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 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 entities 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 $!x$ 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 ML or C programs, demanding general reasoning principles that can precisely describe and verify programs' behaviour in the presence of aliasing. However a satisfactory solution to this problem is hard to reach: for example the proof rules of Hoare's original program logic [17], one of the representative reasoning methods, are sound only when aliasing is absent (see [2, 7]: we shall illustrate this point in detail in Section 4).

Resuming a study by Cartwright and Oppen from 25 years ago [5, 6], which shares some of its approach with [24, 35, 36], the present paper introduces a simple and tractable compositional program logic for general aliasing and imperative higher-order functions. A central observation in [5, 6] is that (in)equations over names, simple as they may seem, are expressive enough to describe general aliasing in first-order procedural languages, provided that we distinguish between reference names x and the corresponding content $!x$ in assertions. In particular, Cartwright and Oppen have shown that alias robust substitution $\{\!|e/!x|\!\}$, 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, albeit at the expense of an increase in formula size. This immediately gives us the following semantic version of Hoare's assignment axiom:

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

The rule subsumes the original axiom but is now alias-robust. As clear evidence of descriptive power of their approach, Cartwright and Oppen also 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.

Cartwright and Oppen's work, remarkable as it is, still begs 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 awkward because inductive decomposition of $\{\!|e/!x|\!\}$ into (in)equations is the only available syntactic tool in [5, 6]. For example, calculating such a simple equivalence as $C\{\!|\mathsf{c}/!x|\!\}\{\!|e/!x|\!\} \equiv C\{\!|\mathsf{c}/!x|\!\}$ for general C, with c being a constant, cannot be done directly, but demands meta-logical reasoning, induction on C, or an appeal to semantic means. Because such calculation is a key part of program proving (cf. [17]), practical usage of the approach in [5, 6] becomes unclear. A second and related problem is the lack of structured reasoning principles for deriving precise description of program behaviour with aliasing. This makes reasoning hard, because how programs interact through shared references depends on the details of how references are aliased. Finally, the logic in [5, 6] does not offer a general treatment of higher-order procedures and data types.

We address these technical issues by augmenting the logic for imperative higher-order functions studied in [22] with a pair of mutually dual logical primitives. They offer an effective middle layer for reasoning about aliasing, with a clear logical status. The existential part of the primitives, written $\langle !x\rangle C$, is defined by the following equivalence:

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

The defining clause says: "for some possible content of a reference named x, $\mathcal{M}$ satisfies C" (which is *not* about the present state, but about a possible world which can have a different 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 several notable aspects of these primitives. First, semantic substitution now has a syntactic representation 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. Second, these operators are equipped with simple axioms (Figure 2 in Section 7), which follow those of standard modal operators [3]. In contrast to semantic substitution, these axioms enable well-structured syntactic calculation of validity, as we shall demonstrate later. Their use leads to a level of generality not know before: we can do with a single rule of inference for the general form of assignment $M := N$, where M and N (of appropriate types) may contain calls to higher-order procedures and arbitrary data structures, as discussed in Section 7. When combined with other standard logical primitives, especially (in)equations over reference names, content quantification offers basic building blocks for structured reasoning on programs with aliasing at a high-level of abstraction. As a result we obtain derived proof rules and axioms which offer a powerful tool for compositionally reasoning about programs with aliasing, on which we shall discuss in detail in Section 7. Finally these merits come, somewhat paradoxically, without any additional expressive power: any formula which contains content quantifications can be translated into one without them up to logical equivalence, as we shall prove in Section 7. The translation reveals close, organic connection among content quantifications, name equations and semantic substitutions, offering suggestions on their usage in practical reasoning.

The descriptive power of the logic is formally clarified in Section 8 by showing the assertion language to be *observationally complete* in the sense that two programs are contextually indistinguishable exactly when they satisfy the same set of assertions.

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 presents several basic technical results for the present 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 [27].

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 [33], given by the following grammar.

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

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. [14, 33]. 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 [22], 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. $\Gamma, \Gamma' \dots$ range over bases. $\mathsf{dom}(\Gamma)$ denotes the domain of Γ. The typing rules are standard [33] 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 [28]).

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 $\Delta, \dots$ 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 the environment basis).

We often call variables of reference types *reference names* or simply *referenc es*. 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 [14, 33]. 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):

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

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 we regard contexts are always typed. We use the following subset of typed contexts in the subsequent technical development.

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

Note a semi-closing context always has the form $C[\ \cdot\ ]^{\Delta';\alpha'}_{\Gamma;\Delta;\alpha}$ with $\Delta' \supset \Delta$. An example of semi-closing 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. Anticipating the development in Section 3, the above definition can be replaced by saying that $\cong$ is the maximum typed relation satisfying $\Gamma \vdash M_1 \cong M_2 : \alpha$ if and only if:

$$\forall\sigma,\ C[\ \cdot\ ].\ (\ (C[M_1],\sigma)\Downarrow \quad \Leftrightarrow \quad (C[M_2],\sigma)\Downarrow\ ). \tag{2.2}$$

This in turn can be rephrased by saying $\cong$ is the maximum typed relation satisfying $\Gamma \vdash M_1 \cong M_2 : \alpha$ if and only if:

$$\forall\delta,\ \sigma,\ \text{semi-closing}\ C[\ \cdot\ ].\ (\ (C[M_1\delta],\sigma)\Downarrow\ \Leftrightarrow\ (C[M_2\delta],\sigma)\Downarrow), \tag{2.3}$$

where δ ranges over all well-typed substitution of reference names for reference names (which may not be injective). The last characterisation says all experiments we need to inspect the contextual behaviour of programs are combination of semi-closing contexts and possible ways to collapse reference names.

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 [29], rather than introducing an additional set of location labels (as in the dynamic semantics of ML [30]). 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 $\mathcal{D}$). 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

$M \not\cong ()$ 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 semi-closing $C[\cdot]$ 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 §2.3, Page 7) 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 $\mathcal{D}$. Now we give an equivalent but simpler definition.

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) in Section 2.3, Page 7.

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 $\mathcal{M}^{\Gamma;\Delta \cdot x:\mathsf{Ref}(\alpha)} \stackrel{\text{def}}{=} (\mathcal{D}, \xi, \sigma \cdot \mathbf{i} \mapsto \mathbf{w})$ 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. □

Definition 3.11 (extension) Let $\mathcal{M}_i^{\Gamma_i,\Delta_i} \stackrel{\text{def}}{=} (\mathcal{D}_i, \xi_i, \sigma_i)$ $(i = 1,2)$ be models. Then we say $\mathcal{M}_2$ *extends* $\mathcal{M}_1$, written $\mathcal{M}_1 \leq \mathcal{M}_2$, iff: (1) $\Gamma_1 \subset \Gamma_2$; (2) $\Delta_1 \subset \Delta_2$; (3) $\mathcal{D}_1 \leq \mathcal{D}_1$ (cf. Definition 3.1); (4) $\xi_1(x) = \xi_2(x)$ for each $x \in \mathsf{dom}(\xi_1)$; and (5) $\sigma_1(x) = \sigma_2(x)$ for each $x \in \mathsf{dom}(\sigma_1)$.

Above we can neglect typing in the clauses (4) and (5) thanks to Proposition 3.10. Intuitively, $\mathcal{M} \leq \mathcal{M}'$ says that $\mathcal{M}'$ contains $\mathcal{M}$ as a submodel, and that all the information held in the former is consistent with that in the latter. Immediately $\leq$ is the partial order. The following notation is often useful.

Notation 3.12 $\mathcal{M}_1 \leq_{x:\alpha} \mathcal{M}_2$ stands for $\mathcal{M}_1 \leq \mathcal{M}_2$ and $\mathsf{dom}(\mathcal{M}_2) \backslash \mathsf{dom}(\mathcal{M}_1) = \{x\}$.

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 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.

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

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 [17] dictates that a valid judgment 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 C, ML and Haskell, that the need of assigning correct types to expressions driven us to strict differentiation between references and their content.

Aliasing and Assignment Axiom. 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.

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

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. [2, 7]. 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 §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 Introduction informally, but now with a clear technical content.

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

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 $\exists x.(C \wedge !x{=}e)$ since $!x$ may occur 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 let 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, rather than x itself, arriving at the existential content quantification $\langle !x \rangle C$, which says C holds under some (possibly hypothetical) content of x. Its universal counterpart $[!x]C$ arises precisely 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 tract-ably 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 [27, § 2.8] extended with assertions for evaluation and those for quantification over store content (the latter is the only addition to the logic in [22] and is highlighted below).

$$e \ ::= \ x^{\alpha} \ | \ () \ | \ \mathsf{n} \ | \ \mathsf{b} \ | \ \mathsf{op}(\tilde{e}) \ | \ \langle e, e' \rangle \ | \ \pi_i(e) \ | \ \mathsf{inj}_i^{\alpha+\beta}(e) \ | \ !e$$

$$C \ ::= \ e = e' \ | \ \neg C \ | \ C \star C' \ | \ Qx^{\alpha}.C \ | \ \{C\}\, e \bullet e' \searrow x\, \{C'\} \ | \ \ | \ [!e]C \ | \ \langle !e \rangle C$$

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* [22], 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 [27, §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/postconditions of an evaluation formula are implicitly universally quantified). The need for these conditions is similar. We discuss the the case of 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:

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

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 *content substitutions*. We shall derive $\overline{C\{\!|e_2/!e_1|\!\}} \equiv C\{\!|e_2/!e_1|\!\}$ 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. [27, §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 Assertions

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.

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

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 [22]; (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 Ref(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 "$x := m$".

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

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 fcell, $!x = 2\ \wedge\ !x = m$. What this means is that the assertion

[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.

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 [27, §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 "$(!x = 2)\{\!|\, !y/m \,|\!\}$" 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 which is distinct from 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, rather than required as a side-condition.

Another place where combination of evaluation formulae and universal content becomes useful arises 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 §4. This is followed by the classical "swap" and then by assignment to a circular reference, all of which are substantially affected by aliasing. These specification are proved in Section 9.

We continue with the questionable double, whose definition is reproduced from §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 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:

$$\mathsf{Swap}_0(u) \quad \stackrel{\text{def}}{=} \quad \{!x = i \wedge !y = j\}u \bullet \langle x,y\rangle\{!x = j \wedge !y = i\}.$$

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:

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

Types are taken syntactically, so that the satisfaction for the new constructs are simply given as follows (let α be closed below).

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

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.

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

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 $\tilde{w}$.

We often call the first two implications *weakening* and *thinning* for located evaluation formulae. Located assertions are one of the central elements in the present logical theory.

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. As is standard [17], we shall assume that axioms, rules and theorems from propositional calculus, first-order logic with equality [27, §2.8], and formal number theory are freely available.[4]

We start from axioms for content quantification. In axiomatisations of conventional first-order quantification [27, §2], one stipulates the single inference rule which infers $\forall x.A$ from A provided x does not appear freely in assumptions, and one or two axioms (for example $\forall x.(A \supset B) \supset A \supset \forall x.B$ provided x does not occur in A). This is all we need, assuming existential quantification is treated as a derived construct. Here we instead follow the approach of standard modal operators [3], where logical equivalences and entailment involving modal operators are given explicitly. This presentation clarifies how the new modal operators and connectives/quantifiers interact, besides providing immediately useful laws of reasoning. An axiomatisation of content-quantifiers following the standard axiomatisation of quantification theory is outlined in Appendix B for reference. Via the elimination theorem discussed later, one can show that these two approaches lead to equipotent derivability.

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

$$
\begin{aligned}
[!x]\,C' \quad &\supset \quad [!x]\,((C' \supset C) \supset C) && (6.1)\\
[!x]\,C \wedge [!x]\,C' \quad &\equiv \quad [!x]\,(C \wedge C') && (6.2)\\
[!x]\,C \vee [!x]\,C' \quad &\supset \quad [!x]\,(C \vee C') && (6.3)\\
[!x]\,(C \vee C') \quad &\supset \quad [!x]\,C \vee \langle !x \rangle C' && (6.4)\\
[!x]\,C \quad &\supset \quad C && (6.5)
\end{aligned}
$$

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

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.

$$
\begin{aligned}
\forall y.[!x]\,C \quad &\supset \quad [!x]\,\forall y.C && (6.6)\\
[!y]\,[!x]\,C \quad &\supset \quad [!x]\,[!y]\,C && (6.7)\\
\langle !x \rangle\,[!x]\,C \quad &\supset \quad [!x]\,C && (6.8)\\
\forall x.!x \neq y \quad &\supset \quad \mathsf{F} && (6.9)
\end{aligned}
$$

Again they have dual versions. All these entailments are logical equivalences, with the reverse direction being derivable: for (6.6), 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

[4] Quantification theories and theories of equality involve non-trivial use of syntactic substitutions in deductions, whose precise treatment in the present logic was detailed in Section 5.2.

$\forall y.[!x]C$; (6.7) is already symmetric; finally the converse of (6.8) 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 (6.9) 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 is closely related with the modal operator. See Remark 6.8 for further illustration.

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

$$[!x]\,!x \neq y \quad \supset \quad \mathsf{F} \tag{6.10}$$
$$C^{-!x} \quad \supset \quad [!x]C \tag{6.11}$$

In both, the entailments are indeed logical equivalences. (6.10) is easily understood as an analogue of $\forall x.(x \neq y) \supset y \neq y$ ($\equiv \mathsf{F}$). In (6.11), we use the notation $C^{-!x}$ which means "any derefence in C cannot be about x", and is formally given as follows. Below we recall, in $\{C\}e \bullet e' \searrow z\{C'\}$, C and C' are its pre/post conditions.

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.

Definition 6.2 (syntactic $!x$-freedom) We say C is *syntactically $!x$-free*, written $C^{-!x}$, when C has the shape $\exists \tilde{r}.(C' \wedge \bigwedge_i r_i \neq x)$ where $!\tilde{r}$ exhaust active dereferences in C' of type α assuming x is typed $\mathsf{Ref}(\alpha)$ in C.

Mentioning the type of x in the last definition is not necessary since two terms of different types are never equated, but makes it easier to verify for syntactic $!x$-freedom. As simple examples, T and F are syntactically $!x$-free. Similarly $[!x]C$ and $\langle !x \rangle C$ are always syntactically $!x$-free. Another example is $x \neq y \wedge !y = 3$ in its slightly altered form, $\exists r.(!r = 3 \wedge y = r \wedge x \neq r)$ (which is easily transformed to $x \neq y \wedge !y = 3$ preserving $\equiv$: we often regard such formulae as being syntactically $!x$-free). However $!y = 3$ is *not* syntactically $!x$-free. We discuss significance of the semantic version of $!x$-freedom later. Intuitively, $C^{-!x}$ says C does not mention the content of x: thus (6.11) is nothing but the analogue of the standard entailment $A \supset \forall x.A$ when $x \notin \mathsf{fv}(A)$.

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

$$\neg[!x]C \quad \equiv \quad \langle !x \rangle \neg C \tag{6.12}$$
$$[!x]\,(!x = m \supset C) \quad \equiv \quad \langle !x \rangle\,(C \wedge\, !x = m) \tag{6.13}$$

(6.12) is standard. The second axiom (6.13), however, may not be, relating two dual quantifiers *without dualisation*: its origin lies in the following logical equivalence, briefly mentioned in the Introduction where m is fresh.

$$\forall x.(x = m \supset C) \quad \equiv \quad C[m/x] \quad \equiv \quad \exists x.(C \wedge x = m) \tag{6.14}$$

From (6.13) we immediately infer the equivalence between the two forms of logical substitutions introduced in Definition 5.4: $C\{\!|e'/!e|\!\} \equiv \overline{C\{\!|e'/!e|\!\}}$ for any C, e' and $!e$. The axiom (6.13) plays a fundamental role in the present theory.

Fig. 2 Axioms for Content Quantification.

(u0)	$[!x]C'$	$\supset$	$[!x]((C' \supset C) \supset C)$	(e0)	$\langle !x\rangle (C \wedge (C \supset C'))$	$\supset$	$\langle !x\rangle C'$
(u1)	$[!x]C \wedge [!x]C'$	$\equiv$	$[!x](C \wedge C')$	(e1)	$\langle !x\rangle (C \vee C')$	$\equiv$	$\langle !x\rangle C \vee \langle !x\rangle C'$
(u2)	$[!x]C \vee [!x]C'$	$\supset$	$[!x](C \vee C')$	(e2)	$\langle !x\rangle (C \wedge C')$	$\supset$	$\langle !x\rangle C \wedge \langle !x\rangle C'$
(u3)	$[!x](C \vee C')$	$\supset$	$[!x]C \vee \langle !x\rangle C'$	(e3)	$\langle !x\rangle C \wedge [!x]C'$	$\supset$	$\langle !x\rangle (C \wedge C')$
(u4)	$[!x]C$	$\supset$	C	(e4)	C	$\supset$	$\langle !x\rangle C$
(u5)	$\forall y.[!x]C$	$\supset$	$[!x]\forall y.C$	(e5)	$\langle !x\rangle \exists y.C$	$\supset$	$\exists y.\langle !x\rangle C$
(u6)	$\forall x.!x \neq y$	$\supset$	F	(e6)	T	$\supset$	$\exists x.!x = y$
(u7)	$[!y][!x]C$	$\supset$	$[!x][!y]C$	(e7)	$\langle !y\rangle \langle !x\rangle C$	$\supset$	$\langle !x\rangle \langle !y\rangle C$
(u8)	$\langle !x\rangle [!x]C$	$\supset$	$[!x]C$	(e8)	$\langle !x\rangle C$	$\supset$	$[!x]\langle !x\rangle C$
(u9)	$[!x]\,!x^{\mathsf{Ref}(\alpha)} \neq y^{\alpha}$	$\supset$	F	(e9)	T	$\supset$	$\langle !x\rangle\,!x^{\mathsf{Ref}(\alpha)} = y^{\alpha}$
(ua)	$C^{\text{-}!x}$	$\supset$	$[!x]C$	(ea)	$\langle !x\rangle C^{\text{-}!x}$	$\supset$	C

In (u5, c5), y and x are distinct symbols.

Figures 2 and 3 summarise the axioms. In Figure 2, the right-hand side presents the axioms for existential quantifications, which are simply the dual of those for universals. Figure 3 presents the self-dual axioms. Later in Section 8, we shall show that these axioms are complete in a certain sense.

Let us record some of the significant remaining logical laws for content quantifications which arise as theorems derivable from these axioms. They shed light on the nature of these primitives, as well as being useful in practice. We first 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 derivable 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 6.11, 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. However note C is $!x$-free iff $C \equiv C_0$ such that C_0 is syntactically $!x$-free.

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 (and unsatisfiable formulae) are $!x$-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.

Fig. 3 Axioms for Content Quantification (2): self-dual axioms.

$$
\begin{array}{llcl}
\text{(ue0)} & \neg[!x]\,C & \equiv & \langle !x\rangle\,\neg C \\
\text{(ue1)} & [!x]\,(!x = m \;\supset\; C) & \equiv & \langle !x\rangle\,(C \wedge\; !x = m)
\end{array}
$$

As we noted when the syntactic analogue of $!x$-freedom was discussed (cf. Definition 6.2), $!x$-freedom in C allows us to reach $[!x]\,C$ from C in the same way the lack of (free) occurrences of x in C allows us to reach $\forall x.C$ from C.

Since $[!x]\,C \supset C$ for any C by (6.5), we know C is $!x$-free if and only if $C \supset [!x]\,C$. This characterisation immediately gives us the axiom which was conspicuously missing from the preceding discussion:

$$[!x]\,C \quad\supset\quad [!x]\,[!x]\,C \tag{6.15}$$

The entailment is in fact a logical equivalence since $[!x]\,[!x]\,C \supset [!x]\,C$ by (6.5). Dually:

$$\langle !x\rangle\,\langle !x\rangle\,C \quad\supset\quad \langle !x\rangle\,C \tag{6.16}$$

which can again be strengthened to a logical equivalence. From these observations we know $\langle !x\rangle\,C \equiv C$ also characterises $!x$-freedom (which is often useful in practice) and that the converse of (6.8) 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:*

$$
\begin{aligned}
[!x]\,(C_1 \,\vee\, C_2) &\quad\equiv\quad C_1 \,\vee\, [!x]\,C_2 && (6.17)\\
\langle !x\rangle\,(C_1 \,\wedge\, C_2) &\quad\equiv\quad C_1 \,\wedge\, \langle !x\rangle\,C_2 && (6.18)\\
[!x]\,(C_1 \,\supset\, C_2) &\quad\equiv\quad C_1 \,\supset\, [!x]\,C_2. && (6.19)
\end{aligned}
$$

Proof By duality and since (6.19) merely rephrases (6.17), it suffices to derive (6.17).

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

For the other direction:

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

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

Note (6.19) 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,* $C[e/!x] \equiv C\{\!|e/!x|\!\}$.

Proof For (1):

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

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

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

For (3), the first statement:

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

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

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

For the other direction, again with m fresh:

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

As already noted, the results in (4) depend on $[!x]C$ being $!x$-free. Let x is of type $\mathsf{Ref}(\alpha)$. For simplicity assume there is only one active dereference of type α, say $!e$, in C which we write $C(!e)$ with the obvious meaning. Then we have, with r and z fresh:

$$\begin{array}{lcl} [!x]C & \equiv & \exists rz.[!x](C(z) \wedge r = e \wedge z = !r) \\ & \equiv & \exists rz.[!x]((r \neq x \wedge C(z) \wedge r = e \wedge z = !r) \vee (r = x \wedge C(z) \wedge r = e \wedge z = !r)) \\ & \equiv & \exists rz.((r \neq x \wedge C(z) \wedge r = e \wedge z = !r) \vee [!x](r = x \wedge C(z) \wedge r = e \wedge z = !r)) \\ & \equiv & \exists rz.((r \neq x \wedge C(z) \wedge r = e \wedge z = !r) \vee \mathsf{F}) \\ & \equiv & \exists r.(r \neq x \wedge \exists z.(C(z) \wedge r = e \wedge z = !r)) \end{array}$$

The argument easily extends to the general case. (5) is easy and omitted. For (6), one direction:

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

For the other direction:

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

In the first line the well-definedness of the syntactic substitution is important. □

Remark 6.8 The axiom (6.9) does not mention content quantification: however it may be regarded as a special case of the following rule:

$$\forall x.C^{\,\iota x} \quad \supset \quad \forall x.[!x]C^{\,\iota x} \tag{6.20}$$

where $C^{\iota x}$ indicates x is not aliased to other names in C. An assertion in which x is unalised is $!x = 3$; an assertion in which x can be aliased is $x = y \wedge !y = 1$. Semantically $C^{\iota x}$ indicates the following entailment:

$$(\xi \cdot x\tilde{y}\!:\!\mathbf{i},\, \sigma \cdot \mathbf{i} \mapsto \mathbf{v}) \models C \;\;\supset\;\; (\xi \cdot x\!:\!\mathbf{i}_x \cdot \tilde{y}\!:\!\mathbf{i}_y,\, \sigma[\mathbf{i}_{\tilde{y}}/\mathbf{i}] \cdot \mathbf{i}_x\mathbf{i}_{\tilde{y}} \mapsto \mathbf{v}) \models C \quad (\tilde{y} \neq \varepsilon) \tag{6.21}$$

where we write $\mathbf{i}_{\tilde{z}}$ etc. for the projection of $\mathbf{i}$ onto $\{\tilde{z}\}$ and $x\tilde{y}\!:\!\mathbf{i}$ indicates the obvious join of maps with the same target. (6.21) says that if C holds in a model in which x is aliased to other names then C also holds in its variant in which x is distinct from other names but which otherwise remains identical (without this condition, the rule (6.20) is invalid, for example, using its dual, we do not want to conclude $\exists x.(x = y \wedge !y = 1)$ from $\exists x.\langle !x\rangle (x = y \wedge !y = 1)$, which is a tautology). Under a sufficiently general syntactic approximation to (6.21), we may regard (6.20) as a basic form of the axiom. So far such an approximation is not known. From the viewpoint of sheer provability, however, the rule (6.9) and its dual are enough to derive such axioms through the elimination theorem given later.

Fig. 4 Axioms for Evaluation Formulae.

(e1)	$\{C_1\}x \bullet y \searrow z\{C\} \wedge \{C_2\}x \bullet y \searrow z\{C\}$	$\equiv$	$\{C_1 \vee C_2\}\, x \bullet y \searrow z\, \{C\}$
(e2)	$\{C\}x \bullet y \searrow z\{C_1\} \wedge \{C\}x \bullet y \searrow z\{C_2\}$	$\equiv$	$\{C\}\, x \bullet y \searrow z\, \{C_1 \wedge C_2\}$
(e3)	$\{\exists w^{\alpha}.C\}\, x \bullet y \searrow z\, \{C'^{-w}\}$	$\equiv$	$\forall w^{\alpha}.\{C\}\, x \bullet y \searrow z\, \{C'\}$
(e4)	$\{C^{-w}\}\, x \bullet y \searrow z\, \{\forall w^{\alpha}.C'\}$	$\equiv$	$\forall w^{\alpha}.\{C\}\, x \bullet y \searrow z\, \{C'\}$
(e5)	$\{A \wedge C\}\, x \bullet y \searrow z\, \{C'\}$	$\equiv$	$A \supset \{C\}\, x \bullet y \searrow z\, \{C'\}$
(e6)	$\{C\}\, x \bullet y \searrow z\, \{A^{-z} \supset C'\}$	$\supset$	$A \supset \{C\}\, x \bullet y \searrow z\, \{C'\}$
(e7)	$\{C\}x \bullet y \searrow z\{C'\}$	$\supset$	$\{C \wedge A\}x \bullet y \searrow z\{C' \wedge A\}$
(e8)	$[!\tilde{w}]\,(C \supset C_0) \wedge \{C_0\}x \bullet y \searrow z\{C'_0\} \wedge [!\tilde{w}]\,(C'_0 \supset C)$	$\supset$	$\{C\}\, x \bullet y \searrow z\, \{C'\}$
(e9)	$\{C\}x \bullet y \searrow z\{C'\}$	$\supset$	$[!u]\,\{C\}x \bullet y \searrow z\{C'\}$
(e0)	$\langle !x \rangle\, \{C\}y \bullet z \searrow w\{C'\}$	$\supset$	$\{C\}y \bullet z \searrow w\{C'\}$
(ext)	$\mathsf{Ext}^{\Delta;\alpha\Rightarrow\beta}(x,y)$	$\supset$	$x = y$

6.2 Axioms for Evaluation Formulae

The set of axioms for evaluation formulae are given in Figure 4. Most come from the axioms in [22]. We assume the following convention used throughout the paper.

Convention 6.9 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), §5.5 (which is closely related with Kleymann's strengthened consequence rule [25]). 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).

(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. §5.4).

In (ext), the extensionality formula augments the corresponding formulae for alias-free sublanguage in [22] with located assertions.

Definition 6.10 (extensionality formulae) Let $\Delta = \tilde{r} : \mathsf{Ref}(\tilde{\gamma})$.

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

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

Fig. 5 Axioms for Located Evaluation Formulae.

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

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 5, we list axioms for located assertions, which refine the original axioms in Figure 4, as well as adding two new axioms for manipulating write effects (we do not include (ext) since it is already about located assertions). 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), §5.5, Page 22. While (le7) is stronger than (e7), it is also true that (le7) is justifiable using (e7). For concreteness, take the assertion (5.14) in §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:

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

Now we use (e7) and get:

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

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

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

Fig. 6 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	$\supset$	$\mathtt{in}_1(x) \neq \mathtt{in}_2(y)$

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 4, even though the latter rules are subsumed by these refined rules for located assertions.

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 6. All axioms have already appeared in [22]: 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. [21]. 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:

$$\begin{array}{lll} (\text{types}) & \alpha ::= ... \mid \alpha[] \\ (\text{programs}) & M ::= ... \mid M[N] \end{array} \qquad \frac{-}{\Gamma \vdash a : \alpha[]} \qquad \frac{\Gamma \vdash M : \alpha[] \quad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash M[N] : \mathsf{Ref}(\alpha)}$$

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 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 is undefined. Terms and assertions are augmented accordingly:

$$e \quad ::= \quad \ldots \quad | \quad a \quad | \quad e[e'] \qquad\qquad C \quad ::= \quad \ldots \quad | \quad \mathsf{Size}(e_1^{\alpha[]}, e_2^{\mathsf{Nat}})$$

The added terms are typed in the obvious way. We add one axiom for arrays to ensure that an array of size n is made up of n distinct references.

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

Models need a corresponding restriction. 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.

7 Logic (3): Judgements and Proof Rules

7.1 Judgements and their Semantics

Following Hoare [17], 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*.[5] We say $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is *well-typed* iff (1) $\Gamma;\Delta \vdash M : \alpha$ and (2) $\Gamma';\Delta' \vdash C$ and $\Gamma' \cdot u : \alpha; \Delta' \vdash C'$ for $\Gamma' \supset \Gamma$ and $\Delta' \supset \Delta$ such that $\mathsf{dom}(\Gamma' \cup \Delta') \cap \{u\} = \emptyset$. *Henceforth we only treat judgements which are well-typed.* Following Convention 5.1 (5), the notation $\{C\}M\{C'\}$ stands for $\{C\}M :_u \{u = () \wedge C'\}$ such that (1) u is typed with Unit and (2) $u \notin \mathsf{fv}(C')$.

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. §2.3/§3.1. Formally we stipulate:

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 $\mathcal{M} \models C$ then $(M\xi, \sigma) \Downarrow (\mathbf{v}, \sigma')$ such that $(\mathcal{M} \cdot u : \mathbf{v},\ \sigma') \models C'$.

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

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

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 [22] except for a minimal and unavoidable refinement of the rule for assignment, apart from adaptation for the generalised syntax (in dereference, where $!M$ is allowed in addition to $!x$, and in assignment, where $M := N$ is allowed in addition to $x := N$) This is in accordance with our logical language, which increments that in [22] 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 we have been investigating.

Following [22], 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 6.9, $A, A', B, B', \ldots$ 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 7 (the difference from [22] 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 7 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 [22], we illustrate two rules for imperative constructs, $[Deref]$ and $[Assign]$ in the following.

Fig. 7 Proof Rules (1): compositional rules.)

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

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

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

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

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

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

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

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

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

[*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 [22]).

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

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:

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

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 the second inference 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 result in 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 $\{C\}M := N :_u \{u = () \wedge C'\}$. The rule is again easily understood by contrasting with the corresponding rule for the non-aliased sublanguage in [22]. The rule reads (using the abbreviation for unit noted above):

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

There are two differences from [*Assign*] in Figure 7. 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 logical one (cf.Section 5.3). Let us first take off the second restriction, obtaining:

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

where the highlighted part is the difference from [*AssignOrg*]. Clearly [*AssignVar*] is derivable from [*Assign*] through [*Var*]. As an example of its use, we can infer:

$$\frac{\dfrac{1.\ \{2=2\}\ 2 :_m \{m=2\} \qquad \text{(Const)}}{2.\ \{\mathsf{T}\}\ 2 :_m \{\langle !x\rangle(!x=2 \wedge !x=m)\} \qquad \text{(Conseq)}}}{3.\ \{\mathsf{T}\}\ x := 2\,\{!x=2\} \qquad \text{(AssignVar)}}$$

Above Line 2 uses:

$$m=2 \ \equiv\ (\langle !x\rangle\, !x=2) \wedge m=2 \ \equiv\ \langle !x\rangle(!x=2 \wedge m=2) \ \equiv\ \langle !x\rangle(!x=2 \wedge !x=m)$$

where the first equivalence is by Axiom (e9), the second by Proposition 6.6.

In many programs, it is often the case that both sides of the assignment are expressions which are nevertheless simple in the sense that they do not contgain calls to procedures or abstractions, as in the following judgement:

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

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:

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

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).

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

Above the use of e as a program indicates that it is at the same time a term in the logic and a program in our programming language. 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. [2]) arises from [*AssignSimple*] as such a special rule. Using this rule, we can derive (7.4) as follows.

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

Line 2 uses:

$$\begin{array}{ll} x = y \wedge Even(!!y) \wedge Odd(!!y+1) & \\ \supset \quad m = !x = !y \wedge Odd(!!y+1) & \\ \equiv \quad \langle !!x \rangle (!x = !y \wedge Odd(!!y+1) \wedge !!x = !!y+1) & \text{(Prop. 6.7, (6.18))} \\ \supset \quad \langle !!x \rangle (Odd(!!x) \wedge Odd(!!y) \wedge !!x = !!y+1) & \end{array}$$

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 7. A simple example of its usage follows.

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

Line 3 is derived as

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

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 $\xi_0 = \xi \cdot m : \mathbf{i}$.

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

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).

Fig. 8 Structural Rules.

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

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

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

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

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

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

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

In [*Consequence-Aux*], we set $!\tilde{e}$ (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 $\tilde{i}$.

7.3 Proof Rules (2): Structural Rules

In Figure 8, 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 [25]: 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 them 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 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 §7.1. We set $\models \{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\} \mathbin{@} \tilde{e}$ holds iff

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

holds, with y and i fresh and distinct and typed as $\mathsf{Ref}(\mathrm{X})$ and X respectively (cf. §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 satisfaction relation, cf. Section 7.1, so that its meaning closely corresponds to that of located assertions.[6]

Example 7.6 (located judgement)

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 $M \stackrel{\text{def}}{=} \mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ x \times f(x-1)$. Then we have

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

   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:

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

   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.

Fig. 9 Derivable proof rules with located judgements.

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

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

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

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

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

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

$$[\mathit{Pair}]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\}@\tilde{e_i} \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1,m_2\rangle/u]\}@\tilde{e_2}}{\{C\}\ \langle M_1,M_2\rangle :_u \{C'\}@\tilde{e_1}\tilde{e_2}} \qquad [\mathit{Proj}_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}@\tilde{e}}{\{C\}\ \pi_1(M) :_u \{C'\}@\tilde{e}}$$

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

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

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

The proof rules for located judgements are given in Figure 9 (for compositional rules) and Figure 10 (for structural rules). The former closely follows Figure 7. In [*Assign*], we demand that C_0 says m (the target of writing) should be in the write effect. The set membership notation "$\in$" is understood to denote a disjunction of equations. Among the structural rules in Figure 10, there are three fundamental rules which deserve some illustration.

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

Fig. 10 Derivable structural rules with modified sets.

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

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

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

$$[Aux_\exists]\ \frac{\{C\}\ M :_u \{C'^{-i}\}@\tilde{e}}{\{\exists i.C\}\ M :_u \{C'\}@\tilde{e}} \qquad [Aux_\forall]\ \frac{\{C^{-i}\}\ M :_u \{C'\}@\tilde{e}}{\{C\}\ M :_u \{\forall i.C'\}@\tilde{e}}$$

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

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

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

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

$$\dfrac{\dfrac{1.\quad (!x = i)\{\!|!x/!x|\!\} \;\equiv\; !x = i \;\supset\; x \in \{x\}}{2.\quad \{!x = i\}\; x := !x\; \{!x = i\} \;@\; x \quad \text{(Assign-Simple)}}}{3.\quad \{\mathsf{T}\}\; x := !x\; \{\mathsf{T}\} \;@\; \emptyset \qquad\qquad \text{(Thinning)}}$$

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*] Recall that $C_0^{-!\tilde{x}}$ indicates C_0's being syntactically $!x$-free (cf. Definition 6.2). Since C is $!x$-free (cf. Definition 6.3) iff $C \equiv C_0$ such that C_0 is syntactically $!x$-free, we may equivalently write this rule as:

$$[\mathit{Invariance}]\,\frac{\{C\}\; M :_u \{C'\} @\tilde{e} \quad C_0 \text{ is } !x\text{-free}}{\{C \;\wedge\; C_0\}\; M :_u \{C' \wedge C_0\} @\tilde{e}} \tag{7.6}$$

which is equivalently to:

$$[\mathit{Invariance}]\,\frac{\{C\}\; M :_u \{C'\} @\tilde{e}}{\{C \;\wedge\; [!\tilde{e}]C_0\}\; M :_u \{C' \wedge [!\tilde{e}]\} @\tilde{e}} \tag{7.7}$$

since, by Proposition 6.7 (4) we have: C is $!x$-free iff $C \equiv [!x]C'$ for some C'. This last presentation is sometimes useful since it says that whenever we wish to add an invariance C_0 to a located judgement with a write set $\tilde{e}$, we can always add $[!\tilde{e}]C$, without loss of generality or safety.

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

$$\frac{\{C\}\; P\; \{C'\} \quad P \text{ does not touch variables in } C_0}{\{C \wedge C_0\}\; P\; \{C' \wedge C_0\}} \tag{7.8}$$

and the invariance rule for non-located judgements in Figure 8 (from [22]):

$$\frac{\{C\}\; M :_u \;\{C'\}}{\{C \wedge A\}\; M :_u \;\{C' \wedge A\}} \tag{7.9}$$

The rule may also be regarded as an analogue of a similar rule by Reynolds, O'Hearn and others in [31, 34]. See Section 10 for full technical comparisons. Since a weakened stateless formula A in (7.9) is by definition $!x$-free for any x, [*Invariance*] above subsumes (7.9), except for now using located judgements. On the other hand, [*Invariance*] is justifiable using 7.9), cf. Section 6.2. 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. Another use of located judgements is when we prove observational completeness, one of the basic results about our logic, to be discussed in Section 8.

Fig. 11 Located Proof rules for Imperative Idioms.

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

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

$$[\mathit{IfThenSimple}]\ \frac{C_0 \wedge e\ \supset\ C' \quad \{C \wedge e\}\ M\ \{C'\}@\tilde{g}}{\{C\}\ \mathtt{if}\ e\ \mathtt{then}\ M\ \{C'\}@\tilde{g}}$$

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

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

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

7.5 Proof Rules for Imperative Idioms

Figure 11 lists several located proof rules for basic imperative idioms one may often encounter in practice.

1. The initial four assignment rules are directly derivable from the general assignment rule in Figure 9.
2. The next two rules for the one-branch conditional are also easily derivable from the general conditional rule in Figure 9. In [*IfThenSimple*], we use the fact that a term e of a boolean type becomes a formula by a translation $e = \mathsf{t}$ (in many cases the translation itself is not necessary).
3. The rules for the while loop augments the standard total correctness proof rule for the while loop by Floyd [11]. In both rules, e' (of Nat-type) functions as an index of the loop, which should be decremented at each step. In [*WhileSimple*], which is as in [11] except it is augmented with a write effect, the guard is a simple expression. In [*While*], we generalise the original rule so that the guard is a general program, possibly with a side effect (which however does not add 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 [22, §7.1].

We can similarly consider rules for "do-while" commands, as well as assignment rules specialised to specific data types. We shall see one such example in the case of arrays next.

Fig. 12 Located Proof Rules for Arrays.

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

$$[ArraySimple]\ \frac{C[\ e[e']/u\ ] \ \supset\ \exists j.(0 \leq e' < j \ \wedge\ \mathsf{Size}(e, j))}{\{C[\ e[e']/u\ ]\}\ e[e'] :_u \{C\}\ @\ \emptyset}$$

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, we can for example think of a product component of which is a reference which stores an array whose components are procedures, 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 12, 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. [2]. The second rule in Figure 12 is the simple and derivable version of [*Array*] which is often useful. Below we give the reading of [*AssignArray*].

> *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:

$$[AssignArray]\frac{C[\ !a[e]\,/\,e'\ ] \ \ \supset\ \ \exists i.0 \leq e < i \ \wedge\ \mathsf{Size}(a, i)}{\{C\{\!|\,e'\,/\,!a[e]\,|\!\}\}\ a[e] := e'\ \{C\}}$$

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

$$C\{\!|\,e'\,/\,!a[e]\,|\!\} \quad \stackrel{\text{def}}{=} \quad \exists m.(\langle !\,a[e]\,\rangle\,(C \wedge !a[e] = m) \ \wedge\ m = e') \qquad (7.10)$$

In the left-hand side of (7.10), 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 [2, 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 [2] 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 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 it precise the logical status of these modal operators; in particular, semantically, we now know they add no more complexity than 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, those 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.

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

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

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

hence done. □

The following Lemma is from Cartwright and Oppen [6]. 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 $\mathcal{Q} \in \{\forall,\exists\}$.*

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

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

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

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

For $\forall$:

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

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 axiom. The fourth line is immediate from (e9) and (e0) of Figure 4 as well as the standard axiom for the existential quantification. the final line is immediate from (ua) and (ea). $\square$

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:*

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

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. $\square$

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 the main result of this section.

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:*

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

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 $[!x]\,!r = z \equiv x \neq r$. □

This formally shows that it is redundant to mention the content quantification $[!x]$ in Definition 6.3.

We conclude this section 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 [22, §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:

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

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.

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

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

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

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 $\mathtt{t}$ and $\mathtt{f}$. 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, 5 and 6 are valid.*

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

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

For (u1):

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

For (u2):

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

For (u3):

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

For (u4):

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

For (u5):

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

For (u6):

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

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 $\xi(x) \neq \mathbf{i}$:

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

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).

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

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 [15], 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, again restricted to a semi-closed value, 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 [22]. involving the following three steps.

Step 1: We introduce a variant of *finite canonical forms* (FCFs) [1, 20, 23] 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.

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. $\mathsf{fv}(C) \backslash \mathsf{dom}(\Gamma)$).

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

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

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

which means, through the β_V-equality:

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

where we set $W \stackrel{\text{def}}{=} \lambda x.C[x]$. Note the convergence in $(WM_1,\ \tilde{r} \mapsto \tilde{V}) \Downarrow$ takes, by the very definition, only a finite number of reductions. Let it be n. Then (occurrences of) λ-abstractions in W and $\tilde{V}$ can only be applied up to n-times, similarly for other destructors. Using this, we transform these programs into FCF values maintaining the convergence property while being made less defined (we leave the details to [22, 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 [22, §6 and Appendix A]. □

Fig. 13 Proof Rules for Characteristic Assertions of FCFs with Aliasing

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

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

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

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

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

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

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

8.5 Observational Completeness (3): Deriving CAPs for FCFs

In Figure 13, 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 r and j being fresh, 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 [22, §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 would be, and whatever the content of references, a program changes no state and results in a name x.

   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 [22, §6.6] instead use type information for the same end).

2. The rule for the let-application follows the corresponding rule in [22, §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 [22, §6.6] is how to stipulate the "current state". For this purpose it uses a fresh (auxiliary) reference name and its content, which, by the semantics of auxiliary names, can be instantiated into arbitrary 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 [22, §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,

$$F' \stackrel{\text{def}}{=} \texttt{case}\, x\, \texttt{of}\ \langle \texttt{n}_\texttt{i} : F_i \rangle_i, \quad C \stackrel{\text{def}}{=} \vee_i (x = y_i \wedge C_i), \quad C' \stackrel{\text{def}}{=} \vee_i (x{=}y_i \wedge C'_i).$$

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:

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

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

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

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

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

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

(Let-Application) 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 $\tilde{r} = r_1..r_i..r_n$ and $\tilde{j} = j_1..j_i..j_n$ $(n \geq 0)$:

$$\begin{array}{rcll} F' & \stackrel{\text{def}}{=} & \texttt{let}\, x = fU\ \texttt{in}\, F & (8.1) \\ \xi_0 & = & \tilde{y} : \tilde{S} & (8.2) \\ \xi_1 & = & \tilde{j} : \tilde{V} & (8.3) \\ \xi & = & \xi_0 \cdot f : W & (8.4) \\ \sigma & = & \tilde{r} \mapsto \tilde{V} & (8.5) \\ C_0 & \stackrel{\text{def}}{=} & \forall r', j'.\ (\,!r' = j'\ \supset\ \forall z.\{A \wedge !r' = j'\} f \bullet z \searrow x\{C\}\,) & (8.6) \\ C_1 & \stackrel{\text{def}}{=} & \forall \tilde{j}.\ (\,!\tilde{r} = \tilde{j}\ \supset\ \forall z.\{A \wedge !\tilde{r} = \tilde{j}\} f \bullet z \searrow x\{C\}\,) & (8.7) \end{array}$$

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

First we show C_0 is an MTC for F'. First notice:

$$(\xi, \sigma) \models C_0 \quad \supset \quad (\xi, \sigma) \models C_1 \tag{8.8}$$

since

$$C_0 \quad \supset \quad (!r_i = j_i \supset \forall z.\{A \wedge !r_i = j_i\} f \bullet z \searrow x\{C\})$$

for each $1 \leq i \leq n$, we have:

$$C_0 \quad \supset \quad (\bigwedge_i !r_i = j_i) \supset \bigwedge_i \forall z.\{A \wedge !r_i = j_i\} f \bullet z \searrow x\{C\})$$

But we have:

$$\bigwedge_i \forall z.\{A \wedge !r_i = j_i\} f \bullet z \searrow x\{C\}) \quad \equiv \quad \forall z.\{A \wedge !\tilde{r} = \tilde{j}\} f \bullet z \searrow x\{C\})$$

(6.22), Section 6.2, Page 32, hence as required. On the other hand, since neither W nor U has any free name outside of $\mathsf{dom}(\Gamma)$ hence we can neglect the range of r' outside of $\tilde{r}$, obtaining:

$$(\xi, \sigma) \models C_1 \quad \supset \quad (\xi, \sigma) \models C_2 \tag{8.9}$$

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

$$\xi_0 \cdot z : U\xi_0 \models A \tag{8.10}$$

We first show $(\xi, \sigma) \models C_0$ implies $(F'\xi, \sigma) \Downarrow$.

$$\begin{array}{lll}
(\xi,\, \sigma) \models C_0 & & \\
\Rightarrow & (\xi,\, \sigma) \models C_1 & (8.8) \\
\Rightarrow & (\xi \cdot \xi_1,\, \sigma) \models (\, !\tilde{r} = \tilde{j} \;\supset\; \forall z.\{A \wedge !\tilde{r} = \tilde{j}\} f \bullet z \searrow x\{C\}\,) & (\forall \text{ inst}) \\
\Rightarrow & (\xi \cdot \xi_1,\, \sigma) \models \forall z.\{A \wedge !\tilde{r} = \tilde{j}\} f \bullet z \searrow x\{C\}\,) & (8.5) \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \;\supset\; (z : U_1 \cdot \xi_0 \cdot \xi_1 \cdot \xi,\, \sigma) \models \{!\tilde{r} = \tilde{j}\} f \bullet z \searrow x\{C\} \wedge !\tilde{r} = \tilde{j} & \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \;\supset\; (\xi \cdot x : WU_1,\, \sigma) \Downarrow (\xi \cdot x : S_1,\, \sigma_1') \models C & \\
\Rightarrow & \forall U_1 \sqsupseteq U\xi_0 \;\supset\; (\xi \cdot x : WU_1,\, \sigma) \Downarrow (\xi \cdot x : S_1,\, \sigma_1') \models C & (\text{IH2}) \\
\Rightarrow & (\xi_0 \cdot x : WU,\, \sigma) \Downarrow (\xi \cdot x : S,\, \sigma) \models C & \\
\Rightarrow & (F'\xi,\, \sigma) \Downarrow & (\text{IH1})
\end{array}$$

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

$$\begin{array}{lll}
(F'\xi, \sigma) \Downarrow & & \\
\Rightarrow & (\xi_0 \cdot x : WU,\, \sigma) \Downarrow (\xi \cdot x : S,\, \sigma) \models C & (\text{IH1}) \\
\Rightarrow & \forall U_1 \sqsupseteq U\xi_0 \;\supset\; (\xi \cdot x : WU_1,\, \sigma) \Downarrow (\xi \cdot x : S_1,\, \sigma_1') \models C & (\text{IH2}) \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \;\supset\; (\xi \cdot x : WU_1,\, \sigma) \Downarrow (\xi \cdot x : S_1,\, \sigma_1') \models C & \\
\Rightarrow & z : U_1 \cdot \xi_0 \models A \;\supset\; (z : U_1 \cdot \xi,\, \sigma) \models^{I \cdot \tilde{j} : \tilde{V}} \{!\tilde{r} = \tilde{j}\} f \bullet z \searrow x\{C\} \wedge !\tilde{r} = \tilde{j} & \\
\Rightarrow & (\xi \cdot \xi_1,\, \sigma) \models \{A \wedge !\tilde{r} : \tilde{j}\} f \bullet z \searrow x\{C\} \wedge !\tilde{r} = \tilde{j} & (\text{e5}) \\
\Rightarrow & (\xi \cdot \xi_1,\, \sigma) \models !\tilde{r} = \tilde{j} \supset \{A \wedge !\tilde{r} : \tilde{j}\} f \bullet z \searrow x\{C\} \wedge !\tilde{r} = \tilde{j} & \\
\Rightarrow & \forall \tilde{V}'.\, (\, (\xi \cdot \tilde{j} : \tilde{V}',\, \sigma) \models \; !\tilde{r} = \tilde{j} \;\supset\; \{A \wedge !\tilde{r} : \tilde{j}\} f \bullet z \searrow x\{C\} \wedge !\tilde{r} = \tilde{j}\,) & (8.5) \\
\Rightarrow & (\xi,\, \sigma) \models C_1 & \\
\Rightarrow & (\xi,\, \sigma) \models C_0 & (8.9)
\end{array}$$

The reader can observe all inferences are the inverse of the first part. Separated presentation may however be more intuitive.

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

$$\{C_0 \wedge E\} M :_u \{C'\}. \tag{8.11}$$

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.

$$M_0 \stackrel{\text{def}}{=} \mathtt{let}\ \tilde{z} = !\tilde{r}\ \mathtt{in}\ \mathtt{let}\ x = yU\ \mathtt{in}\ (\tilde{r} := \tilde{z}\ ;\ M)$$
$$F_0' \stackrel{\text{def}}{=} \mathtt{let}\ \tilde{z} = !\tilde{r}\ \mathtt{in}\ \mathtt{let}\ x = yU\ \mathtt{in}\ F$$

By checking the reduction we have:

$$M \cong M_0 \qquad \text{and} \qquad F' \cong F_0. \tag{8.12}$$

Further by induction hypothesis on F, we have:

$$F \quad \sqsubseteq \quad \tilde{r} := \tilde{z}\ ;\ M \tag{8.13}$$

From (8.13) and precongruency of $\sqsubseteq$ we obtain:

$$F_0' \quad \sqsubseteq \quad M_0. \tag{8.14}$$

By (8.12) and (8.14) we obtain

$$M \quad \cong \quad M_0 \quad \sqsubseteq \quad F_0' \quad \cong \quad F'.$$

By $\cong \circ \sqsubseteq \circ \cong \subset \sqsubseteq$, this means $M \sqsubseteq F'$, as required. □

We conclude this section with the establishment of observational completeness. We first define the standard logical equivalence, cf. [15].

Definition 8.17 (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. The proof is given below for reference, though it is identical with the one given in [22, §6.7].

Theorem 8.18 *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 $M_1 \not\cong M_2$. 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,

$$(FM_1, \tilde{r} \mapsto \tilde{U}) \Downarrow \qquad \text{and} \qquad (FM_2, \tilde{r} \mapsto \tilde{U}) \Uparrow. \tag{8.15}$$

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:

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

But by (8.15) we have

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

that is $M_1 \ncong_{\mathcal{L}} M_2$, a contradiction. Thus we conclude $M_1 \cong M_2$, as required. $\square$

Fig. 14 Questionable Double with added assertions.

$\{u : T\}$
```
\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
```
$\{\forall x, y.!x = i \wedge !y = j \wedge x \neq y u(x,y)!x = 2 \cdot i \wedge !y = 2 \cdot j\}$

9 Reasoning Examples

This section revisits the programming examples discussed in Introduction and present the derivation of their specifications. We start with two simple examples, double? and `swap`, already encountered in the introduction. Then look at reasoning about circular data. We conclude by proving the correctness of `qsort`.

9.1 Questionable Double

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 are going to 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 $2 \times i$.

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

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:

$$\begin{array}{lll} x \neq y \wedge\ !x = i \wedge\ !y = j & & \\ \quad \equiv & \exists m.(x \neq y \wedge\ !x = i \wedge\ !y = j\ \wedge\ m = 2i) & \\ \quad \supset & \exists m.(\langle !x \rangle (x \neq y \wedge\ !x = 2i \wedge\ !y = j\ \wedge\ m = 2i)\ \wedge\ m = 2i \wedge\ !x = i) & \text{(Pro. 6.6 (6.18))} \\ \quad \supset & \exists m.(\langle !x \rangle (x \neq y \wedge\ !x = 2i \wedge\ !y = j\ \wedge\ m = !x)\ \wedge\ m = !x + !x) & \end{array}$$

The reasoning for (9.3) is identical.

$$
\begin{array}{lll}
x \neq y \wedge\ !x = 2i \wedge\ !y = j & & \\
\quad \equiv & \exists m.(x \neq y \wedge\ !x = 2i \wedge\ !y = j \ \wedge\ m = 2j) & \\
\quad \supset & \exists m.(\langle !y\rangle (\,!x = 2i \wedge\ !y = 2j \ \wedge\ m = 2j) \ \wedge\ m = 2j \wedge\ !y = j) & \text{(Pro. 6.6 (6.18))} \\
\quad \supset & \exists m.(\langle !y\rangle (\,!x = 2i \wedge\ !y = 2j \ \wedge\ m = !y) \ \wedge\ m = !y + !y) &
\end{array}
$$

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.	$\{(x \neq y \wedge !x = 2i \wedge\ !y = j)\{\!\|\,!x{+}!x/!x\|\!\}\}\ x := !x{+}!x\ \{x \neq y \wedge !x = 2i \wedge\ !y = j\}$	(AssignV)
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.	$\{(!x{=}2i \wedge !y{=}2j)\{\!\|\,!y{+}!y/!y\|\!\}\}\ y := !y{+}!y\ \{!x{=}2i \wedge !y{=}2j\}$	(AssignV)
6.	$\{x \neq y \wedge !x = 2i \wedge\ !y = j\}\ y := !y{+}!y\ \{!x{=}2i \wedge !y{=}2j\}$	(4, 5, Consequence)
7.	$\{x \neq y \wedge !x = i \wedge\ !y = j\}\ x := !x{+}!x\ ;\ y := !y{+}!y\ \{!x{=}2i \wedge !y{=}2j\}$	(AssignV)
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 derivation for (9.1) 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.

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

We now infer:

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

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.	$(!x = 2i \wedge\ !y = 2j)\{\!\|\,!x{+}!x/!x\|\!\}\ x := !x{+}!x\ \{!x = 2i \wedge\ !y = 2j\}$	(AssignV)
10.	$\{x = y \wedge\ !x = i \wedge\ !y = j\}\ x := !x{+}!x\ \{!x = 2i \wedge\ !y = 2j\}$	(1, 2, Consequence)
11.	$\{!x = i \wedge\ !y = j\}\ \mathtt{if}\ x \neq y\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2\ \{!x = 2i \wedge\ !y = 2j\}$	(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. 15 Swap with added assertions.

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

9.2 Swap

Now we verify `swap`, a program that exchanges the content of two reference cells.

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

Recall that we must establish that

$$\mathsf{Swap}_0(u) \stackrel{\text{def}}{=} \{\mathtt{t}\}\ \mathtt{swap} :_u \{\forall x.\forall y.\{!x = i \wedge !y = j\} u \bullet \langle x,y\rangle \{!x = j \wedge !y = i\}\}.$$

Swap is the classical example treated in many preceding work (cf. [5, 6, 9]). 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 being alias and the other with the case when they are distinct. Let $C \stackrel{\text{def}}{=} !x = i \wedge !y = j$ and $C' \stackrel{\text{def}}{=} !x = j \wedge !y = i$. Assume we could show that

$$\{!y = j \wedge z = i\}\ y := z\ \{!y = i \wedge !x = j\} \tag{9.6}$$

Then we could make the following derivation.

1. $(!y = j \wedge z = i) = (!x = j \wedge z = i)[!y/!x] \equiv (!x = j \wedge z = i)\{\!|!y/!x|\!\}$	(Lemma 6.7)
2. $\{!y = j \wedge z = i\}\ x := !y\ \{!x = j \wedge z = i\}$	(AssignVar, Cons, 9)
3. $\{!y = j \wedge z = i\}\ x := !y; y := z\ \{C'\}$	(Seq)
4. $(C \wedge x = m) \supset (!y = j \wedge z = i)[!m/z] \equiv (!y = j \wedge z = i)\{\!|!m/z|\!\}$	(6.7.5)
5. $\{C\}\ x :_m \{(!y = j \wedge z = i)\{\!|!m/z|\!\}\}$	(Cons, Var, 6)
6. $\{C\}\ !x :_z \{!y = j \wedge z = i\}$	(Deref, 5)
7. $\{C\}\ \mathtt{let}\ z = !x\ \mathtt{in}\ x := !y; y := z\ \{C'\}$	(Let, 4, 7)
8. $\{\mathtt{t}\}\ \mathtt{swap} :_u \{\mathsf{Swap}_0(u)\}$	(Abs, 2, 3)

This leaves (9.6) to be established. So far reasoning in this section was regardless of whether x and y are coalesced or not. For the assignment $y := z$ this is no longer the case. This may seem strange as x does not even occur in the program. But if we want to use [*AssignVar*] to infer

$$\{C' \{\!|z/!y|\!\}\}\ y := z\ \{C'\} \tag{9.7}$$

we need to apply $\exists !y$ to a statement containing both $!x$ and $!y$. This is best done by splitting $!x = j \wedge z = i$, our premise which should imply $C'\{\!|z/!y|\!\}$ in two:

$$(x = y \wedge !x = j \wedge z = i) \supset C'\{\!|z/!y|\!\} \qquad (x \neq y \wedge !x = j \wedge z = i) \supset C'\{\!|z/!y|\!\}$$

Then $(!x = j \wedge z = i) \supset C'\{\!|z/!y|\!\}$ and the inference

$$\dfrac{\dfrac{-}{\{C'\{\!|z/!y|\!\}\}\ y := z\ \{C'\}}[AssignVar]}{\{!y = j \wedge z = i\}\ y := z\ \{C'\}}[Cons]$$

leads to (9.6). The case $x = y$ is simpler.

$$\begin{array}{rcl}(x = y \wedge z = i \wedge !x = j) & = & (x = y \wedge !y = i \wedge !x = j)[z/!y] \\ & \equiv & (x = y \wedge !y = i \wedge !x = j)\{\!|z/!y|\!\}\end{array}$$

For $x \neq y$ we must do a bit more. By (sea) we have

$$\mathtt{t} \quad \supset \quad \langle !y \rangle\, !y = j.$$

We can also derive

$$\begin{array}{rcl}(x \neq y \wedge z = i \wedge !x = j) & \supset & (x \neq y \wedge z = i \wedge !x = j) \\ & \supset & (x \neq y \wedge z = i) \\ & \supset & \forall !y.z = i\end{array}$$

using (sub).

$$\begin{array}{rcl}(x \neq y \wedge z = i \wedge !x = j) & \supset & (\langle !y \rangle\, !y = j) \wedge (\forall !y.z = i) \wedge \forall !x.!x = j \\ & \supset & \langle !y \rangle\, (!y = j \wedge z = i \wedge !x = j) \\ & \supset & \langle !y \rangle\, (!y = j \wedge !y = z \wedge !x = j) \\ & \supset & (!y = j \wedge !x = j)\{\!|z/!y|\!\}\end{array}$$

9.3 Circular References

Reasoning about $x :=!!x$, the example that uses circular data structures is simpler, but proceeds much as in the previous examples. We want to show

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

The correctness proof is as follows. We start by converting the pre-condition into a form that is usable by [*AssignVar*].

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

From here it is easy to get:

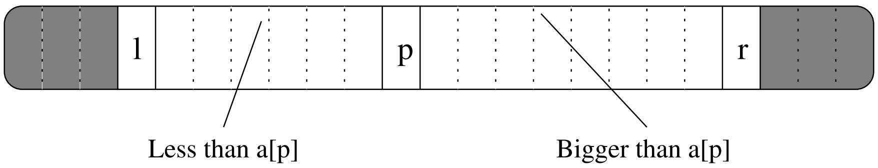
1. $\{(!x = x)\{\!|!!x/!x|\!\}\}\ !!x :_m \{!x = x\}$	(Simple)
2. $\{(!x = x)\{\!|!!x/!x|\!\}\}\ x := !!x\ \{!x = x\}$	(AssignVar, 1)
3. $(!x = y \wedge !y = x) \supset ((!x = x)\{\!|!!x/!x|\!\})$	
4. $\{(!x = x)\{\!|!!x/!x|\!\}\}\ x := !!x\ \{!x = x\}$	(Consequence, 2, 3)

9.4 QuickSort (1): Informal Illustration and Code.

Hoare's Quicksort [16] needs little introduction. We explain its specification using a simplified version; later we verify the algorithms in a slightly different form more natural to our programming language. Assume given an array a of size n and indices $0 \leq l, r < n$. We want the subarray of a delimited by l and r sorted. Quicksort's overall structure is this (indentation is used for scoping like in Haskell and Python):

```
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 one is less than every member of the other, as shown in the next figure.



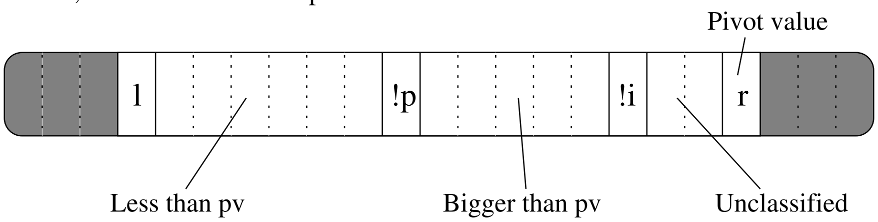
This is done by calling `partition`. Its return value p is called the *pivot point*. With p we sort the two subarrays by recursive calls to Quicksort. Sorting happens with respect to a total order [7] $<$. The algorithm implementing $<$ must terminate on any input.

[7] For simplicity we are being cavalier with mathematical precision in this informal description and $<, \leq$, "less than", "bigger" etc. to mean the result of the relevant total order which may or may not be a strict ordering.

Partitioning is at the heart of Quicksort and it can be done in various ways. Our approach is described in the next bit of code, once again simplified for the ease of presentation (we start from Line 3 for consistency with the line numbering for the full algorithm given later).

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

In Line 3, we determine the pivot value relative to which the array is partitioned. Any array index from l to r may be used and we choose r. It is sometimes advantageous to randomise this choice, but we shall not do so for simplicity. Next the algorithm scans through the entire subarray starting at l, the left end, going right. It maintains two pointers i and p. The idea is that $!i$ points to the current element and all array entries from l to, but not including $!p$ are less than the pivot value pv, while everything starting from $!p$ going right to $!i$ is bigger than pv. Right of $!i$ and left of r we have what is left to be done, as shown in the next picture.



This is essentially the loop invariant as we shall see later, except for the relation between $!a[!p]$ and pv. In regards to that the algorithm guarantees: whenever $!p <!i$ then $pv \leq !a[!p]$. Let's look at the loop body to see how this invariant is maintained.

```
7                   if !a[!i] < pv then
8                           swap( a[!p], a[!i] )
9                           p := !p + 1
10                  i := !i + 1
```

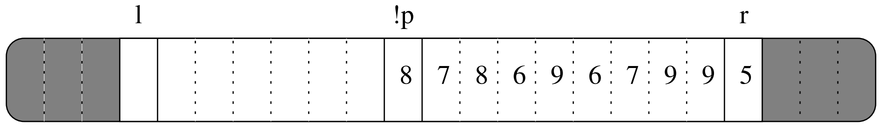
Line 7 compares the current element $!a[!i]$, i.e. the leftmost element we have not yet considered, with the pivot value. If it is less, then we swap it with $a[!p]$. Here `swap` is a generic procedure, for example like that of §9.2, about which we make no assumptions other than that it exchanges the content of its two arguments, terminates on all inputs and leaves everything else unchanged. There are two cases:

- $!p = !i$, in which case swapping has no effect, but the invariant is trivially maintained; or
- $!p <!i$, but then swapping means that (1) everything to the left of $!p+1$ is less than the pivot value because by the comparison we know that what we've just swapped

with $a[!p]$ is smaller than pv and everything else is unchanged; (2) the element now at $a[!i]$ is at least as big as pv. This is by the invariant.

Line 9 then increases p by one. Why we can safely do this if the first of these cases obtains has already been explained. Otherwise, by the loop invariant, everything to the left of $!p$ is less than pv while there are no elements right of $!p$ but to the left of $!i$. Line 10, the last of the loop body, just reaches for the next unprocessed array element.

When the loop terminates the invariant guarantees that everything from $!p$ to, but not including $!i = r$ is bigger than pv which is stored at $a[r]$. But the following is still possible:



We rectify this anomaly in Line 11 by swapping $a[r]$ with $a[!p]$. This is always safe by the invariant because the $!a[!p] \geq pv$ and $pv \geq pv$. After this last swap $!p$ contains the array index to the left of which everything is smaller than the pivot value $a[!p]$ and to the right of which everything is bigger than $a[!p]$, as required.

One may ask how we know when we swap in Line 11 that pv is less than $!a[!p]$. Again there are two cases. The trivial one is $!p = !i = r$. So assume $!p < !i = r$. Then there must have been a first time that swapping does not happen while scanning through the array. In this case $!a[!p] \geq pv$. Now we have two possibilities. Either $!p$ is not increased any further in which case swapping in Line 11 is correct. Alternatively, more swapping happens which means $!p$ is increased more. But that increment moves $!p$ to point to an array element x that was previously between the old value of $!p$ and $!i$, so by the invariant $x \geq pv$.

An alternative specification and verification of a Quicksort variant can be found in [10] where the algorithm and a logic are implemented in Coq. The specification there is not as general as ours because it does not constrain effects in the way our located assertions do.

Our implementation of Quicksort is almost as just outlined, but we make a few simplifications so the verification is easier. In particular, we inline partitioning to avoid a procedure call. As our programming language does not have local references, p and i are global variables. We also generalise the algorithm in the sense that we allow it to work with an arbitrary comparison function **`c`**, about we make no assumptions other than that it implements a total order, terminates on all inputs and is functional, i.e. has no observable side effects. We assume that **`c`** is passed as a parameter to Quicksort while **`swap`** is global. We use two different forms of supplying procedural parameters to an algorithm to demonstrate how our logic can cope with either. The full program is given in Figure 9.4.

9.5 Quicksort (2): Formal Specification.

We shall now give a step-by-step explanation of Quicksort's formal specification that follows the outline above. Figure 9.5 gives a quick overview of the algorithm and its

Fig. 16 The implementation of Quicksort to be verified.

```
λ⟨a,c⟩.μq.λ⟨l,r⟩
     if l < r then
            let pv = !a[r] in
                   p := l
                   i := l
                   while !i < r
                          if c(!a[!i], pv) then
                                 swap( a[!p], a[!i] )
                                 p := !p + 1
                          i := !i + 1
                   swap(a[r], a[!p])
            let p′ = !p in
                   q(l, !p−1)
                   q(p′+1, r)
```

assertions, slightly simplified to aid readability. One of our goals here is to obtain a specification that is as strong as possible in the sense that any program satisfying this specification is contextually indistinguishable from `qsort`, cf.§8. To achieve this goal, the specification must be precise not only about what the program does but also about what it doesn't. This means to specify references that remain unchanged. For if we'd only specify what `qsort` did to the array and the two auxiliary variables, the variant `qsort` which is exactly like `qsort`, except that it also assigns 32 to a fresh variable x would also meet such a specification, and yet, it is not contextually congruent with `qsort`. To make precise the set of locations modified by `qsort` we use the located assertions from §7.4. This choice simplifies the specifications and their proofs quite significantly. A natural way of formalising Quicksort's in our language using located assertions would be to require

$$\{\mathsf{T}\}\ \mathtt{qsort} :_u \{\mathsf{Qsort}_u\}@\emptyset. \tag{9.8}$$

Here Qsort_u is the located predicate

$$\{\mathsf{T}\}u \bullet \langle a,c\rangle \searrow v\{\mathsf{Qsort}^{post}_v\}@\emptyset,$$

where Qsort^{post}_v holds if and only if

$$\{\mathsf{Qsort}^{pre}\}v \bullet \langle l,r\rangle\{\mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)\}@\{a[l...r]ip\}.$$

The effect set $\{a[l...r]ip\}$ is short for $\{a[l],...,a[r],i,p\}$. We must now describe the pre and postconditions in detail. We start with the former. Qsort^{pre} is given as

$$0 \le l,r < n \wedge \mathsf{Swap}_0(s) \wedge \mathsf{Order}_c \wedge \mathsf{Perm}_n(ab) \wedge \mathsf{Distinct}_n(abpi).$$

We now define those five predicates used in Qsort^{pre}. $\mathsf{Swap}_0(s)$ is very much like the predicate defined in §9.2, except that we use a located assertion now, to ensure that swapping really only modifies the two references it is passed and nothing else.

$$\mathsf{Swap}_0(u) \stackrel{\text{def}}{=} \{\mathsf{T}\}\ \mathtt{swap} :_u \{\mathsf{Swap}_0(u)'\}$$
$$\mathsf{Swap}_0(u)' \stackrel{\text{def}}{=} \forall x.\forall y.\{!x = i \wedge !y = j\} u \bullet \langle x, y\rangle \{!x = j \wedge !y = i\} @xy$$

Order_c as a predicate ensures totality of the order and termination everywhere of **c**:

$$\begin{array}{c} \forall xy.\{\mathsf{T}\} c \bullet \langle x,y\rangle \{\mathsf{T}\} @\emptyset \\ \wedge \\ \forall x \neq y. \begin{pmatrix} \{\mathsf{T}\} c \bullet \langle x,y\rangle \searrow b\{b = \mathsf{T}\} \\ \vee \\ \{\mathsf{T}\} c \bullet \langle y,x\rangle \searrow b\{b = \mathsf{T}\} \end{pmatrix} \end{array}$$

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. We do this by stipulating a second array b which stores exactly the same data as a, but is not affected by running the algorithm. This is done in three steps. The first is $\mathsf{Perm}_n(ab)$. It ensures that the arrays a and b are permutations of each other by checking if one array can be obtained from the other by a finite number of binary permutations from each other.

$$\mathsf{Perm}_n(ab) \stackrel{\text{def}}{\equiv} \exists m.\mathsf{Perm}_n^m(ab)$$

The predicate $\mathsf{Perm}_n^0(ab)$ is inductively defined, starting from the base case, simple equality.

$$\mathsf{Perm}_n^0(ab) \stackrel{\text{def}}{\equiv} \mathsf{Equal}_n(ab)$$
$$\mathsf{Perm}_n^{m+1}(ab) \stackrel{\text{def}}{=} \exists c.(\mathsf{Size}(c,n) \wedge \mathsf{Perm}_n^m(bc) \wedge \exists 0 \leq i < j < n.2\mathsf{Perm}_n(abij))$$
$$2\mathsf{Perm}_n(abij) \stackrel{\text{def}}{=} !a[i] = !c[j] \wedge !a[j] = !c[i] \wedge \forall 0 \leq k < n.(k \neq i, j \supset !a[k] = !c[k])$$

Equality is defined element-wise, but also checks that both arrays are of the same length.

$$\mathsf{Equal}_n(ab) \stackrel{\text{def}}{\equiv} \mathsf{Size}(a,n) \wedge \mathsf{Size}(b,n) \wedge \forall 0 \leq i < n.!a[i] = !b[i]$$

An immediate consequence of this definition is that

$$\mathsf{Qsort}^{pre} \quad \supset \quad \mathsf{Size}(a,n) \wedge \mathsf{Size}(b,n). \tag{9.9}$$

At this point one may ask why we assume $\mathsf{Perm}_n(ab)$ in Quicksort's precondition and not the easier $\mathsf{Equal}_n(ab)$ and why we don't restrict permutation to the subarray given by l, r. The reason for this strengthening of the assumptions is that this simplifies handling recursive calls.

$\mathsf{Distinct}_n(abpi)$ ensures that all references used in the algorithm and its specification, the arrays a, b, but also p and i, are no only distinct but recursively so. What that means is made precise now. The predicate $\mathsf{Distinct}_n(S)$ is recursively defined over lists S of variables.

$$\mathsf{Distinct}_n(\varepsilon) \stackrel{\text{def}}{\equiv} \mathsf{T} \qquad \mathsf{Distinct}_n(x,S) \stackrel{\text{def}}{\equiv} \mathsf{Distinct}_n(S) \wedge \forall y \in S.x \neq_n y$$

Here $\neq_n$ has the following definition.

$$\begin{aligned} x^{\alpha} \neq_n y^{\alpha} &\stackrel{\text{def}}{\equiv} x \neq y \qquad\qquad (\alpha \text{ not an array type}) \\ x^{\alpha} \neq_n a^{\alpha[]} &\stackrel{\text{def}}{\equiv} \forall 0 \leq i < n.x \neq a[i] \\ a^{\alpha[]} \neq_n b^{\alpha[]} &\stackrel{\text{def}}{\equiv} a \neq b \wedge \forall 0 \leq i, j < n.a[i] \neq b[j] \end{aligned}$$

We have omitted the case symmetric to the second. For the postcondition it remains to define $\mathsf{Sorted}_n(aclr)$ which ensures that the array is in fact sorted relative to the ordering implemented by `c`, once Quicksort has terminated.

$$\mathsf{Sorted}(aclr) \quad \stackrel{\text{def}}{\equiv} \quad \forall l \leq i < j \leq r.c \bullet \langle !a[i], !a[j] \rangle \searrow \mathsf{T}$$

This predicate is why we need the second array b.

9.6 Quicksort (3): Derivation.

We have now got all the tools together to prove that our implementation of Quicksort meets its specification. We define a few auxiliary predicates to make the derivation more transparent. We start with Qsort_k^{pre} which enriches Qsort^{pre} with a constraint on the relation between l and r.

$$\mathsf{Qsort}_k^{pre} \quad \stackrel{\text{def}}{\equiv} \quad \mathsf{Qsort}^{pre} \wedge 0 \leq r - l \leq k < n$$

This predicate is used in the following context that is for reasoning about recursion.

$$\mathsf{Qsort}_{vk}^{post} \quad \stackrel{\text{def}}{\equiv} \quad \{\mathsf{Qsort}_k^{pre}\} v \bullet \langle l, r \rangle \{\mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)\} @ a[l...r] ip$$

Let's start the derivation of (9.8). In what follows, we focus on derivations in the program logic and shall omit all details of inferences $C \supset C'$ taking place in first-order logic with equality only. By **`qsort'`** we mean the implementation of Quicksort given in Figure 9.4, but with the initial λs and μ removed, i.e. Line 2 to Line 14.

1.1 $\{\mathsf{Qsort}_k^{pre} \wedge \forall j < k.\mathsf{Qsort}_{qj}^{post}\}$ `qsort'` $\{\mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)\} @ a[l...r] ip$	
1.2 $\{\forall j < k.\mathsf{Qsort}_{qj}^{post}\}$ $\lambda \langle lr \rangle$.`qsort'` $:_v$ $\{\mathsf{Qsort}_{vk}^{post}\}$	(Abs', 1.1)
1.3 $\{\mathsf{T}\}$ $\mu q.\lambda \langle lr \rangle$.`qsort'` $:_v$ $\{\forall k.\mathsf{Qsort}_{vk}^{post}\}$	(Rec, 1.2)
1.4 $\forall k.\mathsf{Qsort}_{vk}^{post} \supset \mathsf{Qsort}_v^{post}$	
1.5 $\{\mathsf{T}\}$ $\mu q.\lambda \langle lr \rangle$.`qsort'` $:_v$ $\{\mathsf{Qsort}_v^{post}\}$	(Cons, 1.3, 1.4)
1.6 $\{\mathsf{T}\}$ `qsort` $:_u$ $\{\mathsf{Qsort}_u\}$	Abs', 1.5

Fig. 17 Quicksort with (simplified) Assertions.

$\lambda\langle \mathtt{a,c}\rangle.\mu\, \mathtt{q}.\lambda\langle \mathtt{l,r}\rangle.$
$\{\mathsf{Qsort}^{pre} \wedge 0 \le r-l \le k < n\}$
if l < r then
$\{0 < l < r < n\}$
let pv = !a[r] in
$\{pv = v\}$
p := l
$\{!p = l\}$
i := l
$\{!i = l\}$
while !i < r
$\{\mathsf{Invar}\}$
if !a[!i] < pv then
$\{!a[p_0] < pv\}$
swap(a[!p], a[!i])
$\{\mathsf{Invar}\{\!|!i+1/!i|\!\}\{\!|!p+1/!p|\!\}\}$
p := !p + 1
$\{\mathsf{Invar}\{\!|!i+1/!i|\!\}\}$
i := !i + 1
$\{\mathsf{Invar}\}$
swap(a[r], a[!p])
$\{\mathsf{Leq}(acl(!p-1)!a[!p]) \wedge \mathsf{Geq}(ac(!p+1)r!a[!p]) \wedge \mathsf{Perm}_n(ab)\}$
let p′ = !p in
$\{!p = p'\}$
q(l, !p-1)
$\{\mathsf{Sorted}_n(l!p)\}$
q(p′+1, r)
$\{\mathsf{Sorted}_n(lr)\}$
$\{\{\mathsf{T}\}\ \mathtt{qsort} :_u \{\mathsf{Qsort}_u\}\}$

This leaves (1.1) to be established. Considering [*If*], it is easy to see it that (1.1) would be straightforward to derive, had we managed to verify

- $(l > r \wedge \mathsf{Qsort}_k^{pre} \wedge \forall j < k.\mathsf{Qsort}_{qj}^{post}) \supset \mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)$,
- $(l = r \wedge \mathsf{Qsort}_k^{pre} \wedge \forall j < k.\mathsf{Qsort}_{qj}^{post}) \supset \mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)$,
- $\{l < r \wedge \mathsf{Qsort}_k^{pre} \wedge \forall j < k.\mathsf{Qsort}_{qj}^{post}\}\ \mathtt{qsort}''\ \{\mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)\} @ a[l...r]ip$

where $\mathtt{qsort}''$ is the program starting on Line 3, going on to the end. Of these three assertions, the first is vacuously true, while the second is also easy once one realises that $\mathsf{Qsort}_k^{prepost} \wedge l = r$ implies $\mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)$. Hence, all that remains to be proven is the last. Setting

$$C_0 \quad \stackrel{\text{def}}{\equiv} \quad l < r \wedge \mathsf{Qsort}_k^{pre} \wedge \forall j < k.\mathsf{Qsort}_{qj}^{post}$$

we reason following the syntactic structure where `partition` is the program fragment starting on Line 3 and ending with Line 11.

2.1 $\{C_0\}$ `partition` $\{C_1\}@a[l..r]pi$	
2.2 $\{C_1\}$ $!p :_{p'}$ $\{C_1 \wedge !p = p'\}@a[l...!p-1]pi$	
2.2 $\{C_1 \wedge !p = p'\}$ $q(l, !p-1)$ $\{C_2\}@a[l...!p-1]pi$	
2.3 $\{C_2\}$ $q(!p+1, r)$ $\{\mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)\}@a[!p+1...r]pi$	
2.4 $\{C_0\}$ `qsort`$''$ $\{\mathsf{Perm}_n(ab) \wedge \mathsf{Sorted}_n(aclr)\}@a[l..r]pi$	(Seq, Let', 2.1 - 2.3)

This derivation combines sequential composition and using [*Let*] into one, for simplicity. The recursive calls (2.2) and (2.3) are immediate from the assumptions and their derivation is hence omitted. So it remains to prove (2.1) and to determine C_1, C_2. Clearly

$$C_2 \quad \stackrel{\text{def}}{\equiv} \quad C_1 \wedge !p = p' \wedge \mathsf{Sorted}_n(acli_0 - 1).$$

The definition of C_1 is key:

$$C_1 \stackrel{\text{def}}{=} \exists l \leq i_0 \leq r. \begin{pmatrix} \mathsf{Qsort}_k^{pre} \wedge \forall j < k.\mathsf{Qsort}_{qj}^{post} \wedge !p = i_0 \\ \wedge \\ \mathsf{Leq}(acl(i_0-1)!a[i_0]) \wedge \mathsf{Geq}(ac(i_0+1)r!a[i_0]) \end{pmatrix}.$$

The predicate $\mathsf{Leq}(aclrv)$ is

$$\mathsf{Leq}(aclrv) \quad \stackrel{\text{def}}{\equiv} \quad \forall l \leq k \leq r.(!a[k] = v \ \vee \ c \bullet \langle !a[k], v\rangle \searrow \mathsf{T}).$$

It says that every element in the subarray given by l and r is either equal to a given value v or bigger. Dually we define:

$$\mathsf{Geq}(aclrv) \quad \stackrel{\text{def}}{\equiv} \quad \forall l \leq k \leq r.\neg(c \bullet \langle !a[k], v\rangle \ \searrow \mathsf{T}).$$

The next step is to verify the heart of the algorithm where the array is partitioned into a part which contains only elements smaller than $!a[r]$ and one where all elements are bigger than $!a[r]$, both with respect to the total comparison procedure that was passed as an argument. We aim to prove (2.2) which is $\{C_1\}$ `partition` $\{C_2\}@a[l..r]pi$. To this end we use the sequence of inferences given next.

4.1 $\{C_0\}$ $p := l$ $\{C_0 \wedge !p = l\}@p$	(Assign)
4.2 $\{C_0 \wedge !p = l\}$ $i := l$ $\{C_0 \wedge !p = l \wedge !i = l\}@i$	(Assign)
4.3 $\{C_0 \wedge !p = l \wedge !i = l\}$ `loop` $\{C_1'\}@a[l...r-1]pi$	
4.4 $\{C_1'\}$ $s(a[r], a[!p])$ $\{C_1\}@a[r], a[!p]$	

In this derivation `loop` refers to the program fragment from Line 6 to 10 in Figure 9.4. Inferences (4.1) and (4.2) are trivial because neither p nor i are aliased by the assumptions given in C_0, so we concentrate on the latter two. Swapping in (4.4) is easy from the assumptions. The loop invariant Invar is

$$C_0 \wedge \exists l \leq p_0, i_0 \leq r. \left(\begin{array}{c} !p = p_0 \wedge !i = i_0 \wedge \mathsf{Leq}(acl(p_0-1)pv) \\ \wedge \\ \mathsf{Geq}(ac(p_0+1)(i_0-1)pv) \wedge (p_0 < i_0 \supset !a[p_0] \geq pv) \end{array} \right).$$

Now clearly $(C_0 \wedge !p = l \wedge !i = l) \supset \mathsf{Invar}$. If we define $C_1' \stackrel{\text{def}}{\equiv} \mathsf{Invar} \wedge !i \geq r$, all we need to do to establish (4.3) is to show that $\{\mathsf{Invar}\}\ \texttt{loop}\ \{C_1'\}@a[l...r-1]pi$. This is done in two steps. We being with the `then` branch of the conditional. Let $\mathsf{Invar}' \stackrel{\text{def}}{\equiv} \mathsf{Invar}\{\!|!i + 1/!i|\!\}\{\!|!p + 1/!p|\!\}$.

5.1 $\{\mathsf{Invar} \wedge \{\}c \bullet \langle !a[i_0], pv \rangle \searrow \mathsf{T}\{\}@\emptyset\}\ \texttt{swap}(a[!p], a[!i])\ \{\mathsf{Invar}'\}@a[!p]a[!i]$	
5.2 $\{\mathsf{Invar}'\}\ p := !p + 1\ \{\mathsf{Invar}\{\!|!i + 1/!i|\!\}\}@p$	(Assign)
5.3 $\{\mathsf{Invar}\{\!|!i + 1/!i|\!\}\}\ i := !i + 1\ \{\mathsf{Invar}\}@i$	(Assign)

The two assignments are straightforward as the variables are not aliased. (5.1) is by the assumptions on swapping. The second step, corresponding to the conditional branch not being taken is much simpler.

6.1 $\{\mathsf{Invar}\{\!|!i + 1/!i|\!\}\}\ i := !i + 1\ \{\mathsf{Invar}\}@i$	(Assign)
6.2 $(\mathsf{Invar} \wedge \{\}c \bullet \langle !a[i_0], pv \rangle \searrow \mathsf{f}\{\}@\emptyset) \supset \mathsf{Invar}\{\!|!i + 1/!i|\!\}$	
6.3 $\{\mathsf{Invar} \wedge \{\}c \bullet \langle !a[i_0], pv \rangle \searrow \mathsf{f}\{\}@\emptyset\}\ i := !i + 1\ \{\mathsf{Invar}\}@i$	(Cons)

10 Discussion

Local References. Our logic can easily accommodate richer data types. Integration of polymorphism and recursive types is straightforward following [21]. Record and variants types (labelled unions) are treated like products and sums. Treatment of arrays has been discussed in the main sections: lists and vectors are similarly treated.

On the basis of the present logic, we can further incorporate treatment of new (or local) reference in the program logic, which allows even richer data structures. For this purpose we add formulae $(\nu\tilde{x})\tilde{u}$ `in` C, which state that names $\tilde{u}$ in C use $\tilde{x}$ as shared local references. While details are treated in a coming exposition, one point is worth observing. In procedural languages such as C and Java, a procedure body uses *stack variables*, which are essentially local variables used for storing parameters and local datum. The semantic aspect of such variables are easily captured using new reference generation combined with other higher-order imperative idioms. In particular, the involved parameter passing mechanism is easily mimicked by combining call-by-value and new reference generation (cf. [8]), in the following way. Given a procedure definition with a header **`f(x, y) ...`** in a procedural language, we transform this into the following λ-abstraction and "pre-processing" section.

```
\lambda ( x', y'). let x = new( x' ) in let y = new( y' ) in ....
```

Since x and y are freshly generated, they are never aliased (this is captured by an axiom in the logic with new reference generation). Thus, at least semantically, the distinction between stack-variables and heap-variables is explained away by the combination of more primitive constructs. This analytical nature is the central aspect of the present logic, from which we believe not only scientific clarification but also new engineering prospects will emerge.

Related Work. Below we discuss directly related Hoare-like program logics which treat aliasing (for further discussions on related works see [18, 19, 21]). Early work in this direction is Olderog [32], which presents a sound and complete Hoare logic for an Algol-like language. He treats instantiation of reference names in procedure bodies by considering all possible ways of collapsing them and verifying each case separately. Significance of distinction between reference names and their content in Hoare Logics have been discussed in [24, 35, 36] and, more recently, in [9, 12]. The assignment rule based on semantic substitution is presented in [24, 36] and in the aforementioned work by Cartwright and Oppen. Trakhtenbrot et al. [36] explore proof rules for Algol-like languages with denotational orientation, presenting an invariance rule which is reminiscent of ours. As arrays introduce aliasing between elements, studies of proof rules for arrays such as [2, 13] contain logical analyses of aliasing. More recently Kulczycki et al. [9] study possible ways to reason about aliasing induced by call-by-reference procedure calls.

A different approach to the formal treatment of aliasing based on Burstall's early work, is *Separation Logic* by Reynolds, O'Hearn and others [31, 34] They introduce a novel conjunction $*$ that stipulates, as well as the standard conjunction, disjointness of memory regions underlying two conjuncts. Separation Logic uses the semantics and rules of Hoare logic for alias-free stack-allocated variables while introducing alias-aware rules for variables on heaps. We discuss their work in some detail since it exhibits an interesting contrast with our approach, both philosophically and technically. Their logic starts from a resource-aware assignment (and symmetric dereference) rule [34]: $\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}$ where e and e' do not include dereference of heap variables. The rule *demands* that a memory cell is available at address e. Consequently, $\{\top\}\ [e] := [e]\ \{\top\}$ is unsound in their logic, reflecting the low-level nature of semantics considered in [31, 34], where resource allocation needs be done explicitly. The command corresponds to $x := !x$ in our notation which is, in our language, observationally equivalent to `skip`. $\{\top\}\ x := !x\ \{\top\}$ is trivially sound in Hoare's and our logic.

On the basis of these resource-oriented proof rules, [31, 34] propose a variant of the invariance rule, using their separating conjunction:

$$\frac{\{C\}\ P\ \{C'\} \quad \mathsf{fv}(C_0) \cap \mathsf{modify}(P) = \emptyset}{\{C * C_0\}\ P\ \{C' * C_0\}} \tag{10.1}$$

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 all the variables (addresses) in the heap which P may write to are explicitly mentioned in C. Thus $C * C_0$ indicates 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.

Separation Logic's ability to reason about aliased references depends crucially on its resource-oriented nature and on the use of separating conjunction $*$ and a special predicate $\mapsto$ to represent content of memory cells. Resource-oriented nature in turn comes from the orientation in this logic to capture low-level aspects of language dynamics, among others resource usage. In contrast, the present work aims at a precise logical articulation of observational meaning of programs in high-level programming languages in the traditions of both Hennessy-Milner logic and Hoare's logic, as exemplified by Theorem 8.18. Another difference is that our logic is built on the standard logical apparatus of predicate calculus with equality to represent general aliasing situations (cf. Theorem 8.4). This difference comes to life for example in the [*Invariance*] rule of §7, which plays a role similar to (10.1). Our rule relies on purely compositional reasoning about observable behaviour, which, as our experience in §9 suggests, contributes to tractability in practical reasoning (in particular $!x$-freedom is almost always automatically ensured).

The present logic also offers a simple treatment, clarification and generalisation of a refined invariance rule involving procedures by Yang, O'Hearn and Reynolds [31]. The rule has several side conditions and restriction on the use of formulae. We below offer the corresponding rule in the present logic, taking a simple case for brevity.

$$\frac{C_1\ !\tilde{x}\text{-free} \quad \{C_0\}\ N\ \{C_0' * C_1\} @ \tilde{x}\tilde{y} \quad \{C^{\text{-}f} \wedge \{C_0\}\ f\ \{C_0'\} @ \tilde{x}\}\ M :_u \{C'\} @ \tilde{x}}{\{C \wedge C_1\}\ \mathtt{let}\ f = \lambda().N\ \mathtt{in}\ M :_u \{C' \wedge C_1\} @ \tilde{x}\tilde{y}} \tag{10.2}$$

where we assume f is *ephemeral* in the sense that it occurs in M always in the shape of $f()$ and never under λ-abstraction (which is simply checkable by typing). The rule says if a program M uses a procedure f assuming 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 stored/abstracted behaviour becomes imprecise: in contrast, in the above case, we can adjust it at the time of instantiation once and for all. In comparison with the rule in [31], the rule in(10.2) differs in that it is purely compositional (i.e. does not demand conditions on behaviours of M and N outside of judgements). Note also it 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 is possible in the specific setting which [31] treats.

Another difference is data structures: so far Separation Logic does not include assertions for higher-order expressions and data structures in general in its logical language, apart from treatment of a sequence of memory cells. This is again natural from the orientation towards low-level programs in their work: at a low-level computation, it is often the case that there is no structure except a linear memory and a CPU (and perhaps registers), for example. The present work differs in that it tries to capture abstract program structures and their reasoning principles in the context of program logic.

An elimination procedure similar to our Theorem 8.4 can be found in work by Calcagno, Gardner and Hague [4] who decompose a decidable propositional fragment of a logic with separated conjunction into first-order logic. Lozes [26] takes a similar but more semantic approach. While the target logic is different, [4] is closer 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, and/or low-level, features such as resource bounds.
Note. Further related work will be added in future versions.

References

1. Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. 163:409–470, 2000.
2. K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.
3. Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.
4. Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *FoSSaCs'05 (to appear)*.
5. Robert Cartwright and Derek C. Oppen. Unrestricted procedure calls in Hoare's logic. In *Proc. POPL*, pages 131–140, 1978.
6. Robert Cartwright and Derek C. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.
7. Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.
8. Erik Crank and Matthias Felleisen. Parameter-passing and the lambda-calculus. In *POPL '90*. ACM Press, 1990.

9. Gregory W. Kulczycki et al. 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.
10. Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.
11. Robert W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
12. Dan R. Ghica. *A Games-Based Foundation for Compositional Software Model Checking*. PhD thesis, Queen's University School of Computing, Kingston, Ontario, 2002.
13. David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst.*, 2(4):564–579, 1980.
14. Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.
15. Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.
16. C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.
17. Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
18. 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.
19. Kohei Honda. Process Logic and Duality: Part (1) Sequential Processes. Available at: www.dcs.qmul.ac.uk/˜kohei/logics, March 2004. Typescript, 234 pages.
20. Kohei Honda and Nobuko Yoshida. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.
21. Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *Proc. PPDP'04*. ACM Press, 2004.
22. 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.
23. J. Martin E. Hyland and C. H. Luke Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
24. 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.
25. Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.
26. Etienne Lozes. Elimination of spatial connectives in static spatial logics. *TCS*, to appear.
27. Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
28. Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables. In *Proc. POPL'88*, 1988.
29. 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.
30. Robin Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
31. Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. POPL'04*, 2004.
32. Ernst-Rüdiger Olderog. Sound and complete hoare-like calculi based on copy rules. *Acta Inf.*, 16:161–197, 1981.
33. Benjamin C. Pierce. *Type Systems and Programming Languages*. MIT Press, 2002.
34. John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. LICS'02*, 2002.
35. Richard L. Schwartz. An aximatic treatment of algol 68 routines. In *Proc. ICALP*, pages 530–545, 1979.
36. 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.

A Comments on Semantics of Assertions

Quantification over Reference Names In [22] 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 [22], 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, [22] 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

$$V \stackrel{\text{def}}{=} \lambda x.\mathtt{if}\ x = y\ \mathtt{then}\ 1\ \mathtt{else}\ 2,$$

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

$$\mathcal{M} \models \{\top\}\ V :_u \{\forall x.\{x \Downarrow\} u \bullet x \searrow n\{n = 2\}\}$$

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

$$\mathcal{M} \models \{\top\}\ V :_u \{\forall x.\{x \Downarrow\} u \bullet x \searrow n\{n = 1\}\}$$

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.

And if α is not a reference type? Consider the formula $\exists x.x = 3$. When should $\mathcal{M} \models \exists x.x = 3$ hold? By the definition above 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 is also contrary to our conventional understanding of formulae like $\exists x.x = 3$.

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 Alternative Axiomatisation of Content Quantification

An alternative axiomatisation uses one inference rule and one axiom for universal quantifications, and another which relates the universal to its dual. First, if we consider deduction in the style of natural deduction, we stipulate an inference rule:

> If each assumption is syntactically $!x$-free, and if the conclusion is C, then we can infer $[!x]C$.

The inference rule is the precise analogue of what we find in the standard quantification theory except "x not occurring freely in assumptions" in the standard rule is replaced by "assumptions are syntactically $!x$-free". Similarly we have the rule for the sequent calculus presentation where we infer $\Gamma \vdash \Delta, [!x]C$ from $\Gamma \vdash \Delta, C$ provided Γ and Δ are syntactically $!x$-free.

The axiom is again the analogue of the standard one, stipulating:

$$[!x](A \supset B) \supset A \supset [!x]B$$

provided B is syntactically $!x$-free.

The existential content quantificationon is now a derived construct, defined by the equation:

$$\langle !x \rangle C \quad \stackrel{\text{def}}{\equiv} \quad \neg[!x]\neg C.$$

Finally, using this notation, we stipulate the axiom (ue1), which we reproduce below:

$$[!x](!x = m \supset C) \quad \equiv \quad \langle !x \rangle (C \wedge !x = m)$$

We can easily check all axioms presented in Section 6.1 are derivable from these rules.