g, n) \ \wedge\ g \notin \mathsf{dom}(org) \tag{7.11}$$

entails GG.

First, for the $\sharp^{\text{-}x}$-free names, a trivial set-theoretic reasoning gives us, under (7.11):

$$\{z' \mid \exists z.(\langle z, z' \rangle \in !x \backslash org)\} \tag{7.12}$$

Second, we derive the body, i.e.

$$\begin{aligned} &(g \in \mathsf{dom}(org) \supset !x = org) \ \wedge \\ &(g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x, g, r) \wedge !x = org \cup \langle g, g' \rangle_r^*)\ ) \end{aligned}$$

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 (7.11). First we note, because g is an atom:

$$\mathsf{underR}(h, g, r) \ = \ \{g\} \tag{7.13}$$

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 (7.11) hence (7.13):

$$\begin{aligned} \forall g_0.(\mathsf{aroundR}(g_0, g, r) \supset g_0 \in \mathsf{dom}(!x)) &\Leftrightarrow \forall g_0.(\mathsf{beforeOrAboveR}(g_0, g, r) \supset g_0 \in \mathsf{dom}(!x))) \\ &\Leftarrow \mathsf{conPre}(org, g, r). \end{aligned}$$

where

$$\mathsf{beforeOrAboveR}(h, g, r) \stackrel{\text{def}}{=} \mathsf{beforeR}(h, g, r) \vee \mathsf{aboveR}(h, g, r)$$

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 (7.11) (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`
 $\{\sharp^{-x} g'.(!x = org \cup \{\langle g, g'\rangle\})\}$
$g' : \{\ldots\}$ $(\mathtt{seq})$
 $\{\mathsf{T}\}$ $(\mathtt{assvar}\,\sharp)$
 `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 $\texttt{inr}(\langle fg_1, fg_2\rangle)$.

1 $\{g \notin \mathsf{dom}(org) \ \wedge\ !x = org \cup \langle g, g' \rangle \ \wedge\ \mathsf{conPre}(org, g, r)\}$
2 $\{\forall i < j.\ \mathsf{gc}(i,r)(f) \ \wedge\ \mathsf{sizeR}(g,r,j)\}$
3 $\{\mathsf{branch}(!g, g_1, g_2)\}$
4 $n : \{\mathsf{T}\}\ \ (\texttt{inr})$
5 **inr**
6 $\quad m : \{\mathsf{T}\}\ \ (\texttt{pair}\,\sharp)$
7 $\quad$ **pair**
8 $\qquad g'_1 : \{\mathsf{T}\}$
9 $\qquad$ **f g1**
10 $\qquad \{GG(g_1, g'_1, org)\}@x$
11 $\qquad g'_2 : \{...\}$
12 $\qquad$ **f g2**
13 $\qquad \{GG1(g_1, g_2, g'_1, g'_2, org')\}@x$
14 $\quad \{GG2(g_1, g_2, m, org')\}@x$
15 $\{GG3(g_1, g_2, n, org')\}@x$

Above we use a parametrised version of GG and its refinements:

$$
\begin{array}{rcl}
GG(g, g', org) & \stackrel{\text{def}}{=} & \{\ \sharp^{\text{-}x}\{z' \mid \exists z.(\langle z, z'\rangle \in !x \ \wedge\ \langle z, z'\rangle \notin org)\}. \\
& & \ (\ (g \in \mathsf{dom}(org) \supset !x = org) \wedge \\
& & \quad (g \notin \mathsf{dom}(org) \supset (\mathsf{conPost}(!x, g, r) \wedge !x{=}org \cup \langle g, g'\rangle_r^*))) \\
GG1(g_1, g_2, g'_1, g'_2, org) & \stackrel{\text{def}}{=} & \{\ \sharp^{\text{-}x}\{z' \mid \exists z.(\langle z, z'\rangle \in !x \ \wedge\ \langle z, z'\rangle \notin org)\}. \\
& & ((g_{1,2} \in \mathsf{dom}(org) \supset !x = org) \ \wedge \\
& & ((g_{1,2} \in \mathsf{dom}(org) \supset !x = org) \supset \\
& & \quad (\mathsf{conPost}(!x, g, r) \wedge !x{=}org \cup \bigcup_{i=1,2} \langle g_i, g'_i\rangle_r^*)) \ \wedge \\
& & ((g_1 \in \mathsf{dom}(org) \wedge g_2 \notin \mathsf{dom}(org)) \supset \\
& & \quad (\mathsf{conPost}(!x, g, r) \wedge !x{=}org \cup \langle g_2, g'_2\rangle_r^*)) \ \wedge \\
& & ((g_2 \in \mathsf{dom}(org) \wedge g_1 \notin \mathsf{dom}(org)) \supset \\
& & \quad (\mathsf{conPost}(!x, g, r) \wedge !x{=}org \cup \langle g_1, g'_1\rangle_r^*))) \\
GG2(g_1, g_2, m, org) & \stackrel{\text{def}}{=} & \exists g'_{1,2}.(m = \langle g'_1, g'_2\rangle \wedge GG1(g_1, g_2, g'_1, g'_2, org)) \\
GG3(g_1, g_2, n, org) & \stackrel{\text{def}}{=} & \exists g'_{1,2}.(n = \texttt{inr}(\langle g'_1, g'_2\rangle) \wedge GG1(g_1, g_2, g'_1, g'_2, org))
\end{array}
$$

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 $org0 \stackrel{\text{def}}{=} org \cup \langle g,g' \rangle$.

$$\mathsf{conPre}(org,g_1,r) \ \wedge \ g_1 \notin \mathsf{dom}(org0) \supset \mathsf{sizeR}(g_1,r,n)$$

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, now 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 now 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:

$$
\begin{aligned}
&\forall h_{1,2}, h'_{1,2}.\\
&(\ (\ \mathsf{beforeR}(h_1, g_2, r)\ \wedge\ \mathsf{beforeOrAboveR}'(h_2, g_2, r)\ \wedge\\
&\quad \langle h_1, h'_1\rangle, \langle h_2, h'_2\rangle \in org1\ )\\
&\quad \supset\ \mathsf{iso1}(\langle h_1, h'_1\rangle,\ \langle h_2, h'_2\rangle)\ )
\end{aligned}
\tag{7.14}
$$

In order to derive the assertion (7.14) 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 now 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:

$$
\begin{aligned}
&\mathsf{conPost}(org1, g1, r)\\
&\equiv \forall h.((\mathsf{aroundR}(h, g, r) \supset h \in \mathsf{dom}(org1))\ \wedge\\
&\quad \forall h_{1,2}, h'_{1,2}.\\
&\quad (\ (\ \mathsf{beforeOrUnderR}(h_1, g_1, r)\ \wedge\ \mathsf{aroundR}(h_2, g_1, r)\ \wedge\\
&\qquad \langle h_1, h'_1\rangle, \langle h_2, h'_2\rangle \in org1\ )\\
&\qquad \supset\ \mathsf{iso1}(\langle h_1, h'_1\rangle,\ \langle h_2, h'_2\rangle)\ ))
\end{aligned}
\tag{7.15}
$$

If (I) is the case, then the isomorphism in (7.15) subsumes that of (7.14) 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:

$$
\begin{aligned}
&\forall h_{1,2}, h'_{1,2}.\\
&(\ (\ \mathsf{beforeR}'(h_1, g, r)\ \wedge\ \mathsf{beforeOrAboveR}'(h_2, g, r)\ \wedge\\
&\quad \langle h_1, h'_1\rangle, \langle h_2, h'_2\rangle \in org1\ )\\
&\quad \supset\ \mathsf{iso1}(\langle h_1, h'_1\rangle,\ \langle h_2, h'_2\rangle)\ )
\end{aligned}
\tag{7.16}
$$

Now 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`.

7.5 Landin's Factorial

Outline. As the last example, we treat Landin's factorial discussed in Section 3. We first reproduce the program from § 3.4 (6) (recall W is an arbitrary closed program of type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$).

$$F \stackrel{\text{def}}{=} \texttt{new}\, x := W \;\texttt{in}\; (\texttt{circFact};\, !x)$$
$$\texttt{circFact} \stackrel{\text{def}}{=} x := \lambda y.\texttt{if}\; y = 0 \;\texttt{then}\; 1 \;\texttt{else}\; y \times (!x)(y-1)$$

Our purpose is to derive:

$$\{\mathsf{T}\}\, F :_u \{\forall n. u \bullet n \searrow n!\} \tag{7.17}$$

After the submission, we realised a simpler and more intuitive rule to derive (7.17) than one in the main paper. We demonstrate derivation of the factorial with the following new rule. First to apply this rule, we derive the following post-condition.

$$\{\mathsf{T}\}\, F :_u \{\nu \star x. \exists g. (Fact(g, u, x) \wedge !x = g)\} \tag{7.18}$$

where $Fact(g, u, x) \stackrel{\text{def}}{=} \{!x = g\} u \bullet n = n! \{!x = g\} @ \emptyset$ and $\{\mathsf{T}\}\, M :_u \{\nu \star x. C'\}$ stands for $\{\mathsf{T}\}\, M :_u \{\exists x. (x \star i \wedge C')\}$ with i fresh (see Notation 5.5). The post-condition said that: *after executing the program, hidden x stores a procedure which would calculate a factorial if x indeed stores that behaviour itself, and that x does store that behaviour.*

The principle for information hiding follows.

$$\frac{\{C\}\, M^{\Delta;\alpha} :_u \{C'\} \quad \mathsf{fv}(C_0) \subset \{xg\} \quad \mathsf{fv}(C_1) \subset \{zy\}}{\{C\}\, M^{\Delta;\alpha} :_u \{\{\mathsf{T}\} u \bullet y = z \{C_1\} @ \emptyset\}} \tag{7.19}$$

where we set:

$$C' \stackrel{\text{def}}{=} \nu \star x. \exists g. (C_0 \;\wedge\; \{C_0\} u \bullet y = z \{C_0 \wedge x \sharp z \wedge C_1\} @ \emptyset)$$

(7.19) says:

Assume that C_0 holds for x and g which are separated with other names and that, under C_0, u never exports x to the outside. Then hiding x means we can neglect C_0 from the specification.

For establishing the soundness of (7.19), observe the conditions above mean: (1) x is never reachable from any reference (except x); (2) u never makes x reachable; and (3) g can never export x once it is hidden (which is direct from the semantics of $\star$). Thus the current (hidden) state satisfying C_0 can only be touched by u, ensuring the satisfaction of the evaluation formula in the conclusion. By applying (7.19) to (7.18), we arrive at (7.17).

Apart from obvious (and easy) extensions of (7.19) such as making each of x and g a vector, we may have a non-trivial precondition in the evaluation formula in the conclusion by stipulating a constraint on u separately, which widens its applicability.

Inference of Landin's Factorial In this subsection, we infer F. Under the typing $x : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat})$, we infer (7.18). We use the following term and assertions.

$$\begin{array}{rcl} V & \stackrel{\text{def}}{=} & \lambda y.\mathtt{if}\ y = 0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1) \\ A(u,g,j) & \stackrel{\text{def}}{=} & \{!x = g\} u \bullet j = j! \{!x = g\}. \\ C(!x,g,i) & \stackrel{\text{def}}{=} & \forall j \leq i.\ A(!x,g,j) \wedge !x = g. \end{array}$$

We use the following derived rules appeared in [20].

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

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

We now infer, letting y be typed with Nat and omitting simple applications of Consequence Rule:

1. $\{C(!x,g,y)\ \wedge\ y = 0\}\ 1 :_m \{m = y!\ \wedge\ C(!x,g,y)\}$ (Simple)
2. $\{C(!x,g,y)\ \wedge\ \neg y = 0\}\ y \times (!x)(y-1) :_m \{m = y!\ \wedge\ C(!x,g,y)\}$ (Simple, App)
3. $\{C(!x,g,y)\}\ \mathtt{if}\ y = 0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1) :_m \{m = y!\ \wedge\ C(!x,g,y)\}$ (IfH)
4. $\{\mathsf{T}\}\ \lambda y.\mathtt{if}\ y = 0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1) :_u$
   $\{\ \forall gy.\{C(!x,g,y)\} u \bullet y = y!\ \{C(!x,g,y)\}\ \}$ (Abs)
5. $\{\mathsf{T}\}\ \lambda y.\mathtt{if}\ y = 0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1) :_u$
   $\{\ \forall j.(u \hookrightarrow i \supset j = x) \wedge \forall gy.\{C(!x,g,y)\} u \bullet y = y!\ \{C(!x,g,y)\}\ \}$
6. $\{x \star i\}\ \lambda y.\mathtt{if}\ y = 0\ \mathtt{then}\ 1\ \mathtt{else}\ y \times (!x)(y-1) :_u$
   $\{\ x \star i \wedge \forall j.(u \hookrightarrow j \supset j = x) \wedge \forall gy.\{C(!x,g,y)\} u \bullet y = y!\ \{C(!x,g,y)\}\ \}$ i fresh (Inv)
7. $\{x \star i\}\ x := V\ \{\ x \star i \wedge \forall yg.\{C(!x,g,y)\} !x \bullet y = y!\ \{C(!x,g,y)\}\ \}$ (Assign)
8. $\{x \star i\}\ \mathtt{circFact}\ \{\ x \star i \wedge \exists g.\ (\forall i.A(!x,g,i) \wedge !x = g)\ \}$ (Consequence)

The application of (Consequence) in Line 8 uses the following entailment:

$$\forall yg.\{C(!x,g,y)\} !x \bullet y = y!\ \{C(!x,g,y)\}\ \supset\ \exists g.\ (\forall i.A(!x,g,i) \wedge !x = g)$$

which is inferred as follows.

$\forall yg.\{C(!x,g,y)\}\ !x\bullet y=y!\ \{C(!x,g,y)\}$

$$\begin{aligned}
&\equiv\ \forall y.\forall g.\{\forall j\lneq y.A(g,g,j)\wedge !x=g\}\ !x\bullet y=y!\ \{\forall j\lneq y.A(g,g,j)\wedge !x=g\}\\
&\equiv\ \forall g.\forall y.\{\forall j\lneq y.A(g,g,j)\wedge !x=g\}\ !x\bullet y=y!\ \{\forall j\lneq y.A(g,g,j)\wedge !x=g\}\\
&\equiv\ \forall g.\forall y.\ ((\forall j\lneq y.A(g,g,j)\ \supset\ \{!x=g\}\ !x\bullet y=y!\ \{!x=g\}))\\
&\supset\ \exists g.\ (\forall y.\ (\forall j\lneq y.A(g,g,j)\ \supset\ \{!x=g\}\ g\bullet y=y!\ \{!x=g\})\wedge !x=g)\\
&\stackrel{\text{def}}{=}\ \exists g.(\forall y.\ (\forall j\lneq y.A(g,g,j)\ \supset\ A(g,g,y))\ \wedge\ !x=g)\\
&\supset\ \exists g.(\ \forall y.A(g,g,y)\ \wedge\ !x=g\ )\\
&\supset\ \exists g.(\ \forall y.A(!x,g,y)\ \wedge\ !x=g\ ),
\end{aligned}$$

as required. In the forth line, we used the following axioms of the evaluation formula from [20].

$$\{A\wedge C\}\ x\bullet y=z\ \{C'\}\equiv A\ \supset\ \{C\}\ x\bullet y=z\ \{C'\}$$
$$\{C_0\}x\bullet y=z\{C_0'\}\supset\{C\}\ x\bullet y=z\ \{C'\}$$
$$\text{where } C\supset C_0 \text{ and } C_0'\supset C'.$$

In the fifth line above, we have used:

$$\forall x.A\quad\supset\quad A[y/x]\quad\equiv\quad\exists x.(A\wedge x=y)$$

which holds for an arbitrary y. From Line 6, by (Deref) and (Seq-Inv), we have:

$$\{x\star i\}\,(\texttt{circFact};\,!x):_u\ \{x\star i\wedge\exists g.(Fact(g,u,x)\wedge\ !x=g)\}$$

where $Fact(g,u,x)\stackrel{\text{def}}{=}A(u,g,n)\stackrel{\text{def}}{=}\{!x=g\}u\bullet n=n!\{!x=g\}$. Immediately by (New) (noting W is a closed term), we have:

$$\{\mathsf{T}\}\,F:_u\ \{\nu\star x.\exists g.(Fact(g,u,x)\wedge\ !x=g)\}\tag{7.20}$$

Soundness of the Inference Rule To obtain the final assertion (7.17), we finally show the soundness of the inference rule for information hiding, (7.19). First we show that (7.20) is matched with the premise of the rule (7.19). For this, we use the following axiom for the evaluation formula.

$$\{C\}u\bullet y=z\{C'\}\ @\,\emptyset\quad\Rightarrow\quad\{C\wedge C_0\}u\bullet y=z\{C'\wedge C_0\}\ @\,\emptyset$$

The axiom said that if a function gives no effect, then we can use the invariance. Applying the above to $Fact(g,u,x)$, we obtain $Fact(g,u,x)\supset Fact'(g,u,x)$ where

$$Fact'(g,u,x)\stackrel{\text{def}}{=}\{!x=g\}u\bullet n=z\{!x=g\wedge x\sharp z\wedge z=n!\}@\emptyset$$

Thus (7.20) maches (7.19) by setting:

$$\begin{aligned}
C&=\mathsf{T}\\
C_0&=\ !x=g\\
C_1&=z=n!\wedge y=n\\
C'&=\nu\star x.\exists g.(C_0\ \wedge\ Fact'(g,u,x))
\end{aligned}$$

In the following, we show the soundness of (7.19) formally. We prove

$$\models \{C\}\, M :_u \{C'\} \quad \text{implies} \quad \models \{C\}\, M :_u \{\{\mathsf{T}\} u \bullet y = z\{C_1\} @ \emptyset\}$$

Let $C_2 = \{\mathsf{T}\} u \bullet y = z\{C_1\} @ \emptyset$. By the definition of the satisfaction, the above statement means that we have to prove:

$$\forall \mathcal{M}.(\mathcal{M} \models C \;\Rightarrow\; \mathcal{M}[u : M] \Downarrow \mathcal{M}' \models C')$$

then

$$\forall \mathcal{M}.(\mathcal{M} \models C \;\Rightarrow\; \mathcal{M}[u : M] \Downarrow \mathcal{M}' \models C_2)$$

The above is equivalent to prove:

$$\forall \mathcal{M}.(\mathcal{M} \models C \;\wedge\; \mathcal{M}[u : M] \Downarrow \mathcal{M}' \models C') \quad \text{implies} \quad \mathcal{M}' \models C_2$$

By the definition of $\mathcal{M}' \models C'$, we can set:

$$\mathcal{M}_0 \models \{C_0\} u \bullet y = z\{C_0 \wedge x \sharp z \wedge C_1\} @ \emptyset \stackrel{\text{def}}{=} C_3 \tag{7.21}$$

where

$$\mathcal{M}_0 = (\nu \tilde{y})(\xi \cdot x : \{x\} \cdot g : W,\ \sigma \cdot [x \mapsto W])$$

By the definition of $\star$, w.o.l.g. we can set $\mathsf{fv}(W) \subseteq \{x\}$ and $g, x \notin \mathsf{fv}(\sigma, \xi)$. Now we have to prove the above $\mathcal{M}_0$ derives the satisfaction:

$$\mathcal{M}_0 \models \{\mathsf{T}\} u \bullet y = z\{C_1\} @ \emptyset$$

Recall the definition of the satisfaction of the evaluation formula in § 4.6, that is:

- $\mathcal{M} \models \{C\} e \bullet e' = x\{C'\}$ if, whenever $\mathcal{M}_0$ is such that $\mathcal{M} \rightsquigarrow \mathcal{M}_0$ and $\mathcal{M}_0 \models C$, we have $\mathcal{M}_0[x : ee'] \Downarrow \mathcal{M}' \models C'$.

We have to prove whenever $\mathcal{M}_0'$ is such that $\mathcal{M}_0 \rightsquigarrow \mathcal{M}_0'$, $\mathcal{M}_0' \models C_0$ and $\mathcal{M}_0'[z : uy] \Downarrow \mathcal{M}' \models C_3$ imply $\mathcal{M}_0'[z : uy] \Downarrow \mathcal{M}' \models C_2$. By assumption, for all such $\mathcal{M}_0'$, we know $\mathcal{M}_0' \models C_0$. Hence we only have to show:

$\mathcal{M}_0'[z : uy] \Downarrow \mathcal{M}' \models C_3$ implies $\mathcal{M}_0'[z : uy] \Downarrow \mathcal{M}' \models C_1$.

By the side condition and the assumption, we know:

$$(\nu \tilde{y})(\xi,\ \sigma) \models C_1$$

This implies $\mathcal{M}_0'[x : uy] \Downarrow \mathcal{M}' \models C_1$.

By applying (7.19) to (7.18), we arrive at (7.17), as desired.

8 Discussions

8.1 Summary

This paper presented a program logic for imperative higher-order functions with full aliasing and local state, with simple syntax and semantic basis. As far as we know, this is the first time a compositional program logic for higher-order programming languages with a general form of local state is developed. Target languages treated in our preceding logics [5, 18–20] do not include local state: the proposed extension allows us to reason about a rich class of behaviour, including dynamic mutable data structures and stored functions with local state. None of the examples with `new` or `ref` in this paper can be treated in our previous logic [5, 18–20]: nor do we know any preceding work by others which offer proof rules capturing the full semantics of `new` and `ref`, or enjoying observational completeness.

In summary, distinguishing features of our logic include:

– An assertional method for a general class of imperative higher-order functions with local state. The logic works with all standard types in full type hierarchy.
– A conservative, minimal syntactic extension of the logic proposed and studied in [5, 18–20] and hence of Hoare logic, despite a significant enrichment of target behaviour.
– A clean semantic basis of the logic, and establishment of soundness and observational completeness of the logic, following our preceding work [5, 18–20].

We have also demonstrated efficient reasoning methods which either expand the treatable class of behaviours (e.g. stored procedures with local state) or relatively improves on the existing ones (e.g. dynamic mutable data structures).

8.2 Related Work.

In a later version, we plan to include a detailed historical survey on logics which treat some aspects of local reference generation. The following discussion focusses on those works which are directly related to the present work.

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 [15]. 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 6, Page 41 (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 $\sharp$-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.

Logics for Dynamic Mutable Data Structure. Dynamically generated data structures have been treated by many researchers, including Luckham and Suzuki [23], Kowaltowski [21], Burstall [9] Morris [30, 31], and more recently by Reynolds, O'Hearn, Bornat and others [7, 35, 41]. It is interesting to observe that these preceding works treat dynamic data structures without treating the full semantics of new name generation. 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 9.

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

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.

Logics for Fresh Names. Freshness of names is recently studied from the viewpoint of formalising binding relations by Pitts and Gabbay [11, 38]; and Miller and Tiu [26]. 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 [26] 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 axiomatisation of reachability including function types (which is one of the important future topics).

Mutable Data Structure in Separation Logic. Separation Logic by Reynolds, O'Hearn, Bornat and others [7, 35, 41] is an extension of Hoare logic which focusses on a conjunction which at the same time asserts on disjointness of memory regions. One of the Reynolds' aims in proposing this logic [40] is to treat dynamic mutable data structure in the context of low-level programming idioms. We first observe two basic differences between the logic proposed in the present paper and Separation logic.

1. Separation Logic does not offer a framework for compositional reasoning of general data structures (which the present logic is intended to present). For example, there are no proof rules for basic data types such as products and sums, not to speak of higher-order functions.
2. Separation Logic does not treat the full semantics of new reference generation (or corresponding "ref" expressions), neither in its proof rules nor in the semantics of logic. In particular, the rule for local variables given by Reynolds and others is the same rule as what Hoare and Wirth presented initially.

A remarkable point is that, in spite of these two points (especially the second one), Separation logic does offer a way to assert and reason about certain forms of dynamically generated and exported mutable structure using separating connectives as a key logical tool. This is done by providing specific reasoning rules for each given data structure, close to the methods taken by other preceding researchers discussed above (except for the use of separating connectives). One of the fundamental differences between two approaches lies in the use of freshness predicates (in several guises) in the present logic, whereas Separation logic uses the separating conjunction. Before analysing this point fully, we give the general idea of this fundamental point. The assertion Reynolds used for specifying the tree copy program (essentially the same program as treated in Section 7.2) in [41] can be written, by combining the separating conjunction $*$ as part of our logic:

$$\{\mathsf{tree}\,\tau\,(x)\}u \bullet x = y\{\mathsf{tree}\,\tau\,(x) * \mathsf{tree}\,\tau\,(y)\} \tag{8.7}$$

where we may recall $C_1 * C_2$ means [41]:

> A heap consists of two disjoint (separate) subheaps, one satisfying C_1 and another satisfying C_2.

Note this does *not* say a data structure in one subheap is reachable from a datum from another subheap (in fact, the notion of reachability in this sense makes the essential use of type structures, as one may see throughout the technical development in the present work (see e.g. §3.6). Thus for (8.7) to assert two trees are disjoint, it is necessary for $\mathsf{tree}\,\tau\,(x)$ to assert existence of all the nodes of the tree (which $\mathsf{tree}\,\tau\,(x)$ does do). In contrast, the assertion we used in Section 7.2 for the same purpose is (omitting he write effects):

$$\{\mathsf{tree}\,\tau\,(x)\}u \bullet x = y\{\star y.\mathsf{tree}\,\tau\,(y)\} \tag{8.8}$$

which says that the procedure u generates a tree of the same shape all of whose nodes are fresh w.r.t. any existing datum. This meaning in turn comes from the semantics of assertions and judgements, where newly created names are always distinct from existing names in the sense that they can never be coalesced with either existing names or other freshly generated names. Including this point, let us offer a comparison of two approaches in some detail. The observations below are based on our exploration of several basic reasoning examples treated by Reynolds, Bornat and others [7, 41], partly reported in Section 7.

1. In typical examples, assertions which use freshness (as in (8.8)) are stronger, and more concise, than those based on separation (as in (8.7)). In fact separation-based assertions do not in general rule out pathological behaviours, as acknowledged in [7, §4]. Concretely, for example, the assertion for the graph copy program in [7] does not offer a full specification of `graphCopy`, at least in the form given in [7]: whereas our assertion for `graphCopy` based on freshness is not only much more concise but also stronger than the one used in [7], offering a full specification of this program (the reason for this difference is discussed in the following items).
2. As noted already, the use of separating connectives in Separation logic means that, in assertions such as (8.7), we need to assert the existence of all mutable cells to stipulate their disjointness. This can be done fairly easily for trees, but not so for dags, and becomes highly intricate for graphs. Recall, in Section 7.4, we did not stipulate any concrete structure of graph in our assertions for the graph copy, but asserted as:

$$\forall g^{\mathit{Tree}}.\{\mathsf{T}\}u \bullet g = g'\{\star g'.\mathsf{iso}(g, g')\}@\emptyset \tag{8.9}$$

Here the notion of isomorphism completely encodes, in a generic and terse way, all that we need for saying "a fresh copy of the same shape as the original graph, the latter of an arbitrary shape, is created". In contrast, Bornat et al. in [7] uses a direct encoding of a graph structure as a formula, which can be written as:

$$\{\mathsf{graph}\,\tau\,(x)\}u \bullet x = y\{\mathsf{graph}\,\tau\,(x) * \mathsf{graph}\,\tau\,(y)\} \tag{8.10}$$

where τ this time encodes a graph's sharing structure. Note this means, in order to reason about a certain graph, their method should encode the whole sharing structure of concrete graphs into a formula. The present logical method does not need any such encoding but captures the general feature through the path predicate and induced isomorphism, as seen in (8.9). While we do appreciate the efforts by Bornat et al. in logically encoding a sharing structure of a graph given in [7], as far as our purpose is asserting and reasoning about the behavioural property of `graphCopy`, it suffices to use the simpler and generic predicate based on paths and isomorphism. We also suspect that representing sharing through navigating paths through graphs fits well with the programmers' intuitive understanding of the behaviour of `graphCopy` and involved data structures.

3. Partly due to the preceding points, derivation of assertions involving data structures with non-trivial sharing (e.g. dags) using separating connectives tends to become notably more complex than in the present logic. For example, intermediate assertions for dagcopy in [7] becomes quite delicate and involved (which the authors of [7] themselves consider "very subtle"), since they should always combine two assertions, one on the original copy and another on the new one.
4. Another source of complexity in the intermediate assertions used in [7] is the need to structure their assertions using separating connectives so that disjointness is expressed as syntactic structure of assertions. When the sharing relationship becomes complex, the generality of classical connectives combined with the use of inequality and $\sharp$, as we do in the proposed logic, offers a more flexible medium.
5. Finally we come back to the point we mentioned already, the difference in the key logical connectives used in two methods for treating dynamic mutable data structures, reflecting their different orientations. As already noted, $C_1 * C_2$ talks about heaps and subheaps, *not* about hereditary (un)reachability between data structures: whereas such predicates $e_1 \star e_2$ *does* directly assert hereditary (un)reachability of two data structures. Thus the separating conjunction $C_1 * C_2$ is semantically quite different from $e_1 \star e_2$: the predicate $\star$ makes the best of type structures in given programs, has a clear behavioural (observational) meaning, and, as seen in Theorem 6.10, Page 51, can be completely reducible to (in)equations when restricted to finite and finite recursive types (which are the data domain separation logic mainly treats).[8] Philosophically, this comes from our orientation to consider complex data structures as being naturally equipped with corresponding type structures, even if typing is not explicit in a given programming language. In this context, it is an interesting subject for further study if the present framework is smoothly extensible to logical treatment of data manipulation in systems-level programming.

[8] We also conjecture that the same elimination result also holds when function types are involved with an extension of the logic.

Assertions and proofs in the proposed logic can also make the best of cleanly articulated structure in the typed target language (e.g. typed pattern matching), leading to concision and clarity. Separation logic mainly focusses on untyped languages and does not treat higher-order functions including stored ones nor standard data types such as products and sums. In summary, we believe it is safe to say that the main methodological appeal of the present framework are to be able to capture, as assertions, essentially arbitrary properties of a general data structure using standard logical connectives; and to be able to use these assertions in inferences, making the most of standard inference methods for the predicate logic with equality, combined with cleanly articulated logical constructs catering for distinct aspects of behaviours representable by higher-order imperative programs. The difference in orientation between the present framework and the method based on separating connectives may also be exploited fruitfully, as we suggested in [5, §10.2].

Recent work by Birkedal et al. [6] 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 involving higher-order functions we studied in Section 7. The interplay between types and program logics itself is however an important topic.

8.3 Future Work.

The present work offers a program logic which covers a basic class of ML-like functional languages. Two issues stand out as further work. The first topic is an application of our logic to other important sequential languages. One is object oriented high-level languages like Java. What is still missing are logical descriptions of classes, objects, subtyping and related constructs. We believe that logical framework to be suitable for integrating these cleanly, guided by the well-known encodings of objects into the typed λ-calculus [3, 36]. At the other end of the scale are low-level languages, like C. Issues here include pointer arithmetic, casting and strong updates. Of particular interest would be C constructs like `malloc`, which implement new reference generation as library functions. Some of the properties of `new` should therefore be derivable in logics for C-like languages [1, 12, 32].

The second topic is the interplay between our logic and the semantic study of local state and higher-order functions. Meyer and Sieber's [25] is an influential investigation of early domain theoretic approaches to modelling imperative higher-order languages with first-order local references. They investigate subtle seven examples of equivalences between programs that are nevertheless distinguished by the models proposed for such languages at the time. Later, various researchers used these examples to argue for the correctness and power of their semantics means [33, 34, 42]. In particular, Stark's Reduced ML [42] studies scope-escaping first-order local references, as a continuation of his research on the nu-calculus. He introduces logical-relation based techniques for operational reasoning and shows them to be complete in the first-order case. This step-up extends smoothly to Reduced ML. Some of Meyer-Sieber examples can be verified, but his semantics is not fully-abstract in the higher-order case. In this context, based on the observation completeness, we are currently working on a stronger property, called *descriptive completeness*: a unique formula can describe the whole behaviour of a program up to the observational congruence. Once we obtain this result, Meyer-Sieber examples can be used as a benchmark to construct effective logical axioms for bridging two different assertions of equivalent programs. This may lead to the use of the proposed logic for checking contextual equivalence on imperative higher-order programs with local state through a logical means.

References

1. C– home page. http://www.cminusminus.org.
2. The Flint project. http://flint.cs.yale.edu/flint/people/.
3. Martn Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
4. Friedrich L. Bauer, Edsger W. Dijkstra, and Tony Hoare, editors. *Theoretical Foundations of Programming Methodology, Lecture Notes of an International Summer School*. Reidel, 1982.
5. Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *ICFP'05*, 2005. www.dcs.qmul.ac.uk/˜kohei/logics.
6. 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.
7. Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation and aliasing. In *Workshop SPACE*, 2004.
8. Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. *Infomation and Computation*, 155(1/2):108–133, 1999.
9. R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7, 1972.
10. Luís Caires and Luca Cardelli. A Spatial Logic for Concurrency (Part I). *I & C*, 186(2):194–235, 2003.
11. Murdoch Gabbay and Andrew Pitts. A New Approach to Abstract Syntax Involving Binders. In *Proc. LICS '99*, pages 214–224, 1999.
12. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI'02*. ACM, 2002.
13. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
14. Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
15. Hoare and Wirth. Axiomatic semantics of pascal. *ACM Trans. Program. Lang. Syst.*, 1(2):226–244, 1979.
16. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
17. Tony Hoare. Notes on data structuring. *Structured Programming*, pages 83–174, 1972.
18. Kohei Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.
19. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202, 2004.
20. Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*. www.dcs.qmul.ac.uk/˜kohei/logics.
21. Tomasz Kowaltowski. Data structures and correctness of programs. *J. ACM*, 26(2):283–301, 1979.
22. Peter Landin. A correspondence between algol 60 and church's lambda-notation. *Comm. ACM*, 8:2, 1965.
23. David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Trans. Program. Lang. Syst.*, 1(2):226–244, 1979.
24. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
25. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *POPL'88*, 1988.
26. Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Transactions on Computational Logic*, to appear.
27. Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.

28. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1):1–77, 1992.
29. Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *TCS*, 114:149–171, 1993.
30. Joseph M. Morris. Assignment and linked data structures. In [4], pages 35–43, 1982.
31. Joseph M. Morris. A proof of the Schorr-Wait algorithm. In [4], pages 44–52. Reidel, 1982.
32. 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.
33. Peter O'Hearn and Robert Tennent. Semantics of local variables. In *Applications of Categories in Computer Science*, pages 217–238.
34. Peter O'Hearn and Robert Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995.
35. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL'04*, 2004.
36. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
37. Benjamin C. Pierce and David N. Turner. Simple type-theoretic foumdations for objecr-oriented programming. *JFP*, 4(2):207–247, 1993.
38. Andy M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.
39. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
40. John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, 2000.
41. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, 2002.
42. Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994.
43. Glynn Winskel. *The formal semantics of programming languages*. MIT Press, 1993.

A Proof of Axioms in Remark 3.3

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 assetion is valid, i.e. is true in any model:*

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x.j \sharp x) \quad \supset \quad (\langle !x \rangle (i^{\alpha} \hookrightarrow x^{\mathsf{Ref}(\beta)}) \;\equiv\; [!x](i^{\alpha} \hookrightarrow x^{\mathsf{Ref}(\beta)}) \;\equiv\; i^{\alpha} \hookrightarrow x^{\mathsf{Ref}(\beta)}) \tag{A.1}$$

which is equivalent to, through dualisation:

$$(\forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x.j \sharp x) \quad \supset \quad (\langle !x \rangle i^{\alpha} \sharp x^{\mathsf{Ref}(\beta)} \;\equiv\; [!x] i^{\alpha} \sharp x^{\mathsf{Ref}(\beta)} \;\equiv\; x^{\mathsf{Ref}(\beta)}) \tag{A.2}$$

PROOF: Assume β is recursively finite and that $\mathsf{Ref}(\beta)$ does not occur in β up to the type isomorphism. Further let:

$$\mathcal{M} \stackrel{\text{def}}{=} (\tilde{y})(\xi \cdot x^{\mathsf{Ref}(\beta)} : \mathbf{j},\, \sigma \cdot \mathbf{j} \mapsto W)$$

Now suppose we have:

$$\mathcal{M} \models \forall \mathrm{X}.\forall j^{\mathrm{X}} \neq x.j \sharp x.$$

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. $\square$