A Logical Analysis of Aliasing in Imperative Higher-Order Functions

Martin Berger∗ Kohei Honda∗ Nobuko Yoshida†

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 parts of data structures. The program logic extends our earlier logic for alias-free imperative higher-order functions with new modal operators which serve as building blocks for clean structural reasoning about programs and data structures in the presence of aliasing. This has been an open issue since the pioneering work by Cartwright-Oppen and Morris twenty-five years ago. We illustrate usage of the logic for description and reasoning through concrete examples including a higher-order polymorphic Quicksort. 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.

Contents

∗ Department of Computer Science, Queen Mary, University of London.
† Deptartment of Computing, Imperial College London.

1 Introduction

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

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

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

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

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

$$R \stackrel{\text{def}}{=} \lambda(fxy).\ (\ \texttt{let}\ z = !x\ \texttt{in}\ !x := 1; !y := 2; f(x,y); z := 3\ )$$

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 (for simplicity let's assume x and y store distinct references). 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 $\texttt{swap} \stackrel{\text{def}}{=} \lambda ab.\texttt{let}\ c = !b\ \texttt{in}\ (b := !a; a := c)$ is passed, the content of x and y is swapped and $!x$ now stores 2 while $!y$ stores 3. Such interplay between higher-order procedures and aliasing is common in many non-trivial programs in ML, C and more recent typed and untyped low-level languages (Peyton Jones *et al.*, 1999; Grossman *et al.*, 2002; Shao, 1997).

Hoare logic (Hoare, 1969), developed on the basis of Floyd's assertion method (Floyd, 1967), has been studied extensively as a verification method for first-order imperative programs with diverse applications. However Hoare's original proof system is sound only when aliasing is absent (Apt, 1981; Cousot, 1999): while various extensions have been studied, a general solution which extends the original method to treat aliasing, retaining its semantic basis (Greif & Meyer, 1981; Hoare & Jifeng, 1998) and tractability, has not been known, not to speak of its combination with arbitrary imperative higher-order functions (our earlier work (Honda *et al.*, 2005) extends Hoare logic with a treatment for a general class of higher-order imperative functions including stored procedures, but does not treat aliasing).

Resuming studies by Cartwright-Oppen and Morris from 25 years ago (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981; Morris, 1982b), the present paper introduces a simple and tractable compositional program logic for general aliasing and imperative higher-order functions. A central observation in (Cartwright & Oppen, 1978; Cartwright &

Oppen, 1981; Morris, 1982b) is that (in)equations over names, simple as they may seem, are expressive enough to describe general aliasing in first-order procedural languages, provided we distinguish between reference names (which we write x) and the corresponding content (which we write $!x$) in assertions. In particular, their work has shown that alias robust substitution, written $C\{\!|e/!x|\!\}$ in our notation, defined by:

$$\mathcal{M} \models C\{\!|e/!x|\!\} \quad \text{iff} \quad \mathcal{M}\,[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \tag{1}$$

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

$$\{C\{\!|e/!x|\!\}\}\, x := e\, \{C\} \tag{2}$$

where the pre-condition in fact stands for the translated form mentioned above. The rule subsumes the original axiom but is now alias-robust. As clear evidence of descriptive power of this approach, Cartwright and Oppen showed that the use of (2) leads to a sound and (relatively) complete logic for a programming language with first-order procedures and full aliasing (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981): Morris showed many non-trivial reasoning examples for data structures with destructive update, including the reasoning for Schorr-Waite algorithm (Morris, 1982b).

The works by Cartwright-Oppen and Morris, remarkable as they are, still beg the question how to reason about programs with aliasing in a tractable way. The first issue is calculation of validity in assertions involving semantic substitutions. Cartwright and Oppen's inductive decomposition of $\{\!|e/!x|\!\}$ into (in)equations has been the only syntactic tool available, and is hardly practical. As demonstrated through many examples by Morris (Morris, 1982b) and, more recently, Bornat (Bornat, 2000), this decomposition should be distributed to every part of a given formula even if that part is irrelevant to the state change under consideration, making reasoning extremely cumbersome. As one typical example, if we use the decomposition method for calculating the logical equivalence

$$C\{\!|\mathsf{c}/!x|\!\}\{\!|e/!x|\!\} \quad \equiv \quad C\{\!|\mathsf{c}/!x|\!\}$$

for general C, with c being a constant, we need either meta-logical reasoning, induction on C, or an appeal to semantic means. Because such logical calculation is a key part of program proving (Hoare, 1969), practical usability of this approach becomes unclear. The second problem is the lack of structured reasoning principles for deriving precise description of extensional program behaviour with aliasing. This makes reasoning hard, because properties of complex programs often depend crucially on how sub-programs interact through shared, possibly aliased references. Finally, the logics in (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981; Morris, 1982b) and their successors do not offer a general treatment of higher-order procedures and mutable data structures which may store such procedures.

We address these technical issues by augmenting the logic for imperative higher-order functions introduced in (Honda *et al.*, 2005) with a pair of mutually dual logical primitives called *content quantifiers*. They offer an effective middle layer with clear logical status for reasoning about aliasing. The existential part of the primitives, written $\langle !x \rangle C$, is defined by

the following equivalence:

$$\mathcal{M} \models \langle !x \rangle C \quad \stackrel{\text{def}}{\equiv} \quad \exists V.(\mathcal{M}\,[x \mapsto V] \models C) \tag{3}$$

The defining clause says: "for some possible content of a reference named x, $\mathcal{M}$ satisfies C" (which may *not* be about the current state, but about a possible state, hence the notation). Syntactically $\langle !x \rangle C$ does *not* bind free occurrences of x in C. Its universal counterpart is written $[!x]C$, with the obvious semantics.

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

$$C\{\!|e/!x|\!\} \quad \equiv \quad \exists m.(\langle !x \rangle (C \wedge !x = m) \wedge m = e). \tag{4}$$

From (3) and (4), the logical equivalence (1) is immediate, recovering (2) as a syntactic axiom. Not only does $C\{\!|e/!x|\!\}$ now have concrete syntactic shape without needing global distribution of update operations, but these operators also offer a rich set of logical laws coming from standard quantifiers and modal operators, enabling efficient and tractable calculation of validity while subsuming Cartwright-Oppen/Morris's methods. Intuitively this is because logical calculation can now focus on those parts which do get affected by state change: just like lazy evaluation, we do not have to calculate those parts which are not immediately needed. In later sections we shall demonstrate this point through examples.

Closely related with its use in logical calculation is a powerful descriptive/reasoning framework enabled by content quantification in conjunction with standard logical primitives. By allowing hypothetical statements about the content of references separate from reference names themselves (which is the central logical feature of these operators), complex aliasing situations are given clean, succinct descriptions, combined with effective compositional reasoning principles. This is particularly visible when we describe and reason about disjointness and sharing of mutable data structures (in this sense it expands the central merits of "separating connectives" (O'Hearn *et al.*, 2004; Reynolds, 2002), as we shall discuss in later sections). The primitives work seamlessly with the logical machinery for capturing pure and imperative higher-order behaviour studied in (Honda, 2004; Honda & Yoshida, 2004; Honda *et al.*, 2005), enabling precise description and efficient reasoning for a large class of higher-order behaviour and data structures.

Third, and somewhat paradoxically, these merits of content quantification come without additional expressive power: any formula which contains content quantification can be translated, up to logical equivalence, into one without. While establishing this result, we shall also show that content quantification and semantic update are mutually definable. Thus name (in)equations, content quantification and semantic update are all equivalent in sheer expressive power: the laws of content quantification are reducible to the standard axioms for predicate calculus with equality, which in turn are equivalent to semantic update through its axioms for decomposition. This does not however diminish the significance of content quantification: without identifying it as a proper logical primitive with associated axioms, it is hard to consider its use in reasoning, both in logical calculation and in its applications to structured reasoning for programs and data structures in the presence of general aliasing.

Structure of the Paper

In the rest of the paper, Section 2 briefly summarises the programming language. Section 3 introduces models for the logic. Section 4 illustrates the key ideas underlying content quantification, 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 discusses several key technical properties of the proposed logic: eliminability of content quantification and soundness of axioms and proof rules. Sections 9 gives non-trivial reasoning examples. Section 10 is devoted to discussing related work and further topics.

This paper is a full version of (Berger *et al.*, 2005), with complete definitions and detailed proofs. The present version not only gives more illustration of axioms and proof rules, but also more comprehensive comparisons with related work.

Our previous work on logic for imperative higher-order functions (Honda *et al.*, 2005), treated a sublanguage of the language investigated here, different only in that reference types are never carried by other types. This small syntactic change in types leads to a significant difference in realisable behaviour. This difference in behaviour and how it can be handled, logically as well as semantically, is the main focus of the present work.

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 (Pierce, 2002) and given by the following grammar.

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

Abstraction, recursion and the case construct are annotated by types. Constants ($\mathtt{c}, \mathtt{c}', \ldots$) include unit $()$, natural numbers $\mathtt{n}$ and booleans $\mathtt{b}$ (either true $\mathtt{t}$ or false $\mathtt{f}$). $\mathtt{op}(\tilde{M})$ (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. (Pierce, 2002; Gunter, 1995). The notions of binding and α-convertibility are also conventional and $\mathsf{fv}(M)$ denotes the set of free variables in M. We use abbreviations such as:

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

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

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

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

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

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

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

Fig. 1. Typing rules.

Types are ranged over by $\alpha,\beta,\ldots$ and are given by the following grammar.

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

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 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.

A *basis* is a finite map from names to types. $\Gamma,\Gamma'\ldots$ range over bases and $\mathsf{dom}(\Gamma)$ denotes the domain of Γ, while $\mathsf{cod}(\Gamma)$ denotes the range of Γ. The typing rules are standard (Pierce, 2002) and listed in Figure 1, using sequents $\Gamma\vdash M:\alpha$, which say that M has type α under basis Γ.

The following subclass of programs is important in the subsequent development (its original appearance may be (Meyer & Sieber, 1988)).

Definition 1 A typed program $\Gamma\vdash M:\alpha$ is *semi-closed* when $\mathsf{cod}(\Gamma)$ only includes 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 to be instantiated into concrete (semi-closed) values. Having free reference variables in a program is quite different, since that program needs them to interact with the store. If reference names are λ-abstracted, programs can touch references only after the abstracted names are instantiated into concrete names by application. In the light of the above discussion, it is often convenient to single out the reference-type part of a basis. We let $\Delta, \ldots$ range over bases whose codomains 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 $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. Semi-closed programs can always be written $\Delta \vdash M : \alpha$ (however, writing $\Gamma \vdash M : \alpha$ does *not* mean the lack of reference-typed variables in M: it is only in the notation $\Gamma;\Delta$ that Γ denotes an environment basis). We often call variables of reference types *reference names* or simply *references*. By abuse of terminology we shall sometimes use "reference" to denote the named memory cells 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 codomain of) σ. A *configuration* is a pair of a semi-closed program and a store. Then *reduction* is a binary relation over configurations, written $(M, \sigma) \longrightarrow (M', \sigma')$, generated by the rules below (Gunter, 1995; Pierce, 2002). We use left-to-right evaluation, but the proposed logic can treat other evaluation strategies and allows us to infer properties which hold regardless of evaluation strategy. First we generate reductions over programs (not configurations) based on the usual reduction rules for 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 first-order operators, pairs, projection and injection. Evaluation contexts are given by the grammar presented next.

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

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 for all n there is a reduction sequence $(M,\sigma)\longrightarrow^n (M',\sigma')$. Here $\longrightarrow^n$ is the n-fold relational composition of $\longrightarrow$.

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 codomain of σ also occur in its domain. We set:

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

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 1 (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 1 Henceforth we restrict the reduction relation to well-typed configurations, that is whenever we write $(M,\sigma)\longrightarrow(M',\sigma')$, we assume $\Delta\vdash(M,\sigma)$ for some Δ.

2.3 Contexts and Contextual Congruence

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

Definition 2 (modest contexts) A typed context $C[\cdot]$ is *semi-closing* if its resulting program is semi-closed. It is *modest* if it is semi-closing and, moreover, it does not abstract any reference name in the hole.

Note a modest context always has the form $C[\ \cdot\ ]^{\Delta';\alpha'}_{\Gamma;\Delta;\alpha}$ with $\Delta'\supseteq\Delta$, and does not collapse reference names in a program. An example of a modest context is

$$(\lambda x^{\mathsf{Nat}}.[\ \cdot\ ])^{y:\mathsf{Ref}(\mathsf{Nat});\mathsf{Nat}\Rightarrow\mathsf{Bool}}_{x:\mathsf{Nat},y:\mathsf{Ref}(\mathsf{Nat});\mathsf{Bool}}$$

which abstracts a value-typed variable x, whereas

$$(\lambda z^{\mathsf{Ref}(\mathsf{Nat})}.[\ \cdot\ ])^{\mathsf{Ref}(\mathsf{Nat})\Rightarrow\mathsf{Bool}}_{z:\mathsf{Ref}(\mathsf{Nat});\mathsf{Bool}}$$

is not (since a reference name z in the hole is abstracted).

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

$$\forall \sigma.(\ (M_1,\sigma)\Downarrow \quad \Leftrightarrow \quad (M_2,\sigma)\Downarrow\ ) \tag{5}$$

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

$$\forall \sigma,\ \text{semi-closing}\ C[\ \cdot\ ]^{\mathsf{Unit}}.\ (\ (C[M_1],\sigma)\Downarrow \quad \Leftrightarrow \quad (C[M_2],\sigma)\Downarrow\ ) \tag{6}$$

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

$$\forall \delta,\ \sigma,\ \text{modest}\ C[\ \cdot\ ]^{\mathsf{Unit}}.\ (\ (C[M_1\delta],\sigma)\Downarrow \quad \Leftrightarrow \quad (C[M_2\delta],\sigma)\Downarrow\ ) \tag{7}$$

where δ ranges over (possibly non-injective) well-typed substitution of reference names for reference names. This characterisation says all experiments we need to inspect the contextual behaviour of a program are combination of modest contexts and possible ways to collapse reference names. To check the equivalence, if M_1 and M_2 satisfies (6), then surely they also satisfy (7), by taking appropriate contexts in (6). For the other direction, suppose M_1 and M_2 satisfy (7). Then (again by taking appropriate contexts) they also satisfy $(C[M_1\xi],\sigma)\Downarrow$ iff $(E[M_2\xi],\sigma)\Downarrow$ for any σ, ξ, and modest $C[\ \cdot\ ]$, where ξ ranges over well-typed substitutions of (both non-reference and reference) variables for semi-closed values. This means we can always replace $M_1\xi$ and $M_2\xi$ in a hole of a context without changing termination behaviour of the whole. Now assume, for a possibly non-modest context $C[\ \cdot\ ]$, that $C[M_2]$ converges. Tracing the reductions starting from $C[M_1]$, whenever a duplicate of M_1 is launched into an evaluation context, in the form $M_1\xi$, we replace it with $M_2\xi$, so that when $C[M_2]$ terminates, a residual of $C[M_2]$ (with replacements), say N_2, is identical with that of $C[M_1]$, say N_1, except for duplicates of M_1 under λ-abstraction. Since if N_2 has no redex then N_1 cannot have any redex, so we know that $C[M_1]$ also converges. Symmetrically, if $C[M_1]$ converges then $C[M_2]$ converges, hence we obtain property (6).

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

Definition 3 *Let $\Gamma \vdash M_{1,2} : \alpha$ and assume Δ covers Γ. Then we set $\Gamma \vdash M_1 \cong_\Delta M_2 : \alpha$ when the following condition holds.*

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

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

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

Proof
The "only if" direction is immediate. For the "if" direction, suppose $\Gamma \vdash M_1 \cong_\Delta M_2 : \alpha$ for each Δ which covers Γ. We show M_1 and M_2 satisfy the characterisation (7). Let $\Gamma = \Gamma_0; \Delta_0$ and suppose $C[M_1\delta] \Downarrow$ for some modest $C[\ \cdot\ ]$. Let, without loss of generality (through injective renaming and weakening), we have $\Delta \vdash \sigma$ such that $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Gamma) = \emptyset$ and $\Delta_0 \subseteq \Delta$. Since $M_1 \cong_\Delta M_2$ we have $C[M_2\delta] \Downarrow$ as required. □

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

3 Models

We introduce a class of models which concisely represent computational situations of interest. We follow our previous work (Honda *et al.*, 2005) except for the additional use of *distinctions* to describe aliasing, an innovation coming from the π-calculus (Milner *et al.*, 1992). Our models are immediately faithful to the observable behaviour of programs, which is important for our logic's observational completeness.

3.1 Distinctions

In (Honda *et al.*, 2005) a model was a pair (ξ, σ) where ξ mapped non-reference names to semi-closed values and σ was a store mapping reference names to semi-closed values. One of the key operations on models was $\mathcal{M}\ [x \mapsto V]$ which returns a model that is exactly like $\mathcal{M}$, except that the reference name x, assumed to occur in $\mathcal{M}$, now maps to V. In the presence of aliasing, where distinct reference names may refer to a store location, this operations cannot just work on the given name x, but must also update what is stored at all of x's aliases. For this purpose we use *distinctions*, equivalence classes of reference names, following Milner, Parrow and Walker (Milner *et al.*, 1992), rather than introducing an additional set of location labels. The latter approach can be found in the dynamic semantics of ML (Milner *et al.*, 1990). The notion of distinction distills the idea of aliasing at a high level of abstraction.

Definition 4 (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 $x \mathcal{D}' y$.

Example 1 *Assume* $\Delta = x, y, z : \mathsf{Ref}(\mathsf{Nat}), a, b : \mathsf{Ref}(\mathsf{Nat} \Rightarrow \mathsf{Nat})$. *Then*

$$\mathcal{D} = \{(x,x),(y,y),(z,z),(a,a),(b,b),(x,y),(y,x),(a,b),(b,a)\},$$

the least equivalence over $\mathsf{dom}(\Delta)$ *relating x with y and a with b, is a distinction of type* Δ *with identicals* $\{x,y\},\{z\},\{a,b\}$. *The full distinction of type* $\Delta^- = x, y : \mathsf{Ref}(\mathsf{Nat}), a :$

$\mathsf{Ref}(\mathsf{Nat}\Rightarrow\mathsf{Nat})$ *clearly satisfies* $\mathcal{D}^- \leq \mathcal{D}$. *But if* $\mathcal{D}'$ *is the full distinction of type* Δ, *then* $\mathcal{D} \not\leq \mathcal{D}'$ *and* $\mathcal{D}' \not\leq \mathcal{D}$.

We continue with some notational conventions that are useful for handling distinctions.

Notation 1 $\mathcal{D} - \mathbf{i} \stackrel{\text{def}}{=} \mathcal{D} \setminus \{\mathbf{i}\times\mathbf{i}\}$ (assuming $\mathbf{i}$ is an identical of $\mathcal{D}$). Dually $\mathcal{D} + \mathbf{i} = \mathcal{D} \cup \{\mathbf{i}\times\mathbf{i}\}$ (assuming $\mathbf{i} \cap (\cup\mathcal{D}) = \emptyset$). We write $\Delta\mathcal{D}$ for the base which has $\mathcal{D}$-identicals as domain of definition and maps $\mathbf{i}$ to $\Delta(x)$, provided $x \in \mathbf{i}$. Given $\Gamma;\Delta \vdash M : \alpha$ and $\mathcal{D}$ is a distinction of type Δ, then $M\mathcal{D}$ is obtained by replacing each $x \in \mathsf{dom}(\Delta)$ in M with the unique $\mathcal{D}$-identical $\mathbf{i}$ such that $x \in \mathbf{i}$. More precisely, if $\mathsf{dom}(\Delta) = \{x_1,\ldots,x_n\}$ and $x_j \in \mathbf{i}_j$, then $M\mathcal{D} \stackrel{\text{def}}{=} M[\mathbf{i}_1/x_1]\cdots[\mathbf{i}_n/x_n]$.

Example 2 *With* $\mathcal{D}^\Delta$ *from Example 1 above, we have*

$$\Delta\mathcal{D} = \{x,y\} : \mathsf{Ref}(\mathsf{Nat}), \{z\} : \mathsf{Ref}(\mathsf{Nat}), \{a,b\} : \mathsf{Ref}(\mathsf{Nat}\Rightarrow\mathsf{Nat}).$$

Now let M be the program $x :=\!!a+3;\ z :=\!!y$. *Then* $\Delta \vdash M : \mathsf{Unit}$ *and, setting* $\mathbf{i} \stackrel{def}{=} \{x,y\}, \mathbf{j} \stackrel{def}{=} \{a,b\}, \mathbf{k} \stackrel{def}{=} \{z\}$, *we have*

$$M\mathcal{D} \quad = \quad \mathbf{i} :=\!!\mathbf{j}+3;\ \mathbf{k} :=\!!\mathbf{i} \qquad\qquad \Delta\mathcal{D} \vdash \mathcal{M}\ \mathcal{D} : \mathsf{Unit}.$$

We construct models relative to a distinction. This is fundamental to our concern since the logical description of program 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. But 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.

Consequently, our semantic domains are constructed from semi-closed programs (up to the observational congruence) parameterised by distinctions. This accords with our intuitive understanding of observational indistinguishability under potential aliasing. A program's behaviours relative to a given distinction $\mathcal{D}$ can be made explicit by using $\mathcal{D}$-identicals as reference names, as already demonstrated in Example 2. The next example shows the difference in dynamics engendered by varying distinctions more clearly. 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.

Since our definition of configurations (page 6) treated store names atomically, we are free to use identicals as store domains. All the corresponding constructs (store typings, reductions, substitution, etc. stay unchanged). For example if N is the program $x := 3; !y$ typed under $\Delta \stackrel{\text{def}}{=} x : \mathsf{Ref}(\mathsf{Nat}),\ y : \mathsf{Ref}(\mathsf{Nat})$, and $\mathcal{D}^{\Delta}$ identifies x with y, i.e. $x,y \in \mathbf{i}$, then $[\mathbf{i} \mapsto 7]$ is a well-typed store under $\Delta\mathcal{D}$. Moreover, 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 in this way.

However, $\cong$ as defined in Section 2.3 is too fine for semantics of programs w.r.t. distinctions. To see why, consider

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

where ω is some diverging term of Unit type. If we consider a distinction in which x and y are equated, then we expect M and $()$ to be contextually equivalent. But $\cong$ says $M \ncong ()$ because it considers arbitrary aliasing: if x and y are distinct, then we do have $M \Uparrow$, so we cannot generally say $M \cong ()$.

In order to deal with this phenomenon, we define a distinction-respecting congruence. Below we say Δ is *complete* if whenever a reference type (say α) occurs in any reference type in its codomain 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 5 ($\mathcal{D}$-respecting congruence) Let Δ be complete and $\mathcal{D}$ be a distinction over Δ. Then we write $\Gamma;\Delta \vdash M_1 \cong_{\mathcal{D}} M_2 : \alpha$ for $\Gamma;\Delta \vdash M_{1,2} : \alpha$ iff, for each modest $C[\ \cdot\ ]$ and corresponding store σ, we have:

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

As is customary, we often simply write $M \cong_{\mathcal{D}} N$ when the 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 (7) (in Section 2.3, Page 8) to be $\mathcal{D}$-respecting, i.e. we only consider substitutions which collapse names that are equal in $\mathcal{D}$ but leave distinct those which are distinct in $\mathcal{D}$. Conversely, $\cong$ arises from $\cong_{\mathcal{D}}$ by ranging $\mathcal{D}$ over all possible distinctions.

Proposition 3 *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 $\Gamma;\Delta \vdash M \cong_{\mathcal{D}} N : \alpha$.*

This Proposition is an easy corollary of $\cong$'s characterisation in (7). We close this subsection with a small observation about the effect that extending a distinction has on $\cong_{\mathcal{D}}$.

Proposition 4 *Let $\Gamma;\Delta \vdash M_{1,2} : \alpha$ and assume $\Gamma \subseteq \Gamma'$ and $\Delta \subseteq \Delta'$. Assume further $\mathcal{D}^{\Delta} \leq \mathcal{D}'^{\Delta'}$. 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 5 do not vary by extending bases and distinctions. □

3.2 Models

We can now define models, but rather than taking tuples (ξ,σ) for models where ξ maps non-reference names to semi-closed values and ξ is a store, i.e. a mapping from reference names to semi-closed values (this was the model construction in (Honda *et al.*, 2005)), we now take triples $(\mathcal{D},\xi,\sigma)$.

Definition 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 semi-closed values such that each $x\in\mathsf{dom}(\Gamma)$ is mapped to V such that $\Delta\mathcal{D}\vdash V:\Gamma(x)$ and each $x\in\mathsf{dom}(\Delta)$ is mapped to the unique $\mathcal{D}$-identical containing x.
3. σ is a *store*, that is a finite map from the identicals of $\mathcal{D}$ to semi-closed values so that an identical of type $\mathsf{Ref}(\alpha)$ is mapped to a term $\Delta\mathcal{D}\vdash V:\alpha$.

$\mathcal{M},\mathcal{M}',\ldots$ 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 $\mathsf{dom}(\mathcal{M})\stackrel{\text{def}}{=}\mathsf{dom}(\Gamma\cup\Delta)$.

Example 3 *Let* Γ *be* $x:\mathsf{Nat}, f:\mathsf{Nat}\Rightarrow\mathsf{Nat}$ *and assume* Δ *is* $y,z:\mathsf{Ref}(\mathsf{Nat})$. *Assume* $\mathcal{D}$ *is the distinction of type* Δ *identifying* y *with* z, *i.e.* $\{x,y\}=\mathbf{i}$. *With*

$$\xi\;=\;x:7,f:\lambda n.1+!\mathbf{i},y:\mathbf{i},z:\mathbf{i}\qquad\qquad\sigma\;=\;[\mathbf{i}\mapsto 9],$$

$(\mathcal{D},\xi,\sigma)$ *is a model of type* $\Gamma;\Delta$. *Clearly* $\mathsf{dom}(\mathcal{D},\xi,\sigma)=\{x,y,z,f\}$.

Convention 2 (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 value, indicating the effect of assignment commands.

Definition 7 (semantic update) Let $\mathcal{M}^{\,\Gamma;\Delta,x{:}\mathsf{Ref}(\alpha)}\stackrel{\text{def}}{=}(\mathcal{D},\,\xi,\,\sigma\cdot\mathbf{i}\mapsto W)$ with $x\in\mathbf{i}$. Further let $\Delta\vdash V:\alpha$, Then the expression $\mathcal{M}[x\mapsto V]$ or, alternatively, $\mathcal{M}[\mathbf{i}\mapsto V]$, denotes $(\mathcal{D},\,\xi,\,\sigma\cdot\mathbf{i}\mapsto V)$. Clearly (1) $\mathcal{M}[x\mapsto V](x)=V$ and (2) for each y that is not in x's identical we have $\mathcal{M}[x\mapsto V](y)=\mathcal{M}(y)$.

Notation 2 Given $\mathcal{M} = (\mathcal{D}, \xi, \sigma)^{\Gamma;\Delta}$, $u \notin \mathsf{fv}(\mathcal{M})$ and $\mathcal{D}\Delta \vdash V : \alpha$, we write $\mathcal{M} \cdot u : V$, or often $(\xi \cdot u : V, \sigma)$, for a model that extends $\mathcal{M}$ by one entry with the value V. Formally we set $\mathcal{M} \cdot u : V$ to be a model $\mathcal{M}'$ such that:

1. If α is a value type then $\mathcal{M}' = (\mathcal{D}, \xi \cdot u : V, \sigma)^{\Gamma \cdot u:\alpha;\Delta}$; and
2. If α is a reference type then, with $\mathbf{i} \stackrel{\text{def}}{=} V \cup \{u\}$,

$$\mathcal{M}' = (\mathcal{D} - V + \mathbf{i}, \xi[\mathbf{i}/V] \cdot u : \mathbf{i}, \sigma[\mathbf{i}/V])^{\Gamma;\Delta \cdot u:\alpha}$$

where the substitutions on environments and stores have the following definitions: $\sigma[\mathbf{i}/\mathbf{j}]$ is defined as $\emptyset[\mathbf{i}/\mathbf{j}] = \emptyset$ and $(\sigma \cdot \mathbf{i}' \mapsto U)[\mathbf{i}/\mathbf{j}] = \sigma[\mathbf{i}/\mathbf{j}] \cdot \mathbf{i}'[\mathbf{i}/\mathbf{j}] \mapsto U[\mathbf{i}/\mathbf{j}]$. Similarly: $(\xi \cdot x : U)[\mathbf{i}/\mathbf{j}] = (\xi[\mathbf{i}/\mathbf{j}]) \cdot U[\mathbf{i}/\mathbf{j}]$.

In general, in Clause 2 above, we cannot have $\mathcal{M}'(u) = V$ since for reference types u itself is adjoined to an existing identical. Note that such $\mathcal{M}'$ is determined uniquely.

4 Two Modal Operators

This section motivates content quantification and its genesis in the analysis of the soundness proof for Hoare's original logic.

Aliasing and Assignment. As illustrated in the introduction, interaction between aliasing and assignment leads to difficulties in reasoning. For concreteness let's consider the following program.

$$\texttt{double?} \quad \stackrel{\text{def}}{=} \quad \lambda x^{\mathsf{Ref(Nat)}}.\lambda y^{\mathsf{Ref(Nat)}}.(x :=\!!x+\!!x \; ; \; y :=\!!y+\!!y) \tag{8}$$

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 that way if we apply the *same* reference to this program twice, as in $((\texttt{double?})r)r$. For suppose r originally stores 2. Then, after execution, we obtain 8 instead of 4 as 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 (Hoare, 1969) dictates that a valid judgement should be derived compositionally, i.e. precisely following the program text. Let us consider how this may be done for double?, focussing on the second command "$y :=\!!y+\!!y$". Suppose for concreteness that the content of both x and y is 2 at the entry point. If we were without aliasing, we would have the following specification.

$$\{!x = !y = 2\}\, y :=\!!y+\!!y\, \{!x = 2 \wedge !y = 4\} \tag{9}$$

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, (9) is *not* a precise specification of the assignment command in the presence of aliasing. So how can we amend (9)? Since the postcondition of (9) *is* correct if x and y are distinct references, the following gives a natural refinement of (9).

$$\{x \neq y \wedge !x = !y = 2\}\, y :=\!!y+\!!y\, \{!x = 2 \wedge !y = 4\} \tag{10}$$

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 distinguished the content of a register R from the content of a memory cell whose address is held in R. At the level of programming languages, it is in typed languages like ML and Haskell, that the need of assigning correct types to expressions have led to strict differentiation between references and their content.

Assignment Axiom with Aliasing. But how can we derive specifications such as (10) 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.

$$[\mathit{Assign\text{-}Org}]\ \frac{-}{\{C[e/!x]\}\ x := e\ \{C\}} \tag{11}$$

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. (Apt, 1981; Cousot, 1999). For example, in the case of double? and the post-condition $!x = 2 \wedge !y = 4$, we easily calculate, with $\equiv$ indicating logical equivalence:

$$(!x = 2 \wedge !y = 4)[!y{+}!y\,/\,!y] \quad \equiv \quad !x = 2 \wedge !y = 2 \tag{12}$$

which gives the pre-condition in (9) in the alias-free setting, rather than what we want, (10). Another slightly different Hoare triple for the same command makes the underlying issue more vivid. For the program $y := !y{+}!y$ and the post-condition $!x = 2$, we want to derive:

$$\{\,(x = y \wedge !y = 1)\ \vee\ (x \neq y \wedge !x = 2)\,\}\ \ y := !y{+}!y\ \ \{\,!x = 2\,\} \tag{13}$$

By informal reasoning, we can see that the judgement (13) is operationally reasonable. But if we apply the syntactic substitution to the given post-condition, we obtain;

$$(!x = 2)[!y{+}!y\,/\,!y] \quad \equiv \quad !x = 2 \tag{14}$$

In view of the pre-condition in (13), we can see (14) precisely leaves out the case when x and y are aliased. Indeed, to obtain the pre-condition of (13) from $!x = 2$, the syntactic substitution $[!y{+}!y\,/\,!y]$ is powerless, since y does not even occur in the postcondition.

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

$$\mathcal{M} \models C_0 \quad \text{iff} \quad \mathcal{M}\,[\mathbf{x} \mapsto [\![e]\!]_{\mathcal{M}}] \models C. \tag{15}$$

Above the boldfaced $\mathbf{x}$ signifies the identical containing x. $\mathcal{M}$ represents the state *before* the assignment, while $\mathcal{M}\,[\mathbf{x} \mapsto [\![e]\!]_{\mathcal{M}}]$, 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 7, even if x is aliased, $\mathcal{M}\,[\mathbf{x} \mapsto [\![e]\!]_{\mathcal{M}}]$ gives the correct update. Thus (15) 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 $\mathcal{M}\,[\mathbf{x} \mapsto [\![e]\!]_{\mathcal{M}}] \models C$ 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 did it work in the alias-free setting? In brief, this is thanks to the following logical equivalence.

$$C[e/!x] \quad \equiv \quad \exists m.(\exists x.(C \wedge !x = m) \wedge m = e) \tag{16}$$

Note that we cannot simplify the right-hand side into $\exists x.(C \wedge !x = e)$ because $!x$ may occur in e. Using (16), we justify (15) as follows, assuming $\mathcal{M}$ is alias-free, i.e. its distinction is full (hence we write x, not $\mathbf{x}$).

$$\mathcal{M}\,[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \quad \Leftrightarrow \quad \mathcal{M} \cdot m : [\![e]\!]_{\mathcal{M}}\,[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \wedge !x = m \tag{17}$$

$$\Leftrightarrow \quad \mathcal{M} \cdot m : [\![e]\!]_{\mathcal{M}} \models \exists x.(C \wedge !x = m) \tag{18}$$

$$\Leftrightarrow \quad \mathcal{M} \models \exists m.(\exists x.(C \wedge !x = m) \wedge m = e) \tag{19}$$

$$\Leftrightarrow \quad \mathcal{M} \models C[e/!x] \tag{20}$$

All are standard logical equivalences under the full distinction, clarifying the status of the logical equivalence (16) in Hoare's original assignment axiom.

The key step in our analysis is that from (17) to (18): we had to get rid of the model update $[x \mapsto [\![e]\!]_{\mathcal{M}}]$, and to do so we must ensure that the truth value of the formula on the right of the satisfaction relation is independent of what is stored at x. Without aliasing we can achieve this by simply hiding x through existential abstraction, because in this setting, the only (non-trivial) thing we can do with a reference name in a logical formula is to dereference it. Hence, if x is not a free name of a formula, the formula is true/false independently from what the model stores at x.

Now consider why this proof above no longer works in the presence of aliasing an what can be done about it. Remember that we want to find a formula C_0 such that (15) holds. Yet, when trying to mimic derivation (17 - 20) in the presence of aliasing we find that while the first step is as (17) before, the second fails:

$$\begin{array}{lll} \mathcal{M}\,[\mathbf{x} \mapsto [\![e]\!]_{\mathcal{M}}] \models C & \Leftrightarrow & \mathcal{M} \cdot m : [\![e]\!]_{\mathcal{M}}\,[\mathbf{x} \mapsto [\![e]\!]_{\mathcal{M}}] \models C \wedge !x = m \\ & \not\Leftrightarrow & \mathcal{M} \cdot m : [\![e]\!]_{\mathcal{M}} \models \exists x.(C \wedge !x = m) \end{array}$$

The problem is that now a formula's truth value may depend on what is stored at (the identical containing) x, even when x does not occur freely in the formula on the right of the satisfaction relation. The addition of aliasing increased the expressiveness of the assertion language. To see how to deal with this conundrum, we note that for all $\mathcal{M}'$,C':

$$\mathcal{M}'[\mathbf{x} \mapsto [\![e]\!]_{\mathcal{M}}] \models C' \quad \equiv \quad \exists V.\mathcal{M}'[\mathbf{x} \mapsto V] \models C' \wedge !x = e.$$

Since we need to make a formula independent from what stored at $\mathbf{x}$ in the model, not from x itself, this last equivalence is suggestive of a new quantifier: assume we had an operator $\langle !x \rangle C$ in our logic with the following semantics, cf. (3):

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

It is the content V of x, rather than x itself, that is existentially abstracted. C' may still talk about x, for example saying that $x = y$, but the truth value of $\langle !x\rangle C'$ is now independent from what $\mathcal{M}'$ stores at $\mathbf{x}$. If content quantification were part of our assertion language, we could reason:

$$\mathcal{M}\,[\mathbf{x}\mapsto[\![e]\!]_{\mathcal{M}}]\models C \quad\Leftrightarrow\quad \mathcal{M}\cdot m:[\![e]\!]_{\mathcal{M}}\,[\mathbf{x}\mapsto[\![e]\!]_{\mathcal{M}}]\models C\wedge !x=m \tag{21}$$

$$\Leftrightarrow\quad \mathcal{M}\cdot m:[\![e]\!]_{\mathcal{M}}\models\langle !x\rangle\,(C\wedge !x=m) \tag{22}$$

$$\Leftrightarrow\quad \mathcal{M}\models\exists m.(m=e\ \wedge\ \langle !x\rangle\,(C\wedge !x=m)) \tag{23}$$

Hence content quantification allows to re-introduce the equivalence (16) that witnessed the correctness of the original Hoare rule, but enhanced, so it is robust under aliasing. We can then define

$$C\{\!|e/!x|\!\}\ \stackrel{\text{def}}{=}\ \exists m.(\langle !x\rangle\,(C\wedge !x=m)\wedge m=e). \tag{24}$$

We call $\{\!|e/!x|\!\}$ *semantic substitution* or *logical substitution*. By the semantics of content quantification, in (21) - (23), we have re-established the logical equivalence in (15), replacing $C[e/!x]$ with $C\{\!|e/!x|\!\}$, mimicking (17–19) above. Thus we now arrive at the following proof rule.

$$[\mathit{AssignBasic}]\ \frac{-}{\{\,C\{\!|e/!x|\!\}\,\}\,x:=e\,\{C\}} \tag{25}$$

This rule subsumes the original rule (25) since $C\{\!|e/!x|\!\}$ coincides with $C[e/!x]$ under the full distinction. The semantic status of (25) 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 (13) easily? Does it allow extensions/generalisation to higher-order programming languages, for example those with the generalised assignment of the form $M := N$ where both M and N are appropriately typed arbitrary expressions? And can we reason about programs with aliasing tractably and modularly using content quantification? 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 standard first-order logic with equality (Mendelson, 1987) extended with assertions for evaluation and quantification over store content. The latter is the only addition to the logic in (Honda *et al.*, 2005).

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

Here $\star\in\{\wedge,\vee,\supset\}$ and $Q\in\{\forall,\exists\}$. The first set of expressions (ranged over by $e,e',\ldots$) are *terms* while the second set are *formulae* (ranged over by $A,B,C,C'\ldots$). The constants

$$\frac{(\Gamma\cup\Delta)(x)=\alpha}{\Gamma;\Delta\vdash x:\alpha} \quad \frac{-}{\Gamma;\Delta\vdash \mathsf{n}:\mathsf{Nat}} \quad \frac{\Gamma;\Delta\vdash e:\mathsf{Bool}}{\Gamma;\Delta\vdash \neg e:\mathsf{Bool}} \quad \frac{\Gamma;\Delta\vdash e_i:\alpha_i\ (i=1,2)}{\Gamma;\Delta\vdash (e_1,e_2):\alpha_1\times\alpha_2} \quad \frac{\Gamma;\Delta\vdash e_{1,2}:\alpha}{\Gamma;\Delta\vdash e_1=e_2}$$

$$\frac{\Gamma;\Delta\vdash e:\alpha_1\times\alpha_2}{\Gamma;\Delta\vdash \pi_i(e):\alpha_i} \quad \frac{\Gamma;\Delta\vdash e:\alpha_i\ (i\in\{1,2\})}{\Gamma;\Delta\vdash \mathsf{inj}_i^{\alpha_1+\alpha_2}(e):\alpha_1+\alpha_2} \quad \frac{\Gamma;\Delta\vdash e:\mathsf{Ref}(\alpha)}{\Gamma;\Delta\vdash !e:\alpha}$$

$$\frac{\Gamma;\Delta\vdash C_{1,2}}{\Gamma;\Delta\vdash C_1\star C_2}\star\in\{\wedge,\vee,\supset\} \quad \frac{\Gamma,x{:}\alpha;\Delta\vdash C}{\Gamma;\Delta\vdash \mathcal{Q}x^{\alpha}.C}\mathcal{Q}\in\{\forall,\exists\} \quad \frac{\Gamma;\Delta\vdash e:\mathsf{Ref}(\alpha)\quad \Gamma;\Delta\vdash C}{\Gamma;\Delta\vdash \langle !e\rangle C}$$

$$\frac{\Gamma;\Delta\vdash e:\mathsf{Ref}(\alpha)\quad \Gamma;\Delta\vdash C}{\Gamma;\Delta\vdash [!e]C} \quad \frac{\Gamma;\Delta\vdash e_1:\alpha\Rightarrow\beta\quad \Gamma;\Delta\vdash e_2:\alpha\quad \Gamma;\Delta\vdash C\quad (\Gamma;\Delta)\cdot z:\beta\vdash C'}{\Gamma;\Delta\vdash \{C\}\ e_1\bullet e_2\ =\ z\ \{C'\}}$$

Fig. 2. Typing Rules for Terms and Formulae

(c, c′,...) include unit (), numerals n and booleans b (either true t or false f). 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[1] and injection operation. The final term, $!e$, denotes the dereference of e, i.e. the content of a store denoted by e.

The predicate $\{C\}\ e\bullet e'\ =\ x\ \{C'\}$ is called *evaluation formula* (Honda *et al.*, 2005), where the name x binds its free occurrences in C'. C and C' are called *(internal) pre/post conditions*. Intuitively, $\{C\}\ e\bullet e'\ =\ x\ \{C'\}$ 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: 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 and the evaluation formulae. 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. Since $\langle !e\rangle C$ is logically equivalent to $\exists x.(e=x\wedge\langle !x\rangle C)$ if x is fresh, content quantification in its full generality is not needed for expressivity, only $\langle !x\rangle C$ and its de Morgan dual $[!x]C$.

Terms are typed inductively starting from types for variables and constants and signatures for operators. The key typing rules are given in Figure 2. Recalling that $\Gamma;\Delta$ indicates

[1] The projection operator $\pi_i(e)$ has been included for convenient presentation of some proof rules but is redundant: for example the formulae $\pi_1(e)=e'$ can be expressed as $\exists xy.(e=\langle x,y\rangle\wedge x=e')$.

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 $\Gamma;\Delta \vdash C$ if C is well-typed under $\Gamma;\Delta$. *Henceforth we only treat well-typed terms and formulae.*

Further notational conventions follow.

Convention 3 (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, $\{C\}\, e_1 \bullet e_2 \,=\, e'\, \{C'\}$ stands for $\{C\}\, e_1 \bullet e_2 \,=\, x\, \{x = e' \wedge C'\}$, with x fresh; and $\{C\}\, e_1 \bullet e_2\, \{C'\}$ stands for $\{C\}\, e_1 \bullet e_2 \,=\, ()\, \{C'\}$.

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 (Mendelson, 1987).

Definition 8 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 $Q\,y.C$ 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 $\{C_1\}\, e_1 \bullet e_2 \,=\, y\, \{C_2\}$ if
   - e is free for x in C_1 and C_2,
   - $e = \mathcal{E}[!e']$ 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
- $e = \mathcal{E}[!e']$ such that e' and e_0 having the same type, implies $x \notin \mathsf{fv}(C)$.

7. The case $\langle !e_0 \rangle C$ is similar to the last.

In (5, 6) $\mathcal{E}[\cdot]$ is a one-holed expression context, we omit the straightforward definition.

The last two conditions, 5 and 6, concern name capture by content quantification. As we formalise later, the semantics of evaluation formulae says that dereferences in pre/post-conditions of evaluation formulae are implicitly universally quantified. Avoiding inappropriate name-capture with content quantifiers is similar to the same problem for conventional quantifiers. This is illustrated next, by considering Clause 6 below. Consider the following assertion:

$$C \quad \stackrel{\text{def}}{=} \quad z = 3 \quad \supset \quad [!y]\, z = 3 \tag{26}$$

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 (26) when we apply the substitution $[!y/z]$ naively, is *not* a tautology (in fact it is unsatisfiable).

$$C[!y/z] \quad \stackrel{\text{def}}{=} \quad !y = 3 \quad \supset \quad [!y]\, !y = 3. \tag{27}$$

Note $!y$ is not free for z in C due to content quantification on $!y$. (27) 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 (26). To safely apply $[!y/z]$ to C, we transform C to the following formula, up to logical equivalence:

$$C' \quad \stackrel{\text{def}}{=} \quad \exists z'.(\ (z = 3 \quad \supset \quad [!y]\, z' = 3) \ \wedge \ z = z'\ ) \tag{28}$$

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

$$C'[!y/z] \ \stackrel{\text{def}}{=} \ \exists z'.(\ (!y = 3 \quad \supset \quad [!y]\, z' = 3) \ \wedge \ !y = z'\ ) \tag{29}$$

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

Convention 4 *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

As already explained, the present logic, makes extensive use of a logical version of substitution defined below.

Definition 9 (logical substitutions) We set:

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

with m fresh. Dually we set

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

again with m fresh.

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

Logical substitutions behave well in the present theory. In particular, content substitution interacts with content quantification just as syntactic substitution does with conventional quantification (cf. (Mendelson, 1987)). The smooth interplay is aided by suitable axioms for content quantification, to be presented in Section 6. For example, we have $[!x]C \supset C\{\!|e/!x|\!\}$ for any (well-typed) x, e and C, which corresponds to the familiar axiom $\forall x.C \supset C[e/x]$. It should then be no surprise that $C\{\!|e/!x|\!\} \supset \langle !x\rangle C$ also holds, corresponding to the standard entailment $C[e/!x] \supset \exists x.C$. Properties of content quantifications/substitutions will be studied in detail later.

5.4 Semantics of Terms and Formulae

The interpretation of terms is straightforward, given as follows.

Definition 10 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.

$$\begin{array}{rcl} [\![x^\alpha]\!]_{\mathcal{M}} & = & \xi(x) \\ [\![!e]\!]_{\mathcal{M}} & = & \sigma([\![e]\!]_{\mathcal{M}}) \\ [\![\mathsf{c}^\alpha]\!]_{\mathcal{M}} & = & \mathsf{c} \\ [\![\mathtt{op}(\tilde{e})]\!]_{\mathcal{M}} & = & \mathtt{op}([\![\tilde{e}]\!]_{\mathcal{M}}) \end{array} \qquad \begin{array}{rcl} [\![\langle e, e'\rangle]\!]_{\mathcal{M}} & = & \langle [\![e]\!]_{\mathcal{M}}, [\![e']\!]_{\mathcal{M}}\rangle \\ [\![\pi_i(e)]\!]_{\mathcal{M}} & = & V_i \quad \text{where } [\![e]\!]_{\mathcal{M}} = \langle V_1, V_2\rangle \\ [\![\mathtt{inj}_i(e)]\!]_{\mathcal{M}} & = & \mathsf{inj}_i([\![e]\!]_{\mathcal{M}}) \end{array}$$

In the clause for op we omit to detail the straightforward workings of op on first-order values.

We use the following notation to define the satisfaction relation.

Notation 3 Given $x \notin \mathsf{fv}(\mathcal{M}_1)$, $\mathcal{M}_1 \leq_{x:\alpha} \mathcal{M}_2$ if for some appropriately typed V in the sense of Notation 2: either

- $\mathcal{M}_2 \stackrel{\text{def}}{=} \mathcal{M}_1 \cdot x : V^\alpha$; or
- $\alpha = \mathsf{Ref}(\beta)$ and $\mathcal{M}_1 = (\mathcal{D}, \xi, \sigma)$, $\mathcal{M}_2 = (\mathcal{D} + \{x\},\ \xi \cdot x : \{x\},\ \sigma \cdot \{x\} \mapsto V^\beta)$.

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

there is always a new entry in $\mathcal{M}_2$ which maps x to an appropriate value. Models extensions $\leq_{x:\alpha}$ are used in interpretating first-order quantifiers.

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 (Honda *et al.*, 2005); (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.

Definition 11 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 $[\![e_1]\!]_{\mathcal{M}} \cong_{\mathcal{D}} [\![e_2]\!]_{\mathcal{M}}$.
- $\mathcal{M} \models \neg C$ 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 $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$.
- $\mathcal{M} \models \{C\}e \bullet e' = x\{C'\}$ 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 semi-closed V of appropriate type:
  — $([\![e]\!]_{\mathcal{M}'}[\![e']\!]_{\mathcal{M}'}, \sigma') \Downarrow (V, \sigma'')$ and
  — $(\mathcal{D},\ \xi{\cdot}x{:}V,\ \sigma'') \models C'$.
- $\mathcal{M} \models [!e^{\mathsf{Ref}(\alpha)}]C$ if $[\![e]\!]_{\mathcal{M}} = \mathbf{i}$ and for each $V \in [\![\alpha]\!]^{\Delta}_{\mathcal{D}}$ we have $\mathcal{M}[\mathbf{i} \mapsto V] \models C$.
- $\mathcal{M} \models \langle !e^{\mathsf{Ref}(\alpha)} \rangle C$ if $[\![e]\!]_{\mathcal{M}} = \mathbf{i}$ and some $V \in [\![\alpha]\!]^{\Delta}_{\mathcal{M}}$ exists with $\mathcal{M}[\mathbf{i} \mapsto V] \models C$.

Some observations follow.

1. The clauses for universal and existential quantification give the standard definition whenever α is a value type. If it is a reference type, it allows x to be aliased to existing identicals, but does not require aliasing.
2. The clause for $\mathcal{M} \models \langle !e \rangle C$ says: in order to see if $\langle !e \rangle C$ 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

The assertion "$y = 6$" says y is equal to 6. In fact, we should write "$y^{\mathsf{Nat}} = 6$" with a type annotation on y, but we often omit such obvious or irrelevant detail. 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 type annotation, says the content of a memory cell named

y is equal to 6. 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 with 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

The following assertion can be considered as a specification for the program $\lambda z.z :=!z \times 2$, named u.

$$\forall x.\forall i.\{!x = i\}\, u \bullet x\, \{!x = 2 \times i\} \tag{30}$$

We recall from Convention 3 that the formula "$\{!x = i\}\, u \bullet x\, \{!x = 2 \times i\}$" is an abbreviation for "$\{!x = i\}\, u \bullet x = z\, \{z = () \wedge\ !x = 2 \times i\}$". The returned value () 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 (30), 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.

$$\forall x,y,i,j.\ \{!x = i\ \wedge\ x \neq y \wedge\ !y = j\}\, u \bullet x\, \{!x = 2 \times i\ \wedge\ x \neq y \wedge\ !y = j\}. \tag{31}$$

The assertion says that, in addition to the property already stated in (30), the program guarantees that x is the only reference it may alter.[2] It will be convenient to use the following abbreviation for (31).

$$\forall x,i.\ \{!x = i\}\, u \bullet x\, \{!x = 2 \times i\}\, @\, x \tag{32}$$

Such assertions are called *located assertions*. (32) says the same thing as (31) but more concisely. This is discussed in more detail later, starting with Section 5.6.

5.5.1 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 our new forms of quantifications.

[2] In (31), y and j refer to an arbitrary reference and its content, which cannot be typed by the monomorphic type discipline. There are two straightforward resolutions to this issue. We could add ML-style implicit polymorphism to our assertion language (but not to the programming language). Alternatively, we note that if two references y and z are of different types, writing to y cannot change what's stored by z. However, from the effect set, which is finite, we can determine set of all reference types where effects may happen. As this set must be finite (due to the lack of recursive types), we treat each of these types separately. See Section 5.6 for details.

First, as a very simple example, consider an assertion

$$\langle !y \rangle\, !y = 1 \tag{33}$$

where we have omitted to annotate y with $\mathsf{Ref}(\mathsf{Nat})$. The assertion says:

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

In a hypothetical state, the content of a store may differ from the current one. Since we can surely hypothesise such a state, the statement is always true, so that (33) 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 "$x := m$".

$$\langle !x \rangle\, (!x = 2 \wedge\ !x = m). \tag{34}$$

A model $\mathcal{M}$ satisfies this assertion if and only if there is a model $\mathcal{M}'$ which is exactly like $\mathcal{M}$ except possibly for the value stored at a memory cell referred to by x and which satisfies, at that memory cell, $!x = 2 \wedge\ !x = m$. What this means is that the assertion above does not talk about what is stored at x. All it says that it is possible to fill a memory cell named x such that we have both $!x = 2$ and $m = !x$. Note this entails m and 2 should be equal (which is a stateless fact). As this does not claim anything about the content of x, only about its possible content, the only thing being asserted here is that m denotes 2 in the model, hence (34) is logically equivalent to $m = 2$.

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

$$\langle !x \rangle\, (x = y \wedge\ !y = 1). \tag{35}$$

It hides the content of x, but 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. Because x and y denote the same cell, the quantification not only hides the content of x but also that of y. This is an immediate consequence of the standard equality law (Mendelson, 1987), "$x = y \wedge C(x,x) \supset C(x,y)$" 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 (35) is logically equivalent to $x = y$.

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

$$\langle !x \rangle\, (x \neq y \wedge\ !y = 1). \tag{36}$$

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 (36) 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 (34) into "$!y = m$", obtaining:

$$\langle !y \rangle\, (!x = 2 \wedge\ !y = m) \tag{37}$$

which is the same thing as "$(!x = 2)\{\!|m/!y|\!\}$" up to logical equivalence. Thus (37) 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, (37) is equivalent to $(x = y \supset m = 2) \wedge (x \neq y \supset !x = 2)$, or equivalently to $(x = y \wedge m = 2) \vee (x \neq y \wedge !x = 2)$. This is quite different from, say, $\exists i.(!y = i \wedge !x = 2 \wedge m = !y)$.

5.5.2 Content Quantification (2): Universal

The following two examples use universal content quantification. It is the de Morgan dual of its existential counterpart: $[!e]C$ is equivalent to $\neg\langle !e\rangle\neg C$. 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

$$[!x]\,!y = 3 \tag{38}$$

assuming x is typed with $\mathsf{Ref}(\mathsf{Nat})$. By definition, (38) 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 (38) 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.

Universal content quantification offers a powerful tool when combined with located evaluation formulae. Recall the located assertion (32) which is for the program $\lambda z.z :=!z \times 2$, reproduced below:

$$\forall x, i.\ \{!x = i\}\, u \bullet x\, \{!x = 2 \times i\}\, @x \tag{39}$$

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

$$\forall x, i.\ \{!x = i \wedge [!x]\mathit{Even}(!y)\}\, u \bullet x\, \{!x = 2 \times i \wedge [!x]\mathit{Even}(!y)\}\, @x \tag{40}$$

which is a consequence of (39) (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 (39) to (40) 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. It might be useful to note that $[!x]C$ does *not* say that C does not dereference x. $[!x]C$ merely asserts that the truth of C is independent from x's content. That this is a different statement is clear because for example $[!x]\,!x =!x$ holds.

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

$$!x = 2 \ \wedge \ [!x]\,(!x = 3 \supset \mathsf{Odd}(!x)) \ \wedge \ \{\mathsf{Odd}(!x)\}\,u \bullet ()\,\{\mathsf{Even}(!x)\} \tag{41}$$

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 in Section 6):

$$!x = 2 \ \wedge \ [!x]\,(!x = 3 \supset \mathsf{Odd}(!x)) \ \wedge \ \{!x = 3\}\,u \bullet ()\,\{\mathsf{Even}(!x)\} \tag{42}$$

By comparing (41) with the following assertion we can see the role of 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 obtain $\{!x = 3\}\,u \bullet ()\,\{\mathsf{Even}(!x)\}$ by the entailment.

5.5.3 Assertions for Double

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

First we treat the Questionable Double, whose definition in Section 4 was the following.

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

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

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

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.

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

This meets the "expected" specification:

$$\forall x,y,i,j.\ \{!x = i \wedge !y = j\}\ u \bullet (x,y)\ \{!x = 2i \wedge !y = 2j\} \tag{43}$$

If we use a located assertion (cf. Section 5.6 below), we can further refine (43) to:

$$\forall x,y,i,j.\ \ \{!x = i \wedge !y = j\}\ u \bullet (x,y)\ \{!x = 2i \wedge !y = 2j\}\,@xy \tag{44}$$

The quantification of x and y extends to the whole formula, including the terminal $@xy$. (44) says that we can guarantee, in addition to the functional property described above, that no reference cells other than those passed as arguments to this program are modified.

5.5.4 Assertions for Swap

A classical example for reasoning about aliasing (cf. (Cartwright & Oppen, 1981; Cartwright & Oppen, 1978; Kulczycki *et al.*, 2003)) is the swapping routing:

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

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

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

Again we can refine the program using a located assertion:

$$\mathsf{Swap}(u) \quad \stackrel{\text{def}}{=} \quad \forall xyij.\{!x = i \wedge !y = j\}\,u \bullet \langle x,y\rangle\,\{!x = j \wedge !y = i\}\,@xy \tag{45}$$

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

5.6 Located Evaluation Formulae

Before moving to the next section, we present the formal definition of located evaluation formulae, motivated in the previous subsection. Our aim here is to add located evaluation formulae

$$\{C\}e \bullet e' = x\{C'\}\,@\tilde{g} \tag{46}$$

as derived constructs to our logic, assuming that $\tilde{g}$ represents a finite set of reference typed expressions. The intended reading or (46) is:

> *Evaluating the application of (the denotations of) $e \bullet e'$ in a context that is correctly described by C will terminate and yield a result. This result, named x, together with the state after the evaluation is correctly described by C'. In addition, any reference cell* not *denoted by any member of $\tilde{g}$ stores the same content before and after execution of the application.*

This reading suggests to take (46) as standing for

$$\forall y.\forall j.\{C \wedge y \neq \tilde{g} \wedge !y = j\}e \bullet e' = x\{C' \wedge !y = j\} \tag{47}$$

where y, j are fresh and $y \neq \tilde{g}$ stands for $\wedge_i y \neq g_i$. Note that (47) is not typable in general: $\forall y$ cannot range over all reference types, and similarly for $\forall j$. Relatedly, $y \neq \tilde{g}$ may feature inequalities between expressions of different types, which is prohibited by the typing rules in Figure 2.

As mentioned in Footnote 2 on page 21, there are two straightforward approaches that overcome this problem. The first adds a weak form of polymorphism that allows y and j to range over all appropriate types. The second, rather than having a single variable y to range over all reference cells not denoted by $\tilde{g}$, takes one such variable for each relevant type (and likewise for j). This is possible as effects happen only at a finite number of types.

The next proposition gives an example of what we would like to be able to prove for located assertions, regardless of how they are implemented.

Proposition 5 (located assertions) *The following assertions are tautologies.*

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

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

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

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

We now discuss both implementation options and with sketching how located evaluation formulae deal with potentially unbounded effects, as may happen in the presence of recursive types.

5.6.1 Located Evaluation Formulae (1): Polymorphic Implementation

To make (46) typable using polymorphism, we add type variables to the grammar of types used in assertions and universal quantification over types and its dual to the grammar of assertions:

$$\alpha ::= \ldots \mid \mathrm{X} \qquad C ::= \ldots \mid \forall \mathrm{X}.C \mid \exists \mathrm{X}.C$$

Types are taken syntactically, and to accommodate type variables, the satisfaction relation is now a triple $\mathcal{M}^{\Gamma'\Delta'} \models^I C^{\Gamma\Delta}$ where I is a finite map from type variables to closed types such that $(\Gamma\Delta)I = \Gamma'\Delta'$, see (Honda & Yoshida, 2004) for details. We must also mildly change well-formedness and interpretation of expressions to accommodate type variables.

- Well-formedness of equations is now given as: $\Gamma;\Delta \vdash e_1 = e_2$ iff $\Gamma;\Delta \vdash e_i : \alpha_i$. In other words, we no longer require e_1 and e_2 to have the same type.
- Equations of expressions with different types are always false. This is reflected in the following modification of the satisfaction relation. Let $\mathcal{M} = (\mathcal{D}, \xi, \sigma)$.

$$\mathcal{M} \models e_1^{\alpha_i} = e_2^{\alpha_2} \text{ if } \begin{cases} \mathsf{F} & \alpha_1 \neq \alpha_2 \\ [\![e_1]\!]_{\mathcal{M}} \cong_{\mathcal{D}} [\![e_2]\!]_{\mathcal{M}} & \alpha_1 = \alpha_2 \end{cases}$$

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, set union and set difference).

Definition 12 (located assertions (1)) The notation $\{C\}e \bullet e' = x\{C'\}@\tilde{g}$, called *evaluation formula located at* $\tilde{g}$ with each g_i of reference type, but not containing a dereference, denotes the following formula, with X, y and j fresh.

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

The set of expressions $\tilde{g}$ in $\{C\}e \bullet e' = x\{C'\}@\tilde{g}$ 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, $\{C\}e \bullet e' = x\{C'\}@\tilde{w}$ 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 that during this evaluation only references named by $\tilde{w}$ can be changed.

5.6.2 Located Evaluation Formulae (2): Monomorphic Implementation

Implementing located evaluation formulae with polymorphism is fine but needs slightly changed models. We now show how one can avoid this change in setup by extracting more information from the effects' types.

The key insight is that effects happen only at a finite number of reference types. For example if the effect set was $\{e_1^{\mathsf{Ref}(\alpha)}, e_2^{\mathsf{Ref}(\alpha)}, e_3^{\mathsf{Ref}(\beta)}\}$, instead of the polymorphic representation:

$$y^{\mathsf{Ref}(\mathrm{X})} \neq \tilde{e} \wedge !y = j^{\mathrm{X}}$$

from Definition 12, we could express the same constraints through the monomorphic

$$(y^{\mathsf{Ref}(\alpha)} \neq e_1 \wedge y \neq e_2 \wedge !y = j^{\alpha}) \wedge (z^{\mathsf{Ref}(\beta)} \neq e_3 \wedge !z = k^{\beta}).$$

This is reflected in the following monomorphic definition for located evaluation formulae.

Definition 13 *Given a set* $S = \{e_1^{\alpha_1}, \ldots, e_n^{\alpha_n}\}$ *and* β, *we define* $S|_\beta \stackrel{def}{=} \{e^{\alpha_i} \in S \mid \alpha_i = \beta\}$, *the* restriction of S to type β.

Using this construct, we define located evaluation formulae as the following variant of Definition 12.

Definition 14 (located assertions (2)) Assume $\tilde{g}$ is a finite set of reference typed expressions, not containing a dereference, which together have types $\{\mathsf{Ref}(\alpha_1), \ldots, \mathsf{Ref}(\alpha_n)\}$. The notation $\{C\}e \bullet e' = x\{C'\}@\tilde{g}$ (called *evaluation formula located at* $\tilde{g}$) denotes the following formula.

$$\forall \tilde{y}^{\mathsf{Ref}(\tilde{\alpha})}.\, \forall \tilde{j}^{\tilde{\alpha}}.\, \{C \wedge \bigwedge_i (y_i \neq \tilde{g}|_{\mathsf{Ref}(\alpha_i)} \wedge\, !y_i = j_i)\}e \bullet e' = x\{C' \wedge \bigwedge_i !y_i = j_i\}$$

Here each $y_i^{\mathsf{Ref}(\alpha_i)}$ and j^{α_i} is fresh.

5.6.3 Located Evaluation Formulae (3): Recursive Types

The development of located evaluation formulae so far assumed finite effect sets. For the present programming language this is appropriate because in the absence of recursive types, no typable program can have unbounded effects. This changes with recursive types. Consider a program, like $\mathtt{map}(\lambda x^{\mathsf{Ref(Nat)}}.x :=\!x+1)$ which takes a list of references storing numbers and increments the content of each reference in the list. If the program is denoted by f, then we might want to specify

$$\{\mathsf{T}\}f \bullet x :: y\{\mathsf{T}\}@xy \qquad\qquad \{\mathsf{T}\}f \bullet x :: y :: z\{\mathsf{T}\}@xyz$$

where $x :: y$ is the list having x as its first element and y as its second, etc. Now the effect set's size depends on f's argument. But what is the effect set S for

$$\{\mathsf{T}\}f \bullet l\{\mathsf{T}\}@S$$

when all we know (by typing) about l is that it is a list? The appropriate effect set in this case cannot be expressed directly with evaluation formulae like those discussed above.

To deal with this situation we propose using *effect comprehensions*

$$\{C_1\}e \bullet e' = x\{C_2\}@C(y). \tag{48}$$

Here C is a formula in our logic and y a variable. The intuition behind this construct is that if z is a reference cell and $C[z/y]$ holds, then the content of z may be changed by evaluating $e \bullet e'$. Conversely, if $\neg C[z/y]$, then z stores the same value before and after. Hence (48) can be taken to stand for

$$\forall y.\forall j.\{C_1 \wedge \neg C(y) \wedge !y = j\}e \bullet e' = x\{C_2 \wedge !y = j\}. \tag{49}$$

As before, (49) cannot be typed directly. The two proposals above can again be employed to solve this problem. However, this time the polymorphic approach seems easier, because, the set of reference types affected by a program is less immediately expressed, though still finite (up to tree-isomorphism). We leave a detailed investigation of effect comprehensions to a forthcoming exposition, but note in closing that reasoning with effect comprehensions is virtually as straightforward as with finite effects.

5.6.4 Polymorphic Swap

Our `swap` above in fact works for a pair of references of an arbitrary type, and is indeed typable as such in polymorphic programming languages like ML and Haskell. Following (Honda & Yoshida, 2004), we can capture its polymorphic behaviour by adding $\forall \mathrm{X}.C$ (and dually $\exists \mathrm{X}.C$) to the assertion language, with the grammar of types extended with type variables $(\mathrm{X}, \mathrm{Y}, \ldots)$ and quantifiers ($\forall \mathrm{X}.\alpha$ and $\exists \mathrm{X}.\alpha$). With this extension, we can refine (45).

$$\begin{array}{l}\forall \mathrm{X}.\forall x^{\mathsf{Ref(X)}}.\forall y^{\mathsf{Ref(X)}}.\forall i^{\mathrm{X}}.\forall j^{\mathrm{X}}. \\ \qquad \{!x = i \wedge !y = j\}u \bullet \langle x,y\rangle\{!x = j \wedge !y = i\}@xy\end{array} \tag{50}$$

The assertion should be readable naturally.

5.6.5 Circular References

We close this chapter on assertions with discussing assignments to circular references. For example, we would like to assert about $x :=!!x$ which is not well-typed in the programming language considered so far. Typing of such a reference needs recursive types, which we outline first. We take the equi-isomorphic approach (Pierce, 2002) where recursively defined types are equated iff their representation as regular trees are isomorphic. The grammar of types is extended as follows, for both the programming language and for the assertion language.

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

The typing rules do not change except for the change in types. Accordingly no change is needed in the axioms and proof rules, but one possible, yet not necessary option is to introduce a recursively defined assertion.

An assertion for $x :=!!x$ could be the following.

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

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 effort.

Similarly we can easily specify

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

where x is typed with $\mu X.\mathsf{Ref}((\mathsf{Nat} \times (\mathsf{Unit} + X)))$, the type of a mutable list of natural numbers (one may also use the null pointer as a terminator of a list). The assertion $!x = \langle 1, \mathtt{inr}(x)\rangle$ says x stores a pair of 1 and the right injection of a reference to itself, precisely capturing the graphical structure of the datum.

6 Logic (2): Axioms

The purpose of this section is to introduce axioms for deriving valid assertions in our assertion language. We take for granted the usual notions of axiom system, inference rule, deduction and the like. As is standard (Hoare, 1969), we shall assume that the axioms and rules from propositional calculus, first-order logic with equality (Mendelson, 1987) and formal number theory are freely available.

6.1 Axioms for Content Quantification

We start with the axioms for content quantification. Hoare's logic (Hoare, 1969) allows tractable reasoning about simple stateful programs because, due to the lack of aliasing, state change by assignment has a logical description, obtained from an analysis of syntactic substitution. This logical description leads to succinct logical laws and reasoning principles, because the logical operations used in the decomposition of substitution come with associated logical laws and reasoning principles.

For similarly tractable reasoning about stateful programs with aliasing we likewise need

$$\text{(CA1)}\quad [!x]\,(C_1^{\neg !x} \supset C_2) \supset (C_1 \supset [!x]C_2) \qquad \text{(CA2)}\quad [!x]C \supset C$$
$$\text{(CA3)}\quad [!x]\,(!x = m \supset C) \equiv \langle !x\rangle\,(C \wedge !x = m) \qquad \text{(CGen)}\quad \frac{C}{[!x]C}$$

Fig. 3. Axioms and Rule of Inference for content quantification.

succinct logical laws and reasoning principles, but for logical substitution. To obtain such, we need axioms and reasoning principles for content quantification. We obtain those by analogy with the axiomatisations of first-order quantification.

For example (Mendelson, 1987) axiomatises first-order universal quantification with two axioms and a single rule of inference (in addition to Modus Ponens):

- $\forall x.(A \supset B) \supset A \supset \forall x.B$ provided x does not occur in A and
- $\forall x.A \supset A[e/x]$.
- infer $\forall x.A$ from A provided x does not appear freely in assumptions.

Our axiomatisation of content quantification given in Figure 3 is analogous. First, we regard $\langle !x\rangle C$ as standing for $\neg[!x]\,(\neg C)$. There are three axioms (CA1–CA3). In (CA1), $C^{\neg !x}$ indicates C is *syntactically* $!x$*-free*, defined next.

Definition 15 (active dereference) The *active dereferences* of an expression e, $\mathsf{ad}(e)$, are inductively defined:

$$\mathsf{ad}(x) = \mathsf{ad}(\mathsf{c}) \stackrel{\text{def}}{=} \emptyset \qquad \mathsf{ad}(\mathsf{op}(\tilde{e})) \stackrel{\text{def}}{=} \bigcup_i \mathsf{ad}(e_i) \qquad \ldots \qquad \mathsf{ad}(!e) \stackrel{\text{def}}{=} \{!e\} \cup \mathsf{ad}(e)$$

The *active dereferences* of a formula C, $\mathsf{ad}(C)$, have the definition given next.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{ad}(e = e') & \stackrel{\text{def}}{=} & \mathsf{ad}(e) \cup \mathsf{ad}(e') & \mathsf{ad}(\neg C) & \stackrel{\text{def}}{=} & \mathsf{ad}(C) \\
\mathsf{ad}(C \star C') & \stackrel{\text{def}}{=} & \mathsf{ad}(C) \cup \mathsf{ad}(C') & \mathsf{ad}(\{C\}e \bullet e' = x\{C'\}) & \stackrel{\text{def}}{=} & \mathsf{ad}(e) \cup \mathsf{ad}(e') \\
\mathsf{ad}([!e]C) & \stackrel{\text{def}}{=} & (\mathsf{ad}(C) \setminus \{!e\}) \cup \mathsf{ad}(e) & \mathsf{ad}(\langle !e\rangle C) & \stackrel{\text{def}}{=} & (\mathsf{ad}(C) \setminus \{!e\}) \cup \mathsf{ad}(e) \\
\mathsf{ad}(Qx.C) & \stackrel{\text{def}}{=} & \mathsf{ad}(C) & & &
\end{array}$$

Example 4 (active dereferences)

1. T and F contain no active dereferences.
2. $!x = 3$ has $!x$ as sole active dereference.
3. In $!!x = !y$ we have three: $!y$, $!x$ and $!!x$.
4. The evaluation formula $\{!x = 2\}!f \bullet !y = z\{!z = 1\}$ has $!f$ and $!y$ as active dereferences.
5. $[!!x]\,(!!x = !y)$ has two active dereferences, $!x$ and $!y$.
6. Finally, $\forall x.!!x = !y$ has $!!x, !x$ and $!y$ as active dereferences, but the α-equivalent $\forall z.!!z = !y$ has $!!z, !z$ and $!y$. Hence active dereferences are *not* stable under renaming of bound variables.

The intuition behind $\mathsf{ad}(\cdot)$ is that if two models $\mathcal{M}_1, \mathcal{M}_2$ agree on their stateless part and on $\mathsf{ad}(e)$, then $[\![e]\!]_{\mathcal{M}_1} \cong [\![e]\!]_{\mathcal{M}_2}$, and similarly for formulae. The need for the – on first glance possibly peculiar – definition $\mathsf{ad}([!e]C) \stackrel{\text{def}}{=} (\mathsf{ad}(C) \setminus \{!e\}) \cup \mathsf{ad}(e)$, and likewise for

existential content quantification, is this: the truth-value of $[!!x]C$ does not depend on what a model stores at (the identical containing) $!!x$. It does however depend on what is being stored at $!x$. Assume that $\mathcal{M} \models !x = y$ and $\mathcal{M}' \models !x \neq y$. Then

$$\mathcal{M} \models [!!x]\,!!x = !y \qquad\qquad \mathcal{M}' \not\models [!!x]\,!!x = !y.$$

Definition 16 (syntactic $!x$-freedom) We generate the set of syntactically $!x$-free formulae, $\mathcal{S}^{\text{-}!x}$, as follows:

1. $[!x]C \in \mathcal{S}^{\text{-}!x}$, dually $\langle !x\rangle C \in \mathcal{S}^{\text{-}!x}$.
2. $C \wedge \bigwedge_i e_i \neq x \in \mathcal{S}^{\text{-}!x}$ and, dually, $\bigwedge_i e_i \neq x \supset C \in \mathcal{S}^{\text{-}!x}$, in both cases assuming that $\{!e_i\} = \mathsf{ad}(C)$ and that no occurrence of a free name in an e_i is bound in C.
3. The result of applying any of the logical connectives (including negation) or standard/content quantifiers, except $\forall x$ and $\exists x$, to formulae in $\mathcal{S}^{\text{-}!x}$ is again in $\mathcal{S}^{\text{-}!x}$.

Example 5 (syntactic $!x$-freedom)

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

Among the axioms, (CA1) corresponds to familiar $\forall x.(C_1^{-x} \supset C_2) \supset (C_1 \supset \forall x.C_2)$ except that we require C_1 to be syntactically $!x$-free instead of x-free. (CA2) is a degenerate form of $\forall x.C \supset C[e/x]$. (CA3) says that the two ways of representing logical substitutions coincide, which is important to recover all properties of semantic update (Cartwright & Oppen, 1981; Cartwright & Oppen, 1978; Morris, 1982a; Morris, 1982d; Morris, 1982c), as discussed in the next section. Finally, we add an inference rule (CGen), that is the analogue of standard generalisation, which says: "If we can derive C from the axioms, then we may conclude $[!x]C$". This rule assumes deductions without assumptions (e.g. all leaves of a proof tree should be axioms). If we *are* to use deduction with non-trivial assumptions, we demand assumptions to be syntactically $!x$-free if the deduction uses (CGen) for $!x$. By a standard argument, we obtain a deduction theorem (Mendelson, 1987). Once a deduction theorem is proven, we can use it to derive many laws for content quantification.[3]

For example, given the assumption $[!x]\,(C_1 \wedge C_2)$, we can derive $C_1 \wedge C_2$ by (CA2) and Modus Ponens. Then we obtain C_1 by the elimination rule for $\wedge$. To the latter we apply (CGen), which is possible because the assumptions are $!x$-free, to obtain $[!x]C_1$; similarly we get $[!x]C_2$, so we obtain $[!x]C_1 \wedge [!x]C_2$ by the $\wedge$-introduction rule; the other way round is similar. We also note that (CA2) is not restrictive since from $[!x]C$ we can derive $C\{\!|m/!x|\!\}$ for arbitrary m.

We now present several such laws. We begin by focussing on the universal part of the

[3] A different and equivalent axiomatisation of content quantification can be given, again following a first-order logic, by replacing the rule (CGen) with the axiom $C^{\text{-}!x} \supset [!x]C$, and closing all axioms under universal content quantification, cf. citeenderton

laws without loss of generality. Later we summarise all laws including their existential counterparts.

$$[!x]C' \quad \supset \quad [!x]((C' \supset C) \supset C) \tag{51}$$

$$[!x]C \wedge [!x]C' \quad \equiv \quad [!x](C \wedge C') \tag{52}$$

$$[!x]C \quad \supset \quad [!x][!x]C \tag{53}$$

$$[!x]C \vee [!x]C' \quad \supset \quad [!x](C \vee C') \tag{54}$$

$$[!x](C \vee C') \quad \supset \quad [!x]C \vee \langle !x\rangle C' \tag{55}$$

The existential counterpart of these laws is by dualisation discussed below. (51) allows us to infer $[!x]C'$ from $[!x]C$ when $C \supset C'$ is a tautology. The existential counterpart of (53) is:

$$\langle !x\rangle\langle !x\rangle C \quad \supset \quad \langle !x\rangle C. \tag{56}$$

These rules are reminiscent of axiomatisations for the modal "necessity" operator.[4]

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

$$\forall y.[!x]C \quad \supset \quad [!x]\forall y.C \tag{57}$$

$$[!y][!x]C \quad \supset \quad [!x][!y]C \tag{58}$$

$$\langle !x\rangle[!x]C \quad \supset \quad [!x]C \tag{59}$$

Again they have dual versions. All these entailments are logical equivalences, with the reverse direction being derivable: for (57), if we have $[!x]\forall y.C$ and x and y are distinct, then by y not free in the formula we have $\forall y.[!x]\forall y.C$, from which we conclude $\forall y.[!x]C$; (58) is already symmetric; finally the converse of (59) uses the notion of x-freedom of $[!x]C$ discussed later. We have another derived rule for first-order quantification.

$$\exists x.!x = y \tag{60}$$

This assertion does not mention content quantification but its derivation needs it.

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

$$\neg[!x]\,!x \neq y \tag{61}$$

$$C^{-!x} \quad \supset \quad [!x]C \tag{62}$$

Please note that the reverse of (62) does not hold: $[!x]\,!x = !x$ is true, despite $!x = !x$ not being syntactically $!x$-free. (61) is easily understood as an analogue of $\forall x.(x \neq y) \supset y \neq y$ ($\equiv \mathsf{F}$).

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

$$\neg[!x]C \quad \equiv \quad \langle !x\rangle\neg C \tag{63}$$

$$[!x](!x = m \supset C) \quad \equiv \quad \langle !x\rangle(C \wedge !x = m) \tag{64}$$

(63) directly comes from our definition of existential quantification in our axiom system. The second law (64) is (CA3), which relates two dual quantifiers *without dualisation*: its

[4] We believe that it is possible to give an alternative and equivalent modal axiomatisation of content quantification, although what may be a minimal and natural set of such axioms is not clear.

origin lies in the logical equivalence $\forall x.(x = m \supset C) \equiv C[m/x] \equiv \exists x.(C \wedge x = m)$, briefly mentioned in the introduction.

From (64) we immediately infer the equivalence between the two forms of logical substitutions introduced in Definition 9:

$$C\{\!|e'/!e|\!\} \quad \equiv \quad C\overline{\{\!|e'/!e|\!\}} \tag{65}$$

for any C, e' and $!e$. The axiom (64) plays a fundamental role in the present theory. From this we also infer:

$$[!x]C \quad \supset \quad C\{\!|e/x|\!\} \tag{66}$$

$$C\{\!|e/x|\!\} \quad \supset \quad \langle !x\rangle C \tag{67}$$

Proposition 6 *All the laws (51) - (67) are derivable.*

Proof
The derivations are straightforward. For example, we derive (51) as follows:

1. $[!x]C \supset (([!x]C \supset C') \supset C')$	(Tautology)
2. $[!x]([!x]C \supset (([!x]C \supset C') \supset C'))$	(CGen, 1)
3. $[!x]C \supset [!x]([!x]C \supset C') \supset C')$	(CA1, 2)
4. $[!x]C \supset C$	(CA2)
5. $[!x]C \supset [!x]((C \supset C') \supset C')$	(3, 4)

For (58) we use:

1. $[!y][!x]C \supset C$	(CA2)
2. $[!y]([!y][!x]C \supset C)$	(CGen, 1)
3. $[!y][!x]C \supset [!y]C$	(CA1, 2)
4. $[!x]([!y][!x]C \supset [!y]C)$	(CGen, 3)
5. $[!y][!x]C \supset [!x][!y]C$	(CA1, 4)

The other derivations are equally easy. □

For the remaining derived laws for content quantifications, we introduce the semantic version of Definition 16.

Definition 17 C is $!e$*-free* when $[!e]C \equiv C$.

Remark 2 We usually regard $\equiv$ in Definition 17 as a syntactic notion (i.e. derivability of $[!x]C \equiv C$ as a theorem in the present logic, involving the axioms in the present section as well as the ambient logical system such as Peano Arithmetic).

By (62), any syntactically $!x$-free assertion is $!x$-free but the reverse implication does not hold, for example $!x = !x$ is semantically but not syntactically $!x$-free. Some examples of $!x$-free formulae follow.

Example 6 ($!x$-freedom)

1. As noted, any syntactic $!x$-free formula is $!x$-free. In particular T and F are $!x$-free.

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

Since $[!x]C \supset C$ for any C by (CA2), we know C is $!x$-free if and only if $C \supset [!x]C$. Note $\langle !x\rangle C \equiv C$ also characterises $!x$-freedom (which is often useful in practice) and that the converse of (59) 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 7 *If C_1 is $!x$-free, then:*

$$[!x]\,(C_1 \vee C_2) \quad \equiv \quad C_1 \vee [!x]C_2 \tag{68}$$

$$\langle !x\rangle\,(C_1 \wedge C_2) \quad \equiv \quad C_1 \wedge \langle !x\rangle C_2 \tag{69}$$

$$[!x]\,(C_1 \supset C_2) \quad \equiv \quad C_1 \supset [!x]C_2. \tag{70}$$

Proof
By duality and since (70) merely rephrases (68), it suffices to derive (68).

$$[!x]\,(C_1^{-!x} \vee C_2) \quad \supset \quad \langle !x\rangle C_1^{-!x} \vee [!x]C_2 \quad \equiv \quad C_1^{-!x} \vee [!x]C_2$$

$$C_1^{-!x} \vee [!x]C_2 \quad \equiv \quad [!x]C_1^{-!x} \vee [!x]C_2 \quad \supset \quad [!x]\,(C_1^{-!x} \vee C_2)$$

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

Note (70) 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 8 (derived axioms)

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

Proof

For (1):

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

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

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

For (3), the first statement:

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

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

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

For the other direction, again with m fresh:

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

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

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

□

Finally, as a simple application of content quantification, we calculate an example from the

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

Fig. 4. Axioms for evaluation formulae.

introduction.

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

where $(*)$ uses $\langle !x\rangle\,(\langle !x\rangle C\wedge C')\equiv\langle !x\rangle C\wedge\langle !x\rangle C'$, which is direct from Proposition 7.

6.2 Axioms for Evaluation Formulae

The set of axioms for evaluation formulae are given in Figure 4. With the exception of (e8) and (ext), all are unchanged from the axioms in (Honda *et al.*, 2005). We assume the following convention used throughout the paper.

Convention 5 *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 (3)), while $C,C',\ldots$ still range over general formulae.*

In (e8), we use content quantifications to stipulate hypothetical entailment, cf. (41) in Section 5.5.1 (which is closely related with Kleymann's strengthened consequence rule (Kleymann, 1998)). In the rule, we assume the $\tilde{w}$ to exhaust all active dereferences in C,C_0,C_0' and C'. (e2) and (e8) together give the following axiom which is often useful:

$$\{C_1\}\, x\bullet y=z\,\{C_1'\}\;\wedge\;\{C_2\}\, x\bullet y=z\,\{C_2'\}\;\supset\;\{C_1\wedge C_2\}\, x\bullet y=z\,\{C_1'\wedge C_2'\} \qquad (71)$$

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

In (ext), the extensionality formula augments the corresponding formulae for alias-free sublanguage in (Honda *et al.*, 2005) with located assertions.

Definition 18 (extensionality formulae) Let $\Delta=\tilde{r}:\mathsf{Ref}(\tilde{\gamma})$ and x and y be typed as $\alpha\Rightarrow\beta$. Then set set:

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

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

Fig. 5. Axioms for located evaluation formulae.

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

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

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

The use of write effects is fundamental to describe extensionality since, without write effects, there could be different effects on unspecified memory cells.

In Figure 5, we list axioms for located assertions, which refine the original axioms in Figure 4 (except (ext) which is already about located assertions), as well as adding two new axioms for manipulating write effects. The axioms from (le1) to (le6) simply add write effects to assertions. However (le7) allows us to add universally content-quantified stateful formulae to the pre/post conditions, strengthening (e7). The reader may recall having already seen an instance of this rule in (39) and (40), Section 5.5, Page 24. (le7) is more general than (e7) in that weakened assertion can be stateful. At the same time (le7) is justifiable using (e7). For concreteness, take the assertion (39) in Section 5.5:

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

To this assertion we apply the first-order law $\forall x.C\supset C[e/x]$ to obtain for a concrete y:

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

Now we use (e7) and get:

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

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

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

Hence by (e8) we obtain:

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

as required. As in this example, all these rules are easily justifiable using the axioms rules in Figure 4.

Finally (weak) and (thin) correspond to the first two implications in Proposition 5. They are reminiscent of the weakening rules and thinning rules in various type disciplines, hence the names.

6.3 Axioms for Data Types

Now we introduce axioms for data types. All axioms such as "$x^{\mathsf{Unit}} = ()$" have already appeared in (Honda *et al.*, 2005). The only difference is that we no longer have the axiom saying syntactically distinct reference names never equated, 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. (Honda & Yoshida, 2004). Here we allow reference types to appear anywhere in types, so that data structures can now be destructively updated in their parts. In the next section we shall see a generalised assignment axiom which can treat assignment of an arbitrary data structure to an arbitrary (mutable part of) data structure, which is quite common in systems programming (e.g. a part of a record referred to by another record is replaced with another pointer).

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

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

together with the typing rules:

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

The construction above assumes that the identifier of each array to be used is given as a constant (ranged over by $a, b, \ldots$). We further regard expressions $a[0], a[1], \ldots, a[n-1]$ for some n as values of reference types. These values form part of the domain of a concrete store: it is also convenient, though not necessary, to include them as part of a reference basis so that the size of an array is determined from a basis. For statically sized arrays, this offers clean typing, though there are other approaches. For defining the dynamics of arrays there are various alternative approaches that differ mostly in how out-of-bounds errors are handled. Here we assume that an out-of-bound access generates nil of the corresponding reference type; the dereference of nil leads to err, and err, when evaluated, leads to err of the whole expression, which follows a standard treatment of type error (Milner, 1978).

Terms are augmented accordingly:

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

where, in $e[e']$, we type e with an array type (say $\alpha[]$) and e' with Nat, with the whole term given the type in $\mathsf{size}(e)$ (which denotes the size of an array e), we type e with an array type, with the whole term typed with Nat; $\mathsf{nil}^{\mathsf{Ref}(\alpha)}$, which denotes the null pointer and

whose type we usually omit, is typed by $\mathsf{Ref}(\alpha)$; and err^{α} denotes a (dereference) error of type α, for each α.

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

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

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

$$\forall i, j.\ (\ 0 \leq i, j \lneq \mathsf{size}(x)\ \ \wedge\ \ i \neq j\ \ \supset\ \ x[i] \neq x[j]\ ) \tag{72}$$

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

$$(\mathsf{size}(x) = \mathsf{size}(y)\ \wedge\ \forall i.\ (\ 0 \leq i < \mathsf{size}(x) - 1\ \supset\ x[i] = y[i]\ )\ \ \supset\ \ x = y \tag{73}$$

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

$$x \neq y\ \ \supset\ \ \forall i, j.\ (\ 0 \leq i < \mathsf{size}(x) - 1\ \wedge\ 0 \leq j < \mathsf{size}(y) - 1\ \ \supset\ \ x[i] \neq y[j]\ ) \tag{74}$$

which says two distinct arrays never overlap (note this axiom is not applicable to, for example, C). Note that (74) is equivalent to:

$$\exists i, j.\ (\ 0 \leq i < \mathsf{size}(x) - 1\ \wedge\ 0 \leq j < \mathsf{size}(y) - 1\ \wedge\ x[i] = y[j]\ )\ \ \supset\ \ x = y. \tag{75}$$

For those axioms which involve nil and err, see Remark 3 below.

In models, we may treat an array as simply a function from natural numbers to references such that it maps all numbers within its range to distinct references and others to nil, cf. (Apt, 1981). Other constraints can be considered following the axioms as given above.

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

Remark 3 (axioms for nil and err) For reference we list basic axioms involving nil and err. While these constructs are introduced for a wholesome semantic treatment of assertions, the need to use them may not be as frequent as other "normal" term constructors (however their treatment becomes essential when we consider e.g. error recovery routines). First of all, out-of-bound errors are treated as:

$$i \gtrsim \mathsf{size}(x)\ \ \supset\ \ x[i] = \mathsf{nil} \tag{76}$$

Further we stipulate:

$$!\mathsf{nil}\ \ =\ \ \mathsf{err}. \tag{77}$$

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

$$\mathcal{E}\,(\mathsf{err})\ \ =\ \ \mathsf{err} \tag{78}$$

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

7 Logic (3): Judgements and Proof Rules

7.1 Judgements and their Semantics

Following Hoare (Hoare, 1969), a judgement in the present program logic consists of two formulae and a program, augmented with a fresh name called *anchor*:

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

(We often drop typing annotations for readability.) 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 $\{C\}\ M :_u \{C'\}$, M 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

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

Henceforth we only treat judgements which are well-typed. Following Convention 3 (5), $\{C\}\,M\,\{C'\}$ stands for $\{C\}\,M :_u \{u = () \wedge C'\}$ where u is a fresh name, typed as Unit.

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

Definition 19 (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 *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 and 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 sheer necessity because, for example, a store for $x : \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat})) \vdash !!x : \mathsf{Nat}$ should have not only x but another reference which stores $!x$ (the same is true for auxiliary names). Second this is consistent with $\cong_{\mathcal{D}}$ (as $\cong$) being considered under all extensions of a given basis, cf. Section 2.3/Section 3.1. Formally we stipulate as follows (see Notation 2, Page 13, for the notation $(\xi \cdot u : V,\ \sigma')$).

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

Definition 20 (semantics of judgements) We say the judgement $\models\{C\}\,M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ is *valid*, written $\models\{C\}\,M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, iff: for each model $\mathcal{M}^{\Gamma';\Delta'} \stackrel{\text{def}}{=} (\xi,\sigma)$ where $\Gamma' \supseteq \Gamma$, $\Delta' \supseteq \Delta$, $\Gamma';\Delta' \vdash C$ and $\Gamma' \cdot u:\alpha;\Delta' \vdash C'$, if $(\xi,\sigma) \models C$ then $(M\xi,\ \sigma) \Downarrow (V,\sigma')$ such that $(\xi \cdot u:V,\ \sigma') \models C'$.

Note that the standard practice of considering all possible models for validity means considering all possible forms of aliasing conforming to precondition C.

7.2 Proof Rules (1): Compositional Rules

We now present the proof rules for deriving valid judgements for imperative PCFv with aliasing. There is one compositional proof rule for each programming language construct which precisely follows syntactic structure. There are additional 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 (Honda *et al.*, 2005) except for a minimal and unavoidable refinement of the rule for assignment, which now uses $\{\!|e'/!e|\!\}$ instead of syntactic substitution $[e'/!e]$ (cf. Section 4) and an adaptation to our generalised syntax in dereference and assignment. This is in accordance with our logical language, which increments that in (Honda *et al.*, 2005) 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 the precise correspondence between type structure and logical apparatus, is central to the family of program logics under investigation by the present authors.

Following (Honda *et al.*, 2005), we stipulate the following conventions for proof rules.

Convention 6 (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 that 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) to 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 5, $A, A', B, B', \ldots$ 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 15).

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

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

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

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

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

$$[In_1]\ \frac{\{C\}\ M :_v \{C'[\mathsf{inj}_1(v)/u]\}}{\{C\}\ \mathtt{in}_1(M) :_u \{C'\}} \qquad [Case]\ \frac{\{C^{\tilde{}\tilde{x}}\}\ M :_m \{C_0^{\tilde{}\tilde{x}}\} \quad \{C_0[\mathsf{inj}_i(x_i)/m]\}\ M_i :_u \{C'^{\tilde{}\tilde{x}}\}}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}}$$

$$[Pair]\ \frac{\{C\}\ M_1 :_{m_1} \{C_0\} \quad \{C_0\}\ M_2 :_{m_2} \{C'[\langle m_1, m_2\rangle/u]\}}{\{C\}\ \langle M_1, M_2\rangle :_u \{C'\}} \qquad [Proj_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

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

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

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

The compositional proof rules of the program logic are given in Figure 6. [*Op*] is a general rule for first-order operators, and subsumes [*Const*] when arity is zero. As noted already, the shape of all the rules in Figure 6 are identical character-by-character with the compositional rules for the imperative PCFv without aliasing except for [*Assign*] which uses logical substitution and hence content quantification. Leaving detailed explanation of the remaining rules to (Honda *et al.*, 2005), we illustrate the two new rules for imperative constructs, [*Deref*] and [*Assign*] in the following.

[***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 (Honda *et al.*, 2005)).

$$[Deref\text{-}Org]\ \frac{-}{\{C[!x/u]\}\ !x :_u \{C\}} \tag{79}$$

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

$$\frac{-}{\{Even(!x)\}\ !x :_u\ \{Even(u)\}} \tag{80}$$

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. An example follows (below and henceforth we often do not expand simple applications of [*Cons*]).

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

As another simple example, let C be given by:

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

Then we infer:

$$\{C \wedge !x = 1\}\ !(fx) :_u \{u = 2\ \wedge\ !x = 2\} \tag{81}$$

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

Note that 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.

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 $(M\xi, \sigma) \Downarrow_m (\xi \cdot m : V, \sigma')$ for $(M\xi, \sigma) \Downarrow (V, \sigma')$).

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

The second inference above is valid because dereferencing does not change the store, noting the freshness of m.

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

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

Please remember from Section 7 that [*Assign*] omits mentioning the conclusion's anchor (of Unit type) and a substitution of $()$, the unique Unit-value: $\{C\}\,M := N\,\{C'\}$ stands for $\{C\}\,M := N :_u \{u = () \wedge C'\}$ with u fresh. This is justified because $C[()/x] \equiv C$ always holds when x has the unit type. Hence we can always ignore this substitution. A simple example of its usage follows (the first line is already reasoned in the previous page).

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

Line 3 is derived as:

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

The rule may be understood by contrasting it with the corresponding rule for the non-aliased sublanguage in (Honda *et al.*, 2005). There the assignment rule reads:

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

There are two differences between this original rule and [*Assign*] in Figure 6. First, [*AssignOrg*] only allows a variable as the left-value, while the [*Assign*] allows an arbitrary program. Second, the original rule uses syntactic substitution, while the present system uses the logical counterpart (cf. Section 5.3). The corresponding rule in the present context (only incorporating the second point) is:

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

Clearly [*AssignVar*] is derivable from [*Assign*] through [*Var*].

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

$$\{x = y \wedge \mathit{Even}(!!y)\}\ !x := !!y + 1\ \{\mathit{Odd}(!!x) \wedge \mathit{Odd}(!!y)\} \tag{82}$$

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

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

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

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

Above the use of e as a program indicates that it is a term in the logic and a program in our programming language at the same time. In various programming examples, we often assign part of a complex data structure to a part of another complex data structure. The rule [*AssignSimple*] gives a general rule for such cases.

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

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

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

7.3 Proof Rules (2): Structural Rules

As already mentioned, structural rules manipulate formulae only. They are important for a logic's expressivity. A well-known example of a structural rules is:

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

With one exceptions our structural rules are unchanged from (Honda *et al.*, 2005), where much illustration can be found, and their details are discussed later, in the next section, where we present located proof rules, which are a derivable generalisation from which the original structural rules can easily be recovered.

7.4 Located Judgement and their Proof Rules

One of the central problems in large-scale software development is to prevent inadvertent interference between programs through shared variables, especially in the presence of aliasing. The located assertions in Section 5.6 address this concern by delineating part of the store a program may affect. Below we extend this idea to judgements. Roughly, we consider

$$\{C\}\,M :_u \{C'\}\,@\,\tilde{e} \quad \stackrel{\text{def}}{=} \quad \{C \wedge y \neq \tilde{e} \wedge !y = i\}\,M :_u \{C' \wedge y \neq \tilde{e} \wedge !y = i\}.$$

where y and i are fresh and distinct. Once again the problem is how to type this judgement. As before there are two immediate approaches, the polymorphic one where – omitting straightforward details – y and i are respectively typed as $\mathsf{Ref}(\mathrm{X})$ and X for a fresh X, and the monomorphic counterpart, formalised next.

Definition 21 (located judgement) Given C, $\Gamma;\Delta \vdash M : \alpha$ and C', as well as a finite set of terms $\{\tilde{e}\}$ having types $\{\mathsf{Ref}(\alpha_1),...,\mathsf{Ref}(\alpha_n)\}$ and not containing a dereference. A *located judgement* has the following shape.

$$\{C\}\, M^{\Gamma;\Delta;\alpha} :_u \{C'\}\, @\, \tilde{e} \tag{83}$$

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

$$\models \{C \wedge \bigwedge_i y_i \neq \tilde{e}|_{\mathsf{Ref}(\alpha_i)} \wedge !y_i = j_i\}\, M^{\Gamma;\Delta;\alpha} :_u \{C' \wedge \bigwedge_i !y_i = j_i\}$$

holds, with the y_i and j_i fresh and distinct and typed as $\mathsf{Ref}(\alpha_i)$ and α_i respectively (cf. Section 5.6). As in located assertions, $\tilde{e}$ in (83) 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.

Starting from Section 5, we have seen several examples of using located assertions. Like located judgements, 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 present some examples of located judgements.

Example 8 (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, in particular, x is sole reference whose content may change.
2. Let $M \stackrel{\text{def}}{=} \mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ x \times f(x-1)$. Then we have
$$\{\mathsf{Fact}(f)\}\ M^{\Gamma:\mathsf{Nat}} :_u \{u = x!\} @ \emptyset$$
with $\Gamma \stackrel{\text{def}}{=} f : \mathsf{Nat} \Rightarrow \mathsf{Nat} \cdot x : \mathsf{Nat}$ and $\mathsf{Fact}(f) \stackrel{\text{def}}{=} \forall i \lneq x.\{\mathsf{T}\}\, f \bullet i = i!\{\mathsf{T}\}@\emptyset$.
3. For the same M, we have:
$$\{\mathsf{Fact}'(f)\}\ M^{\Gamma:\mathsf{Nat}} :_u \{u = x!\}\ @\, w$$
where $\mathsf{Fact}'(f) \stackrel{\text{def}}{=} \forall i \lneq x.\{\mathsf{T}\}\, f \bullet i = i!\{\mathsf{T}\}@w$. Note that 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.

Valid located judgements are derivable with the proof rules for non-located judgements by translating located judgements to non-located ones. A more efficient method is to use compositional proof rules which are derivable in the original system but which are tailored

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

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

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

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

$$[In_1]\ \frac{\{C\}\ M :_v \{C'[\mathsf{inj}_1(v)/u]\}@\tilde{e}}{\{C\}\ \mathtt{in}_1(M) :_u \{C'\}@\tilde{e}} \qquad [Case]\ \frac{\{C^{\tilde{}\tilde{x}}\}\ M :_m \{C_0^{\tilde{}\tilde{x}}\}@\tilde{e} \quad \{C_0[\mathsf{inj}_i(x_i)/m]\}\ M_i :_u \{C'^{\tilde{}\tilde{x}}\}@\tilde{e_i}'}{\{C\}\ \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i).M_i\}_{i\in\{1,2\}} :_u \{C'\}@\tilde{e}\tilde{e_1'}\tilde{e_2'}}$$

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

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

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

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

Fig. 7. Derivable proof rules with located judgements.

for located judgements. The proof rules for located judgements are given in Figure 7 (for compositional rules) and Figure 8 (for structural rules).

The compositional rules are entirely straightforward, closely following Figure 6, just accounting for the effects, usually by accumulating effects computed in the premise. The only exceptions are [*Var*, *Abs*, *App*, *Assign*]. In [*Var*] we declare the effect to be empty by fiat. The correctness of this is immediate from the semantics of variables. [*Abs*] internalises the premise's effect $\tilde{e}$ into the conclusion's evaluation formula. [*App*] does the inverse of this. The only place where new effects are inevitable is [*Assign*], which demands that C_0 says m (the target of writing) is in the write effect (the set membership notation "$\in$" is understood to denote a disjunction of equations).

Among the structural rules in Figure 8, five may deserve illustration, [*Weak*], [*Thinning*], [*Invariance*], [*Cons-Aux*] and [*Rename*]. All others are straightforwardly derived from their non-located counterparts, given in (Honda *et al.*, 2005). Conversely, all non-located structural rules of our logic are immediately obtained by simple removal of the effect set.

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

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

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

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

$$[\mathit{Invariance}]\ \frac{\{C\}\ M :_u \{C'\}@\tilde{e} \quad C_0 \text{ is } !\tilde{e}\text{-free}}{\{C\ \wedge\ C_0\}\ M :_u \{C'\wedge C_0\}@\tilde{e}} \qquad [\mathit{Rename}]\ \frac{\{C\}\ M :_u \{C'\}@\tilde{e} \quad \sigma \text{ injective renaming}}{\{C\sigma\}\ M\sigma :_{\sigma(u)} \{C'\sigma\}@\tilde{e}\sigma}$$

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

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

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

Fig. 8. Derivable structural rules for located judgements.

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

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

In Line 3, we have $(!x = 3)\{\!\{n/!x\}\!\} \equiv n = 3$ by Proposition 8 (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 fewer 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

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

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

Fig. 9. Derivable invariance rules for located judgements.

in x, it does not alter its content. As an example usage of [*Thinning*], we infer, noting $C\{\!|!x/!x|\!\} \equiv C$ (cf. Proposition 8 (3)):

$$\begin{array}{lll} 1. & (!x = i)\{\!|!x/!x|\!\} \ \equiv\ !x = i \ \supset\ x \in \{x\} & \\ \hline 2. & \{!x = i\}\ x := !x\ \{!x = i\}@\ x & \text{(Assign-Simple)} \\ \hline 3. & \{\mathsf{T}\}\ x := !x\ \{\mathsf{T}\}@\ \emptyset & \text{(Thinning)} \end{array}$$

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*]. This rule says that, if we know that a program only touches a certain set of references, and if C_0 only asserts on a state which does not concern (the content of) these references, then C_0 can be added to pre/post conditions as invariant for that program. In practice, we may use the two derivable (and essentially equivalent) rules given next (derivability is through Proposition 8 (3)).

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

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

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

$$\frac{\vdash_{\text{Hoare}} \{C\}\ P\ \{C'\} \quad P \text{ does not touch variables in } C_0}{\vdash_{\text{Hoare}} \{C \wedge C_0\}\ P\ \{C' \wedge C_0\}} \tag{84}$$

and the invariance rule for non-located judgements (from (Honda *et al.*, 2005), here omitted):

$$\frac{\{C\}\ M :_u\ \{C'\}}{\{C \wedge A\}\ M :_u\ \{C' \wedge A\}} \tag{85}$$

The rule may also be regarded as an analogue of a similar rule studied by Reynolds, O'Hearn and others in (Reynolds, 2002; O'Hearn *et al.*, 2004). Section 10.3 has a full technical comparison. Since a weakened stateless formula A in (85) is by definition $!x$-free for any x, [*Invariance*] above subsumes (85) (except we are now using located judgements). On the other hand, [*Invariance*] is justifiable using (85), cf. Section 6.2.

$$[\textit{Op-eoi}]\ \frac{\{C_i\}\ M_i :_{m_i} \{C_i'\}@\tilde{e}_i\ \ (1\le i\le n)\quad \bigwedge_i \langle !\tilde{e}_{i+1}..\tilde{e}_n\rangle C_i'\ \supset\ C'[\mathsf{op}(m_1..m_n)/u]}{\{\bigwedge_i [!\tilde{e}_1..\tilde{e}_{i-1}]C_i\}\ \mathsf{op}(M_1,...,M_n) :_u \{C'\}@\tilde{e_1}...\tilde{e_n}}$$

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

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

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

Fig. 10. Evaluation-order-independent proof rules for located judgements.

7.4.1 Evaluation Order Independence

The derived invariance rules can further be combined with compositional rules for located judgements in Figure 7 to obtain proof rules which are independent from particular evaluation order, in the sense that the correctness of the inference does not depend on the order of evaluation of expressions appearing in the rule (recall the proof rules for operators, applications, pairs, etc. all assume a fixed evaluation order, i.e. from left to right). Evaluation-order independence (EOI for short) in the most general case holds when two (or more) expressions involved only write to separate stores and, moreover, their resulting properties only rely on invariants which hold regardless of the state change induced by other expressions. Here we use a slightly stronger constraint, when the properties of each expression does not at all depend on written sets of the remaining expressions. Figure 10 lists the EOI-refinement of (located) operator/application/assignment/pairing rules. These rules are all inferred from the original rule together with two variants of the invariance rule, [*InvUniv*] and [*InvEx*].

We illustrate the situation for sequential composition, recalling that $M;N$ is short for $(\lambda().N)M$. For *modular* reasoning we would like to infer a judgement for $M;N$ from judgements $\{C_1\}\ M\ \{C_1'\}$ and $\{C_2\}\ N\ \{C_2'\}$, where C_1, C_1' should not talk about things that are only relevant for inferring $\{C_2\}\ N\ \{C_2'\}$ and vice versa. Ideally we would like a rule as easily applicable as:

$$\frac{\{C_1\}\ M\ \{C_1'\}\quad \{C_2\}\ N\ \{C_2'\}}{\{C_1\wedge C_2\}\ M;N\ \{C_1'\wedge C_2'\}} \tag{86}$$

But this is unsound. The execution of M might invalidate assumptions inscribed in C_2. Similarly, running N may destroy the guarantees made by C_1'. However, if we'd knew that C_2's truth-value was independent from M's effects, and that C_1' was likewise isolated from N's destructive updates, (86) would in fact be admissible. With content quantification, this is easily expressed: assume all of M's effects were in $\tilde{e_1}$, then $[!\tilde{e_1}]C_2$ was $!\tilde{e_1}$-free, i.e. independent from M's effects. Similarly, with N's effects in $\tilde{e_2}$, $\langle !\tilde{e_2}\rangle C_1'$ is $!\tilde{e_2}$-free.

Hence the following refinement of (86) is sound:

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

It is noteworthy that this rule does *not* require $\tilde{e_1}$ and $\tilde{e_2}$ to be disjoint, or that C_2 does not mention names in $\tilde{e_1}$ and vice versa. The rule directly infers a judgement for a sequenced pair of programs from independent judgements for the component programs. Here we show a simple example usage.

$$\frac{\begin{array}{ll} 1\ \{\mathsf{T}\}\ x := 2\ \{!x = 2\}@x & \textsf{(AssignS)} \\ \hline 2\ \{\mathsf{T}\}\ y :=!z\ \{!y = !z\}@y & \textsf{(AssignS)} \end{array}}{3\ \{\mathsf{T}\}\ x := 2; y :=!z\ \{\langle !y\rangle\, !x = 2 \wedge\ !y =!z\}@xy \quad \textsf{(Seq-I)}}$$

Note $\langle !y\rangle\, !x = 2$ is equivalent to $x \neq y \supset !x = 2$. We used the following located version of [*AssignS*]:

$$[\textit{AssignS}] \quad \{C\{\!|e_2/!e_1|\!\}\}\ e_1 := e_2\ \{C\}@\tilde{e} \qquad (C \supset e_1 \in \tilde{e})$$

Similarly, one obtains EOI-rules for operators, application, assignment etc., as given in Figure 10. All EOI-rules are proved from the corresponding original rule together with Invariance rules [*InvUniv*] and [*InvEx*]. In later sections we shall show a few examples using located assertions and judgements. Located judgements also play an essential role for proving observational completeness, one of the basic results about our logic, briefly discussed in Section 10.

We close this section with a result relating derivability between located and unlocated judgements. The proof is easy and omitted (to derive [*Thinning*] we need [*Cons-Aux*]). Below a *translation* of a located judgement is one of the instances of those presented in Definition 21.

Proposition 9 *$\{C\}\ M :_m \{C'\}@\tilde{g}$ is derivable in the proof rules for located judgements iff its translation is derivable in the proof rules for non-located judgements.*

7.5 Proof Rules for Imperative Idioms

For reasoning about programs written in an imperative idiom, derived proof rules are sometimes simpler to apply directly than the original rules. Figure 11 lists several located proof rules for this purpose. The initial four assignment rules are directly derivable from the general assignment rule in Figure 7. The next two rules for the one-branch conditional are also easily derivable from the general conditional rule in Figure 7. In [*IfThenSimple*], we assume e is also a term of boolean type in the assertion language (in fact any term e of a boolean type becomes a formula by $e = \mathsf{t}$, though such translation is seldom necessary).

The two rules for while loops augment the standard total correctness rule by Floyd (Floyd, 1967). In both rules, e' (of Nat-type) functions as an index of the loop, which should be decremented at each step. In [*WhileSimple*], the guard is a simple expression. In [*While*], the guard is a general program, possibly with a side effect (which however should not increase an index). A^b means that if there is a primary name in A, it must be

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

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

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

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

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

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

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

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

Fig. 11. Located proof rules for imperative idioms.

b. Both rules are directly derivable from the original rules through the standard encoding, as illustrated in detail in (Honda *et al.*, 2005). Finally the aforediscussed [*Seq-I*] (I is for independence) is the EOI-version of the standard rule [*Seq*].

One of the notable aspects of the presented logic is uniform treatment of data types. 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 with a corresponding axiomatisation. Figure 12 presents the located version of the proof rules for arrays. [*Array*], together with the axioms 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

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

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

Fig. 12. Located proof rules for arrays.

as a string of references, cf. (Apt, 1981). The second rule in Figure 12 is a derivable version of [*Array*] for simple expressions which is often useful. Below we give the reading of [*SimpleArray*].

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

In comparison [*Array*] rule just adds 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 about using these rules. For example if you wish to assign a value to an array at a particular index, which is an operation often found in practice, we can simply specialise e and e' in [*ArraySimple*] to reach the following rule:

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

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

$$C\{\!|\,e'\,/\,!a[e]\,|\!\} \quad \stackrel{\text{def}}{=} \quad \exists m.(\langle !\,a[e]\,\rangle\,(C \wedge\ !a[e] = m)\ \wedge\ m = e') \tag{87}$$

In the right-hand side of (87), 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 (Apt, 1981), 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.

8 Elimination of Content Quantification, Soundness

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

8.1 Elimination of Content Quantification

Using the axioms for content quantification introduced in Section 6, we establish a major technical result about our logic, eliminability of content quantification. In other words, any assertion written using content quantification can be equivalently expressed without. Before going into technical development, we discuss this fact.

- The result clarifies the logical status of these modal operators; in particular, semantically, we now know they add no more complexity than (in)equations on reference names. Since (in)equations on reference names can be easily defined using content quantifiers, we know these two notions – quantifying over content of references and discussing equalities of reference names – are essentially one and the same thing.
- As a consequence, apart from the use of evaluation formulae, validity in the assertion language is that of the standard predicate calculus with equality.

- The elimination procedure only uses the axioms for content quantifications discussed in Section 6.1 combined with the well-known axioms for equality and (standard) quantifiers. Thus, relative to the underlying axioms of the predicate calculus with equality as well as those for evaluation formulae, the axioms give complete characterisation of these modal operators.

The arguments towards the elimination theorem reveal the 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.

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 (as the transformation is local, this suffices).

1. We transform 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 does not contain active dereference while C_2 extracts all active dereferences occurring in C. This step is not necessary strictly speaking but contributes to the concision 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 the close connection between content quantification and logical substitution.

Proposition 10 *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 record it anyway:

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

hence done. □

Below the condition $z \notin \{x,y\}$ is not substantial since z can be renamed by α-convertibility.

Lemma 1 *The following equivalences hold with $\star \in \{\wedge, \vee, \supset\}$ and $\mathcal{Q} \in \{\forall, \exists\}$.*

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

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 $\wedge$:

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

For $\forall$:

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

Finally negation:

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

At the last step we again use self-duality of logical substitution. □

Now we move to the second step.

Lemma 2 *Assume C does not contain content quantification and first-order quantification. Then we can rewrite $\exists \tilde{x}.C$ in the following form up to logical equivalence:*

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

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

Proof

We construct C_n inductively: first we set $C_0 \stackrel{\text{def}}{=} C$. Now assume that C_n is of the form $C_n[!e_n]$ where $!e_n$ is active in C_n and e_n does not contain any dereferences. Then we set:

$$C_{n+1} \quad \stackrel{\text{def}}{=} \quad C[c_n] \wedge r_n = e_n$$

with c_n, r_n being fresh. Since C has only a finite number of active dereferences, the inductive construction will come to a halt eventually, say at C_m, i.e. C_m is free from active dereferences. Then we set $C' \stackrel{\text{def}}{=} C_m$. Logical equivalence is immediate. □

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

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

Write $[\![(!u = z)\{\!|m/!x|\!\}]\!]$ for the final formula above. Using notation from Lemma 2, and assuming C does not contain content quantifications, we reason (with m etc. fresh), and noting, when m is fresh, we have $C\{\!|m/!x|\!\} \equiv \langle !x\rangle\,(C \wedge !x = m)$:

$$\begin{array}{lcll}
\langle !x\rangle C & \equiv & \exists m.C\{\!|m/!x|\!\} & \text{(Lem.10)} \\
& \equiv & \exists m.(\exists \tilde{r}\tilde{c}.(\ (\wedge_i !r_i = c_i) \ \wedge \ C'))\{\!|m/!x|\!\} & \text{(Lem.2)} \\
& \equiv & \exists m.(\exists \tilde{r}\tilde{c}.(\ (\wedge_i !r_i = c_i)\{\!|m/!x|\!\} \ \wedge \ C')) & \text{(Lem.1)} \\
& \equiv & \exists m.(\exists \tilde{r}\tilde{c}.(\ (\wedge_i [\![(!r_i = c_i)\{\!|m/!x|\!\}]\!]) \ \wedge \ C')) &
\end{array}$$

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

Theorem 1 *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'.*

The elimination procedure also tells us:

Proposition 11 *For any C, $[!x]C$ is equivalent to a formula of the shape:*

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

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

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

We conclude this subsection with the following observation. Let $x = y$ be an equation on reference names. It is easy to check this equation is logically equivalent to $[!x]\,[!y]\,!x = !y$, except when x and y are of the type $\mathsf{Ref}(\mathsf{Unit})$. Thus we can replace all (in)equations on reference names with content quantifications as far as we exclude the trivial store of type $\mathsf{Ref}(\mathsf{Unit})$ from our discussion. Together with Theorem 1, we conclude that content quantifications and reference name (in)equations are mutually representable.

8.2 Soundness

In this subsection we present a key result about our logic, soundness of axioms and proof rules for non-located and located judgements.

8.2.1 Soundness of Proof Rules

We have already seen in Section 7 that [*Deref*] and [*Assign*], the only non-structural rules which differ in comparison with their counterparts in (Honda *et al.*, 2005), are semantically justifiable. As noted there, all other rules are similarly easily justified, with the proofs, given next, following closely those in (Honda *et al.*, 2005), Section 5.

Convention 7 *We write* $(\xi\cdot m:M,\ \sigma)\Downarrow(\xi\cdot m:V,\ \sigma')\models C$ *when* $(M\xi,\ \sigma)\Downarrow(V,\ \sigma')$ *and* $(\xi\cdot m:V,\ \sigma')\models C$ *for some* V *and* σ'.

We begin with [*Var*].

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

The proof for [*Const*] is the essentially the same as above and omitted. For [*Op*] we show the case $n=2$ for readability.

$$\begin{array}{l}(\xi,\ \sigma)\models C[x/u]\wedge\ \models\{C\}M_1:_{m_1}\{C_1\}\wedge\ \models\{C_1\}M_2:_{m_2}\{C_2[\mathsf{op}(m_1m_2)/u]\}\\ \quad\Rightarrow\quad (\xi\cdot m_1:M_1,\ \sigma)\Downarrow(\xi\cdot m_1:V_1,\ \sigma_1)\wedge\\ \qquad\quad (\xi\cdot m_1:V_1\cdot m_2:M_2,\ \sigma_1)\Downarrow(\xi\cdot m_1:V_1\cdot m_2:V_2,\ \sigma')\models C_2\ \wedge\ u=\mathsf{op}(m_1m_2)\\ \quad\Rightarrow\quad (\xi\cdot u:\mathsf{op}(M_1M_2),\ \sigma)\Downarrow(\xi\cdot u:\mathsf{op}(V_1V_2),\ \sigma')\models C_2\end{array}$$

The general n-ary case is similarly.

For [*Abs*] let $\xi'\stackrel{\text{def}}{=}\xi\cdot x:V$ below.

$$\begin{array}{l}(\xi,\sigma)\models A\\ \quad\Rightarrow\quad \forall V.(\ (\xi\cdot x:V,\ \sigma)\models A\wedge C\ \supset\ (M\xi',\ \sigma)\Downarrow(\xi'\cdot m:W,\ \sigma')\models C'\ )\\ \quad\Rightarrow\quad \forall V.((\xi\cdot x:V,\ \sigma)\models A\wedge C\ \supset\ ((\lambda x.M)\xi V,\ \sigma)\Downarrow(\xi'\cdot m:W,\ \sigma')\models C')\\ \quad\Rightarrow\quad (\xi\cdot u:(\lambda x.M)\xi,\ \sigma)\models\forall x.\{C\}u\bullet x=m\{C'\}\end{array}$$

For [*App*] we infer, with $\xi_0=\xi\cdot m:V$:

$$\begin{array}{l}(\xi,\ \sigma)\models C\\ \quad\Rightarrow\quad (M\xi,\ \sigma)\Downarrow(\xi\cdot m:V,\ \sigma_0)\models C_0\\ \quad\Rightarrow\quad (N\xi_0,\ \sigma_0)\Downarrow(\xi_0\cdot n:W,\ \sigma_1)\models C_1\wedge\{C_1\}m\bullet n=u\{C'\}\\ \quad\Rightarrow\quad (VW,\ \sigma)\Downarrow_u(\xi\cdot u:U,\ \sigma')\models C'\\ \quad\Rightarrow\quad ((MN)\xi,\ \sigma)\Downarrow_u(\xi\cdot u:U,\ \sigma')\models C'\end{array}$$

[*Pair*] and [*Proj*] are similar.

For the conditional [*If*] we set $b_1\stackrel{\text{def}}{=}\mathsf{t}$ and $b_2\stackrel{\text{def}}{=}\mathsf{f}$.

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

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

The proof for [*Case*] is equally straightforward.

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

Above we used the fact that closed values of sum types are of the form $\mathtt{in}_i(V)$ with $i\in\{1,2\}$. Next we turn to the structural rules, given in their located variant in Figure 8. Most of these rules, in the variant without effects, are proved as the corresponding rules in (Honda *et al.*, 2005). The proofs of rules which make essential use of effects, [*Invariance*], [*Weak*] and [*Thinning*], are straightforward, and hence omitted. For [*Cons-Aux*] we need a preparatory lemma.

Lemma 3 1. *If* $\mathcal{M}\models C$ *and* u *is fresh, then also* $\mathcal{M}\cdot u:V\models C$

2. *Whenever* $(M,\sigma)\longrightarrow(M',\sigma')$ *and* ρ *is an injective renaming, then also* $(M\rho,\sigma\rho)\longrightarrow(M'\rho,\sigma'\rho)$*, where we omit the straightforward definitions for applying renamings to terms and stores.*
3. *Assume that* M *is typable under* $\Gamma;\Delta$, $\Gamma\subseteq\Gamma',\Delta\subseteq\Delta'$. *Then:* $(M,\sigma)\longrightarrow(M',\sigma')$ *iff* $(M|_{\Gamma;\Delta},\sigma|_{\Gamma;\Delta})\longrightarrow(M'|_{\Gamma;\Delta},\sigma'|_{\Gamma;\Delta})$*, where we omit the straightforward definitions of the restriction operator* $\cdot|_{\Gamma;\Delta}$.
4. *If* C *is typable under* $\Gamma;\Delta$, $\Gamma\subseteq\Gamma',\Delta\subseteq\Delta'$ *and* $\mathcal{M}^{\,\Gamma';\Delta'}\models C$ *then also* $\mathcal{M}\,|_{\Gamma;\Delta}\models C$.

Proof

The proofs are direct from the definitions. □

For the derivation of [*Cons-Aux*] assume: $\Gamma;\Delta\vdash M:\beta$. Let $\mathcal{M}=(\xi,\sigma)$ and assume the $\tilde{j}$ in [*Cons-Aux*] are of type $\tilde{\alpha}$. Then

$$\begin{array}{l}1.\ \ \mathcal{M}\models C\\ \hline 2.\ \ \exists\mathcal{M}\ \leq_{\tilde{j};\tilde{\alpha}}(\xi',\sigma')\models C_0[\tilde{j}/\tilde{i}]\\ \hline 3.\ \ \exists\mathcal{M}\ \leq_{\tilde{j};\tilde{\alpha}}(\xi',\sigma'),(\xi'[\tilde{i}/\tilde{j}],\sigma'[\tilde{i}/\tilde{j}])\models C_0\end{array}$$

In Steps (2, 3) we ignore the anchor u that occurs in the models by typing, as it is not of relevance for C_0. Then:

4. $(M\xi'[\tilde{i}/\tilde{j}],\sigma'[\tilde{i}/\tilde{j}])\Downarrow(V,\sigma'''[\tilde{i}/\tilde{j}])\ \wedge\ (\xi'[\tilde{i}/\tilde{j}]\cdot u:V,\sigma'''[\tilde{i}/\tilde{j}])\models C_0'$	(IH, 3, Lem. 3.2)
5. $(M\xi',\sigma')\Downarrow(V[\tilde{j}/\tilde{i}],\sigma''')$	(Lem. 3.2)
6. $(M\xi',\sigma')\Downarrow(V,\sigma''')$	($\tilde{i}$ auxiliary)
7. $(M\xi'|_{\Gamma;\Delta},\sigma'|_\Delta)\Downarrow(V,\sigma'''|_\Delta)$	(Lem. 3.3)
8. $(M\xi,\sigma)\Downarrow(V,\sigma'''|_\Delta)$	

Continuing with parts of (4), we get

$$\begin{array}{ll}10.\ \ (\xi'[\tilde{i}/\tilde{j}]\cdot u:V,\sigma'''[\tilde{i}/\tilde{j}])\models C_0' & \\ \hline 11.\ \ (\xi'\cdot u:V[\tilde{j}/\tilde{i}],\sigma''')\models C_0'[\tilde{j}/\tilde{i}] & \text{(Lem. 3.2)}\\ \hline 12.\ \ (\xi'\cdot u:V,\sigma''')\models C_0'[\tilde{j}/\tilde{i}] & (\tilde{i}\text{ auxiliary})\end{array}$$

Using the premise once again we derive

13. $(\xi, \sigma) \models C$	
14. $(\xi \cdot u : V, \sigma) \models C$	(Lem. 3.1)
15. $(\xi' \cdot u : V, \sigma') \models [!\tilde{e}]\,(C_0'[\tilde{j}/\tilde{i}] \supset C')$	
16. $(\xi' \cdot u : V, \sigma''') \models C_0'[\tilde{j}/\tilde{i}] \supset C'$	
17. $(\xi' \cdot u : V, \sigma''') \models C'$	
18. $(\xi \cdot u : V, \sigma'''|_\Delta) \models C'$	(Lem. 3.4)

Here (16) follows from (15) because the effects of M's evaluation are in $\tilde{e}$ by construction. This validates [*Cons-Aux*]. Finally [*Rename*] holds easily as all relevant operations on models and the reduction relation is closed under injective renaming, cf. Lemma 3. Hence we have established the next theorem.

Theorem 2 *(soundness) If* $\vdash \{C\}\ M :_u \{C'\}$ *then* $\models \{C\}\ M :_u \{C'\}$.

8.2.2 Soundness of Axioms

We now show correctness of all axioms.

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

We begin with the axiomatisation of content quantification in Figure 3. We need a some preliminary facts.

Lemma 4 $\mathcal{M}\,[x \mapsto V] \leq_{x:\alpha} \mathcal{M}'$ *if and only if* $\exists \mathcal{M}''.(\mathcal{M} \leq \mathcal{M}'' \wedge \mathcal{M}''[x \mapsto V] = \mathcal{M}')$.

Proof
Straightforward from the definitions. □

Proposition 12
1. *Assume* $\mathsf{ad}(e) \subseteq \{\tilde{e}\}$*: if* $\mathcal{M} \models x \neq e_i$ *for all i, then* $[\![e]\!]_{\mathcal{M}\,[x \mapsto V]} = [\![e]\!]_{\mathcal{M}\,[x \mapsto W]}$.
2. *Assume* $\mathsf{ad}(C) \subseteq \{\tilde{e}\}$*: no occurrence of a free name in e_i is bound in C, and* $\mathcal{M} \models x \neq e_i$ *for all i. Then for all V, W,* $\mathcal{M}\,[x \mapsto V] \models C$ *iff* $\mathcal{M}\,[x \mapsto W] \models C$.
3. *If C is syntactically $!x$-free, then for all V, W,* $\mathcal{M}\,[x \mapsto V] \models C$ *iff* $\mathcal{M}\,[x \mapsto W] \models C$.

Proof
We show (1) by induction on e. The only interesting case in $e = !e'$. By the induction hypothesis (IH) $[\![e']\!]_{\mathcal{M}\,[x \mapsto V]} = [\![e']\!]_{\mathcal{M}\,[x \mapsto W]} \stackrel{\text{def}}{=} \mathbf{i}$. But $\mathcal{M} \models x \neq e_i$, hence $x \notin \mathbf{i}$, hence with $\mathcal{M} = (\xi, \sigma)$, cf. Proposition 7:

$$\sigma[x \mapsto V](\mathbf{i}) = \sigma(\mathbf{i}) = \sigma[x \mapsto W](\mathbf{i}).$$

But then

$$[\![e]\!]_{\mathcal{M}\,[x \mapsto V]} = \sigma[x \mapsto V](\mathbf{i}) = \sigma[x \mapsto W](\mathbf{i}). = [\![e]\!]_{\mathcal{M}\,[x \mapsto W]}.$$

For (2) we use induction on C. The case $e = e'$ is by (1) and $C \star C'$ as well as $\neg C$ are

immediate by the (IH). For $[!e]C$ $\langle !e\rangle C$ the result follows directly from the semantics of content quantification. For the case $\forall x^{\alpha}.C$ we assume $x \neq y$, the case $x = y$ being straightforward. Then

$$\begin{aligned}
\mathcal{M}\,[x \mapsto V] \models \forall y^{\alpha}.C &\equiv \forall \mathcal{M}'.(\mathcal{M}\,[x \mapsto V] \leq_{y:\alpha} \mathcal{M}' \supset \mathcal{M}' \models C) \\
&\equiv \forall \mathcal{M}'.((\exists \mathcal{M}''.\mathcal{M} \leq_{y:\alpha} \mathcal{M}'', \mathcal{M}''[x \mapsto V] = \mathcal{M}') \supset \mathcal{M}' \models C) \qquad (88)\\
&\equiv \forall \mathcal{M}''.(\mathcal{M} \leq_{y:\alpha} \mathcal{M}'' \supset \mathcal{M}''[x \mapsto V] \models C) \\
&\equiv \forall \mathcal{M}''.(\mathcal{M} \leq_{y:\alpha} \mathcal{M}'' \supset \mathcal{M}''[x \mapsto W] \models C) \qquad (89)\\
&\equiv \forall \mathcal{M}'.((\exists \mathcal{M}''.\mathcal{M} \leq_{y:\alpha} \mathcal{M}'', \mathcal{M}''[x \mapsto W] = \mathcal{M}') \supset \mathcal{M}' \models C) \\
&\equiv \forall \mathcal{M}'.(\mathcal{M}\,[x \mapsto W] \leq_{y:\alpha} \mathcal{M}' \supset \mathcal{M}' \models C) \qquad (90)\\
&\equiv \mathcal{M}\,[x \mapsto W] \models \forall y^{\alpha}.C
\end{aligned}$$

Here (89) is by (IH) and (88, 90) are by Lemma 4.

Finally, the case of evaluation formulae is immediate because for those, the satisfaction relation 'throws away' the store part of a model, hence annihilates the effect of update operations $[x \mapsto V]$ etc.

For (3) we proceed by induction on the generation of the assertion $C^{\neg !x}$. The case of outermost content quantification is immediate. For $C \wedge x \neq \tilde{e}$ where $\mathsf{ad}(C) \subseteq \{\tilde{e}\}$ and no name is inappropriately bound we assume

$$\mathcal{M}\,[x \mapsto V] \models C \wedge x \neq \tilde{e}.$$

Hence clearly also $\mathcal{M} \models x \neq \tilde{e}$. Thus we can apply (2) to obtain

$$\mathcal{M}\,[x \mapsto V] \models C \qquad \text{iff} \qquad \mathcal{M}\,[x \mapsto W] \models C$$

which immediately implies the required result. Closure under content quantification and propositional connectives is immediate. Finally, the case of prefixing with quantifiers is also by the (IH) and almost identical to the corresponding case in (2).

□

We now begin the proof of Theorem 3.

Lemma 5 *The axioms and the rule in Figure 3 are sound.*

Proof
For (CA1) we argue as follows

$$\begin{aligned}
\mathcal{M} \models [!x]\,(C_1^{\neg !x} \supset C_2) &\equiv \forall V.\mathcal{M}\,[x \mapsto V] \models (C_1 \supset C_2) \\
&\equiv \forall V.(\mathcal{M}\,[x \mapsto V] \models C_1 \supset \mathcal{M}\,[x \mapsto V] \models C_2) \\
&\equiv \mathcal{M} \models C_1 \supset \forall V.\mathcal{M}\,[x \mapsto V] \models C_2 \qquad \text{(Prop. 12.3)} \\
&\equiv \mathcal{M} \models C_1 \supset \mathcal{M} \models [!x]C_2 \\
&\equiv \mathcal{M} \models (C_1 \supset [!x]C_2)
\end{aligned}$$

(CA2) has the following justification.

$$\begin{aligned}
\mathcal{M} \models [!x]C &\equiv \forall V.\mathcal{M}\,[x \mapsto V] \models C \\
&\supset\equiv \mathcal{M} \models C
\end{aligned}$$

For (CA3) we derive

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

Finally, for the inference rule (CGen), we proceed by induction on the length of the proof. All the axioms are syntactically $!x$-free, and none of the proof rules of first-order logic changes this fact, hence the result is again a consequence of Proposition 12.3. This concludes the proof for the axioms and the rule in Figure 3. □

Next is are the axioms for the evaluation formula in Figure 4.

Lemma 6 *All axioms in Figure 4 are sound.*

Proof
Proofs for Axioms (e1) to (e7) are like the corresponding derivations in (Honda *et al.*, 2005). Axiom (e8) is immediately from the semantics of evaluation formulae. The extionsionality axiom (ext) is also immediate from the definition of $\cong$. □

Lemmas 6 and 5 together verify Theorem 3.

8.2.3 Soundness of Located Proof Rules and Axioms

Soundness of the located proof rules can be established in two straightforward ways: we can show them to be derivable using the original non-located rules, or, alternatively, we can reason directly. In either case, the only non-trivial case is [*Thinning*]. This is reasoned using simple instances of [*Cons-Aux*] (renaming of auxiliary names) combined with disjunction on pre/post conditions (derived from [$\vee$-*pre*] and [*Cons*]). To make the proof more transparent, we assume all effects to have the same type.

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

Soundness of other located rules is as for the corresponding unlocated rules. We conclude, with respect to the semantics given in Definition 21:

Theorem 4 *(soundness of located judgements) If* $\vdash \{C\}\ M :_u \{C'\}\ @\ \tilde{g}$ *then we have* $\models \{C\}\ M :_u \{C'\}\ @\ \tilde{g}$.

Theorem 5 *All axioms in Figures 5 are sound.*

Proof
The proofs are straightforward, either by translation into formulae without effects, or directly semantically. □

9 Reasoning Examples

One of the key criteria in evaluating a program logic's abilities is ease-of-use in verification. This section illustrates how our logic can be used for reasoning about the correctness of programs, starting with simple examples discussed in the introduction and Section 4. We conclude our exhibition of the logic's reasoning abilities by proving the correctness of higher-order, generic Quicksort.

9.1 Questionable Double (1): Direct Reasoning

In Section 5.5.3 we introduced the "Questionable Double", a program behaving differently under different distinctions. Let us reproduce the program.

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

We establish the following judgement which says that, if we assume its two arguments to be distinct, then the program does indeed double the content of the arguments references.

$$\{\mathsf{T}\}\ \texttt{double?} :_u \{\ \forall x,y.\{x\neq y\wedge !x=i\wedge !y=j\}\,u\bullet(x,y)\,\{!x=2i\wedge !y=2j\}\ \} \quad (91)$$

To infer the judgement (91), we use the following two implications.

$$x\neq y\wedge !x=i\wedge !y=j \quad\supset\quad (x\neq y\wedge !x=2i\wedge !y=j)\{\!|!x+!x/!x|\!\} \quad (92)$$

$$x\neq y\wedge !x=2i\wedge !y=j \quad\supset\quad (!x=2i\wedge !y=2j)\{\!|!y+!y/!y|\!\} \quad (93)$$

We first establish (92) and (93). For the former:

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

The reasoning for (93) is identical and hence omitted. We can now present the inference. We use [*AssignVar*] discussed already, as well as the obvious extension of [*Abs*] to cater for a vector of names, also called [*Abs*].

1.	$x\neq y\wedge !x=i\wedge !y=j \quad\supset\quad (x\neq y\wedge !x=2i\wedge !y=j)\{\!|!x+!x/!x|\!\}$	(92)
2.	$\{(x\neq y\wedge !x=2i\wedge !y=j)\{\!|!x+!x/!x|\!\}\}\ x:=!x+!x\ \{x\neq y\wedge !x=2i\wedge !y=j\}$	(AssignVar)
3.	$\{x\neq y\wedge !x=i\wedge !y=j\}\ x:=!x+!x\ \{x\neq y\wedge !x=2i\wedge !y=j\}$	(1, 2, Cons)
4.	$x\neq y\wedge !x=2i\wedge !y=j \quad\supset\quad (!x=2i\wedge !y=2j)\{\!|!y+!y/!y|\!\}$	(93)
5.	$\{(!x=2i\wedge !y=2j)\{\!|!y+!y/!y|\!\}\}\ y:=!y+!y\ \{!x=2i\wedge !y=2j\}$	(AssignVar)
6.	$\{x\neq y\wedge !x=2i\wedge !y=j\}\ y:=!y+!y\ \{!x=2i\wedge !y=2j\}$	(4, 5, Cons)
7.	$\{x\neq y\wedge !x=i\wedge !y=j\}\ x:=!x+!x\,;\,y:=!y+!y\ \{!x=2i\wedge !y=2j\}$	(3, 6, Seq)
8.	$\{\mathsf{T}\}\ \texttt{double?} :_u \{\ \forall x,y.\{x\neq y\wedge !x=i\wedge !x=j\}\,u\bullet(x,y)\,\{!x=2i\wedge !x=2j\}\ \}$	(Abs)

Save for unavoidable uses of [*Cons*], the structure of this derivation follows the syntax of the program under investigation. The derivation also suggests how to refine this program to make it alias-robust. This is done by "internalising" the condition $x\neq y$ as follows.

$$\texttt{double!} \stackrel{\text{def}}{=} \lambda(x,y).(\texttt{if}\ x\neq y\ \texttt{then}\ x:=!x+!x;y:=!y+!y\ \texttt{else}\ x:=!x+!x) \quad (94)$$

We now infer:

$$\{\mathsf{T}\}\ \mathsf{double!} :_u \ \{\ \forall x,y.\{!x = i \wedge !x = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge !x = 2j\}\ \} \qquad (95)$$

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

1 – 7. (As above).

8. $x = y \wedge\ !x = i \wedge\ !y = j \quad \supset \quad (!x = 2i \wedge\ !y = 2j)\{\!\{!x + !x / !x\}\!\}$	
9. $(!x = 2i \wedge\ !y = 2j)\{\!\{!x + !x / !x\}\!\}\ \ x := !x + !x\ \ \{!x = 2i \wedge\ !y = 2j\}$	(AssignVar)
10. $\{x = y \wedge\ !x = i \wedge\ !y = j\}\ \ x := !x + !x\ \ \{!x = 2i \wedge\ !y = 2j\}$	(1, 2, Cons)
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, 10, 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 omit detailing the calculation for Line 8.

9.2 Questionable Double (2): Located Reasoning

We have seen, in Section 5.5.3, that we can use a located assertion to obtain a more "precise" specification for the Questionable Double. In this case we wish to say that no references apart from those passed as arguments are potentially modified. Hence we derive:

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

In the following proof, we derive this assertion using a fully extensional judgement for each subpart of the program. For combining two assignments, we use [*Seq-I*] in Figure 11.

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

Line 5 adds $x \neq y$ to pre/post conditions. Using the EOI rule [*Seq-I*] may be considered a semantic strengthening of the "local reasoning", as advocated in Separation Logic (Reynolds, 2002; O'Hearn *et al.*, 2004). The conclusion discusses this phenomenon in detail.

9.3 Swap

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

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

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

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

Using this predicate we wish to establish:

$$\{\mathsf{T}\}\ \texttt{swap} :_u \{\mathsf{Swap}(u)\}\ @\ \emptyset. \tag{96}$$

Swap is the classical example, treated in much preceding work (cf. (Cartwright & Oppen, 1981; Cartwright & Oppen, 1978; Kulczycki *et al.*, 2003)). An interesting point is that the derivation does have to deal with aliasing, despite the specification's (96) not mention it. The proof has two parts: one dealing with the case when x and y are aliased, the other applying when they are not. Informally:

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

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

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

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

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

Reasoning based on Traditional Methods. Reasoning for `swap` and `double`? was elegant in the sense that properties were inferred compositionally from properties of subterms, but *without* using assumption in the verification of these subterms that are irrelevant to the subterms and are only required later, when combining the properties to get properties of the overall program. That our logic facilitates such compositional reasoning – and we will see

more of that in the verification of Quicksort below – is taken as indicative of our logic's usability. For contrast, we now present a derivation of the same specification using the traditional method a la Morris/Cartwright-Oppen (expressed in the present framework).

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

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

$$
\begin{array}{lll}
(!x=j\wedge!y=i)\{\!\{z/!y\}\!\}\{\!\{!y/!x\}\!\} & & \\
\quad\equiv\quad (!x=j)\{\!\{z/!y\}\!\}\{\!\{!y/!x\}\!\}\wedge(!y=i)\{\!\{z/!y\}\!\}\{\!\{!y/!x\}\!\} & & \text{(Pro. 10 (2))}\\
\quad\equiv\quad ((x=y\supset z=j)\ \wedge\ (x\neq y\supset!x=j))\{\!\{!y/!x\}\!\}\ \wedge\ (z=i)\{\!\{!y/!x\}\!\} & & (S1)\\
\quad\equiv\quad (x=y\supset z=j)\{\!\{!y/!x\}\!\}\ \wedge\ (x\neq y\supset!x=j)\{\!\{!y/!x\}\!\}\ \wedge\ (z=i)\{\!\{!y/!x\}\!\} & & \text{(Pro. 10 (2))}\\
\quad\equiv\quad (x=y\supset z=j)\ \wedge\ (x\neq y\supset!x=j\{\!\{!y/!x\}\!\})\ \wedge\ z=i & & \text{(Pro. 7)}\\
\quad\equiv\quad (x=y\supset z=j)\ \wedge\ (x\neq y\supset!y=j)\ \wedge\ z=i & & \text{(S1)}\\
\quad\subset\quad !x=i\wedge!y=j\ \wedge\ z=i & &
\end{array}
$$

This derivation uses Property (S1):

$$e'=!e\{\!\{e''/!e_2\}\!\}\qquad\equiv\qquad((e=e_2\wedge e'=e'')\vee(e\neq e_2\wedge e'=!e))$$

or, as its special instance $e'=!e\{\!\{e''/!e\}\!\}\equiv e'=e''$, in both cases assuming e and e' do not contain dereferences. The proof is immediate from the axioms.

While the traditional reasoning gives a slightly shorter derivation, it involves non-trivial inferences at the assertion level. This is because the traditional method (or separation-based methods a la Burstall) cannot exploit semantic independence between two assignments, unlike ours, via [*Seq-I*].

9.4 Circular References

We next show the reasoning for $x:=!!x$, the example, appearing in Section 5, that uses circular data structures. Reproducing the assertion in Section 5, we wish to prove the following judgement.

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

```
1  μq. λ(a,c,l,r).
2        if l < r then
3              let p' = partition(a, c, l, r) in
4                    q(a, c, l, p'-1);
5                    q(a, c, p'+1, r)
```

Fig. 13. Quicksort with a comparison procedure as a parameter.

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

$$\begin{array}{lll} !x = y \wedge !y = x & \Rightarrow & !!x = x \\ & \Rightarrow & \exists m.(m = !!x \wedge \langle !!x \rangle\,(!x = x \wedge !x = m)) \\ & \Rightarrow & !x = x\{\!|!!x/m|\!\} \end{array}$$

From here it is easy to get:

1. $(!x = y \wedge !y = x) \supset ((!x = x)\{\!|!!x/!x|\!\})$	
2. $\{(!x = x)\{\!|!!x/!x|\!\}\}\ x := !!x\ \{!x = x\}$	(AssignSimple)
3. $\{!x = y \wedge !y = x\}\ x := !!x\ \{!x = x\}$	(1, 2, Cons)

The next assertion, also already discussed in Section 5, can similarly easily be derived.

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

9.5 A Polymorphic, Higher-Order Procedure: Quicksort

Hoare's Quicksort is an efficient algorithm for sorting arrays. Apart from recursive calls to itself, Quicksort calls Partition, a procedure which permutes elements of an array so that they are divided into two contiguous parts, the left containing elements less than a "pivot value" *pv* and the right those greater than *pv*. The pivot value *pv* is one of the array elements which may ideally be their mean value. In the following we specify and derive a full specification of one instance of the algorithm, directly taken from its well-known C version (Kernighan & Ritchie, 1988). Using indentation for scoping, Figures 13 and 14 present the code, assuming a generic swapping procedure like that from Section 5.6.4 being globally available (we could have passed the swapping routine as a parameter, like we do with the comparison function c, without significant effect on specification or proof complexity, but we wanted to show how our logic can deal with either). We use indentation for scoping. In these programs we omit type annotations for variables, the main ones of which (for both programs) are:

$$a : \mathrm{X}[\,] \qquad c : (\mathrm{X} \times \mathrm{X}) \Rightarrow \mathsf{Bool} \qquad l, r : \mathsf{Nat} \qquad \mathtt{swap} : (\mathsf{Ref}(\mathrm{X}) \times \mathsf{Ref}(\mathrm{X})) \Rightarrow \mathsf{Unit}$$

X[ ] is the type of a generic array (details of polymorphic arrays omitted). Quicksort itself has the function type from the product of these types to Unit. Partition is the same except that it return type is Nat.

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

Fig. 14. Partitioning algorithm.

This program exhibits several features which are interesting from the viewpoint of capturing and verifying behavioural properties using the present logic.

- Correctness crucially relies on the extensional behaviour of each part: when recursively calling itself twice in Lines 4 and 5 of Figure 13, it is essential that each call modifies only the local subarray it is working with, without any overlap. We shall show how this aspect is transparently reflected in the structures of assertions and reasoning, realising what O'Hearn and Reynolds called "local reasoning" (O'Hearn *et al.*, 2004; Reynolds, 2002) through the use of logical primitives of general nature rather than those introduced for that specific purpose.
- The program is higher-order, receiving as its argument a comparison procedure.
- The program is fully polymorphic, in the sense that it can sort an array of any type (as far as a proper comparison procedure is provided).

In the following we shall discuss how these aspects can be treated in the present logic. Even including a recent formal verification of Quicksort in Coq (Filliâtre & Magaud, 1999), we believe a rigorous verification of Quicksort's extensional behaviour with higher-order procedures and polymorphism is given here for the first time.

Specification. We now present a full specification of Quicksort (For simplicity, `partition` and `swap` are assumed inlined: treating them as external procedures is straightforward).

$$\{\mathsf{T}\}\ \texttt{qsort} :_u \{\forall \mathrm{X}.\mathsf{Qsort}(u)\}@\emptyset. \tag{97}$$

where we set, omitting types:

$$\mathsf{Qsort}(u) \quad \stackrel{\text{def}}{=} \quad \forall abclr.\begin{pmatrix}\{\mathsf{Eq}(ablr)\wedge\mathsf{Order}(c)\}\\ u\bullet(a,c,l,r)\\ \{\mathsf{Perm}(ablr)\wedge\mathsf{Sorted}(aclr)\}@a[l...r]ip\end{pmatrix} \tag{98}$$

Here $a[l...r]ip$ is short for $a[l],...,a[r],i,p$. The variable b is auxiliary and is of the same array type as a, denoting an initial copy of a, so we can specify the change of a in the post-condition is only in the ordering of its elements. Each predicate used in (98) has the following meaning. For the precondition:

- First, the predicates $\mathsf{Eq}(ablr)$ and $\mathsf{Perm}(ablr)$ use a distinctness condition on elements of a as well as b, p and i, which we write Dist. Formally: define

$$\mathsf{Distinct}(e_1..e_n) \quad \stackrel{\text{def}}{=} \quad \wedge_{1\le i\neq j\le n} e_i \neq e_j,$$

  then we set

$$\mathsf{Dist} \quad \stackrel{\text{def}}{=} \quad \mathsf{Distinct}(\ a[0]...a[\mathsf{size}(a)-1]b[0]...b[\mathsf{size}(b)-1]pi\ ).$$

- $\mathsf{Eq}(ablr)$ says: *distinct arrays a and b coincide in their content in the range from l to r (with l and r being in the array bound).* In addition, it also stipulates freshness and distinctness of variables p and i. The formal definition of $\mathsf{Eq}(ablr)$ is:

$$0 \le l, r \le \mathsf{size}(a) = \mathsf{size}(b) \ \wedge\ \forall j.(l \le j \le r \ \supset\ !a[j] = !b[j]) \ \wedge\ \mathsf{Dist}.$$

- $\mathsf{Order}(c)$ says: *c calculates a total order without side effects.* Formally, it is the conjunction of:
  - — $\forall xy.\,(c \bullet (x,y) \searrow \mathsf{T} \ \vee\ c \bullet (x,y) \searrow \mathsf{F})$. In this assertion "$c \bullet (x,y) \searrow e$" stands for "$\{\mathsf{T}\}c \bullet (x,y) = z\{z = e\}\,@\,\emptyset$" ("the comparison terminates and has no side effects");
  - — $\forall xy.(x \neq y \ \supset\ (c \bullet (x,y) \searrow \mathsf{T} \vee c \bullet (y,x) \searrow \mathsf{T}))$ ("two distinct elements are always ordered"); and
  - — $(c \bullet (x,y) \searrow \mathsf{T} \wedge c \bullet (y,z) \searrow \mathsf{T}) \supset c \bullet (x,z) \searrow \mathsf{T}$ ("the ordering is transitive").

  The use of this predicate instead of (say) a boolean condition embodies the higher-order nature of Quicksort.

For the post-condition:

- $\mathsf{Perm}(ablr)$ says: *entries of a and b in the range from l to r are permutations of each other in content.* It also stipulates the same distinctness condition as $\mathsf{Eq}(ablr)$. Formally:

$$\begin{aligned}\mathsf{SPerm}(ablr) \quad \stackrel{\text{def}}{=} \quad & \exists i, j.(l \le i, j \le r \wedge\ !a[i] = !b[j] \wedge\ !a[j] = !b[i] \ \wedge \\ & \forall h.(\ (l \le h \le r \ \wedge\ h \notin \{i, j\})\ \supset\ !a[h] = !b[h])\ ) \ \wedge \\ & \mathsf{size}(a) = \mathsf{size}(b) \ \wedge\ \mathsf{Dist}\end{aligned}$$

  The result of permuting n times is then given by:

$$\begin{aligned}\mathsf{Perm}^{(0)}(ablr) \quad & \stackrel{\text{def}}{=} \quad \mathsf{Eq}(ablr) \\ \mathsf{Perm}^{(n+1)}(ablr) \quad & \stackrel{\text{def}}{=} \quad \exists a'.(\mathsf{Perm}^{(n)}(aa'lr) \ \wedge\ \mathsf{SPerm}(a'blr) \ \wedge\ \mathsf{Dist}[a'/b])\end{aligned}$$

  Note that, as in $\mathsf{Eq}(ablr)$, the permutation predicate asserts the full distinction of all relevant references.
- $\mathsf{Sorted}(alrc)$ says: *the content of a in the range from l to r are sorted w.r.t. the total order implemented by c.* Formally: $\mathsf{Sorted}(aclr) \stackrel{\text{def}}{=} \forall i, j.(l \le i < j \le r \ \supset\ c \bullet (!a[i], !a[j]) \searrow \mathsf{T})$.

So $\mathsf{Qsort}(u)$ in (98) as a whole says:

Initially we assume two distinct arrays, a and b, of the same content from l to r ($\mathsf{Eq}(ablr)$*), together with a procedure which realises a total order (*$\mathsf{Order}(c)$*). After the program runs, one array remains unchanged (because the assertion says it touches only a), and this changed array is such that it is the permutation of the original one (*$\mathsf{Perm}(ablr)$*) and that it is well-sorted w.r.t. c (*$\mathsf{Sorted}(aclr)$*).*

Located assertions play a fundamental role in this specification: for example, it is crucial to be able to assert c has no unwanted side effects. In the rest of this section, we present highlights and key steps of the full derivation of the judgement (97). Straightforward steps are mostly omitted, as they can be filled in easily, since reasoning follows the syntactic structure of the algorithm precisely.

Reasoning (1): Sorting Disjoint Subarrays. First we focus on Lines 4 and 5 in Figure 13), which sort subarrays by recursive calls. The reasoning demonstrates how the use of our refined invariance rule offers a quick inference by combining two local, extensional specifications. Concretely, our aim is to establish:

$$\{C_1\}\ \ \mathtt{q}(a,c,l,p'-1)\ ;\ \mathtt{q}(a,c,p'+1,r)\ \ \{C_1'\}@a[l..r]ip \tag{99}$$

where

$$C_1 \stackrel{\text{def}}{=} \mathsf{Perm}(ablr)\ \wedge\ \mathsf{Parted}(aclrp')\ \wedge\ \mathsf{Order}(c)\ \wedge\ \forall j{<}k.\mathsf{QsortBounded}(qj)\ \wedge\ r-l{\leq}k$$

$$C_1' \stackrel{\text{def}}{=} \mathsf{Perm}(ablr)\wedge\mathsf{Sorted}(aclr).$$

Two newly introduced predicates are illustrated below.

$\mathsf{QsortBounded}(qj)$ with j of Nat type is used as an inductive hypothesis for recursion. It is the same as $\mathsf{Qsort}(q)$, given in (98), Page 68, except that it only works for a range no more than j and that it replaces "$\mathsf{Eq}(ablr)$" in the precondition of (98) with "$\mathsf{Perm}(ablr)$", which is necessary for the induction to go through. $\mathsf{Parted}(aclrk)$ says the subarray of a from l to r is partitioned at an intermediate index k w.r.t. the order defined by c. Formally it is given as:

$$\mathsf{Parted}(aclrk) \stackrel{\text{def}}{=} \left(\begin{array}{c} l\leq k\leq r\ \ \wedge\ \ \forall j.(l\leq j\leq k\supset c\bullet(!a[j],!a[k])\searrow\mathsf{T}) \\ \wedge \\ \forall j.(k\leq j\leq r\supset c\bullet(!a[k],!a[j])\searrow\mathsf{T}) \end{array}\right)$$

A key feature of these two recursive calls is that neither modifies/depends on subarrays written by the other. As mentioned already, this feature allows us to *localise* reasoning: the specification and deduction of each part has only to mention local information it is concerned with. Joining the resulting two specifications is then transparent through the invariance rule and basic laws of content quantification. Let $\tilde{e_2} \stackrel{\text{def}}{=} a[l..p'-1]pi$ and $\tilde{e_3} \stackrel{\text{def}}{=} a[p'+1..r]pi$ (which are the parts touched by the first/second calls, respectively). We now derive:

R.1.	$\{C_2\}\ \mathtt{q}(\mathtt{l},\mathtt{p}'-1)\ \{C_2'\}\ @\ \tilde{e_2}$
R.2.	$\{C_3\}\ \mathtt{q}(\mathtt{p}'+1,\mathtt{r})\ \{C_3'\}\ @\ \tilde{e_3}$
R.3.	$\{C_2\ \wedge\ [!\tilde{e_2}]C_3\}\ \mathtt{q}(\mathtt{l},\mathtt{p}'-1)\ ;\ \mathtt{q}(\mathtt{p}'+1,\mathtt{r})\ \{\langle!\tilde{e_3}\rangle C_2'\ \wedge\ C_3'\}@\tilde{e}_2\tilde{e}_3$
R.4.	$C_1\ \supset\ \exists b'.(([!\tilde{e_3}]C_2\ \wedge\ C_2\ \wedge\ [!\tilde{e}_2\tilde{e}_3]\,(C_2'\wedge\langle!\tilde{e_2}\rangle C_3'\ \supset\ C_1')))$
R.5.	$\{C_1\}\ \mathtt{q}(\mathtt{l},\mathtt{p}'-1)\ ;\ \mathtt{q}(\mathtt{p}'+1,\mathtt{r})\ \ \{C_1'\}@\tilde{e}_2\tilde{e}_3$ (Cons-Aux)

Line (R.3) uses (R.1-2, Seq-I), the first two (AppS). The derivation uses the following abbreviations.

$$\begin{array}{rcl} C_2 & \stackrel{\text{def}}{=} & \mathsf{Eq}(ab'l(p'-1)) \wedge \mathsf{Order}(c) \wedge \forall j<k.\mathsf{QsortBounded}(qj) \\ & & \wedge\ p'-1-l<k \\ C_2' & \stackrel{\text{def}}{=} & \mathsf{Perm}(ab'l(p'-1)) \wedge \mathsf{Sorted}(acl(p'-1)) \\ C_3 & \stackrel{\text{def}}{=} & \mathsf{Eq}(ab'(p'+1)r) \wedge \mathsf{Order}(c) \wedge \forall j<k.\mathsf{QsortBounded}(qj) \wedge \\ & & r-(p'+1)<k \\ C_3' & \stackrel{\text{def}}{=} & \mathsf{Perm}(ab'(p'+1)r) \wedge \mathsf{Sorted}(ac(p'+1)r) \end{array}$$

Note each of C_2/C_2' and C_3/C_3' mentions only the local subarray each call works with. The auxiliary variable b' serves as a fresh copy of a immediately before these calls (we cannot use b since, e.g. $\mathsf{Perm}(abl(p'-1))$ does not hold). (R.1–3) are asserted and reasoned using b', which (R.4) mediates into the judgement on b, so that (R.5) only mentions b. The inference uses [*Cons-Aux*], our rendition of Kleyman's Rule from Figure 8. In addition, we need another straightforwardly derived rule:

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

Using these rules and [*Seq-I*], (R.1/2/3/5) are immediate. The remaining step is the derivation of (R.4), the condition for [*Cons-Aux*].

First-order logic allows the following entailment

$$C_1 \quad \Leftrightarrow \quad C_1 \wedge \exists b'.(\mathsf{Eq}(ab'lr) \wedge \mathsf{Dist}') \quad \Rightarrow \quad \exists b'.D$$

where the definition of D is next.

$$D \stackrel{\text{def}}{=} \left(\begin{array}{c} \mathsf{Eq}(ab'lr) \wedge \mathsf{Parted}(b'clrp') \wedge \mathsf{Perm}(ab'lr) \wedge \mathsf{Perm}(ablr) \\ \wedge \\ \mathsf{Order}(c) \wedge l \leq p' \leq r \wedge \mathsf{Dist}' \wedge \forall j < k.\mathsf{QsortBounded}(qj) \end{array} \right)$$

Now clearly

$$D \quad \Rightarrow \quad C_2 \wedge C_3 \quad \Rightarrow \quad C_2 \wedge [!\tilde{e}_2]C_3,$$

The former implication being first-order logic which the latter using (ua), since $C_3^{\text{-}!\tilde{e}_2}$. It is also the case that

$$D \quad \Rightarrow \quad \mathsf{Parted}(b'clrp') \wedge !a[p'] = !b[p'] \wedge \mathsf{Dist}'$$

1. $C_2' \wedge C_3'$	
2. $\mathsf{Perm}(ab'l(p'-1)) \wedge \mathsf{Perm}(ab'(p'+1)r)$	(1)
3. $!a[p'] = !b'[p']$	
4. $\mathsf{Perm}(ab'lr)$	(2, 3)
5. $\mathsf{Perm}(bb'lr)$	
6. $\mathsf{Perm}(ablr)$	(4, 5)
7. $\mathsf{Sorted}(acl(p'-1)) \wedge \mathsf{Sorted}(ac(p'+1)r)$	(1)
8. $\mathsf{Parted}(bclrp')$	
9. $\mathsf{Sorted}(aclr)$	(4, 7, 8)

Hence in fact

$$(!a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset ((C_2' \wedge C_3') \supset C_1')$$

which in turn implies

$$(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset ((C_2' \wedge C_3') \supset C_1').$$

To this tautology we add universal content quantification with respect to $\tilde{e} \stackrel{\text{def}}{=} \tilde{e}_2\tilde{e}_3$ to obtain

$$[!\tilde{e}]\,(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset ((C_2' \wedge C_3') \supset C_1').$$

But in view of Dist', all terms in the premise of that last term, are $!\tilde{e}$-free, hence we apply Proposition 7.

$$(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset [!\tilde{e}]\,((C_2' \wedge C_3') \supset C_1').$$

Now, with Dist', C_2' is $!\tilde{e}_3$-free, so C_2' and $\langle !\tilde{e}_3\rangle C_2'$ are in fact equivalent, using (e4, ea). That means we can refine that last big implication.

$$(\mathsf{Dist}' \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset [!\tilde{e}]\,((\langle !\tilde{e}_3\rangle C_2' \wedge C_3') \supset C_1').$$

Combining all this, yields the assertion

$$C_1 \quad \supset \quad C_2 \wedge [!\tilde{e}_2]\,C_3 \wedge [!\tilde{e}]\,((\langle !\tilde{e}_3\rangle C_2' \wedge C_3') \supset C_1')$$

which is (R.4) used above.

Reasoning (2): Using Comparison. Next we focus on the use of a comparison procedure in the while loop in `partition`, which is originally passed to Partition as an argument. We start with the loop invariant.

$$\mathsf{Invar} \stackrel{\text{def}}{=} \left(\begin{array}{c} C_{part}^{pre} \wedge l \leq !p, !i \leq r \wedge \mathsf{Leq}(acl(!p-1)pv) \\ \wedge \\ \mathsf{Geq}(ac(!p_0)(!i-1)pv) \wedge (!p < !i \supset c \bullet (!a[!p], pv) \searrow \mathsf{T}) \end{array} \right)$$

$\mathsf{Leq}(aclrv)$ (resp. $\mathsf{Geq}(aclrv)$) says the entries from l to r in a are smaller (resp. bigger) than v. When inside the loop, the values of p and i differ from the invariant slightly, so that we also make use of: $C_{inloop} \stackrel{\text{def}}{=} \mathsf{Invar} \wedge !i < r \wedge r - !i = j$. The following assertions specify

two cases of the conditional branch.

$$C_{then} \stackrel{\text{def}}{=} C_{inloop} \wedge c \bullet (!a[!i], pv) \searrow \mathsf{T} \quad C_{\neg then} \stackrel{\text{def}}{=} C_{inloop} \wedge c \bullet (!a[!i], pv) \searrow \mathsf{F}.$$

We now present the derivation for the if sentence of the loop, where the comparison procedure received as an an argument is used at the conditional branch. Below we assume the conditional body ("`ifbody`") has been verified already and let j to be a freshly chosen variable of Nat-type.

$$\dfrac{(\mathsf{Invar} \wedge r - !i > 0) \supset \begin{pmatrix} \{\mathsf{Invar} \wedge r - !i > 0\} \\ \quad c \bullet (!a[!i], pv) = z \\ \{c \bullet (!a[!i], pv) \searrow z \wedge \mathsf{Invar} \wedge r - !i > 0\}@\emptyset \end{pmatrix}}{\begin{array}{l} \{\mathsf{Invar} \wedge r - !i > 0\} \\ \quad c(!a[!i], pv) :_z \\ \{c \bullet (!a[!i], pv) \searrow z \wedge \mathsf{Invar} \wedge r - !i > 0\}@\emptyset \end{array}} (\text{AppSimple})$$

$$\dfrac{\{C_{then}\}\ \texttt{ifbody}\ \{\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r - !i \leq j)\}\,@\,a[l...r-1]ip}{C_{\neg then} \supset (\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r - !i \leq j)} (\text{omitted})$$

$$\dfrac{\{C_{inloop}\}\texttt{if}\ c(!a[!i], pv)\ \texttt{then ifbody}}{\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r - !i \leq j)\{a[l...r-1]pi\}@} (\text{IfThen})$$

Thus reasoning about a conditional branch which involves a call to a received procedure is no more difficult than treating first-order expressions. The rest of the verification for `partition` is mechanical, so that we reach the following natural judgement:

$$\begin{array}{l} \{\mathsf{Perm}(ablr) \wedge \mathsf{Order}(c)\} \\ \quad \texttt{partition}(a, c, l, r)\ :_{p'} \\ \{\mathsf{Parted}(aclrp') \wedge \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c)\}@a[l..r]pi \end{array} .$$

Reasoning (3): Polymorphism. We are now ready to derive the whole specification of Quicksort (97). As noted, the algorithm is generic in the type of data being sorted, so we conclude with deriving its polymorphic specification. We need one additional rule for type abstraction (for further details of treatment of polymorphism, see (Honda & Yoshida, 2004)). We also list the rule for "let" which is easily derivable from [*Abs*] and [*App*] through the standard encoding. Below, $\mathsf{ftv}(\Theta)$ indicates the type variables in Θ, similarly for $\mathsf{ftv}(C)$.

$$[TAbs]\ \dfrac{\{C\}\ V^{\Gamma;\Delta;\alpha} :_m \{C'\} \quad \mathrm{X} \notin \mathsf{ftv}(\Gamma, \Delta) \cup \mathsf{ftv}(C)}{\{C\}\ V^{\Gamma,\Delta;\forall \mathrm{X}.\alpha} :_u \{\forall \mathrm{X}.C'\}}$$

$$[Let]\ \dfrac{\{C\}\ M :_x \{C_0\}\,@\,\tilde{e} \quad \{C_0\}\ N :_u \{C'\}\,@\,\tilde{e'}}{\{C\}\ \texttt{let}\ x = M\ \texttt{in}\ N :_u \{C'\}\,@\,\tilde{e}\tilde{e'}}$$

We now present the derivation. For brevity we use the following abbreviations: $C_\star \stackrel{\text{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)$, $B' \stackrel{\text{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c) \wedge \forall j < k.\mathsf{QsortBounded}(qj) \wedge r - l \leq k$, and $B \stackrel{\text{def}}{=} B' \wedge l < r$. We also write $\texttt{qsort}'$ for `qsort` in Page 67 without the first line (i.e. without μ/λ-abstractions), M for $\mathtt{q}(a, c, l, p' - 1)$; $\mathtt{q}(a, c, p' + 1, r)$ and N for

$\mathtt{q}(l,p'-1)\ ;\ \mathtt{q}(p'+1,r)$.

$\{B\}\ \mathtt{partition}(a,c,l,r) :_{p'} \{\mathsf{Parted}(aclrp') \wedge B\}@a[l..r]pi$	(Invariance)
$\{\mathsf{Parted}(aclrp') \wedge B\}\ M\ \{C_\star\}@a[l...r]ip$	(R.5)
$\{B\}\ \mathtt{let}\ p' = \mathtt{partition}(a,l,r,c)\ \mathtt{in}\ N\ \{C_\star\}@a[l...r]ip$	(Let)
$\{B'\}\ \mathtt{qsort}'\ \{C_\star\}@a[l...r]ip$	(IfThen)
$\{\forall j < k.\mathsf{QsortBounded}(qj)\}\ \lambda(a,c,l,r).\mathtt{qsort}'\ :_m \{\mathsf{QsortBounded}(mk)\}@\emptyset$	(Abs)
$\{\mathsf{T}\}\ \mathtt{qsort}\ :_u \{\mathsf{Qsort}(u)\}@\emptyset$	(Rec, Cons)
$\{\mathsf{T}\}\ \mathtt{qsort} :_u \{\forall \mathrm{X}.\mathsf{Qsort}(u)\}@\emptyset$	(TAbs)

This concludes the derivation of a full specification for polymorphic Quicksort.

10 Conclusion

This paper introduced a program logic for imperative higher-order functions with general forms of aliasing, presented its basic theory, and explored its use for specification and verification through simple but non-trivial examples. Distinguishing features of the proposed program logic include: a general treatment of imperative higher-order functions and aliasing; its precise correspondence with observational semantics (Greif & Meyer, 1981; Hennessy & Milner, 1985); provision of structured assertion and reasoning methods for higher-order behaviour with shared data in the presence of aliasing; and clean extensibility to data structures. We expect that compositional program logics, capturing fully the behaviour of higher-order programs, will have applications not only in specification and verification of individual programs but also in combination with other engineering activities for safety guarantees of programs.

The logic is built on our earlier work (Honda *et al.*, 2005), where we introduced a logic for imperative higher-order functions without aliasing. In (Honda *et al.*, 2005), a reference type in both the programming and assertion languages is never carried by another type, which leads to the lack of aliasing: operationally, in that work, a procedure never received or returned (and a reference never stored) references, while logically, equating two distinct reference names was contradictory. In the present work we have taken off this restriction. This leads to substantially richer and more complex program behaviour, which is met by a minimal but powerful enrichment in the logic, both in semantics (through introduction of distinctions) and in syntax (by content quantification). The added machinery allows us to reason about a general form of assignment, $M := N$, to treat a large class of mutable data structures and to reason about many programs of practical significance such as Quicksort, all of which have not been possible in (Honda *et al.*, 2005). We conclude the paper with discussions on remaining topics and related work.

10.1 Observational Completeness

A central property of our logic is its precise correspondence with the observational congruence of the programming language, in the sense that two programs are contextually equivalent iff they satisfy the same set of assertions. We term this coincidence between a programming language and its logic *observational completeness*. It offers foundations

for modular software engineering, where replacement of one module by another with the same specification does not violate the observable behaviour of the whole software, up to the latter's global specification.

Theorem 6 (observational completeness) *Assuming appropriate typing, the following two statements are equivalent.*

- $M \cong N$
- *For all* C, C': $\models \{C\}M :_u \{C'\}$ *iff* $\models \{C\}N :_u \{C'\}$.

The proof of observable completeness, omitted for brevity, extends the method used in (Honda *et al.*, 2005). A detailed proof will be presented in a forthcoming paper on completeness phenomena.

10.2 Local References

Apart from aliasing and higher-order behaviours, one of the focal points in reasoning about (imperative) higher-order functions is new name generation or local references, as studied by Pitts and Stark (Pitts & Stark, 1998). Its clean logical treatment is possible through a rigorous stratification on top of the present logic. At the level of programming language, the grammar is extended by $\texttt{new}\ x := M\ \texttt{in}\ N$ with $x \notin \mathsf{fv}(M)$. For its logical treatment, there are two layers. In one, local references are never allowed to go out of the original scope (hence they are freshly created and used at each run of a program or a procedure body, to be thrown away after termination or return). In this case, we do not have to change the assertion language but only add what corresponds to the standard proof rule for locally declared variables. Below we present a simpler case when name comparison is not allowed in the target programming language.

$$\frac{\{C^{\frown x}\}\, N :_n \{C_0\} \quad \{([!x]C_0)[!x/n]\}\, M^{\Gamma;\Delta\cdot x:\mathsf{Ref}(\alpha);\beta} :_m \{C'^{\frown x}\}}{\{C\}\ \texttt{new}\ x := N\ \texttt{in}\ M^{\Gamma;\Delta;\beta} :_u \{C'\}} \tag{100}$$

This rule says that, when inferring for M, we can safely assume that the newly generated x is distinct from existing reference names, and that the description of the resulting state and value, C', should not mention this new reference. It is notable that this rule and its refinement for the restricted form of local references allow us to treat the standard parameter passing mechanism in procedural languages such as C and Java through the following simple translation: a procedure definition "$\texttt{f(x,y)}\ \{...\}$" is transformed into

$$\lambda(x',y').\texttt{new}\ x := x'\ \texttt{in}\ \texttt{new}\ y := y'\ \texttt{in}\ ....$$

Since x and y are freshly generated, they are never aliased with each other nor with existing reference names. This aspect is logically captured by (100). Thus the (lack of) aliasing in stack variables can be analysed as a special case of aliasing in general references, allowing uniform understanding.

In the fully general form of local references, a newly generated reference can be exported to the outside of its original scope, reminiscent of scope extrusion in the π-calculus (Milner *et al.*, 1992), and may outlive the generating procedure, e.g. $\lambda n.\texttt{new}\ x := n\ \texttt{in}\ x$. A procedure can now have local state, possibly changing behaviour at each run, reflecting

not only a given argument and global state but also its local state, the latter invisible to the environment. This leads to greater complexity in behaviour, demanding a further enrichment in logics. How this can be handled with a clean and minimal extension to the present logic will be discussed in the forthcoming (Yoshida *et al.*, 2006).

10.3 Related Work

A detailed historical survey of the last three decades' work on program logics and reasoning methods which treat aliasing is beyond the scope of the present paper. Instead we focus on some pioneering and directly related Hoare-like program logics for aliasing. Janssen and van Emde Boas (Janssen & van Emde Boas, 1977) first introduce distinctions between reference names and their content in the assertion method. The assignment rule based on semantic substitution is discussed by Cartwright and Oppen (Cartwright & Oppen, 1981), Morris (Morris, 1982b) and Trakhtenbrot, Halpern and Meyer (Trakhtenbrot *et al.*, 1984). The work by Cartwright and Oppen (Cartwright & Oppen, 1981) presented a (relative) completeness result for a language with aliasing and procedures. Morris (Morris, 1982b) gives extensive reasoning examples. The work by Cartwright, Oppen and Morris is discussed in more detail below. Bornat (Bornat, 2000) further explored Morris's reasoning method. Trakhtenbrot et al. (Trakhtenbrot *et al.*, 1984) also propose an invariance rule reminiscent of ours, as well as using the dereference notation in the assertion language for the first time. As arrays and other mutable data structures introduce aliasing between elements, studies of their proof rules such as (Gries & Levin, 1980; Luckham & Suzuki, 1979; Apt, 1981) contain logical analyses of aliasing (which goes back to (McCarthy, 1962)). More recently Kulczycki et al. (Kulczycki *et al.*, 2003) study possible ways to reason about aliasing induced by call-by-reference procedure calls.

Cartwright and Oppen. Cartwright and Oppen (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981) show how to use distinctions on reference names and semantic update as part of Hoare Logic's standard assertion language. They present a formal result which decomposes semantic update into reference name (in)equations. They treat a programming language with multiple assignment, (recursive) first-order procedures and pointers. Their assertion language uses a specific predicate which says reference names *per se* are distinct, rather than having an explicit dereference operator. The underlying model is inspired by McCarthy's articulation of imperative computation (McCarthy, 1962) and (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981) present two related logics.

- First, a logic where the above "distinct" predicate and semantic update are present, but the programming language has no pointers (hence no aliasing except that coming from arrays). After observing this semantic update to coincide with syntactic update in the absence of aliasing, they establish soundness and relative completeness of their proof rules.
- The second logic extends the first with pointers, at the level of both programs and assertions. For assignment $!x := e$ (in our notation), it is observed that the assignment rule $\{C\{\!|e/!!x|\!\}\}!x := e\{C\}$ (again our notation) suffices, but semantic update is no longer replaceable by a syntactic counterpart. Then a compositional translation of the

semantic update is presented which uses the "distinct" predicate. They also propose a rule for procedures which allow pointer passing and discuss its soundness and completeness.

Despite complexity in presentation, their work is a milestone in the treatment of aliasing in Hoare's logic, by (1) distinguishing reference names and content, (2) introducing semantic update in the assertion language, and (3) showing how semantic update can be eliminated through decomposition into (in)equations of reference names. Note that (3) is fundamental for keeping compositional proof rules syntactic in principle.

In the introduction, we already discussed a basic issue of the logic(s) in (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981): while semantic update becomes "syntactic" by decomposition, in practice it is hard to carry out real logical calculation. This problem is acknowledged in (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981). Another problem was the lack of structured reasoning principles about extensional behaviour of aliased programs (Cartwright & Oppen, 1978; Cartwright & Oppen, 1981). Treatment of a higher-order procedures and various data structures (which was beyond the state of the art at the time) is also left as a future issue. The present work addresses these issues by clarifying the logical status of semantic update through modal operators and integrating them with a standard assertion language. At the level of models, our use of distinctions in models arguably also contributes to the present logic's simplicity.

Morris. Independently, Morris, in a sequence of works (Morris, 1982a; Morris, 1982d; Morris, 1982c), presented essentially the same ideas as Cartwright and Oppen, but in a syntactically more tractable and uniform framework with treatment of general data structures including pointers. His approach is an elegant extension of Hoare logic based on conditional update. Morris also distinguishes a reference name and its content, using $x\downarrow$ to denote the address of x (which is symmetric to the pointer notation $x\uparrow$ in Pascal). His technical treatment centres on the conditional expression rather than semantic update. He starts from a notion of conditional substitution given as follows, assuming x and y are reference names of the same type in a given program.

$$y\{\!|e/x|\!\} \quad \stackrel{\text{def}}{=} \quad \texttt{if}\ x\downarrow = y\downarrow\ \texttt{then}\ e\ \texttt{else}\ y$$

Here a term of type $\mathsf{Ref}(\alpha)$ denotes its content in the assertion language, hence (in)equality of names proceeds by taking their addresses. Morris showed, through examples, that his conditional update is extensible to complex expressions, but a precise axiomatic treatment is first given by Bornat in (Bornat, 2000). We reproduce one of his calculations below (following the original presentation in using Pascal-like field-selection notation and omitting obvious $\downarrow$'s):

$$\begin{aligned}
&(p.s.s)\{\!|v/u.s|\!\}\\
&\quad\equiv\quad ((p.s)\{\!|v/u.s|\!\}.s)\{\!|v/u.s|\!\}\\
&\quad\equiv\quad (\texttt{if}\ u = p\ \texttt{then}\ v\ \texttt{else}\ p.s.s)\{\!|v/u.s|\!\}\\
&\quad\equiv\quad \texttt{if}\ u = p\ \texttt{then}\ (\texttt{if}\ v = u\ \texttt{then}\ v\ \texttt{else}\ v.s)\ \texttt{else}\ (\texttt{if}\ p = u\ \texttt{then}\ v\ \texttt{else}\ p.s.s)
\end{aligned}$$

One may observe that the above inference assumes the data structure allows recursive typing. Since $p.s.s$ is written $(!(!p.s)).s$ in imperative PCFv, this calculation corresponds to

$(m = (!(!p.s)).s)\{\!|v/!(!u.s)|\!\}$ in the present logic, though in many cases either such expansion is unnecessary or partial expansion suffices. Since the operation easily extends to formulae, we can now express a corresponding general axiom:

$$\{C\{\!|e'/e|\!\}\}\ e := e'\ \{C\}$$

which, because of the definition of conditional update above, means the same thing as $\{C\{\!|e'/!e|\!\}\}\ e := e'\ \{C\}$ in our notation.

Morris's approach is equivalent to Cartwright and Oppen's in the sense that formulae with conditional expressions are easily decomposable into those without it using (in)equations on reference names. Morris's approach is more syntactic and is presented purely in the setting of the first-order logic with equality. Morris (Morris, 1982a; Morris, 1982d; Morris, 1982c) further extends his method with axioms for linked lists, and used the resulting framework for verification of a Schorr-Waite algorithm.

Separation Logic. A different approach to the logical treatment of aliasing, based on Burstall's early work, is *Separation Logic* by Reynolds, O'Hearn and others (Reynolds, 2002; O'Hearn *et al.*, 2004). They introduce a novel conjunction $*$ that also stipulates disjointness of memory regions. Separation Logic uses the semantics and rules of Hoare logic for alias-free stack-allocated variables while introducing alias-sensitive rules for variables on heaps. We discuss their work in some detail since it contrast interestingly with ours, both philosophically and technically. Their logic starts from a resource-aware assignment rule (Reynolds, 2002): $\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}$ where e and e' do not include dereference of heap variables and "$x \mapsto -$" stands for "$\exists i.(x \mapsto i)$". The rule *demands* that a memory cell be available at address e, demonstrating the resource-oriented nature of the logic (motivated by reasoning for low-level code). Consequently, $\{\mathsf{T}\}\ [e] := [e]\ \{\mathsf{T}\}$ is unsound in their logic. This command corresponds to $x := !x$ in our notation. $\{\mathsf{T}\}\ x := !x\ \{\mathsf{T}\}$ is trivially sound in original Hoare logic (Hoare, 1969) and ours.

On the basis of these resource-oriented proof rules, (Reynolds, 2002; O'Hearn *et al.*, 2004) propose a variant of the invariance rule.

$$\frac{\{C\}\ P\ \{C'\} \quad \mathsf{fv}(C_0) \cap \mathsf{modify}(P) = \emptyset}{\{C * C_0\}\ P\ \{C' * C_0\}} \tag{101}$$

The second premise is standard side condition in Hoare logic ($\mathsf{modify}(P)$ 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 described above, by which all the variables (addresses) in the heap which P may write to are explicitly mentioned in C. Like the standard invariance rule, this rule is intended to serve as an aid for modular verification of program correctness.

Separation Logic's ability to reason about aliased references crucially depends on its resource-oriented nature, the separating conjunction $*$ and a special predicate $\mapsto$ to represent content of memory cells. In contrast, the present work aims at a precise logical articulation of observational meaning of programs in the traditions of both Hennessy-Milner logic and Hoare logic, as exemplified by Theorem 6. Another difference is that our logic aims to make the best of first-order logic with equality to represent general aliasing situations. These differences come to life for example in the [*Invariance*] rule of Section

7, which plays a role similar to (101). Our rule relies on purely compositional reasoning about observable behaviour, which, as examples in the previous section may suggest, contributes to tractability in reasoning. A concrete derivation may elucidate the difference, for example the inference below for $x := 2; y :=!z$ through a direct application of (101) and [*Assign, Inv, Seq, Cons*].

$$\dfrac{\dfrac{\{x \mapsto -\}\, x := 2\, \{x \mapsto 2\}}{\{y \mapsto - \ \wedge\ z \mapsto i\}\, y :=!z\, \{y \mapsto i \ \wedge\ z \mapsto i\}}}{\{x \mapsto - \ *\ (y \mapsto - \ \wedge\ z \mapsto -)\}\, x := 2; y :=!z\, \{x \mapsto 2 \ *\ \exists i.(y \mapsto i \ \wedge\ z \mapsto i)\}}$$

For the same program, a direct application of our invariance rule [*Seq-I*] gives:

$$\dfrac{\dfrac{\{\mathsf{T}\}\, x := 2\, \{!x = 2\}@x \quad \text{(Assign)}}{\{\mathsf{T}\}\, y :=!z\, \{!y =!z\}@y \quad \text{(Assign)}}}{\{\mathsf{T}\}\, x := 2; y :=!z\, \{\langle !y\rangle\, !x = 2 \wedge\ !y =!z\}@xy \quad \text{(Seq-I)}}$$

Reflecting observational nature, the pre-condition simply stays empty. Our inference does not require x and y to be distinct: $\langle !y\rangle\, !x = 2 \ \wedge\ !y =!z$ is equivalent to $(x \neq y \supset !x = 2) \ \wedge\ !y =!z$, which is more general than $x \mapsto 2 \ *\ \exists i.(y \mapsto i \ \wedge\ z \mapsto i)$. Intuitively this is because content quantification, here $\langle !y\rangle$, offers a more refined form of protection from sharing/aliasing.

These examples suggest a gain in generality by using the proposed logical framework for representation of sharing and disjointness of data structures. While $C_1 * C_2$ is practically embeddable as $[!\tilde{e}_2]C_1 \wedge [!\tilde{e}_1]C_2$ where $\tilde{e}_i$ exhausts active dereferences of C_i, the examples argue that the use of write sets in located judgements/assertions offers a more precise description and smooth reasoning. On its observational basis, the present logic may incorporate resource-sensitive aspects through separate predicates (e.g. a predicate $\mathsf{allocated}(e)$ may say e of a reference type is allocated). Because of differences in orientation, we expect a fruitful interplay between Separation Logic and our proposal.

One example of such interplay, applying the analytical power of the present logic, is a simplification and generalisation of a refined invariance rule involving procedures by O'Hearn, Yang and Reynolds (O'Hearn *et al.*, 2004). Their rule has several side conditions about the behaviour of programs, including an operational condition on write effects, and restrictions on the use of formulae: below we present the corresponding rule in our logic.

$$\dfrac{C_1\ !\tilde{x}\text{-free} \quad \{C_0\}\, N\, \{C_0' * C_1\}@\tilde{x}\tilde{y} \quad \{C^{-f} \wedge \{C_0\}\, f\, \{C_0'\}@\tilde{x}\}\, M :_u\, \{C'\}@\tilde{x}}{\{C \wedge C_1\}\ \mathtt{let}\ f = \lambda().N\ \mathtt{in}\ M :_u\, \{C' \wedge C_1\}@\tilde{x}\tilde{y}} \qquad (102)$$

Here f should be *ephemeral* in the sense that it occurs in M only in the shape of $f()$ and never under λ-abstraction. This is easily checkable by typing. The rule says if a program M uses a procedure f assuming that it only alters $\tilde{x}$, and under that condition M only alters the content of $\tilde{x}$, then if we instantiate f to a real program and it touches reference names distinct from $\tilde{x}$ but maintains the invariance at those reference names, then instantiating that procedure maintains the invariance. Ephemerality of f is needed, for if we store f or place it under abstraction, the invariance in stored/abstracted behaviour cannot be maintained: in contrast, in the above case, we can adjust the invariance at the time of instantiation once and for all. In comparison with the rule in (O'Hearn *et al.*, 2004), (102) differs in that it is

purely compositional, i.e. does not demand conditions on behaviours of M and N outside of judgements. Further, our rule does not restrict the use of stored higher-order procedures etc. in non-ephemeral procedure labels. This generality is obtained because we can now identify precisely why strengthening of invariance is possible in the specific setting the invariance rule in (O'Hearn *et al.*, 2004) deals with.

Further Related Work. There are other reasoning methods for programs with aliasing that are not directly about compositional program logics. In this category we find, for example, operational reasoning methods studied by Mason (Mason & Talcott, 1991) and Pitts and Stark (Pitts & Stark, 1998) (both also deal with local references). These approaches are complementary and their integration with logical methods such as ours an interesting subjects for further studies.

Aliasing is an essential feature in low-level code and system-level software. Apart from Separation Logic, there are several recent approaches which address formal safety guarantee of low-level code addressing higher-order procedures and aliasing in an organised way. An example of work in this direction is (Hamid & Shao, 2004), where integration of typed assembly code (Morrisett *et al.*, 1999) and Floyd-Hoare logic is studied to offer a formal framework to guarantee expressive safety properties for assembly code with references to higher-order code. How the present approach may be usable with lower level languages is currently being investigated.

One issue not discussed here is *data hiding*: for example a call $\texttt{putchar}(\texttt{buff}, \texttt{c})$ might, form the client's point of view, affect only the abstract buffer `buff`. But from the system's perspective the buffer implementation and the precise effect description would be complicated. The problem is that the system's perspective on `putchar` is hidden from the user. With this constraint, is it possible to obtain precise *specifications* even at the user level without revealing implementation detail? To achieve a smooth interplay between specification and hiding Leino and Nelson (Leino & Nelson, 2002) propose *abstraction dependencies*, a new construct that allows to specify how the user-level view of effects relates to the implementation view, but without sacrificing on the modularity afforded by hiding. Since the aliasing problem becomes more complicated with the diverging perspectives on software introduced by hiding, studying content quantification in this setting is sure to be interesting.

In (Ahmed *et al.*, 2005), Ahmed, Morrisett and Fluet present a framework ensuring type-safety for a higher-order call-by-value imperative language in the presence of *strong update*, i.e. the update of a variable which can change its type. This may be considered an extreme form of aliasing. We believe that content quantification can be generalised to allow compositional and observationally complete logical reasoning even with strong update.

References

Ahmed, Amal, Morrisett, Greg, & Fluet, Matthew. (2005). L3: A Linear Language with Locations. *Pages 293–307 of: Proc. TLCA'05*. LNCS, vol. 3461.

Apt, Krzysztof R. (1981). Ten Years of Hoare Logic: a survey. *TOPLAS*, **3**, 431–483.

Bauer, Friedrich L., Dijkstra, Edsger W., & Hoare, Tony (eds). (1982). *Theoretical Foundations of Programming Methodology*. Reidel.

Berger, Martin, Honda, Kohei, & Yoshida, Nobuko. (2005). A Logical Analysis of Aliasing in Imperative Higher-Order Functions. *Pages 280–293 of: Proc. ICFP'05*.

Bornat, Richard. (2000). Proving Pointer Programs in Hoare Logic. *Proc. Mathematics of Program Construction*. LNCS, vol. 1837. Springer-Verlag.

Cartwright, Robert, & Oppen, Derek C. (1978). Unrestricted procedure calls in Hoare's logic. *Pages 131–140 of: Proc. POPL*.

Cartwright, Robert, & Oppen, Derek C. (1981). The Logic of Aliasing. *Acta Inf.*, **15**, 365–384.

Cousot, Patrick. (1999). Methods and Logics for Proving Programs. *Pages 843–993 of: Handbook of Theoretical Computer Science*, vol. B.

Filliâtre, Jean-Christophe, & Magaud, Nicolas. (1999). Certification of Sorting Algorithms in the System Coq. *Proc. Theorem Proving in Higher Order Logics*.

Floyd, Robert W. (1967). Assigning Meaning to Programs. *Proc. Symp. in Applied Mathematics*, vol. 19.

Greif, Irene, & Meyer, Albert R. (1981). Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer. *Acm Trans. Program. Lang. Syst.*, **3**(4).

Gries, David, & Levin, Gary. (1980). Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst.*, **2**(4), 564–579.

Grossman, Dan, Morrisett, Greg, Jim, Trevor, Hicks, Michael, Wang, Yanling, & Cheney, James. (2002). Region-Based Memory Management in Cyclone. *Proc. PLDI'02*.

Gunter, Carl A. (1995). *Semantics of Programming Languages*. MIT Press.

Hamid, Nadeem A., & Shao, Zhong. 2004 (September). Interfacing Hoare Logic and Type Systems for Foundational Proof-Carrying Code. *Pages 118–135 of: Proc. TPHOL'04*. LNCS, vol. 3223.

Hennessy, Matthew, & Milner, Robin. (1985). Algebraic Laws for Non-Determinism and Concurrency. *JACM*, **32**(1).

Hoare, Tony. (1969). An Axiomatic Basis of Computer Crogramming. *CACM*, **12**.

Hoare, Tony, & Jifeng, He. (1998). *Unifying Theories of Programming*. Prentice-Hall International.

Honda, Kohei. (2004). From Process Logic to Program Logic. *Pages 163–174 of: Proc. ICFP'04*. ACM Press. A long version available from www.dcs.qmul.ac.uk/˜kohei/logics.

Honda, Kohei, & Yoshida, Nobuko. (2004). A Compositional Logic for Polymorphic Higher-Order Functions. *Pages 191–202 of: Proc. PPDP'04*.

Honda, Kohei, Yoshida, Nobuko, & Berger, Martin. (2005). An Observationally Complete Program Logic for Imperative Higher-Order Functions. *Pages 270–279 of: Proc. LICS'05*. Full version available from: www.dcs.qmul.ac.uk/˜kohei/logics.

Janssen, T. M. V., & van Emde Boas, Peter. (1977). On the Proper Treatment of Referencing, Dereferencing and Assignment. *Pages 282–300 of: Proc. ICALP'77*.

Kernighan, Brian W., & Ritchie, Dennis M. (1988). *The C Programming Language, Second Ed.* Englewood Cliffs, New Jersey: Prentice-Hall.

Kleymann, Thomas. (1998). *Hoare Logic and Auxiliary Variables*. Tech. rept. ECS-LFCS-98-399. University of Edinburgh, LFCS.

Kulczycki, Gregory W., Sitaraman, Murali, Ogden, William F., & Leavens, Gary T. 2003 (December). *Reasoning about procedure calls with repeated arguments and the reference-value distinction*. Tech. rept. TR ♯02-13a. Dept. of Comp. Sci., Iowa State Univ.

Leino, K. Rustan M., & Nelson, Greg. (2002). Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.*, **24**(5), 491–553.

Luckham, David C., & Suzuki, Norihisa. (1979). Verification of Array, Record, and Pointer Operations in Pascal. *ACM Trans. Program. Lang. Syst.*, **1**(2), 226–244.

Mason, Ian, & Talcott, Carolyn. (1991). Equivalence in functional languages with effects. *JFP*, **1**(3), 287–327.

McCarthy, John L. (1962). Towards a Mathematical Science of Computation. *Pages 21–28 of: Proc. IFIP Congress*.

Mendelson, Elliot. (1987). *Introduction to Mathematical Logic*. Wadsworth Inc.

Meyer, Albert R., & Sieber, Kurt. (1988). Towards fully abstract semantics for local variables. *Proc. POPL'88*.

Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17**(Aug.), 348–375.

Milner, Robin, Tofte, M., & Harper, R. (1990). *The Definition of Standard ML*. MIT Press.

Milner, Robin, Parrow, Joachim, & Walker, David. (1992). A Calculus of Mobile Processes, Parts I and II. *Info. & Comp.*, **100**(1), 1–77.

Morris, Joseph M. (1982a). A General Axiom of Assignment. *Pages 25–34 of:* (Bauer *et al.*, 1982). Reidel.

Morris, Joseph M. (1982b). A General Axiom of Assignment/Assignment and Linked Data Structures/ A proof of the Schorr-Waite Algorithm. *Pages 25–52 of:* (Bauer *et al.*, 1982). Reidel.

Morris, Joseph M. (1982c). A Proof of the Schorr-WaiteAlgorithm. *Pages 44–52 of:* (Bauer *et al.*, 1982). Reidel.

Morris, Joseph M. (1982d). Assignment and Linked Data Structures. *Pages 35–43 of:* (Bauer *et al.*, 1982).

Morrisett, Greg, Walker, David, Crary, Karl, & Glew, Neal. (1999). From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*, **21**(3), 527–568.

O'Hearn, Peter, Yang, Hongseok, & Reynolds, John C. (2004). Separation and Information Hiding. *Proc. POPL'04*.

Peyton Jones, Simon, Ramsey, Norman, & Reig, Fermin. (1999). C–: a Portable Assembly Language that Supports Garbage Collection. *Proc. PPDP*.

Pierce, Benjamin C. (2002). *Type Systems and Programming Languages*. MIT Press.

Pitts, Andy M., & Stark, Ian D. B. (1998). Operational Reasoning for Functions with Local State. *Pages 227–273 of: HOOTS'98*.

Reynolds, John C. (2002). Separation logic: a logic for shared mutable data structures. *Proc. LICS'02*.

Shao, Zhong. (1997). An Overview of the FLINT/ML Compiler. *Proc. Workshop on Types in Compilation (TIC'97)*.

Trakhtenbrot, Boris, Halpern, Joseph, & Meyer, Albert. (1984). From Denotational to Operational and Axiomatic Semantics for ALGOL-like Languages: an Overview. *Pages 474–500 of: Proc. CMU Workshop on Logic of Programs*. LNCS, vol. 164.

Yoshida, Nobuko, Honda, Kohei, & Berger, Martin. (2006). *Local State in Hoare Logic for Imperative Higher-Order Functions*. draft.