A Logical Analysis of Aliasing in Imperative Higher-Order Functions

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

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

Abstract. We present a compositional program logic for call-by-value imperative higher-order functions with general forms of aliasing, which can arise from the use of reference names as function parameters, return values, content of references and part of data structures. The program logic extends the logic for alias-free imperative higher-order functions in our previous work with new modal operators which serve as building blocks for clean structural reasoning about programs and data structures in the presence of aliasing. The logical status of the new operators is clarified by translating them into (in)equalities of reference names. The logic is observationally complete in the sense that two programs are observationally indistinguishable iff they satisfy the same set of assertions.

Table of Contents

1 Introduction

In high-level programming languages names can be used to indicate either stateless entities like procedures, or stateful constructs such as imperative variables. *Aliasing*, where distinct names refer to the same entity, has no observable effects for the former, but strongly affects the latter. This is because if state changes, that change should affect all names referring to that entity. Consider for example

$$P \stackrel{\text{def}}{=} x := 1;\ y := !z;\ !y := 2,$$

where, following ML notation, $!x$ stands for the content of an imperative variable or *reference* x. If z stores a reference name x initially, then the content of x after P runs is 2; if z stores something else, the final content of x is 1. But if it is unclear what z stores, we cannot know if $!y$ is aliased to x or not, which makes reasoning difficult. Or consider a program

$$Q \stackrel{\text{def}}{=} \lambda y.(x := 1; y := 2).$$

If Q is invoked with an argument x, the content of x ends up as 2, otherwise it stays 1. In these examples, what have been syntactically distinct reference names in the program text may be coalesced during execution, making it difficult to judge which name refers to which store from the program text alone. The situation gets further complicated with higher-order functions because programs with side effects can be passed to procedures and stored in references. For example let:

$$R \stackrel{\text{def}}{=} \lambda(f^{\alpha\times\alpha\Rightarrow\mathsf{Unit}}, x^{\alpha}, y^{\alpha}).\ (\ \mathtt{let}\ z = !x\ \mathtt{in}\ !x := 1;\ !y := 2;\ f(x,y);\ z := 3\ )$$

where $\alpha = \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat}))$. R receives a function f and two references x and y. Its behaviour is different depending on what it receives as f. If we pass a function $\lambda xy.()$ as f, then, after execution, $!x$ stores 3 and $!y$ stores 2. But if the standard swapping function $\mathtt{swap} \stackrel{\text{def}}{=} \lambda ab.\mathtt{let}\ c = !b\ \mathtt{in}\ (b := !a; a := c)$ is passed, the content of x and y is swapped and $!x$ now stores 2 while $!y$ stores 3. Such interplay between higher-order procedures and aliasing is common in many non-trivial programs in ML, C and more recent typed and untyped low-level languages [1, 19, 68].

Hoare logic [31], developed on the basis of Floyd's assertion method [15], has been extensively studied as a verification method for first-order imperative programs, with diverse applications. However Hoare's original proof system is sound only when aliasing is absent [4, 13]: while various extensions have been studied, a general solution which extends the original method to treat aliasing, retaining its semantic basis [17, 32] and tractability, has not been known, not to speak of its combination with arbitrary imperative higher-order functions.

Resuming studies by Cartwright-Oppen and Morris from 25 years ago [10, 11, 56–58], the present paper introduces a simple and tractable compositional program logic for general aliasing and imperative higher-order functions. A central observation in [10, 11, 56–58] is that (in)equations over names, simple as they may seem, are expressive enough to describe general aliasing in first-order procedural languages, provided we distinguish between reference names (which we write x) and the corresponding content

(which we write $!x$) in assertions. In particular, their work has shown that alias robust substitution, written $C\{\!|e/!x|\!\}$ in our notation, defined by:

$$\mathcal{M} \models C\{\!|e/!x|\!\} \quad \text{iff} \quad \mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \tag{1.1}$$

(i.e. an update of a store at a memory cell referred to by x with value e) can be translated into (in)equations of names through inductive decomposition of C, albeit at the expense of an increase in formula size. This gives us the following semantic version of Hoare's assignment axiom:

$$\{C\{\!|e/!x|\!\}\}\, x := e\, \{C\} \tag{1.2}$$

where the pre-condition in fact stands for the translated form mentioned above. The rule subsumes the original axiom but is now alias-robust. As clear evidence of descriptive power of this approach, Cartwright and Oppen showed that the use of (1.2) leads to a sound and (relatively) complete logic for a programming language with first-order procedures and full aliasing [10, 11]: Morris showed many non-trivial reasoning examples for data structures with destructive update, including the reasoning for Schorr-Waite algorithm [56–58].

The works by Cartwright-Oppen and Morris, remarkable as they are, still beg the question how to reason about programs with aliasing in a tractable way. The first issue is calculation of validity in assertions involving semantic substitutions. This is hardly practical because inductive decomposition of $\{\!|e/!x|\!\}$ into (in)equations has been the only syntactic tool available. As demonstrated through many examples by Morris [56–58] and, more recently, Bornat [7], this decomposition should be distributed to every part of a given formula even if that part is irrelevant to the concerned state change, making reasoning extremely cumbersome. As one typical example, if we use the decomposition method for calculating the logical equivalence $C\{\!|\mathsf{c}/!x|\!\}\{\!|e/!x|\!\} \equiv C\{\!|\mathsf{c}/!x|\!\}$ for general C, with c being a constant, we need either meta-logical reasoning, induction on C, or an appeal to a semantic means. Because such logical calculation is a key part of program proving (cf. [31]), practical usability of this approach becomes unclear. The second problem is the lack of structured reasoning principles for deriving precise description of extensional program behaviour with aliasing. This makes reasoning hard, because properties of complex programs often crucially depend on how sub-programs interact through shared, possibly aliased references. Finally, the logics in [10, 11, 56–58] and its successors do not offer a general treatment of higher-order procedures as well as mutable data structures which may store such procedures.

We address these technical issues by augmenting the logic for imperative higher-order functions introduced in [37] with a pair of mutually dual logical primitives called *content quantifiers*. They offer an effective middle layer with clear logical status for reasoning about aliasing. The existential part of the primitives, written $\langle !x \rangle C$, is defined by the following equivalence:

$$\mathcal{M} \models \langle !x \rangle C \quad \stackrel{\text{def}}{\equiv} \quad \exists V.\, (\mathcal{M}[x \mapsto V] \models C) \tag{1.3}$$

The defining clause says: "for some possible content of a reference named x, $\mathcal{M}$ satisfies C" (which may *not* be about the current state, but about a possible state, hence the

notation). Syntactically $\langle !x \rangle C$ does *not* bind free occurrences of x in C. Its universal counterpart is written $[!x]C$, with the obvious semantics.

We mention a couple of notable aspects of these operators. First, introduction of these operators gives us a tractable method for logically calculating assertions with semantic update, solving a central issue posed by Cartwright-Oppen and Morris 25 years ago. We start from the following syntactic representation of semantic update using the well-known decomposition:

$$C\{\!|e/!x|\!\} \quad \equiv \quad \exists m.(\langle !x \rangle (C \wedge !x = m) \wedge m = e). \tag{1.4}$$

From (1.3) and (1.4), the logical equivalence (1.1) is immediate, recovering (1.2) as a syntactic axiom. Not only does $C\{\!|e/!x|\!\}$ now have concrete syntactic shape without needs of global distribution of update operations, but also these operators offer a rich set of logical laws coming from standard quantifiers and modal operators, enabling efficient and tractable calculation of validity while subsuming Cartwright-Oppen/Morris's methods. Intuitively this is because logical calculation can now focus on those parts which do get affected by state change: just like lazy evaluation, we do not have to calculate those parts which are not immediately needed. In later sections we shall demonstrate this point through examples.

Closely related with its use in logical calculation is a powerful descriptive/reasoning framework enabled by content quantification, in conjunction with standard logical primitives. By allowing hypothetical statement on content of references separately from reference names themselves (which is the central logical feature of these operators), complex aliasing situations are given clean, succinct description, combined with effective compositional reasoning principles. This is particularly visible when we describe and reason about disjointness and sharing of mutable data structures (in this sense it expands the central merits of "separating connectives" [61, 66], as we shall discuss in later sections). The primitives work seamlessly with the logical machinery for capturing pure and imperative higher-order behaviour studied in [33, 36, 37], enabling precise description and efficient reasoning for a large class of higher-order behaviour and data structures. The descriptive power of the logic is formally clarified in Section 8 by showing the assertion language is *observationally complete* in the sense that two programs are contextually indistinguishable exactly when they satisfy the same set of assertions.

Third, and somewhat paradoxically, these merits of content quantification come without any additional expressive power: any formula which contains content quantification can be translated, up to logical equivalence, into one without. While establishing this result, we shall also show that content quantification and semantic update are mutually definable. Thus name (in)equations, content quantification and semantic update are all equivalent in sheer expressive power: the laws of content quantification are reducible to the standard axioms for the predicate calculus with equality, which in turn are equivalent to semantic update through its axioms for decomposition. This does not however diminish the significance of content quantification: without identifying it as a proper logical primitive with associated axioms, it is hard to consider its use in reasoning, both in logical calculation and in its applications to structured reasoning for programs and shared data structures in the presence of general aliasing. To our knowledge (cf.Section 10.2), neither the calculation method nor the reasoning principle proposed in the present paper is discussed in the foregoing work.

Structure of Paper. In the rest of the paper, Section 2 briefly summarises the programming language. Section 3 introduces the models of the logic. Section 4 briefly illustrates the key ideas underlying content quantifications, expanding some of the themes noted above. Section 5 introduces the assertion language and its semantics. Section 6 discusses syntactic axioms for the assertion language. Section 7 introduces compositional proof rules for the logic, and discusses structured reasoning principles for programs in the presence of aliasing. Section 8 establishes several key technical properties of the proposed logic: the elimination of content quantifications, the soundness of axioms and proof rules, and observational completeness. Sections 9 gives non-trivial reasoning examples. Section 10 is devoted to discussions on related work and further topics. Our main reference on the predicate logic with equality is Mendelson's textbook [49].

2 Language

2.1 Syntax and Typing

The programming language we shall use in the present study is call-by-value PCF with unit, sums and products, augmented with imperative variables. Assume given an infinite set of *variables* ($x, y, z, \ldots$, also called *names*). The syntax of programs is standard [63], given by the following grammar.

$$\begin{array}{lll} \text{(values)} & V, W & ::= \mathtt{c} \mid x \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M \mid \langle V, W \rangle \mid \mathtt{in}_i(V) \\ \text{(program)} & M, N & ::= V \mid MN \mid M := N \mid !M \mid \mathtt{op}(\tilde{M}) \mid \pi_i(M) \mid \langle M, N \rangle \mid \mathtt{in}_i(M) \\ & & \quad \mid \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \mid \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i \in \{1,2\}} \end{array}$$

Constants ($\mathtt{c}, \mathtt{c}', \ldots$) include unit $()$, natural numbers and booleans. $\mathtt{op}(\tilde{M})$ (where $\tilde{M}$ is a vector of programs) is a standard n-ary first-order operation, such as $+$, $-$, $\times$, $=$ (equality of two numbers or that of reference names), $\neg$ (negation), $\wedge$ and $\vee$. $!M$ dereferences M while $M := N$ first evaluates M and obtains a reference (say x), evaluates N and obtains a value (say V), and assigns V to x. All these constructs are standard, cf. [20, 63]. The notions of binding and α-convertibility are also standard. $\mathsf{fv}(M)$ denotes the set of free variables in M. We use the standard abbreviations such as:

$$\begin{array}{rcll} \lambda().M & \stackrel{\text{def}}{=} & \lambda x^{\mathsf{Unit}}.M & (x \notin \mathsf{fv}(M)) \\ M;N & \stackrel{\text{def}}{=} & (\lambda().N)M & \\ \mathtt{let}\ x = M\ \mathtt{in}\ N & \stackrel{\text{def}}{=} & (\lambda x.N)M & (x \notin \mathsf{fv}(M)) \end{array}$$

In the grammar above, abstraction, recursion and the case construct are annotated by types. Types are ranged over by $\alpha, \beta, \ldots$ and are given by the following grammar.

$$\text{(types)} \quad \alpha, \beta \quad ::= \quad \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \mid \mathsf{Ref}(\alpha)$$

We call types of the form $\mathsf{Ref}(\alpha)$ *reference types*. All others are *value types*. Although the grammar is standard, some points are worth noting in the light of their status in the present theory.

Remark 2.1 (type structure)

1. Both reference types and value types may carry reference types. This allows programs which write to a dereference of a variable (e.g. $!x := 3$), or take a reference as argument and return a reference (e.g. $\lambda x.(x := !x + 1; x)$), leading to a strong form of aliasing illustrated in the Introduction.
2. Having reference types as part of arbitrary data types also allows various "destructive" data structures to be represented. For example, $\mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat}) \times \mathsf{Ref}(\mathsf{Nat})$ is a type for a record whose first component is a pointer to a function of type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ while its second a reference to a natural number.
3. In our previous work on the logic for imperative higher-order functions [37], we treated the sublanguage of the above language which only differs in that a reference type is never carried in other types. Note a small change in types leads to a non-trivial extension of the class of behaviours to be realisable in the language.

Fig. 1 Typing rules.

$$[Var]\ \frac{}{\Gamma, x:\alpha \vdash x:\alpha} \quad [Unit]\ \frac{}{\Gamma \vdash () : \mathsf{Unit}} \quad [Bool]\ \frac{}{\Gamma \vdash \mathtt{b} : \mathsf{Bool}} \quad [Num]\ \frac{}{\Gamma \vdash \mathtt{n} : \mathsf{Nat}}$$

$$[Eq]\ \frac{\Gamma \vdash M_{1,2} : \mathsf{Nat}}{\Gamma \vdash M_1 = M_2 : \mathsf{Bool}} \quad [Abs]\ \frac{\Gamma, x:\alpha \vdash M : \beta}{\Gamma \vdash \lambda x^{\alpha}.M : \alpha \Rightarrow \beta} \quad [Rec]\ \frac{\Gamma, x:\alpha \Rightarrow \beta \vdash \lambda y^{\alpha}.M : \alpha \Rightarrow \beta}{\Gamma \vdash \mu x^{\alpha \Rightarrow \beta}.\lambda y^{\alpha}.M : \alpha \Rightarrow \beta}$$

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

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

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

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

A *basis* is a finite map from names to types. $\Gamma, \Gamma' \ldots$ range over bases. $\mathsf{dom}(\Gamma)$ denotes the domain of Γ, while $\mathsf{cod}(\Gamma)$ denotes the range of Γ. The typing rules are standard [63] and listed in Figure 1, using sequents $\Gamma \vdash M : \alpha$, which say M is typable under a basis Γ.

The following subclass of programs is important in the subsequent development (its original appearance may be [50]).

Definition 2.2 A typed program $\Gamma \vdash M : \alpha$ is *semi-closed* when $\mathsf{cod}(\Gamma)$ only include reference types. We also say M is *semi-closed* when $\Gamma \vdash M : \alpha$ is semi-closed for some Γ and α.

Underlying this definition is the distinction between functional variables and imperative variables: the former *denote*, or *stand for*, values, while imperative variables *refer to*, or *name*, memory cells. We may consider a program with free functional variables to be incomplete: for it to function properly, those variables need be instantiated into concrete (semi-closed) values. Having free reference variables in a program is quite different, since that program needs to keep them free in order to interact with the store. If reference names are λ-abstracted, programs can touch references only after the abstracted names are instantiated into concrete names by application.

In the light of the above discussion, it is often convenient to single out the reference-type part of a basis. We let $\Delta, \ldots$ range over bases whose co-domains are reference types and write $\Gamma; \Delta$ for a basis where Γ maps names to value types (called *environment basis*) and Δ maps names to reference types (called *reference basis*), always assuming $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. Note semi-closed programs can always be written $\Delta \vdash M : \alpha$ (note however writing $\Gamma \vdash M : \alpha$ does *not* mean the lack of reference-typed variables in M: it is only in the notation $\Gamma; \Delta$ that Γ denotes an environment basis).

We often call variables of reference types *reference names* or simply *references*. By abuse of terminology we shall sometimes use "reference" to denote memory cells named by them as far as no confusion arises.

2.2 Dynamics

A *store* $(\sigma, \sigma', ...)$ is a finite map from reference 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. A *configuration* is a pair of a semi-closed program and a store. Then the *reduction* is the binary relation over configurations, written $(M,\sigma) \longrightarrow (M',\sigma')$, generated by the rules below [20, 63]. We use the left-to-right evaluation, but the proposed logic can treat other evaluation strategies, and allows us to infer properties which hold regardless of the choice of evaluation strategies. First, we generate the reduction over programs (not configurations) based on the standard reduction rules of call-by-value PCF, omitting obvious symmetric rules and the rules for first-order operators.

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

The rules for assignment and dereference are given next. Below $\sigma[x \mapsto V]$ denotes the store which maps x to V and otherwise agrees with σ. In both rules we let $x \in \mathsf{dom}(\sigma)$.

$$\begin{array}{rcl}
(!x,\ \sigma) & \to & (\sigma(x),\ \sigma) \\
(x := V,\ \sigma) & \to & ((),\ \sigma[x \mapsto V])
\end{array}$$

Note V in $x := V$ is semi-closed by $x := V$ being semi-closed by the definition of configurations. Finally the contextual rules are given as follows.

$$\frac{M \to M'}{(M,\sigma) \to (M',\sigma)} \qquad \frac{(M,\sigma) \to (M',\sigma')}{(\mathcal{E}[M],\sigma) \to (\mathcal{E}[M'],\sigma')}$$

where $\mathcal{E}[\cdot]$ is the left-to-right evaluation context with eager evaluation for pairs, projection and injection, inductively of the shape (omitting the first-order operators):

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

We write $(M,\sigma) \Downarrow (V,\sigma')$ iff $(M,\sigma) \longrightarrow^* (V,\sigma')$, $(M,\sigma) \Downarrow$ iff $(M,\sigma) \Downarrow (V,\sigma')$ for some V and σ', and $(M,\sigma) \Uparrow$ iff $(M,\sigma) \longrightarrow^n$ for each natural number n.

To have subject reduction, we need to type stores in addition to programs. Write $\Delta \vdash \sigma$ when $\mathsf{dom}(\Delta) = \mathsf{dom}(\sigma) = \mathsf{fv}(\sigma)$ and, moreover, the types of σ match Δ, i.e. for each $x \in \mathsf{dom}(\sigma)$ we have $\Delta \vdash \sigma(x) : \alpha$ iff $\Delta(x) = \mathsf{Ref}(\alpha)$. Note $\mathsf{dom}(\sigma) = \mathsf{fv}(\sigma)$ means reference names which occur in the co-domain of σ also occur in its domain. We set:

$$\Delta \vdash (M,\sigma) \quad \stackrel{\text{def}}{\equiv} \quad (\ \Delta \vdash M : \alpha \wedge \Delta \vdash \sigma\ )$$

For example, given $M \stackrel{\text{def}}{=} !x := 3$ and $\sigma \stackrel{\text{def}}{=} \{x \mapsto y,\ y \mapsto 2\}$, we have

$$x : \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat})),\ y : \mathsf{Ref}(\mathsf{Nat}) \ \vdash\ (M, \sigma)$$

Note that $x : \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat})) \vdash M : \mathsf{Unit}$: however we need a reference stored in x to have a well-typed configuration for this assignment to work.

Proposition 2.3 (subject reduction) *Suppose* $\Delta \vdash M : \alpha$ *and* $\Delta \vdash (M, \sigma)$. *Then* $(M, \sigma) \longrightarrow (M', \sigma')$ *implies* $\Delta \vdash M' : \alpha$ *and* $\Delta \vdash (M', \sigma')$.

Convention 2.4 *Henceforth we restrict the reduction relation to the well-typed one, that is whenever we write* $(M, \sigma) \longrightarrow (M', \sigma')$*, we assume* $\Delta \vdash (M, \sigma)$ *for some* Δ.

2.3 Contexts and Contextual Congruence

Write $C[\cdot]^{\Gamma';\alpha'}_{\Gamma;\alpha}$ for a typed context such that $\Gamma' \vdash C[M] : \alpha'$ whenever $\Gamma \vdash M : \alpha$. We often simply write $C[\cdot]$ for a typed context, leaving their domain and codomain implicit, though formally contexts are always considered to be typed. We often use the following subset of typed contexts.

Definition 2.5 (modest contexts) A typed context $C[\cdot]$ is *semi-closing* if its resulting program is semi-closed. It is *modest* if it is semi-closing and, moreover, it does not abstract any reference name in the hole and the resulting program is semi-closed.

Note a modest context always has the form $C[\,\cdot\,]^{\Delta';\alpha'}_{\Gamma;\Delta;\alpha}$ with $\Delta' \supset \Delta$, and does not collapse names in a program. An example of a modest context is $(\lambda x^{\mathsf{Nat}}.[\,\cdot\,])^{y:\mathsf{Ref}(\mathsf{Nat})\ ;\ \mathsf{Nat}\Rightarrow\mathsf{Bool}}_{x:\mathsf{Nat},y:\mathsf{Ref}(\mathsf{Nat})\ ;\ \mathsf{Bool}}$, which abstracts a value-typed variable x, whereas $(\lambda z^{\mathsf{Ref}(\mathsf{Nat})}.[\,\cdot\,])^{\mathsf{Ref}(\mathsf{Nat})\Rightarrow\mathsf{Bool}}_{z:\mathsf{Ref}(\mathsf{Nat})\ ;\ \mathsf{Bool}}$ is not (since a reference name z in the hole is abstracted).

The contextual congruence for the language, denoted $\cong$, is defined in the standard way, i.e. as the maximum typed congruence satisfying: $\Delta \vdash M_1 \cong M_2 : \mathsf{Unit}$ iff:

$$\forall \sigma.(\ (M_1, \sigma) \Downarrow \quad \Leftrightarrow \quad (M_2, \sigma) \Downarrow\ ) \tag{2.1}$$

Above we assume well-typedness of $(M_{1,2}, \sigma)$ following Convention 2.4, similarly henceforth. The definition is immediately equivalent to saying that $\cong$ is the maximum typed relation satisfying $\Gamma \vdash M_1 \cong M_2 : \alpha$ if and only if:

$$\forall \sigma,\ \text{semi-closing}\ C[\,\cdot\,]^{\mathsf{Unit}}.\ (\ (C[M_1], \sigma) \Downarrow \quad \Leftrightarrow \quad (C[M_2], \sigma) \Downarrow\ ) \tag{2.2}$$

where $C[\,\cdot\,]^{\mathsf{Unit}}$ indicates the resulting type is Unit (with some unspecified reference basis) and (following our convention) we assume well-typedness of configurations (i.e. $\Delta \vdash (C[M_{1,2}], \sigma)$ for some Δ in the above clause). This in turn is equivalent to saying that $\cong$ is the maximum typed relation satisfying $\Gamma \vdash M_1 \cong M_2 : \alpha$ if and only if, again assuming well-typedness:

$$\forall \delta,\ \sigma,\ \text{modest}\ C[\,\cdot\,]^{\mathsf{Unit}}.\ (\ (C[M_1\delta], \sigma) \Downarrow \quad \Leftrightarrow \quad (C[M_2\delta], \sigma) \Downarrow\ ) \tag{2.3}$$

where δ ranges over (possibly non-injective) well-typed substitution of reference names for reference names. This characterisation says all experiments we need to inspect the contextual behaviour of programs are combination of modest contexts and possible ways to collapse reference names. To check the equivalence, if M_1 and M_2 satisfies (2.2), then surely they also satisfy (2.3), by taking appropriate contexts in (2.2). For the other direction, suppose M_1 and M_2 satisfy (2.3). Then (again by taking appropriate contexts) they also satisfy $(C[M_1\xi], \sigma) \Downarrow$ iff $(E[M_2\xi], \sigma) \Downarrow$ for any σ, ξ, and modest $C[\ \cdot\ ]$, where ξ ranges over well-typed substitutions of (both non-reference and reference) variables for semi-closed values. This means we can always replace $M_1\xi$ and $M_2\xi$ in a hole of a context without changing termination behaviour of the whole. Now assume, for a possibly non-modest context $C[\ \cdot\ ]$, $C[M_2]$ converges. Tracing the reductions starting from $C[M_1]$, whenever a duplicate of M_1 is launched into an evaluation context, in the form $M_1\xi$, we replace it with $M_2\xi$, so that when $C[M_2]$ terminates, a residual of $C[M_2]$ (with replacements), say N_2, is identical with that of $C[M_1]$, say N_1, except diplicates of M_1 under λ-abstraction. Since if N_2 has no redex then N_1 cannot have any redex we know $C[M_1]$ also converges. Symmetrically if $C[M_1]$ converges then $C[M_2]$ converges, hence we obtain the property (2.2).

A further characterisation of $\cong$ can be obtained by parameterising $\cong$ with a reference basis. Let us say Γ (which may map both non-reference and reference names) is *covered by* Δ when the maximal reference basis in Γ is a subset of Δ, i.e. when $\Gamma = \Gamma_0; \Delta_0$ such that $\Delta_0 \subset \Delta$.

Definition 2.6 *Let $\Gamma \vdash M_{1,2} : \alpha$ and assume Δ covers Γ. Then we set $\Gamma \vdash M_1 \cong_\Delta M_2 : \alpha$ when the following condition holds.*

$$\forall\ \Delta \vdash \delta,\ \Delta \vdash \sigma,\ \textit{modest}\ C[\ \cdot\ ]^{\Delta;\mathsf{Unit}}_{\Gamma;\alpha}.\ (\,(C[M_1\delta], \sigma) \Downarrow\ \Leftrightarrow\ (C[M_2\delta], \sigma) \Downarrow\,)$$

where $\Delta \vdash \delta$ indicates δ is a well-typed substitutions over $\mathsf{dom}(\Delta)$.

Proposition 2.7 *Let $\Gamma \vdash M_{1,2} : \alpha$. Then $\Gamma \vdash M_1 \cong M_2 : \alpha$ if and only if $\Gamma \vdash M_1 \cong_\Delta M_2 : \alpha$ for each Δ which covers Γ.*

Proof The "only if" direction is immediate. For the "if" direction, suppose $\Gamma \vdash M_1 \cong_\Delta M_2 : \alpha$ for each Δ which covers Γ. We show M_1 and M_2 satisfy the characterisation (2.3). Let $\Gamma = \Gamma_0; \Delta_0$ and suppose $C[M_1\delta] \Downarrow$ for some modest $C[\ \cdot\ ]$. Let, without loss of generality (through injective renaming and weakening), we have $\Delta \vdash \sigma$ such that $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Gamma) = \emptyset$ and $\Delta_0 \subset \Delta$. Since $M_1 \cong_\Delta M_2$ we have $C[M_2\delta] \Downarrow$ as required. $\square$

We may further restrict contexts in Definition 2.7 to evaluation contexts, combined with closing substitutions for non-reference variables on $M_{1,2}$.

3 Models

3.1 Distinction

We use a simple store-based model to describe the computational circumstances we are interested in. For treating aliasing, where distinct reference names may refer to an identical store location, we use equivalence classes of reference names, following Milner, Parrow and Walker [53], rather than introducing an additional set of location labels (as in the dynamic semantics of ML [54]). The notion of distinction distills the idea of aliasing at a high level of abstraction.

Definition 3.1 (distinction) A *distinction over* Δ is an equivalence on $\mathsf{dom}(\Delta)$ relating names of the same type. $\mathcal{D}, \mathcal{D}', \ldots$ range over distinctions. We write $\Delta \vdash \mathcal{D}$ or just $\mathcal{D}^{\Delta}$ to indicate the typing of $\mathcal{D}$, and $\mathsf{dom}(\mathcal{D})$ for $\mathsf{dom}(\Delta)$. $\mathcal{D}$-*identicals* or simply identicals, leaving $\mathcal{D}$ implicit, are the $\mathcal{D}$-equivalence classes. We let $\mathbf{i}, \mathbf{j}, \ldots$ range over identicals. The type of an identical is that of its members. The *full distinction* on Δ is $\{\{x\} \mid x \in \mathsf{dom}(\Delta)\}$ (distinguishing all names in Δ). $\mathcal{D}'$ *extends* $\mathcal{D}$, written $\mathcal{D} \leq \mathcal{D}'$, provided $\mathsf{dom}(\mathcal{D}) \subseteq \mathsf{dom}(\mathcal{D}')$ and for all $x, y \in \mathsf{dom}(\mathcal{D})$ we have $x \mathcal{D} y$ iff $x \mathcal{D}' y$.

Notation 3.2 We write $\mathcal{D}(x) \stackrel{\text{def}}{=} \{y \mid (x,y) \in \mathcal{D}\}$. If x is fresh, $\mathcal{D} + x = \mathcal{D} \cup \{(x,x)\}$ and $\mathcal{D} + (x = y) \stackrel{\text{def}}{=} \mathcal{D} \cup \{(x,a),(a,x) \mid a \in \mathcal{D}(y) \cup \{x\}\}$. Similarly $\mathsf{dom}(\mathcal{D}^{\Delta}) \stackrel{\text{def}}{=} \mathsf{dom}(\Delta)$. $\mathcal{D} - \mathbf{i} \stackrel{\text{def}}{=} \mathcal{D} \setminus \{\mathbf{i} \times \mathbf{i}\}$ (assuming $\mathbf{i}$ is an identical of $\mathcal{D}$). Dually $\mathcal{D} + \mathbf{i} = \mathcal{D} \cup \{\mathbf{i} \times \mathbf{i}\}$ (assuming $\mathbf{i} \cap (\cup \mathcal{D}) = \emptyset$). We write $x \, \mathcal{D} \, y$ or $\mathcal{D} \vdash x = y$ for $(x,y) \in \mathcal{D}$, similarly $\neg x \, \mathcal{D} \, y$ and $\mathcal{D} \vdash x \neq y$. We write $\Delta\mathcal{D}$ for the base which has $\mathcal{D}$-identicals are domain of definition and maps $\mathbf{i}$ to $\Delta(x)$, provided $x \in \mathbf{i}$.

We construct models relative to a distinction. This is fundamental to our concern since the logical description of programs' behaviour generally depends on distinctions. For example, we may wish to say:

> *The command* $x := 1; y := 2$ *results in the state where* x *and* y *store* 1 *and* 2 *respectively,* provided $x \neq y$*, i.e. if* x *and* y *are distinct references.*

For giving a meaning to such description, we need to set up a semantic domain in which x and y are $\mathsf{Ref}(\mathsf{Nat})$-references and in which x and y are distinct. In a different world, where x and y are aliased, we may have the following description:

> *The command* $x := 1; y := 2$ *results in a state where* x *and* y *store* 2*,* provided $x = y$*, i.e. if* x *and* y *denote the same reference.*

which is quite different from the first one.

The semantic domain is constructed from congruence classes of semi-closed programs parameterised by distinctions. This accords with our intuitive understanding of observational indistinguishability under potential aliasing. However, $\cong$ as defined in Section 2.3 is too fine for semantics of programs w.r.t. distinctions. To see why, consider

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

where ω is some diverging term of Unit type. If we consider a distinction in which x and y are equated, then we expect M and $()$ to be contextually equivalent. But $\cong$ says $M \ncong ()$ because it considers arbitrary aliasing: if x and y are distinct, then we do have $M \Uparrow$, so we cannot generally say $M \cong ()$. A programs' behaviours relative to a given distinction $\mathcal{D}$ can be made explicit by using $\mathcal{D}$-identicals as reference names. Then $M\mathcal{D}$ is the program obtained from M by replacing its free names x with the identical $\mathcal{D}(x)$. For example, let:

$$M \quad \stackrel{\text{def}}{=} \quad \texttt{if}\ x = y\ \texttt{then}\ 0\ \texttt{else}\ 1$$

where x and y are of a reference type. If $\mathcal{D}$ equates x and y and, moreover, if $\mathbf{i}$ is the identical containing x, y, then we have:

$$M\mathcal{D} \quad = \quad \texttt{if}\ \mathbf{i} = \mathbf{i}\ \texttt{then}\ 0\ \texttt{else}\ 1$$

$M\mathcal{D}$ immediately converges to 0 unlike M itself, making clear the effect of distinctions on observable behaviour of programs. Similarly, we let stores use identicals as their codomain. As before, $\Delta\mathcal{D} \vdash \sigma$ when σ is typed under Δ substituted by $\mathcal{D}$. Reduction relation etc. stay precisely as before. For example if $N \stackrel{\text{def}}{=} x := 3; !y$ typed under $\Delta \stackrel{\text{def}}{=} x : \mathsf{Ref}(\mathsf{Nat}),\ y : \mathsf{Ref}(\mathsf{Nat})$, and moreover $x, y \in \mathbf{i}$ in a distinction $\mathcal{D}$ for Δ, then we have the following reductions:

$$(\underbrace{\mathbf{i} := 3; \mathbf{i}}_{N\mathcal{D}},\ [\mathbf{i} \mapsto 7]) \quad \to \quad (!\mathbf{i},\ [\mathbf{i} \mapsto 3]) \quad \to \quad (3,\ [\mathbf{i} \mapsto 3]).$$

We hereafter freely use identicals for substitutions and stores in this way. This simplifies our presentation considerably. Using this convention, the following defines the distinction-respecting congruence. Below we say Δ is *complete* if whenever a reference type (say α) occurs in any reference type in its co-domain then α is also in its codomain. So $x : \mathsf{Ref}(\mathsf{Ref}(\alpha))$ is not complete but $x : \mathsf{Ref}(\mathsf{Ref}(\alpha)),\ y : \mathsf{Ref}(\alpha)$ is.

Definition 3.3 ($\mathcal{D}$-respecting congruence) Let Δ be complete and $\mathcal{D}$ be a distinction over Δ. Then we write $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{D}} M_2 : \alpha$ for $\Gamma; \Delta \vdash M_{1,2} : \alpha$ iff, for each modest $C[\,\cdot\,]$ and corresponding store σ, we have:

$$(C[M_1\mathcal{D}], \sigma) \Downarrow \quad \Leftrightarrow \quad (C[M_2\mathcal{D}], \sigma) \Downarrow$$

We often simply write $M \cong_{\mathcal{D}} N$ when typing is clear or irrelevant in a given context.

Immediately $\cong_{\mathcal{D}}$ is a typed equivalence. Observe also $\cong_{\mathcal{D}}$ is nothing but the result of restricting δ in the characterisation of $\cong$ in (2.3) (in Section 2.3, Page 8) to be $\mathcal{D}$-respecting, i.e. we only consider substitutions which collapse names that are equal in $\mathcal{D}$ but leave distinct those which are distinct in $\mathcal{D}$. Conversely, $\cong$ arises from $\cong_{\mathcal{D}}$ by ranging $\mathcal{D}$ over all possible distinctions.

Proposition 3.4 *Let Δ be complete. Then $\Gamma; \Delta \vdash M \cong N : \alpha$ if and only if, for each distinction $\mathcal{D}$ over Δ, we have $\Gamma; \Delta \vdash M \cong_{\mathcal{D}} N : \alpha$.*

which is an easy corollary of the characterisation of $\cong$ in (2.3).

3.2 Models

Models use semi-closed values modulo the $\mathcal{D}$-congruence.

Definition 3.5 An *abstract value of type* $(\mathcal{D};\Delta;\alpha)$ is a $\cong_{\mathcal{D}}$-congruence class of semi-closed values of type α under a basis Δ. We let $\mathbf{v}^{\mathcal{D};\Delta;\alpha}, \mathbf{w}^{\mathcal{D};\Delta;\alpha}, \ldots$ or, omitting types, $\mathbf{v}, \mathbf{w}, \ldots$ range over abstract values. We write $[V]^{\mathcal{D};\Delta;\alpha}$ for an abstract value whose representative is V, and $[\![\alpha]\!]^{\Delta}_{\mathcal{D}}$ for the set of all abstract values of type $\mathcal{D};\Delta;\alpha$.

Abstract values of type $(\mathcal{D};\Delta;\alpha)$ include identicals of $\mathcal{D}$, since reference names are semi-closed values.

Definition 3.6 Let Δ be complete. A *model of type* $\Gamma;\Delta$ is a triple $(\mathcal{D},\xi,\sigma)$ where

1. $\mathcal{D}$ is a distinction on $\mathsf{dom}(\Delta)$;
2. ξ maps $\mathsf{dom}(\Gamma \cup \Delta)$ to abstract values such that each $x \in \mathsf{dom}(\Gamma)$ is mapped to an abstract value of type $\mathcal{D};\Delta;\Gamma(x)$ and each $x \in \mathsf{dom}(\Delta)$ is mapped to $\mathcal{D}(x)$.
3. σ is an *abstract store*, or often simply a *store*, which is a finite map from the identicals of $\mathcal{D}$ to abstract values so that an identical of type $\mathsf{Ref}(\alpha)$ is mapped to an abstract value of type $\mathcal{D};\Delta;\alpha$.

$\mathcal{M}, \mathcal{M}', \ldots$ range over models. If $\mathcal{M} \stackrel{\text{def}}{=} (\mathcal{D},\xi,\sigma)$, then $\mathcal{D}$ (resp. ξ, resp. σ) is the *distinction* (resp. *environment*, resp. *store*) of $\mathcal{M}$. We write $\Gamma;\Delta \vdash \mathcal{M}$ or $\mathcal{M}^{\Gamma;\Delta}$ when $\mathcal{M}$ is a model of type $\Gamma;\Delta$. Given $\mathcal{M}^{\Gamma;\Delta}$, we set $\mathsf{dom}(\mathcal{M}) \stackrel{\text{def}}{=} \mathsf{dom}(\Gamma \cup \Delta)$.

Convention 3.7 (notation for models)

1. We often write (ξ,σ) to denote a model $(\mathcal{D},\xi,\sigma)$ where $\mathcal{D}$ is recovered from ξ in the obvious way.
2. Given a model $\mathcal{M} \stackrel{\text{def}}{=} (\mathcal{D},\xi,\sigma)$ of type $\Gamma;\Delta$, the notation $\mathcal{M}(x)$ with $x \in \mathsf{dom}(\xi \cup \sigma)$ denotes either: (1) $\xi(x)$ if $x \in \mathsf{dom}(\Gamma)$; (2) $\sigma(\mathbf{i})$ if $x \in \mathsf{dom}(\Delta)$ and $x \in \mathbf{i}$; or (3) $\sigma(x)$ if x is a $\mathcal{D}$-identical.
3. $\cong_{\mathcal{M}}$ stands for $\cong_{\mathcal{D}}$ with $\mathcal{D}$ being the distinction of $\mathcal{M}$.

There are two important constructions we use with models. The first is an update of the abstract store of a model with a new abstract value, indicating the effect of assignment command.

Definition 3.8 (semantic update) Let $\mathcal{M}^{\Gamma;\Delta x \mathsf{Ref}(\alpha)} \stackrel{\text{def}}{=} (\mathcal{D},\, \xi,\, \sigma \cdot \mathbf{i} \mapsto \mathbf{w})$ for some $\mathbf{w}$, with $x \in \mathbf{i}$. Further let $\Delta \vdash V : \alpha \in \mathbf{v}^{\mathcal{D};\Delta;\alpha}$. Then the expression $\mathcal{M}[x \mapsto V]$ or, alternatively, $\mathcal{M}[x \mapsto \mathbf{v}]$ or, alternatively, $\mathcal{M}[\mathbf{i} \mapsto \mathbf{v}]$, denotes $(\mathcal{D},\, \xi,\, \sigma \cdot \mathbf{i} \mapsto \mathbf{v})$.

Immediately:

Proposition 3.9 (semantic update) *Let x,y be references used in $\mathcal{M}$.*

1. $\mathcal{M}[x \mapsto V](x) = [V]$.
2. *For each y such that $\mathcal{D} \vdash x \neq y$, we have* $\mathcal{M}[x \mapsto V](y) = \mathcal{M}(y)$.

The following defines extension of models, after a small observation on $\cong_{\mathcal{D}}$.

Proposition 3.10 *Let* $\Gamma;\Delta \vdash M_{1,2} : \alpha$ *and assume* $\Gamma \subset \Gamma'$ *and* $\Delta \subset \Delta'$. *Assume further* $\mathcal{D}$ *on* Δ *extends* $\mathcal{D}'$ *on* Δ'. *Then* $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{D}} M_2 : \alpha$ *iff* $\Gamma';\Delta' \vdash M_1 \cong_{\mathcal{D}'} M_2 : \alpha$.

Proof Since the set of semi-closing contexts one can use in Definition 3.3 do not vary by extending bases and distinctions. □

Notation 3.11 Given $\mathcal{M} = (\mathcal{D},\xi,\sigma)^{\Gamma;\Delta}$, $u \notin \mathsf{fv}(\mathcal{M})$ and $\mathbf{v}^{\mathcal{D};\Delta;\alpha}$, we write $\mathcal{M}\cdot u : \mathbf{v}$, or often $(\xi \cdot u : \mathbf{v},\sigma)$, for a model that extends $\mathcal{M}$ by one entry with abstract value $\mathbf{v}$. Formally we set $\mathcal{M}\cdot u : \mathbf{v}$ to be a model $\mathcal{M}'$ such that:

1. If α is a value type then $\mathcal{M}' = (\mathcal{D},\xi\cdot u : \mathbf{v},\sigma)^{\Gamma;\Delta}$; and
2. If α is a reference type then, with $\mathbf{w} \stackrel{\text{def}}{=} \mathbf{v}\cup\{u\}$, $\mathcal{M}' = (\mathcal{D}-\mathbf{v}+\mathbf{w},\xi\cdot u:\mathbf{w},\sigma[\mathbf{w}/\mathbf{v}])^{\Gamma;\Delta}$ where $\sigma[\mathbf{w}/\mathbf{v}]$ is defined as $\emptyset[\mathbf{w}/\mathbf{v}] = \emptyset$ and $\sigma\cup[\mathbf{i}\mapsto\mathbf{w}'][\mathbf{w}/\mathbf{v}] = \sigma[\mathbf{w}/\mathbf{v}]\cup[\mathbf{i}[\mathbf{w}/\mathbf{v}]\mapsto \mathbf{w}'[\mathbf{w}/\mathbf{v}]]$.

Remark. In Clause 2 above, we cannot have $\mathcal{M}'(u) = \mathbf{v}$ since u itself is ajoined to an existing identical. Note that such $\mathcal{M}'$ is determinied uniquely.

Notation 3.12 Given $x \notin \mathsf{fv}(\mathcal{M}_1)$, $\mathcal{M}_1 \leq_{x:\alpha} \mathcal{M}_2$ stands for $\mathcal{M}_2 \stackrel{\text{def}}{=} \mathcal{M}_1\cdot x : \mathbf{v}^{\alpha}$ in the sense of Notation 3.11, or, with $\mathcal{M}_1 = (\mathcal{D},\xi,\sigma)$, $\mathcal{M}_2 = (\mathcal{D}+\{x\},\ \xi\cdot x:\{x\},\ \sigma\cdot\{x\}\mapsto \mathbf{v}^{\beta})$ where we assume $\alpha = \mathsf{Ref}(\beta)$.

Informally $\mathcal{M}_1 \leq_{x:\alpha} \mathcal{M}_2$ when $\mathcal{M}_2$ is the result of adding exactly one free name to $\mathcal{M}_1$. If α is a reference type, then $\mathcal{M}_2$ either adds an identical $\{x\}$ and a value stored in it, or, alternatively, coalesces x to an existing identical. If on the other hand α is a value type, then there is always a new entry in $\mathcal{M}_2$ which maps x to an appropriate abstract value.

4 Two Modal Operators

Aliasing and Assignment. This section informally motivates content quantification by expanding some of the points briefly touched in Introduction. As illustrated in the Introduction, interaction between aliasing and assignment leads to difficulties in reasoning. For concreteness let's consider the following program.

$$\mathsf{double?} \quad \stackrel{\text{def}}{=} \quad \lambda x^{\mathsf{Ref}(\mathsf{Nat})}.\lambda y^{\mathsf{Ref}(\mathsf{Nat})}.(x :=!x+!x \,;\, y :=!y+!y) \tag{4.1}$$

It is intended to assign the double of the original value for each of two references it receives as arguments. However, as one can easily see, the program will not behave in that way if we apply the *same* reference to this program twice, as in $((\mathsf{double?})r)r$. For suppose r originally stores 2. Then, after execution, we obtain 8 instead of 4 as the new value stored in r. This is because x and y, distinct variables in the procedure body, are coalesced into one variable through repeated arguments. But if we apply two distinct references to double?, it will surely double the content of each argument.

Hoare's principle of logical reasoning [31] dictates that a valid judgement should be derived compositionally, i.e. precisely following the program text. Let us consider how this may be done for double?, focussing on the second command "$y :=!y+!y$". Suppose for concreteness that the content of both x and y is 2 at the entry point. If we were without aliasing, we would have the following specification.

$$\{!x = !y = 2\}\, y :=!y+!y\, \{!x = 2 \wedge\, !y = 4\} \tag{4.2}$$

As x and y can get coalesced into a single name if the arguments are repeated, the assignment to y may affect the content of x. From this viewpoint, (4.2) is *not* a precise specification of the assignment command in the presence of aliasing. So how can we amend (4.2)? Since the postcondition of (4.2) *is* correct if x and y are distinct references, the following gives a natural refinement of (4.2).

$$\{x \neq y \wedge\, !x = !y = 2\}\, y :=!y+!y\, \{!x = 2 \wedge\, !y = 4\} \tag{4.3}$$

The pre-condition $x \neq y$ says that x and y are distinct as names; then $!x = !y = 2$ says that, in spite of this distinction, their content is the same. The notational difference between x (denoting a reference name of type $\mathsf{Ref}(\mathsf{Nat})$) and $!x$ (denoting its content, of type Nat) is fundamental in this assertion. The origin of this differentiation may be traced back to the early days of computing where, in assembly languages, one distinguishes the content of a register R from the content of a memory cell whose address is held in R. At the level of programming languages, it is in typed languages like ML and Haskell, that the need of assigning correct types to expressions have led to strict differentiation between references and their content.

Assignment Axiom with Aliasing. But how can we derive specifications such as (4.3) syntactically? Hoare logic has a simple rule to derive a sound (and indeed best possible) pre-condition for any given a post-condition and an assignment command, elegantly using a syntactic substitution.

$$[\textit{Assign-Org}]\ \frac{-}{\{C[e/!x]\}\, x := e\, \{C\}}\ . \tag{4.4}$$

where $[e/!x]$ is the syntactic substitution replacing occurrences of $!x$ with e in C. However this rule is not valid in the presence of aliasing, as has been known from early times, cf. [4, 13]. For example, in the case of double? and the post-condition $!x = 2 \wedge !y = 4$, we easily calculate, with $\equiv$ indicating logical equivalence:

$$(!x = 2 \wedge !y = 4)[!y+!y\,/\,!y] \quad \equiv \quad !x = 2 \wedge !y = 2 \tag{4.5}$$

which gives the pre-condition in (4.2) in the alias-free setting, rather than what we want, (4.3). Another slightly different Hoare triple for the same command makes the underlying issue more vivid.

$$\{\,(x = y \wedge !y = 1) \;\vee\; (x \neq y \wedge !x = 2)\,\}\;\; y := !y+!y \;\; \{\,!x = 2\,\} \tag{4.6}$$

By informal reasoning, we can see that the judgement (4.6) is operationally reasonable. But if we apply the syntactic substitution to the given post-condition, we obtain;

$$(!x = 2)[!y+!y\,/\,!y] \quad \equiv \quad !x = 2 \tag{4.7}$$

In view of the pre-condition in (4.6), we can see (4.7) precisely leaves out the case when x and y are aliased. Indeed, to obtain the pre-condition of (4.6) from $!x = 2$, the syntactic substitution $[!y+!y\,/\,!y]$ is powerless, since y does not even occur in the postcondition.

Content Quantification. At a semantic level, the distinction-based models introduced in Section 3.2 give a clear idea about how our answer should behave, if not the answer itself. This is through the following logical equivalence, which already appeared in the Introduction informally, but now with a clear technical content. What we are looking for is a formula C_0 such that

$$\mathcal{M} \models C_0 \quad \text{iff} \quad \mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C. \tag{4.8}$$

Above $\mathcal{M}$ represents the state *before* the assignment, while $\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}]$, the update of that state by the value denoted by e, is the state *after* assigning the value denoted by e (calculated in the initial state $\mathcal{M}$) to the memory cell referred to by x. By Definition 3.8, even if x is aliased, $\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}]$ gives the correct update. Thus, in brief, (4.8) says that, for C to hold as the description *after* the assignment $x := e$, the pre-condition C_0 should be such that $\mathcal{M} \models C_0$ holds if and only if $\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C$ holds.

We already know we cannot use the result of syntactic substitution $C[e/!x]$ for C_0 in the presence of aliasing. But why then did it work in the alias-free setting? In brief this is thanks to the following logical equivalence.

$$C[e/!x] \quad \equiv \quad \exists m.(\exists x.(C \wedge !x = m) \wedge m = e) \tag{4.9}$$

Note we cannot simplify the right-hand side into $\exists x.(C \wedge !x = e)$ as far as $!x$ occurs in e; we also note (4.9) has a well-known alternative universal presentation. Using (4.9), we justify (4.8) as follows, assuming $\mathcal{M}$ is alias-free, i.e. its distinction is full.

$$\begin{aligned}
\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \quad &\Leftrightarrow \quad \mathcal{M} \cdot m : [\![e]\!]_{\mathcal{M}} \models \exists x.(C \wedge !x = m) && (4.10)\\
&\Leftrightarrow \quad \mathcal{M} \models \exists m.(\exists x.(C \wedge !x = m) \wedge m = e) && (4.11)\\
&\Leftrightarrow \quad \mathcal{M} \models C[e/!x] && (4.12)
\end{aligned}$$

All are standard logical equivalences under full distinction, clarifying the key status of the logical equivalence (4.9) in Hoare's original assignment axiom.

It is through the analysis of this logical equivalence that we arrive at content quantification. When there is no aliasing, the reference name uniquely represents its content, which is why the existential hiding of x in (4.9) does abstract away its "current" content, described in C. In the presence of aliasing, this is no longer so: indeed hiding a name and hiding a content are now completely different businesses. A concrete example readily illustrates this point. Assume $\mathcal{M}$ identifies x and y and says the corresponding variable stores 1. Further let $C \stackrel{\text{def}}{=} x=y \wedge\ !y=2$. Suppose C is a postcondition of the assignment $x := 2$, as in (4.6). In order to obtain its sound precondition, we reason:

$$\mathcal{M}[x \mapsto 2] \models C \quad \equiv \quad \exists V.\ (\mathcal{M}[x \mapsto V] \models C \wedge\ !x = 2) \quad \equiv \quad \mathcal{M} \models \exists ...(C \wedge\ !x = 2)$$

In the middle, it is the content V of x, rather than x itself, that is existentially abstracted, since abstracting away x lets us lose $x = y$ from the pre-condition, which is far from what we want. Thus the required formula should say: *"For some possible content of x (which may differ from the present one), the model $\mathcal{M}$ satisfies $C \wedge !x = 2$"*. We are thus motivated to fill "..." with what denotes the content of x, not x itself, arriving at the existential content quantification $\langle !x \rangle C$, which says C holds under some (hypothetical) content of x. Its universal counterpart $[!x]C$ arises precisely in the same way through the universal presentation of substitution.

As briefly noted in Introduction, the introduction of the content quantification allows us to re-introduce the equivalence (4.9) that witnessed the correctness of the original Hoare rule, empowered so that it is robust under aliasing.

$$C\{\!|e/!x|\!\} \quad \stackrel{\text{def}}{=} \quad \exists m.(\langle !x \rangle (C \wedge !x = m) \wedge m = e) \tag{4.13}$$

By the semantics of content quantification, we can re-establish the logical equivalence in (4.8), replacing $C[e/!x]$ with $C\{\!|e/!x|\!\}$, literally mimicking (4.10–4.11) above. Thus we now arrive at the following proof rule.

$$[\textit{Assign-basic}]\ \frac{-}{\{\, C\{\!|e/!x|\!\}\,\}\ x := e\ \{C\}} \tag{4.14}$$

The rule subsumes the original rule (4.14) since $C\{\!|e/!x|\!\}$ coincides with $C[e/!x]$ under the full distinction. The semantic status of (4.14) is clear from the semantics of content quantification, offering the weakest precondition of C under arbitrary aliasing.

So we seem to have arrived at an analogue of Hoare's assignment axiom in the presence of full aliasing, by replacing a syntactic substitution by its logical counterpart. But does this new setting help us reason about programs with various forms of aliasing after all? More concretely, can we derive the judgement such as (4.6) easily? Does it allow extensions/generalisation to higher-order programming languages (for example those with the generalised assignment of the form $M := N$ where both M and N are appropriately typed arbitrary expressions)? And can we reason about programs with aliasing tractably and modularly using these quantifications? These are the topics we shall explore in the following sections.

5 Logic (1): Assertions

5.1 Terms and Formulae

This section introduces our logical language and formalises its semantics. The logical language is that of the standard first-order logic with equality [49, Section 2.8] extended with assertions for evaluation and quantification over store content (the latter is the only addition to the logic in [37] and is highlighted below).

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

Here $\star \in \{\wedge, \vee, \supset\}$ and $Q \in \{\forall, \exists\}$. The first set of expressions (ranged over by $e, e', \ldots$) are *terms* while the second set are *formulae* (ranged over by $A, B, C, C' \ldots$). The constants include unit, numerals and booleans. Operators $\mathsf{op}(\tilde{e})$ range over first-order operations from the target programming language, including the standard arithmetical operations over natural numbers. In addition, we have paring, projection and injection operation. The final term, $!e$, denotes the dereference of e, i.e. the content of a store denoted by e.

The predicate $\{C\}\, e \bullet e' \searrow x\, \{C'\}$ is called *evaluation formula* [37], where the name x binds its free occurrences in C'. C and C' are called *(internal) pre/post conditions*. Intuitively, $\{C\}\, e \bullet e' \searrow x\, \{C'\}$ asserts that an invocation of e with an argument e' under the initial state C terminates with a final state and a resulting value, named u, both described by C'. Clearly $\bullet$ is non-commutative.

The remaining two constructs are non-standard quantifications which are at the heart of the present logic. $[!e]C$ is *universal content quantification of e over C*, while $\langle !e \rangle C$ is *existential content quantification of e over C*. In both, e should have a reference type. Informally:

- $[!e]C$ says C holds regardless of the value stored in a memory cell named e.
- $\langle !e \rangle C$ says C holds for some value that may be stored in the memory cell named e.

In both, what is being quantified is the content of a store, *not* the name of that store. In $[!e]C$ and $\langle !e \rangle C$, C is the *scope* of the quantification. Free names in e are not binders, so that we have $\mathsf{fv}(\langle !e \rangle C) = \mathsf{fv}([!e]C) = \mathsf{fv}(e) \cup \mathsf{fv}(C)$. In particular, x is *not* a binder in $[!x]C$ and $\langle !x \rangle C$. Content quantification obeys all standard axioms of modal operators (hence the notation), as we explore in Section 6. Binding in formulae is induced only by standard quantifiers. Formulae are taken up to the induced α-convertibility. $\mathsf{fv}(C)$ (resp. $\mathsf{bv}(C)$) denotes the set of free variables (resp. bound variables) in C.

Terms are typed inductively starting from types for variables and constants and signatures for operators. Recalling that $\Gamma;\Delta$ indicates a map from names to types such that Γ (resp. Δ) is about non-reference types (resp. reference types), we write $\Gamma;\Delta \vdash e : \alpha$ when e has type α such that free names in e have types following $\Gamma;\Delta$; and $\Gamma;\Delta \vdash C$ when all terms in C are well-typed under $\Gamma;\Delta$. We also write $\Theta \vdash C$ if C is well-typed under Θ where $\Theta, \Theta', \ldots$ range over finite maps which may combine two kinds of bases. *Henceforth we only treat well-typed terms and formulae.*

Further notational conventions follow.

Convention 5.1 (assertions)

1. In the subsequent technical development, logical connectives are used with their standard precedence/association, with content quantification given the same precedence as standard quantification (i.e. they associate stronger than binary connectives). For example,
$$\neg A \wedge B \supset \forall x.C \vee \langle !e \rangle D \supset E$$
is a shorthand for
$$((\neg A) \wedge B) \supset (((\forall x.C) \vee (\langle !e \rangle D)) \supset E).$$
$C_1 \equiv C_2$ stands for $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$, stating the logical equivalence of C_1 and C_2. $e \neq e'$ stands for $\neg e = e'$. We also use truth T (definable as $1 = 1$) and falsity F (which is $\neg\mathsf{T}$). The standard binding convention is always assumed.
2. Logical connectives are used not only syntactically but also semantically, i.e. when discussing meta-logical and other notions of validity.
3. If e' is not a variable, $\{C\}\ e_1 \bullet e_2 \searrow e'\ \{C'\}$ stands for $\{C\}\ e_1 \bullet e_2 \searrow x\ \{x = e' \wedge C'\}$, with x fresh; and $\{C\}\ e_1 \bullet e_2\ \{C'\}$ stands for $\{C\}\ e_1 \bullet e_2 \searrow ()\ \{C'\}$.

5.2 Syntactic Substitution and Name Capture

In the standard predicate calculus with quantification and/or equality, direct syntactic substitutions on formulae play a fundamental role in reasoning. Using syntactic substitution needs care in the present assertion language due to implicit capture of names introduced by content quantification and evaluation formulae. The following definition extends the standard notion "e is free for x in C" as found in [49, Section 2.1].

Definition 5.2 We say a term e^α is *free for* x^α *in* C if one of the following clauses holds.

1. e is free for x in $e_1 = e_2$.
2. e is free for x in $\neg C$ if it is free for x in C.
3. e is free for x in $C_1 \star C_2$ with $\star \in \{\wedge, \vee, \supset\}$ if it is free for x in C_1 and C_2.
4. e is free for x in $Qy.C$ with $Q \in \{\forall, \exists\}$ if e is free for x in C, and, moreover, $y \in \mathsf{fv}(e)$ implies $x \notin \mathsf{fv}(C)$.
5. e is free for x in $\{C_1\}\ e_1 \bullet e_2 \searrow y\ \{C_2\}$ if
   - e is free for x in C_1 and C_2,
   - $e = \mathcal{C}[!e']$ implies $x \notin \mathsf{fv}(C_1) \cup \mathsf{fv}(C_2)$, and
   - if $y \in \mathsf{fv}(e)$ then $x \notin \mathsf{fv}(C_1, C_2, e_1, e_2)$.
6. e is free for x in $[!e_0]C$ if
   - e is free for x in C; and
   - $e = \mathcal{C}[!e']$ such that e' and e_0 having the same type, implies $x \notin \mathsf{fv}(C)$.

   Similarly for $\langle !e_0 \rangle C$.

The last two conditions, 5 and 6, concern name capture by quantification over content (as we formalise later, semantics of evaluation formulae says dereferences in pre/post-conditions of an evaluation formula are implicitly universally quantified). The need

for these conditions is similar. We illustrate Clause 6 below. Consider the following assertion:

$$C \quad \stackrel{\text{def}}{=} \quad z = 3 \quad \supset \quad [!y]\, z = 3 \tag{5.1}$$

The assertion is a tautology (i.e. true in any model), saying: if z is 3, then whatever value a cell named y stores, z is still 3. However the following assertion, resulting from (5.1) when we apply the substitution $[!y/z]$ naively, is *not* a tautology (in fact it is unsatisfiable).

$$C[!y/z] \quad \stackrel{\text{def}}{=} \quad !y = 3 \quad \supset \quad [!y]\, !y = 3. \tag{5.2}$$

Note $!y$ is not free for z in C by the content quantification on $!y$. (5.2) says that, if the value currently stored in y is 3, then any value storeable in y coincides with 3, a sheer absurdity. Thus we should prohibit such substitution being applied to C.

In the standard quantification theory, we can always rename bound variables to avoid capture of names. In the present case, what we do is to use (standard) existential quantification to "flush out" all names in dangerous positions. As an example, take C in (5.1). To safely apply $[!y/z]$ to C, we transform C to the following formula, up to logical equivalence:

$$C' \quad \stackrel{\text{def}}{=} \quad \exists z'.(\ (z = 3 \quad \supset \quad [!y]\, z' = 3) \ \wedge \ z = z'\ ) \tag{5.3}$$

Note $!y$ is now free for z in C'. We can now safely perform the substitution:

$$C'[!y/z] \ \stackrel{\text{def}}{=} \ \exists z'.(\ (!y = 3 \quad \supset \quad [!y]\, z' = 3) \ \wedge \ !y = z'\ ) \tag{5.4}$$

which is again a tautology (as it should be). By carrying out such transformations, we can always assume e is free for x in a formula whenever we wish to apply $[e/x]$ to C. Thus we stipulate:

Convention 5.3 *From now on, whenever we write $C[e/x]$ in statements and judgements, we assume e is free for x in C, unless otherwise specified.*

In practical examples, the transformation as given above is rarely necessary.

5.3 Logical Substitution

In the present logic, we extensively use a logical version of substitution defined below.

Definition 5.4 (logical substitutions) We set:

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

with m fresh. Dually we set

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

again with m fresh.

These substitutions may be called *logical content substitutions* or simply *logical substitutions*. We shall derive $C\{\!|e_2/!e_1|\!\} \equiv C\overline{\{\!|e_2/!e_1|\!\}}$ later with the help of appropriate axioms. In practice we mostly use the existential rather than the universal variant of logical substitution.

Logical substitutions behave well in the present theory. In particular, content substitution interacts with content quantification just as syntactic substitution does with conventional quantification (cf. [49, Section 2]). The smooth interplay is aided by suitable axioms for content quantification, to be presented in Section 6. For example, we have $[!x]C \supset C\{\!|e/!x|\!\}$ for any (well-typed) x, e and C, which corresponds to the familiar axiom $\forall x.C \supset C[e/x]$. It should then be no surprise that $C\{\!|e/!x|\!\} \supset \langle !x\rangle C$ also holds, corresponding to the standard entailment $C[e/!x] \supset \exists x.C$. Properties of content quantifications/substitutions will be studied in detail later.

5.4 Semantics of Terms and Formulae

The interpretation of terms is straightforward, given as follows.

Definition 5.5 Let $\Gamma;\Delta \vdash e : \alpha$, $\Gamma;\Delta \vdash \mathcal{M}$ and $\mathcal{M} = (\xi, \mathcal{D}, \sigma)$. Then the *interpretation of e under $\mathcal{M}$*, denoted $[\![e]\!]_{\mathcal{M}}$ is inductively given by the clauses below.

- $[\![x^{\alpha}]\!]_{\mathcal{M}} = \xi(x)$.
- $[\![!e]\!]_{\mathcal{M}} = \sigma([\![e]\!]_{\mathcal{M}})$.
- $[\![\mathsf{c}^{\alpha}]\!]_{\mathcal{M}} = [\mathsf{c}]_{\mathcal{M}}$.
- $[\![\mathtt{op}(\vec{e})]\!]_{\mathcal{M}} = \mathsf{op}([\![\vec{e}]\!]_{\mathcal{M}})$.
- $[\![\langle e, e'\rangle]\!]_{\mathcal{M}} = \langle [\![e]\!]_{\mathcal{M}}, [\![e']\!]_{\mathcal{M}}\rangle$.
- $[\![\pi_i(e)]\!]_{\mathcal{M}} = \pi_i([\![e]\!]_{\mathcal{M}})$.
- $[\![\mathtt{inj}_i(e)]\!]_{\mathcal{M}} = \mathsf{inj}_i([\![e]\!]_{\mathcal{M}})$.

where the operators π_i etc. are abused to denote the corresponding ones over abstract values (which are well-defined because they act congruently on $\cong_{\mathcal{D}}$-congruence classes.

Next we present the satisfaction relation $\mathcal{M} \models C$. All definitions are standard (equality is interpreted as identity on abstract values) except for (1) evaluation formulae which follow [37]; (2) content quantification, where we use semantic updates introduced in Section 3.2; and (3) standard quantification, for which we use the notion of model extension introduced in Section 3.2 (except for quantification over reference names the definition coincides with the standard one). All cases below coincide with the standard definition.

Definition 5.6 Assume $\mathcal{M}^{\Gamma;\Delta} = (\mathcal{D}, \xi, \sigma)$ is a model. Assume in addition that $\Gamma;\Delta \vdash C$. Then we say $\mathcal{M}$ *satisfies* C, written $\mathcal{M} \models C$, if the following conditions hold inductively.

- $\mathcal{M} \models e_1 = e_2$ if $[\![e_1]\!]_{\mathcal{M}} = [\![e_2]\!]_{\mathcal{M}}$.
- $\mathcal{M} \models \neg C$ if $\mathcal{M} \not\models C$, i.e. if it is not the case $\mathcal{M} \models C$.
- $\mathcal{M} \models C_1 \wedge C_2$ if $\mathcal{M} \models C_1$ and $\mathcal{M} \models C_2$.
- $\mathcal{M} \models C_1 \vee C_2$ if $\mathcal{M} \models C_1$ or $\mathcal{M} \models C_2$.
- $\mathcal{M} \models C_1 \supset C_2$ if $\mathcal{M} \models C_1$ implies $\mathcal{M} \models C_2$.

- $\mathcal{M} \models \forall x^{\alpha}.C$ if $\mathcal{M}' \models C$ for each $\mathcal{M}'$ such that $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$.
- $\mathcal{M} \models \exists x^{\alpha}.C$ if $\mathcal{M}' \models C$ for some $\mathcal{M}'$ such that $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$.
- $\mathcal{M} \models \{C\}e \bullet e' \searrow x\{C'\}$ if, for each $\mathcal{M}' \stackrel{\text{def}}{=} (\mathcal{D}, \xi, \sigma')$ of type $\Gamma;\Delta$ such that $\mathcal{M}' \models C$, we have, for some $\mathbf{v}$:
  - $([\![e]\!]_{\mathcal{M}'}[\![e']\!]_{\mathcal{M}'},\ \sigma') \Downarrow (\mathbf{v}, \sigma'')$ and
  - $(\mathcal{D},\ \xi{\cdot}x{:}\mathbf{v},\ \sigma'') \models C'$.
- $\mathcal{M} \models [!e^{\mathsf{Ref}(\alpha)}].C$ if $[\![e]\!]_{\mathcal{M}} = [x]_{\mathcal{M}}$ and for each $V \in [\![\alpha]\!]^{\Delta}_{\mathcal{D}}$ we have $\mathcal{M}[x \mapsto V] \models C$.
- $\mathcal{M} \models \langle !e^{\mathsf{Ref}(\alpha)} \rangle .C$ if $[\![e]\!]_{\mathcal{M}} = [x]_{\mathcal{M}}$ and some $V \in [\![\alpha]\!]^{\Delta}_{\mathcal{M}}$ exists with $\mathcal{M}[x \mapsto V] \models C$.

Some observations follow.

1. The clauses for quantifications give the standard definition when α is a value type: when it is a reference type, it allows x to be aliased to existing identicals (for detailed illustration, see Appendix A).
2. In the clause for evaluation formulae, $(([\![e]\!]_{\mathcal{M}'}[\![e']\!]_{\mathcal{M}'}), \sigma') \Downarrow (\mathbf{v}, \sigma'')$ stands for the evaluation of abstract values induced by that which comes from the corresponding concrete values and a store, saying: *a value (say W) in $[\![e]\!]_{\mathcal{M}'}$ and a value (say W′) in $[\![e']\!]_{\mathcal{M}'}$ is applied under a concrete store corresponding to σ' and a distinction $\mathcal{D}$ and results in a value and a store whose $\cong_{\mathcal{D}}$-congruence classes are $\mathbf{v}$ and σ''.*
3. The clause for $\mathcal{M} \models \langle !e^{\mathsf{Ref}(\alpha)} \rangle C$ says: in order to see if $\langle !e^{\mathsf{Ref}(\alpha)} \rangle .C$ holds in $\mathcal{M}$, we evaluate e to see which identical it denotes. Let it be $\mathbf{i}$. Then the value stored at $\mathbf{i}$ in $\mathcal{M}$ is irrelevant, all we need to know is if there is some value $V \in [\![\alpha]\!]_{\mathcal{M}}$ such that $\mathcal{M}[x \mapsto V]$ satisfies C.

5.5 Examples of Assertions

Dereference. "$y = 6$" says y is equal to 6. Actually we should write this formula "$y^{\mathsf{Nat}} = 6$" with a type annotation on y, but we often omit such obvious or irrelevant type annotations. A program which satisfies this assertion is 6 itself, named y. Another program which satisfies this assertion is $3+3$, again named y.

Next, "$!y = 6$", again omitting the type annotation, says the content of a memory cell named y is equal to 6 (with y typed as $\mathsf{Ref}(\mathsf{Nat})$ formally). If both z and y refer to the same cell, and if the above assertion holds, then $!y = 6$ entails $!z = 6$. In the model, distinctions account for such aliasing.

A reference can store another reference in the target programming language, which is easily describable as assertions. For example, "$!!y = 6$" (with y formally typed as $\mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat}))$) says that the content of a memory cell whose name is stored in another memory cell y, is equal to 6. Any store where a memory cell named y stores some reference name which in turn names another cell that stores 6, satisfies this assertion. Of course neither of these cells may be aliased.

Evaluation Formulae. We next consider an assertion which involves an evaluation formula. The following assertion can be considered as a specification for the program $\lambda z.z := !z \times 2$, named u.

$$\forall x.\forall i.\{!x = i\}\, u \bullet x\, \{!x = 2 \times i\} \tag{5.5}$$

We recall from Convention 5.1 that the formula "$\{!x = i\}\, u \bullet x\, \{!x = 2 \times i\}$" is an abbreviation for "$\{!x = i\}\, u \bullet x \searrow z\, \{z = () \wedge !x = 2 \times i\}$". The returned unit can be omitted because it is insignificant — $()$ is the unique inhabitant of type Unit, so no other values are possible. The shorthand also conforms nicely to standard Hoare triples. The assertion says that u, which denotes a procedure, always doubles the content of an argument, which should be a reference storing a natural number.

The following assertion refines (5.5), giving a more focussed specification for $\lambda z.z := !z \times 2$. It shows how we can use inequalities on reference names in combination with an evaluation formula to assert a strong property of imperative behaviour.

$$\forall x, y, i, j.\ \{!x = i \wedge x \neq y \wedge !y = j\}\, u \bullet x\, \{!x = 2 \times i \wedge x \neq y \wedge !y = j\}. \tag{5.6}$$

The assertion says that, in addition to the property already stated in (5.5), the program guarantees that x is the only reference it may alter.[3] It will turn out to be convenient to use the following abbreviation for (5.6).

$$\forall x, i.\ \{!x = i\}\, u \bullet x\, \{!x = 2 \times i\}\ @\, x \tag{5.7}$$

Such assertions are called *located assertions*. (5.7) says the same thing as (5.6) but more concisely.

Content Quantification (1): Existential. We now consider assertions which involve content quantification and substitution. These examples demonstrate how a complex situation can be written down concisely using these new forms of quantifications.

First, as a very simple example, consider an assertion

$$\langle !y \rangle\, !y = 1 \tag{5.8}$$

where we have omitted to annotate y with $\mathsf{Ref}(\mathsf{Nat})$. The assertion claims that, for some possible content of a cell named y, the assertion $!y = 1$ is true. More concretely, it says:

In some possible state, the reference cell y (of type $\mathsf{Ref}(\mathsf{Nat})$*) may store 1.*

In a hypothetical state, the content of a store may differ from the current one. Since we can surely hypothesise such a state, the statement is always true, so that (5.8) is a tautology.

Next we consider an assertion which, by a trivial transformation, is $(!x = 2)\{\!|m/!x|\!\}$ and may be considered as the precondition for having "$!x = 2$" after executing the assignment "$x := m$".

$$\langle !x \rangle\, (!x = 2 \wedge\ !x = m). \tag{5.9}$$

[3] In (5.6), y and j refer to an arbitrary reference and its content, which cannot be typed by the monomorphic type discipline. Formally we add a ML-like implicit polymorphism to our assertion language (but not to the programming language), see Section 5.6 for details.

A model $\mathcal{M}$ satisfies this assertion if and only if there is a model $\mathcal{M}'$ which is exactly like $\mathcal{M}$ except possibly for the value stored at a memory cell referred to by x and which satisfies, at that memory cell, $!x = 2 \wedge\ !x = m$. What this means is that the assertion above does not talk about what is stored at x. All it says that it is possible to fill a memory cell named x such that we have both $!x = 2$ and $m = !x$. Note this entails m and 2 should be equal (which is a stateless fact). As this does not claim anything about the content of x, only about its possible content, the only thing being asserted here is that m denotes 2 in the model, hence (5.9) is logically equivalent to $m = 2$.

The next two examples show how equality and inequality over names interact with existential content quantification. First, consider

$$\langle !x \rangle\, (x = y \wedge\ !y = 1). \tag{5.10}$$

It hides the content of x, but it also claims that both x and y name the same memory cell. This latter information is not existentially abstracted by the content quantification (since it is about x and y, not their content). Since x and y denote the same cell, the quantification not only hides the content of x but also the content of y. This is an immediate consequence of the standard equality law [49, Section 2.8], "$x = y \wedge C(x,x) \supset C(x,y)$" where $C(x,y)$ rewrites some of the free occurrences of x in $C(x,x)$ (to be precise this rule is applicable since x is free for y in "$x = y \wedge !y = 1$"). Hence (5.10) is logically equivalent to $x = y$.

The next example uses inequality instead of equality in the assertion above.

$$\langle !x \rangle\, (x \neq y \wedge\ !y = 1). \tag{5.11}$$

Again $x \neq y$ is independent from any content quantification. Because of this inequality, we also know that the content of y is independent from that of x: in other words, $\langle !x \rangle$ does not hide the content of y, hence (5.11) is logically equivalent to $x \neq y \wedge\ !y = 1$, i.e. we can take off the content quantification completely.

Now consider changing "$!x = m$" in (5.9) into "$!y = m$", obtaining:

$$\langle !y \rangle\, (!x = 2 \wedge\ !y = m) \tag{5.12}$$

which is the same thing as "$(!x = 2)\{\!|m/!y|\!\}$" up to logical equivalence. Thus (5.12) may be considered as representing the precondition for arriving at "$!x = 2$" after executing the assignment command "$y := m$". From our previous examples, we know there are two cases to consider.

1. If $x = y$, then the content quantification hides both $!y$ and $!x$ (which are one and the same thing), hence the formula says $m = 2$.
2. If $x \neq y$, then $!y$ is hidden so m cannot be determined, while x is not hidden. Hence in this case the formula says $!x = 2$.

In summary, (5.12) is equivalent to the conjunction of $x = y \supset m = 2$ and $x \neq y \supset !x = 2$ (or, equivalently, the disjunction of $x = y \wedge m = 2$ and $x \neq y \wedge !x = 2$). This is quite different from, say, $\exists i.(!y = i \wedge !x = 2 \wedge\ m = !y)$.

Content Quantification (2): Universal. The following two examples show the use of universal content quantification. In general, $[!x]C$ says that C does not mention anything substantial about the content of (a memory cell named by) x. As a first example, consider the assertion

$$[!x]\,!y = 3 \tag{5.13}$$

assuming x is typed with $\mathsf{Ref}(\mathsf{Nat})$. By definition, (5.13) literally says the following.

Whatever natural number we may store in x, the number stored in y is 3.

When can this be satisfied? Clearly the content of y should be 3. Moreover, this should be true when we store in x something different from 3, say 0, so it also says x and y name distinct memory cells. Thus the assertion (5.13) is logically equivalent to "$x \neq y \wedge\, !y = 3$". From this we can easily see $[!x]\,!x = 3$ is equivalent to falsity since it should mean $x \neq x \wedge\, !x = 3$ which is impossible.

The universal content quantification offers a powerful tool when combined with located evaluation formulae. Recall the located assertion (5.7) which is for the program $\lambda z.z := !z \times 2$, reproduced below:

$$\forall x, i.\ \{!x = i\}\, u \bullet x\, \{!x = 2 \times i\}\ @\, x \tag{5.14}$$

(5.14) says the program leaves untouched any property of a memory cell except for what it receives as an argument. So, for example, if the program is fed with x, then, after running, it leaves an even number in y still even, as far as y is distinct from x.

$$\forall x, i.\ \{!x = i \ \wedge\ [!x]\mathit{Even}(!y)\}\, u \bullet x\, \{!x = 2 \times i \ \wedge\ [!x]\mathit{Even}(!y)\}\ @\, x \tag{5.15}$$

which is a consequence of (5.14) (hence holds for $\lambda z.z := !z \times 2$ named u), remembering $[!x]\mathit{Even}(!y)$ says the content of y is even regardless of the content of x, that is we have *both* $\mathit{Even}(!y)$ and $y \neq x$. The entailment from (5.14) to (5.15) is the analogue of the standard invariance rule, albeit it is purely logical — the notorious side condition, that a program does not touch a variable, is directly asserted.

Another occasion where combination of evaluation formulae and universal content quantification becomes useful is when we wish to perform the analogue of the consequence rule at the level of evaluation formulae. Here it is essential to be able to have hypothetical assertions on state, as the following example shows.

$$!x = 2 \ \wedge\ [!x]\,(!x = 3 \supset \mathsf{Odd}(!x)) \ \wedge\ \{\mathsf{Odd}(!x)\}\, u \bullet ()\, \{\mathsf{Even}(!x)\} \tag{5.16}$$

It says that the current content of a memory cell named x is 2, the assertion $!x = 3 \supset \mathsf{Odd}(!x)$ should hold in all hypothetical situations about the content of x, and that invoking at u will turn an odd content of x to an even one. It is thus natural to conclude (formally using axioms discussed later):

$$!x = 2 \ \wedge\ [!x]\,(!x = 3 \supset \mathsf{Odd}(!x)) \ \wedge\ \{!x = 3\}\, u \bullet ()\, \{\mathsf{Even}(!x)\} \tag{5.17}$$

By comparing (5.16) with the following assertion we can see the role of the content quantification in the assertion above.

$$!x = 2 \ \wedge\ (!x = 3 \supset \mathsf{Odd}(!x)) \ \wedge\ \{\mathsf{Odd}(!x)\}\, u \bullet ()\, \{\mathsf{Even}(!x)\}$$

But if $!x = 2$ holds then the assertion "$!x = 3 \supset \mathsf{Odd}(!x)$" (which is now also about the current state) is always true, hence we can no longer reach (5.17).

Assertions for Double. We continue with assertions for three short programs, one of which, the "questionable double", already appeared in Section 4. This is followed by the classical "swap" and then by assignment to a circular reference, all of which are substantially affected by aliasing. In Section 9, we shall show the given programs do satisfy these specifications using the proof rules of the logic (to be introduced in Section 7).

First we treat the questionable double, whose definition is reproduced from Section 4) in the following.

$$\texttt{double?} \quad \stackrel{\text{def}}{=} \quad \lambda(x,y).(x:=!x+!x;y:=!y+!y)$$

The program takes a pair of two names, which is syntactic sugar for two subsequent λ-abstractions, can be given the following specification.

$$\forall x,y,i,j.\{x \neq y \wedge !x = i \wedge !x = j\}u \bullet (x,y)\{!x = 2i \wedge !x = 2j\}$$

The assertion is silent on what happens when $x = y$. The next specification, which is also satisfied by `double?`, talks just about this case.

$$\forall x,y,i,j.\ \{x = y \wedge !x = i\}u \bullet (x,y)\{!x = 4i\}$$

Combining these two, we get a fuller specification.

$$\forall x,y,i,j.\ \{!x = i \wedge\ !y = j\}u \bullet (x,y)\{(x = y \wedge !x = 4i) \vee (\ x \neq y \wedge !x = 2i \wedge !y = 2j)\}$$

The specification for `double?` suggests how we can refine this program so that it is robust with respect to aliasing. This is done by "internalising" the condition $x \neq y$ as follows.

$$\texttt{double!} \quad \stackrel{\text{def}}{=} \quad \lambda(x,y).\texttt{if } x = y \texttt{ then } x:=!x+!x \texttt{ else } x:=!x+!x;y:=!y+!y$$

This meets the "expected" specification:

$$\forall x,y,i,j.\ \{!x = i \wedge !y = j\}\ u \bullet (x,y)\ \{!x = 2i \wedge !y = 2j\} \tag{5.18}$$

If we use a located assertion, we can further refine (5.18) to:

$$\forall x,y,i,j.\ \ \{!x = i \wedge !y = j\}\ u \bullet (x,y)\ \{!x = 2i \wedge !y = 2j\}\ @\,xy \tag{5.19}$$

The quantification of x and y extends to the whole formula, including $@xy$. (5.19) says that we can guarantee, in addition to its functional property described above, that no reference cells other than those passed as arguments to this program are modified.

Assertions for Swap. A classical example for reasoning about aliasing (cf. [10, 11, 44]) is the swapping routing:

$$\texttt{swap} \quad \stackrel{\text{def}}{=} \quad \lambda(x,y).\texttt{let } z = !x \texttt{ in } (x:=!y;y:=x)$$

It receives two references of the same type and exchanges their content. The assertion which specifies the behaviour of `swap` named u is:

$$\mathsf{Swap}(u) \quad \stackrel{\text{def}}{=} \quad \{!x = i \wedge !y = j\} u \bullet \langle x,y \rangle \{!x = j \wedge !y = i\}.$$

where x, y, i and j are all universally quantified as before. Again we can refine the program using a located assertion:

$$\mathsf{Swap}(u) \quad \stackrel{\text{def}}{=} \quad \{!x = i \wedge !y = j\}\, u \bullet \langle x,y \rangle\, \{!x = j \wedge !y = i\}\, @\, xy$$

which gives the full specification for `swap` in the sense that it characterises behaviour of programs up to $\cong$.

Circular Reference. Finally an assignment to a circular reference can be asserted as follows.

$$\{\ !x = y \wedge !y = x\ \} \quad x\ := !!x \quad \{\ !x = x\ \} \tag{5.20}$$

Since originally x and y refer to each other, after putting $!!x$ to x, x should be pointing to itself. Correct treatment of circular references is often significant in low-level systems programming: as seen above, the proposed logical framework can treat programs with circular references without no extra machinery.

5.6 Located Evaluation Formulae

Before moving to the next section, we present the formal definition of located evaluation formulae, motivated by examples in the previous subsection. As noted in the Footnote 3 in Section 5.5, we add type variables to the grammar of types used in assertions and the universal quantification over types and its dual to the grammar of assertions:

$$\begin{array}{lcl} \alpha & ::= & \ldots \mid \mathrm{X} \\ C & ::= & \ldots \mid \forall \mathrm{X}.C \mid \exists \mathrm{X}.C \end{array}$$

Types are taken syntactically, so that the satisfaction for the new constructs are simply given as follows (let *ClTp* be the set of closed types).

$$\begin{array}{lll} \mathcal{M} \models \forall \mathrm{X}.C & \text{if} & \forall \alpha \in \mathit{ClTp}.\, \mathcal{M} \models C[\alpha/\mathrm{X}] \\ \mathcal{M} \models \exists \mathrm{X}.C & \text{if} & \exists \alpha \in \mathit{ClTp}.\, \mathcal{M} \models C[\alpha/\mathrm{X}] \end{array}$$

Using these constructs, we define located evaluation formulae as follows. As usual we use the vector notation $\tilde{x}$ under the assumption names in $\tilde{x}$ are pairwise disjoint, to denote a finite set of names (on which we freely perform permutations as well as set union, set difference, etc.). Below for readability we let $\tilde{g}$ stand for a finite set of logical terms of reference types.

Definition 5.7 (located assertions) The notation $\{C\}e \bullet e' \searrow x\{C'\}@\tilde{g}$ (called *evaluation formula located at* $\tilde{g}$) with each g_i not containing a dereference, denotes the following formula, with X, y and j fresh.

$$\forall \mathrm{X}.\, \forall y^{\mathsf{Ref}(\mathrm{X})}.\, \forall j^{\mathrm{X}}.\, \{C \wedge y \neq \tilde{g} \wedge\ !y = j\} e \bullet e' \searrow x\{C' \wedge\ !y = j\}$$

where $y \neq \tilde{g}$ stands for the conjunction of inequations $\wedge_{e_i \in \tilde{g}} y \neq e_i$. The set of terms $\tilde{g}$ in $\{C\}e \bullet e' \searrow x\{C'\}@\tilde{g}$ is called *write effect* or *modified set*.

It is often sufficient, for example in axioms, to use a set of names $\tilde{w}$ instead of expressions $\tilde{g}$, though sometimes the general case is needed. As we have already encountered in the examples in Sections 5.5, $\{C\}e \bullet e' \searrow x\{C'\}@\tilde{w}$ indicates not only that the invocation of e with argument e' starting from the initial state described by C terminates with the final state C', the latter also describing the resulting value named x, but also during this evaluation only references named by $\tilde{w}$ can be changed. In Definition 5.7, we can replace $y \neq \tilde{w} \wedge !y = j$ by $[!\tilde{w}]\,!y = j$ without changing meaning. Basic observations on located assertions follow.

Proposition 5.8 (located assertions) *The following assertions are tautologies.*

$$\{C\}e \bullet e' \searrow x\{C'\}@\tilde{w} \quad \supset \quad \{C\}e \bullet e' \searrow x\{C'\}@\tilde{w} \cup \tilde{v}$$

$$\forall j.\{C \wedge !u = j\}e \bullet e' \searrow x\{C' \wedge !u = j\}@\tilde{w} \quad \supset \quad \{C\}e \bullet e' \searrow x\{C'\}@\tilde{w} \backslash u$$

$$\forall j.\{C \wedge !u = j \wedge u \neq \tilde{w}\}e \bullet e' \searrow x\{C' \wedge !u = j\}@\tilde{w}u \quad \equiv \quad \{C\}e \bullet e' \searrow x\{C'\}@\tilde{w}$$

where, in the second line, $\tilde{w} \backslash u$ denotes the result of taking off u from $\tilde{w}$, and, in the second/third lines, j should be fresh.

We often call the first two implications *weakening* and *thinning* for located assertions. Located assertions are extensively used in the subsequent technical development.

Fig. 2 Axioms and Rule of Inference for content quantification.

$$\begin{array}{ll} \text{(CA1)} & [!x]\,(C_1^{\text{-}!x} \supset C_2) \supset (C_1 \supset [!x]\,C_2) \\ \text{(CA2)} & [!x]\,C \supset C \\ \text{(CA3)} & [!x]\,(!x = m \supset C) \equiv \langle !x \rangle\,(C \wedge !x = m) \\ \text{(CGen)} & \dfrac{C}{[!x]\,C} \end{array}$$

6 Logic (2): Axioms

6.1 Axioms for Content Quantification.

The purpose of this section is to introduce axioms for deriving valid assertions in our assertion language. We assume the standard notion of axiom system, where a deduction starts from axioms and reaches its root by repeated applications of inference rules. As is standard [31], we shall assume axioms and rules from propositional calculus, first-order logic with equality [49, Section 2.8], and formal number theory are freely available.

We start from axioms for content quantification. In axiomatisations of conventional first-order quantification [49, Section 2], one stipulates the single inference rule which infers $\forall x.A$ from A provided x does not appear freely in assumptions, and two axioms $(\forall x.(A \supset B) \supset A \supset \forall x.B$ provided x does not occur in A and $\forall x.A \supset A[e/x])$. This is all we need, assuming existential quantification is treated as a derived construct. In the following we present the analogous axiomatisation of content quantification. Our development closely follows [49, Section 2.3].

First, we regard $\langle !x \rangle\, C$ as standing for $\neg[!x]\,(\neg C)$. Then there are three axioms (CA1–CA3), listed in Figure 2. In (CA1), $C^{\text{-}!x}$ indicates C is *syntactically* $!x$*-free*. To define this notion and for other parts of the theory the following notion becomes important.

Definition 6.1 (active dereference) An occurrence of a dereference $!e$ in C is *active* if it occurs in C but it does not occur (1) in the scope of $\langle !e \rangle$ or $[!e]$, or (2) in the pre/post conditions of an evaluation formula.

As simple examples, T and F are syntactically $!x$-free. Similarly for $[!x]\,C$ and $\langle !x \rangle\, C$, as well as $!y = 3 \wedge x \neq y$. The assertion $!!y = 3 \ \wedge \ x \neq !y$ is not syntactically $!x$-free as it is, but the equivalent $\exists r.(!r = 3 \wedge r = !y \wedge r \neq !y)$ is. On the other hand, $!y = 3$ is not syntactically $!x$-free, even up to $\equiv$. Intuitively, $C^{\text{-}!x}$ says C does not mention the content of x.

Definition 6.2 (syntactic $!x$-freedom) We generate the set of syntactically $!x$-free formulae, $\mathcal{S}^{\text{-}!x}$, as follows: (1) $[!x]C \in \mathcal{S}^{\text{-}!x}$, dually $\langle !x \rangle\, C \in \mathcal{S}^{\text{-}!x}$. (2) $C \wedge \bigwedge_i e_i \neq x \in \mathcal{S}^{\text{-}!x}$ and, dually, $\bigwedge_i e_i \neq x \supset C \in \mathcal{S}^{\text{-}!x}$, in both cases assuming $\{!e_i\}$ exhaust all active dereferences in C; (3) The result of applying any of the logical connectives (including negation) or standard/content quantifiers, except $\forall x$ and $\exists x$, to formulae in $\mathcal{S}^{\text{-}!x}$ is again in $\mathcal{S}^{\text{-}!x}$.

Among the axioms, (CA1) corresponds to familiar $\forall x.(C_1 \supset C_2) \supset (C_1 \supset \forall x.C_2)$ with $x \notin \mathsf{fv}(C_1)$. (CA2) is a degenerate form of $\forall x.C \supset C[e/x]$. (CA3) says two ways to represent logical substitutions coincide, which is important to recover all properties of semantic update as studied in [10, 11, 56–58] as discussed in the next section.

Finally, to the rules of inference, we add the analogue of standard generalisation, given as (CGen) in Figure 2, which says: "If we can derive C from axioms, then conclude $[!x]C$". This rule assumes deductions without assumptions (e.g. all leaves of a proof tree should be axioms). If we *are* to use deduction with non-trivial assumptions, we demand assumptions are syntactically $!x$-free if the deduction uses (CGen) for $!x$. By the standard argument, we obtain the deduction theorem [49, Section 2.4].

Once a deduction theorem is given, we can use it to derive many laws for content quantification. For example, given the assumption $[!x](C_1 \wedge C_2)$ from which we can derive $C_1 \wedge C_2$ by (CA1) and modus ponens: then we obtain C_1 by the elimination rule for $\wedge$: to which we apply [*CGen*], which is possible because the assumptions are $!x$-free, to obtain $[!x]C_1$, similarly we get $[!x]C_2$, so we obtain $[!x]C_1 \wedge [!x]C_2$ by the $\wedge$-introduction rule; the other way round is similar. We also note (CA2) is not restrictive since from $[!x]C$ we can derive $C\{\!|m/!x|\!\}$ for arbitrary m. We now present several such laws in the following (these laws can be considered to constitute axioms in their own right, following those of standard modal operators [6]).

We begin by focussing on the universal part of the laws without loss of generality: later we summarise all laws including their existential counterparts. The first five rules are standard laws for the "necessity" modal operator.

$$[!x]C' \quad \supset \quad [!x]\,((C' \supset C) \supset C) \tag{6.1}$$

$$[!x]C \wedge [!x]C' \quad \equiv \quad [!x]\,(C \wedge C') \tag{6.2}$$

$$[!x]C \quad \supset \quad [!x]\,[!x]C \tag{6.3}$$

$$[!x]C \vee [!x]C' \quad \supset \quad [!x]\,(C \vee C') \tag{6.4}$$

$$[!x]\,(C \vee C') \quad \supset \quad [!x]C \vee \langle !x\rangle C' \tag{6.5}$$

$$[!x]C \quad \supset \quad C \tag{6.6}$$

The existential counterpart of these laws is by dualisation discussed below. (6.1) allows us to infer $[!x]C'$ from $[!x]C$ when $C \supset C'$ is a tautology. Note the existential counterpart of (6.3) is:

$$\langle !x\rangle\,\langle !x\rangle C \quad \supset \quad \langle !x\rangle C \tag{6.7}$$

which can again be strengthened to a logical equivalence.

The next four rules permute and increment quantifiers, again following treatment of the necessity modal operator. In the first rule, we assume x and y are distinct symbols.

$$\forall y.[!x]C \quad \supset \quad [!x]\forall y.C \tag{6.8}$$

$$[!y]\,[!x]C \quad \supset \quad [!x]\,[!y]C \tag{6.9}$$

$$\langle !x\rangle\,[!x]C \quad \supset \quad [!x]C \tag{6.10}$$

$$\forall x.!x \neq y \quad \supset \quad \mathsf{F} \tag{6.11}$$

Again they have dual versions. All these entailments are logical equivalences, with the reverse direction being derivable: for (6.8), if we have $[!x]\forall y.C$ and x and y are

distinct, then by y not free in the formula we have $\forall y.[!x]\forall y.C$, from which we conclude $\forall y.[!x]C$; (6.9) is already symmetric; finally the converse of (6.10) uses the notion of x-freedom of $[!x]C$ discussed later. One may observe the standard idempotence axiom for the universal modal operator is missing: we shall see that our corresponding axiom is derivable. The last rule (6.11) says it is not possible to constrain the content of x non-trivially if x can range over any name (whose existential dual, $\exists x.!x = e$, is often more useful). The axiom does not mention content quantification but its derivation needs it.

The next two laws allow us to eliminate and introduce universal content quantifications, and play the key role in reasoning about aliasing.

$$[!x]\,!x \neq y \quad \supset \quad \mathsf{F} \tag{6.12}$$

$$C^{\text{-}!x} \quad \supset \quad [!x]C \tag{6.13}$$

In both, the entailments are indeed logical equivalences. (6.12) is easily understood as an analogue of $\forall x.(x \neq y) \supset y \neq y$ ($\equiv \mathsf{F}$).

The following two laws connect universal content quantification and its dual.

$$\neg[!x]C \quad \equiv \quad \langle !x\rangle\,\neg C \tag{6.14}$$

$$[!x]\,(!x = m \supset C) \quad \equiv \quad \langle !x\rangle\,(C \wedge\, !x = m) \tag{6.15}$$

(6.14) directly comes from our definition of existential quantification in our axiom system. The second law (6.15) is (CA3), which relates two dual quantifiers *without dualisation*: its origin lies in the logical equivalence $\forall x.(x = m \supset C) \quad \equiv \quad C[m/x] \quad \equiv \exists x.(C \wedge x = m)$, briefly mentioned in the Introduction.

From (6.15) we immediately infer the equivalence between the two forms of logical substitutions introduced in Definition 5.4:

$$C\{\!|e'/!e|\!\} \equiv C\overline{\{\!|e'/!e|\!\}} \tag{6.16}$$

for any C, e' and $!e$. The axiom (6.15) plays a fundamental role in the present theory. From this we also infer:

$$[!x]C \quad \supset \quad C\{\!|e/x|\!\} \tag{6.17}$$

$$C\{\!|e/x|\!\} \quad \supset \quad \langle !x\rangle C \tag{6.18}$$

For the remaining laws for content quantifications, we introduce the semantic version of Definition 6.2.

Definition 6.3 C is $!x$*-free* when $[!x]C \equiv C$.

Remark 6.4 We usually regard $\equiv$ in Definition 6.3 as a syntactic notion (i.e. derivability of $[!x]C \equiv C$ as a theorem in the present logic, involving the axioms in the present section as well as the ambient logical system such as Peano Arithmetic).

By Axiom 6.13, any syntactically $!x$-free assertion is $!x$-free but reverse implication does not hold, for example $!x = !x$ is semantically but not syntactically $!x$-free. Some examples of $!x$-free formulae follow.

Example 6.5 (!x-freedom)

1. As noted, any syntactic !x-free formula is !x-free. In particular T and F are !x-free.
2. Similarly $[!x]C$ and $\langle !x\rangle C$ are immediately !x-free.
3. Since !x-freedom is closed under $\equiv$ by definition, any tautologies/unsatisfiable formulae are !x-free. Also C is !x-free iff $C \equiv C_0$ such that C_0 is syntactically !x-free.
4. Assume $C \stackrel{\text{def}}{=} !!e = 3 \wedge !e \neq x$ (so x is of type $\mathsf{Ref}(\mathsf{Nat})$). Then C is !x-free. Indeed, we can write $C \equiv \exists r.(!e = r \wedge !r = 3 \wedge r \neq x)$.
5. (α-stateless formulae) Let us say a formula C is α-*stateless* (resp. *stateless*) if C has no active dereferences of type α (resp. of any type). Then C being α-stateless and x being typed by $\mathsf{Ref}(\alpha)$ in C imply C is !x-free.

Since $[!x]C \supset C$ for any C by (6.6), we know C is !x-free if and only if $C \supset [!x]C$. Note $\langle !x\rangle C \equiv C$ also characterises !x-freedom (which is often useful in practice) and that the converse of (6.10) does hold.

The following results strengthen our observation that "!x-freedom of C" acts as a substitute for "x not occurring in C" in standard quantification theory.

Proposition 6.6 *If C_1 is !x-free, then:*

$$[!x](C_1 \vee C_2) \equiv C_1 \vee [!x]C_2 \tag{6.19}$$
$$\langle !x\rangle (C_1 \wedge C_2) \equiv C_1 \wedge \langle !x\rangle C_2 \tag{6.20}$$
$$[!x](C_1 \supset C_2) \equiv C_1 \supset [!x]C_2. \tag{6.21}$$

Proof By duality and since (6.21) merely rephrases (6.19), it suffices to derive (6.19).

$$\begin{array}{rcl} [!x](C_1^{-!x} \vee C_2) & \supset & \langle !x\rangle C_1^{-!x} \vee [!x]C_2 \\ & \equiv & C_1^{-!x} \vee [!x]C_2 \\ C_1^{-!x} \vee [!x]C_2 & \equiv & [!x]C_1^{-!x} \vee [!x]C_2 \\ & \supset & [!x](C_1^{-!x} \vee C_2) \end{array}$$

Both universal and existential characterisations of !x-freedom are needed to obtain the desired logical equivalence. □

Note (6.21) is the same thing as saying $[!x](C_1 \supset C_2) \supset C_1 \supset [!x]C_2$ whenever C_1 is !x-free, the analogue of the standard axiom for universal quantifications.

Proposition 6.7 (derived axioms)

1. $[!x](C \wedge (C \supset C')) \supset [!x]C'$, *dually* $\langle !x\rangle C \supset \langle !x\rangle ((C \supset C') \supset C')$.
2. *If* $C \supset C'$ *is a tautology then* $[!x]C \supset [!x]C'$.
3. $[!x]C \supset C\{\!|e/!x|\!\}$, *dually* $C\{\!|e/!x|\!\} \supset \langle !x\rangle C$. *Further* $C\{\!|!x/!x|\!\} \equiv C$.
4. C *is !x-free iff* $C \equiv \langle !x\rangle C$ *iff* $\exists C'.(C \equiv \langle !x\rangle C')$ *iff* $[!x]C \equiv C$ *iff* $\exists C'.(C \equiv [!x]C')$.
5. *If* $C_{1,2}$ *are !x-free, then* $C_1 \star C_2$ *(*$\star \in \{\wedge, \vee, \supset\}$*) is !$x$-free. If C is !x-free, then* $\neg C$ *is !x-free. If C is !x-free and $x \neq y$, then* $\forall y.C$ *and* $\exists y.C$ *are both !x-free. If C is !x-free, then* $[!y]C$ *and* $\langle !y\rangle C$ *are both !x-free.*
6. *If* e^{α} *is free for !x in C and both* $C[e/!x]$ *and e are α-stateless,* $C[e/!x] \equiv C\{\!|e/!x|\!\}$ *(where e is free for !x is defined as in Section 5.2 and* $C\{\!|e/!x|\!\}$ *is the result of substituting e for each active occurrence of !x).*

Proof For (1):

$$\begin{array}{lcl}
[!x]\,(C \wedge (C \supset C')) & \equiv & [!x]C \ \wedge \ [!x]\,(\neg C \vee C') \\
& \supset & [!x]C \ \wedge \ (\langle !x\rangle\, \neg C \vee [!x]C') \\
& \equiv & ([!x]C \wedge \langle !x\rangle\, \neg C) \vee ([!x]C \wedge [!x]C') \\
& \equiv & \mathsf{F} \vee ([!x]C \wedge [!x]C') \\
& \supset & [!x]C'
\end{array}$$

For (2), observing any tautology is $!x$-free:

$$\begin{array}{lcl}
[!x]C & \equiv & [!x]C \ \wedge \ (C \supset C') \\
& \equiv & [!x]C \ \wedge \ [!x]\,(C \supset C') \\
& \equiv & [!x]\,(C \ \wedge \ (C \supset C')) \\
& \supset & [!x]C'
\end{array}$$

For (3), the first statement:

$$\begin{array}{lcl}
[!x]C & \equiv & [!x]C \wedge \langle !x\rangle\, !x = m \\
& \supset & \langle !x\rangle\, (C \wedge !x = m) \\
& \equiv & \forall m.\langle !x\rangle\, (C \wedge !x = m) \wedge \exists m.m = e \\
& \supset & \exists m.(\langle !x\rangle\, (C \wedge !x = m) \wedge m = e) \\
& \equiv & C\{\!|\, !e/!x \,|\!\}
\end{array}$$

The second statement is the dual of the first statement. For one direction of the third statement, with m fresh:

$$\begin{array}{lcl}
C & \equiv & \exists m.(C \ \wedge \ !x = m \ \wedge \ !x = m) \\
& \supset & \exists m.(\langle !x\rangle C \ \wedge \ !x = m \ \wedge \ !x = m) \\
& \stackrel{\text{def}}{\equiv} & C\{\!|\, !x/!x \,|\!\}.
\end{array}$$

For the other direction, again with m fresh:

$$\begin{array}{lcl}
C\{\!|\, !x/!x \,|\!\} & \equiv & C\overline{\{\!|\, !x/!x \,|\!\}} \\
& \stackrel{\text{def}}{\equiv} & \forall m.(m = !x \supset [!x]\, !x = m \supset C) \\
& \supset & \forall m.(m = !x \supset !x = m \supset C) \\
& \supset & C
\end{array}$$

(4) and (5) are easy and omitted. For (6):

$$\begin{array}{lcl}
C\{\!|\, e/!x \,|\!\} & \stackrel{\text{def}}{=} & \exists m.(\langle !x\rangle\, (C \wedge !x = m) \wedge m = e) \\
& \equiv & \langle !x\rangle\, (C \wedge !x = e) \\
& \equiv & \langle !x\rangle\, (C[e/!x] \wedge !x = e) \\
& \equiv & C[e/!x] \ \wedge \ \langle !x\rangle\, !x = e \\
& \equiv & C[e/!x] \qquad\qquad \square
\end{array}$$

Finally, as a simple application of the content quantification, we calculate an example we treated in Introduction.

$$\begin{array}{lcll}
C\{\!|\, \mathsf{c}/!x \,|\!\}\{\!|\, e/!x \,|\!\} & \equiv & \exists m.(\langle !x\rangle\, (\langle !x\rangle\, (C \ \wedge \ !x = \mathsf{c}) \ \wedge \ !x = m) \ \wedge \ m = e) & \\
& \equiv & \exists m.(\langle !x\rangle\, (C \ \wedge \ !x = \mathsf{c}) \ \wedge \ (\langle !x\rangle\, !x = m) \ \wedge \ m = e) & (*) \\
& \equiv & \langle !x\rangle\, (C \ \wedge \ !x = \mathsf{c}) \quad \equiv \quad C\{\!|\, \mathsf{c}/!x \,|\!\} & \text{(e9)}
\end{array}$$

where $(*)$ uses $\langle !x\rangle\, (\langle !x\rangle C \wedge C') \equiv \langle !x\rangle C \wedge \langle !x\rangle C'$, which is direct from Proposition 6.6.

Fig. 3 Axioms for evaluation formulae.

$$
\begin{array}{lrcl}
\text{(e1)} & \{C_1\}x\bullet y\searrow z\{C\} \ \wedge\ \{C_2\}x\bullet y\searrow z\{C\} & \equiv & \{C_1\vee C_2\}\,x\bullet y\searrow z\,\{C\} \\
\text{(e2)} & \{C\}x\bullet y\searrow z\{C_1\} \ \wedge\ \{C\}x\bullet y\searrow z\{C_2\} & \equiv & \{C\}\,x\bullet y\searrow z\,\{C_1\wedge C_2\} \\
\text{(e3)} & \{\exists w^{\alpha}.C\}\,x\bullet y\searrow z\,\{C'^{-w}\} & \equiv & \forall w^{\alpha}.\{C\}\,x\bullet y\searrow z\,\{C'\} \\
\text{(e4)} & \{C^{-w}\}\,x\bullet y\searrow z\,\{\forall w^{\alpha}.C'\} & \equiv & \forall w^{\alpha}.\{C\}\,x\bullet y\searrow z\,\{C'\} \\
\text{(e5)} & \{A\wedge C\}\,x\bullet y\searrow z\,\{C'\} & \equiv & A\supset\{C\}\,x\bullet y\searrow z\,\{C'\} \\
\text{(e6)} & \{C\}\,x\bullet y\searrow z\,\{A^{-z}\supset C'\} & \supset & A\supset\{C\}\,x\bullet y\searrow z\,\{C'\} \\
\text{(e7)} & \{C\}x\bullet y\searrow z\{C'\} & \supset & \{C\wedge A\}x\bullet y\searrow z\{C'\wedge A\} \\
\text{(e8)} & [!\tilde{w}]\,(C\supset C_0)\ \wedge\ \{C_0\}x\bullet y\searrow z\{C_0'\}\ \wedge\ [!\tilde{w}]\,(C_0'\supset C) & \supset & \{C\}\,x\bullet y\searrow z\,\{C'\} \\
\text{(e9)} & \{C\}x\bullet y\searrow z\{C'\} & \supset & [!u]\,\{C\}x\bullet y\searrow z\{C'\} \\
\text{(e0)} & \langle !x\rangle\,\{C\}y\bullet z\searrow w\{C'\} & \supset & \{C\}y\bullet z\searrow w\{C'\} \\
\text{(ext)} & \mathsf{Ext}^{\Delta;\alpha\Rightarrow\beta}(x,y) & \supset & x=y
\end{array}
$$

6.2 Axioms for Evaluation Formulae

The set of axioms for evaluation formulae are given in Figure 3. Most come from the axioms in [37]. We assume the following convention used throughout the paper.

Convention 6.8 From now on $A,A',B,B',\ldots$ (possibly subscripts) range over *stateless formulae*, i.e. those formulae without any active dereferences (cf. Example 6.5 (3)), while $C,C',\ldots$ still range over general formulae.

In (e8), we use content quantifications to stipulate hypothetical entailment, cf. (5.16), Section 5.5 (which is closely related with Kleymann's strengthened consequence rule [41]). In the rule, we assume $\tilde{w}$ exhaust all active dereferences in C,C_0,C_0' and C'. (e2) and (e8) gives the following axiom which is often useful:

$$\{C_1\}x\bullet y\searrow z\{C_1'\}\ \wedge\ \{C_2\}x\bullet y\searrow z\{C_2'\}\ \supset\ \{C_1\wedge C_2\}\,x\bullet y\searrow z\,\{C_1'\wedge C_2'\} \quad (6.22)$$

The dual axiom (for disjunction) is similarly obtained from (e1) and (e8).

The axioms (e9) and (e0) say the content quantification has no affect on pre/post conditions of evaluation formulae (since they are already content-quantified semantically, cf. Section 5.4).

In (ext), the extensionality formula augments the corresponding formulae for alias-free sublanguage in [37] with located assertions.

Definition 6.9 (extensionality formulae) Let $\Delta=\tilde{r}:\mathsf{Ref}(\tilde{\gamma})$ and x and y be typed as $\alpha\Rightarrow\beta$. Then set set:

$$
\begin{array}{rcl}
\mathsf{Ext}^{\Delta;\alpha\Rightarrow\beta}(x,y) & \stackrel{\text{def}}{=} & \forall h^{\alpha},i^{\beta},\ \tilde{j}^{\tilde{\gamma}},\tilde{j}'^{\tilde{\gamma}}.\ (\ \{!\tilde{r}=\tilde{j}\}\,x\bullet h\searrow z\,\{z=i\wedge !\tilde{r}=\tilde{j}'\}\,@\,\tilde{r} \\
 & & \qquad\equiv\ \{!\tilde{r}=\tilde{j}\}\,y\bullet h\searrow w\{w=i\wedge !\tilde{r}=\tilde{j}'\}\,@\,\tilde{r}\ )
\end{array}
$$

We call $\mathsf{Ext}^{\Delta;\alpha\Rightarrow\beta}(x,y)$, *the extensionality formula for x and y of type* $\alpha\Rightarrow\beta$ *under* Δ or, more briefly, *the extensionality formula for x and y*.

Fig. 4 Axioms for located evaluation formulae.

$$
\begin{array}{lrcl}
\text{(le1)} & \{C_1\}x\bullet y\searrow z\{C\}@\tilde{w} \wedge \{C_2\}x\bullet y\searrow z\{C\}@\tilde{w} & \equiv & \{C_1\vee C_2\}x\bullet y\searrow z\,\{C\}@\tilde{w} \\
\text{(le2)} & \{C\}x\bullet y\searrow z\{C_1\}@\tilde{w}\wedge\{C\}x\bullet y\searrow z\{C_2\}@\tilde{w} & \equiv & \{C\}x\bullet y\searrow z\{C_1\wedge C_2\}@\tilde{w} \\
\text{(le3)} & \{\exists u^{\alpha}.C\}x\bullet y\searrow z\,\{C'^{\neg u}\}@\tilde{w} & \equiv & \forall u^{\alpha}.\{C\}x\bullet y\searrow z\,\{C'\}@\tilde{w} \\
\text{(le4)} & \{C^{\neg u}\}x\bullet y\searrow z\,\{\forall u^{\alpha}.C'\}@\tilde{w} & \equiv & \forall u^{\alpha}.\{C\}x\bullet y\searrow z\,\{C'\}@\tilde{w} \\
\text{(le5)} & \{A\wedge C\}x\bullet y\searrow z\,\{C'\}@\tilde{w} & \equiv & A\supset\{C\}x\bullet y\searrow z\,\{C'\}@\tilde{w} \\
\text{(le6)} & \{C\}x\bullet y\searrow z\,\{A^{\neg z}\supset C'\}@\tilde{w} & \supset & A\supset\{C\}x\bullet y\searrow z\,\{C'\}@\tilde{w} \\
\text{(le7)} & \{C\}x\bullet y\searrow z\{C'\}@\tilde{w} & \supset & \{C\wedge[!\tilde{w}]C_0\}x\bullet y\searrow z\{C'\wedge[!\tilde{w}]C_0\}@\tilde{w} \\
\text{(le8)} & [!\tilde{w}]\,(C\supset C_0)\wedge\{C_0\}x\bullet y\searrow z\{C_0'\}@\tilde{w}\wedge[!\tilde{w}]\,(C_0'\supset C) & \supset & \{C\}x\bullet y\searrow z\,\{C'\}@\tilde{w} \\
\text{(le9)} & \{C\}x\bullet y\searrow z\{C'\}@\tilde{w} & \supset & [!u]\,\{C\}x\bullet y\searrow z\{C'\}@\tilde{w} \\
\text{(le0)} & \langle!x\rangle\,\{C\}y\bullet z\searrow w\{C'\}@\tilde{w} & \supset & \{C\}y\bullet z\searrow w\{C'\}@\tilde{w} \\
\text{(weak)} & \{C\}x\bullet y\searrow z\{C'\}@\tilde{v} & \supset & \{C\}x\bullet y\searrow z\{C'\}@\tilde{v}\tilde{w} \\
\text{(thin)} & \forall u,i.\{C\wedge!u=i\}x\bullet y\searrow z\{C'\wedge!u=i\}@\tilde{w} & \supset & \{C\}x\bullet y\searrow z\{C'\}@\tilde{w}\backslash u
\end{array}
$$

The extensionality formula expresses an extensional equality of two imperative sequential higher-order behaviours. The predicate says, letting $\mathsf{dom}(\Delta)=\tilde{r}$:

> *Whenever x converges for some argument and for some stored values at $\tilde{r}$ and returns some value, then y does the same with the same return value. In addition no other reference cells are altered by x or y.*

Note the use of write effects is fundamental to describe extensionality since, if not, they may have different effects on unspecified memory cells.

We have already seen that located assertions play an essential role in the extensionality axiom. In Figure 4, we list axioms for located assertions, which refine the original axioms in Figure 3 (except (ext) which is already about located assertions), as well as adding two new axioms for manipulating write effects. The axioms from (le1) to (le6) simply add write effects to assertions. However (le7) allows us to add universally content-quantified stateful formulae to the pre/post conditions, strengthening (e7). The reader may recall having already seen an instance of this rule in (5.14) and (5.15), Section 5.5, Page 24. (le7) is more general than (e7) in the sense that weakened assertion can be stateful. At the same time, it is also true that (le7) is justifiable using (e7). For concreteness, take the assertion (5.14) in Section 5.5:

$$\{!x=i\}\,u\bullet x\,\{!x=2\times i\}\,@\,x$$

To this assertion we apply the standard universality law to obtain for a concrete y:

$$\forall j.\{!x=i\ \wedge\ x\neq y\wedge\ !y=j\}\,u\bullet x\,\{!x=2\times i\ \wedge\ x\neq y ANDl\, !y=j\}\,@\,x.$$

Now we use (e7) and get:

$$\forall j.\{!x=i\ \wedge\ x\neq y\wedge\ !y=j\ \wedge\ Even(j)\}\,u\bullet x\,\{!x=2\times i\wedge\ !y=j\ \wedge\ Even(j)\}\,@\,x$$

Fig. 5 Axioms for data types.

(t1)	T	$\supset$	$x^{\mathsf{Unit}} = ()$
(t2)	T	$\supset$	$x^{\mathsf{Bool}} = \mathsf{t} \vee x^{\mathsf{Bool}} = \mathsf{f}$
(t3)	T	$\supset$	$\mathsf{t} \neq \mathsf{f}$
(t4)	T	$\equiv$	$\pi_i(x_1, x_2) = x_i$
(t5)	T	$\equiv$	$(\pi_1(x), \pi_2(x)) = x$
(t6)	$\mathtt{in}_i(x) = \mathtt{in}_i(y)$	$\supset$	$x = y$
(t7)	T	$\supset$	$\mathtt{in}_1(x) \neq \mathtt{in}_2(y)$

By the law of equality and (e3) we infer:

$$\{\exists j.(!x = i \wedge x \neq y \wedge Even(!y) \wedge !y = j)\}\, u \bullet x\, \{!x = 2 \times i \wedge x \neq y \wedge Even(!y)\}\, @\, x$$

Hence by (e8) we obtain:

$$\{!x = i \wedge Even(!y)\}\, u \bullet x\, \{!x = 2 \times i \wedge Even(!y)\}\, @\, x,$$

as required. As in this example, all these rules are easily justifiable using the axioms rules in Figure 3.

Finally (weak) and (thin) correspond to the first two implications in Proposition 5.8. They are reminiscent of the weakening rules and thinning rules in various type disciplines, hence the names.

6.3 Axioms for Data Types

Finally we introduce axioms for data types, listed in Figure 5. All axioms have already appeared in [37]: the only difference is that we no longer have the axiom saying two distinct reference names are always distinct, because that no longer holds. One of the central features of the present logic is its general treatment of data types. Examples for stateless data types are already illustrated in our previous work, cf. [36]. Here we allow reference types to appear anywhere in types, so that data structures can now be destructively updated in their parts. In the next section we shall see a generalised assignment axiom which can treat assignment of an arbitrary data structure to an arbitrary (mutable part of) data structure, which is quite common in systems programming (e.g. a part of a record referred to by another record is replaced with another pointer).

The data types treated above come from imperative PCFv. In practice, we may incorporate other standard data types, such as unions, vectors and arrays. Below we consider how arrays can be treated. At the level of the programming language we add:

(types)	α	$::=$	...	$\mid$	$\alpha[]$
(programs)	M	$::=$	...	$\mid$	$M[N]$

together with the typing rules:

$$\frac{-}{\Gamma \vdash a : \alpha[]} \qquad \frac{\Gamma \vdash M : \alpha[] \quad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash M[N] : \mathsf{Ref}(\alpha)}$$

The construction above assumes that the identifier of each array to be used is given as a constant (ranged over by $a, b, \ldots$). We further regard the expressions of the form $a[0], a[1], \ldots, a[n-1]$ for some n as values of reference types. These values form part of the domain of a concrete store: it is also convenient, though not necessary, to include them as part of a reference basis so that the size of an array is determined from a basis. For statically sized arrays, this offers clean typing, though there are other approaches. Regarding dynamics, out-of-bound read of an array may as well become nil, whose dereference leads to an error err. For defining reduction, we may as well include typing for these constructs (there are several approaches to the treatment of the types/semantics of an array-bound error: here we consider an out-of-bound access generates nil of the corresponding reference type; the dereference of nil leads to err; and that err, when evaluated, leads to err of the whole expression, which follows the standard treatment of type error [51]).

Terms are augmented accordingly:

$$e \quad ::= \quad \ldots \quad | \quad a \quad | \quad e[e'] \quad | \quad \mathsf{size}(e) \quad | \quad \mathsf{nil}^{\mathsf{Ref}(\alpha)} \quad | \quad \mathsf{err}^{\alpha}$$

where, in $e[e']$, we type e with an array type (say $\alpha[]$) and e' with Nat, with the whole term given the type in $\mathsf{size}(e)$ (which denotes the size of an array e), we type e with an array type, with the whole term typed with Nat; $\mathsf{nil}^{\mathsf{Ref}(\alpha)}$, which denotes the null pointer and whose type we usually omit, is typed by $\mathit{ref}(\alpha)$; and err^{α} denotes a (dereference) error of type α, for each α.

We list some of the main axioms for arrays. First, for each constant a of type $\alpha[]$, we stipulate its size:

$$\mathsf{size}(a) = n$$

for a specific $n \in \mathsf{Nat}$ (which should conform to the reference basis if stipulated). Next we have the following axiom for all arrays to ensure that an array of size n is made up of n distinct references.

$$\forall i, j.\ (\ 0 \leq i, j \lneq \mathsf{size}(x) \ \wedge\ i \neq j \quad \supset \quad x[i] \neq x[j]\ ) \tag{6.23}$$

Another basic axiom for arrays is for their equality (for two arrays of the same type):

$$(\mathsf{size}(x) = \mathsf{size}(y)\ \wedge\ \forall i.\ (\ 0 \leq i < \mathsf{size}(x) - 1\ \supset\ x[i] = y[i]\ ) \quad \supset \quad x = y \tag{6.24}$$

In some languages (such as Pascal), we may also stipulate the inequality axiom:

$$x \neq y \quad \supset \quad \forall i, j.\ (\ 0 \leq i < \mathsf{size}(x) - 1\ \wedge\ 0 \leq j < \mathsf{size}(y) - 1\ \supset\ x[i] \neq y[j]\ ) \tag{6.25}$$

which says two distinct arrays never overlap (note this axiom is not applicable to, for example, C). Note (6.25) is equivalent to:

$$\exists i, j.\ (\ 0 \leq i < \mathsf{size}(x) - 1\ \wedge\ 0 \leq j < \mathsf{size}(y) - 1\ \wedge\ x[i] = y[j]\ ) \quad \supset \quad x = y. \tag{6.26}$$

For those axioms which involve nil and err, see Remark below.

In models, we may treat an array as simply a function from natural numbers to references such that it maps all numbers within its range to distinct references and

others to nil, cf.[4]. Other constraints can be considered following the axioms as given above.

As we shall see later, the addition of arrays to data types precisely adds to the compositional proof system one introduction rule (as a constant) and one elimination rule (for indexing). This modularity is one of the key features of the present logic.

Remark 6.10 (axioms for nil and err) For reference we list basic axioms involving nil and err. While these constructs are introduced for a wholesome semantic treatment of assertions, the need to use them may not be as frequent as other "normal" term constructors (however their treatment becomes essential when we consider e.g. error recovery routines). First of all, the out of bound is treated as:

$$i \gtrsim \mathsf{size}(x) \quad \supset \quad x[i] = \mathsf{nil} \tag{6.27}$$

Further we stipulate:

$$!\mathsf{nil} \;=\; \mathsf{err}. \tag{6.28}$$

Further we stipulate err when used as part of an expression always leads to err:

$$\mathcal{E}(\mathsf{err}) \;=\; \mathsf{err} \tag{6.29}$$

where $\mathcal{E}[\cdot]$ is an arbitrary term context. We observe that there can be other choices for the behaviour of these exceptional terms, whose investigation is deferred to a future occasion.

7 Logic (3): Judgements and Proof Rules

7.1 Judgements and their Semantics

Following Hoare [31], a judgement in the present program logic consists of a pair of formulae and a program, augmented with a fresh name called *anchor*, which takes the following shape.

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

This 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 $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, $M^{\Gamma;\Delta;\alpha}$ is the *subject* of the judgement; u its *anchor*, which should not be in $\mathsf{dom}(\Gamma,\Delta) \cup \mathsf{fv}(C)$; C its *pre-condition*; and C' its *post-condition*.[4] We say $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is *well-typed* iff

- $\Gamma;\Delta \vdash M : \alpha$.
- $\Gamma';\Delta' \vdash C$.
- For some $\Gamma' \supset \Gamma$ and $\Delta' \supset \Delta$ such that $\mathsf{dom}(\Gamma' \cup \Delta') \cap \{u\} = \emptyset$ we have
  - $\Gamma' \cdot u : \alpha; \Delta' \vdash C'$, if α is not a reference,
  - $\Gamma'; \Delta' \cdot u : \alpha \vdash C'$, if α is a reference.

Henceforth we only treat judgements which are well-typed. Following Convention 5.1 (5), the notation $\{C\}\,M\,\{C'\}$ stands for $\{C\}\,M :_u \{u = () \wedge C'\}$ where u is a fresh name, typed with Unit.

As in Hoare logic, the distinction between primary names and auxiliary names plays an important role in both proof rules and semantics of the logic.

Definition 7.1 (primary/auxiliary names) Let $\models\{C\}\,M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ be well-typed. Then the *primary names* in this judgement are $\mathsf{dom}(\Gamma,\Delta) \cup \{u\}$. The *auxiliary names* in the judgement are those free names in C and C' that are not primary.

Example 7.2 In a judgement "$\{x = i\}2 \times x^{x:\mathsf{Nat};\mathsf{Nat}} :_u \{u = 2 \times i\}$", x and u are primary while i is auxiliary. u is in addition its anchor.

Intuitively, $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ says:

> *if* $\Gamma;\Delta \vdash M : \alpha$ *is closed by values satisfying C (for* $\mathsf{dom}(\Gamma)$*) and runs starting from a store satisfying C (for* $\mathsf{dom}(\Delta)$ *and maybe more), then it terminates so that the final state and the resulting value named u together satisfy* C'.

A store considered for a model may have a domain greater than Δ. First this is a sheer necessity because, for example, a store for $x : \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat})) \vdash !!x : \mathsf{Nat}$ should have not only x but another reference which stores x (the same is true for auxiliary names). Second this is consistent with $\cong_{\mathcal{D}}$ (as $\cong$) being considered under all extensions of a given basis, cf. Section 2.3/Section 3.1. Formally we stipulate as follows (see Notation 3.11, Page 13, for the notation $(\xi \cdot u : \mathbf{v},\ \sigma')$).

[4] In spite of the designations "pre/post-conditions", these assertions also describe complex (stateless) properties about higher-order behaviour and data structures.

Definition 7.3 (semantics of judgement) We say the judgement $\models\{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is *valid*, written $\models\{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, iff the following condition holds: for each model $\mathcal{M}^{\Gamma';\Delta'} \stackrel{\text{def}}{=} (\xi,\sigma)$ where $\Gamma' \supset \Gamma$, $\Delta' \supset \Delta$, $\Gamma';\Delta' \vdash C$ and $\Gamma'\cdot u:\alpha;\Delta' \vdash C'$, if $(\xi,\sigma) \models C$ then $(M\xi,\ \sigma) \Downarrow (\mathbf{v},\sigma')$ such that $(\xi\cdot u:\mathbf{v},\ \sigma') \models C'$.

Note the standard practice of considering all possible models for validity means, in the present context, considering all possible aliasing conforming to the precondition C.

7.2 Proof Rules (1): Compositional Rules

We now present the proof rules for deriving valid judgements for the imperative PCFv with aliasing. There is one compositional proof rule for each construct of the underlying programming language, which precisely follows its syntactic structure. In addition, there are structural rules which only manipulate formulae. We can also consider additional inference rules which are useful for economical reasoning and which are justifiable (admissible) in the present system. We shall discuss later in some detail inference rules of this third kind, specialised into located assertions and their counterpart in judgements.

This subsection introduces the compositional proof rules. Their shape is unchanged from the proof rules for the sublanguage without aliasing [37] except for a minimal and unavoidable refinement of the rule for assignment, that is, the use of logical content substitution instead of syntactic substitution (cf. Section 4) (in addition to adaptation to generalised syntax in dereference and assignment). This is in accordance with our logical language, which increments that in [37] by two dual modal operators for reasoning about aliasing. More fundamentally, the refinement in the assertion language and the proof rules reflects that of the type structure of the programming language, i.e. the extension to allow reference types to be carried by other types. This incremental nature, especially precise correspondence between the type structure and the logical apparatus, is central to the family of program logics under investigation by the present authors.

Following [37], we stipulate the following conventions for proof rules.

Convention 7.4 (proof rules)

- Variables $i, j, \ldots$ that occur freely in a formula range over auxiliary names in a given judgement.
- $C^{\text{-}\tilde{x}}$ is C in which no name from $\tilde{x}$ freely occurs (note this is very different from $C^{\text{-}!\tilde{x}}$).
- In each proof rule, we assume all occurring judgements to be well-typed and no primary names in the premise(s) occur as auxiliary names in the conclusion. This may be considered as a variant of the standard bound name convention.
- Whenever a syntactic substitution is used in a proof rule, it should avoid capture of names, i.e. it should be safe in the sense detailed in Section 5.2.
- Following Convention 6.8, $A, A', B, B', \ldots$ range over *stateless formulae*, i.e. those formulae which do not contain active dereferences (active dereferences are those dereferences which do not occur in pre/post conditions of evaluation formulae, cf. Definition 6.1).

The compositional proof rules of the program logic are given in Figure 6 (the difference from [37] is highlighted). [*Op*] is a general rule for first-order operators, and subsumes [*Const*] when the arity is zero. As noted already, the shape of all the rules in Figure 6 are identical character-by-character with the compositional rules for the imperative PCFv without aliasing except for 9[*Assign*] which uses logical substitution and hence content quantification. Leaving detailed illustration of the remaining rules to [37], we illustrate two rules for imperative constructs, [*Deref*] and [*Assign*] in the following.

Fig. 6 Proof rules (1): compositional rules.

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

$$[\mathit{Op}]\ \frac{C_0 \stackrel{\mathrm{def}}{=} C \quad \{C_i\}\, M_i :_{m_i} \{C_{i+1}\}\ (0 \le i \le n-1) \quad C_n \stackrel{\mathrm{def}}{=} C'[\mathtt{op}(m_0..m_{n-1})/u]}{\{C\}\ \mathtt{op}(M_0..M_{n-1}) :_u \{C'\}}$$

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

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

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

$$[\mathit{In}_1]\ \frac{\{C\}\ M :_v \{C'[\mathsf{inj}_1(v)/u]\}}{\{C\}\ \mathtt{in}_1(M) :_u \{C'\}} \qquad [\mathit{Case}]\ \frac{\{C^{-\tilde{x}}\}\ M :_m \{C_0^{-\tilde{x}}\} \quad \{C_0[\mathsf{inj}_i(x_i)/m]\}\ M_i :_u \{C'^{-\tilde{x}}\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

$$[\mathit{Pair}]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\} \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1, m_2\rangle/u]\}}{\{C\}\ \langle M_1, M_2\rangle :_u \{C'\}} \qquad [\mathit{Proj}_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

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

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

[***Deref***]**.** The rule [*Deref*] says that:

> *If we wish to have C for !M named u, then we should assume the same thing about M, its content, substituting !x for u in C.*

To understand this rule, we may start from the following simpler version (which appeared in [37]).

$$[\textit{Deref-Org}]\ \frac{-}{\{C[!x/u]\}\ !x :_u \{C\}} \tag{7.1}$$

The rule says that, if we wish to have C for $!x$ (as a program) named u, then we should assume the same thing about the content of x, substituting $!x$ for u in C. For example we may infer:

$$\frac{-}{\{Even(!x)\}\ !x :_u \ \{Even(u)\}} \tag{7.2}$$

which is also sound in the present target language and logic. [*Deref*] generalises [*Deref-Org*] so that it can treat the case when the dereference is done for an arbitrary program of a reference type, which can even include invocation of imperative procedures. This becomes possible by the change of type structure, where references can be used as return values or as components of data types. A simple example follows (below and henceforth we often do not expand simple applications of [*Consequence*]).

1. $\{\mathsf{T}\}\ x :_z \{z = x\}$	(Var)
2. $\{\mathsf{T}\}\ \lambda x.x :_m \{\forall x.\{\mathsf{T}\} m \bullet x \searrow z\{z = x\}\}$	(Abs)
3. $\{\forall x.\{\mathsf{T}\} m \bullet x \searrow z\{z = x\}\}\ y :_n \{n = y \ \wedge\ \{\mathsf{T}\} m \bullet n \searrow z\{z = y\}\}$	(Var, Consequence)
4. $\{\mathsf{T}\}\ (\lambda x.x)y :_m \{!m = !y\}$	(App, Consequence)
5. $\{\mathsf{T}\}\ !((\lambda x.x)y) :_u \{u = !y\}$	(Deref)

As another simple example, let C be given by:

$$C \stackrel{\text{def}}{=} \forall x, i.\{!x = i\}\, f \bullet x \searrow z\{z = x \wedge\ !x = i + 1\},$$

Then we infer:

$$\{C \wedge !x = 1\}\ !(fx) :_u \{u = x \ \wedge\ \ !x = 2\} \tag{7.3}$$

by the following derivation.

1. $\{C \wedge !x = 1\}\ f :_m \{C[m/f] \wedge\ !x = 1\}$	(Var)
2. $\{C[m/f] \wedge !x = 1\}\ x :_n \{C[m/f]\ \wedge\ n = x \wedge\ !x = 1\}$	(Var)
3. $\{C[m/f] \wedge !x = 1\}\ x :_n \{!x = 1 \wedge \{!x = 1\} m \bullet n \searrow z\{z = x \wedge !x = 2\}\}$	(2, Conseq)
4. $\{C \wedge !x = 1\}\ fx :_l \{l = x \ \wedge\ !x = 2\}$	(Var)
5. $\{C \wedge !x = 1\}\ fx :_l \{!l = 2 \ \wedge\ !x = 2\}$	(3, Conseq)
6. $\{C \wedge !x = 1\}\ !(fx) :_u \{u = 2 \ \wedge\ !x = 2\}$	(Deref)

Note the application above not only returns a reference but also has a side effect. In this way we can use [*Deref*] for dereferences of arbitrary programs. It is worth observing that [*Deref-Org*] is more efficient when a single variable is dereferenced, which may be frequent in practice. We could have used a similarly specialised rule for an application of variables instead of the first five lines.

Soundness of [*Deref*]. The shape of [*Deref*] and other proof rules has a direct semantic justification: it is born from the semantics. The following semantic justification of the rule makes this clear (below we write $(M\xi,\sigma) \Downarrow_m (\xi{\cdot}m : \mathbf{v},\sigma')$ for $(M\xi,\sigma) \Downarrow (\mathbf{v},\sigma')$).

$$\begin{array}{lll} (\xi,\, \sigma) \models C & \Rightarrow & (M\xi,\, \sigma)\ \Downarrow_m\ (\xi{\cdot}m{:}\mathbf{i},\, \sigma')\ \models C'[!m/u] \\ & \Rightarrow & ((!M)\xi,\, \sigma)\ \Downarrow_u\ (\xi{\cdot}u{:}\sigma'(\mathbf{i}),\, \sigma')\ \models C' \end{array}$$

where we confuse (substitutions for) abstract values and (substitutions for) concrete values. The second inference above is valid because dereferencing does not change the store, noting the freshness of m.

[*Assign*]. The rule [*Assign*] says that:

> *If, starting from C, we wish the result of executing $M := N$ to satisfy C', then we demand, starting from C, M named m terminates (and becomes a reference label) to reach C_0, and, in turn, N named n evaluates from C_0 to reach C' with its occurrences of n substituted for $!m$.*

Note $\{C\}\,M := N\,\{C'\}$ stands for $\{C\}\,M := N :_u \{u = () \wedge C'\}$ with u fresh (which is justified because $C[()/x] \equiv C$ always holds when x has the unit type). A simple example of its usage follows. Finally when either or both sides of the assignment involve programs which are not terms in the logic (such as calls to procedures), we can use the most general rule, [*Assign*] in Figure 6. A simple example of its usage follows (the first line is already reasoned in the previous page).

1. $\{\mathsf{T}\}\ (\lambda x.x)y :_m \{m = y\}$	(Var, Abs, App)
2. $\{m = y\ \wedge\ 1 = 1\}\ 1 :_n \{m = y \wedge n = 1\}$	(Const)
3. $(m = y \wedge n = 1) \quad \supset \quad (!y = 1)\{\!\lfloor n/!m \rfloor\!\}$	
4. $\{m = y\ \wedge\ 1 = 1\}\ 1 :_n \{(!y = 1)\{\!\lfloor n/!m \rfloor\!\}\}$	(Consequence)
5. $\{\mathsf{T}\}\ (\lambda x.x)y\ :=\ 1\,\{!y = 1\}$	(1, 4, Assign)

Line 3 is derived as:

$$\begin{array}{lll} (m{=}y \wedge n{=}1) & \supset & [!m]\,(m{=}y \wedge n{=}1) \wedge \langle !m\rangle\, !m{=}n \\ & \supset & \langle !m\rangle\,(m{=}y \wedge n{=}1 \wedge !m{=}n) \\ & \supset & (!y{=}1)\{\!\lfloor n/!m \rfloor\!\}. \end{array}$$

The rule may be understood by contrasting it with the corresponding rule for the non-aliased sublanguage in [37]. The original rule reads.

$$[\mathit{AssignOrg}]\ \frac{\{C\}\ M :_m \{C'[m/\,!x]\}}{\{C\}\ x := M\ \{C'\}}$$

There are two differences between this original rule and [*Assign*] in Figure 6. First, [*AssignOrg*] only allows a variable as the left-value, while the [*Assign*] allows an arbitrary program. Second, the original rule uses syntactic substitution, while the present

one uses the logical one (cf.Section 5.3). The corresponding rule in the present context (only incorporating the second point) is:

$$[\mathit{AssignVar}]\ \frac{\{C\}\ M :_m \{C'\{\!|m/!x|\!\}\}}{\{C\}\ x := M\ \{C'\}}$$

where the highlighted part is the difference from [*AssignOrg*]. Clearly [*AssignVar*] is derivable from [*Assign*] through [*Var*].

In many programs, it is often the case that both sides of the assignment are expressions which are simple in the sense that they do not contain calls to procedures or abstractions. One such example is a simple assignment to a variable. A little more complex case may involve simple expressions on both sides of the assignment. One example follows.

$$\{x = y \ \wedge\ \mathit{Even}(!!y)\}\ !x := !!y + 1\ \{\mathit{Odd}(!!x)\ \wedge\ \mathit{Odd}(!!y)\} \tag{7.4}$$

Note both "$!x$" and "$!!y+1$" do not have side effects: one may also observe they are both terms of our assertion language. In such cases, we can use the following rule:

$$[\mathit{AssignSimple}]\ \frac{-}{\{C\{\!|e_2/!e_1|\!\}\}\ e_1 := e_2\ \{C\}}$$

The rule is directly derivable from [*Assign*] and the following rule (which is derivable from other rules: the derivability of this rule is easy by induction on e).

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

Above the use of e as a program indicates that it is a term in the logic and a program in our programming language at the same time. In various programming examples, we often assign part of a complex data structure to a part of another complex data structure. The rule [*AssignSimple*] gives a general rule for such cases. It should be noted that, when treating specific data structure, it is often easier to use a rule tailored for that data structure: In fact, one of the traditional rules for assignment from an array element to an array (cf. [4]) arises from [*AssignSimple*] as such a special rule. Using this rule, we can derive (7.4) as follows.

1. $x=y\wedge \mathit{Even}(!!y)\ \supset\ x=y\wedge \mathit{Even}(!!y)\wedge \mathit{Odd}(!!y+1)$

2. $x=y\wedge \mathit{Even}(!!y)\wedge \mathit{Odd}(n)\ \supset\ (\mathit{Odd}(!!x)\wedge \mathit{Odd}(!!y))\{\!|!!y+1/!(!x)|\!\}$

3. $\{x=y\wedge \mathit{Even}(!!y)\}\ !x := !!y+1\ \{\mathit{Odd}(!!x)\wedge \mathit{Odd}(!!y)\}$ (AssignSimple)

Line 2 uses:

$$\begin{array}{ll} & x = y \wedge \mathit{Even}(!!y) \wedge \mathit{Odd}(!!y+1) \\ \supset & m = !x = !y \wedge \mathit{Odd}(!!y+1) \\ \equiv & \langle !!x\rangle\,(!x = !y \wedge \mathit{Odd}(!!y+1) \wedge !!x = !!y+1) \quad \text{(Prop. 6.7, (6.20))} \\ \supset & \langle !!x\rangle\,(\mathit{Odd}(!!x) \wedge \mathit{Odd}(!!y) \wedge !!x = !!y+1) \end{array}$$

Fig. 7 Structural rules.

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

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

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

$$[\mathit{Aux}_\exists]\ \frac{\{C\}\ M :_u \{C'^{-i}\}}{\{\exists i.C\}\ M :_u \{C'\}} \quad [\mathit{Aux}_\forall]\ \frac{\{C^{-i}\}\ M :_u \{C'\}}{\{C\}\ M :_u \{\forall i.C'\}}$$

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

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

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

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 $\tilde{i}$.

Soundness of [*Assign*]. Again the proof rule for assignment is nothing but a logical way to write down the semantics of the assignment, $M := N$, as the following semantic justification of the rule shows. Below we let $\xi_0 = \xi \cdot m : \mathbf{i}$ (note $N\xi = N\xi_0$).

$$\begin{array}{lll} (\xi,\ \sigma) \models C & \Rightarrow & (M\xi,\ \sigma)\ \Downarrow_m\ (\xi \cdot m : \mathbf{i},\ \sigma_0)\ \models C_0 \\ & \Rightarrow & (N\xi_0,\ \sigma_0)\ \Downarrow_n\ (\xi_0 \cdot n : \mathbf{w},\ \sigma')\ \models C'\{\!|n/!m|\!\} \\ & \Rightarrow & ((M := N)\xi,\ \sigma)\ \Downarrow_u\ (\xi_0 \cdot u : (),\ \sigma'[\mathbf{i} \mapsto \mathbf{w}])\ \models C' \end{array}$$

where the last line is by the logical equivalence between two judgements, $\mathcal{M} \models C'\{\!|n/!m|\!\}$ and $\mathcal{M}[[\![m]\!]\mathcal{M} \mapsto [\![n]\!]\mathcal{M} \models C'$ (cf. Sections 4).

7.3 Proof Rules (2): Structural Rules

In Figure 7, we present a few structural rules. [*Promote*] and [*Consequence*] are essential for reasoning. [*Promote*] adds state information to a judgement about values: since a value never induces state change, we can assume any initial state will reach the final state without change. In [*Consequence*], we use validity in the underlying logic for inferring a weaker judgement from the premise. [*Consequence*] can be strengthened

into [*Consequence-Aux*] (due to Kleymann [41]: the use of universal content quantification is already implicit in his work), which is often useful. [*Invariance*] adds a stateless formula to the pre/post condition. The rule is immediately sound by the following reasoning:

$$\begin{array}{lll}(\xi,\,\sigma) \models C \wedge A & \Rightarrow & (M\xi,\,\sigma)\ \Downarrow_m\ (\xi{\cdot}u{:}\mathbf{v},\,\sigma')\ \models\ C' \\ & \Rightarrow & (M\xi,\,\sigma)\ \Downarrow_m\ (\xi{\cdot}u{:}\mathbf{v},\,\sigma')\ \models\ C' \wedge A\end{array}$$

The final step is because A is initially satisfied by ξ and ξ never changes by computation, i.e. its interpretation does not depend on the state part of the model. Later we show how this rule can justify a general form of the invariance rule as a derivable rule up to the consequence rule.

7.4 Located Judgement and their Proof Rules

Starting from Section 5, we have seen several examples of the usage of located assertions (whose formal definition is given in Section 5.6). Located assertions are useful because explicitly delineated write effects are essential in the presence of aliasing for precisely describing observable behaviours of programs. A conspicuous example is its use in the definition of extensionality formulae in Section 6.2. In the following we extend this to judgements and present compositional proof rules for located judgements. While these rules are derivable from the original ones, it is much more efficient to derive located judgements directly and compositionally, offering an essential tool for structured reasoning in the presence of aliasing. Along the way, we shall encounter a alias-robust version of the standard invariance rule. We first define:

Definition 7.5 (located judgement) Given C, $\Gamma;\Delta \vdash M : \alpha$ and C', as well as a finite set of terms $\{\tilde{e}\}$ of reference types, a *located judgement* has the following shape.

$$\{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\} \mathbin{@} \tilde{e} \tag{7.5}$$

where $\{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is well-typed following Section 7.1. We set $\models \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\} \mathbin{@} \tilde{e}$ holds iff

$$\models \{C \wedge y \neq \tilde{e} \wedge !y = i\}\, M^{\Gamma;\Delta;\alpha} :_u \{C' \wedge !y = i\}$$

holds, with X, y and i fresh and distinct and typed as $\mathsf{Ref}(\mathrm{X})$ and X respectively (cf. Section 5.6).

As in located assertions, $\tilde{e}$ in (7.5) is called *write effect*, or often simply *effect*, which is often just a subset of reference names from the basis. A write effect is treated as a finite set rather than as a sequence. Note y and i in the above definition are implicitly universally quantified by the definition of the satisfaction relation, cf. Section 7.1, so that its meaning closely corresponds to that of located assertions.[5]

Example 7.6 (located judgement)

[5] Since all syntactic and semantic notions in the present logic are closed under injective renaming, we can pick up arbitrary fresh and distinct y and i with exactly the same result as choosing any other.

1. A judgement { !x=i } x:=!x+1 { !x=i+1 } @ x says that the program increments the content of x and does nothing else.
2. Let $M \stackrel{\text{def}}{=} \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1)$. Then we have

$$\{\mathsf{Fact}(f)\}\ M^{\Gamma:\mathsf{Nat}}\ :_u\ \{u = x!\}\ @\emptyset$$

   with $\Gamma \stackrel{\text{def}}{=} f : \mathsf{Nat} \Rightarrow \mathsf{Nat} \cdot x : \mathsf{Nat}$ and $\mathsf{Fact}(f) \stackrel{\text{def}}{=} \forall i \lneq x.\{\mathsf{T}\}\, f \bullet i \searrow i!\{\mathsf{T}\}@\emptyset$.
3. For the same M, we have:

$$\{\mathsf{Fact}'(f)\}\ M^{\Gamma:\mathsf{Nat}}\ :_u\ \{u = x!\}\ @\, w$$

   where $\mathsf{Fact}'(f) \stackrel{\text{def}}{=} \forall i \lneq x.\{\mathsf{T}\}\, f \bullet i \searrow i!\{\mathsf{T}\}@w$. Note w is auxiliary. The judgement says: if f may have an effect at some reference, then M itself may have an effect on that reference.

The proof rules for located judgements are given in Figure 8 (for compositional rules) and Figure 9 (for structural rules). For the compositional rules, which closely follows Figure 6, the rule [*Assign*] demands that C_0 says m (the target of writing) is in the write effect (the set membership notation "$\in$" is understood to denote a disjunction of equations).

Among the structural rules in Figure 9, there are three rules which may deserve some illustration.

[*Weak*] The rule [*Weak*] adds a name to an effect, which is surely safe. As an example usage of [*Weak*], we infer :

1.	$\{\mathsf{T}\}x :_m \{m = x\}\ @\ \emptyset$	(Var)
2.	$\{\mathsf{T}\}x :_m \{m = x\}\ @\ x$	(Weak)
3.	$m = x\ \supset\ m \in \{x\}$	
4.	$\{\mathsf{T}\}3 :_n \{(!x = 3)\{\!|n/!x|\!\}\}\ @\ \emptyset$	(Const)
5.	$\{\mathsf{T}\}x := 3\{!x = 3\}\ @\ x$	(3, 4, Assign)

In Line 3, we have $(!x = 3)\{\!|n/!x|\!\} \equiv n = 3$ by Proposition 6.7 (6). Of course we can assign more complicated expressions. For example, we infer:

1.	$\{!x = 1\}x :_m \{m = x \wedge\ !x = 1\}\ @\ x \quad (m = x \wedge\ !x = 1) \supset m \in \{x\}$	
2.	$\{m = x \wedge\ !x = 1\}!x + 1 :_n \{(!x = 2)\{\!|n/!x|\!\}\}\ @\ \emptyset$	
3.	$\{!x = 1\}x := !x + 1 :_n \{n = 2\}\ @\ x$	(1, 2, Assign)

[*Thinning*] The rule symmetric to [*Weak*] is [*Thinning*], which removes a reference name from a write set. Hence the judgement becomes stronger, saying a given program modifies (if ever) content of less references. This becomes possible when the premise guarantees that the program does not change the content of the variable to be removed.

Fig. 8 Derivable proof rules with located judgements.

$$[Var]\ \frac{}{\{C[x/u]\}\ x :_u \{C\} @ \emptyset}$$

$$[Op]\ \frac{\{C\}\ M_1 :_{m_1} \{C_1\} @ \tilde{e_1} \quad \ldots \quad \{C_{n-1}\}\ M_n :_{m_n} \{C'[\mathsf{op}(m_1,\ldots,m_n)/u]\} @ \tilde{e_n}}{\{C\}\ \mathsf{op}(M_1,\ldots,M_n) :_u \{C'\} @ \tilde{e_1}\ldots\tilde{e_n}}$$

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

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

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

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

$$[Case]\ \frac{\{C^{\neg \tilde{e}}\}\ M :_m \{C_0^{\neg \tilde{e}}\} @ \tilde{e} \quad \{C_0[\mathsf{inj}_i(x_i)/m]\}\ M_i :_u \{C'^{\neg \tilde{e}}\} @ \tilde{e_i'}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\} @ \tilde{e}\tilde{e_1'}\tilde{e_2'}}$$

$$[Pair]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\} @ \tilde{e_i} \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1, m_2\rangle/u]\} @ \tilde{e_2}}{\{C\}\ \langle M_1, M_2\rangle :_u \{C'\} @ \tilde{e_1}\tilde{e_2}}$$

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

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

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

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

Note i is fresh, so that there is no constraint on i — the judgement thus says whichever value is stored in x, it does not alter its content. As an example usage of [*Thinning*], we infer, noting $C\{\!|!x/!x|\!\} \equiv C$ (cf. Proposition 6.7 (3)):

1. $(!x = i)\{\!|!x/!x|\!\} \equiv !x = i \supset x \in \{x\}$

2. $\{!x = i\}\ x := !x\ \{!x = i\}\ @\ x$ (Assign-Simple)

3. $\{\mathsf{T}\}\ x := !x\ \{\mathsf{T}\}\ @\ \emptyset$ (Thinning)

Fig. 9 Derivable structural rules for located judgements.

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

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

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

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

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

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

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

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

The inference suggests that through the use of [*Thinning*], the extensional nature of the logic is maintained in the proof rules for located judgements.

[*Invariance*] The rule says that, if we know that a program only touches a certain set of references, and if C_0 only asserts on a state which does not concern (content of) these references, then C_0 can be added to pre/post conditions as invariant for that program. In practice, we may use the two derivable (and essentially equivalent) rules given in Figure 10 (the derivability is through Proposition 6.7 (3). The first derivable rule, [*InvUniv*],

Fig. 10 Derivable invariance rules for located judgements.

$$[\mathit{InvUniv}]\ \frac{\{C\}\ M :_u \{C'\} @ \tilde{e}}{\{C\ \wedge\ [!\tilde{e}]C_0\}\ M :_u \{C'\ \wedge\ [!\tilde{e}]C_0\} @ \tilde{e}}$$

$$[\mathit{InvEx}]\ \frac{\{C\}\ M :_u \{C'\} @ \tilde{e}}{\{C\ \wedge\ \langle !\tilde{e}\rangle C_0\}\ M :_u \{C'\ \wedge\ \langle !\tilde{e}\rangle C_0\} @ \tilde{e}}$$

says that we demand all actively dereferenced names in C_0 are distinct from $\tilde{e}$, in which case surely it is invariance. In the second derived rule, [*InvEx*], we stipulate that we demand C_0 to hold only when all actively dereferenced names in C_0 are distinct from $\tilde{e}$. These two derivable rules are sometimes useful since, using them, we can add any invariance C_0 to a located judgement with a write set $\tilde{e}$ by simply prefixing with content quanfiers.

Fig. 11 Evaluation-order-independent proof rules for located judgements.

$$[\textit{Op-eoi}]\ \frac{\{C_i\}\ M_i :_{m_i} \{C_i'\} @ \tilde{e}_i\ \ (1 \le i \le n) \quad \bigwedge_i \langle !\tilde{e}_{i+1}..\tilde{e}_n\rangle C_i' \supset C'[\mathsf{op}(m_1..m_n)/u]}{\{\bigwedge_i [!\tilde{e}_1..\tilde{e}_{i-1}]C_i\}\ \mathsf{op}(M_1,...,M_n) :_u \{C'\} @ \tilde{e}_1...\tilde{e}_n}$$

$$[\textit{App-eoi}]\ \frac{\{C_1\}M :_m \{C_1'\} @ \tilde{e_1} \quad \{C_2\}\ N :_n \{\ C_2' \wedge \{\langle !\tilde{e_2}\rangle C_1' \wedge C_2'\}m \bullet n \searrow u\{C'\} @ \tilde{e_3}\} @ \tilde{e_2}}{\{C_1 \wedge [!\tilde{e}_1]C_2\}MN :_u \{C'\} @ \tilde{e_1}\tilde{e_2}\tilde{e_3}}$$

$$[\textit{Assign-eoi}]\ \frac{\{C_1\}\ M :_m \{C_1'\} @ \tilde{e_1}\ \ \{C_2\}\ N :_n \{C_2'\} @ \tilde{e_2}\ \ (\langle !\tilde{e_2}\rangle C_1' \wedge C_2) \supset (C'\{\!|n/\,!m|\!\} \wedge m \in \tilde{e})}{\{C_1 \wedge [!\tilde{e_1}]C_2\}\ M := N\ \{C'\}\ @\ \tilde{e}\tilde{e_1}\tilde{e_2}}$$

$$[\textit{Pair-eoi}]\ \frac{\{C_1\}\ M_1 :_{m_1} \{C_1'\} @ \tilde{e_1} \quad \{C_2\}\ M_2 :_{m_2} \{C_2'\} @ \tilde{e_2} \quad \langle !\tilde{e_2}\rangle C_1 \wedge C_2 \ \supset\ C'[\langle m_1, m_2\rangle/u]}{\{C_1 \wedge [!e_1]C_2\}\ \langle M_1, M_2\rangle :_u \{C'\} @ \tilde{e_1}\tilde{e_2}}$$

As one can easily observe, [*Invariance*] is a refinement of both the standard invariance rule in Hoare logic, which has the shape:

$$\frac{\vdash_{\text{Hoare}} \{C\}\ P\ \{C'\} \quad P \text{ does not touch variables in } C_0}{\vdash_{\text{Hoare}} \{C \wedge C_0\}\ P\ \{C' \wedge C_0\}} \tag{7.6}$$

and the invariance rule for non-located judgements in Figure 7 (from [37]):

$$\frac{\{C\}\ M :_u\ \{C'\}}{\{C \wedge A\}\ M :_u\ \{C' \wedge A\}} \tag{7.7}$$

The rule may also be regarded as an analogue of a similar rule studied by Reynolds, O'Hearn and others in [61, 66]. See Section 10 for full technical comparisons. Since a weakened stateless formula A in (7.7) is by definition $!x$-free for any x, [*Invariance*] above subsumes (7.7) (except we are now using located judgements). On the other hand, [*Invariance*] is justifiable using 7.7), cf. Section 6.2.

The derived invariance rules can further be combined with compositional rules for located judgements in Figure 8 to obtain the proof rules which are independent from particular evaluation order, in the sense that the correctness of the inference does not depend on the order of evaluation of expressions appearing in the rule (recall the proof rules for operators, applications, pairs, etc. all assume a fixed evaluation order, i.e. from left to right). The evaluation-order independence (**EOI** for short) in the most general case holds when two (or more) expressions involved only writes to separate stores and,

moreover, their resulting properties only rely on invariants which hold regardless of the state change induced by other expressions. Here we use a slightly stronger constraint, when the properties of each expression does not at all depend on written sets of the remaining expressions. Figure 11 lists the EOI-refinement of (located) operator/application/assignment/pairing rules. These rules are all inferred from the original rule together with two variants of the invariance rule, [*InvUniv*] and [*InvEx*].

The proof of the following result is easy and omitted (to derive [*Thinning*] we need [*Consequence-Aux*]). Below a *translation* of a located judgement is one of the instances of those presented in Definition 7.5.

Proposition 7.7 *$\{C\}\ M :_m \{C'\} @ \tilde{g}$ is derivable in the proof rules for located judgements iff its translation is derivable in the proof rules for non-located judgements.*

In later sections we shall show a few examples using localted assertions and judgements. Located judgements also play an essential role when we prove observational completeness, one of the basic results about our logic, to be discussed in Section 8.

Fig. 12 Located Proof rules for imperative idioms.

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

$$[\mathit{AssignVInit}]\ \frac{C\{\!|e/!x|\!\}\ \ !x\text{-free} \quad C \supset x = g}{\{C\}\ x := e\ \{C \wedge !x = e\} @ g} \qquad [\mathit{AssignSInit}]\ \frac{C\{\!|e'/!e|\!\}\ \ !e\text{-free} \quad C\{\!|e/!e|\!\} \supset e = g}{\{C\}\ e := e'\ \{C \wedge !e = e'\} @ g}$$

$$[\mathit{IfThenSimple}]\ \frac{\{C \wedge e\}\ M\ \{C'\} @ \tilde{g}}{\{C\}\ \mathtt{if}\ e\ \mathtt{then}\ M\ \{C'\} @ \tilde{g}}$$

$$[\mathit{IfThen}]\ \frac{\{C\}\ M :_m \{C_0\} @ \tilde{g} \quad \{C_0[\mathtt{t}/m]\}\ N\ \{C'\} @ \tilde{g'} \quad C_0[\mathtt{f}/m] \supset C'}{\{C\}\ \mathtt{if}\ M\ \mathtt{then}\ N\ \{C'\} @ \tilde{g}\tilde{g'}}$$

$$[\mathit{WhileSimple}]\ \frac{(C \wedge e)\ \supset\ e' > 0 \qquad \{C \wedge e \wedge e' = i\}\ M\ \{C \wedge e' < i\} @ \tilde{g} \qquad i \text{ fresh}}{\{C\}\ \mathtt{while}\ e\ \mathtt{do}\ M\ \{C \wedge \neg e\} @ \tilde{g}}$$

$$[\mathit{While}]\ \frac{\begin{array}{c}\{C \wedge e' = i\}\ M :_b \{A^b \wedge C \wedge e' <= i\}\ @\ \tilde{g} \\ \{C \wedge A[\mathtt{t}/b] \wedge e' = i\}\ N\ \{C \wedge e' < i\} @ \tilde{g'} \\ C \wedge A[\mathtt{t}/b]\ \supset\ e' > 0 \quad i \text{ fresh}\end{array}}{\{C\}\ \mathtt{while}\ M\ \mathtt{do}\ N\ \{C \wedge \neg e\} @ \tilde{g}\tilde{g'}} \qquad [\mathit{Seq}]\ \frac{\{C\}\ M\ \{C_0\} @ \tilde{g} \quad \{C_0\}\ N\ \{C'\} @ \tilde{g'}}{\{C\}\ M;N\ \{C'\} @ \tilde{g}\tilde{g'}}$$

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

$$[\mathit{AppSimple}]\ \frac{C \supset \{C\}\, e \bullet (e_1..e_n) \searrow u\, \{C'\} @ \tilde{g}}{\{C\}\ e(e_1..e_n) :_u \{C'\} @ \tilde{g}} \qquad [\mathit{Let}]\ \frac{\{C\}\ M :_x \{C_0\} @ \tilde{g} \quad \{C_0\}\ N :_u \{C'\} @ \tilde{g'}}{\{C\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{C'\} @ \tilde{g}\tilde{g'}}$$

7.5 Proof Rules for Imperative Idioms

Figure 12 lists several located proof rules for basic imperative idioms. The initial four assignment rules are directly derivable from the general assignment rule in Figure 8.

The next two rules for the one-branch conditional are also easily derivable from the general conditional rule in Figure 8. In [*IfThenSimple*], we assume e is also a term of boolean type in the assertion language (in fact any term e of a boolean type becomes a formula by $e = \mathtt{t}$ though such translation is seldom necessary).

The two rules for the while loop augments the standard total correctness rule by Floyd [15]. In both rules, e' (of Nat-type) functions as an index of the loop, which should be decremented at each step. In [*WhileSimple*], the guard is a simple expression. In [*While*], the guard is a general program, possibly with a side effect (which however should not increase an index). A^b means that if there is a primary name in A, it must be b. Both rules are directly derivable from the original rules through the standard encoding, as illustrated in detail in [37, Section 7.1]. Finally [*Seq-I*] (I is for indenpendence) is the EOI-version of the standard rule [*Seq*].

Fig. 13 Located proof rules for arrays.

$$[\mathit{Array}]\ \frac{\{C\}\,M :_m \{C_0\}@\tilde{g} \quad \{C_0\}\,N :_n \{C'[m[n]/u]\}@\tilde{g'} \quad C'[m[n]/u] \ \supset\ 0 \leq n < \mathsf{size}(m)}{\{C\}\ M[N] :_u \{C'\}\ @\ \tilde{g}\tilde{g'}}$$

$$[\mathit{ArraySimple}]\ \frac{C[\,e[e']/u\,] \ \supset\ 0 \leq e' < \mathsf{size}(e)}{\{C[\,e[e']/u\,]\}\ e[e'] :_u \{C\}\ @\ \emptyset}$$

One of the notable aspects of the presented logic is uniform treatment of data types. Each data type, either unit, product, or sum, or array, is characterised by a small set of axioms at the level of assertions and another small set of proof rules for judgements. Since data types compose (e.g. we can for example think of a product component of which is a reference which stores an array whose components are procedures etc.), these axioms and proof rules can treat arbitrary complex data structures. This extends to standard data types such as lists, streams, and arrays.

As a basic example, let us take a look at how to incorporate reasoning principle for arrays. Section 6.3, already introduced the array data type to our programming and assertion languages, with a term of the form a and $e[e']$ together with its axiom. In Figure 13, we present the located version of the proof rule for array. [*Array*], together with the axiom introduced in Section 6.3, is all we need to reason about arbitrary arrays and operations on them in imperative PCFv. This simplicity partly comes from treating arrays as a string of references, cf. [4]. The second rule in Figure 13 is a derivable version of [*Array*] for simple expressions, which is often useful. Below we give the reading of [*SimpleArray*].

If the initial state, $C[e[e']/u]$, says that the index e' (of Nat*-type) is within the range of the size of the array e (of $\alpha[]$-type), then we can conclude the array $e[e']$ named u (of type* $\mathsf{Ref}(\alpha)$*) has the property C, with no write effect.*

The first rule just adds the state change by evaluating the array and its index.

It is instructive to see how the dynamics involving arrays, in particular assignments, can be reasoned using these rules. For example if you wish to assign a value to an array at a particular index, which is an operation we often find in practice, we can simply specialise e and e' in [*ArraySimple*] to reach the following rule:

$$[\mathit{AssignArray}]\ \frac{C[\,e'\,/\,!a[e]\,]\ \supset\ 0 \leq e < \mathsf{size}(a)}{\{C\{\!|\,e'\,/\,!a[e]\,|\!\}\}\ a[e] := e'\ \{C\}}$$

The rule is direct combination of [*AssignSimple*] and [*ArraySimple*]. It is worth expanding the precondition in the conclusion. Let m be fresh below.

$$C\{\!|\,e'\,/\,!a[e]\,|\!\} \quad \stackrel{\text{def}}{=} \quad \exists m.(\langle !\,a[e]\rangle\,(C \wedge\ !a[e] = m)\ \wedge\ m = e') \tag{7.8}$$

In the right-hand side of (7.8), if C contains a term of the form $!a[e'']$, then if (C says) $e = e''$ then it is equated with m (hence e'); if not, it is unaffected by m. This case analysis is precisely what underlies the standard proof rule for array assignment, as presented in [4, pp. 461], which is subsumed by the proof rule above. It is notable that [*AssignArray*] can be used when array names themselves can be aliased which is a common situation in systems programming. This is not considered in the standard proof rules for arrays. On the other hand, the specialised rule in [4] is quite useful when such aliasing is guaranteed never to take place and derivable in our system. Ultimately general rules such as [*Array*] and [*Assign*] may not be often used directly except for justifying more specific rules, even in mechanised proofs. But this is precisely how the fundamental axioms are used in mathematics, where basic axioms serve as an unshakable basis upon which a rich theory covering many concrete theorems and observations can be built on and to which one can come back when one wishes to obtain the most distilled presentation of basic concepts (not that we are certain our axioms have reached such a stage, but that should be our aspiration).

8 Elimination, Soundness and Observational Completeness

In this section, we present some of the basic technical results about the proposed logic.

8.1 Elimination

Using the axioms for content quantification introduced in Section 6, we establish a major technical result on our logic, the elimination of content quantification. In other words, any assertion which is written using content quantifications can be equivalently written without them. Some observations on this result before going into technical development:

- The result makes clear the logical status of these modal operators; in particular, semantically, we now know they add no more complexity than (in)equations on reference names. Since (in)equations on reference names can be easily defined using content quantifiers, we know these two notions — quantifying over content of references and discussing equalities of reference names — are essentially one and the same thing.
- As a consequence, the validity of the assertion language is that of the standard predicate calculus with equality apart from the use of evaluation formulae.
- The elimination procedure only uses the axioms for content quantifications discussed in Section 6.1 combined with the well-known axioms for equality and (standard) quantifiers. Thus, relative to the underlying axioms of the predicate calculus with equality as well as those for evaluation formulae, the axioms give complete characterisation of these modal operators.

The arguments towards the elimination theorem reveal close connection between content quantification, logical (semantic) substitutions $C\{\!|e'/!e|\!\}$ and equations on names. Practically, this connection suggests the effectiveness of their combined use in logical calculations.

The elimination is done by syntactically transforming a formula in the following three steps. Assume given $[!e]C$ or $\langle !e\rangle C$ where C does not contain content quantification (since the transformation is local this suffices).

1. We transform the content quantification into the corresponding logical substitution applied to C.
2. We transform C into the form of $\exists \tilde{r}.(C_1 \wedge C_2)$ where C_1 do not contain any active dereference while C_2 extracts all active dereferences occurring in C (this step is not necessary strictly speaking but contributes to the conciseness of the resulting formulae).
3. By the self-dual nature of logical substitutions, we can compositionally dissolve the outermost application of the logical substitution, so that it now only affects each atomic equation in C_2 (C_1 is simply neglected). We then apply the axioms for content quantification to turn each into an assertion without content quantifications.

We start from the first step, which underpins a close connection between content quantifications and logical substitutions.

Proposition 8.1 *With m fresh, we have $[!e]C \equiv \forall m.C\{\!|m/!e|\!\}$. Dually, again with m fresh, we have $\langle !e\rangle C \equiv \exists m.C\{\!|m/!e|\!\}$.*

Proof It suffices to treat the case when $e \stackrel{\text{def}}{=} x$. Let m be fresh below.

$$\begin{array}{rcl} \forall m.C\{\!|m/!x|\!\} & \equiv & \forall m.C\overline{\{\!|m/!x|\!\}} \\ & \equiv & [!x]\,\forall m.(!x = m \supset C) \\ & \equiv & [!x]\,C \end{array}$$

While the second statement is dual, we anyway record it:

$$\begin{array}{rcl} \exists m.C\{\!|m/!x|\!\} & \equiv & \exists m.\langle !x\rangle\,(C \wedge !x = m) \\ & \equiv & \langle !x\rangle\,\exists m.(C \wedge !x = m) \\ & \equiv & \langle !x\rangle\,C \end{array}$$

hence done. □

The following Lemma is from Cartwright and Oppen [11]. The proof uses the self-dual nature of logical substitutions (as above). Below the condition $z \notin \{x,y\}$ is not substantial since z can be renamed by α-convertibility.

Lemma 8.2 *The following equivalences hold with $\star \in \{\wedge,\vee,\supset\}$ and $Q \in \{\forall,\exists\}$.*

$$\begin{array}{rcl} (C_1 \star C_2)\{\!|y/!x|\!\} & \equiv & C_1\{\!|y/x|\!\} \star C_2\{\!|y/!x|\!\} \\ (\neg C)\{\!|y/!x|\!\} & \equiv & \neg(C\{\!|y/!x|\!\}) \\ (Qz.C)\{\!|y/!x|\!\} & \equiv & Qz.(C\{\!|y/!x|\!\}) \\ \{C\}e \bullet e' \searrow x\{C'\}\{\!|y/!x|\!\} & \equiv & \exists uv.\,(\{C\}u \bullet v \searrow w\{C'\} \ \wedge \ (u = e \ \wedge \ v = e')\{\!|y/!x|\!\}) \\ C^{-!x}\{\!|y/!x|\!\} & \equiv & C^{-!x} \end{array}$$

In the third line we assume $z \notin \{x,y\}$.

Proof It suffices to prove the cases of $\star = \wedge$ and $Q = \forall$ as well as the negation. For $\wedge$:

$$\begin{array}{rcl} (C_1 \ \wedge \ C_2)\{\!|y/!x|\!\} & \equiv & (C_1 \ \wedge \ C_2)\overline{\{\!|y/!x|\!\}} \\ & \equiv & \forall m.(y = m \supset [!x]\,(!x = m \supset (C_1 \wedge C_2))) \\ & \equiv & \forall m.(y = m \supset [!x] \wedge_i (!x = m \supset C_i)) \\ & \equiv & \forall m.(y = m \supset \wedge_i [!x]\,(!x = m \supset C_i)) \\ & \equiv & \wedge_i \forall m.(y = m \supset [!x]\,(!x = m \supset C_i)) \\ & \equiv & C_1\{\!|y/!x|\!\} \ \wedge \ C_2\{\!|y/!x|\!\} \end{array}$$

For $\forall$:

$$\begin{array}{rcl} (\forall z.C)\{\!|y/!x|\!\} & \equiv & \forall z.(C\overline{\{\!|y/!x|\!\}}) \\ & \equiv & \forall m.(y = m \supset [!x]\,(!x = m \supset \forall z.C)) \\ & \equiv & \forall m.(y = m \supset [!x]\,\forall z.(!x = m \supset C)) \\ & \equiv & \forall m.(y = m \supset \forall z.[!x]\,(!x = m \supset C)) \\ & \equiv & \forall m.\forall z.(y = m \supset [!x]\,(!x = m \supset C)) \\ & \equiv & \forall z.\forall m.(y = m \supset [!x]\,(!x = m \supset C)) \\ & \equiv & \forall z.(C\{\!|y/!x|\!\}). \end{array}$$

Finally negation:

$$\begin{array}{rcl}
\neg(C\{\!|y/!x|\!\}) & \equiv & \neg(\exists m.(\langle !x\rangle\,(C \wedge !x = m) \ \wedge \ m = y)) \\
& \equiv & \forall m.([!x]\,(\neg C \vee !x \neq m) \ \vee \ m \neq y)) \\
& \equiv & \forall m.(m = y \ \supset \ [!x]\,(!x = m \ \supset \ \neg C)) \\
& \equiv & (\neg C)\,\overline{\{\!|y/!x|\!\}} \\
& \equiv & (\neg C)\{\!|y/!x|\!\}
\end{array}$$

At the last step we use the self-dualisation.

Now we move to the second step.

Lemma 8.3 *Assume C does not contain content quantifications. Then we can rewrite C in the following form up to the logical equivalence:*

$$\exists \tilde{r}.(\ (\bigwedge_i c_i = !r_i) \ \ \wedge \ \ C')$$

where (1) $\tilde{r}\tilde{c}$ are fresh and (2) C' do not contain any active dererefence.

Proof We repeatedly apply the following transformation by induction on i. We first set $C_0 \stackrel{\text{def}}{=} C$. Now assume $!e_i$ occur in C_i such that e_i does not contain a dereference. Then we let C_{i+1} to be the result of replacing $!e_i$ with c_i in C_i and taking the conjunction of the resulting formula and $r_i = e_i$. If there are n active dererefences, then C' will be given as C_n, which, by definition, does not contain any active dereferences. The logical equivalence is immediate. □

Now we are in the final stage. First we note we can decompose the logical substitution $(!u = z)\{\!|m/!x|\!\}$ with m fresh, in the following way.

$$\begin{array}{rcl}
\langle !x\rangle\,(!u = z \wedge !x = m) & \equiv & \langle !x\rangle\,(\ (x = u \wedge !u = z \wedge !x = m) \ \vee \ (x \neq u \wedge !u = z \wedge !x = m)) \\
& \equiv & \langle !x\rangle\,(x = u \wedge !u = z \wedge !x = m) \ \ \vee \ \ \langle !x\rangle\,(x \neq u \wedge !u = z \wedge !x = m) \\
& \equiv & (x = u \wedge m = z) \ \ \vee \ \ \langle !x\rangle\,(x \neq u \wedge !u = z \wedge !x = m) \\
& \equiv & (x = u \wedge m = z) \ \ \vee \ \ ((x \neq u \wedge !u = z) \ \wedge \ \langle !x\rangle\, !x = m) \\
& \equiv & (x = u \wedge m = z) \ \ \vee \ \ (x \neq u \wedge !u = z).
\end{array}$$

Write $[\![(!u = z)\{\!|m/!x|\!\}]\!]$ for the final formula above. Using the notations in Lemma 8.3, and assuming C does not contain content quantifications, we reason, with m etc. fresh and noting, when m is fresh, we have $C\{\!|m/!x|\!\} \equiv \langle !x\rangle\,(C \wedge !x = m)$:

$$\begin{array}{rcll}
\langle !x\rangle C & \equiv & \exists m.C\{\!|m/!x|\!\} & \text{(Lem.8.1)} \\
& \equiv & \exists m.(\exists \tilde{r}\tilde{c}.(\ (\wedge_i !r_i = c_i) \ \ \wedge \ \ C'))\{\!|m/!x|\!\} & \text{(Lem.8.3)} \\
& \equiv & \exists m.(\exists \tilde{r}\tilde{c}.(\ (\wedge_i !r_i = c_i)\{\!|m/!x|\!\} \ \ \wedge \ \ C')) & \text{(Lem.8.2)} \\
& \equiv & \exists m.(\exists \tilde{r}\tilde{c}.(\ (\wedge_i [\![(!r_i = c_i)\{\!|m/!x|\!\}]\!]) \ \ \wedge \ \ C')) &
\end{array}$$

By performing this transformation from each maximal subformula which does not contain content quantifications, we can completely eliminate all content quantifications from any given formula. We have thus arrived at:

Theorem 8.4 *For each well-typed assertion C, there exists C' which satisfies the following properties: (1) $C \equiv C'$ and (2) no content quantification occurs in C'.*

Note the elimination procedure also tells us:

Proposition 8.5 *For any C, $[!x]C$ is equivalent to a formula of the shape:*

$$\exists \tilde{r}.(C' \wedge \bigwedge_i r_i \neq x)$$

where $\tilde{r}$ exhaust all active dereferences in C'.

Proof Just perform the elimination procedure until we reach the final step, at which point we use $[!x]\,!r = z \equiv x \neq r$. □

We conclude this subsection with the following observation. Let $x = y$ be an equation on reference names. It is easy to check this equation is logically equivalent to $[!x]\,[!y]\,!x = !y$, except when x and y are of the type $\mathsf{Ref}(\mathsf{Unit})$. Thus we can replace all (in)equations on reference names with content quantifications as far as we exclude the trivial store of type $\mathsf{Ref}(\mathsf{Unit})$ from our discussion. Together with Theorem 8.4, we conclude that content quantifications and reference name (in)equations are mutually representable.

8.2 Soundness

We next establish soundness of axioms and proof rules for non-located and located judgements. Let us start from the proof rules. We have already seen that [*Assign*] and [*Deref*] are semantically justifiable. As noted there, all other rules are equally easily justified. While many proofs precisely follow those in [37, Section 5], we give the rest below, omitting proofs obvious from those given. We write $(\xi \cdot m{:}M,\ \sigma) \Downarrow (\xi \cdot m{:}\mathbf{v},\ \sigma') \models C$ when $(M\xi,\ \sigma) \Downarrow (\mathbf{v},\ \sigma')$ and $(\xi \cdot m{:}\mathbf{v},\ \sigma') \models C$ for some $\mathbf{v}$ and σ'. First, the rule for a variable:

$$\begin{array}{rl} (\xi,\ \sigma) \models C[x/u] & \Rightarrow\ (\xi \cdot u{:}\xi(x),\ \sigma) \models C \wedge u = x \\ & \Rightarrow\ (\xi \cdot u{:}x,\ \sigma) \Downarrow (\xi \cdot u{:}\xi(x),\ \sigma) \models C \end{array}$$

The rule for a constant is the same as above. For n-ary operator with $n \geq 1$, we show the case $n = 2$ for readability.

$$\begin{array}{l} (\xi,\ \sigma) \models C[x/u] \wedge\ \models \{C\}M_1 :_{m_1} \{C_1\} \wedge\ \models \{C_1\}M_2 :_{m_2} \{C_2[\mathsf{op}(m_1 m_2)/u]\} \\ \quad \Rightarrow\ (\xi \cdot m_1 : M_1,\ \sigma) \Downarrow (\xi \cdot m_1 : \mathbf{v_1},\ \sigma_1) \wedge \\ \qquad (\xi \cdot m_1 : \mathbf{v_1} \cdot m_2 : M_2,\ \sigma_1) \Downarrow (\xi \cdot m_1 : \mathbf{v_1} \cdot m_2 : \mathbf{v_2},\ \sigma') \models C_2 \wedge u = \mathsf{op}(m_1 m_2) \\ \quad \Rightarrow\ (\xi \cdot u{:}\mathsf{op}(M_1 M_2),\ \sigma) \Downarrow (\xi \cdot u{:}\mathsf{op}(\mathbf{v_1}\mathbf{v_2}),\ \sigma') \models C_2 \end{array}$$

The general n-ary case is similarly inferred. For abstraction let $\xi' \stackrel{\text{def}}{=} \xi \cdot x : \mathbf{v}$ below.

$$\begin{array}{ll} (\xi,\sigma) \models A & \\ \quad \Rightarrow & \forall \mathbf{v}.(\ (\xi \cdot x : \mathbf{v},\ \sigma) \models A \wedge C\ \supset\ (M\xi',\ \sigma) \Downarrow_m (\xi' \cdot m : \mathbf{w},\ \sigma') \models C'\ ) \\ \quad \Rightarrow & \forall \mathbf{v}.((\xi \cdot x : \mathbf{v},\ \sigma) \models A \wedge C\ \supset\ ((\lambda x.M)\xi\mathbf{v},\ \sigma) \Downarrow_m (\xi' \cdot m : \mathbf{w},\ \sigma') \models C') \\ \quad \Rightarrow & (\xi \cdot u : (\lambda x.M)\xi,\ \sigma) \models \forall x.\{C\}u \bullet x \searrow m\{C'\} \end{array}$$

For application we infer, with $\xi_0 = \xi \cdot m : \mathbf{v}$:

$$\begin{array}{ll}(\xi,\, \sigma) \models C & \\ \quad \Rightarrow & (M\xi,\, \sigma) \Downarrow_m (\xi \cdot m : \mathbf{v},\, \sigma_0) \models C_0 \\ \quad \Rightarrow & (N\xi_0,\, \sigma_0) \Downarrow_m (\xi_0 \cdot n : \mathbf{w},\, \sigma_1) \models C_1 \wedge \{C_1\} m \bullet n \searrow u \{C'\} \\ \quad \Rightarrow & (\mathbf{vw},\, \sigma) \Downarrow_u (\xi \cdot u : \mathbf{u},\, \sigma') \models C' \\ \quad \Rightarrow & ((MN)\xi,\, \sigma) \Downarrow_u (\xi \cdot u : \mathbf{u},\, \sigma') \models C'\end{array}$$

A pair and a projection are similar. For conditional, we set $\mathbf{b_1} \stackrel{\text{def}}{=} \mathtt{t}$ and $\mathbf{b_2} \stackrel{\text{def}}{=} \mathtt{f}$.

$$\begin{array}{l}(\xi,\, \sigma) \models C \wedge \ \models \{C\} M :_m \{C_0\} \wedge \ \models \{C_0[\mathbf{b_i}/m]\} N_i :_u \{C\} \quad (i \in \{1,2\}) \\ \ \Rightarrow\ (\xi \cdot m : M,\, \sigma) \Downarrow (\xi \cdot m : \mathbf{b_i},\, \sigma_i) \models C_0 \ \wedge\ (\xi \cdot u : N_i,\, \sigma_i) \Downarrow (\xi \cdot u : \mathbf{v_i},\, \sigma') \models C' \ (i \in \{1,2\}) \\ \ \Rightarrow\ (\xi \cdot u : \mathtt{if}\, M\, \mathtt{then}\, N_1\, \mathtt{else}\, N_2,\, \sigma) \Downarrow (\xi \cdot u : \mathbf{w},\, \sigma') \models C'\end{array}$$

Above we used the fact that closed boolean values are exhausted by `t` and `f`. For the case construct.

$$\begin{array}{l}(\xi,\, \sigma) \models C \wedge \ \models \{C\} M :_m \{C_0\} \wedge \ \models \{C_0[\mathtt{in}_i(x)/m]\} N_i :_u \{C\} \quad (i \in \{1,2\}) \\ \ \Rightarrow\ (\xi \cdot m : M,\, \sigma) \Downarrow (\xi \cdot m : \mathtt{in}_i(\mathbf{v_i}),\, \sigma_i) \models C_0 \ \wedge \\ \qquad (\xi \cdot x : \mathbf{v_i} \cdot u : N_i,\, \sigma_i) \Downarrow (\xi \cdot x : \mathbf{v_i} \cdot u : \mathbf{v_i},\, \sigma') \models C' \quad (i \in \{1,2\}) \\ \ \Rightarrow\ (\xi \cdot u : \mathtt{case}\, M\, \mathtt{of}\, \{\mathtt{in}_i(x) N_i\}_{i \in \{1,2\}},\, \sigma) \Downarrow (\xi \cdot u : \mathbf{w},\, \sigma') \models C'\end{array}$$

Above we used the fact that closed values of sum types are of the form $\mathtt{in}_i(V)$ with $i \in \{1,2\}$. Similarly for other compositional and structural rules, so that we conclude:

Theorem 8.6 *(soundness) If* $\vdash \{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ *then* $\models \{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$.

For the proof rules for located judgements, we can show they are derivable from those for non-located ones, or, alternatively, we can reason directly. In either case, the only non-trivial case is [*Thinning*]. This is reasoned using simple instances of [*Consequence-Aux*] (renaming of auxiliary names) combined with disjunction on pre/post conditions (derived from [$\vee$-*pre*] and [*Consequence*]) as follows.

$$\begin{array}{ll}\models \{C \wedge z \neq \tilde{e} e' \wedge !z = i \wedge !e' = i'\}\ M :_u \{C' \wedge z \neq \tilde{e} e' \wedge !z = i \wedge !e' = i'\} & \\ \quad \Rightarrow & \models \{C \wedge z \neq \tilde{e} \wedge z \neq e' \wedge !z = i\}\ M :_u \{C' \wedge z \neq \tilde{e} \wedge z \neq e' \wedge !z = i\}\ \wedge \\ & \models \{C \wedge z \neq \tilde{e} \wedge z = e' \wedge !z = i\}\ M :_u \{C' \wedge z \neq \tilde{e} \wedge z = e' \wedge !z = i\} \\ \quad \Rightarrow & \models \{C \wedge z \neq \tilde{e} \wedge !z = i\}\ M :_u \{C' \wedge z \neq \tilde{e} \wedge !z = i\}\end{array}$$

Soundness of other located rules is as for the corresponding unlocated rules. We conclude, with respect to the semantics given in Definition 7.5:

Theorem 8.7 *(soundness of located judgements) If* $\vdash \{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}\ @\ \tilde{g}$ *then* $\models \{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}\ @\ \tilde{g}$.

We now show correctness of all axioms.

Theorem 8.8 *All axioms in Figures 2, 3, 4 and 5 are valid. Further (CGen) in Figure 2 is sound in the sense that if* C *is valid then so is* $[!x]C$.

Proof We only show the cases for the main axioms for content quantifications. We mainly concentrate on the universal part. For (u0):

$$\begin{array}{lcl}
\mathcal{M} \models [!x]C' & \equiv & \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C' \\
& \supset & \forall \mathbf{v}.(\,\mathcal{M}[x \mapsto \mathbf{v}] \models C' \supset C \text{ implies } \mathcal{M}[x \mapsto \mathbf{v}] \models C\,) \\
& \equiv & \forall \mathbf{v}.\,\mathcal{M}[x \mapsto \mathbf{v}] \models (C' \supset C) \supset C \\
& \equiv & \mathcal{M} \models [!x]\,((C' \supset C) \supset C)
\end{array}$$

For (u1):

$$\begin{array}{lcl}
\mathcal{M} \models ([!x]C_1) \wedge ([!x]C_2) & \equiv & \mathcal{M} \models ([!x]C_1) \text{ and } \mathcal{M} \models ([!x]C_2) \\
& \equiv & \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \text{ and } \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_2 \\
& \equiv & \forall \mathbf{v}.(\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \text{ and } \mathcal{M}[x \mapsto \mathbf{v}] \models C_2) \\
& \equiv & \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \wedge C_2 \\
& \equiv & \mathcal{M} \models [!x]\,(C_1 \wedge C_2)
\end{array}$$

For (u2):

$$\begin{array}{lcl}
\mathcal{M} \models [!x]C_1 \ \vee\ [!x]C_2 & \equiv & \mathcal{M} \models ([!x]C_1) \text{ or } \mathcal{M} \models ([!x]C_2) \\
& \equiv & \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \text{ or } \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_2 \\
& \supset & \forall \mathbf{v}.(\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \text{ or } \mathcal{M}[x \mapsto \mathbf{v}] \models C_2) \\
& \equiv & \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \vee C_2 \\
& \equiv & \mathcal{M} \models [!x]\,(C_1 \vee C_2)
\end{array}$$

For (u3):

$$\begin{array}{lcl}
\mathcal{M} \models [!x]\,(C_1 \vee C_2) & \equiv & \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \vee C_2 \\
& \equiv & \forall \mathbf{v}.(\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \text{ or } \mathcal{M}[x \mapsto \mathbf{v}] \models C_2) \\
& \supset & \exists \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_1 \text{ or } \forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C_2 \\
& \equiv & \mathcal{M} \models \langle !x\rangle C_1 \text{ or } \mathcal{M} \models [!x]C_2 \\
& \equiv & \mathcal{M} \models \langle !x\rangle C_1 \ \vee\ [!x]C_2
\end{array}$$

For (u4):

$$\begin{array}{lcl}
M \models [!x]C & \equiv & \forall \mathbf{w}.M[x \mapsto \mathbf{w}] \models C \\
& \supset & M \models C
\end{array}$$

For (u5):

$$\begin{array}{lcl}
\mathcal{M} \models \forall y.[!x]C & \equiv & \forall \mathbf{w}.\forall \mathbf{v}.(\mathcal{M} \cdot y : \mathbf{v})[x \mapsto \mathbf{w}] \models C \\
& \equiv & \forall \mathbf{v}.\forall \mathbf{w}.(\mathcal{M}[x \mapsto \mathbf{w}]) \cdot y : \mathbf{v} \models C \\
& \equiv & \mathcal{M} \models [!x]\forall y.C
\end{array}$$

For (u6):

$$\begin{array}{lcl}
\mathcal{M} \models \forall x.!x \neq y & \Rightarrow & \forall \mathbf{v}.(\mathcal{M} \cdot x : \{x\} \cdot \{x\} \mapsto \mathbf{v} \models !x \neq y) \\
& \Rightarrow & \mathcal{M} \cdot x : \{x\} \cdot \{x\} \mapsto [\![y]\!]_{\mathcal{M}} \models !x \neq y) \\
& \Rightarrow & \mathsf{F}
\end{array}$$

For (u7), let $\mathcal{M} \vdash x = y$ (the case when $\mathcal{M} \vdash x \neq y$ is trivial).

$$\begin{array}{rcl}
\mathcal{M} \models \forall y.[!x]C & \equiv & \forall \mathbf{w}.\forall \mathbf{v}.(\mathcal{M}[y \mapsto \mathbf{v}][x \mapsto \mathbf{w}] \models C) \\
& \equiv & \forall \mathbf{w}.\forall \mathbf{v}.(\mathcal{M}[x \mapsto \mathbf{w}] \models C) \\
& \equiv & \forall \mathbf{w}.(\mathcal{M}[x \mapsto \mathbf{w}] \models C) \\
& \equiv & \forall \mathbf{w}.(\mathcal{M}[y \mapsto \mathbf{w}] \models C) \\
& \equiv & \forall \mathbf{v}.\forall \mathbf{w}.(\mathcal{M}[y \mapsto \mathbf{w}][x \mapsto \mathbf{v}] \models C) \\
& \equiv & \mathcal{M} \models [!x]\forall y.C
\end{array}$$

For (u8):

$$\begin{array}{rcl}
\mathcal{M} \models \langle !x \rangle [!x]C & \equiv & \forall \mathbf{w}.\exists \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}][x \mapsto \mathbf{w}] \models C \\
& \equiv & \forall \mathbf{w}.\mathcal{M}[x \mapsto \mathbf{w}] \models C \\
& \equiv & \mathcal{M} \models [!x]C
\end{array}$$

For (u9):

$$\begin{array}{rcl}
\mathcal{M} \models [!x]\, !x \neq y & \Rightarrow & \forall \mathbf{v}.(\mathcal{M}[x \mapsto \mathbf{v}] \models !x \neq y) \\
& \Rightarrow & \mathcal{M}[x \mapsto [\![y]\!]_{\mathcal{M}} \models !x \neq y) \\
& \Rightarrow & \mathsf{F}
\end{array}$$

For (ua) we need a lemma:

Lemma. Suppose $\xi(x) \notin \xi(\tilde{y})$ and $!\tilde{y}$ exhaust active dereferences in C. Then $(\xi,\ \sigma) \models C$ implies $(\xi,\ \sigma) \models [!x]C$.

The lemma holds since the satisfiability of C only depends on non-x part of σ. We now reason, assuming for simplicity r is the only active dereference in C':

$$\begin{array}{rcl}
(\xi, \sigma) \models \exists r.(C' \wedge x \neq r) & \equiv & (\xi \cdot r : \{r\},\ \sigma \cdot \{r\} \mapsto \mathbf{v}) \models C' \wedge x \neq r \ \vee \\
& & \exists \mathbf{i}.\ (\xi \cdot r : \mathbf{i},\ \sigma) \models C' \wedge x \neq r
\end{array}$$

where, in the second disjunct, $\mathbf{i}$ is the existing identical to which r is joined. We only show the case of the latter, noting $\xi(x) \neq \mathbf{i}$:

$$\begin{array}{rcl}
(\xi \cdot r : \mathbf{i},\ \sigma) \models C' \wedge x \neq r & \Rightarrow & \forall \mathbf{w}.\ ((\xi \cdot r : \mathbf{i},\ \sigma[x \mapsto \mathbf{w}]) \models C' \wedge x \neq r) \\
& \Rightarrow & \forall \mathbf{w}.\ ((\xi \cdot r : \mathbf{i},\ \sigma[x \mapsto \mathbf{w}]) \models C' \wedge x \neq r) \\
& \Rightarrow & \forall \mathbf{w}.\ ((\xi,\ \sigma[x \mapsto \mathbf{w}]) \models \exists r.(C' \wedge x \neq r) \\
& \Rightarrow & (\xi,\ \sigma) \models [!x]\,\exists r.(C' \wedge x \neq r)
\end{array}$$

as required. For (ue0):

$$\begin{array}{rcl}
\mathcal{M} \models \neg[!x]C & \equiv & \neg\mathcal{M} \models [!x]C \\
& \equiv & \neg\forall \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models C \\
& \equiv & \exists \mathbf{v}.\neg\mathcal{M}[x \mapsto \mathbf{v}] \models C \\
& \equiv & \exists \mathbf{v}.\mathcal{M}[x \mapsto \mathbf{v}] \models \neg C \\
& \equiv & \mathcal{M} \models \langle !x \rangle\, \neg C.
\end{array}$$

We conclude with the proof of (ue1).

$$\begin{array}{rcl}
\mathcal{M} \models [!x]\,(!x = m \supset C) & \equiv & \forall \mathbf{v}.(\mathcal{M}[x \mapsto \mathbf{v}] \models !x = m \supset C \\
& \equiv & \forall \mathbf{v}.\ (\ \mathcal{M}[x \mapsto \mathbf{v}] \models !x = m \quad \supset \quad \mathcal{M}[x \mapsto \mathbf{v}] \models C\ ) \\
& \equiv & \mathcal{M}[x \mapsto [\![m]\!]_{\mathcal{M}}] \models C \\
& \equiv & \mathcal{M}[x \mapsto [\![m]\!]_{\mathcal{M}}] \models C \wedge !x = m \\
& \equiv & \mathcal{M} \models \langle !x \rangle C \wedge !x = m
\end{array}$$

□

8.3 Observational Completeness (1): General Idea

A basic property of the proposed logic, which is directly about validity but which is in fact closely related with provability, is its precise connection with observational semantics of imperative PCFv with full aliasing. We are motivated by the following question:

> *Are two programs contextually equivalent if and only if they satisfy the same set of assertions? That is, does $M_1 \cong M_2$ holds if and only if, for each C and C', $\models \{C\}M_1 :_u \{C'\}$ implies $\models \{C\}M_2 :_u \{C'\}$ and vice versa?*

The property, well-known since Hennessy and Milner [23], attests a program logic concerned talks no more and no less than observational behaviours of programs, up to a canonical congruence. To show this property (which we call observational completeness for brevity) holds for our logic, we show the following stronger property for a subset of programs which represent a limited class of behaviours:

> *For each program, is there an assertion pair (called* characteristic formulae*) which fully describes its behaviour? That is, for each M can we find C and C' such that (i) $\models \{C\}M :_u \{C'\}$ and (ii) if $\models \{C\}N :_u \{C'\}$ implies $M \cong N$?*

Using the notion of characteristic formulae, the proof of observational completeness precisely follows our development in [37]. involving the following three steps.

Step 1: We introduce a variant of *finite canonical forms* (FCFs) [3, 35, 38] which represent a limited class of behaviours and whose properties are readily extracted.

Step 2: We show characteristic formulae of FCFs w.r.t. total correctness are derivable using our proof rules.

Step 3: By reducing a differentiating context of two programs to FCFs and further to their characteristic formulae, we show any semantically distinct programs can be differentiated by an assertion, leading to the characterisation of $\cong$ by validity.

We outline the proof of this result in the following. Step 2 uses a specialised proof rule for FCFs which elucidates the semantic foundation of the presented proof rules.

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.

One 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.

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

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

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

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

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

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

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

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

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

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

8.4 Observational Completeness (2): Characteristic Formulae and FCFs

The notion of characteristic formulae only use the following class of assertions. Below $\sqsubseteq$ is the pre-order counterpart of $\cong$ (i.e. we first define $\sqsubseteq_{\mathcal{D}}$ as in Definition 3.3 except $\Leftrightarrow$ is replaced by $\Rightarrow$, then $M \sqsubseteq N$ iff $M \sqsubseteq_{\mathcal{D}} N$ for each $\mathcal{D}$).

Definition 8.9 (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 8.10 (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 8.9 except changing $\sqsubseteq$ with $\sqsupseteq$).

Definition 8.11 (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 8.12 (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. $\mathsf{fv}(C) \setminus \mathsf{dom}(\Gamma)$).

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 8.13 (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 $(\xi, \sigma) \models C \wedge E$, then $(M\xi, \sigma) \Downarrow (V, \sigma')$ implies $(N\xi, \sigma) \Downarrow (W, \sigma'')$ such that $\Delta \vdash V \sqsubseteq W : \alpha$ and $\Delta \vdash \sigma' \sqsubseteq \sigma''$.

Proposition 8.14 *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. □

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.

Next we introduce a subset of programs which a restricted class of behaviours. In spite of their restricted behaviours, they are powerful enough to differentiate any two semantically distinct programs; while thanks to their restricted behaviours, we can derive strong CAPs for them syntactically.

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. PCFv.

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

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 we regard reference names as constants. We omit the obvious translation to imperative PCFv-terms and typing rules. The following result is basic to our later technical development.

Lemma 8.15 *Let $M_{1,2}$ be values and $\Delta \vdash M_1 \cong\!\!\!\!\!/\; M_2 : \alpha$. Then there exist semi-closed FCF values, F and $\tilde{U}$, which satisfies the following condition: for some $\{\tilde{r}\} \supset \mathsf{dom}(\Delta)$, $(FM_i, \tilde{r} \mapsto \tilde{U}) \Downarrow$ and $(FM_j, \tilde{r} \mapsto \tilde{U}) \Uparrow$ where $(i, j) = (1,2)$ or $(i, j) = (2,1)$.*

Proof (outline) Assume $\Delta \vdash M_1 \cong\!\!\!\!\!/\; M_2 : \alpha$ 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. Using this, we transform these programs into FCF values maintaining the convergence property while being made less defined (we leave the details to [37, 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 further details, see [37, Section 6 and Appendix A]. □

Fig. 14 Proof rules for characteristic assertions of FCFs with aliasing.

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

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

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

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

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

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

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

8.5 Observational Completeness (3): Deriving CAPs for FCFs

In Figure 14, 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 $M^{\Gamma;\alpha}$ where Γ is the union of the environment basis and the reference basis. Some remarks on notations:

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.
3. In the rule for assignment, we use the logical substitution rather than the syntactic one, to deal with arbitrary aliasing.

Note "freshness of names" is measured w.r.t. given formulae and typing: thus the typing plays a fundamental role in the derivation. 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 [37, Section 6.5/6.6] except for the added rules and a refinement in value/let rules.

1. A strong CAP of a reference name x named u is, by expanding the located judgement with the empty write set, the pair of "$!r = i$" and "$u = x \wedge\ !r = i$" with r and i fresh. This pair says:

   Whatever the assumption on free variables and the content of references would be, a program results in a name x without any state change.

   So in effect the program satisfying this specification should be a program which silently converges to x (it cannot be an alias since this should hold under arbitrary distinctions). As in this rule, each rule for values uses the empty write set, to indicate the stateless behaviour of values. By this any program which satisfies the same specifications can only converge without depending on a given state nor inducing a state change (the corresponding rules for values in [37, Section 6.6] instead use type information for the same end).

2. The rule for the let-application follows the corresponding rule in [37, Section 6.6], saying, assuming $(\mathsf{T},\ A_0)$ is a strong CAP for U with the empty write set (as it should be), and (C,C') is a strong CAP of F:

   If f is such that it terminates starting from the current state when applied to any z satisfying A (hence any value which is better than U), then when U is applied to f, it terminates to turn the state into C, which is the minimal state so that F terminates hence it would indeed terminate to give the resulting state and value C'.

   The only, and essential, difference from the corresponding rule in [37, Section 6.6] is how to stipulate the "current state". For this purpose it uses a fresh (auxiliary) reference-typed vector variable and its content (again a vector variable), which, by the semantics of auxiliary names, can be instantiated into arbitrary vectors of reference name and its content. Since we can use the conjunction of pre/post conditions to accumulate all these variants, the rule can in fact says for an arbitrary current state with arbitrary number of reference names f should be such that it results in C when applied to z satisfying A.

Below we show the proof rules do derive strong CAPs for FCFs.

Proposition 8.16 *If $\vdash_{\mathsf{char}} \{C\}\ F :_u \{C'\}$, then (C,C') is a strong CAP of F at u.*

Proof Most cases are identical with those given in [37, Section 6.6/6.7]. We only show the added rules, the reference name and the corresponding case construct, and the let-application rule which is non-trivially refined (the assignment rule is by the same reasoning except the use of logical substitutions).

(Reference Name) Assume given $\Gamma \vdash x : \mathsf{Ref}(\alpha)$. We show, for fresh r and i, $!r = i$ and $u = x \wedge\ !r = i$ is its strong CAP at u. First, $!r = i$ is an MTC of x because, since x is already a value, $(x\xi,\sigma) \Downarrow$ iff (ξ,σ) is well-typed iff $(\xi,\sigma) \models \mathsf{T}$, but $\exists r,i.(!r = i) \equiv \mathsf{T}$

hence done. For closure, we reason, with $\mathsf{rdom}(\xi) = \{\tilde{r}\}$ (where $\mathsf{rdom}(\xi)$ extracts the reference typed-names from the domain of ξ) and letting r' and i' be fresh:

$$\begin{array}{ll} \models \{E \wedge\ !r' = i'\}\ M :_u \{u = x \wedge\ !r' = i'\} & \\ \quad \Rightarrow \quad \models \{E \wedge\ !\tilde{r} = \tilde{i}\}\ M :_u \{u = x \wedge\ !\tilde{r} = \tilde{i}\} & \\ \quad \Rightarrow \quad (\xi \cdot u : M,\ \sigma) \Downarrow (\xi \cdot u : \mathbf{i},\ \sigma) \models u = x \wedge\ !\tilde{r} = \tilde{i} & \end{array}$$

Since this holds for an arbitrary distinction, we know $M \cong x$.

(Case-ReferenceName) Let, for brevity,

$$F' \stackrel{\text{def}}{=} \mathtt{case}\ x\ \mathtt{of}\ \langle \mathtt{n_i} : F_i \rangle_i, \quad C \stackrel{\text{def}}{=} \vee_i (x = y_i \wedge C_i), \quad C' \stackrel{\text{def}}{=} \vee_i (x = y_i \wedge C_i').$$

We show (C, C') is a strong CAP of F' at u, under the assumption that, by (IH), (C_i, C_i') is a strong CAP of F_i at u for each i. For simplicity and without loss of generality we stipulate auxiliary names of each C_i is given by $\tilde{j}$. Let the assumed basis be $\Gamma, x : \mathsf{Ref}\,(\alpha)$ and ξ' a semi-closing substitution for this basis, and that $\xi = \xi'/x$. For MTC we reason:

$$\begin{array}{lll} (F'\xi', \sigma) \Downarrow & \Leftrightarrow & \vee_i (\ \xi'(x) = y_i\ \wedge\ (F\xi, \sigma) \Downarrow) \\ & \Leftrightarrow & \vee_i (\ \xi'(x) = y_i\ \wedge\ \xi \models \exists \tilde{j}.C_i\ ) \\ & \Leftrightarrow & (\xi', \sigma) \models \exists \tilde{j}.C. \end{array}$$

For the closure condition, we reason, again setting $\xi = \xi'/x$:

$$\begin{array}{lll} \models \{C \wedge E\}\ M :_u \{C'\}\ \wedge\ (\xi',\ \sigma) \models C \wedge E & \Rightarrow & \exists i.(\ (M[y_i/x]\xi\ \sqsubseteq\ M\xi')\ ) \\ & \Rightarrow & \exists i.(\ F_i\xi\ \sqsubseteq\ M[y_i/x]\xi\ \sqsubseteq\ M\xi')\ ) \\ & \Rightarrow & F'\xi'\ \sqsubseteq\ M\xi' \end{array}$$

In the second line above we used, for each i:

$$\begin{array}{lll} \models \{C \wedge E\}\ M :_u \{C'\} & \Rightarrow & \models \{C_i \wedge E\}\ M[y_i/x] :_u\ \{C_i'\} \\ & \Rightarrow & \xi \models C_i \wedge E\ \supset\ F_i\xi\ \sqsubseteq\ M[y_i/x]\xi \end{array}$$

whose last entailment comes from (IH) for F' and (C_i, C_i').

(Let-Application) We now treat the let-application. This case is most complicated since we should work with a variable of a function type which, by definition, cannot be given a precise domain for write effects. More concretely, 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 8.17 *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 $U^{\Gamma;\alpha}$, $F^{\Gamma\cdot x;\beta;\gamma}$, names r' and j' be chosen fresh, and $\tilde{y}$ be non-reference names in $\mathsf{dom}(\Gamma)$. We set, with $\tilde{r} = r_1..r_i..r_n$ $(n \geq 0)$:

$$
\begin{array}{rcll}
F' & \stackrel{\text{def}}{=} & \mathtt{let}\, x = fU\ \mathtt{in}\, F & (8.1)\\
\xi_0 & = & \tilde{y} : \tilde{S} & (8.2)\\
\xi_1 & = & \mathfrak{a} : \tilde{r}, \mathfrak{b} : \tilde{V} & (8.3)\\
\xi & = & \xi_0 \cdot f : W & (8.4)\\
\sigma & = & \tilde{r} \mapsto \tilde{V} & (8.5)\\
C_0 & \stackrel{\text{def}}{=} & \forall \mathfrak{a}, \mathfrak{b}.\ (\,(!\mathfrak{a} = \mathfrak{b} \wedge \mathit{Max}(\mathfrak{a}))\ \supset\ \forall z.\{A \wedge\, !\mathfrak{a} = \mathfrak{b}\}\, f \bullet z \searrow x\{C\}\,) & (8.6)
\end{array}
$$

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

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

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

$$\xi_0 \cdot z : U\xi_0 \models A \tag{8.7}$$

Further we observe $(\xi, \sigma) \models \mathit{Max}(\mathfrak{a}^{\mathsf{Ref}\,(\vec{\mathrm{X}})}) \quad \Rightarrow \quad \{[\![\mathfrak{a}]\!]\} = \mathsf{dom}(\sigma)$. 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 $(\xi, \sigma) \models C_0$ implies $(F'\xi, \sigma) \Downarrow$.

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

as required. Next we show $(F'\xi,\ \sigma) \Downarrow$ implies $(\xi, \sigma) \models C_0$.

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

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 $\mathit{Max}(\mathfrak{a})$ hold, (apart from how elements are permuted and duplicated) the content of $\mathfrak{a}$ and $\mathfrak{b}$ are invariant.

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

$$\{C_0 \wedge E\}M :_u \{C'\}. \tag{8.8}$$

Let $\mathsf{dom}(\Delta) = \mathsf{dom}(\sigma) = \{\tilde{r}\}$ (the effect of aliasing can be ignored in the following arguments). Following [37, Section 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.

$$M_0 \quad \stackrel{\text{def}}{=} \quad \mathtt{let}\ \tilde{z} = !\tilde{r}\ \mathtt{in}\ \mathtt{let}\ x = yU\ \mathtt{in}\ (\tilde{r} := \tilde{z}\ ;\ M) \tag{8.9}$$

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: $(\xi,\ \sigma) \models C_0 \wedge E$. By (8.9) we have:

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

where the mixing of reduction and satisfiability should be easily understood. As the above reduction indicates, we can check:

$$\{C\}\,(\tilde{r} := \sigma(\tilde{r}); M)\xi,\ \sigma_0) :_u \{C'\} \tag{8.10}$$

By $\models \{C\}F :_u \{C'\}$ we have:

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

By (IH1) and (8.10) this means $(\xi{\cdot}u : V'',\ \sigma'') \sqsubseteq (\xi{\cdot}u : V',\ \sigma')$, as required. □

We conclude this section with the establishment of observational completeness. We first define the standard logical equivalence, cf. [23].

Definition 8.18 (logical equivalence) Write $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$ when $\models \{C\}M_1^{\Gamma;\Delta\alpha} :_u \{C'\}$ iff $\models \{C\}M_2^{\Gamma;\Delta;\alpha} :_u \{C'\}$.

Note we do not restrict the class of formulae to TCAs. The main result follows.

Theorem 8.19 *Let* $\Gamma;\Delta \vdash M_{1,2} : \alpha$. *Then* $\Gamma;\Delta \vdash M_1 \cong M_2 : \alpha$ *iff* $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$.

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 8.15, there exist semi-closed FCF values F and $\tilde{U}$ such that, say,

$$(FM_1, \tilde{r} \mapsto \tilde{U}) \Downarrow \qquad \text{and} \qquad (FM_2, \tilde{r} \mapsto \tilde{U}) \Uparrow . \tag{8.11}$$

By Proposition 8.16, there are assertions which characterise F and $\tilde{U}$ (in the sense of Definition 8.11). Let the characteristic formula for F at f be written $[\![F]\!](f)$. We now reason:

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

But by (8.11) we have

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

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

Fig. 15 Asserted Questionable Double.

$u : \{\mathsf{T}\}$
`\lambda(x,y).`
$\{!x = i \wedge !y = j \wedge x \neq y\}$
`x := !x + !x;`
$\{!x = 2 \cdot i \wedge !y = j \wedge x \neq y\}$
`y := !y + !y;`
$\{!x = 2 \cdot i \wedge !y = 2j \wedge x \neq y\}$
`end`
$\{\forall x,y.\{!x = i \wedge !y = j \wedge x \neq y\} u \bullet (x,y) \{!x = 2 \cdot i \wedge !y = 2 \cdot j\}\}$

9 Reasoning Examples

This section revisits the programming examples discussed in Introduction and Section 4, presenting the derivation of their specifications. We start with two simple examples, double? and `swap`, followed by a reasoning about circular data. We conclude by proving the correctness of the higher-order quicksort. The following properties of semantic update are used:

(S0) $C\{\!|e'/!e|\!\} \equiv C$, assuming C is $!e$-free.
(S1) (a) $(e' = !e_1)\{\!|e''/!e_2|\!\} \equiv ((e_1 = e_2 \wedge e' = e'') \vee (e_1 \neq e_2 \wedge e' = !e_2))$ or, as its special instance, (b) $(e' = !e)\{\!|e''/!e|\!\} \equiv e' = e''$, in both cases assuming e and e' do not contain dereferences.

Both are immediate from axioms.

9.1 Questionable Double (1): Direct Reasoning

Recall that double? in an example which exhibits different behaviour under different distinctions. Let us reproduce the program.

$$\texttt{double?} \stackrel{\text{def}}{=} \lambda(x,y).(x := !x + !x; y := !y + !y)$$

We establish following judgement which says that, if we can assume two arguments it receives are distinct, then the program does double what are stored in them. For brevity, we write $2i$ for $2 \times i$.

$$\{\mathsf{T}\}\ \text{double?} :_u \ \{\ \forall x,y.\{x \neq y \wedge !x = i \wedge !y = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge !y = 2j\}\ \} \quad (9.1)$$

For inferring the judgement (9.1), we use the following two implications.

$$x \neq y \wedge\ !x = i \wedge\ !y = j \quad \supset \quad (x \neq y \wedge\ !x = 2i \wedge\ !y = j)\{\!|!x + !x/!x|\!\} \quad (9.2)$$
$$x \neq y \wedge\ !x = 2i \wedge\ !y = j \quad \supset \quad (!x = 2i \wedge\ !y = 2j)\{\!|!y + !y/!y|\!\} \quad (9.3)$$

We first establish (9.2) and (9.3). For the former:

$$\begin{array}{ll} & (x \neq y \wedge\ !x = 2i \wedge\ !y = j)\{\!|!x{+}!x/!x|\!\} \\ \equiv & x \neq y \wedge\ !x = 2i\{\!|!x{+}!x/!x|\!\} \wedge\ !y = j\{\!|!x{+}!x/!x|\!\} \\ \equiv & x \neq y \wedge\ !x{+}!x = 2i\ \wedge\ (x \neq y \supset !y = j) \\ \subset & x \neq y \wedge\ !x = i \wedge\ !y = j \end{array}$$

The reasoning for (9.3) is identical hence omitted. We can now present the inference. We use [*AssignVar*] discussed already, as well as the obvious extension of [*Abs*] to cater for a vector of names, also called [*Abs*].

1.	$x \neq y \wedge\ !x = i \wedge\ !y = j \quad \supset \quad (x \neq y \wedge\ !x = 2i \wedge\ !y = j)\{\!|!x{+}!x/!x|\!\}$	(9.2)
2.	$\{(x \neq y \wedge !x = 2i \wedge\ !y = j)\{\!|!x{+}!x/!x|\!\}\}\ x := !x{+}!x\ \{x \neq y \wedge !x = 2i \wedge\ !y = j\}$	(AssignVar)
3.	$\{x \neq y \wedge !x = i \wedge\ !y = j\}\ x := !x{+}!x\ \{x \neq y \wedge !x = 2i \wedge\ !y = j\}$	(1, 2, Consequence)
4.	$x \neq y \wedge\ !x = 2i \wedge\ !y = j \quad \supset \quad (!x{=}2i \wedge !y{=}2j)\{\!|!y{+}!y/!y|\!\}$	(9.3)
5.	$\{(!x{=}2i \wedge !y{=}2j)\{\!|!y{+}!y/!y|\!\}\}\ y := !y{+}!y\ \{!x{=}2i \wedge !y{=}2j\}$	(AssignVar)
6.	$\{x \neq y \wedge !x = 2i \wedge\ !y = j\}\ y := !y{+}!y\ \{!x{=}2i \wedge !y{=}2j\}$	(4, 5, Consequence)
7.	$\{x \neq y \wedge !x = i \wedge\ !y = j\}\ x := !x{+}!x\ ;\ y := !y{+}!y\ \{!x{=}2i \wedge !y{=}2j\}$	(AssignVar)
8.	$\{\mathsf{T}\}\ \mathsf{double?} :_u\ \{\ \forall x,y.\{x \neq y \wedge\ !x = i \wedge\ !x = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge\ !x = 2j\}\ \}$	(Abs)

The structure of the reasoning is summarised in the asserted version of the code in Figure 15, where the anchor is placed at the top for readability (the anchor for a unit type is omitted as in judgements).

The derivation for (9.1) above suggests how we can refine this program so that it is robust with respect to aliasing. This is done by "internalising" the condition $x \neq y$ as follows.

$$\mathsf{double!} \stackrel{\text{def}}{=} \lambda(x,y).(\texttt{if}\ x \neq y\ \texttt{then}\ x{:=}!x{+}!x; y{:=}!y{+}!y\ \texttt{else}\ x{:=}!x{+}!x) \qquad (9.4)$$

We now infer:

$$\{\mathsf{T}\}\ \mathsf{double!} :_u\ \{\ \forall x,y.\{!x = i \wedge\ !x = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge\ !x = 2j\}\ \} \qquad (9.5)$$

Note the judgement indicates double! is robust with respect to aliasing — it satisfies the required functional property without stipulating anything about possible aliasing of arguments. The inference follows, using the first few lines of the previous inference. Below in Line 11 we set $M_1 \stackrel{\text{def}}{=} x{:=}!x{+}!x; y{:=}!y{+}!y$ and $M_2 \stackrel{\text{def}}{=} x{:=}!x{+}!x$.

$1-7.$	(As above).	
8.	$x = y \wedge\ !x = i \wedge\ !y = j \quad \supset \quad (!x = 2i \wedge\ !y = 2j)\{\!|!x{+}!x/!x|\!\}$	
9.	$(!x = 2i \wedge\ !y = 2j)\{\!|!x{+}!x/!x|\!\}\ x := !x{+}!x\ \{!x = 2i \wedge\ !y = 2j\}$	(AssignVar)
10.	$\{x = y \wedge\ !x = i \wedge\ !y = j\}\ x := !x{+}!x\ \{!x = 2i \wedge\ !y = 2j\}$	(1, 2, Consequence)
11.	$\{!x = i \wedge\ !y = j\}\ \texttt{if}\ x \neq y\ \texttt{then}\ M_1\ \texttt{else}\ M_2\ \{!x = 2i \wedge\ !y = 2j\}$	(7, 11, If)
12.	$\{\mathsf{T}\}\ \mathsf{double!} :_u\ \{\ \forall x,y.\{!x = i \wedge\ !x = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge\ !x = 2j\}\ \}$	

We leave the calculation for Line 8 to the reader.

Fig. 16 Asserted Questionable Double (located version).

$u : \{\mathsf{T}\}$
\lambda(x,y).
$\{!x = i \wedge !y = j \wedge x \neq y\}$
$\{!x = i \wedge [!x]\, !y = j\}$

$\{!x = i\}$
x := !x + !x;
$\{!x = 2 \cdot i\} @ x$
$\{!y = j\}$
y := !y + !y;
$\{!y = 2 \cdot j\} @ y$

$\{\langle !y \rangle\, !x = 2 \cdot i \wedge !y = 2 \cdot j\} @ xy$
$\{!x = 2 \cdot i \wedge !y = 2 \cdot j\} @ xy$
end
$\{\forall x,y.\{!x = i \wedge !y = j \wedge x \neq y\} u \bullet (x,y) \{!x = 2 \cdot i \wedge !y = 2 \cdot j\} @ xy\}$

9.2 Questionable Double (2): Located Reasoning

We have seen, in Section 5.5, that we can use a located assertion to obtain a more "extensional" specification for the questionable double. In this case we wish to derive:

$$\{\mathsf{T}\}\,\mathsf{double?} :_u \{\forall x,y.\ (\{x \neq y \wedge !x = i \wedge !y = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge !y = 2j\} @ xy\} @ \emptyset$$

In the following proof, we derive the above judgement using a fully extensional judgement for each subpart of the program. For combining two assignment commands, we use [*Seq-eoi*] in Figure 11.

1. $\{!x = 2i\}\ x := !x + !x\ \{!x = 2i\} @ x$	(AssignVar)
2. $\{!y = j\}\ y := !y + !y\ \{!y = 2j\} @ y$	(AssignVar)
3. $\{!x = i \wedge [!x]\, !y = j\}\ x := !x + !x\ ;\ y := !y + !y\ \{\langle !y \rangle\, !x = 2i \wedge !y = 2j\}$	(Seq-eoi)
4. $\{x \neq y \wedge !x = i \wedge !y = j\}\ x := !x + !x\ ;\ y := !y + !y\ \{(x \neq y \supset !x = 2i) \wedge !y = 2j\} @ x,y$	(Conseq)
5. $\{x \neq y \wedge !x = i \wedge !y = j\}\ x := !x + !x\ ;\ y := !y + !y\ \{!x = 2i \wedge !y = 2j\} @ x,y$	(Invariance)
6. $\{\mathsf{T}\}\,\mathsf{double?} :_u \{\forall x,y.\ (\{x \neq y \wedge !x = i \wedge !x = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge !x = 2j\} @ xy)\ \} @ \emptyset$	(Abs)

In Line 5 we add $x \neq y$ to pre/post conditions.

The asserted version of the code is given in Figure 16 (the dotted lines signify the use of the EOI rule to combine two located assertions). As can be seen, the usage of the EOI rule may be considered as a semantic strengthening of the idea of "local reasoning" advocated in separation logic [61, 66].

Fig. 17 Asserted Swap.

$u : \{\mathsf{T}\}$
\lambda(x, y).
  $\{!x = i \wedge !y = j\}$
  let z=x in
    $\{!x = i \wedge !y = j \wedge z = i\}$
    x:=!y;
    $\{!x = j \wedge !y = j \wedge z = i \wedge (x{=}y \supset i{=}j)\} \mathbin{@} x$
    y:=z;
    $\{!x = j \wedge !y = i\} \mathbin{@} y$
  end
  $\{!x = j \wedge !y = i\} \mathbin{@} xy$
end
$\{\forall x,y\{!x = i \wedge !y = j\}u \bullet (x,y)\{!x = j \wedge !y = i\} \mathbin{@} xy\}$

9.3 Swap

Judgements. Next we verify `swap`, a program that exchanges the content of two reference cells mentioned in Introduction. We reproduce its code below.

$$\texttt{swap} \stackrel{\text{def}}{=} \lambda(x,y).\texttt{let}\ z = !x\ \texttt{in}\ (\ x := !y\ ;\ y := z\ )$$

Let us also set (taking the located version of its specification):

$$\mathsf{Swap}(u) \stackrel{\text{def}}{=} \forall x.\forall y.\{!x = i \wedge !y = j\}u \bullet \langle x,y\rangle\{!x = j \wedge !y = i\} \mathbin{@} xy$$

Using this predicate we wish to establish:

$$\{\mathsf{T}\}\ \texttt{swap} :_u \{\mathsf{Swap}(u)\}\ @\ \emptyset. \tag{9.6}$$

Swap is the classical example treated in many preceding work (cf. [10, 11, 44]). An interesting point is that, while the specification does not mention aliasing, its derivation does have to deal with it, by being split in two parts: one dealing with the case when x and y are alias; and the other with the case when they are distinct. Informally:

1. If x and y are distinct, then two assignments, $x := !y$ and $y := z$, are independent (in the sense that they do not affect each other). Since z does hold the initial content of x, we know these two assignments swap the content of x and y.
2. On the other hand, if x and y are aliased, two assignments, $x := !y$ and $y := z$, in fact affect the same memory cell: but $y := z$ in fact does not change the content of y given z denotes the initial value of x (hence of y), so that these two assignments again perform the (vacuous) swapping of content.

The above observation indicates there is *semantic independence* between two assignment commands, in the sense that their *operational* collision in the case of aliasing does not affect the demanded postcondition.

Located Reasoning. The semantic independence of `swap` is fully exploited using the [*SeqI*]. Let $A \stackrel{\text{def}}{=} x = y \supset i = j$ below. Note A is stateless

1. $\{!y = j\}\ x :=!y\ \{!x = j\} \mathbin{@} x$	(AssignS)
2. $\{z = i\}\ y := z\ \{!y = i\} \mathbin{@} y$	(AssignS)
3. $\{!y = j \wedge [!x]z = i\}\ x :=!y\ ;\ y := z\ \{\langle !y\rangle\, !x = j \wedge\ !y = i\} \mathbin{@} xy$	(1, 2, SeqI)
4. $\{!x = i \wedge !y = j \wedge z = i\}\ x :=!y\ ;\ y := z\ \{(x \neq y \supset !x = j) \wedge\ !y = i\} \mathbin{@} xy$	(3, Conseq)
5. $\{A \wedge !x = i \wedge !y = j \wedge z = i\}\ x :=!y\ ;\ y := z\ \{A \wedge (x \neq y \supset !x = j) \wedge\ !y = i\} \mathbin{@} xy$	(4, Invariance)
6. $\{!x = i \wedge !y = j \wedge z = i\}\ x :=!y\ ;\ y := z\ \{x = j \wedge\ !y = i\} \mathbin{@} xy$	(5, Consequence)
7. $\{!x = i \wedge !y = j\}\ !x :_z \{!x = i \wedge !y = j \wedge z = i\} \mathbin{@} \emptyset$	(Deref)
8. $\{!x = i \wedge !y = j\}\ \mathtt{let}\ z = !x\ \mathtt{in}\ \ (x :=!y\ ;\ y := z)\ \ \{!x = j \wedge !y = i\} \mathbin{@} xy$	(6, 7, Let)
9. $\{\mathsf{T}\}\ \mathtt{swap} :_u \{\mathsf{Swap}(u)\} \mathbin{@} \emptyset$	(8, Abs)

In Line 6, we used $!x = i \wedge !y = i$ entails A. The rest is immediate.

Reasoning based on Traditional Method. For contrast, we also present a derivation of the same specification using the traditional method a la Morris/Cartwright-Oppen (recaptured in the present framework).

1. $\{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}\}\ x :=!y\ \{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\} \mathbin{@} x$	(AssignS)
2. $\{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\}\ y := z\ \{!x = j \wedge !y = i\} \mathbin{@} y$	(AssignS)
3. $\{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}\}\ x :=!y\ ;\ y := z\ \{!x = j \wedge !y = i\} \mathbin{@} xy$	(1, 2, Seq)
4. $(!x = i \wedge !y = j \wedge z = i)\ \supset\ (!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}$	$(\star\star\star)$
5. $\{!x = i \wedge !y = j \wedge z = i\}\ x :=!y\ ;\ y := z\ \{!x = j \wedge !y = i\} \mathbin{@} xy$	(3, 4, Conseq)
6. $\{!x = i \wedge !y = j\}\ !x :_z \{!x = i \wedge !y = j \wedge z = i\} \mathbin{@} \emptyset$	(Deref)
7. $\{!x = i \wedge !y = j\}\ \mathtt{let}\ z = !x\ \mathtt{in}\ \ (x :=!y\ ;\ y := z)\ \ \{!x = j \wedge !y = i\} \mathbin{@} xy$	(5, 6, Let)
8. $\{\mathsf{T}\}\ \mathtt{swap} :_u \{\mathsf{Swap}(u)\} \mathbin{@} \emptyset$	(7, Abs)

Except Line 4, all inferences are direct from the proof rules. Below we derive $(\star\star\star)$, starting from the consequence and reaching the antecedent.

$(!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}$	
$\equiv (!x = j)\{\!|z/!y|\!\}\{\!|!y/!x|\!\} \wedge (!y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}$	(Pro.8.1 (2))
$\equiv ((x = y \supset z = j)\ \wedge\ (x \neq y \supset !x = j))\{\!|!y/!x|\!\}\ \wedge\ (z = i)\{\!|!y/!x|\!\}$	$(S1)$
$\equiv (x = y \supset z = j)\{\!|!y/!x|\!\}\ \wedge\ (x \neq y \supset !x = j)\{\!|!y/!x|\!\}\ \wedge\ (z = i)\{\!|!y/!x|\!\}$	(Pro.8.1 (2))
$\equiv (x = y \supset z = j)\ \wedge\ (x \neq y \supset (!x = j\{\!|!y/!x|\!\})\ \wedge\ z = i$	(L7)
$\equiv (x = y \supset z = j)\ \wedge\ (x \neq y \supset !y = j)\ \wedge\ z = i$	(S1)
$\subset !x = i \wedge !y = j\ \wedge\ z = i$	(fol)

While the traditional reasoning gives a shorter compositional reasoning, it involves non-trivial inferences at the assertion level. This is because the traditional method (or the separation-based method a la Burstall) cannot exploit semantic independence between two assignments, which [*SeqI*] can capture.

9.4 Circular References

We next show the reasoning for $x := !!x$, the example which appeared in Section 5 that uses circular data structures. Note typing of such a data structure needs recursive types, which we briefly outline first. We take the equi-isomorphic approach [63] where recursively defined types are equated iff their representation as regular trees are isomorphic. The grammar of types is extended as follows, for both the programming language and for the assertion language.

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

The typing rules do not change except we now take types up to the tree isomorphisms. Accordingly no change is needed in the axioms and proof rules (one possible, but not necessary, option is to introduce a recursively defined assertion).

Reproducing the assertion in Section 5, we wish ot prove the following judgement.

$$\{!x = y \wedge !y = x\}\ x := !!x\ \{!x = x\}.$$

For the proof we start by converting the pre-condition into a form that is usable by [*AssignVar*].

$$\begin{array}{lll}
!x = y \wedge !y = x & \supset & !!x = x(m = x)[!!x/m] \\
& \supset & (m = x)\{\!|!!x/m|\!\} \qquad \text{(Lemma 6.7.5)} \\
& \supset & (!x = x)[m/!x]\{\!|!!x/m|\!\} \\
& \supset & (!x = x)\{\!|m/!x|\!\}\{\!|!!x/m|\!\} \qquad \text{(Lemma 6.7.6)} \\
& \supset & (!x = x)\{\!|!!x/!x|\!\}
\end{array}$$

From here it is easy to get:

1. $(!x = y \wedge !y = x) \supset ((!x = x)\{\!|!!x/!x|\!\})$

---

2. $\{(!x = x)\{\!|!!x/!x|\!\}\}\ x := !!x\ \{!x = x\}$ (AssignSimple)

---

3. $\{(!x = x)\{\!|!!x/!x|\!\}\}\ x := !!x\ \{!x = x\}$ (1, 2, Consequence)

Similarly we can easily derive:

$$\{!y = x\}\ x := \langle 1, \mathtt{inr}(!y)\rangle\{!x = \langle 1, \mathtt{inr}(x)\rangle\}$$

where x is of type, for example, $\mathsf{Ref}\,(\mu X.(\mathsf{Nat} \times (\mathsf{Unit} + X)))$ (instead of sum types, one may as well use a null pointer as a terminator of recursive data types, as is standard in many languages: in this case the assertion language also incorporates the null pointer as a constant for each reference type). The assertion $!x = \langle 1, \mathtt{inr}(x)\rangle$ says x stores a pair of 1 and (an injection with) a reference to itself, precisely capturing graphical structure of the datum.

9.5 Quicksort (1): Informal Illustration

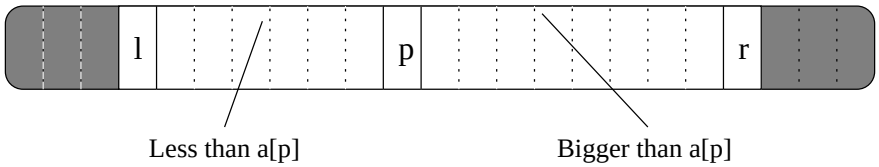
Hoare's Quicksort [12, 25–28, 42] is a terse and efficient algorithm for sorting an array using recursion. Apart from recursive calls to itself, the algorithm calls Partition, another procedure which permutes elements of an array so that they are divided into two contiguous parts, the left containing elements lesser than a "pivot value" *pv* and the right those greater than *pv*. The pivot value *pv* is one of the array elements which may ideally be their mean value (this depends on the choice of a partition algorithm). There are many variants, most notably how the pivot value is determined and how the partition algorithm is organised. In the following we give one of the simplest forms of this algorithm and derive its full specification.

Overall Structure. First we offer an informal illustration of the version of the quicksort we shall use in the rest of the section (those readers who are familiar with the algorithm may as well skip the following illustration and jump to the next subsection). Assume given an array a of size n of some data type and indices $0 \leq l, r < n$. We want to sort the subarray of a delimited by l and r. Sorting happens with respect to a given total order.[6]

Quicksort's overall structure is this (indentation is used for scoping):

```
if l < r then
        let p = partition(a, l, r) in
                quicksort(a, l, p-1);
                quicksort(a, p+1, r)
```

The algorithm works by splitting the subarray to be sorted into two parts, using `partition`, such that every element in the left part is less than every member of the right part, as shown in the next figure.



This is done by calling `partition`. Its return value p is called the *pivot point*, so that $a[p]$ is the pivot value. Thus, as a result of executing the partition, $a[l..p-1]$ are lesser than $a[p]$ while $a[p+1..r]$ are greater than $a[p]$. We then sort these two subarrays by recursive calls to Quicksort which again performs the partition and two recursive calls as far as $l < p < r$. Since the range (either between l and p or between p and r) strictly decreases at each call, at some point either l or r coincides with p, so that the call at that time returns without any further calls. Thus the algorithm terminates and, as a result, the elements in $a[l..r]$ are now fully sorted.

[6] We in fact only require the relation is transitive and total, i.e. orders two distinct elements in one way or the other. In the final program, such an ordering is realised by a procedure passed as an argument. In this subsection it suffices to regard "less than", "bigger than" etc. as some informal notion of ordering on a given data type which may or may not be a strict ordering. On integers, we assume $<$ means "strictly less than" while $\leq$ "no more than".

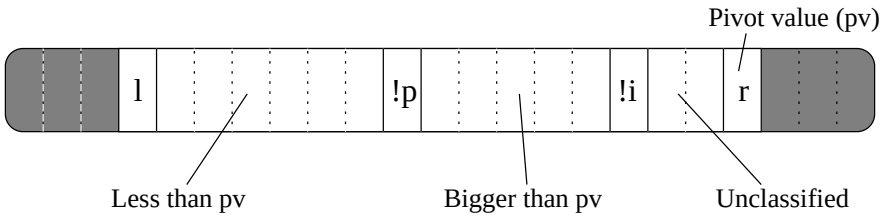
Partition. Partitioning is at the heart of Quicksort and it can be done in various ways. Our partitioning procedure is simple in that it uses only one loop. The code is given below (we start from Line 2 for consistency with the line numbering for the algorithm we shall use for verification given in Figure 20).

```
2       let pv = !a[r] in
3               p := l
4               i := l
5               while !i < r
6                       if !a[!i] < pv then
7                               swap( a[!p], a[!i] )
8                               p := !p + 1
9                       i := !i + 1
10              swap(a[r], a[!p]);
11              !p
```

Line 2 determines the pivot value relative to which the array is partitioned. Any array index from l to r may in principle be used and we choose r, although other choices may at times be advantageous: for example randomising this choice yields a better expected performance. Next we scan through the entire subarray starting at l, the left end, going right. It maintains two pointers i and p. Both are initialised with l. How these variables (l, r, p and i) are related is illustrated below.

- $!i$ points to the current element.
- All array entries of a starting from l and reaching, but not including, $!p$ are less than the pivot value pv.
- All array elements of a starting from $!p$ going right to (but not including) $!i$ is no less than pv.
- Right of $!i$ and left of r we have what is left to be done.
- Finally, as far as $!p < !i$, it is guaranteed that the content of $a[!p]$ is equal or greater than pv.

The overall situation is depicted in the next picture.



This property is maintained at each iteration of the loop at its entry point, i.e. it is a loop invariant. Let's look at the loop body to see how this invariant is maintained.

```
6               if !a[!i] < pv then
7                       swap( a[!p], a[!i] )
8                       p := !p + 1
9               i := !i + 1
```

Line 6 compares the current element $!a[!i]$, i.e. the leftmost element we have not yet considered, with the pivot value. If it is less, then we execute the next two lines. First we swap the content of $a[!i]$ with $a[!p]$ (where we assume `swap` is a program which swaps content of two references passed as arguments and does nothing else, as we have seen in Section 9.3). We inspect the situation immediately after this swapping in the following.

- If $!p = !i$, then swapping them has no effect, so that we have the same condition as before (note in this case we only demand everything before $!p = !i$ is smaller than pv).
- If $!p < !i$, then (1) everything to the left of $!p+1$ (which will become the new value of p) is less than the pivot value, because what we've just swapped with $a[!p]$ is smaller than pv and everything else is unchanged; and (2) the element at $a[!i]$ is at least as big as pv because it originally was before swapping.

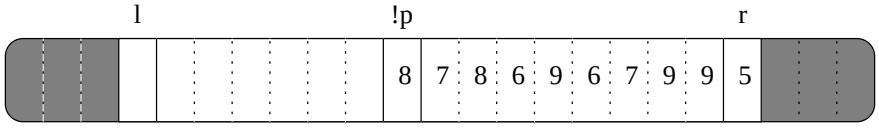
After the swap is executed, Line 8 increments p by one, which is the end of the conditional body. On the left of the new $a[!p]$, we have all the elements which were left of the *original* $a[!p]$, which is now $a[!p-1]$, plus this $a[!p-1]$ itself. The former elements are smaller than pv by the original invariance: The latter is smaller than pv since its content is what used to be $a[!i]$ which, by the guard of the conditional, we already know is smaller than pv.

Finally, regardless of the execution of the conditional statement, Line 9, the last of the loop body, increments i. Again there are two cases.

- If the conditional is executed, i.e. if originally it was $a[!i] < pv$, then the left element of new $a[!i]$ (after incrementing i) is surely bigger than pv by swapping.
- If not, then originally $a[!i] \geq pv$, which directly tells us that the left element of new $a[!i]$ is no less than pv, keeping the invariance.

This concludes the inspection of the loop invariant.

When the loop terminates, the invariant guarantees that everything from $!p$ to, but not including, $!i = r$ is no less than pv. We also recall pv is stored at $a[r]$. But the following situation is still possible.



We rectify this issue in Line 10 by swapping $a[r]$ with $a[!p]$, which leads to the desired partioning, since if $p = i = r$ then this changes nothing, if not we have $!a[!p] \geq pv$ before swapping by the invariant. Now at $!p$ the array contains pv, and each element to its left is smaller than the pivot value $a[!p]$ and to the right bigger (no less) than $a[!p]$, so the partition is complete. Finally, in Line 11, the program returns $!p$, which is the index of the pivot element.

This concludes the introduction to the quicksort algorithm we shall verify. The informal illustration using pictures may contribute to a full operational understanding of the algorithm upon which a formal reasoning should be built.

Fig. 18 Quicksort with a comparison procedure as a parameter.

```
1   μq. λ(a,c,l,r).
2         if l < r then
3               let p' = partition(a, c, l, r) in
4                     q(a, c, l, p'-1);
5                     q(a, c, p'+1, r)
```

9.6 Quicksort (2): Code and Specification.

Code. Figure 18 lists the program. It is identical with the one given in Section 9.5 except a comparison function `c` is passed as an argument to the program, about which we make no assumptions other than that it implements a total order in a loose sense (detailed later, cf. footnote 6), and it is purely functional, i.e. has no observable side effects. The program uses a partition algorithm (given in Figure 20 later).

Specification. We now present the formal specification of Quicksort. One of our goals is to obtain a specification that is as strong as possible in the sense that any program satisfying this specification should be contextually indistinguishable from `qsort`, cf. Section 8. To achieve this goal, the specification must be precise not only about what the program does but also about what it doesn't do (since if not a program with different side effects at some variable may satisfy the same specification). This means we must also specify references that remain unchanged. For making precise the set of locations modified by `qsort` we use the located assertions from Section 7.4. This choice also simplifies the proof significantly. The judgement follows.[7]

$$\{\mathsf{T}\}\ \texttt{qsort} :_u \{\mathsf{Qsort}(u)\} @ \emptyset. \tag{9.7}$$

where we set, omitting types:

$$\mathsf{Qsort}(u) \stackrel{\text{def}}{=} \forall abclr.\{\mathsf{Eq}(ablr) \wedge \mathsf{Order}(c)\}\, u \bullet (a,c,l,r)\,\{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\} @ a[l...r]ip \tag{9.8}$$

The expression $a[l...r]ip$ is short for $a[l],...,a[r],i,p$. The variable b is auxiliary and is of the same type as a, used for specifying that the change of a after the execution is only in the ordering of its elements. Informally the assertion says:

> *Let b be a distinct array coinciding with a at from l to r (where l and r should be within the array bound). Further assume c calculates a total order. Then, w.r.t. that order, calling q with a, c, l and r converges so that the elements of a from l to r still remain a permutation of the corresponding elements of b and are, in addition, well-sorted, touching only $a[l..r]$ and global variables i* and *p*.

[7] We assume `partition` and `swap` are inlined. For treating them as external procedures, we have only to add assertions for them (given later) in the precondition of (9.7).

Note that, to be able to say that the result of running Quicksort on an array a is a sorted version of that array, we need to have access to the original array in the postcondition. This is the role of the second array b which stores exactly the same data as a, but is not affected by running the algorithm. As the reading above suggests, each predicate used in (9.8) for defining $\mathsf{Qsort}(q)$ has the following meaning:

- $\mathsf{Eq}(ablr)$ says a and b coincide in the range from l to r (which should be in array bounds). In addition, it also stipulates a and b are of the same length and do not overlap with each other nor with the auxiliary variables p and i (which themselves should be distinct).
- $\mathsf{Order}(c)$ says c implements a total transitive relation.
- $\mathsf{Perm}(ablr)$ says that entries of a and b in the range from l to r are permutations of each other in the sense that one array can be obtained from the other by a finite number of binary permutations. We also stipulate the same distinctness condition as $\mathsf{Eq}(ablr)$.
- $\mathsf{Sorted}(alrc)$ says that the elements of a in the range from l to r are sorted w.r.t. the total order implemented by c.

We give the formal definition of each predicate below. First, the predicates $\mathsf{Eq}(ablr)$ and $\mathsf{Perm}(ablr)$ use distinctness condition on elements of a, those of b, p and i, which we write Dist (formally: define $\mathsf{Distinct}(e_1..e_n)$ by $\wedge_{1\le i\neq j\le n} e_i \neq e_j$: then we set $\mathsf{Dist} \stackrel{\text{def}}{=} \mathsf{Distinct}(a[0]..a[\mathsf{size}(a)-1]b[0]..b[\mathsf{size}(b)-1]pi)$). We then set:

$$\mathsf{Eq}(ablr) \stackrel{\text{def}}{=} 0 \le l,r \le \mathsf{size}(a) = \mathsf{size}(b) \wedge \forall j.(l \le j \le r \supset\ !a[j] = !b[j]) \wedge \mathsf{Dist}$$

$$\mathsf{Perm}(ablr) \stackrel{\text{def}}{=} 0 \le l,r \le \mathsf{size}(a) = \mathsf{size}(b) \wedge \exists n.\mathsf{Perm}^{(n)}(ablr) \wedge \mathsf{Dist}$$

where $\mathsf{Perm}^{(n)}(ablr)$ indicates a is the result of n-times permuting entries of b in the range from l to r, inductively defined from the single permutation predicate in the obvious way (Appendix B lists details for reference).

Next we define $\mathsf{Order}(c)$ formally.

$$\forall xy.\{\mathsf{T}\}c\bullet(x,y)\{\mathsf{T}\} \mathbin{@} \emptyset \quad\wedge\quad \forall xy.(x\neq y \supset (c\bullet(x,y)\searrow\mathsf{T} \vee c\bullet(y,x)\searrow\mathsf{T})) \wedge \forall xy.(c\bullet(x,y)\searrow\mathsf{T} \wedge c\bullet(y,z)\searrow\mathsf{T}) \supset c\bullet(x,z)\searrow\mathsf{T}$$

The definition allows both strict ordering and non-strict ordering: the partition algorithm, which uses c, works in both cases. When the data type is complex, $x \neq y$ can be hard to calculate, in which case it may as well be reflexive.[8]

Finally we define the predicate $\mathsf{Sorted}(aclr)$, which says a is sorted in the range from l to r w.r.t. the ordering induced by c.

$$\mathsf{Sorted}(aclr) \quad\stackrel{\text{def}}{\equiv}\quad \forall i,j.\ (l \le i < j \le r \supset\ c\bullet(!a[i],!a[j])\searrow\mathsf{T})$$

This completes the specification of `qsort`.

[8] Since c can never differentiate two $\cong$-equated data, if $x = y$ then $c(x,y)$ and $c(y,x)$ cannot differ.

Fig. 19 Asserted Quicksort.

$u : \{\mathsf{T}\}$
$\mu\,\mathbf{q}.$
  $m : \{\forall j \lesssim k.\mathsf{QsortBounded}(qj)\}$
  $\lambda(\mathtt{a,c,l,r})$
    $\{\mathsf{Eq}(ablr) \wedge \mathsf{Order}(c) \wedge r - l \leq k\}$
    `if l < r then`
      $\{\mathsf{Eq}(ablr) \wedge l < r\}$
      `let p' = partition(a, c, l, r) in`
        $\{\mathsf{Perm}(ablr) \wedge \mathsf{Parted}(aclrp') \wedge l < p' \leq r \wedge r - l \leq k\}$
        `q(a, c, l, p'-1);`
        `q(a, c, p'+1, r)`
        $\{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\}\,@\,a[l..r]pc$
      $\{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\}\,@\,a[l..r]pc$
    $\{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\}\,@\,a[l..r]pc$
  $\{\mathsf{QsortBounded}(mk)\}$
$\{\mathsf{Qsort}(u)\}$

9.7 Quicksort (3): Derivation.

Asserted Code and Auxiliary Predicates. We now establish the judgement (9.7). The overall structure of the inference is summarised in the asserted code in Figure 19. As in the previous asserted programs, the anchor for each block is given at the beginning of the block. The asserted code and the derivation given later both use two auxiliary predicates.

- $\mathsf{QsortBounded}(qj)$ with j of Nat type is used as an inductive hypothesis for recursion. It is the same as $\mathsf{Qsort}(q)$, given in (9.8), Page 79, except that it only works for a range no more than j and that it strengthens the assumptions so that the induction goes through. Formally $\mathsf{QsortBounded}(qj)$ is given by:

  $$\forall ablrc.\ \{\mathsf{Perm}(ablr) \wedge A\}\ u \bullet (a,c,l,r)\ \{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\}\,@\,a[l...r]ip$$

  where $A \stackrel{\text{def}}{=} \mathsf{Order}(c)\ \wedge\ r - l \leq j$. The predicate weakens $\mathsf{Eq}(ablr)$ in the precondition of (9.8) to $\mathsf{Perm}(ablr)$, thus strengthening the evaluation formula itself.
- $\mathsf{Parted}(aclrk)$ says the subarray of a from l to r is partitioned at an intermediate index k w.r.t. the order defined by c. Formally it is given as:

  $$l \leq k \leq r\ \wedge\ \forall j.(l \leq j \leq k \supset c \bullet (!a[j], !a[k]) \searrow \mathsf{T})\ \wedge\ \forall j.(k \leq j \leq r \supset c \bullet (!a[k], !a[j]) \searrow \mathsf{T})$$

  By transitivity, it entails $\forall l \leq j_1 < k < j_2 \leq r.\ c \bullet (!a[j_1], !a[j_2]) \searrow \mathsf{T}$. $\mathsf{Parted}(aclrk)$ describes the intermediate state immediately after `partition` is executed.

In addition, the asserted code in Figure 19 assumes that once a non-stateful predicate (e.g. $r - l \leq k$) is stipulated, we do not repeat it in the subsequent intermediate conditions since it continues to be valid anyway.

Derivation (1): Main Derivation. We first present the main derivation to reach the judgement (9.7), which precisely corresponds to the asserted code in Figure 19. Below for brevity we write $\texttt{qsort}'$ for the code given in Figure 18 with the initial line (which is the μ-abstraction followed by λ-abstraction) removed. In Lines 3 and 4, we set $B \stackrel{\text{def}}{=} \forall j < k.\mathsf{QsortBounded}(qj) \ \wedge \ l < r \ \wedge \ r - l \leq k$.

M.1. $\{\mathsf{Perm}(ablr) \wedge \mathsf{Order}(c)\}$
$\quad \texttt{partition}(a,c,l,r) :_{p'}$
$\{\mathsf{Parted}(aclrp') \wedge \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c)\} \mathbin{@} a[l..r]pi$

M.2. $\{\mathsf{Perm}(ablr) \wedge \mathsf{Order}(c) \wedge B\}$
$\quad \texttt{partition}(a,c,l,r) :_{p'}$
$\{\mathsf{Parted}(aclrp') \wedge \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c) \wedge B\} \mathbin{@} a[l..r]pi$ (M.1, Invariance)

M.3. $\{\mathsf{Perm}(ablr) \wedge \mathsf{Parted}(aclrp') \wedge \mathsf{Order}(c) \wedge B\}$
$\quad \texttt{q}(a,c,l,p'-1)\ ;\ \texttt{q}(a,c,p'+1,r)$
$\{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\} \mathbin{@} a[l...r]ip$

M.4. $\{\mathsf{Perm}(ablr) \ \wedge \ \mathsf{Order}(c) \ \wedge \ B\}$
$\quad \texttt{let}\ p' = \texttt{partition}(a,l,r,c)\ \texttt{in}\ (\texttt{q}(l,p'-1)\ ;\ \texttt{q}(p'+1,r))$
$\{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\} \mathbin{@} a[l...r]ip$ (M.2, M.3, Let)

M.5. $\{\mathsf{Perm}(ablr) \wedge \mathsf{Order}(c) \ \wedge \ \forall j < k.\mathsf{QsortBounded}(qj) \ \wedge \ r - l \leq k\}$
$\quad \texttt{qsort}'$
$\{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\} \mathbin{@} a[l...r]ip$ (M.4, IfThenSimple)

M.6. $\{\forall j < k.\mathsf{QsortBounded}(qj)\}$
$\quad \lambda(a,c,l,r).\texttt{qsort}' :_m$
$\{\mathsf{QsortBounded}(mk)\} \mathbin{@} \emptyset$ (M.5, Abs)

M.7. $\{\mathsf{T}\}\ \mu q.\lambda(a,c,l,r).\texttt{qsort}'\ :_u \{\forall k.\ \mathsf{QsortBounded}(uk)\} \mathbin{@} \emptyset$ (M.6, Rec)

M.8. $\{\mathsf{T}\}\ \texttt{qsort}\ :_u \{\mathsf{Qsort}(u)\} \mathbin{@} \emptyset$ (M.7, Consequence)

We observe:

- Line M.1 is the reasoning for the partioning function, and is discussed later (together with the code itself).
- Line M.2 is directly derivable from Line 1 and [*Invariance*] in Figure 8, noting B is stateless (i.e. it does not contain active dereferences).
- Line M.3 is verification for two consecutive recursive calls, which is one of the highlights of the reasoning. We shall treat this part shortly.
- Line M.4 is a direct application of [*Let*] in Figure 12; Line M.5 is a direct application of [*IfThenSimple*] in Figure 12; Line M.6 is a direct application of [*Abs*] (to be precise its multi-parameter version) in Figure 8; and Line M.7 is direct from Line 6 using [*Rec*] in Figure 8, noting $\mathsf{T} \wedge C \equiv C$ always (as here, applications of trivial logical equivalences in the standard first-order logic are henceforth ignored).

– Line M.8 is by the standard consequence rule together with the following implications. For brevity we set $C \stackrel{\text{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c)$, $C_0 \stackrel{\text{def}}{=} \mathsf{Eq}(ablr) \wedge \mathsf{Order}(c)$, and $C' \stackrel{\text{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)$.

$$
\begin{array}{lll}
& \forall k.\ \mathsf{QsortBounded}(uk) & \\
\stackrel{\text{def}}{=} & \forall k.\ \forall ablrc.\ \{C \wedge r-l \leq k\}\ u \bullet (a,c,l,r)\ \{C'\}\ @\, a[l..r]ip & \\
\equiv & \forall ablrc.\ \forall k.\ \{C \wedge r-l \leq k\}\ u \bullet (a,c,l,r)\ \{C'\}\ @\, a[l..r]ip & \\
\equiv & \forall ablrc.\ \{\exists k.(C \wedge r-l \leq k)\}\ u \bullet (a,c,l,r)\ \{C'\}\ @\, a[l..r]ip & (*) \\
\equiv & \forall ablrc.\ \{C \wedge \exists k.(r-l \leq k)\}\ u \bullet (a,c,l,r)\ \{C'\}\ @\, a[l..r]ip & \\
\equiv & \forall ablrc.\ \{C\}\ u \bullet (a,c,l,r)\ \{C'\}\ @\, a[l..r]ip & \\
\supset & \forall ablrc.\ \{C_0\}\ u \bullet (a,c,l,r)\ \{C'\}\ @\, a[l..r]ip & (**) \\
\stackrel{\text{def}}{=} & \mathsf{Qsort}(u) &
\end{array}
$$

where $(*)$ and $(**)$ are by (le7) and (le8) of Figure 4, respectively (for the latter note $C_0 \supset C$).

Thus we have only to establish the judgements in Line M.2 (partition) and Line M.3 (recursive calls). We start from the latter, then conclude with the inferences for the former.

Derivation (2): Recursive Calls. In the following we derive the judgement in M.3 (of the main derivation in Page 82), that is we shall establish:

$$\{C_1\}\ \ \mathsf{q}(a,c,l,p'-1)\ ;\ \mathsf{q}(a,c,p'+1,r)\ \ \{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\}\, @\, a[l...r]ip \tag{9.9}$$

where

$$C_1 \stackrel{\text{def}}{\equiv} \begin{pmatrix} \mathsf{Perm}(ablr)\ \wedge\ \mathsf{Parted}(aclrp')\ \wedge\ \mathsf{Order}(c)\ \wedge \\ \forall j<k.\mathsf{QsortBounded}(qj)\ \wedge\ r-l \leq k \end{pmatrix}.$$

For intermediate inferences, we also set, using fresh b':

$$
\begin{array}{ll}
C_2 \stackrel{\text{def}}{=} & \mathsf{Eq}(ab'l(p'-1)) \wedge \mathsf{Order}(c) \wedge \forall j<k.\mathsf{QsortBounded}(qj) \wedge p'-1-l<k \\
C_2' \stackrel{\text{def}}{=} & \mathsf{Perm}(ab'l(p'-1)) \wedge \mathsf{Sorted}(acl(p'-1)) \\
C_3 \stackrel{\text{def}}{=} & \mathsf{Eq}(ab'(p'+1)r) \wedge \mathsf{Order}(c) \wedge \forall j<k.\mathsf{QsortBounded}(qj) \wedge r-(p'+1)<k \\
C_3' \stackrel{\text{def}}{=} & \mathsf{Perm}(ab'(p'+1)r) \wedge \mathsf{Sorted}(ac(p'+1)r)
\end{array}
$$

These predicates satisfy:

$$C_2 \ \supset\ \{C_2\}\ q \bullet (a,c,l,p'-1)\ \{C_2'\}\ @\ a[l..p'-1]pi \tag{9.10}$$

$$C_3 \ \supset\ \{C_3\}\ q \bullet (a,c,p'+1,r)\ \{C_3'\}\ @\ a[p'+1..r]pi \tag{9.11}$$

We verify (9.10), using (le8) through $\mathsf{Eq}(ab'lp'-1) \supset \mathsf{Perm}(ab'lp'-1)$ in the last implication (the reasoning for (9.11) is essentially identical).

$$
\begin{array}{ll}
C_2 \ \Rightarrow & l \leq p-1 \wedge \mathsf{Order}(c) \wedge \forall j<k.\mathsf{QsortBounded}(qj) \wedge p-1-l<k \\
\Rightarrow & \mathsf{QsortBounded}(q(p'-1-l)) \\
\Rightarrow & \{C_2\}\ u \bullet (acl p'-1)\ \{C_2'\}\ @\ a[l..p'-1]pi\ .
\end{array}
$$

The idea behind the introduction of a fresh array b' is that we let its subarrays $b'[l..p'-1]$ and $b'[p'+1..r]$ serve as new snapshots for the "left" and "right" part of the now partitioned a. We then reason locally for each part, reaching C_2' (resp. C_3') from C_2 (resp. from C_3). By induction hypothesis, each recursive call sorts its subarray without affecting other parts: thus we can add the effects from these two calls using the invariance rule.Finally, by partitioning, we can conclude a is now sorted from l to r and that this subarray is still a permutation of the subarray of b in the same range. To jump from the predicate using b' to the predicate using b, we use Kleymann's consequence rule.

The inference follows. For clarity we let $\tilde{e_2} \stackrel{\text{def}}{=} a[l..p'-1]pi$ and $\tilde{e_3} \stackrel{\text{def}}{=} a[p'+1..r]pi$. We also need

$$C_1' \stackrel{\text{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr).$$

Below [*Seq-eoi*] and [*AppSimple*] are from Figure 12.

R.1. $C_2 \quad \supset \quad \{C_2\}\ q \bullet (a,c,l,p'-1)\ \{C_2'\}\ @\ \tilde{e_2}$	(9.10)
R.2. $\{C_2\}\ \mathtt{q}(\mathtt{l},\mathtt{p}'-1)\ \{C_2'\}\ @\ \tilde{e_2}$	(R.1, AppSimple)
R.3. $C_3 \quad \supset \quad \{C_3\}\ q \bullet (a,c,p'+1,r)\ \{C_3'\}\ @\ \tilde{e_3}$	(9.11)
R.4. $\{C_3\}\ \mathtt{q}(\mathtt{p}'+1,\mathtt{r})\ \{C_3'\}\ @\ \tilde{e_3}$	(R.3, AppSimple)
R.5. $\{C_2 \wedge [!\tilde{e_2}]C_3\}$ $\mathtt{q}(\mathtt{l},\mathtt{p}'-1)\ ;\ \mathtt{q}(\mathtt{p}'+1,\mathtt{r})$ $\{\langle !\tilde{e_3}\rangle C_2' \wedge C_3'\}\ @\ \tilde{e_2}\tilde{e_3}$	(R.2, R.4, Seq-eoi)
R.6. $C_1 \supset \exists b'.(([!\tilde{e_3}]C_2 \wedge C_2 \wedge [!\tilde{e_2}\tilde{e_3}](C_2' \wedge \langle !\tilde{e_2}\rangle C_3' \supset C_1'))$	(see below)
R.7. $\{C_1\}\ \mathtt{q}(\mathtt{l},\mathtt{p}'-1)\ ;\ \mathtt{q}(\mathtt{p}'+1,\mathtt{r})\ \{C_1'\}\ @\ \tilde{e}$	(R.6, Consequence-Aux)

Let $\mathsf{Dist}' \stackrel{\text{def}}{=} \mathsf{Distinct}(a[0]..a[\mathsf{size}(a)-1]b[0]..b[\mathsf{size}(b)-1]b'[0]..b'[\mathsf{size}(b)-1]pi)$. Then the entailment in R.6 above is verified as follows. First-order logic allows the following entailment

$$C_1 \quad \Leftrightarrow \quad C_1 \wedge \exists b'.(\mathsf{Eq}(ab'lr) \wedge \mathsf{Dist}') \quad \Rightarrow \quad \exists b'.D$$

where the definition of D is next.

$$D \stackrel{\text{def}}{=} \begin{pmatrix} \mathsf{Eq}(ab'lr) \wedge \mathsf{Parted}(b'clrp') \wedge \mathsf{Perm}(ab'lr) \wedge \mathsf{Perm}(ablr) \\ \wedge \\ \mathsf{Order}(c) \wedge l \leq p' \leq r \wedge \mathsf{Dist}' \wedge \forall j < k.\mathsf{QsortBounded}(qj) \end{pmatrix}$$

Now clearly

$$D \quad \Rightarrow \quad C_2 \wedge C_3 \quad \Rightarrow \quad C_2 \wedge [!\tilde{e_2}]C_3,$$

The former implication being first-order logic which the latter using (ua), since $C_3^{\text{-}!\tilde{e_2}}$. It is also the case that

$$D \quad \Rightarrow \quad \mathsf{Parted}(b'clrp') \wedge !a[p'] = !b[p'] \wedge \mathsf{Dist}'$$

1. $C_2' \wedge C_3'$	
2. $\mathsf{Perm}(ab'l(p'-1)) \wedge \mathsf{Perm}(ab'(p'+1)r)$	(1)
3. $!a[p'] = !b'[p']$	
4. $\mathsf{Perm}(ab'lr)$	(2, 3)
5. $\mathsf{Perm}(bb'lr)$	
6. $\mathsf{Perm}(ablr)$	(4, 5)
7. $\mathsf{Sorted}(acl(p'-1)) \wedge \mathsf{Sorted}(ac(p'+1)r)$	(1)
8. $\mathsf{Parted}(bclrp')$	
9. $\mathsf{Sorted}(aclr)$	(4, 7, 8)

Hence in fact

$$(!a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset ((C_2' \wedge C_3') \supset C_1')$$

which in turn implies

$$(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset ((C_2' \wedge C_3') \supset C_1').$$

To this tautology we add universal content quantification with respect to $\tilde{e} \stackrel{\text{def}}{=} \tilde{e}_2\tilde{e}_3$ to obtain

$$[!\tilde{e}]\,(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset ((C_2' \wedge C_3') \supset C_1').$$

But in view of Dist', all terms in the premise of that last term, are $!\tilde{e}$-free, hence we apply Proposition 6.6.

$$(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset [!\tilde{e}]\,((C_2' \wedge C_3') \supset C_1').$$

Now, with Dist', C_2' is $!\tilde{e}_3$-free, so C_2' and $\langle !\tilde{e}_3\rangle C_2'$ are in fact equivalent, using (e4, ea). That means we can refine that last big implication.

$$(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset [!\tilde{e}]\,((\langle !\tilde{e}_3\rangle C_2' \wedge C_3') \supset C_1').$$

Combining all this, yields the assertion

$$C_1 \quad \supset \quad \begin{pmatrix} C_2 \wedge [!\tilde{e}_2]C_3 \\ \wedge \\ [!\tilde{e}]\,((\langle !\tilde{e}_3\rangle C_2' \wedge C_3') \supset C_1') \end{pmatrix}$$

which is (R.6) used above.

Fig. 20 Partitioning algorithm.

```
1   λ(a,c,l,r)
2       let pv = !a[r] in
3           p := l;
4           i := l;
5           while !i < r
6                   if c(!a[!i], pv) then
7                           swap( a[!p], a[!i] )
8                           p := !p + 1
9                   i := !i + 1
10          swap(a[r], a[!p]);
11          !p
```

Derivation (3): Partition The final step is to verify `partition`, the heart of the algorithm where the array is divided into a part which contains only elements smaller than $!a[r]$ and one where all elements are bigger than $!a[r]$, both w.r.t. the total comparison procedure that was passed as an argument. Let

$$C^{pre}_{part} \stackrel{\text{def}}{\equiv} \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c).$$
$$C^{post}_{part} \stackrel{\text{def}}{\equiv} \mathsf{Parted}(aclrp') \wedge C^{pre}_{part}$$

The main inference for partition (establishing M1 in the main derivartion) follows.

P.1.	$\{C^{pre}_{part}\}\ !a[r] :_{pv} \{C^{pre}_{part} \wedge pv = !a[r]\} @ \emptyset$	(Deref, Cons)
P.2.	$\{C^{pre}_{part} \wedge pv = !a[r]\}\ p := l\ \{C^{pre}_{part} \wedge pv = !a[r] \wedge !p = l\} @ p$	(AssignVInit)
P.3.	$\{C^{pre}_{part} \wedge pv = !a[r] \wedge !p = l\}\ i := l\ \{C_{preloop}\} @ i$	(AssignVInit)
P.4.	$\{C_{preloop}\}\ \texttt{loop}\ \{C_{postloop}\} @ a[l...r-1]pi$	
P.5.	$\{C_{postloop}\}\ \texttt{swap}(a[r],a[!p])\ \{C^{post}_{part}[!p/p']\} @ a[r],a[!p]$	
P.6.	$\{C^{post}_{part}[!p/p']\}\ !p :_{p'} \{C^{post}_{part}\} @ \emptyset$	(Deref)
P.7.	$\{C^{pre}_{part}\}\ \texttt{let}\ pv = !a[r]\ \texttt{in}\ \texttt{partition}' :_{p'} \{C^{post}_{part}\} @ a[l...r]ip$	(Let, Seq)
P.8.	$\{\mathsf{T}\}$ `partition` $:_u$ $\{\forall aclr.\{C^{pre}_{part}\}u \bullet (a,c,l,r) \searrow p'\{C^{post}_{part}\} @ a[l...r]ip\} @ \emptyset$	(Abs, P.7)
P.9.	$\{C^{pre}_{part}\}$ `partition` $:_u$ $\{C^{pre}_{part} \wedge \{C^{pre}_{part}\}u \bullet (a,c,l,r) \searrow p'\{C^{post}_{part}\} @ a[l...r]ip\} @ \emptyset$	(Invar, Cons, P.8)
P.10.	$\{C^{pre}_{part}\}\ \texttt{partition}(a,c,l,r) :_{p'} \{C^{post}_{part}\} @ a[l..r]pi$	(AppSimple, P.9)

where we set:

- `partition′` is the body of the `let`-construct in `partition`, i.e. Lines 3 to 11 in Figure 20 (the previous page); and
- `loop` refers to the program fragment from Line 5 to 9 in Figure 20.

Further we have used the following pre/post conditions of the main loop:

$$C_{preloop} \stackrel{\text{def}}{\equiv} C^{pre}_{part} \ \wedge \ pv = !a[r] \wedge !p = l \wedge !i = l$$
$$C_{postloop} \stackrel{\text{def}}{\equiv} \mathsf{Invar} \wedge !i \geq r$$

We observe:

- Assignments (P.2) and (P.3) are trivial because neither p nor i are aliased by the assumptions given in C^{pre}_{part}.
- Swapping in (P.5) is immediate from the assumptions and the specification for swap in (9.6) (page 73).
- Each of (P.6–10) is a direct appliction of the respective rule.

Thus it only remains to establish the judgement of for the main loop in (P.4). We use the following intermedaite assertions.

$$\mathsf{Invar} \stackrel{\text{def}}{\equiv} \left(\begin{array}{c} C^{pre}_{part} \ \wedge \ l \leq !p, !i \leq r \ \wedge \ \mathsf{Leq}(acl(!p-1)pv) \\ \wedge \\ \mathsf{Geq}(ac(!p_0)(!i-1)pv) \ \wedge \ (!p < !i \supset c \bullet (!a[!p], pv) \searrow \mathsf{T}) \end{array} \right)$$
$$\mathsf{Leq}(aclrv) \stackrel{\text{def}}{\equiv} \forall l \leq j \leq r.c \bullet (!a[j], v) \searrow \mathsf{T}$$
$$\mathsf{Geq}(aclrv) \stackrel{\text{def}}{\equiv} \forall l \leq j \leq r.c \bullet (v, !a[j]) \searrow \mathsf{T}$$

Above, Invar is the main invariant, corresponding to the condition informally illustrated in Section 9.5. $\mathsf{Leq}(aclrv)$ (resp. $\mathsf{Geq}(aclrv)$) says the entries from l to r in a are smaller (resp. bigger) than v.

When inside the loop, the value of p and i differ from the invariant slightly, so that we also make use of:

$$C_{inloop} \quad \stackrel{\text{def}}{\equiv} \quad \mathsf{Invar} \wedge !i < r \wedge r - !i = j.$$

The following assertions are for the distinct conditions after the conditional branch is evaluated.

$$C_{then} \quad \stackrel{\text{def}}{\equiv} \quad C_{inloop} \wedge C \bullet (!a[!i], pv) \searrow \mathsf{T}$$
$$C_{\neg then} \quad \stackrel{\text{def}}{\equiv} \quad C_{inloop} \wedge \neg C \bullet (!a[!i], pv) \searrow \mathsf{T}.$$

Since we have

$$(C^{pre}_{part} \wedge !p = l \wedge !i = l \wedge pv = !a[r]) \supset \mathsf{Invar},$$

all we need to do for establishing $(P.4)$ is to show that:

$$\{\mathsf{Invar}\}\ \texttt{loop}\ \{C_{postloop}\} @ a[l...r-1]pi.$$

For its derivation we use the following notations.

- $\mathsf{Invar}' \stackrel{\text{def}}{\equiv} \mathsf{Invar}\{\!|!i+1/!i|\!\}\{\!|!p+1/!p|\!\}$
- `loopbody` is the body of the loop in the partitioning algorithm, i.e. Lines 6 to 9 in Figure 20.
- `ifall` is the conditional in the partitioning algorithm, i.e. Lines 6 to 8 in Figure 20.
- `ifbody` is the body of `ifall`, i.e. Lines 7 to 8 in Figure 20.
- We also let j to be a freshly chosen variable of Nat-type.

The derivation follows.

$L.1.$ $(\mathsf{Invar} \wedge !i < r) \supset r - !i > 0$

$L.2.$ $(\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r-!i \leq j)\{\!|!p+1/!p|\!\} \equiv (\mathsf{Invar}' \wedge r-!i \leq j)$

$L.3.$ $\{(\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r-!i \leq j)\{\!|!p+1/!p|\!\}\}$
$\quad p := !p+1$
$\quad \{\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r-!i \leq j\} @ p$ (AssignVInit)

$L.4.$ $\{C_{then}\}\ \texttt{ifbody}\ \{\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r-!i \leq j)\} @ a[l...r-1]ip$ (Seq, L.2, L.3, Weak)

$L.5.$ $C_{\neg then} \supset (\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r-!i \leq j)$

$L.6.$ $C_{cond} \stackrel{\text{def}}{=} (t \equiv c \bullet (!a[!i], pv) \searrow \mathsf{T})$

$L.7.$ $(\mathsf{Invar} \wedge r-!i > 0)$
$\quad \supset$
$\quad (\{\mathsf{Invar} \wedge r-!i > 0\} c \bullet (!a[!i], pv) \searrow t \{C_{cond} \wedge \mathsf{Invar} \wedge r-!i > 0\} @ \emptyset)$

$L.8.$ $\{\mathsf{Invar} \wedge r-!i > 0\}\ c(!a[!i], pv) :_t \{C_{cond} \wedge \mathsf{Invar} \wedge r-!i > 0\} @ \emptyset$ (AppSimple)

$L.9.$ $\{C_{inloop}\}\ \texttt{ifall}\ \{\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r-!i \leq j)\} @ a[l...r-1]pi$ (If, L.4, L.5)

$L.10.$ $(\mathsf{Invar} \wedge r-!i < j)\{\!|!i+1/!i|\!\} \equiv (\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r-!i \leq j)$

$L.11.$ $\{(\mathsf{Invar} \wedge r-!i < j)\{\!|!i+1/!i|\!\}\}\ i := !i+1\ \{\mathsf{Invar} \wedge r-!i < j\} @ i$ (AssignVInit)

$L.12.$ $\{C_{inloop}\}\ \texttt{loopbody}\ \{\mathsf{Invar} \wedge r-!i < j\} @ a[l...r-1]pi$ (Seq, L.9, L.10, L.11)

$L.13.$ $\{\mathsf{Invar}\}\ \texttt{loop}\ \{\mathsf{Invar} \wedge \neg c \bullet (!i, r) \searrow \mathsf{T}\} @ a[l...r-1]pi$ (WhileSimple, L.1, L.10)

$L.14.$ $C_{preloop} \supset \mathsf{Invar}$

$L.15.$ $(\mathsf{Invar} \wedge \neg c \bullet (!i, r) \searrow \mathsf{T}) \supset C_{postloop}$

$L.16.$ $\{C_{preloop}\}\ \texttt{loop}\ \{C_{postloop}\} @ a[l...r-1]pi$ (WhileSimple, L.1, L.13, L.14)

The entailments used in [*Consequence*] in (L.11) and (L.15) are easy and omitted.

Fig. 21 The annotated partitioning algorithm.

$u : \{\mathsf{T}\}$
λ(**a,c,l,r**)
$\{C^{pre}_{part}\}$
let pv = !a[r] in
$\{pv = !a[r]\}$
p := l
$\{!p = l\}$
i := l
$\{!i = l\}$
while !i < r
$\{\mathsf{Invar} \wedge !i < r\}$
if c(!a[!i], pv) then
$\{c \bullet !a[!i], pv \searrow \mathsf{T}\}$
swap(a[!p], a[!i])
$\{\mathsf{Invar}' \wedge r - !i \leq j\}$
p := !p + 1
$\{\mathsf{Invar}\{\!|\, !i+1/!i \,|\!\} \wedge r - !i \leq j\}$
$\{\mathsf{Invar}\{\!|\, !i+1/!i \,|\!\} \wedge r - !i \leq j\}$
i := !i + 1
$\{\mathsf{Invar} \wedge r - !i < j\}$
$\{C_{postloop}\}$
swap(a[r], a[!p])
$\{C^{post}_{part}[!p/p']\}$
!p
$\{C^{post}_{part}\}$
$\{\{\mathsf{T}\}\ \texttt{partition} :_u \{\mathsf{Partition}_u\}\}$

This concludes the proof of partition, establishing (M.2) of the main derivation, hence concluding the verification of the quicksort. Figure 21 presents the asserted version of `partition`.

10 Discussion

10.1 Further Topics.

Data Structures. The proposed logic can easily accommodate richer data structures. Treatment of arrays has been discussed in the main sections: lists and vectors are similarly treated. Integration of polymorphism and recursive types is straightforward following [36]. Record and variants types (labelled unions) are treated like products and sums. As we have seen in the cases for products, sums, and arrays, the presented framework offers a method of reasoning in which essentially arbitrarily complex data structures can be reasoned precisely following their syntactic structure, including those with destructive update. As a simple example, a mutable list is easily treated as a pair which contains two references, the first one containing a datum of type α and the second one containing a datum of this mutable list itself. Formally the type can be given as:

$$\mathsf{MList}(\alpha) \quad \stackrel{\text{def}}{=} \quad \mu X.(\mathsf{Ref}(\alpha) \times \mathsf{Ref}(X))$$

If we wish to define the operation such as e.g. $\mathsf{setcar}(e_1, e_2)$ (which assigns e_2 to the car cell of e_1), then we can regard it as an abbreviation for $\pi_1(e_1) := e_2$, so that it can be reasoned by the obvious refinement of [*AssignSimple*].

Local References. The analytical approach discussed above can encompass a reasoning method for even richer data structures when we augment the present logic with treatment of new (or local) reference generation. At the level of programming language, the grammar is extended as:

$$M \quad ::= \quad \ldots \quad | \quad \mathtt{new}\, x := M \,\mathtt{in}\, N$$

where $x \notin \mathsf{fv}(M)$. The reduction semantics (with configurations of the shape $(\nu \tilde{x})(M, \sigma)$ and associated typing rule are standard [63]. For a logical treatment, a restricted class of behaviours, where local references are never allowed to go out of the original scope, is treated easily following the traditional approach. In this case, local variables are used for storing parameters and local datum only during each run of a procedure body. This class is precisely the standard Hoare's rule for local variable treats. In the present logic, we obtain the following rule.

$$\frac{\{C^{\frown x}\}\, N :_n \{C_0\} \quad \{(C_0 \sharp x)[!x/n]\}\, M^{\Gamma;\Delta\cdot x:\mathsf{Ref}(\alpha);\beta} :_m \{C'^{\frown x}\}}{\{C\}\, \mathtt{new}\, x := N \,\mathtt{in}\, M^{\Gamma;\Delta;\beta} :_u \{C'\}} \tag{10.1}$$

which says that, when inferring for M, we can safely assume that the newly generated x is distinct from existing reference names, and that the description of the resulting state and value, C', should not include this new reference. The predicate $C_0 \sharp x$ (the notation is from Reynolds' specification logic) can be defined as $[!x]C_0$ if the target programming language does not have equality on reference names as an expression. The rule has a "simple" version given as follows.

$$\frac{\{(C \sharp x)\{\!|e/!x|\!\}\}\, M^{\Gamma;\Delta\cdot x:\mathsf{Ref}(\alpha);\beta} :_u \{C'^{\frown x}\}}{\{C\}\, \mathtt{new}\, x := e \,\mathtt{in}\, M^{\Gamma;\Delta;\beta} :_u \{C'\}} \tag{10.2}$$

The rules for this restricted forms of new reference generation allow us to treat the standard parameter passing mechanism in procedural languages such as C and Java through the following simple translation, cf. [14]). Assume given a procedure definition $\mathtt{f}(\mathtt{x},\mathtt{y})\ \{...\}$ in a procedural language. Then it is transformed into the following λ-abstraction augmented with new reference generation (where the part ... gives the code corresponding to the original procedure body):

$$\lambda(x',y').\mathtt{new}\ x := x'\ \mathtt{in}\ \mathtt{new}\ y := y'\ \mathtt{in}\ ...$$

The translation makes it clear the fundamental mechanism of parameter passing in imperative languages. In particular, since x and y are freshly generated, they are never aliased with each other nor with existing reference names. At the level of logic, this aspect is precisely captured by (10.2) (since stack variables are only used within a procedure body, there is no need for C' in (10.2) to mention x, hence the rule (10.2) suffices to reason this case). Thus the (lack of) aliasing in stack variables can be analysed as a special case of aliasing in general reference names, allowing a precise and uniform understanding.

Finally a full class of behaviour with local state allows not only above ephemeral local references but also those which survive, and get extruded, beyond their original scope. For capturing this class of behaviour, we extend the assertion language with a construct reminiscent of the ν-operator in π-calculi, with a rule generalising (10.1). We shall report on this topic in a forthcoming exposition.

10.2 Related Work (1): Compositional Program Logics for Aliasing

After early work on assertion-based methods for imperative programs including those by Turing and Naur, a significant step was taken by Floyd [15] in which flow charts are annotated with assertions, partly based on which Hoare's paper [31] introduced a compositional program logic in its full generality, putting forward a framework in which general properties of a program is verified strictly following syntactic structure of programs in an axiomatic framework. The initial Hoare logic is developed for the language without aliasing. After Hoare's work, several approaches have been experimented to treat aliasing of imperative variables in the framework of Hoare logic. Below we summarise these work with an aim to position the present work in a historical perspective (related comparative discussions are also found in [33, 34, 36]).

Broadly speaking, we may find the following five approaches in the study of reasoning methods for aliasing in the past three decades.

(1) In one approach, we prohibit aliasing by a suitable restriction, especially in procedure calls with reference name parameters. This idea is based on the observation that aliasing is practically nothing but harmful, so that assertions only treat those cases without aliasing [24, 71].
(2) In another approach, we verify programs with aliasing by considering all possible ways to coalesce names in program text and by verifying each case one by one using the non-aliased, original proof system [62, 69, 72].

(3) In the third approach, we verify programs with aliasing using the standard assertion language except we distinguish reference names and their content in notations. State change by assignment is treated using semantic update [10, 11, 39, 44, 67, 70].
(4) A closely related to (3) above is axiomatic treatment of pointers, arrays and other destructive data structures using a conditional expression which branches in (in)equality of pointers [4, 7, 18, 46, 56–58] (which, in hindsight, arises as a special case of (3)).
(5) An approach based on a special connective for "separation" of heap storage and resource-sensitive proof rules, as found in a series of recent remarkable activities by Reynolds, O'Hearn and others [8, 16, 61, 65, 66].

Below we discuss the past (and present) work in each thread one by one.

(1) Prohibition of Aliasing. As we discussed in Introduction, one of the significant places where aliasing arises is in procedure calls, where aliasing is rarely useful. Thus one possible approach to the aliasing in this context is to prohibit aliasing either by checking programs either statically or dynamically. This is the approach initially taken by Hoare and Wirth [24] in their axiomatic treatment of a subset of PASCAL, stipulating the absence of aliasing in arguments of procedure calls. Subsequently the same principle is applied in the axiomatic semantics of Euclid [71], a descendant of Pascal without goto, side effects, global assignments, higher-order or nested procedures. To prevent aliasing, Euclid also requires procedure arguments of reference types to be non-overlapping. More recent treatment of the same idea can be seen in JML [40] and various type-based analyses.

We believe general treatment of aliasing in program logics is valuable even if, in a given programming practice, aliasing is often simply to be avoided so that we seldom need to reason about programs with aliasing. Indeed, without a precise scientific understanding of aliasing, we may not be able to appreciate in what situations it is harmful, in what precise way it makes reasoning hard, and in which cases it can be dealt with tractably. Program logics offer us deep structural understanding of a general class of programs behaviours which can indeed be extended to those with aliasing: the general perspective such an understanding offers would be invaluable even if our sole purpose is to prohibit the use of aliasing. We also observe, in several significant fields of programming practice, aliasing is used as one of the unavoidable, and often useful, features. One may also note they naturally arise from the syntax of programs, and are inevitable whenever we allow references to appear in data structures (indeed this is precisely aliasing is *useful*, since it allows sharing which is sometimes essential not only for efficiency but also for the very functions of the programs: consider a program which always runs without interleaved with other programs but which nevertheless manipulates a data structure shared by other threads).

(2) Alias Analysis based on Syntactic Distinction. Since we have a clean, tractable proof system for programs without aliasing, one may as well consider all possible cases of aliasing for a given program and verify each case separately. This is the approach taken in Alphard by Shaw, Wulf and London [69, 72] and by Olderog [62]. Both use logics for programs without aliasing (whose assignment axiom follows Hoare's original

system). For the purpose of reasoning about programs whose reference names may be coalesced in procedure bodies, they consider each possible way of aliasing arguments. Olderog [62] established soundness and relative completeness of a proof system for a large subset of Algol, where it is shown that this approach goes well with logical treatment of procedures based on the so-called copy rule.

In this approach, we find the first systematic use of name distinction in the sense of Section 3, albeit at a syntactic level. At the same time, this approach does not allow assertions to directly describe properties of a program in the presence of aliasing. More practical issue of this approach is that it arguably violates the basic spirit of compositional program logics, since we cannot reason about the whole of a program starting from its subprograms as given. The issue becomes particularly visible when our concern goes beyond procedure calls with reference parameters, since with programs which treat references of references as well as data structures which carry references, coalescing does not appear in a program text explicitly but is hidden under expressions, as in $!x :=!!y$; $!z :=!!w$ where $!x$ and $!!w$ may be aliased. In such situations, a general treatment of aliasing becomes anyway necessary.

(3) Alias Analysis based on Semantic Update. Since aliasing is about (in)equality of reference names, it is natural to allow an assertion language to express (in)equations on reference names. For this purpose we need to distinguish reference names and their content. This can be done in one of two ways.

- In the first approach, the assertion "$x = y$" for imperative variables x and y means the content of a reference named by x and that of a reference named by y are equal, *not* that x and y refer to the same reference (in other words, these variables have the type, say, Nat rather than type $\mathsf{Ref}(\mathsf{Nat})$). To stipulate (in)equations on reference names, we can use either a special predicate or a term constructor.
- In the second approach, the assertion "$x = y$" for imperative variables x and y means they are identical as references, or, more intuitively, x and y refer to the same memory cell (in other words, these variables have the type, say, $\mathsf{Ref}(\mathsf{Nat})$, just as in ML). This necessitates the use of a special notation for denoting the content of a reference, for which the present work used the ML notation $!x$.

There are no essential difference between these two approaches, even though the second one may be more economical from the viewpoint of the logic with equality since we can conclude $!e_1 =!e_2$ from $e_1 = e_2$ simply by the law of substitutivity. This also indicates these two approaches are (can be made) formally equivalent by adding the axiom "$\&x = \&y \supset x = y$" to the first approach, assuming (as in C) $\&x$ denotes the "address of x".

In either way, distinguishing these notions is fundamental to this thread of study. Historically it would be Strachey who first distinguished these two notions. More recently, in typed languages such as ML and Haskell, this distinction is fundamental at the level of the practice of programming due to the need to assign suitable types to expressions. Once an assertion language has this distinction so that it can treat aliasing of reference names, we then need to consider how to treat state change in the presence of aliasing. Concretely this is tantamount to formulation of axiom(s) for assignment, which may as well be done through semantic update, replacing syntactic update

in Hoare's axiom (the use of semantic update in Hoare logic is closely related with a method for reasoning of arrays and other destructive data structures which we treat as the next thread).

Given the present work is a descendant of this thread of research, there is some interest to trace its historical origins.

Janssen, van Emde Boas [39]. It seems it is Janssen and van Emde Boas [39] who, inspired by Montague semantics for natural languages [55], first consider an assertion language in which reference names and their content are given distinct notations. Their work is not so much about Hoare-like compositional logics but rather about introducing an algorithm for computing (arguably) strongest postconditions. The target language is a finite fragment of ALGOL68 which allows references to carry other references, multi-dimensional arrays and assignment to complex expressions such as

$$a[a[7]] := 3 \qquad (\texttt{if}\ !x = 4\ \texttt{then}\ y\ \texttt{else}\ z) := 5.$$

On the other hand the language does not include loops, recursion nor procedures.

The algorithm for strongest postconditions is specified in an intensional logic containing λ-expressions. The logic uses an explicit dereference operator for distinguishing between a reference and its content. In addition, Janssen and van Emde Boas introduce a substitution operator $\{\!|x/!y|\!\}$ which is novel in that it only work on dereferenced names $!y$ in our notation, not on all free names. So, in our notation, we have a substitution such as $\{\!|x/!y|\!\}y = y$. Each program p is compiled into a predicate transformer

$$[\![p]\!] = \lambda P.C$$

which takes a precondition and returns the corresponding strongest postcondition. For example, using the symbolisms of our logic:

$$[\![x :=!y]\!] = \lambda P.\exists z.(\{\!|z/!x|\!\}P \wedge !x =!y).$$

Feeding $\{!x = 1 \wedge !y = 2\}$ to this predicate transformer yields

$$\begin{aligned}
[\![x :=!y]\!]\{!x = 1 \wedge !y = 2\} &= \exists z.(\{\!|z/!x|\!\}(!x = 1 \wedge !y = 2) \wedge !x =!y) \\
&= \exists z.(\{\!|z/!x|\!\}(!x = 1) \wedge \{\!|z/!x|\!\}(!y = 2) \wedge !x =!y) \\
&= \exists z.(\{\!|z/!x|\!\}!x = \{\!|z/!x|\!\}1 \wedge \{\!|z/!x|\!\}!y = \{\!|z/!x|\!\}2 \wedge !x =!y) \\
&= \exists z.(!z = 1 \wedge !y = 2 \wedge !x =!y) \\
&= !y = 2 \wedge !x =!y
\end{aligned}$$

A key limitation of this approach is that the transformation of $\{\!|z/!x|\!\}(!y = 2)$ into $!y = 2$ is *syntactic*, i.e. does not take into account the possibility that x and y might be identical. Another issue is that it is not clear whether their method extends to languages with loops and recursions given their method is tantamount to abstractly executing each program step one by one. In spite of these limitations, their work is notable in that it first put forwards the significance of distinction between reference names in describing aliased situations logically. A satisfactory treatment of aliasing which uses this idea combined with semantic update is to be treated in the work by Cartwright and Oppen, which we discuss next.

Cartwright and Oppen [10, 11]. As discussed in Introduction, the work by Cartwright and Oppen [10, 11] shows how one can use distinction on reference names and semantic update as part of the standard assertion language of Hoare Logic together with semantic update, and presents a formal result which decomposes semantic update into reference name (in)equations. In [10, 11], the authors treat a programming language with multiple assignment, (recursive) first-order procedures and pointers. The assertion language takes the first approach for distinguishing a reference name from its content, introducing a specific predicate which says reference names *per se* are distinct. The underlying model is inspired by McCarthy's articulation of imperative computation [48]. In this framework, [10, 11] presents two related logics.

1. First they presented a logic where the "distinct" predicate noted above and semantic update are present, but the programming language has no pointers (hence no aliasing except that coming from arrays). After observing this semantic update coincides with syntactic update in the absence of aliasing, they established soundness and relative completeness of their proof rules.
2. The second logic extends the first one with pointers, at the level of both programs and assertions. For assignment of the form (in our notation) $!x := e$, it is observed that the rule of the shape (again in our notation) $\{C\{\!|e/!!x|\!\}\}!x := e\{C\}$ suffices, where the semantic update is no longer replaceable by a syntactic counterpart. Then a compositional translation of the semantic update is presented which uses the "distinct" predicate. They also treat a rule for procedures which allow pointer passing and discussed its soundness and completeness. They used the "swap" program as a reasoning example (cf. Section 9.3). 9.3).

In spite of complexity in presentation, their work sets a clear milestone in the treatment of aliasing in Hoare's logic, by (1) distinguishing reference names and content, (2) introducing semantic update in the assertion language, and (3) showing semantic update can be eliminated through its decomposition into (in)equations on reference names. Note (3) is fundamental to keep compositional proof rules syntactic in principle.

As we discussed in Introduction, a basic issue of the logic(s) as presented in [10, 11] is that, although semantic update becomes "syntactic" by decomposition, it is in practice hard to carry out real logical calculation (this point is indeed acknowledged in [10, 11]). Other issues are the lack of structured reasoning principles about extensional behaviour of aliased programs [10, 11]. Treatment of a wide range of data structures and higher-order procedures (which was anyway beyond the state of the art at the time) is also left as a future issue. The present work addresses these issues by clarifying the logical status of semantic update through modal operators and integrating them with the standard assertion language, combined with a formula which captures imperative applicative behaviour. At the level of models, the use of distinction in the construction of models arguably contributes to its simplicity.

It should be mentioned that there is an independent work by Morris [56–58] which presented essentially the same ideas but in a syntactically more tractable framework and which examined more fully practical consequences of their use (using a slightly different formulation). Since Morris's work is naturally positioned in the next (fourth) thread, his work will be discussed there. Here we proceed to discuss those work which follows the same thread of Cartwright Oppen's study.

Trakhtenbrot, Halpern and Meyer [70]. A work on aliasing which shares the same approach as the one by Cartwright and Oppen is by Trakhtenbrot, Halpern and Meyer [70], which presents an axiomatic semantics to a rich subset of Algol68 guided by a denotational orientation (there is also a work by Schwartz [67] which gives an axiomatic semantics to a subset of Algol68 and which arguably treats aliasing, though the account on aliasing in [67] is brief and informal). The work by Trakhtenbrot, Halpern and Meyer is based on a logical language with clear distinction between reference names and their content, as well as semantic update. Notably they use the second approach for distinguishing a reference name and its content (i.e. in their logic x denotes a reference name while $content(x)$ denotes its content), thus preceding the present work in its literal use of an axiom of the form (in our notation) "$\{C\{\!|e/!x|\!\}\}x := e\{C\}$". In fact, we believe this "content notation" is a basic element to reach content quantification: if this study had been continued, it may as well have reached content quantificaion. In fact, they presented an invariance rule reminiscent of ours, implicitly using universal content quantification, though its meaning is not concretely illustrated. Their semantically oriented technical development is also close to the present work, though it does not treat imperative (higher-order) expressions. This point reflects the underlying language (a subset of Algol), which is based on a strict distinction between imperative procedures and stateless expressions.

Kulczycki, Sitaraman, Ogden, and Leavens. More recently Kulczycki et al. [44] study possible ways to reason about aliasing induced by call-by-reference procedure calls, in the context of JML [40], a behavioural interface specification language for Java in the tradition of Larch [21]. As in this work, JML [40] does have notations to distinguish between reference names and their content, even though this notation seems currently not widely in use.

Ghica. Another recent work which uses distinction between reference names and their content in assertions is the work by Ghica [16], who introduces a variant of Reynolds's Specification Logic for Idealised Algol with first-order procedures and expressions with side effects but without recursion (Specification Logic is briefly discussed later: a fuller discussion can be found in [37, Section 8]), Ghica's emphasis lies in treatment of expressions with side effects in reasoning, for which purpose the concept of *stability* is introduced; and in the application of the resulting logic for model checking. An expression or command in a program is *stable* if it behaves the same way all throughout the computation. A typical example of a stable expression is a constant. Assertions use identifiers that denote only stable objects, which means that $\forall x^{\tau}.C$ cannot quantify over all objects in the semantic domain interpreting the type τ. The stability allows the use of expressions of Idealised Algol as terms of an assertion, though one needs non-trivial proof rules to guarantee stability for expressions.

Ghica's use of dereference notation comes from the notation in his underlying programming language: for treating assignment in his logic, he does not use semantic update but follows Reynolds' approach, using non-interference to guarantee that an inference for assignment is safe in the presence of aliasing (we suspect this approach may not be as general as the one based on semantic update since it prohibits aliasing). These side conditions are semantic, in the sense that we need to reason about behaviour of

programs separately, i.e. outside of the logic, in order to guarantee these side conditions (which is different from the standard Hoare Logic where a proof for a judgement for a program is usually given by a derivation which precisely follow structure of programs combined with those for deriving true sentences in the assertion language). The efforts needed for guaranteeing stability in turn indicates the potential complexity incurred by combining procedures, aliasing and data types. This complexity is addressed in the present work through the standard logical machinery combined with logical primitives for articulating involved computational situations.

In spite of the use of Specification Logic as a basis, Ghica's work focusses on first-order procedures because of his intention to use the logic for model checking. It would be an interesting subject of study to combine the presented (arguably simpler) logical framework for aliasing and his model checking framework.

(4) Logical Analysis of Arrays and Other Mutable Data Structures.

McCarthy [48]. Arrays involve aliasing, in the sense that, for example, in

$$a[3] := 1 \ ; \ a[e] := 2;$$

for some expression e, may or may not result in the content of $a[3]$ being 2. McCarthy's early work [48] presented a logical machinery to describe update of an array, centring on the following axiom:

$$\mathit{select}(\mathit{update}(a,e_1,e_2),e_3) \quad \equiv \quad \texttt{if } e_1 = e3 \texttt{ then } e_2 \texttt{ else } \mathit{select}(a,e_3)$$

where (1) $\mathit{select}(a,e)$ denotes the e-th entry of an array a; (2) $\mathit{update}(a,e_1,e_2)$ denotes a new array in which its e_1-th entry is updated with e_2; and (3) $\texttt{if } A \texttt{ then } e \texttt{ else } e'$ returns e if A is true, e' if not. This last predicate, $\texttt{if } A \texttt{ then } e \texttt{ else } e'$, is central to the reasoning about aliased array entries when A indicates (in)equations on indices. McCarthy's work is done before compositional program logics have been invented: After the logical method by Floyd and Hoare has become wide-spread, there are several work which centres on the use of this conditional expression and its extensions for logical treatment of arrays and other data structures with destructive update.

Luckham and Suzuki [46]. One of the well-known early work in this thread will be the one by Luckham and Suzuki [46] who propose a Hoare logic for Pascal extending McCarthy's approach. Selection and update are extended to (destructive records) and pointers, where for the latter it is assumed that a selection is done from a "reference class". A reference class denotes part of the heap, or, more simply, a collection of pointers. As such, reference classes can be used for specifying, among others, (in)equality on reference names. The main efforts of their work is in stipulating axioms for these extended operations. Compositional proof rules for assignment are given for individual data types, using substitutions that works on each data type, e.g. reference classes.

Other work in this period which work with mutable data structures include Burstall's work [8] which presents a method of reasoning a la Floyd for a list which does not have sharing, Kowaltowki's work [43] which expands Burstall's method to more complex class of data structures, and Reynolds's specification logic [66] which centres on higher-order procedures and prevention of non-interference.

Morris [56, 58]. A more uniform treatment of general data structures including pointers is done by Morris [56, 58], who presented an elegant account of an extension of Hoare logic based on conditional update. Morris's approach can also be positioned in the third thread, since he makes clear the distinction between a reference name and its content, using the notation $x\downarrow$ to denote the address of x (which is symmetric to the pointer notation $x\uparrow$ in Pascal). His technical treatment however centres on the conditional expression rather than semantic update. Morris's treatment starts from a notion of conditional substitution given as follows, assuming x and y are reference names of the same type in a given program.

$$y\{\!|e'/x|\!\} \quad \stackrel{\text{def}}{=} \quad \texttt{if}\ x\downarrow = y\downarrow\ \texttt{then}\ e\ \texttt{else}\ y$$

Here it is assumed that a term of a reference type $\mathsf{Ref}(\alpha)$ denotes, in an assertion language, its content: hence it is necessary to consider (in)equality of their names by taking their addresses. He showed, through examples, that this conditional update is extensible to complex expressions (the precise rule is treated by Bornat, see our later discussions on his work). We reproduce one of his calculations below (following the original presentation we use Pascal-like field-selection notation and omit obvious $\downarrow$'s):

$$\begin{aligned}
&(p.s.s)\{\!|v/u.s|\!\}\\
&\equiv ((p.s)\{\!|v/u.s|\!\}.s)\{\!|v/u.s|\!\}\\
&\equiv (\texttt{if}\ u = p\ \texttt{then}\ v\ \texttt{else}\ p.s.s)\{\!|v/u.s|\!\}\\
&\equiv (\texttt{if}\ u = p\ \texttt{then}\ (\texttt{if}\ v = u\ \texttt{then}\ v\ \texttt{else}\ v.s)\ \texttt{else}\ (\texttt{if}\ p = u\ \texttt{then}\ v\ \texttt{else}\ p.s.s)
\end{aligned}$$

One may observe the above inference assumes the data structure may allow recursive typing. Since $p.s.s$ is written $(!(!p.s)).s)$ in the imperative PCFv, we also observe the calculation given above corresponds to the expansion of $((m = (!(!p.s)).s)\{\!|v/!(!u.s)|\!\}$ in the present logic: though in many cases either such an expansion is unnecessary or partial expansion suffices. Since the operation easily extends to formulae, we now own the general axiom:

$$\{C\{\!|e'/e|\!\}\}\ e := e'\ \{C\}$$

which, because of the definition of conditional update above, means the same thing as $\{C\{\!|e'/!e|\!\}\}\ e := e'\ \{C\}$ in our notation.

As we noted, Morris's approach is equivalent to Cartwright and Oppen's one (whose publication precedes Morris's work, though Morris's work shows real reasoning examples not found in [10, 11]), in the sense that a formula involving the conditional expression are easily decomposable into those without it using (in)equations on reference names. Morris's approach is however more syntactic and is presented purely in the setting of the first-order logic with equality. Morris [56–58] further extends his method with axioms for linked lists, and used the resulting framework for verification of a Schorr-Waite algorithm.

Morris's presentation in [56–58] is characterised by simplicity and elegance. Placed in the setting of Morris's work, one may observe it is not only ease of logical calculation, but also a certain completion of a project of logically articulating aliasing in a classical framework, that the introduction of content quantification is aiming to address, through the compensatory nature of equations, quantifications and substitutions.

Gries and Levin [18]. Gries and Levin [18] treated proof rules for simultaneous assignment involving arrays as well as rules for Algol-like procedure call. Their treatment of array assignment, which extends Hoare and Wirth's rule [24], examines the aliasing of array entries by checking equality of indices. For procedural calls with aliasing (e.g. when array elements used in arguments for call-by-reference can be repeated), Gries and Levin took the approach of the second thread, reasoning for a concrete distinction of interest. The present approach allows us to treat aliasing uniformly, in the sense that, either in procedure calls or not, all programs with potential aliasing can be reasoned using strictly compositional rules — the rule for application, which corresponds to procedure call in their setting, needs no change from the rule in the alias-free logic. At the same time, concrete rules for each situation is invaluable for efficient reasoning. One of the fruitful directions then is to seek for an integrated approach to the proof rules for diverse concrete cases based on the general foundation the present work offers.

Bornat [7]. A recent work by Bornat [7] further examines Morris's method in detail, applying Morris's idea and its ramifications to larger examples. Bornat first makes explicit Morris's generalised calculation rule (Morris himself only presented the explicit rule in the case for a variable) as a formal axiom. The rule becomes, using the dereference notation and the conditional and assuming op in $op(\tilde{e})$ stands for not only a first-order operator but also other constructors *except* dereference:

$$e''\{\!|e'/!e_0|\!\} \stackrel{\text{def}}{=} \begin{cases} op(\tilde{e}\{\!|e'/!e_0|\!\}) & (e'' \stackrel{\text{def}}{=} op(\tilde{e})) \\ \texttt{if } e_0 = e_1 \texttt{ then } e' \texttt{ else } !(e_1\{\!|e'/!e_0|\!\}) & (e'' \stackrel{\text{def}}{=} !e_1) \end{cases}$$

which is the direct counterpart of axioms for content quantification. Using this generalised rule, Bornat considers assertions for several mutable data structures, using inductively defined formulae specific to each data structure. He then gives outlines of reasoning for several short, but non-trivial, algorithms which manipulate data structures. This follows Hoare's early method [29], but is characterised by incorporation of disjointness of regions in which data structures are allocated in heap. By introducing several auxiliary predicates, he aims to build up a reasoning technique in which state change (assignment) can be reasoned using the corresponding syntactic change, as in the original Hoare logic and Morris's work, but without having to worry about assertions on data structures unaffected by that assignment.

This is a common concern by many researchers in this thread: one of the possible methods to solve this issue would be to keep the universal treatment of assignment as given in Morris's (hence Cartwright and Oppen's) framework and to obtain a flexible and general way to assert on sharing structures of mutable data. These are, in retrospect, among the central aims of the present work. However there are other concerns in this problem domain even these two aims are achieved, one of which is discussed in the work by Hoare and Jifeng treated as the last work in this thread.

Hoare and Jifeng [30]. The work by Hoare and Jifeng [30] starts from Morris's work, but instead of centring on syntactic manipulation of formulae based on, say, (in)equations, they offer an assertion method centring on graphical objects (which has a syntactic counterpart), where graphs depicts sharing structures of mutable data (they presented

their ideas in the context of object-oriented programs). Semantically, two graphs are equated through their representation as their traces generated by regarding them as certain automata (which is close to process representation of data structures [52] whose semantics is considered by their transitions). The state change is treated as what they call "pointer swinging", which corresponds to Morris's assignment axiom, but now conceived graphically (and, at the semantic level, in terms of the operation on traces).

Hoare and Jifeng's work presents a fresh approach to the logic with pointers by directly capturing data structures as graphical objects, whose semantics are later treated as traces. Traces allow extensional treatment of these graphical objects, making it possible to treat them as part of the standard assertions. In the present work, the structure of shared pointers is represented though a sequence of "pointer of" relations represented through the dereference operation (i.e. "x points to e" becomes the assertion "$!x = e$"). Combination of such assertions is general enough to capture properties of arbitrary data structures including their shape. This analytical approach contrasts with the synthetic approach found in Hoare and Jifeng's work, in which graphical objects — which many times should play a central role (at least in a programmer's mind) when program design and development actually take place — are positioned as its basis.

These two approaches conspicuously differ, but at the same time may as well be complementary, both at the mathematical level and at the pragmatic level. For the former, an interesting subject of study is to inquire precise connection between the graphical/trace-based assertions (on a data object) and descriptions in the presented assertion language (on the same data object). For the latter we foresee the reasoning method where there are several representations of the same ideas which are mutually related (or even equivalent), which may as well include both graphical objects and a fine-grained, and general, logical presentation. One of the central roles of a framework for reasoning which we intend to build in the present and associated study should lie in its effective use in such reciprocal enrichment towards a truly general integrated logical foundations for software design and development.

(5) Separation Logic.

Prelude: Specification Logic. Specification logic by Reynolds [65] is a program logic for Idealised Algol which combine Hoare logic and, in some aspects, LCF, where Hoare triples appear textually in assertions. It aims to capture the whole semantics of Idealised Algol, a distilled version of Algol-68 introduced by Reynolds himself. One of the main emphasis (and motivations) of the logic is to be able to specify and guarantee the lack of interference between expressions, needed to tame intractability of write effects in call-by-name evaluations. For this purpose he used an assertion of the form $M_1 \sharp M_2$, which intuitively says two expressions do not interfere through shared variables. Since the logic is not a compositional program logic in the standard sense (for example its formula includes judgements on programs as its part) and because it is not directly about reasoning on aliasing, we do not treat Specification Logic here in detail (see [37, Section 8] for further account). However this concern on noninterference prepares another, and more recent, logic by Reynolds and others which is about aliasing, where the operator $\sharp$ is to be recaptured at the level of assertions.

Aliasing and Separation. Motivated by Burstall's early work discussed above as well as (arguably) by Reynolds's own work on noninterference, Reynolds, O'Hearn and others [61, 66] recently introduced a novel conjunction $*$ that stipulates, in addition to the standard conjunction, disjointness of the memory regions asserted on in its two conjuncts, for reasoning about low-level programs' behaviour with a focus on aliasing. Note $*$ can be considered as logical instantiation of $\sharp$ above. The resulting logic, called *Separation Logic*, uses Hoare logic for alias-free stack-allocated variables while introducing alias-aware rules for variables on heaps. This work can be positioned in the tradition of the fourth thread above but deserves an independent discussion because Separation Logic presents new logical machinery which, while addressing a related concerns, exhibits an interesting contrast with our work, both philosophically and technically.

Separation Logic starts from a resource-sensitive assignment:

$$\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}$$

where e is an integer expression, $[e]$ denotes the store with address e if it appears on the left-hand side, and its content if it appears on the right-hand side (we also assume e and e' do not include dereference of heap variables, i.e. expressions of the form $[e]$)). "$e \mapsto -$" denotes "$\exists i.e \mapsto i$". The rule, through the assertion $e \mapsto -$, *demands* that the memory cell assigned to is available at address e. This assertion roughly corresponds to $\exists y.!x = y$ in our logic, but the meaning is quite different, for example in that it is not logically equivalent to T. Separation Logic also has a symmetric dereference rule which demands the existence of a dereferenced variable [66]. As a typical reasoning example which demonstrates Separation Logic's resource-oriented nature, consider an assertion, sound in Hoare Logic and our's:

$$\{\mathsf{T}\}\ x :=!x\ \{\mathsf{T}\},$$

Note the command $x :=!x$ is (in our language) observationally equivalent to `skip`. The corresponding statement in Separation Logic can be written $[e] := [e]$ with a slight abuse of notation. Now we assert:

$$\{[e] \mapsto -\}\ [e] := [e]\ \{\mathsf{T}\}$$

whose left-hand side is the weakest pre-condition of T. In fact $\{\mathsf{T}\}\ [e] := [e]\ \{\mathsf{T}\}$, is unsound in their logic.

The rationale underlying the lack of derivability of $\{\mathsf{T}\}\ [e] := [e]\ \{\mathsf{T}\}$ is that in low-level languages, if we wish to assign a value to an address in a heap storage, that storage should be allocated first. That makes `skip` semantically distinct from $[e] := [e]$ and shows that the resource oriented nature is hard-wired into the core of Separation Logic (this may also be seen as a requirement demanded by the separating conjunction, rather then coming from the choice of programming language).

The reasoning rule $\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}$ does not treat the general case when the pre/post conditions are general formulae. This is obtained by a variant of the invariance rule, which uses separating conjunction:

$$\frac{\{C\}\ P\ \{C'\} \quad \mathsf{fv}(C_0) \cap \mathsf{modify}(P) = \emptyset}{\{C * C_0\}\ P\ \{C' * C_0\}} \tag{10.3}$$

The second premise is the standard side condition for invariance, where $\mathsf{modify}(P)$ is the set of all stack-allocated variables which P may write to. Apart from this side condition, soundness of this rule hinges on the resource-oriented assignment/dereference rules noted above, by which the existence of all the variables (addresses) in the heap which P may write to is explicitly stipulated in C. Thus $C * C_0$ can assert that C_0 does not talk about any variables P may modify. Like the standard invariance rule, it is intended to serve as an aid for modular verification of program correctness: indeed, this is one of the focal points of their logic.

The following short reasoning for the command $x := 2; y :=\!!z$ may demonstrate how reasoning in Separation Logic can use the separating conjunction in reasoning.

$$\frac{1\ \ \{x \mapsto -\}\ x := 2\ \{!x = 2\} \qquad \text{(Assign-SL)}}{2\ \ \{y \mapsto -\ \wedge\ z \mapsto i\}\ y :=\!!z\ \{y \mapsto i\ \wedge\ z \mapsto i\} \qquad \text{(Assign-SL)}}$$

$$\frac{}{3\ \ \{x \mapsto -\ *\ (y \mapsto -\ \wedge\ z \mapsto -)\}\ x := 2; y :=\!!z\ \{x \mapsto 2\ *\ \exists i.(y \mapsto i\ \wedge\ z \mapsto i)\} \qquad \text{(Seq)}}$$

Above, [*Assign-SL*] is an assignment rule of Separation Logic; [*seq*] combines the invariance rule, called *frame rule* by O'Hearn, with the standard sequencing rule; and "$x \mapsto ?$" stands for "$\exists i.(x \mapsto i)$". In the last step, we also use (obvious) structural rules.

Resources and Observability. Separation Logic's ability to reason about aliased references depends crucially on its resource-oriented nature and on the use of the separating conjunction $*$ and a special predicate $\mapsto$ to represent the existence and content of memory cells. Resource-orientation in turn comes from the focus on low-level programs' behaviour, including resource usage. In contrast, the present work aims at a precise logical articulation of observational meaning of programs, starting from high-level programming languages in the traditions of both Hennessy-Milner logic and Hoare's logic, as exemplified by Theorem 8.19. Another difference is that our logic is built on the standard logical apparatus of predicate calculus with equality to represent general aliasing (cf. Theorem 8.4). This difference comes to life for example in the [*Invariance*] rule discussed in Section 7, which plays a role similar to (10.3). Note that the write set in located formulae is always a subset of variables which the premise of the corresponding formulae in Separation Logic needs to mention. Sometimes the inclusion is proper, as we show below. Our rule relies on purely compositional reasoning about observable behaviour, which arguably contributes to tractability in practical reasoning; among others, as we have seen in Section 7, the use of $!x$-freedom in particular would almost always be automatically ensured.

To see concretely the difference between resource-oriented reasoning and observational reasoning, we derive a specification for $x := 2; y :=\!!z$ from above, but this time in our logic.

$$\frac{1\ \ \{\mathsf{T}\}\ x := 2\ \{!x = 2\}\ @\ x \qquad \text{(Assign)}}{2\ \ \{\mathsf{T}\}\ y :=\!!z\ \{!y = !z\}\ @\ y \qquad \text{(Assign)}}$$

$$\frac{}{3\ \ \{\mathsf{T}\}\ x := 2; y :=\!!z\ \{\langle !y\rangle\ !x = 2\ \wedge\ !y = !z\}\ @\ xy \qquad \text{(seq-I)}}$$

Reflecting observational nature, the pre-condition simply stays empty. Note also that $\langle !y\rangle\ !x = 2\ \wedge\ !y =\!!z$ is equivalent to $(x \neq y \supset !x = 2)\ \wedge\ !y =\!!z$, which is more general

than $x \mapsto 2 \ * \ \exists i.(y \mapsto i \ \wedge \ z \mapsto i)$. In fact located reasoning gives a full specification. The corresponding post-condition is derivable using separating connectives (since inequations are representable by $*$), we cannot directly use the invariance rule as in the present logic since z can be shared.

The above example suggests we may gain generality and flexibility by using the proposed logical framework. Indeed, it is clear that any reasoning in which Separating Logic's invariance/frame rule is used non-trivially, we can use the corresponding reasoning through the invariance rule and located judgements, while the converse is not generally true. In our logic treatment of resource-sensitive aspects, such as allocation, would then be done through a separate predicate (for example we may write $\mathsf{allocated}(e)$ to say e of a reference type is allocated. While $C_1 * C_2$ is practically embeddable as $[!\tilde{e}_2]C_1 \wedge [!\tilde{e}_1]C_2$ where $\tilde{e}_i$ exhausts active dereferences of C_i, the examples suggest the use of write sets in located judgements/assertions offers a more precise description and smooth reasoning.

Just as first-order logic with equality on which it is based, our framework aims to offer distilled primitives with a clear logical status for capturing complex computational behaviours. On its basis, further building blocks for description and reasoning arise, leading to a general descriptive power as well as principled reasoning. This aspect is demonstrated through the treatment of diverse data structures and higher-order procedures in the preceding sections. On the other hand, activities centring on Separation Logic offer valuable contributions to the deepening of understanding on the use of logics to (among others) low-level computation. In particular, based on intuitive abstraction, they put forward some of the notable (low-level) computational situations as targets of reasoning, together with associated reasoning techniques. This orientation can in turn have a fruitful interplay with the present approach, where a general framework is applied to the analysis of concrete, significant engineering ideas. This poses an interesting question, not only about Separation Logic but also all other previous work we have discussed so far: can we effectively incorporate into the present framework various insights concerning how complex programs and data structures can be reasoned? Does it lead, through such incorporations, to an integrated framework which makes it practical to guarantee correctness of large and complex software?

We close this comparative discussions by presenting one concrete example of such interplay, applying the analytical power of the present logic to simplification and generalisation of a refined invariance rule involving procedures by O'Hearn, Yang and Reynolds [61]. The original rule has several side conditions on behaviour of programs (including an operational condition on write effects) and restrictions on the use of formulae: below we present the corresponding rule in the present logic.

$$\frac{C_1 \ !\tilde{x}\text{-free} \quad \{C_0\}\ N\ \{C_0' * C_1\}\ @\,\tilde{x}\tilde{y} \quad \{C^{-f} \wedge \{C_0\}\ f\ \{C_0'\}@\tilde{x}\}\ M :_u \ \{C'\}@\tilde{x}}{\{C \wedge C_1\}\ \mathtt{let}\ f = \lambda().N\ \mathtt{in}\ M :_u \ \{C' \wedge C_1\}\ @\,\tilde{x}\tilde{y}} \quad (10.4)$$

where we assume f to be *ephemeral* in the sense that it occurs in M only in the shape of $f()$ and never under λ-abstraction. This is easily checkable by typing. The rule says if a program M uses a procedure f assuming that it only alters $\tilde{x}$, and under that condition M only alters the content of $\tilde{x}$, then if we instantiate f to a real program and it touches reference names distinct from $\tilde{x}$ but maintains invariance at those reference names, then

instantiating that procedure results in maintaining the invariance. Ephemerality of f is needed since if we store f or place it under abstraction, then the invariance in stored/abstracted behaviour cannot be maintained: in contrast, in the above case, we can adjust the invariance at the time of instantiation once and for all. In comparison with the rule in [61], the rule (10.4) differs in that it is purely compositional (i.e. does not demand conditions on behaviours of M and N outside of judgements). Note also our version of the rule does not restrict the use of stored higher-order procedures etc. in non-ephemeral procedure labels. The generality is obtained because we can now precisely identify why the strengthening of invariance is possible in the specific setting which the refined invariance rule in [61] treats.

10.3 Related Work (2): Other Related Work

A Reasoning Method based on Operational Reasoning Technique. There are threads of study on the reasoning methods for programs with aliasing that are not directly about compositional program logics. In this category we find, among others, development of operational reasoning methods studied by Maison [47] and Pitts and Stark [64] (both treat local references in the sense of Section 10.1). These methods are complementary in the sense that, for example, the correctness of proof rules may have to be verified using such operational methods; or, in a converse direction, we may sometimes be able to use logical methods to make some part of the operational proofs simpler. Such an integrated use of different methods is one of the interesting subjects for further studies.

Elimination Results. An elimination procedure similar to our Theorem 8.4 can be found in work by Calcagno, Gardner and Hague [9] who decompose a decidable propositional fragment of a logic with separated conjunction into first-order logic. Lozes [45] takes a similar but more semantic approach. While the target logic is different, [9] is close to our work as it is more syntactic and compositional. The main difficulties these authors had to face seem the finite and resource-oriented nature of the separating connectives. It is interesting to study whether the logical apparatus proposed in the present work can be used for building a clean descriptive framework for more intensional features such as resource bound.

Verification of Low-Level Code. Combination of aliasing and higher-order procedures often arises conspicuously in low-level (systems-level) programs. This is partly because, in systems-level code, programs arise as targets of manipulation, to be stored in memories and launched into activity as needed. Sharing is an essential feature in low-level code partly because of efficiency but more importantly because manipulating and enabling resource sharing is one of the central aims of various systems-level software. Apart from Separation Logic by Reynolds and O'Hearn on which we mentioned already, there are several recent works which address formal safety guarantee of low-level code addressing these features — higher-order procedures and aliasing — in an organised way. Since this is currently an active research area, we do not try to be exhaustive, but merely pick up two different, but related, approaches to this problem.

In [22, 60], Shao, Hamid and Ni study integration of typed assembly code [59] and Floyd-Hoare logic, which aims to offer a formal framework to guarantee expressive safety properties for assembly code. Among others their systems allow treatment of pointers (references) to higher-order code and mutable data. For this purpose they introduce a logical language which asserts on states of a typed-version of assembly language including these concerns. Their logic is mechanised in Coq. Their interest differs from ours in that their target is an assembly language while ours is a high-level language; both however share the interest in treating higher-order code and pointers. It would be an interesting topic for further study how the present approach on pointers and higher-order code may be usable at the level of assembly and other low-level languages.

In [2], Amal, Morrisett and Fluet present a framework to guarantee type-safety for a higher-order call-by-value imperative language in the presence of *strong update*, i.e. the update of a variable which can change its type (this may be considered as an extreme form of aliasing). The typing systems were developed which annotate programs with operations which manipulate types. The typing system is highly elaborate. As the authors of [2] noted, refined program analyses such as those studied in [2] may as well be founded on program logics, especially when they should treat complex behaviours such as aliasing. An interesting question would be whether the present logic and its ramifications can offer such a basis for program analysis for low-level language features such as treated in their work.

References

1. C– home page. http://www.cminusminus.org.
2. G. Morrisett A. Amal and M. Fluet. L3: A linear language with locations. In *the Seventh International Conference on Typed Lambda Calculi and Applications*, 2005.
3. Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. 163:409–470, 2000.
4. K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.
5. Friedrich L. Bauer, Edsger W. Dijkstra, and Tony Hoare, editors. *Theoretical Foundations of Programming Methodology, Lecture Notes of an International Summer School*. Reidel, 1982.
6. Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.
7. Richard Bornat. Proving pointer programs in hoare logic. In *Proc. Conf. on Mathematics of Program Construction*, LNCS. Springer-Verlag, 2000.
8. R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7, 1972.
9. Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *FoSSaCs'05 (to appear)*.
10. Robert Cartwright and Derek C. Oppen. Unrestricted procedure calls in Hoare's logic. In *Proc. POPL*, pages 131–140, 1978.
11. Robert Cartwright and Derek C. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.
12. Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
13. Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.

14. Erik Crank and Matthias Felleisen. Parameter-passing and the lambda-calculus. In *POPL '90*. ACM Press, 1990.
15. Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
16. Dan R. Ghica. *A Games-Based Foundation for Compositional Software Model Checking*. PhD thesis, Queen's University School of Computing, Kingston, Ontario, 2002.
17. Irene Greif and Albert R. Meyer. Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer. *ACM Trans. Program. Lang. Syst.*, 3(4), 1981.
18. David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst.*, 2(4):564–579, 1980.
19. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI'02*. ACM, 2002.
20. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
21. John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specifi cation*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.
22. Nadeem A. Hamid and Zhong Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proc. 17th International Conference on the Applications of Higher Order Logic Theorem Proving (TPHOLs'04)*, volume 3223 of *LNCS*, pages 118–135, September 2004.
23. Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
24. Hoare and Wirth. Axiomatic semantics of pascal. *ACM Trans. Program. Lang. Syst.*, 1(2):226–244, 1979.
25. C. A. R. Hoare. Algorithm 63: Partition. *Comm. ACM*, 4(7), 1961.
26. C. A. R. Hoare. Algorithm 64: Quicksort. *Comm. ACM*, 4(7), 1961.
27. C. A. R. Hoare. Algorithm 65: Find. *Comm. ACM*, 4(7), 1961.
28. C. A. R. Hoare. Proof of a program: FIND. *Comm. ACM*, 14(1), 1971.
29. C. A. R. Hoare. Notes on data structuring. In *Structured Programming*, pages 83–174. s. Academic Press, 1972.
30. C. A. R. Hoare and He Jifeng. A trace model for pointers and objects. pages 223–245, 2003.
31. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
32. Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
33. Kohei Honda. From process logic to program logic. In *Proc. ICFP'04*. ACM Press, 2004. A long version available from www.dcs.qmul.ac.uk/˜kohei/logics.
34. Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2004. Typescript, 234 pages.
35. Kohei Honda and Nobuko Yoshida. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.
36. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP'04*. ACM Press, 2004.
37. Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. Available at www.dcs.qmul.ac.uk/˜kohei/logics.
38. J. Martin E. Hyland and C. H. Luke Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
39. T. M. V. Janssen and Peter van Emde Boas. On the proper treatment of referencing, dereferencing and assignment. In *Proc. ICALP*, pages 282–300, 1977.
40. The Java Modeling Language (JML) home page. http://www.jmlspecs.org/.
41. Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.

42. Donald E. Knuth. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley, 1973.
43. Tomasz Kowaltowski. Data structures and correctness of programs. *J. ACM*, 26(2):283–301, 1979.
44. Gregory W. Kulczycki, Murali Sitaraman, William F. Ogden, and Gary T. Leavens. Reasoning about procedure calls with repeated arguments and the reference-value distinction. Technical Report TR ♯02-13a, Dept. of Comp. Sci., Iowa State Univ., December 2003.
45. Etienne Lozes. Elimination of spatial connectives in static spatial logics. *TCS*, to appear.
46. 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.
47. Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.
48. John L. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
49. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
50. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Proc. POPL'88*, 1988.
51. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.
52. Robin Milner. The polyadic π-calculus: A tutorial. In *Proceedings of the International Summer School on Logic Algebra of Specifi cation*. Marktoberdorf, 1992.
53. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.
54. Robin Milner, M. Tofte, and R. Harper. *The Defi nition of Standard ML*. MIT Press, 1990.
55. Richard Montague. *Formal Philosophy, Selected papers of Richard Montague*, chapter The proper treatment of quantification in ordinary English. Yale Univ. Press, 1973.
56. Joseph M. Morris. Assignment and linked data structures. In [5], pages 35–43, 1982.
57. Joseph M. Morris. A general axiom of assignment. In [5], pages 25–34. Reidel, 1982.
58. Joseph M. Morris. A proof of the Schorr-Wait algorithm. In [5], pages 44–52. Reidel, 1982.
59. 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.
60. Zhaozhong Ni. The interaction of type system and floyd-hoare logic, 2005. typescript.
61. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, 2004.
62. Ernst-Rüdiger Olderog. Sound and complete hoare-like calculi based on copy rules. *Acta Inf.*, 16:161–197, 1981.
63. Benjamin C. Pierce. *Type Systems and Programming Languages*. MIT Press, 2002.
64. Andy M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In *HOOTS'98*, CUP, pages 227–273, 1998.
65. John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, 1982.
66. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, 2002.
67. Richard L. Schwartz. An aximatic treatment of algol 68 routines. In *Proc. ICALP*, pages 530–545, 1979.
68. Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
69. Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and verification in alphard: defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564, 1977.

70. Boris Trakhtenbrot, Joseph Halpern, and Albert Meyer. From Denotational to Operational and Axiomatic Semantics for ALGOL-like languages: an overview. In *Proc. CMU workshop on Logic of Programs*, volume 164 of *LNCS*, pages 474–500, 1984.
71. Ted Venema and Jim des Rivieres. Euclid and pascal. *SIGPLAN Not.*, 13(3):57–69, 1978.
72. William A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of alphard programs. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976.

A Comments on Semantics of Assertions

In the following, we offer additional illustration on the satisfaction of (1) (standard) quantification and (2) evaluation formulae, both presented in Section 5.4.

Quantification over Reference Names. Below we illustrate the clause for satisfaction of (standard) quantification, which we reproduce below for convenience:

$$\mathcal{M} \models \forall x^{\alpha}.C \text{ if } \mathcal{M}' \models C \text{ for each } \mathcal{M}' \text{ such that } \mathcal{M} \leq_{x:\alpha} \mathcal{M}'.$$

As noted in Section 5.4, the definition coincides with the standard one when α is not a reference name, since, in that case, $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$ means the new entry for x is added to the *environment* part of the model. As an example, consider the formula $\exists x.x = 3$. When should $\mathcal{M} \models \exists x.x = 3$ hold? By the definition above, we know it holds when some $\mathcal{M} \leq_{x:\mathsf{Nat}} \mathcal{M}'$ exists with $\mathcal{M}' \models x = 3$. If we had decided that x in $\mathcal{M}$ must be fused with a name already in $\mathcal{M}$ of the same type, the truth of the existentially quantified formula would depend on $\mathcal{M}$ already having another name which is mapped to 3. This contradicts the standard interpretation of the existential quantification.

Next we consider the case when α is a reference type, contrasting to the treatment in the logic without aliasing, developed in [37]. In [37] quantifying over reference-typed variables was not available. That this logic was nevertheless observationally complete, shows that quantification over references is unnecessary in that setting. The reason for this is the absence of aliasing in that context. Reference names are constants there, not variables. They are akin, on other words, to 3 and 5, and we don't allow $\forall 5.C$ either. Just like a number can never stand for another number, in [37], a name $x^{\mathsf{Ref}(\alpha)}$ can never refer to a distinct other variable. This is very different now. Our programming language leads to the fusion of variables: $(\lambda xy.M)zz$. Since λ-abstraction in a programming language corresponds to $\forall$-quantification in the corresponding logic, we now need quantification over reference variables, because our we are now allowed to form terms $\lambda x^{\mathsf{Ref}(\alpha)}.M$. Syntactically, this trivial, what is not immediate is when $\mathcal{M} \models \forall x^{\mathsf{Ref}(\alpha)}.C$ is true. For non-reference types α, [37] defines

$$\mathcal{M} \models \exists x^{\alpha}.C \text{ iff some } V \in [\![\alpha]\!] \text{ exists with } \mathcal{M}[x:V] \models C.$$

Here $\mathcal{M}[x:V]$ is the model obtained from $\mathcal{M}$ by adding a new entry $x:[V]$. What would happen if we did the same for reference types? If we used

$$\mathcal{M} \models \exists x^{\mathsf{Ref}(\alpha)}.C \text{ iff some } V \in [\![\alpha]\!] \text{ exists with } \mathcal{M}[x \mapsto V] \models C$$

there'd be a problem: typing assures that x is not a name that occurs in $\mathcal{M}$ at all. That means that the status of x in the distinctions of $\mathcal{M}[x \mapsto V]$ is unclear. Prima facie there are three possible choices for the distinction of $\mathcal{M}[x \mapsto V] = (\mathcal{D}', \xi', \sigma')$, assuming $\mathcal{M} = (\mathcal{D}, \xi, \sigma)$.

1. The new name x is not fused with any existing names: $\mathcal{D}' = \mathcal{D} + x$.
2. The quantification $\forall x.C$ ranges only over already existing storage cells, i.e. x fused with an existing name: $\mathcal{D}' = \mathcal{D} + (x = y)$ for some $y \in \mathsf{dom}(\mathcal{D})$.

3. Either of the preceding two: $\mathcal{D}' = \mathcal{D} + x$ or some $y \in \mathsf{dom}(\mathcal{D})$ exists with $\mathcal{D}' = \mathcal{D} + (x = y)$.

We shall now explain that (3) is the correct choice for quantification at reference types while (1) should be used for all other types. Then we introduce a formalism that encompasses both forms of at once.

First, for the sake of the explanation, assume $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$ above in the definition of $\models$ meant that $\mathcal{M}'$ is a model that is exactly like $\mathcal{M}$, except that it has a fresh new entry for x, mapped to some value, but not coalesced with any of $\mathcal{M}$'s existing names. Now assume that α is a reference type. Consider the program

$$V \stackrel{\text{def}}{=} \lambda x.\texttt{if}\ x = y\ \texttt{then}\ 1\ \texttt{else}\ 2,$$

typable under $y : \mathsf{Ref}(\alpha)$. One expects that

$$\mathcal{M} \not\models \{\mathsf{T}\}\ V :_u \{\forall x.\{x \Downarrow\}u \bullet x \searrow n\{n = 2\}\}$$

for any model $\mathcal{M}$. So choose an appropriately typed $\mathcal{M}$. Here $M \Downarrow$ is a shorthand for $\{\mathsf{T}\}\ M :_u \{\mathsf{T}\}$. As $\mathcal{M} \models \mathsf{T}$, we need to consider if

$$\mathcal{M}[u : [V]] \models \forall x.\{x \Downarrow\}u \bullet x \searrow n\{n = 2\}.$$

This hold if for all $\mathcal{M}[u : [V]] \leq_{x:\alpha} \mathcal{M}'$, all $M \in \mathcal{M}'(x)$ and for all σ':

- $((VM)\mathcal{D}, \sigma') \Downarrow (\{k\}, \sigma'')$ for some $k \in \{1, 2\}$, assuming M converges, $\mathcal{M}' = (\mathcal{D}, \xi, \sigma)$, and
- $k = 2$.

Since the former is clearly true, satisfaction hinges on $k = 2$. The idea behind V is that Vy should converge to 1. But if we had decided that $\mathcal{M}[u : [V]] \leq_{x:\alpha} \mathcal{M}'$ means that x cannot be fused with any name already in $\mathcal{M}[u : [V]]$, then $(\mathcal{M}\mathcal{D}, \sigma')$ can never converge to y, which means $k = 2$ always holds. This would imply

$$\mathcal{M} \models \{\mathsf{T}\}\ V :_u \{\forall x.\{x \Downarrow\}u \bullet x \searrow n\{n = 2\}\}$$

for any model $\mathcal{M}$ and this is contrary to our expectations. This suggests that $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$ should allow fusing x with names already in $\mathcal{M}$, provided that α is a reference type. But should x in these circumstances always be coalesced? The answer is no and the example just given can be used to demonstrate why: if $\mathcal{M}$ contained only one reference of type α, which in this case would have to be y by typing, we would get

$$\mathcal{M} \models \{\mathsf{T}\}\ V :_u \{\forall x.\{x \Downarrow\}u \bullet x \searrow n\{n = 1\}\}$$

which is also not what we would like. Hence we need both: the possibility of fusion between a new name and one that is already in $\mathcal{M}$, but also the absence of coalescence. This is exactly what our satisfaction relation gives.

Evaluation Formula. We close this section with an explanation of the semantics of the evaluation formulae. When we assert $\mathcal{M} \models \{C\}e \bullet e' \searrow u\{C'\}$ of some expression e, usually a name, what we say is *roughly* this:

> Whenever, in the course of the evaluation of the current program we evaluate what e stands for, i.e. if we compute $[\![e]\!]_{\mathcal{M}'}$ in model $\mathcal{M}' = (\mathcal{D}, \xi, \sigma)$ satisfying C, which represents the state of the computation at the time of evaluation, we get a procedure M and and similarly an argument N. Then evaluating these converges: $((MN)\mathcal{D}, \sigma) \Downarrow (V, \sigma')$ and $(\mathcal{D}, \xi, \sigma')[x : [V\mathcal{D}^{-1}]] \models C'$.

Several things are worth pointing out. $\{C\}e \bullet e' \searrow u\{C'\}$ is asserted of the current model $\mathcal{M}$ which represents the current state of the current program's execution. The evaluation formula describes the behaviour of e's denote throughout the entire course of the computation, starting from the present, as given by $\mathcal{M}$. Hence σ may not coincide with $\mathcal{M}$'s store. On the other hand environments don't change by computation, so ξ would be the same as $\mathcal{M}$'s. We have chosen to keep the distinction constant, too, that is $\mathcal{D}$ is also the distinction found in $\mathcal{M}$. This is sound, cf. Theorem 8.6, but one could conceivably have made a different choice by allowing $\mathcal{D}$ to fuse more names. The semantic consequences of such an alternative approach are unclear. The evaluation above is of $((MN)\mathcal{D}, \sigma)$, not of (MN, σ) because σ is defined on identicals, unlike M and N. Conversely, V's identifiers are identicals and must be defused before plugged into $[x : \cdot]$ which expects semi-closed values over the carrier set of these identicals.

B Auxiliary Predicates for Quicksort

The single permutation predicate is given by:

$$\begin{aligned}\mathsf{SPerm}(ablr) \quad \stackrel{\text{def}}{=} \quad & \exists i, j.(l \le i, j \le r \wedge\ !a[i] =\!!b[j] \wedge\ !a[j] =\!!b[i]\ \wedge \\ & \forall h.(\ (l \le h \le r\ \wedge\ h \notin \{i, j\})\ \supset\ !a[h] =\!!b[h])\ )\ \wedge \\ & \mathsf{size}(a) = \mathsf{size}(b)\ \wedge\ \mathsf{Dist}\end{aligned}$$

The result of permuting n times is then given by:

$$\begin{aligned}\mathsf{Perm}^{(0)}(ablr) \quad & \stackrel{\text{def}}{=} \quad \mathsf{Eq}(ablr) \\ \mathsf{Perm}^{(n+1)}(ablr) \quad & \stackrel{\text{def}}{=} \quad \exists a'.(\mathsf{Perm}^{(n)}(aa'lr)\ \wedge\ \mathsf{SPerm}(a'blr)\ \wedge\ \mathsf{Dist}[a'/b])\end{aligned}$$

Note that, as in $\mathsf{Eq}(ablr)$, the permutation predicate includes the assertion on (full) distinction.