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 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.

Contents

1 Introduction 1
2 Language 4
3 Logic (1): Assertions 6
3.1 Terms and Formulae 6
3.2 Models and the Semantics of Terms and Formulae 8
3.3 Content Quantification 9
3.4 Examples of Assertions 12
4 Logic (2): Axioms 17
4.1 Axioms for Content Quantification 17
4.2 Axioms for Evaluation Formulae 22
4.3 Axioms for Data Types 22
5 Logic (3): Judgements and Proof Rules 23
5.1 Judgements and their Semantics 24
5.2 Proof Rules 25
6 Soundness and Elimination of Content Quantification 28
6.1 Elimination of Content Quantification 28
6.2 Soundness 32

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

7 Located Assertions and Reasoning 32
  7.1 Motivation and Syntax 32
  7.2 Rules and Axioms for Located Assertions 34
  7.3 Proof Rules for Imperative Idioms 41
  7.4 Generalisation of Effects in Located Assertions 42
8 Reasoning Examples 44
  8.1 Questionable Double (1): Direct Reasoning 44
  8.2 Questionable Double (2): Located Reasoning 46
  8.3 Swap 46
  8.4 Circular References 48
  8.5 A Polymorphic, Higher-Order Procedure: Quicksort 48
9 Conclusion 55
  9.1 Local References 56
  9.2 Related Work 56
References 61
A Language Details 63
  A.1 Typing 63
  A.2 Dynamics 64
B Syntactic Substitution and Name Capture 65
C Some Proofs for Propositions 1, 2 and 3 66
D Soundness 68

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

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

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.

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:

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

where $\alpha = \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat}))$ is the type of x, y. 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 us assume x and y store distinct references). If we pass a function $\lambda xy.()$, which takes two arguments and returns the unique value of Unit-type, as f, then, after execution, $!x$ stores 3 and $!y$ stores 2. But if the standard swapping function $\mathtt{swap} \overset{\text{def}}{=} \lambda ab.\mathtt{let}\ c = !b\ \mathtt{in}\ (b := !a; a := c)$ is passed, the content of x and y is swapped and $!x$ now stores 2 while $!y$ stores 3. Such interplay between higher-order procedures and aliasing is common in many non-trivial programs in ML, C and more recent typed and untyped low-level languages (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 (written 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 work by Cartwright-Oppen and Morris, remarkable as it is, still begs the question how to reason about programs with aliasing in a tractable way. The first issue is calculation of validity in assertions involving semantic substitutions. 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 an 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 descriptions 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:

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

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, content quantification gives a clear logical description of alias-robust substitution:

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

From (3) and (4), the logical equivalence (1) is immediate, recovering (2) as a rule of inference. As content quantification has a straightforward axiomatisation, this decomposition enables a rich set of methods and axioms provided by first-order logic, leading to efficient calculation of validity, while subsuming Cartwright-Oppen/Morris's methods. This is because logical calculation can now focus on those parts of a formula which do get affected by state change: just like lazy evaluation, we do not have to calculate parts not immediately needed. For example let $C \overset{\text{def}}{\equiv} C_1 \wedge [!x]C_2 \wedge \langle !x \rangle C_3$. To calculate $C\{\!|e/!x|\!\}$, we only have to consider $C_1\{\!|e/!x|\!\}$, since, by axioms discussed later:

$$(C_1 \wedge [!x]C_2 \wedge \langle !x \rangle C_3)\{\!|e/!x|\!\} \;\equiv\; C_1\{\!|e/!x|\!\} \wedge ([!x]C_2 \wedge \langle !x \rangle C_3)$$

Here the two content quantifications, $\langle !x \rangle$ and $[!x]$, respectively protect C_1 and C_2 from manipulation of content (here substitution) of x. In later sections we shall demonstrate this point through examples.

Secondly, content quantification provides a powerful descriptive and reasoning framework when used in conjunction with the standard logical primitives. By allowing hypothetical statements about the content of references separate from statements about reference names themselves (which is the central logical feature of these operators), complex aliasing situations are given simple, succinct descriptions: intuitively, $[!x]C$ asserts, in one go, that C holds regardless of what is stored at x and, in addition, if C makes a non-trivial assertion about the content of a syntactically distinct reference y (for example C may state $!y = 3$), then x and y cannot be aliased (and dually for $\langle !x \rangle C$). This is often useful, for example when reasoning about the aliasing taking place when dealing with arrays and stack-located references. Content quantification works seamlessly with the logical machinery for capturing pure and imperative higher-order behaviour studied in (Honda, 2004; Honda & Yoshida, 2004; Honda *et al.*, 2005) and thus facilitates precise description and efficient reasoning for a large class of higher-order behaviour and data structures.

Third, and somewhat paradoxically, the merits of content quantification come without additional expressive power: any formula containing content quantification can be translated, up to logical equivalence, into one without. While establishing this result, we 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. This does not diminish the significance of content quantification: It is only through identifying this concept as proper logical primitives with associated axioms that we obtain a precise logical understanding of the lazy update calculation and the concise descriptions of complex aliasing noted above. Further, the elimination result allows straightforward extension of content quantifications to e.g. quantification of an arbitrary set of references, as we shall see later. These merits suggest that content quantification is a distilled and therefore efficient logical construct for reasoning about aliasing, augmenting the standard logical constructs.

Structure of the Paper

In the rest of the paper, Section 2 briefly summarises the programming language. Section 3 introduces the assertion language and its semantics. Section 4 discusses axioms. Section 5 introduces basic proof rules for the logic. Section 6 discusses several key technical properties of the proposed logic: elimination of content quantification and soundness of axioms and proof rules. Section 7 introduces located assertions and associated reasoning principles for effective reasoning about programs with aliasing. Section 8 gives non-trivial reasoning examples using the logic, including that of polymorphic higher-order Quicksort, taken from the corresponding C program by Kernighan and Ritchie. Section 9 discusses 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 examples and comprehensive comparisons with related work.

Position of the Logic for Aliasing in Related Logics

The logic presented here is part of a family of stratified program logics, starting from one for pure higher-order functions (Honda *et al.*, 2006a; Honda & Yoshida, 2004; Honda, 2004) and its immediately generalisation to imperative higher-order functions (Honda *et al.*, 2005), to logics for languages with more complex behaviours. This allows us to use simple reasoning methods for more straightforward behaviour such as imperative programs without aliasing, and resorting to complex logical technology only for more complex behaviour. This is the rationale for studying logics for cleanly delineated classes of behaviour, and this work focuses on general aliasing, an important instance of such a class, extending our preceding program logic (Honda *et al.*, 2005). Many practical programs written in C and ML combine higher-order functions, aliasing, and reference declarations which are never exported beyond their scope (so-called stack-allocated variables); this class of programs can be reasoned about in the logic in this paper (see Section 9).

2 Language

The programming language we shall use in the present study is call-by-value PCF with unit, sums and products, augmented with imperative variables, but without local reference generation (investigated in (Yoshida *et al.*, 2006)). Assuming 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.

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

Abstraction, recursion and the case construct are annotated by types. Constants $(\mathtt{c}, \mathtt{c}', \ldots)$ include unit $()$, natural numbers $\mathtt{n}$, booleans $\mathtt{b}$ (either true $\mathtt{t}$ or false $\mathtt{f}$) and locations $(l, l', \ldots)$. $\mathsf{op}(\tilde{M})$ (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 location (say l), evaluates N and obtains a value (say V), and assigns V to l. All these constructs are standard, cf. (Pierce, 2002; Gunter, 1995). The notions of binding and α-convertibility are also conventional. $\mathsf{fv}(M)$ and $\mathsf{fl}(M)$ denotes the sets of free variables and locations in M, respectively. We use abbreviations such as:

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

Let $\mathrm{X}, \mathrm{Y}, \ldots$ range over an infinite set of type variables. Types are ranged over by $\alpha, \beta, \ldots$ and are given by the following grammar.

$$\alpha, \beta \quad ::= \quad \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \alpha \Rightarrow \beta \mid \alpha \times \beta \mid \alpha + \beta \mid \mathsf{Ref}(\alpha) \mid \mathrm{X} \mid \mu\mathrm{X}.\alpha$$

We call types of the form $\mathsf{Ref}(\alpha)$ *reference types*. All others are *value types*. A type is *closed* if it does not contain free occurrences of type variables. Although the grammar is standard, some points are worth noting in the light of their status in the present theory. We write $\mathsf{ftv}(\alpha)$ to mean the set of α's free type variables. The type List_α is given by the recursive definition below. $\mathsf{List}_\alpha \stackrel{\text{def}}{=} \mu\mathrm{X}.(\mathsf{Unit} + (\alpha \times \mathsf{Ref}(\mathrm{X})))$ As this type will be often used in examples, we introduce some shorthands. We write nil for $\mathtt{inj}_1(())$ and $a :: b$ to abbreviate $\mathtt{inj}_2(\langle a, b \rangle)$. To facilitate reasoning, we sugar the $\mathtt{case}$-construct: $\mathtt{case}\ e\ \mathtt{of}\ \{\mathsf{nil} \triangleright M_1 \mid a :: l \triangleright M_2\}$ is a shorthand for $\mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{in}_i(x_i).N_i\}_{i \in \{1,2\}}$ where $N_1 = M_1$ and $N_2 = M_2[\pi_1(x_2)/a][\pi_2(x_2)/b]$. Naturally, e must be of type List_α.

A *basis* is a finite map from names and locations to closed 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. 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, always assuming $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. We take the equi-isomorphic approach for recursive types, and will identify types and programs up to equi-isomorphism. The typing rules are standard (Pierce, 2002) and listed in Appendix A, using sequents $\Gamma \vdash M : \alpha$, which say that M has type α under basis Γ. We often write $M^{\Gamma;\alpha}$ for $\Gamma \vdash M : \alpha$.

The dynamics of the language is given by straightforward call-by-value reductions using a store (Gunter, 1995; Pierce, 2002), where a *store* $(\sigma, \sigma', \ldots)$ is a finite map from locations to closed values. We write $\mathsf{dom}(\sigma)$ for the domain of σ. A *configuration* is a pair of a closed program and a store. Then *reduction* is a binary relation over configurations, written $(M, \sigma) \longrightarrow (M', \sigma')$, generated by the rules in Appendix A. 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.

3 Logic (1): Assertions

3.1 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 quantification over type variables, evaluation and quantification over store content. The latter is the only addition substantial to the logic in (Honda *et al.*, 2005).

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

Here $\star \in \{\wedge, \vee, \supset\}$ and $\mathcal{Q} \in \{\forall, \exists\}$. 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 (c, c',...) include unit $()$, numerals n, booleans b (either true t or false f) and labels l. 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 pairing and the 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 an *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 x, 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. Both are explained in detail below, but 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* bound in $[!x]C$ and $\langle !x \rangle C$. 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. $\mathsf{fl}(C)$, $\mathsf{bv}(C)$) denotes the set of free variables (resp. free locations, 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 $[!x]C$. Note that expressions do not contain abstractions, applications and assignments because they involve non-trivial dynamics (possibly infinite reductions). Similarly, for sums, and products, we do not provide destructors (like $\pi_i(\cdot)$), only constructors in the expression language. Nevertheless, our language is sufficiently expressive for reasoning about arbitrary data-structures.

Terms are typed inductively starting from types for variables and constants and signatures for operators. The typing rules are given in Figure 1. Recalling that $\Gamma; \Delta$ indicates a map from names to types such that Γ (resp. Δ) is about non-reference types (resp. reference types), we write $\Gamma; \Delta \vdash e : \alpha$ when e has type α such that free names in e have types

$$\frac{-}{\Gamma;\Delta \vdash x : \Gamma(x)} \quad \frac{-}{\Gamma;\Delta \vdash \mathsf{n} : \mathsf{Nat}} \quad \frac{-}{\Gamma;\Delta \vdash \mathsf{t},\mathsf{f} : \mathsf{Bool}} \quad \frac{-}{\Gamma;\Delta \vdash l : \Delta(l)} \quad \frac{\Gamma;\Delta \vdash e : \mathsf{Bool}}{\Gamma;\Delta \vdash \neg e : \mathsf{Bool}}$$

$$\frac{\Gamma;\Delta \vdash e_i : \alpha_i}{\Gamma;\Delta \vdash e_1 = e_2} \quad \frac{\Gamma;\Delta \vdash e_i : \alpha_i}{\Gamma;\Delta \vdash (e_1,e_2) : \alpha_1 \times \alpha_2} \quad \frac{\Gamma;\Delta \vdash e : \alpha_i}{\Gamma;\Delta \vdash \mathsf{inj}_i^{\alpha_1+\alpha_2}(e) : \alpha_1 + \alpha_2} \quad \frac{\Gamma;\Delta \vdash e : \mathsf{Ref}(\alpha)}{\Gamma;\Delta \vdash !e : \alpha}$$

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

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

Fig. 1. Typing Rules for Terms and Formulae

following $\Gamma;\Delta$; and $\Gamma;\Delta \vdash C$ when all terms in C are well-typed under $\Gamma;\Delta$. It may be worth pointing out that in equations $e = e'$ we do not require e and e' to have the same type. This allows us to type equations like $y^{\mathsf{Ref(Nat)}} = a^{\mathsf{Ref(X)}} \vee y^{\mathsf{Ref}(\alpha \Rightarrow \beta)} = a^{\mathsf{Ref(X)}}$ using type variables. This will be useful for reasoning about effectful programs as we demonstrate later. Equations between terms will always evaluate to F. In the introduction rule for first-order quantifiers, the variable under abstraction can be a reference or not. *Syntactic substitution* $C[e/!x]$ is also used frequently: the definition is standard, save for some subtlety regarding substitution into the pre/postcondition of evaluation formulae, details can be found in Appendix B. We shall also use positive inductive formulae freely, without further comment. *Henceforth we only treat well-typed terms and formulae.*

Further notational conventions follow.

Convention 1 (assertions)

- 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.
- Logical connectives are used not only syntactically but also semantically, i.e. when discussing meta-logical and other notions of validity.
- 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 $\{C\}\, e_1 \bullet e_2 \;=\; ()\, \{C'\}$.
- For convenience of rule presentation we will use projections $\pi_i(e)$ as a derived term. They are redundant in that any formula containing projections can be translated into one without: for example $\pi_1(e) = e'$ can be expressed as $\exists y.e = \langle e', y \rangle$.

3.2 Models and the Semantics of Terms and Formulae

We continue by formalising the semantics of expressions and assertions term models. A detailed and informal description content quantification follows in Section 3.3.

Models in the present setting are very much like those of (Honda *et al.*, 2005), which used pairs (ξ,σ), where ξ maps non-reference names to its denotations and σ is a store. The only change is that to be able to model aliasing, we employ locations: the denotation of a reference name is now a location and stores are maps from locations.

Definition 1 (models) A term is *closed* if it has not free variables. A *model of type* $\Theta = \Gamma;\Delta$, ranged over by $\mathcal{M},\mathcal{M}',...$, with $\mathsf{fv}(\Delta)\cup = \mathsf{ftv}(\Delta) = \emptyset$, is a tuple (ξ,σ) where:

- ξ, called *environment*, is a finite map from (1) $\mathsf{dom}(\Theta)$ to closed values such that, for each $x \in \mathsf{dom}(\Gamma)$, $\xi(x)$ is typed as $\Theta(x)$ under Δ, i.e. $\Delta \vdash \xi(x) : \Theta(x)$; and (2) from type variables to closed types.
- σ, called *store*, is a finite map from labels to closed values such that for each $l \in \mathsf{dom}(\sigma)$, if $\Delta(l)$ has type $\mathsf{Ref}(\alpha)$, then $\sigma(l)$ has type α under Δ, i.e. $\Delta \vdash \sigma(l) : \alpha$.

The interpretation of terms is straightforward.

Definition 2 Let $\Gamma;\Delta \vdash e : \alpha$, $\Gamma;\Delta \vdash \mathcal{M}$ and $\mathcal{M} = (\xi,\sigma)$. Then the *interpretation of e under* $\mathcal{M}$, denoted $[\![e]\!]_{\mathcal{M}}$ is inductively given by the clauses below.

$$\begin{array}{rclrcl} [\![x^\alpha]\!]_{\mathcal{M}} & = & \xi(x) & [\![\mathsf{op}(\tilde{e})]\!]_{\mathcal{M}} & = & \mathsf{op}([\![\tilde{e}]\!]_{\mathcal{M}}) \\ [\![!e]\!]_{\mathcal{M}} & = & \sigma([\![e]\!]_{\mathcal{M}}) & [\![\langle e,e'\rangle]\!]_{\mathcal{M}} & = & \langle [\![e]\!]_{\mathcal{M}}, [\![e']\!]_{\mathcal{M}} \rangle \\ [\![\mathsf{c}^\alpha]\!]_{\mathcal{M}} & = & \mathsf{c} & [\![\mathtt{inj}_i(e)]\!]_{\mathcal{M}} & = & \mathsf{inj}_i([\![e]\!]_{\mathcal{M}}) \end{array}$$

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

Notation 1 The following notation is useful later. Let $\mathcal{M} = (\xi,\sigma)$.

- Given $u \notin \mathsf{fv}(\mathcal{M})$, 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, and similarly for $\mathcal{M}\cdot \mathrm{X} : \alpha$, provided $\mathrm{X} \notin \mathsf{ftv}(\mathcal{M})$ and α closed.
- If $l \in \mathsf{dom}(\sigma)$, $\mathcal{M}\cdot[l \mapsto V]$ is the model obtained from $\mathcal{M}$ by updating the store at l with V. Similarly, and assuming appropriate typing, $\mathcal{M}[x \mapsto V]$ means $\mathcal{M}[l \mapsto V]$, where the reference x is mapped to location l by $\mathcal{M}$.
- Given $x \notin \mathsf{fv}(\mathcal{M}_1)$, we write $\mathcal{M}_1 \leq_{x:\alpha} \mathcal{M}_2$ if, for some V, either $\mathcal{M}_2 \overset{\text{def}}{=} \mathcal{M}_1 \cdot x : V^\alpha$; or $\alpha = \mathsf{Ref}(\beta)$ and $\mathcal{M}_2 \overset{\text{def}}{=} \mathcal{M}_1 \cdot x : l \cdot [l \mapsto V]$ with $l \notin \mathsf{fl}(\mathcal{M}_1)$. We write $\mathcal{M} \leq_{\tilde{x}:\tilde{\alpha}} \mathcal{M}'$ for $\mathcal{M} \leq_{x_0:\alpha_0} \cdots \leq_{x_{n-1}:\alpha_{n-1}} \mathcal{M}'$.

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 a fresh location l as denotation for x and a value stored at l, or, alternatively, coalesces x with another, existing reference name, by letting the x's denotation be an already existing location. 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 the interpretation of first-order quantifiers.

We use the following standard observational equivalence between terms.

Definition 3 (observational congruence) Assume that $\Gamma;\Delta \vdash M_{1,2} : \alpha$. We write $\Gamma;\Delta \vdash (M_1,\sigma_1) \cong (M_2,\sigma_2)$ if, for each typed context $C[\cdot]$ such that $\Delta \vdash C[M_i] : \mathsf{Unit}$ for $i = 1,2$: $(C[M_1],\ \sigma_1) \Downarrow$ iff $(C[M_2],\ \sigma_2) \Downarrow$.

Next we present the satisfaction relation $\mathcal{M} \models C$. All definitions are standard except for evaluation formulae which follow (Honda *et al.*, 2005) content quantification and standard quantifiers, which use model extensions as introduced above.

Definition 4 Assume $\mathcal{M} = (\xi,\sigma)$ 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.

- $\mathcal{M} \models e_1^\alpha = e_2^\beta$ if $\alpha = \beta$ and $([\![e_1]\!]_{\mathcal{M}},\sigma) \cong ([\![e_2]\!]_{\mathcal{M}},\sigma)$.
- $\mathcal{M} \models \neg C$ 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 $\mathcal{M} \models C_2$.
- $\mathcal{M} \models \forall \mathrm{X}.C$ if for all closed types α, $\mathcal{M}\cdot\mathrm{X} : \alpha \models C$.
- $\mathcal{M} \models \exists \mathrm{X}.C$ if for some closed types α, $\mathcal{M}\cdot\mathrm{X} : \alpha \models C$.
- $\mathcal{M} \models \forall x^\alpha.C$ 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 $\mathcal{M}' \stackrel{\text{def}}{=} (\xi,\sigma')$ of type $\Gamma;\Delta$ such that $\mathcal{M}' \models C$, we have, for some V of appropriate type, we have $([\![e]\!]_{\mathcal{M}}[\![e']\!]_{\mathcal{M}},\ \sigma') \Downarrow (V,\sigma'')$ and $(\xi\cdot x{:}V,\ \sigma'') \models C'$.
- $\mathcal{M} \models [!e]C$ if $[\![e]\!]_{\mathcal{M}} = l$ and for all V of appropriate type, we have $\mathcal{M}[l \mapsto V] \models C$.
- $\mathcal{M} \models \langle !e \rangle C$ if $[\![e]\!]_{\mathcal{M}} = l$ and for some V of appropriate type, we have $\mathcal{M}[l \mapsto V] \models C$.

Some observations follow.

- 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 locations, but does not require aliasing.
- 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 location it denotes. Let it be l. Then the value stored at l in $\mathcal{M}$ is irrelevant, all we need to know is if there is some value V such that $\mathcal{M}[l \mapsto V]$ satisfies C.

3.3 Content Quantification

We continue with a more in-depth explanation of content quantification and its genesis.

Aliasing and Assignment. A good way of motivating content quantification might be by analysing Hoare's original assignment rule and its soundness proof.

$$[\textit{Assign-Orig}]\ \frac{-}{\{C[e/!x]\}\ x := e\ \{C\}} \tag{5}$$

In the absence of aliasing, this rule allows us to derive a sound and indeed best possible precondition for any program $x := e$, given a post-condition C. The rule works by applying a *syntactic* substitution $[e/!x]$ to C which replaces every occurrence of $!x$ with e. As an example, since $(!y = 2)[!x+1/!x]$ is equal to $!y = 2$ in the absence of aliasing,

$$\{!y = 2\}\ x := !x+1\ \{!y = 2\} \tag{6}$$

can be derived from [*Assign-Orig*]. But if aliasing is a possibility, this last assertion is inappropriate. Instead we need to consider two possibilities:

$$\{x \neq y \wedge !y = 2\}\ x := !x+1\ \{!y = 2\} \qquad\qquad \{x = y \wedge !y = 2\}\ x := !x+1\ \{!y = 2\}$$

Note that [*Assign-Orig*] corresponds to the left of those, but is useless for deriving the assertion on the right, due to the syntactic nature of the rule's substitution. In fact, we don't usually want two assertions here, but rather one, that covers both cases, that when x and y are aliases and that where they are not:

$$\{(x \neq y \supset !y = 2) \wedge (x = y \supset !y = 1)\}\ x := !x+1\ \{!y = 2\} \tag{7}$$

The key question is: what kind of rule would allow to derive assertions like (7) conveniently?

Content Quantification. To explain the role content quantifiers play in answering the last section's closing question, we consider Hoare's [*Assign-Orig*] once more. Proving its soundness amounts to establishing that $C[e/!x]$ is the unique (up to logical equivalence) C_0 such that equivalence

$$\mathcal{M} \models C_0 \quad \text{iff} \quad \mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \tag{8}$$

holds. Here $\mathcal{M}$ is an arbitrary model, $[\![e]\!]_{\mathcal{M}}$ gives the denotation of e in that model and $\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}]$ is the model that coincides with $\mathcal{M}$ everywhere, except that the reference x now stores $[\![e]\!]_{\mathcal{M}}$. We leave the details of models informal as models of Hoare's original logic are well-understood. Later we shall be more precise.

$\mathcal{M}$ represents the state *before* the assignment, while $\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}]$, the update of that state by e's denotation, is the state *after* assigning the denotation of e (calculated in the initial state $\mathcal{M}$) to the location referred to by x. Even if x is aliased, $\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}]$ gives the correct update. Thus (8) says that, for C to hold as the description *after* the assignment $x := e$, the pre-condition C_0 should be such that $\mathcal{M} \models C_0$ holds if and only if $\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C$ holds. We already know we cannot use the result of syntactic substitution $C[e/!x]$ for C_0 in the presence of aliasing. But why did it work in the alias-free setting? Let us consider a typical soundness argument.

$$\begin{aligned}
\mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \quad & \Leftrightarrow \quad \mathcal{M}\cdot m : [\![e]\!]_{\mathcal{M}}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \wedge !x = m && (9)\\
& \Leftrightarrow \quad \mathcal{M}\cdot m : [\![e]\!]_{\mathcal{M}} \models \exists x.(C \wedge !x = m) && (10)\\
& \Leftrightarrow \quad \mathcal{M} \models \exists m.(\exists x.(C \wedge !x = m) \wedge m = e) && (11)\\
& \Leftrightarrow \quad \mathcal{M} \models C[e/!x] && (12)
\end{aligned}$$

In (9), we simply adjoin a fresh name m, denoting $[\![e]\!]_{\mathcal{M}}$ to $\mathcal{M}$ and add $!x = m$ to the formula. In the next step (11) we hide x by existential abstraction, thus making the truth value of $\exists x.(C \wedge !x = m)$ independent from what the model stores at x. Hence we can drop the update operation $[x \mapsto [\![e]\!]_{\mathcal{M}}]$. This independence of the formula's truth value from x and its content is holds because in the absence of aliasing the only way to access x

or its content is by explicit dereference $!x$ of x (note that in Hoare's original logic non-trivial equations between references are prohibited). Equivalence (11) hides m, again using existential abstraction. The last line appeals to the equivalence

$$C[e/!x] \quad \equiv \quad \exists m.(\exists x.(C \wedge !x = m) \wedge m = e) \tag{13}$$

which gives a logical characterisation of syntactic substitution (we cannot simplify the right-hand side into $\exists x.(C \wedge !x = e)$ because $!x$ may occur in e).

We wish to extend this result so it also holds when references may be aliased. This means to find, given a postcondition C and an assignment $x := e$, a formula $C\{\!|e/!x|\!\}$ such that

$$\mathcal{M} \models C\{\!|e/!x|\!\} \quad \text{iff} \quad \mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C. \tag{14}$$

To find this formula, we mimic the derivation above: the first step is as (9) before, but the second fails:

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

The problem is that although x is no longer free in $\exists x.(C \wedge !m = x)$, the truth value of this formula may still depend on what is stored at x: for example C may be $!y = 7$ and the model might stipulate that y is an alias of x. To see how to deal with this conundrum, we note that for all $\mathcal{M}', C'$:

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

Since we are looking to make the truth value of $C \wedge !x = m$ independent from what stored at x in the model, not from x itself, this last equivalence is suggestive of our new quantifier $\langle !x \rangle C$ 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 x. With content quantification we could reason:

$$\begin{array}{rcll} \mathcal{M}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C & \Leftrightarrow & \mathcal{M} \cdot m : [\![e]\!]_{\mathcal{M}}[x \mapsto [\![e]\!]_{\mathcal{M}}] \models C \wedge !x = m & (15) \\ & \Leftrightarrow & \mathcal{M} \cdot m : [\![e]\!]_{\mathcal{M}} \models \langle !x \rangle (C \wedge !x = m) & (16) \\ & \Leftrightarrow & \mathcal{M} \models \exists m.(m = e \wedge \langle !x \rangle (C \wedge !x = m)) & (17) \end{array}$$

Hence content quantification allows to re-introduce the equivalence (13) that witnessed the correctness of the original Hoare rule, but enhanced, so it is robust under aliasing.

Definition 5 (logical substitutions) Assume m is fresh.

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

We call $\{\!|e/!x|\!\}$ *logical substitution* of e_2 for e_1. It substitutes e_2 for $!y$ whenever y is an alias of e_1. There is a dual operation $\overline{C\{\!|e_2/!e_1|\!\}} \stackrel{\text{def}}{=} \forall m.(e_2 = m \supset [!e_1](m = !e_1 \supset C))$. These substitutions may be called *logical content substitutions* or simply *logical substitutions*.

By the semantics of content quantification, derivation (15) - (17) re-establishes the logical equivalence in (8), but in an alias-robust way by replacing $C[e/!x]$ with $C\{\!\!\{e/!x\}\!\!\}$. Thus we now arrive at the following proof rule.

$$[\mathit{AssignBasic}]\ \frac{-}{\{C\{\!\!\{e/!x\}\!\!\}\}\ x := e\ \{C\}} \tag{18}$$

This rule subsumes the original rule (5) since $C\{\!\!\{e/!x\}\!\!\}$ coincides with $C[e/!x]$ whenever there is no aliasing. The semantic status of [*AssignBasic*] 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 syntactic substitution with 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 (7) easily? Does it allow extensions/generalisation to higher-order programming languages, for example those with the generalised assignment of the form $M := N$ where both M and N are appropriately typed arbitrary expressions? And can we reason about programs with aliasing tract ably and modularly using content quantification? We explore these topics in the following sections.

3.4 Examples of Assertions

Before presenting the formal semantics of expressions and formulae in Section 3.1, we discuss various example 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 often omit 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, "$!y = 6$", 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 $!z = 6$. 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.

$$\forall x.\forall i.\{!x = i\}\, u \bullet x\, \{!x = 2 \times i\} \tag{19}$$

We recall from Convention 1 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. 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 (19), giving a more focussed specification for $\lambda z.z := {!z} \times 2$. It uses inequalities on reference names and evaluation formulae to assert a strong property of imperative behaviour.

$$\forall x, i, \mathrm{X}, y^{\mathsf{Ref(X)}}, j^{\mathrm{X}}.\ \{!x = i \wedge x \neq y \wedge !y = j\}\, u \bullet x \{!x = 2 \times i \wedge x \neq y \wedge !y = j\}. \qquad (20)$$

The assertion says that, in addition to the property already stated in (19), the program guarantees that x is the only reference it may alter. It will be convenient to use the following abbreviation for (20).

$$\forall x, i.\ \{!x = i\}\, u \bullet x \{!x = 2 \times i\} \ @\, x \qquad (21)$$

Such assertions are called *located assertions.* (21) says the same thing as (20) but more concisely. This is discussed in more detail in Section 7.

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 quantifiers.

First, as a very simple example, consider an assertion

$$\langle !y \rangle\, !y = 1 \qquad (22)$$

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

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

In a hypothetical state, the content of a store may differ from the current one. Since we can surely hypothesise such a state, the statement is always true, so that (22) 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 "$x := m$".

$$\langle !x \rangle\, (!x = 2 \wedge\ !x = m). \qquad (23)$$

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 $m = !x$. This entails m and 2 being equal. As this does not claim anything about the content of x, only about its possible content, the only thing being asserted in (23) is that m denotes 2 in the model, hence (23) is logically equivalent to $m = 2$.

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

$$\langle !x \rangle\, (x = y \wedge\ !y = 1). \qquad (24)$$

This formula 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 (24) is logically equivalent to $x = y$.

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

$$\langle !x \rangle\,(x \neq y \wedge\ !y = 1). \tag{25}$$

The truth value of $x \neq y$ is independent from 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 (25) 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 (23) into "$!y = m$", obtaining:

$$\langle !y \rangle\,(!x = 2 \wedge\ !y = m) \tag{26}$$

which is the same thing as "$(!x = 2)\{\!|m/!y|\!\}$" up to logical equivalence. Thus (26) 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, (26) 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, $\exists i.(!y = i \wedge\ !x = 2 \wedge\ m = !y)$.

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

$$[!x]\,!y = 3 \tag{27}$$

assuming x is typed with $\mathsf{Ref(Nat)}$. By definition, (27) 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 (27) 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 (21) which is for the program $\lambda z.z := !z \times 2$, reproduced below:

$$\forall x, i.\ \{!x = i\}\, u \bullet x \{!x = 2 \times i\}\ @\, x \tag{28}$$

(28) 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.

$$\forall x,i.\ \{!x=i\ \wedge\ [!x]\mathit{Even}(!y)\}\,u\bullet x\,\{!x=2\times i\ \wedge\ [!x]\mathit{Even}(!y)\}\ @\,x \tag{29}$$

which is a consequence of (28) (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 (28) to (29) 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.

$$!x=2\ \wedge\ [!x]\,(!x=3\supset\mathsf{Odd}(!x))\ \wedge\ \{\mathsf{Odd}(!x)\}\,u\bullet()\,\{\mathsf{Even}(!x)\} \tag{30}$$

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 4):

$$!x=2\ \wedge\ [!x]\,(!x=3\supset\mathsf{Odd}(!x))\ \wedge\ \{!x=3\}\,u\bullet()\,\{\mathsf{Even}(!x)\} \tag{31}$$

By comparing (30) 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.

Assertions for the "Questionable Double". We continue with assertions for two simple programs. In Section 8 we shall show that these programs do satisfy these specifications using the proof rules of the logic to be introduced in Section 5.

The first program is the "Questionable Double":

$$\texttt{double?}\ \stackrel{\text{def}}{=}\ \lambda x^{\mathsf{Ref(Nat)}}.\lambda y^{\mathsf{Ref(Nat)}}.(x:=!x+!x\,;\,y:=!y+!y) \tag{32}$$

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. 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.

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

This meets the "expected" specification:

$$\forall x,y,i,j.\ \{!x = i \wedge\ !y = j\}\ u \bullet (x,y)\ \{!x = 2i \wedge\ !y = 2j\} \tag{33}$$

If we use a located assertion, we can further refine (33) to:

$$\forall x,y,i,j.\ \ \{!x = i \wedge\ !y = j\}\ u \bullet (x,y)\ \{!x = 2i \wedge\ !y = 2j\}\ @xy \tag{34}$$

The quantification of x and y extends to the whole formula, including the terminal $@xy$. (34) 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.

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

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

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

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

Again we can refine the program using a located assertion:

$$\mathsf{Swap}(u) \quad \stackrel{\text{def}}{=} \quad \forall xyij.\{!x = i \wedge !y = j\}\ u \bullet (x,y) \{!x = j \wedge !y = i\}\ @xy \tag{35}$$

which gives the full specification for `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. Although the programming language under consideration does not offer polymorphism, we could easily add this feature following (Honda & Yoshida, 2004). With this extension, we can refine (35).

$$\begin{array}{l} \forall \mathrm{X}.\forall x^{\mathsf{Ref(X)}}.\forall y^{\mathsf{Ref(X)}}.\forall i^{\mathrm{X}}.\forall j^{\mathrm{X}}. \\ \qquad \{!x = i \wedge !y = j\} u \bullet (x,y) \{!x = j \wedge !y = i\} @xy \end{array} \tag{36}$$

Circular References. We close this run of example assertions with discussing assignment to circular references. 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 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.

4 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.

4.1 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 given in (13), 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 succinct logical laws and reasoning principles, but for logical substitution. Since logical substitution has a logical decomposition through content quantification (Def. 5), we need to axiomatise the new quantifiers. The latters' semantics suggests fashioning this axiomatisation along the lines of axiomatisations for first-order quantifiers. E.g. (Mendelson, 1987) uses two axioms and a single rule of inference (in addition to Modus Ponens) as a formalisation of first-order universal quantification:

- $\forall x.(A \supset B) \supset A \supset \forall x.B$ provided x does not occur freely 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 2 is analogous: we replace first-order universal quantification by universal content quantification and instead of requiring "provided x does not occur freely in ..." we stipulate that the formula in question is syntactically $!x$-free, to be defined below. Just like "provided x does not occur in A" is a syntactic approximation to A's truth value being independent from what x's denotation may

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

Fig. 2. Axioms and rule of inference for content quantification.

be, so C's syntactic $!x$-freedom, written $C^{\text{-}!x}$, is a sufficient condition for C's truth value being independent from what a model stores at x. Finally, we regard $\langle !x \rangle C$ as standing for $\neg[!x]\,(\neg C)$ and add an axiom that connects the two forms of logical substitution given in Definition 5.

Syntactic $!x$***-Freedom.*** As just mentioned, syntactic $!x$-freedom is needed to express a crucial axiom for content quantification. To define this, we begin with the notion of active dereference $\mathsf{ad}(\cdot)$. 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 $[\![e]\!]_{\mathcal{M}_1}$ and $[\![e]\!]_{\mathcal{M}_2}$ are observationally equivalent, and similarly for formulae.

Definition 6 (active dereference) The *active dereferences* of an expression e, $\mathsf{ad}(e)$, are inductively defined:

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

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

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

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

Example 1 (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. $\{!x = 2\}!f \bullet !y = z\{!z = 1\}$ has $!f$ and $!y$ as active dereferences.
5. $[!!x]\,(!!x = !y)$ has two active dereferences, $!x$ and $!y$.
6. $\forall x.!!x = !y$ 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. This is not problematic as all subsequent uses of active dereferences will

insist on no member of $\mathsf{ad}(\cdot)$ being quantified in the relevant formula. An α-stable notion of active dereferences can be devised, but would be more complicated.

Definition 7 (syntactic $!x$-freedom) We generate the set of syntactically $!x$-free formulae, $\mathcal{SY}^{-!x}$, as follows:

1. $[!x]C \in \mathcal{SY}^{-!x}$, dually $\langle !x\rangle C \in \mathcal{SY}^{-!x}$.
2. $C \wedge \bigwedge_i e_i \neq x \in \mathcal{SY}^{-!x}$ and, dually, $\bigwedge_i e_i \neq x \supset C \in \mathcal{SY}^{-!x}$, 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 $\mathcal{SY}^{-!x}$ is again in $\mathcal{SY}^{-!x}$.

We write $C^{-!x}$ to indicate that $C \in \mathcal{SY}^{-!x}$.

Example 2 (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. $!!y = 3 \,\wedge\, x \neq !y$ is not syntactically $!x$-free, but $!!y = 3 \,\wedge\, x \neq !y \,\wedge\, x \neq y$ is.
4. On the other hand, $!y = 3$ is not syntactically $!x$-free, even up to $\equiv$. Intuitively, $C^{-!x}$ says C does not mention the content of x.

Explanation of the Axiomatisation. Among the axioms, ($\mathsf{CA1}$) corresponds to the 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. ($\mathsf{CA2}$) is a degenerate form of $\forall x.C \supset C[e/x]$. ($\mathsf{CA3}$) 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.[1]

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.

---

[1] 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 $C^{-!x} \supset [!x]C$, and closing all axioms under universal content quantification, cf. (Enderton, 2001)

Derived Laws. Now we discuss various useful formulae that are derivable in our axiomatisation of content quantification. Proofs are straightforward and mostly omitted, but a few are listed in Appendix 1.

Proposition 1 *(modal laws) All these laws have existential counterparts.*

1. $[!x](C_1 \supset C_2) \supset [!x]C_1 \supset [!x]C_2$.
2. $[!x]C' \equiv [!x]((C' \supset C) \supset C)$.
3. $([!x]C \wedge [!x]C') \equiv [!x](C \wedge C')$.
4. $[!x]C \equiv [!x][!x]C$.
5. $([!x]C \vee [!x]C') \supset [!x](C \vee C')$.
6. $[!x](C \vee C') \supset ([!x]C \vee \langle !x \rangle C')$.

(2) allows us to infer $[!x]C$ from $[!x]C'$ when $C' \supset C$ is a tautology. The existential counterpart of (4) is: $\langle !x \rangle \langle !x \rangle C \supset \langle !x \rangle C$.

Proposition 2 *(miscellaneous laws) In (1, 2, 3) below we have omitted the dual existential counterparts.*

1. *Let x and y be distinct symbols. Then:* $\forall y.[!x]C \equiv [!x]\forall y.C$, *and* $\exists y.\langle !x \rangle C \equiv \langle !x \rangle \exists y.C$.
2. $[!y][!x]C \equiv [!x][!y]C$.
3. $\langle !x \rangle [!x]C \equiv [!x]C$.
4. $\exists x.!x = y$.
5. $\neg[!x]\, !x \neq y$.
6. $C^{\text{-}!x} \supset [!x]C$.
7. $\neg[!x]C \equiv \langle !x \rangle \neg C$.
8. $[!x](!x = m \supset C) \equiv \langle !x \rangle (C \wedge !x = m)$.
9. $C\{\!|e'/!e|\!\} \equiv C\overline{\{\!|e'/!e|\!\}}$.
10. $[!x]C \supset C\{\!|e/x|\!\}$.
11. $C\{\!|e/x|\!\} \supset \langle !x \rangle C$.

Derived axiom (4) does not mention content quantification, but its derivation seems to require it. Laws (5) and (6) allow us to eliminate and introduce universal content quantifications, and play the key role in reasoning about aliasing. (5) is easily understood as an analogue of $\forall x.(x \neq y) \supset y \neq y \; (\equiv \mathsf{F})$. Note that the reverse of (6) does not hold: $[!x]\, !x = !x$ is true, despite $!x = !x$ not being syntactically $!x$-free. Laws (7) and (8) connect universal content quantification and its dual. From (8) we immediately infer (9), the equivalence between the two forms of logical substitutions introduced in Definition 5. (10) corresponds to the well-known implication $\forall x.A \supset A$ and (11) has the same relationship to $A[e/x] \supset \exists x.A$.

To state further derivable laws, we need the semantic counterpart of syntactic $!x$-freeness, given next.

Definition 8 ($!x$-free and stateless) C is *semantically* $!e$*-free* or simply $!e$*-free* when $[!e]C \equiv C$. C is α*-stateless* (resp. *stateless*) if C has no active dereferences of type α (resp. of any type).

Clearly 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 $C \supset [!x]C$. Further $\langle !x\rangle C \equiv C$ also characterises $!x$-freedom.

Remark 1 We usually regard $\equiv$ in Definition 8 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).

Example 3 ($!x$-freedom)

- By Axiom 6, any syntactically $!x$-free assertion is $!x$-free. Thus T and F are $!x$-free; so are $[!x]C$ and $\langle !x\rangle C$. However reverse implication does not hold, e.g. $!x = !x$ is easily $!x$-free but not syntactically so.
- 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.
- Assume $C \stackrel{\text{def}}{\equiv} !!e = 3 \wedge !e \neq x$ (where x is of type $\mathsf{Ref}(\mathsf{Nat})$). Then C is $!x$-free. Indeed, we can write $C \equiv \exists r.(!e = r \wedge !r = 3 \wedge r \neq x)$.

Proposition 3 (further derived laws) *In (1, 2, 3) below we assume C_1 to be $!x$-free.*

1. $[!x](C_1 \vee C_2) \equiv (C_1 \vee [!x]C_2)$.
2. $\langle !x\rangle (C_1 \wedge C_2) \equiv (C_1 \wedge \langle !x\rangle C_2)$.
3. $[!x](C_1 \supset C_2) \equiv (C_1 \supset [!x]C_2)$.
4. $[!x](C \wedge (C \supset C')) \supset [!x]C'$, *dually* $\langle !x\rangle C \supset \langle !x\rangle ((C \supset C') \supset C')$.
5. *If* $C \supset C'$ *is a tautology then* $[!x]C \supset [!x]C'$.
6. $[!x]C \supset C\{\!|e/!x|\!\}$, *dually* $C\{\!|e/!x|\!\} \supset \langle !x\rangle C$. *Further* $C\{\!|!x/!x|\!\} \equiv C$.
7. *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* $\exists C'.(C \equiv [!x]C')$.
8. *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.*
9. *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 Appendix B and $C\{\!|e/!x|\!\}$ is the result of substituting e for each free and active dereference of $!x$).*

Through (1, 2, 3) Prop. 3 strengthens our observation that "$!x$-freedom of C" acts as a substitute for "x not occurring in C" in standard quantification theory. Note that (3) 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.

Finally, as a simple application of content quantification, we calculate an example from the 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 3.

$$
\begin{array}{lrcl}
\text{(e1)} & \{C_1\}x\bullet y{=}z\{C\} \ \wedge\ \{C_2\}x\bullet y{=}z\{C\} & \equiv & \{C_1 \vee C_2\}\, x\bullet y = z\, \{C\} \\
\text{(e2)} & \{C\}x\bullet y{=}z\{C_1\} \ \wedge\ \{C\}x\bullet y{=}z\{C_2\} & \equiv & \{C\}\, x\bullet y = z\, \{C_1 \wedge C_2\} \\
\text{(e3)} & \{\exists w^{\alpha}.C\}\, x\bullet y = z\, \{C'^{-w}\} & \equiv & \forall w^{\alpha}.\{C\}\, x\bullet y = z\, \{C'\} \\
\text{(e4)} & \{C^{-w}\}\, x\bullet y = z\, \{\forall w^{\alpha}.C'\} & \equiv & \forall w^{\alpha}.\{C\}\, x\bullet y = z\, \{C'\} \\
\text{(e5)} & \{A \wedge C\}\, x\bullet y = z\, \{C'\} & \equiv & A \supset \{C\}\, x\bullet y = z\, \{C'\} \\
\text{(e6)} & \{C\}\, x\bullet y = z\, \{A^{-z} \supset C'\} & \supset & A \supset \{C\}\, x\bullet y = z\, \{C'\} \\
\text{(e7)} & \{C\}x\bullet y{=}z\{C'\} & \supset & \{C \wedge A\}x\bullet y{=}z\{C' \wedge A\} \\
\text{(e8)} & [!\tilde{w}]\,(C \supset C_0) \ \wedge\ \{C_0\}x\bullet y{=}z\{C_0'\} \ \wedge\ [!\tilde{w}]\,(C_0' \supset C') & \supset & \{C\}\, x\bullet y = z\, \{C'\}
\end{array}
$$

Fig. 3. Axioms for evaluation formulae.

4.2 *Axioms for Evaluation Formulae*

The set of axioms for evaluation formulae are given in Figure 3. We write C^{-x} to indicate $x \notin \mathsf{fv}(C)$. With the exception of (e8) all are unchanged from the corresponding axioms in (Honda *et al.*, 2005). We assume the following convention used throughout the paper.

Convention 2 *From now on $A, A', B, B', \ldots$ (possibly subscripts) range over* stateless formulae, *i.e. those formulae without any active dereferences (cf. Example 3 (3)), while $C, C', \ldots$ still range over general formulae.*

4.3 *Axioms for Data Types*

One of the central features of the present logic is its general treatment of data types. Here we allow reference types to appear anywhere in types, so that data structures can now be destructively updated in their parts. We incorporate the standard data types, such as unions, vectors and arrays. Below we consider how arrays can be treated. At the level of the programming language we add:

$$
\begin{array}{lccl}
\text{(types)} & \alpha & ::= & \ldots \mid \alpha[] \\
\text{(programs)} & M & ::= & \ldots \mid M[N]
\end{array}
$$

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:

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

where, in $e[e']$ has type $\mathsf{Ref}(\alpha)$, provided we can type e with an array type (say $\alpha[]$) and e' as Nat. The type of $\mathsf{size}(e)$, denoting size an array e, is Nat, whenever we can type e with an array type. $\mathsf{nil}^{\mathsf{Ref}(\alpha)}$, which denotes the null pointer and whose type we usually omit, is typed by $\mathsf{Ref}(\alpha)$. 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.

$$\forall i, j.\ (\ 0 \le i, j \lneq \mathsf{size}(x) \ \wedge \ i \neq j \quad \supset \quad x[i] \neq x[j]\ ) \tag{37}$$

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

$$(\mathsf{size}(x) = \mathsf{size}(y) \ \wedge \ \forall i.\ (\ 0 \le i < \mathsf{size}(x) - 1 \ \supset \ x[i] = y[i]\ ) \quad \supset \quad x = y \tag{38}$$

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

$$x \neq y \quad \supset \quad \forall i, j.\ (\ 0 \le i < \mathsf{size}(x) - 1 \ \wedge \ 0 \le j < \mathsf{size}(y) - 1 \ \supset \ x[i] \neq y[j]\ ) \tag{39}$$

which says two distinct arrays never overlap (note that this axiom is not applicable to, for example, languages like C/C++ which employ a richer, and less safe, notion of array). Note that (39) is equivalent to:

$$\exists i, j.\ (\ 0 \le i < \mathsf{size}(x) - 1 \ \wedge \ 0 \le j < \mathsf{size}(y) - 1 \ \wedge \ x[i] = y[j]\ ) \quad \supset \quad x = y. \tag{40}$$

For those axioms which involve nil and err, out-of-bound errors are treated as:

$$i \ge \mathsf{size}(x) \quad \supset \quad x[i] = \mathsf{nil} \tag{41}$$

Further we stipulate:

$$!\mathsf{nil} \ = \ \mathsf{err} \quad \text{and} \quad \mathcal{E}(\mathsf{err}) \ = \ \mathsf{err} \tag{42}$$

where $\mathcal{E}[\cdot]$ is an arbitrary term context. The latter means err used as part of an expression always leads to err.

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, to obtain a compositional proof system for arrays, we add precisely one introduction rule (for a constant) and one elimination rule (for indexing). This modularity in introducing new data structures is one of the key features of the present reasoning framework.

5 Logic (3): Judgements and Proof Rules

This section presents and discusses judgements and proof rules for total correctness.

5.1 Judgements and their Semantics

Following Hoare (Hoare, 1969), a judgement in the present program logic for total correctness 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*.[2] 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 well-typed judgements. Following Convention 1 (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 9 (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 4 *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'.

Definition 10 (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 $(\Gamma';\Delta')\cdot u : \alpha \vdash C'$, if $(\xi,\sigma) \models C$ then $(M\xi,\ \sigma) \Downarrow (V,\sigma')$ such that $(\xi \cdot u : V,\ \sigma') \models C'$. Here $(\Gamma';\Delta')\cdot u : \alpha$ means $\Gamma'\cdot u : \alpha;\Delta'$ whenever α is not a reference type, otherwise it stands for $\Gamma';(\Delta'\cdot u : \alpha)$.

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

---

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

5.2 Proof 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. Their shape is unchanged from the proof rules for the sublanguage without aliasing except for a minimal refinement of the rule for assignment, which now uses $\{\!|e'/!e|\!\}$ instead of syntactic substitution $[e'/!e]$ (cf. Section 3.3) and an adaptation to our generalised syntax in dereference and assignment. 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.

Recall 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 different from $C^{\text{-}!\tilde{x}}$); and $A, A', B, B', \ldots$ range over stateless formulae as defined in Convention 2.

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 Appendix B.

The compositional proof rules of the program logic are given in Figure 4. $[\mathit{Op}]$ is a general rule for first-order operators, and subsumes $[\mathit{Const}]$ when arity is zero. We illustrate the two new rules for imperative constructs, $[\mathit{Deref}]$ and $[\mathit{Assign}]$ in the following.

[*Deref*]. The rule $[\mathit{Deref}]$ says that:

> *If, assuming a precondition C, we wish to derive the postcondition C' for the program $!M$ (whose result we name u), then we should be able to derive from C the same thing about M (named m), except that we substitute $!m$ for u in C'.*

To understand this rule, we may start from the following simpler version.

$$[\mathit{Deref\text{-}Orig}]\ \frac{-}{\{C[!x/u]\}\ !x :_u \{C\}} \tag{43}$$

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

$$\frac{-}{\{\mathit{Even}(!x)\}\ !x :_u\ \{\mathit{Even}(u)\}} \tag{44}$$

which is also sound in the present target language and logic. $[\mathit{Deref}]$ generalises $[\mathit{Deref\text{-}Orig}]$ 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 $[\mathit{Cons}]$, the well-know consequence rule, cf. Figure 7

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

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

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

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

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

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

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

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

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

Fig. 4. Compositional Proof Rules.

below).

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:

$$\{C \wedge !x = 1\}\ !(fx) :_u \{u = 2 \ \wedge\ !x = 2\} \tag{45}$$

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-Orig*] is more efficient when a single variable is dereferenced, which may be frequent in practice.

[*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 5 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:

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

The rule may be understood by contrasting it with the corresponding rule for the sublanguage without aliasing. There the assignment rule reads:

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

There are two differences between this original rule and [*Assign*] in Figure 4. First, [*AssignOrig*] only allows a variable as the left-value, while [*Assign*] allows an arbitrary program. Second, the original rule uses syntactic substitution, while the present system uses the logical counterpart (cf. Section 3.3). 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.

$$\{x = y\ \wedge\ \mathit{Even}(!!y)\}\ !x := !!y + 1\ \{\mathit{Odd}(!!x)\ \wedge\ \mathit{Odd}(!!y)\} \tag{46}$$

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.

As already mentioned, structural rules manipulate formulae only. A well-known example of a structural rules is:

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

Section 7 present located proof rules, which are a derivable generalisation from which the original structural rules can easily be recovered.

6 Soundness and Elimination of Content Quantification

In this section we present some of the basic technical results about the proposed logic, including soundness of the proof rules and axioms, and showing that content quantification can be eliminated.

6.1 Elimination of Content Quantification

Using the axioms for content quantification introduced in Section 4, 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 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 inter-definable.

- 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 result also gives a basis for generalising content quantification, as we do in Section 7.4.
- The elimination procedure only uses the axioms for content quantifications discussed in Section 4.1 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 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 conciseness of the resulting formulae.
3. By the self-dual nature of logical substitutions ($C\{\!\!\{e'/!e\}\!\!\} \equiv C\overline{\{\!\!\{e'/!e\}\!\!\}}$, cf. Prop. 2.9), 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 4 *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 $e \stackrel{\text{def}}{=} x$ because each content quantification $[!e]C$ can be represented as $\exists x.(x = e/\ [!x]C)$, and likewise for existential content quantification. Let m

be fresh below.

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

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

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

hence done. □

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* $\Omega \in \{\forall,\exists\}$.

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

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

Proof

It suffices to prove the cases of $\star = \wedge$ and $\Omega = \forall$ 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 $C_0 \overset{\text{def}}{=} C$. 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:

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

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 $C' \overset{\text{def}}{=} C_m$. 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.

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

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 $C\{\!|m/!x|\!\} \equiv \langle !x\rangle\,(C \wedge !x = m)$:

$$\begin{aligned}
\langle !x\rangle C \quad &\equiv \quad \exists m.C\{\!|m/!x|\!\} && \text{(Lem.4)}\\
&\equiv \quad \exists m.(\exists\tilde{r}\tilde{c}.(\,(\wedge_i !r_i = c_i) \;\wedge\; C'))\{\!|m/!x|\!\} && \text{(Lem.2)}\\
&\equiv \quad \exists m.(\exists\tilde{r}\tilde{c}.(\,(\wedge_i !r_i = c_i)\{\!|m/!x|\!\} \;\wedge\; C')) && \text{(Lem.1)}\\
&\equiv \quad \exists m.(\exists\tilde{r}\tilde{c}.(\,(\wedge_i [\![(!r_i = c_i)\{\!|m/!x|\!\}]\!]) \;\wedge\; C'))
\end{aligned}$$

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

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

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

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

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 know that content quantifications and reference name (in)equations are mutually representable.

6.2 Soundness

In this subsection we present a key result about our logic, soundness of axioms and proof rules. This result 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 2 *(soundness) If* $\vdash \{C\}\, M :_u \{C'\}$ *then* $\models \{C\}\, M :_u \{C'\}$.

A similar correctness result holds for all axioms.

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

The straightforward proofs for both theorems can be found in Appendix D.

7 Located Assertions and Reasoning

7.1 Motivation and Syntax

This section formally introduces a located evaluation formula as a derived construct with associated axioms and rules that together facilitate convenient delineation of computational effects. Locations in our sense have been used before, in the context of object-oriented languages, and are sometimes called "modified clauses" (Wing, 1987; Müller *et al.*, 2003). Our approach is novel in the following two points: first, the set of locations that a program can modify is specified entirely within the logic, without appealing to external formalism, hence the entire logical apparatus is available for specification and reasoning about effects. Second, and relatedly, this form of specification can be combined with content quantification for a powerful generalisation of the standard Invariance Rule.

We motivate our approach with an example. 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:

$$\frac{\{C_1\}\, M\, \{C_1'\} \quad \{C_2\}\, N\, \{C_2'\}}{\{C_1 \wedge C_2\}\, M;N\, \{C_1' \wedge C_2'\}} \tag{47}$$

But this is unsound. The execution of M might invalidate assumptions inscribed in C_2. Similarly, running N may destroy guarantees made by C_1'. However, if we 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, (47) would in fact be sound. With content quantification, this is easily expressed: assume all of M's effects were in $\tilde{x}$, then $[!\tilde{x}]C_2$ would be $!\tilde{x}$-free, i.e. independent from M's effects. Similarly, with N's effects in $\tilde{y}$, $\langle !\tilde{y}\rangle C_1'$ is $!\tilde{y}$-free. If we use *located assertions* $\{C\}\ L\ \{C'\}\ @\,\tilde{z}$ as *syntactic sugar* to express that $\{C\}\ L\ \{C'\}$ holds and that all of L's effects are contained in $\tilde{z}$, then the following refinement of (47) is sound:

$$\frac{\{C_1\}\ M\ \{C_1'\}\ @\,\tilde{x} \quad \{C_2\}\ N\ \{C_2'\}\ @\,\tilde{y}}{\{C_1 \wedge [!\tilde{x}]C_2\}\ M;N\ \{C_2' \wedge \langle !\tilde{y}\rangle C_1'\}\ @\,\tilde{x}\tilde{y}} \tag{48}$$

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

The syntax of located evaluation formulae, or located assertions, takes the following form:

$$\{C\}e \bullet e' = x\{C'\}\ @\ \{e_1, e_2, ..., e_n\} \tag{49}$$

where each e_i should be of a reference type. $\{e_1, .., e_n\}$ is called *effect set*. We usually write an effect set as a sequence, i.e. we write the above formula as:

$$\{C\}e \bullet e' = x\{C'\}\ @\ e_1 e_2 ... e_n.$$

Expressions in effect sets are interpreted in the precondition: thus if e_i above includes a variable, it should already occur in the precondition C. Similarly we introduce the notation

$$\{C\}\ M :_u \{C'\} @ \{e_1, e_2, ..., e_n\} \tag{50}$$

where again if e_i includes a free variable then it should occur in C. We again usually write

$$\{C\}e \bullet e' = x\{C'\}\ @\ e_1 e_2 .. e_n.$$

In both located assertions and judgements, the following convention allows more flexible manipulation of effect sets:

Convention 3 *Whenever we use finite effect sets* $\{e_1, ..., e_n\}$ *in located assertions or located judgements, we assume that no* e_i *contains a dereference, except where intensionally demand otherwise.*

We shall later show that this convention does not lose generality.

Above, effect sets are restricted to finite numbers of references. We later discuss how to specify an arbitrary effects.

Example 5 (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 changes.

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

Fig. 5. Axioms for located evaluation formulae.

2. Let $M \overset{\text{def}}{=} \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1)$. Then we have

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

with $\Gamma \overset{\text{def}}{=} f : \mathsf{Nat} \Rightarrow \mathsf{Nat} \cdot x : \mathsf{Nat}$ and $\mathsf{Fact}(f) \overset{\text{def}}{=} \forall i \lnsim x.\{\mathsf{T}\}f \bullet i = i!\{\mathsf{T}\}@\emptyset$.

3. For the same M we have:

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

where $\mathsf{Fact}'(f) \overset{\text{def}}{=} \forall i \lnsim x.\{\mathsf{T}\}f \bullet i = i!\{\mathsf{T}\}@w$. 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.

7.2 Rules and Axioms for Located Assertions

Figure 5 lists axioms for located assertions, which refine the original axioms in Figure 3 and introduce 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 (28) and (29) on Page 15. (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 (28):

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

Using non-located reasoning, we can derive (29) from (28) as follows: we de-sugar this assertion and apply the laws $\forall x.C \supset C[e/x]$, $\forall \mathrm{X}.C \supset C[\alpha/\mathrm{X}]$ to obtain for a concrete $y^{\mathsf{Ref}(\mathsf{Nat})}$:

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

Now we use (e7) to add $[!x]Even(j)$ and get:

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

By the law of equality, the consequence rule and finally (e3) we infer:

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

Hence by (e8) we obtain:

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

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

Next we derive the same assertion using located reasoning. From obtain:

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

The following two implications hold:

- $(x \neq y \wedge Even(!y)) \supset [!x] Even(!y)$, an straightforward consequence of Prop. 1;
- $[!x] Even(!y) \supset Even(!y)$ by (CA2).

Hence [*Cons*] gives:

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

By (le1) and [*Cons*] we obtain:

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

which is more structured.

Finally (weak) and (thin) are reminiscent of the weakening rules and thinning rules in various type disciplines, hence the names.

Regarding reasoning with effect sets, valid located judgements are derivable with the proof rules for non-located judgements by translating located judgements to non-located ones. However, doing so would invalidate one of the key points for introducing locations, namely semi-automatic maintenance of effects. A more efficient method is to use compositional proof rules that are derivable in the original system, but are tailored for located judgements. Figures 6 (for compositional rules) and 7 (for structural rules) introduce such roof rules.

The compositional rules are entirely straightforward, closely following Figure 4, and listed in Figure 6 to illustrate some subtleties in key rules.

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 7, [*Weak*], [*Thinning*] and [*Invariance*] may deserve illustration. All others are straightforwardly derived from their non-located counterparts. [*Cons-Aux*] is easily derived by [*Rename*], [*Cons*], [$Aux_\exists$] and [*Invariance*]. Note also that the original (non-located) structural rules discussed in Section 5.2 are immediately obtained by simple removal of the effect set.

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

$$[\mathit{Op}]\ \frac{\{C\}\ M_1 :_{m_1} \{C_1\}\ @\ \tilde{e}_1 \quad \ldots \quad \{C_{n-1}\}\ M_n :_{m_n} \{C'[\mathsf{op}(m_1,\ldots,m_n)/u]\}\ @\ \tilde{e}_n}{\{C\}\ \mathtt{op}(M_1,\ldots,M_n) :_u \{C'\}\ @\ \tilde{e}_1 \ldots \tilde{e}_n}$$

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

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

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

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

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

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

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

Fig. 6. Proof rules with located judgements.

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

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

In Line 4, we have $(!x = 3)\{\!|n/!x|\!\} \equiv n = 3$ by Proposition 3 (9). Of course we can assign more complicated expressions. For example, we infer:

$$\begin{array}{lll}
1. & \{!x = 1\}x :_m \{m = x\ \wedge\ !x = 1\}\ @\ x & (m = x \wedge\ !x = 1) \supset m \in \{x\} \\
\hline
2. & \{m = x \wedge\ !x = 1\}!x + 1 :_n \{(!x = 2)\{\!|n/!x|\!\}\}\ @\ \emptyset & \\
\hline
3. & \{!x = 1\}x := !x + 1 :_n \{n = 2\}\ @\ x & \text{(1, 2, Assign)}
\end{array}$$

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

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

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

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

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

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

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

Fig. 7. Derivable structural rules for located judgements.

[*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) the contents 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 in x, it does not alter its content. As an example usage of [*Thinning*], we infer, noting $C\{\!|\,!x/!x\,|\!\} \equiv C$ (cf. Proposition 3 (6)):

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

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

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

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

Fig. 8. Derivable invariance rules for located judgements.

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

$$[\mathit{App\text{-}eoi}]\ \frac{\{C_1\}M :_m \{C_1'\} \,@\, \tilde{e}_1 \quad \{C_2\}\ N :_n \{\ C_2' \wedge \{\langle !\tilde{e}_2\rangle C_1' \wedge C_2'\} m \bullet n = u\{C'\} \,@\, \tilde{e}_3\} \,@\, \tilde{e}_2}{\{C_1 \wedge [!\tilde{e}_1] C_2\} MN :_u \{C'\} \,@\tilde{e}_1\tilde{e}_2\tilde{e}_3}$$

$$[\mathit{Assign\text{-}eoi}]\ \frac{\{C_1\}\ M :_m \{C_1'\} \,@\, \tilde{e}_1 \ \ \{C_2\}\ N :_n \{C_2'\} \,@\, \tilde{e}_2 \ \ (\langle !\tilde{e}_2\rangle C_1' \wedge C_2) \supset (C' \{\!\!\{ n/\,!m \}\!\!\} \wedge m \in \tilde{e})}{\{C_1 \wedge [!\tilde{e}_1] C_2\}\ M := N\ \{C'\}\ @\ \tilde{e}\tilde{e}_1\tilde{e}_2}$$

$$[\mathit{Pair\text{-}eoi}]\ \frac{\{C_1\}\ M_1 :_{m_1} \{C_1'\} \,@\, \tilde{e}_1 \quad \{C_2\}\ M_2 :_{m_2} \{C_2'\} \,@\, \tilde{e}_2 \quad \langle !\tilde{e}_2\rangle C_1 \wedge C_2 \ \supset\ C'[\langle m_1, m_2\rangle/u]}{\{C_1 \wedge [!e_1] C_2\}\ \langle M_1, M_2\rangle :_u \{C'\} \,@\, \tilde{e}_1\tilde{e}_2}$$

Fig. 9. Evaluation-order-independent 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 in Figure 8, the derivability of which is through Proposition 3 (6).

The rules [*InvUniv*] and [*InvEx*] say that the truth value of C_0 is independent from M's effects, in which case surely it is invariance. These two derivable rules are sometimes useful since they allow adding 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:

$$\frac{\vdash_{\text{Hoare}} \{C\}\ P\ \{C'\} \quad P \text{ does not touch variables in } C_0}{\vdash_{\text{Hoare}} \{C \wedge C_0\}\ P\ \{C' \wedge C_0\}} \tag{51}$$

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

$$\frac{\{C\}\ M :_u\ \{C'\}}{\{C \wedge A\}\ M :_u\ \{C' \wedge A\}} \tag{52}$$

This rule may also be compared to a similar rule studied by Reynolds, O'Hearn and others in (Reynolds, 2002; O'Hearn *et al.*, 2004): see Section 9.2 for a detailed comparison. Since a weakened stateless formula A in (52) is by definition $!x$-free for any x, [*Invariance*] above subsumes (52) (except we are now using located judgements). On the other hand, [*Invariance*] is justifiable using (52), cf. Section 4.2.

7.2.1 Evaluation Order Independence

The derived invariance rules can further be combined with compositional rules for located judgements in Figure 6 to obtain proof rules which are independent from any 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). This important for modular reasoning, cf. Example 8.3.

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 9 lists the EOI-refinement of (located) operator/application/assignment/pairing rules. These rules are all inferred from the original rule together with two the variants of the invariance rule, [*InvUniv*] and [*InvEx*].

At the onset of this section we had already illustrated the situation for sequential composition. Here is a suitable generalisation of (48):

$$[\mathit{Seq\text{-}I}]\ \frac{\{C_1\}\ M\ \{C_1'\}\ @\ \tilde{e}_1 \quad \{C_2\}\ N\ \{C_2'\}\ @\ \tilde{e}_2}{\{C_1 \wedge [!\tilde{e}_1]C_2\}\ M;N\ \{C_2' \wedge \langle !\tilde{e}_2\rangle C_1'\}\ @\ \tilde{e}_1\tilde{e}_2}$$

We emphasise once again 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. We continue with a simple example. Let M be the program $y := !y+1; z := !z+2$.

1	$\{!y = i+1\{\!\!\vert !y+1/!y \vert\!\!\}\}\ y := !y+1\ \{!y = i+1\}\ @\ y$	(AssignS)
2	$\{!y = i\}\ y := !y+1\ \{!y = i+1\}\ @\ y$	(Cons), 1
3	$\{!z = j+2\{\!\!\vert !z+1/!z \vert\!\!\}\}\ z := !z+2\ \{!z = j+2\}\ @\ z$	(AssignS)
4	$\{!z = j\}\ z := !z+2\ \{!z = j+2\}\ @\ z$	(Cons), 3
5	$\{!y = i \wedge [!y]\ !z = j\}\ M\ \{\langle !z\rangle\ !y = i+1 \wedge !z = j+2\}\ @\ yz$	(Seq-I), 2, 4
6	$([!y]\ !z = j) \supset (y \neq z \wedge !z = j)$	
7	$(\langle !z\rangle\ !y = i+1) \supset (y \neq z \supset !y = i+1)$	
8	$\{y \neq z \wedge !y = i \wedge !z = j\}\ M\ \{(y \neq z \supset !y = i+1) \wedge !z = j+2\}\ @\ yz$	(Cons), 5, 6, 7
9	$\{y \neq z \wedge !y = i \wedge !z = j\}\ M\ \{y \neq z \wedge (y \neq z \supset !y = i+1) \wedge !z = j+2\}\ @\ yz$	(Invariance), 8
10	$\{y \neq z \wedge !y = i \wedge !z = j\}\ M\ \{!y = i+1 \wedge !z = j+2\}\ @\ yz$	(Cons), 9

We used the following located version of [*AssignS*]:

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

Please note that this simple derivation allowed to remove the content quantifiers introduced by application of [*Seq-I*] and [*Cons*]. Interestingly, although [*Seq-I*] does not require separation between the two terms under composition, in the last derivation, a distinction between y and z is a natural consequence. In some cases this may be too restrictive. But it is easy to generalise [*Seq-I*]:

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

Now some of M's postcondition can be used in N's precondition. Clearly, both [*Seq*] and [*I-Seq*] are special cases up to some applications of [*Cons*]. As an example of using

[*Seq-I'*], assuming

$$\{!x=i\wedge\,!y=j\}\,M\,\{!x=i+2\wedge\,!y=j-3\}\,@\,xy \qquad \{!x=i+2\wedge\,!z=k\}\,N\,\{C\}\,@\,yz$$

are derived. Then we proceed:

1	$\{!x=i\wedge\,!y=j\}\,M\,\{!x=i+2\wedge\,!y=j-3\}\,@\,xy$	
2	$\{!x=i+2\wedge\,!z=k\}\,N\,\{C\}\,@\,xz$	
3	$\{!x=i\wedge\,!y=j\ \wedge\ [!xy]\,!z=k\}\,M;N\,\{\langle !yz\rangle\,!y=j-3\ \wedge\ C\}\,@\,xyz$	(Seq-I'), 1, 2
4	$([!xy]\,!z=k)\supset(x,y\neq z\wedge\,!z=k)$	
5	$\langle !xz\rangle\,(!y=j-3)\supset(x,z\neq y)\supset !y=j-3$	
6	$C_0\overset{\text{def}}{=}!x=i\wedge\,!y=j\wedge\,!z=k$	
7	$\{x,y\neq z\ \wedge\ C_0\}\,M;N\,\{(x,z\neq y\supset !y=j-3)\ \wedge\ C\}\,@\,xyz$	(Cons), 3, 4, 5
8	$\{x,y\neq z\ \wedge\ C_0\}\,M;N\,\{(x\neq y\supset !y=j-3)\ \wedge\ C\}\,@\,xyz$	(Cons, Invariance), 7

Similarly, one obtains EOI-rules for operators, application, assignment etc., as given in Figure 9. All EOI-rules are proved from the corresponding original rule together with Invariance rules [*InvUniv*] and [*InvEx*].

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*]).

Proposition 6 *$\{C\}\,M:_m\{C'\}\,@\,\tilde{g}$ is derivable in the proof rules for located judgements iff its translation is derivable in the proof rules for non-located judgements.*

7.2.2 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.

$$\begin{array}{lll}
\multicolumn{3}{l}{\models\{C\wedge z\neq\tilde{e}e'\wedge !z=i\wedge !e'=i'\}\,M:_u\{C'\wedge z\neq\tilde{e}e'\wedge !z=i\wedge !e'=i'\}}\\
\quad\Rightarrow & & \models\{C\wedge z\neq\tilde{e}\wedge z\neq e'\wedge !z=i\}\,M:_u\{C'\wedge z\neq\tilde{e}\wedge z\neq e'\wedge !z=i\}\ \wedge\\
 & & \models\{C\wedge z\neq\tilde{e}\wedge z=e'\wedge !z=i\}\,M:_u\{C'\wedge z\neq\tilde{e}\wedge z=e'\wedge !z=i\}\\
\quad\Rightarrow & & \models\{C\wedge z\neq\tilde{e}\wedge !z=i\}\,M:_u\{C'\wedge z\neq\tilde{e}\wedge !z=i\}
\end{array}$$

Soundness of other located rules is as for the corresponding unlocated rules.

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. □

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

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

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

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

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

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

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

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

Fig. 10. Located proof rules for imperative idioms.

7.3 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 10 lists several located proof rules for this purpose. The initial four assignment rules are directly derivable from the general assignment rule in Figure 6. The next two rules for the one-branch conditional are also easily derivable from the general conditional rule in Figure 6. 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). We write A^b to mean that if there is a primary name in A, it must be b. Both rules are directly derivable from the original rules through the standard encoding. 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.

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

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

Fig. 11. Located proof rules for arrays.

Section 4.3 already introduced the array data type with a corresponding axiomatisation. Figure 11 presents the located version of the proof rules for arrays. $[\mathit{Array}]$, together with the axioms introduced in Section 4.3 is all we need to reason about arbitrary arrays and operations on them in imperative PCFv. This simplicity partly comes from treating arrays as a string of references, cf. (Apt, 1981). The second rule in Figure 11 is a derivable version of $[\mathit{Array}]$ for simple expressions which is often useful. Below we give the reading of $[\mathit{ArraySimple}]$.

> 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, $[\mathit{Array}]$ 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 $[\mathit{ArraySimple}]$ to reach the following rule:

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

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

$$C\{\!|\, e' \,/\, !a[e] \,|\!\} \quad \stackrel{\text{def}}{=} \quad \exists m.(\langle !a[e] \rangle (C \wedge !a[e] = m)\ \wedge\ m = e') \tag{53}$$

In the right-hand side of (53), 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 $[\mathit{AssignArray}]$ can be used when array names themselves can be aliased which is a common situation in systems programming.

7.4 Generalisation of Effects in Located Assertions

Let S be an intensionally defined set of the shape $\{y \mid C_0\}$, with y a fresh variable of type $\mathsf{Ref}(\mathsf{X})$. In the present paper, we always demand, given such a set, that either $C_0 \equiv \mathsf{F}$ (i.e. S is empty) or, if not, $\exists y.C_0 \equiv \mathsf{T}$. In this way C_0 only elaborates (constrains) y. The generalisation of located assertions can then be written:

$$\{C\}e\bullet e' = x\{C'\}\ @\ S$$

where S, interpreted in the precondition, becomes a potential set of references to be written. Note $\{C\}e\bullet e' = x\{C'\}\ @\ e_1..e_n$ can be rewritten as $\{C\}e\bullet e' = x\{C'\}\ @\ \{y \mid \vee_i\, y = e_i\}$.

In turn, a generalised located assertion $\{C\}e\bullet e' = x\{C'\}\ @\ \{y|C_0\}$ can be translated to the following non-located assertion:

$$\{C\}\ e\bullet e'\ =\ x\,\{C'\}\ \wedge\ \forall \mathrm{X}.\forall y^{\mathsf{Ref(X)}}.\forall i^{\mathrm{X}}.\{C\wedge\neg C_0\wedge !y=i\}\ e\bullet e'\ =\ x\,\{C'\wedge !y=i\} \quad (54)$$

Above we need the first conjunct when $C_0 \equiv \mathsf{T}$, in which case (54) becomes simply $\{C\}\ e\bullet e'\ =\ x\ \{C'\}$ itself (i.e. we do not delineate the range of the write effects). As the other extreme case, if $C_0 \equiv \mathsf{F}$, then (54) becomes $\{C\wedge !y=i\}\ e\bullet e'\ =\ x\,\{C'\wedge !y=i\}$ for fresh y and i, saying, as can be checked, the evaluation has no write effects ever. In this way, we can regard a generalised located assertion as a shorthand for the corresponding non-located assertion, and use the axioms for the latter for reasoning about the former.

Generalised located assertions allow compositional reasoning analogous to their finite counterpart when combined with the content quantification generalised accordingly. We define, with the same condition on C_0 as above:

$$[!\{x|C_0\}]C \stackrel{\mathrm{def}}{=} \forall x.(C_0 \supset [!x]C)$$

The assertion $[!\{x|C_0\}]C$ reads:

> Under any content of the references defined by C_0, the assertion C holds.

The left-hand side has this intended meaning since, by Theorem 1, $\forall x.(C_0 \supset [!x]C)$ says that each x satisfying C_0 is distinct from any dereferenced location in C, that is, all locations satisfying C_0 should be distinct from any dereferenced location in C, giving the intended meaning of $[!\{x|C_0\}]C$.

The reasoning method that uses generalised forms of located assertions/judgements and content quantifications directly comes from their translations to the original finite located assertions. As an example, we have the following invariance rule for generalised located judgements:

$$[\mathit{Invariance\text{-}Gen}]\ \frac{\{C\}\ M :_u \{C'\}\ @\ S}{\{C\ \wedge\ [!S]C_0\}\ M :_u \{C' \wedge C_0\}\ @\ S}$$

Above we do not mention S in the post-condition since it is interpreted in the precondition (to allow more freedom in their use, we can introduce variables which denote such sets, allowing one to write $x = S$ etc.).

As a concrete example, we show how we can use generalised locations for asserting and reasoning about recursively defined data types, which introduce potentially unbounded effects. The program:

$$\mathtt{addOne} \stackrel{\mathrm{def}}{=} \mu g.\lambda x.\mathtt{case}\ !x\ \mathtt{of}\ \{\mathsf{nil} \triangleright () \mid a::y \triangleright (a := !a+1; g\,y)\}$$

modifies the content of every cell reachable from its argument. Hence, naming this program as f, we can derive the following (rather weak) assertion:

$$\{\exists l'l''.(!l = a::l' \wedge !l' = b::l'' \wedge !l'' = \mathsf{nil})\} f \bullet l \{\mathsf{T}\}\ @\ ab \quad (55)$$

if we know the list l is of length 2. However if we do *not* know the length of l, then we need to use generalised located assertions.

$$\mathsf{addoneSpec} \quad \stackrel{\text{def}}{=} \quad \forall l^{\mathsf{List}\alpha}.\{\mathsf{acyclic}(l)\}f \bullet l\{\mathsf{T}\} @ \{a \mid \mathsf{reach}(l,a)\} \tag{56}$$

Above we use the predicates characterised by the following axioms:

$$\begin{array}{rcl} \mathsf{path}(l,n,l') & \equiv & (n{=}0 \supset l{=}l') \wedge (n{>}0 \supset \exists l''a.(l{=}a{::}l'' \wedge \mathsf{path}(!l'',n{-}1,l'))). \\ \mathsf{acyclic}(l) & \equiv & \forall i,j.(i \neq j \supset \mathsf{path}(l,i,l') \supset \mathsf{path}(l,j,l'') \supset l' \neq l'') \\ \mathsf{reach}(l,a) & \equiv & \exists l',i.(\mathsf{path}(l,i,l') \ \wedge \ a = \pi_1(l')) \end{array}$$

Our target judgement is:

$$\{\mathsf{T}\}\ \mathtt{addOne}\ \{\mathsf{addoneSpec}\}, \tag{57}$$

To derive (57), we use induction on the length of the acyclic list. We only show the intermediate judgements for the case statement immediately before the abstraction, which we write M and name it as m. We use the following assertion for brevity:

$$\mathsf{main}(f,x) \stackrel{\text{def}}{=} \{\mathsf{acyclic}(x)\}f \bullet x\{\mathsf{T}\}@\{a \,|\, \mathsf{reach}(x,a)\}$$

Then the judgement for M is given as:

$$\{\mathsf{len}(x,i) \ \wedge \ \forall y.(\mathsf{len}(y,j) \wedge i \not\leq j \ \supset \ \mathsf{main}(g,y))\}\, M\, \{\mathsf{T}\}@\{a \mid \mathsf{reach}(x,a)\} \tag{58}$$

where $\mathsf{len}(x,i)$ asserts the length of an acyclic list x is i, which is easily definable. The judgement (58) itself is proved using such facts as a tail of an acyclic list is again acyclic and its length is strictly less than that of the original list. Once (58) is given, we apply the proof rules for abstraction and recursion to obtain the required assertion (57).

Finally we show the use of [*Invariance-Gen*], using its counterpart at the assertion level. If we know C_0 only asserts on (say) a list disjoint from x, then we can derive, from addoneSpec:

$$\forall l^{\mathsf{List}\alpha}.\{\mathsf{acyclic}(l) \wedge [!\{a \,|\, \mathsf{reach}(l,a)\}]C_0\}f \bullet l\{C_0\} @ \{a \mid \mathsf{reach}(l,a)\}$$

by adding the invariant assertion to the pre/post conditions.

8 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 3.3. We conclude our exhibition of the logic's reasoning abilities by proving the correctness of higher-order, generic Quicksort.

8.1 Questionable Double (1): Direct Reasoning

In Section 3.4 we introduced the "Questionable Double", a program behaving differently under different distinctions. Let us reproduce the program.

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

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 argument references.

$$\{\mathsf{T}\}\ \mathsf{double?} :_u \{\ \forall x,y.\{x \neq y \wedge\ !x = i \wedge\ !y = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge\ !y = 2j\}\ \} \qquad (59)$$

To infer the judgement (59), we use the following two implications.

$$x \neq y \wedge\ !x = i \wedge\ !y = j \quad \supset \quad (x \neq y \wedge\ !x = 2i \wedge\ !y = j)\{\!|\, !x+!x/!x \,|\!\} \qquad (60)$$

$$x \neq y \wedge\ !x = 2i \wedge\ !y = j \quad \supset \quad (!x=2i \wedge\ !y=2j)\{\!|\, !y+!y/!y \,|\!\} \qquad (61)$$

We first establish (60) and (61). For the former:

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

The reasoning for (61) 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.	$x \neq y \wedge\ !x = i \wedge\ !y = j \quad \supset \quad (x \neq y \wedge\ !x = 2i \wedge\ !y = j)\{\!|\, !x+!x/!x \,|\!\}$	(60)
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.	$x \neq y \wedge\ !x = 2i \wedge\ !y = j \quad \supset \quad (!x=2i \wedge !y=2j)\{\!|\, !y+!y/!y \,|\!\}$	(61)
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.	$\{\mathsf{T}\}\ \mathsf{double?} :_u \{\ \forall x,y.\{x \neq y \wedge\ !x = i \wedge\ !y = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge\ !y = 2j\}\ \}$	(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.

$$\mathsf{double!} \stackrel{\text{def}}{=} \lambda(x,y).(\mathtt{if}\ x \neq y\ \mathtt{then}\ x := !x+!x; y := !y+!y\ \mathtt{else}\ x := !x+!x) \qquad (62)$$

We now infer:

$$\{\mathsf{T}\}\ \mathsf{double!} :_u \{\ \forall x,y.\{!x = i \wedge\ !x = j\}\, u \bullet (x,y)\, \{!x = 2i \wedge\ !x = 2j\}\ \} \qquad (63)$$

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

We omit detailing the calculation for Line 8.

8.2 Questionable Double (2): Located Reasoning

We have seen, in Section 3.4, 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 10.

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

8.3 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):

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

Using this predicate we wish to establish:

$$\{\mathsf{T}\}\ \mathtt{swap} :_u \{\mathsf{Swap}(u)\}\ @\ \emptyset. \tag{64}$$

Located Reasoning. The semantic independence of `swap` is fully exploited using [*Seq-I*]. Let $A \overset{\text{def}}{=} x = y \supset i = j$ 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. 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. $\{!x = i \wedge !y = j\}\ \texttt{let}\ z = !x\ \texttt{in}\ (x := !y\ ;\ y := z)\ \ \{!x = j \wedge !y = i\}\ @xy$ (5, 6, Let)
8. $\{\mathsf{T}\}\ \texttt{swap} :_u \{\mathsf{Swap}(u)\}\ @\emptyset$ (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.

$$\begin{array}{lll}
\multicolumn{2}{l}{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}} & \\
\equiv & (!x = j)\{\!|z/!y|\!\}\{\!|!y/!x|\!\} \wedge (!y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\} & \text{(Pro. 4 (2))} \\
\equiv & ((x = y \supset z = j)\ \wedge\ (x \neq y \supset !x = j))\{\!|!y/!x|\!\}\ \wedge\ (z = i)\{\!|!y/!x|\!\} & (S1) \\
\equiv & (x = y \supset z = j)\{\!|!y/!x|\!\}\ \wedge\ (x \neq y \supset !x = j)\{\!|!y/!x|\!\}\ \wedge\ (z = i)\{\!|!y/!x|\!\} & \text{(Pro. 4 (2))} \\
\equiv & (x = y \supset z = j)\ \wedge\ (x \neq y \supset !x = j\{\!|!y/!x|\!\})\ \wedge\ z = i & \text{(Pro. 3)} \\
\equiv & (x = y \supset z = j)\ \wedge\ (x \neq y \supset !y = j)\ \wedge\ z = i & \text{(S1)} \\
\subset & !x = i\ \wedge\ !y = j\ \wedge\ z = i &
\end{array}$$

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.

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

Fig. 12. Quicksort with a comparison procedure as a parameter.

While the traditional reasoning gives a slightly shorter derivation at the level of proof rules, 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*].

8.4 Circular References

We next show the reasoning for $x := !!x$, the example, appearing in Section 3, that uses circular data structures. Reproducing the assertion in Section 3, we wish to prove the following judgement.

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

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

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

From here it is easy to get:

$$\begin{array}{ll} 1.\ (!x = y \wedge !y = x) \supset ((!x = x)\{\!|!!x/!x|\!\}) & \\ \hline 2.\ \{(!x = x)\{\!|!!x/!x|\!\}\}\ x := !!x\ \{!x = x\}\,@\,x & \text{(AssignVar)} \\ \hline 3.\ \{!x = y \wedge !y = x\}\ x := !!x\ \{!x = x\}\,@\,x & \text{(1, 2, Cons)} \end{array}$$

The next assertion, also already discussed in Section 3, can similarly easily be derived.

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

8.5 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

```
λ(a,c,l,r)
        let pv = !a[r] in
              p := l;
              i := l;
              while !i < r
                        if c(!a[!i], pv) then
                                swap( a[!p], a[!i] )
                                p := !p + 1
                        i := !i + 1
              swap(a[r], a[!p]);
              !p
```

Fig. 13. Partitioning algorithm.

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 12 and 13 present the code, assuming a generic swapping procedure like that from Section 36 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). In these programs we omit type annotations for variables, the main ones of which (for both programs) are:

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

$\mathrm{X}[\,]$ 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.

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 12, 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).

$$\{\mathsf{T}\}\ \mathtt{qsort} :_u \{\forall \mathrm{X}.\mathsf{Qsort}(u)\}\ @\,\emptyset. \tag{65}$$

where we set, omitting types:

$$\mathsf{Qsort}(u) \quad \stackrel{\text{def}}{=} \quad \forall abclr. \begin{pmatrix} \{\mathsf{Eq}(ablr) \wedge \mathsf{Order}(c)\} \\ \quad u \bullet (a, c, l, r) \\ \{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\}\,@a[l...r]ip \end{pmatrix} \tag{66}$$

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 (66) 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

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

  then we set

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

- $\mathsf{Eq}(ablr)$ 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");
  - — $\forall xy.(x \ne y\ \supset\ (c \bullet (x,y) \searrow \mathsf{T} \vee c \bullet (y,x) \searrow \mathsf{T}))$ ("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:

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

The result of permuting n times is then given by:

$$\begin{array}{rcl}\mathsf{Perm}^{(0)}(ablr) & \overset{\text{def}}{=} & \mathsf{Eq}(ablr)\\ \mathsf{Perm}^{(n+1)}(ablr) & \overset{\text{def}}{=} & \exists a'.(\mathsf{Perm}^{(n)}(aa'lr)\ \wedge\ \mathsf{SPerm}(a'blr))\end{array}$$

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: $\mathsf{Sorted}(aclr) \overset{\text{def}}{=} \forall i,j.(l \le i < j \le r\ \supset\ c \bullet (!a[i], !a[j]) \searrow \mathsf{T})$.

So $\mathsf{Qsort}(u)$ in (66) 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 (65). 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 12), 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:

$$\{C_1\}\ \ \mathsf{q}(a,c,l,p'-1)\ ;\ \mathsf{q}(a,c,p'+1,r)\ \ \{C_1'\}\ @\,a[l...r]ip \tag{67}$$

where

$$\begin{array}{rcl}C_1 & \overset{\text{def}}{=} & \mathsf{Perm}(ablr)\ \wedge\ \mathsf{Parted}(aclrp')\ \wedge\ \mathsf{Order}(c)\ \wedge\ \forall j{<}k.\mathsf{QsortBounded}(qj)\ \wedge\ r-l{\le}k\\ C_1' & \overset{\text{def}}{=} & \mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr).\end{array}$$

Two newly introduced predicates are illustrated below.

$\mathsf{QsortBounded}(uj)$ with j of Nat type is used as an inductive hypothesis for recursion. It is the same as $\mathsf{Qsort}(u)$, given in (66), Page 50, except that it only works for a range no more than j and that it replaces "$\mathsf{Eq}(ablr)$" in the precondition of (66) with "$\mathsf{Perm}(ablr)$", which is necessary for the induction to go through. Formally: $\mathsf{QsortBounded}(uj)$ is

$$\forall abclr.0 \le r-l \le j\ \supset\ \begin{pmatrix}\{\mathsf{Perm}(ablr) \wedge \mathsf{Order}(c)\}\\ \quad u \bullet (a,c,l,r)\\ \{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\}\,@a[l...r]ip\end{pmatrix}$$

$\mathsf{Parted}(aclrk)$ 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 $\mathsf{Parted}(aclrk)$ is given as:

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

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. $\{C_2\}\ \mathtt{q(a,c,l,p'-1)}\ \{C_2'\}\ @\ \tilde{e_2}$

R.2. $\{C_3\}\ \mathtt{q(a,c,p'+1,r)}\ \{C_3'\}\ @\ \tilde{e_3}$

R.3. $\{C_2 \wedge [!\tilde{e_2}]C_3\}\ \mathtt{q(a,c,l,p'-1)\ ;\ q(a,c,p'+1,r)}\ \{\langle !\tilde{e_3}\rangle C_2' \wedge C_3'\}\ @\ \tilde{e_2}\tilde{e_3}$

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. $\{C_1\}\ \mathtt{q(a,c,l,p'-1)\ ;\ q(a,c,p'+1,r)}\ \{C_1'\}\ @\ \tilde{e_2}\tilde{e_3}$ (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*] (Kleyman's Rule) from Figure 7. 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.

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

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 is by first-order logic while the latter holds since $C_3^{-!\tilde{e}_2}$. 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 $\tilde{e} \overset{\text{def}}{=} \tilde{e}_2\tilde{e}_3$ 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 3.

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

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.

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

$\mathsf{Leq}(aclrv)$ (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: $C_{inloop} \stackrel{\mathrm{def}}{=} \mathsf{Invar} \wedge !i < r \wedge r - !i = j$. The following assertions specify two cases of the conditional branch.

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

We now present the derivation for the `if`-branch of the loop, where the comparison procedure (received as 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.

$$\begin{array}{lr}
(\mathsf{Invar} \wedge r - !i > 0 \wedge r - !i = j) \supset \left(\begin{array}{l} \{\mathsf{Invar} \wedge r - !i > 0 \wedge r - !i = j\} \\ \qquad c \bullet (!a[!i], pv) = z \\ \{c \bullet (!a[!i], pv) \searrow z \wedge \mathsf{Invar} \wedge r - !i > 0 \wedge r - !i = j\} @ \emptyset \end{array} \right) & \\
\hline
\begin{array}{l} \{\mathsf{Invar} \wedge r - !i > 0 \wedge r - !i = j\} \\ \qquad c(!a[!i], pv) \ :_z \\ \{c \bullet (!a[!i], pv) \searrow z \wedge \mathsf{Invar} \wedge r - !i > 0 \wedge r - !i = j\} @ \emptyset \end{array} & \text{(AppSimple)} \\
\hline
\{C_{then}\} \ \mathtt{ifbody} \ \{\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r - !i \leq j)\} \ @ a[l...r-1]ip & \text{(omitted)} \\
\hline
C_{\neg then} \supset (\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r - !i \leq j) & \\
\hline
\begin{array}{l} \{C_{inloop}\} \mathtt{if} \ c(!a[!i], pv) \ \mathtt{then} \ \mathtt{ifbody} \\ \{\mathsf{Invar}\{\!|!i+1/!i|\!\} \wedge r - !i \leq j\} @ a[l...r-1]pi \end{array} & \text{(IfThen)}
\end{array}$$

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:

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

Reasoning (3): Polymorphism. We are now ready to derive the whole specification of Quicksort (65). 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 $\mathsf{ftv}(C)$.

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

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

We now present the derivation. For brevity we use the following abbreviations: $C_\star \overset{\mathrm{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)$, $B' \overset{\mathrm{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c) \wedge \forall j < k.\mathsf{QsortBounded}(qj) \wedge r - l \leq k$, and $B \overset{\mathrm{def}}{=} B' \wedge l < r$. We also write $\mathtt{qsort}'$ for $\mathtt{qsort}$ in Page 48 without the first line (i.e. without μ/λ-abstractions), M for $\mathsf{q}(a,c,l,p'-1)\ ;\ \mathsf{q}(a,c,p'+1,r)$.

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

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

9 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; 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.

9.1 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 $\mathtt{new}\, x := M \,\mathtt{in}\, N$ 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: this is so-called stack-allocated variables). 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.

$$\frac{\{C^{\text{-}x}\} N :_n \{C_0\} \quad \{\exists n.(!x = n \wedge [!x]C_0)\} M^{\Gamma;\Delta\cdot x:\mathsf{Ref}(\alpha);\beta} :_m \{C'^{\text{-}x}\}}{\{C\}\, \mathtt{new}\, x := N \,\mathtt{in}\, M^{\Gamma;\Delta;\beta} :_u \{C'\}} \tag{68}$$

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. Note that the universal content quantification is naturally introduced at the time of variable declaration: this makes it possible to reason about the body M assuming x is disjoint from all other references (in fact we could have used this rule in Quicksort in Section 8.5, by localising the variable i).

It is notable that the rule above also allows us to treat the standard parameter passing mechanism in procedural languages such as C and Java through the following simple translation: a procedure definition "`f(x,y)` {...}" is transformed into

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

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 (68).

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. $\lambda n.\mathtt{new}\, x := n \,\mathtt{in}\, x$. 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 will be explored in our forthcoming report (Yoshida *et al.*, 2006).

9.2 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 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 (the corresponding precise axiomatic treatment is first given by Bornat in (Bornat, 2000)). Morris's conditional update and its calculation correspond to the calculation for logical substitution in the present logic.

Morris's approach is equivalent to Cartwright and Oppen's in the sense that formulae with conditional expressions are easily decomposed into those without, 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.

$$\frac{\{C\}\ P\ \{C'\} \quad \mathsf{fv}(C_0) \cap \mathsf{modify}(P) = \emptyset}{\{C * C_0\}\ P\ \{C' * C_0\}} \tag{69}$$

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 that 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 (Hennessy & Milner, 1985) and Hoare logic (Honda *et al.*, 2006b). 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 5, which plays a role similar to (69). 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 (69) and [*Assign, Inv, Seq, Cons*].

$$\begin{array}{l}
\{x \mapsto -\}\ x := 2\ \{x \mapsto 2\} \\ \hline
\{y \mapsto -\ \wedge\ z \mapsto i\}\ y := !z\ \{y \mapsto i\ \wedge\ z \mapsto i\} \\ \hline
\{x \mapsto -\ *\ (y \mapsto -\ \wedge\ z \mapsto -)\}\ x := 2; y := !z\ \{x \mapsto 2\ *\ \exists i.(y \mapsto i\ \wedge\ z \mapsto i)\}
\end{array}$$

For the same program, a direct application of our invariance rule [*Seq-I*] gives:

$$\begin{array}{lr}
\{\mathsf{T}\}\ x := 2\ \{!x = 2\}\ @\, x & (\mathsf{Assign}) \\ \hline
\{\mathsf{T}\}\ y := !z\ \{!y = !z\}\ @\, y & (\mathsf{Assign}) \\ \hline
\{\mathsf{T}\}\ x := 2; y := !z\ \{\langle !y \rangle\, !x = 2 \wedge\ !y = !z\}\ @\, xy & (\mathsf{Seq\text{-}I})
\end{array}$$

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 $\forall \tilde{x}.([!\tilde{e}_2]\, C_1' \wedge [!\tilde{e}_1]\, C_2')$ where $\tilde{e}_i$ exhausts active dereferences of C_i' and $\forall \tilde{x}.(C_1' * C_2')$ is obtained from $C_1 * C_2$ by moving all quantifiers outside, 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.

$$\frac{C_1\ !\tilde{x}\text{-free} \quad \{C_0\}\,N\,\{C_0'*C_1\}\,@\,\tilde{x}\tilde{y} \quad \{C^{\text{-}f}\wedge\{C_0\}\,f\,\{C_0'\}@\,\tilde{x}\}\,M:_u\ \{C'\}@\,\tilde{x}}{\{C\wedge C_1\}\ \mathtt{let}\ f=\lambda().N\ \mathtt{in}\ M:_u\ \{C'\wedge C_1\}\,@\,\tilde{x}\tilde{y}} \qquad (70)$$

Here f should occur 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. The condition on f above 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), (70) 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 procedure labels not adhering to the above condition.

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 is an interesting subjects for further study.

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 `putchar(buff,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* 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 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 logical reasoning even with strong update.

Nanevski, Morrisett and Birkedal (Nanevski *et al.*, 2006) study *Hoare Type Theory* (HTT) which combines dependent types and Hoare triples with anchors based on a monadic understanding of computation. The aim of HTT is to provide an effective general validation framework that unifies standard static checking techniques (in particular type inference and type checking) with logical verifications. Their system emphasises clean separation between program parts which allow effective validation and parts which involve assertions (represented as types). The assertion language uses an untyped store, and, through the use of polymorphism, can represent key idioms of separation logic. This allows validation of programs with strong updates, but local store is not treated. The interplay of the present assertion-based approach with HTT is an interesting topic for further study, especially regarding the integration of static analysis approaches to program verification and with their assertional counterparts.

Finally, we have recently shown (Honda *et al.*, 2006a) that the logics for pure higher-order functions and imperative ones without aliasing enjoy strong completeness properties, including standard relative completeness, and inductive derivability of a characteristic formula for each program. The method used in (Honda *et al.*, 2006a) however does not directly generalise to aliasing. We leave the question of how to do this open in the present paper.

Acknowledgements. We like to thank the JFP referees whose insightful comments and suggestions lead to a substantial improvement of this paper. We are also grateful for a discussion with Norbert Schirmer and Tobias Nipkow prompted our development of generalised located assertions.

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.

Enderton, Herbert B. (2001). *A Mathematical Introduction to Logic*. Academic Press.

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.

Honda, Kohei, Berger, Martin, & Yoshida, Nobuko. (2006a). Descriptive and relative completeness of logics for higher-order functions. *Pages 360–371 of:* Bugliesi, Michele, Preneel, Bart, Sassone, Vladimiro, & Wegener, Ingo (eds), *Icalp (2)*. Lecture Notes in Computer Science, vol. 4052. Springer.

Honda, Kohei, Berger, Martin, & Yoshida, Nobuko. (2006b). Descriptive and Relative Completeness of Logics for Higher-Order Functions. *33rd International Colloquium on Automata, Languages and Programming*.

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.

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.

Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17**(Aug.), 348–375.

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.

Müller, Peter, Poetzsch-Heffter, Arnd, & Leavens, Gary T. (2003). Modular specification of frame properties in JML. *Concurrency and computation: Practice and experience*, **15**, 117–154.

Nanevski, Alexandar, Morrisett, Greg, & Birkedal, Lars. (2006). Polymorphism and separation in hoare type theory. *Proc. ICFP*.

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.

Wing, Jeannette M. (1987). Writing Larch interface language specifications. *Acm trans. program. lang. syst.*, **9**(1), 1–24.

Yoshida, Nobuko, Honda, Kohei, & Berger, Martin. (2006). *Local Reasoning for Higher-Order Functions with Local State*. Draft.

A Language Details

A.1 Typing

The typing rules are standard (Pierce, 2002) and listed in Figure A 1, using sequents $\Gamma \vdash M : \alpha$, which say that M has type α under basis Γ.

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

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

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

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

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

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

Fig. A 1. Typing rules. A type α is *comparable* if it is in $\{\mathsf{Unit}, \mathsf{Bool}, \mathsf{Nat}, \mathsf{Ref}(\beta)\}$.

A.2 Dynamics

We list the rules that generate the reduction relation. We start with 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.

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

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

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

Finally the contextual rules are given as follows.

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

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.

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

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 $\mathsf{dom}(\Delta) = \mathsf{dom}(\sigma) = \mathsf{fl}(\sigma)$ and, moreover, the types of σ match Δ, i.e. for each $x \in \mathsf{dom}(\sigma)$ we have $\Delta \vdash \sigma(l) : \alpha$ iff $\Delta(l) = \mathsf{Ref}(\alpha)$. Note $\mathsf{dom}(\sigma) = \mathsf{fl}(\sigma)$ means locations 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 $M \stackrel{\text{def}}{=} !l := 3$ and $\sigma \stackrel{\text{def}}{=} \{l \mapsto l',\ l' \mapsto 2\}$, we have

$$l : \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat})),\ l' : \mathsf{Ref}(\mathsf{Nat}) \ \vdash\ (M,\sigma)$$

Note that $l : \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat})) \vdash M : \mathsf{Unit}$.

Proposition 7 (subject reduction) *Suppose* $\Delta \vdash M : \alpha$ *and* $\Delta \vdash (M,\sigma)$. *Then* $(M,\sigma) \longrightarrow (M',\sigma')$ *implies* $\Delta \vdash M' : \alpha$ *and* $\Delta \vdash (M',\sigma')$.

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.

B 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 11 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 $\mathcal{Q}y.C$ with $\mathcal{Q} \in \{\forall, \exists\}$ 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:

$$C \quad \stackrel{\text{def}}{=} \quad z=3 \quad \supset \quad [!y]z=3 \tag{B 1}$$

The assertion is a tautology (i.e. true in any model), saying: if z is 3, then whatever value a cell named y stores, z is still 3. However the following assertion, resulting from (B 1) when we apply the substitution $[!y/z]$ naively, is *not* a tautology (in fact it is unsatisfiable).

$$C[!y/z] \quad \stackrel{\text{def}}{=} \quad !y=3 \quad \supset \quad [!y]\ !y=3. \tag{B 2}$$

Note $!y$ is not free for z in C due to content quantification on $!y$. (B 2) says that, if the value currently stored in y is 3, then any value storeable in y coincides with 3, a sheer absurdity. Thus we should prohibit such substitution being applied to C.

In the standard quantification theory, we can always rename bound variables to avoid capture of names. In the present case, what we do is to use (standard) existential quantification to “flush out” all names in dangerous positions. As an example, take C in (B 1). To safely apply $[!y/z]$ to C, we transform C to the following formula, up to logical equivalence:

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

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

$$C'[!y/z] \ \stackrel{\text{def}}{=} \ \exists z'.(\,(!y=3 \quad \supset \quad [!y]\,z'=3) \ \wedge\ !y=z'\,) \tag{B 4}$$

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.

C Some Proofs for Propositions 1, 2 and 3

Proving Propositions 1 and 2 is straightforward. As an illustration we derive Prop. 1.2 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)

As second example derivation is that for Prop. 2.2:

$$\begin{array}{lr}
1.\ [!y]\,[!x]C \supset C & \text{(CA2)} \\ \hline
2.\ [!y]\,([!y]\,[!x]C \supset C) & \text{(CGen, 1)} \\ \hline
3.\ [!y]\,[!x]C \supset [!y]C & \text{(CA1, 2)} \\ \hline
4.\ [!x]\,([!y]\,[!x]C \supset [!y]C) & \text{(CGen, 3)} \\ \hline
5.\ [!y]\,[!x]C \supset [!x]\,[!y]C & \text{(CA1, 4)}
\end{array}$$

Next we derive Prop. 3.1: recall that C_1 is $!x$-free, i.e. $C_1 \equiv [!x]C_1 \equiv \langle !x \rangle C_1$.

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

For the reverse direction:

$$C_1 \ \vee\ [!x]C_2 \quad \equiv \quad [!x]C_1 \ \vee\ [!x]C_2 \quad \equiv \quad [!x]\,(C_1 \ \vee\ C_2)$$

In both cases we use the fact that $[!x]C_1$ and $\langle !x \rangle C_1$ are $!x$-free. Both universal and existential characterisations of $!x$-freedom are needed to obtain the desired logical equivalence. Prop. 3.2 and Prop. 3.3 follow easily from Prop. 3.1.

We continue with derivations for Prop. 3. For Prop. 3.4:

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

For Prop. 3.5, observing any tautology is $!x$-free:

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

For Prop. 3.6, the first statement:

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

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

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

For the other direction, again with m fresh:

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

Prop. 3.7 and Prop. 3.8 are easy and omitted. For Prop. 3.9:

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

D Soundness

The appendix presents proofs for Theorems 2 and 3.

We have already seen in Section 5 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 5 *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.

$$\begin{array}{ll} \multicolumn{2}{l}{(\xi,\ \sigma) \models C[x/u] \wedge\ \models \{C\}M_1 :_{m_1} \{C_1\} \wedge\ \models \{C_1\}M_2 :_{m_2} \{C_2[\mathsf{op}(m_1m_2)/u]\}} \\ \Rightarrow & (\xi\cdot m_1 : M_1,\ \sigma) \Downarrow (\xi\cdot m_1{:}V_1,\ \sigma_1) \wedge \\ & (\xi\cdot m_1{:}V_1\cdot m_2{:}M_2,\ \sigma_1) \Downarrow (\xi\cdot m_1{:}V_1\cdot m_2{:}V_2,\ \sigma') \models C_2 \ \wedge\ u = \mathsf{op}(m_1m_2) \\ \Rightarrow & (\xi\cdot u{:}\mathsf{op}(M_1M_2),\ \sigma) \Downarrow (\xi\cdot u{:}\mathsf{op}(V_1V_2),\ \sigma') \models C_2 \end{array}$$

The general n-ary case is similar.

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

$$\begin{array}{ll} \multicolumn{2}{l}{(\xi,\sigma) \models A} \\ \Rightarrow & \forall V.(\ (\xi\cdot x : V,\ \sigma) \models A \wedge C\ \supset\ (M\xi',\ \sigma) \Downarrow (\xi'\cdot m : W,\ \sigma') \models C'\ ) \\ \Rightarrow & \forall V.((\xi\cdot x : V,\ \sigma) \models A \wedge C\ \supset\ ((\lambda x.M)\xi V,\ \ \sigma) \Downarrow (\xi'\cdot m : W,\ \sigma') \models C') \\ \Rightarrow & (\xi\cdot u : (\lambda x.M)\xi,\ \sigma) \models \forall x.\{C\}u \bullet x = m\{C'\} \end{array}$$

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

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

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

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

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

Here, in the target of the first implication, i is either 1 or 2. Above we used the fact that closed boolean values are exhausted by `t` and `f`.

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

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

Above we used the fact that closed values of sum types are of the form $\mathtt{in}_i(V)$ with $i \in \{1,2\}$. Again, in the target of the first implication, i is either 1 or 2. Next we turn to the structural rules, given in their located variant in Figure 7. 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. [*Cons-Aux*] is derived from [*Cons-Aux*] is derived by [*Rename*], [*Cons*], [$Aux_\exists$] and [*Invariance*]. Finally [*Rename*] holds easily as all relevant operations on models and the reduction relation is closed under injective renaming. Hence we have established Theorem 2.

Next we establish Theorem 3. We begin with the axiomatisation of content quantification in Figure 2. We need a some preliminary facts.

Lemma 3 $\mathcal{M}[x \mapsto V] \leq_{x:\alpha} \mathcal{M}'$ *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. $\square$

Proposition 8
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) $[\![e']\!]_{\mathcal{M}[x\mapsto V]} = [\![e']\!]_{\mathcal{M}[x\mapsto W]} \stackrel{\text{def}}{=} l$. But $\mathcal{M} \models x \neq e_i$, hence $[\![x]\!]_{\mathcal{M}} \neq l$, hence with $\mathcal{M} = (\xi, \sigma)$:

$$\sigma[x \mapsto V](l) = \sigma(l) = \sigma[x \mapsto W](l).$$

But then

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

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

$$\begin{aligned}
\mathcal{M}[x \mapsto V] \models \forall y^{\alpha}.C &\equiv \forall \mathcal{M}'.(\mathcal{M}[x \mapsto V] \leq_{y:\alpha} \mathcal{M}' \supset \mathcal{M}' \models C) \\
&\equiv \forall \mathcal{M}'.((\exists \mathcal{M}''.\mathcal{M} \leq_{y:\alpha} \mathcal{M}'', \mathcal{M}''[x \mapsto V] = \mathcal{M}') \supset \mathcal{M}' \models C) && \text{(D 1)} \\
&\equiv \forall \mathcal{M}''.(\mathcal{M} \leq_{y:\alpha} \mathcal{M}'' \supset \mathcal{M}''[x \mapsto V] \models C) \\
&\equiv \forall \mathcal{M}''.(\mathcal{M} \leq_{y:\alpha} \mathcal{M}'' \supset \mathcal{M}''[x \mapsto W] \models C) && \text{(D 2)} \\
&\equiv \forall \mathcal{M}'.((\exists \mathcal{M}''.\mathcal{M} \leq_{y:\alpha} \mathcal{M}'', \mathcal{M}''[x \mapsto W] = \mathcal{M}') \supset \mathcal{M}' \models C) \\
&\equiv \forall \mathcal{M}'.(\mathcal{M}[x \mapsto W] \leq_{y:\alpha} \mathcal{M}' \supset \mathcal{M}' \models C) && \text{(D 3)} \\
&\equiv \mathcal{M}[x \mapsto W] \models \forall y^{\alpha}.C
\end{aligned}$$

Here (D 2) is by (IH) and (D 1, D 3) are by Lemma 3.

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 $C^{\text{-}!x}$. 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 4 *The axioms and the rule in Figure 2 are sound.*

Proof

For (CA1) we argue as follows

$$\begin{array}{rcll}
\mathcal{M} \models [!x]\,(C_1^{-!x} \supset C_2) & \equiv & \forall V.\mathcal{M}[x \mapsto V] \models (C_1 \supset C_2) & \\
& \equiv & \forall V.(\mathcal{M}[x \mapsto V] \models C_1 \supset \mathcal{M}[x \mapsto V] \models C_2) & \\
& \equiv & \mathcal{M} \models C_1 \supset \forall V.\mathcal{M}[x \mapsto V] \models C_2 & \text{(Prop. 8.3)} \\
& \equiv & \mathcal{M} \models C_1 \supset \mathcal{M} \models [!x]C_2 & \\
& \equiv & \mathcal{M} \models (C_1 \supset [!x]C_2) &
\end{array}$$

(CA2) has the following justification.

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

For (CA3) we derive

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

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 8.3. This concludes the proof for the axioms and the rule in Figure 2. □

Next is are the axioms for the evaluation formula in Figure 3.

Lemma 5 *All axioms in Figure 3 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. □

Lemmas 5 and 4 together verify Theorem 3.