A Logical Analysis of Aliasing in Imperative Higher-Order Functions

Martin Berger Kohei Honda
Queen Mary, University of London

Nobuko Yoshida
Imperial College London

Abstract

We present a compositional program logic for call-by-value imperative higher-order functions with general forms of aliasing, which can arise from the use of reference names as function parameters, return values, content of references and parts of data structures. The program logic extends our earlier logic for alias-free imperative higher-order functions with new modal operators which serve as building blocks for clean structural reasoning about programs and data structures in the presence of aliasing. This has been an open issue since the pioneering work by Cartwright-Oppen and Morris twenty-five years ago. We illustrate usage of the logic for description and reasoning through concrete examples including a higher-order polymorphic Quicksort. The logical status of the new operators is clarified by translating them into (in)equalities of reference names. The logic is observationally complete in the sense that two programs are observationally indistinguishable iff they satisfy the same set of assertions.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Assertions, logics of programs, specification techniques

General Terms Languages, Reliability, Security, Theory, Verification

Keywords Hoare-Logics, Modalities, Aliasing, Pointers, Typing, Functional Programming, π-Calculus

1. Introduction

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

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

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

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

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

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

where $\alpha = \mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat}))$. R receives a function f and two references x and y. Its behaviour is different depending on what it receives as f (for simplicity let's assume x and y store distinct references). If we pass a function $\lambda xy.()$ as f, then, after execution, $!x$ stores 3 and $!y$ stores 2. But if the standard swapping function $\texttt{swap} \stackrel{\text{def}}{=} \lambda ab.\texttt{let}\ c = !b\ \texttt{in}\ (b := !a; a := c)$ is passed, the content of x and y is swapped and $!x$ now stores 2 while $!y$ stores 3. Such interplay between higher-order procedures and aliasing is common in many non-trivial programs in ML, C and more recent typed and untyped low-level languages [1, 17, 38].

Hoare logic [20], developed on the basis of Floyd's assertion method [14], 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 [4, 11]: while various extensions have been studied, a general solution which extends the original method to treat aliasing, retaining its semantic basis [15, 21] and tractability, has not been known, not to speak of its combination with arbitrary imperative higher-order functions (our earlier work [24] 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 [9, 10, 33], the present paper introduces a simple and tractable compositional program logic for general aliasing and imperative higher-order functions. A central observation in [9, 10, 33] is that (in)equations over names, simple as they may seem, are expressive enough to describe general aliasing in first-order procedural languages, provided we distinguish between reference names (which we write x) and the corresponding content (which we write $!x$) in assertions. In particular, their work has shown that alias robust substitution, written $C\{\!|e/!x|\!\}$ in our notation, defined by:

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

(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 [9, 10]: Morris showed many non-trivial reasoning examples for data structures with destructive update, including the reasoning for Schorr-Waite algorithm [33].

The works by Cartwright-Oppen and Morris, remarkable as they are, still beg the question how to reason about programs with aliasing in a tractable way. The first issue is calculation of validity in assertions involving semantic substitutions. This is hardly practical because inductive decomposition of $\{\!|e/!x|\!\}$ into (in)equations has been the only syntactic tool available. As demonstrated through many examples by Morris [33] and, more recently, Bornat [7], this decomposition should be distributed to every part of a given formula even if that part is irrelevant to the state change under consideration, making reasoning extremely cumbersome. As one typical example, if we use the decomposition method for calculating the logical equivalence

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

for general C, with c being a constant, we need either meta-logical reasoning, induction on C, or an appeal to semantic means. Because such logical calculation is a key part of program proving (cf. [20]), practical usability of this approach becomes unclear. The second problem is the lack of structured reasoning principles for deriving precise description of extensional program behaviour with aliasing. This makes reasoning hard, because properties of complex programs often depend crucially on how sub-programs interact through shared, possibly aliased references. Finally, the logics in [9, 10, 33] and their successors do not offer a general treatment of higher-order procedures as well as mutable data structures which may store such procedures.

We address these technical issues by augmenting the logic for imperative higher-order functions introduced in [24] with a pair of mutually dual logical primitives called *content quantifiers*. They offer an effective middle layer with clear logical status for reasoning about aliasing. The existential part of the primitives, written $\langle !x\rangle C$, is defined by the following equivalence:

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

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

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

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

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

Closely related with its use in logical calculation is a powerful descriptive/reasoning framework enabled by content quantification, in conjunction with standard logical primitives. By allowing hypothetical statements about the content of references separate from reference names themselves (which is the central logical feature of these operators), complex aliasing situations are given clean, succinct description, combined with effective compositional reasoning principles. This is particularly visible when we describe and reason about disjointness and sharing of mutable data structures (in this sense it expands the central merits of "separating connectives" [34, 37], as we shall discuss in later sections). The primitives work seamlessly with the logical machinery for capturing pure and imperative higher-order behaviour studied in [22, 23, 24], enabling precise description and efficient reasoning for a large class of higher-order behaviour and data structures. The descriptive power of the logic is formally clarified in Section 4 by showing the assertion language is *observationally complete* in the sense that two programs are contextually indistinguishable exactly when they satisfy the same set of assertions.

Third, and somewhat paradoxically, these merits of content quantification come without any additional expressive power: any formula which contains content quantification can be translated, up to logical equivalence, into one without. While establishing this result, we shall also show that content quantification and semantic update are mutually definable. Thus name (in)equations, content quantification and semantic update are all equivalent in sheer expressive power: the laws of content quantification are reducible to the standard axioms for the predicate calculus with equality, which in turn are equivalent to semantic update through its axioms for decomposition. This does not however diminish the significance of content quantification: without identifying it as a proper logical primitive with associated axioms, it is hard to consider its use in reasoning, both in logical calculation and in its applications to structured reasoning for programs and data structures in the presence of general aliasing. To our knowledge [2, §10], neither the calculation method nor the reasoning principle proposed in the present paper is discussed in the foregoing work.

In the remainder, Section 2 introduces the programming/assertion languages. Section 3 presents axioms and proof rules. Section 4 records key technical results on the logic. Section 5 illustrates the use of the logic through concrete reasoning examples including a higher-order, polymorphic Quicksort. Section 6 discusses related work (including earlier logics for sublanguages of Algol and more recent ones such as Separation Logic) and concludes with further issues. A full version [2] contains detailed proofs, further examples and an extensive historical survey.

2. Assertions and their Semantics

2.1 Programming Language.

As a target programming language, we use call-by-value PCF with unit, sums and products, augmented with imperative constructs [35]. Let $x, y, \ldots$ range over an infinite set of *names*. Then types, values and programs are given by the following grammar.

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

$$V ::= \mathsf{c} \mid x \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha\Rightarrow\beta}.\lambda y^{\alpha}.M \mid \langle V, W\rangle \mid \mathtt{in}_i(V)$$

$$M ::= V \mid MN \mid M := N \mid !M \mid \mathsf{op}(\tilde{M}) \mid \pi_i(M) \mid \langle M, N\rangle \mid \mathtt{in}_i(M)$$

$$\mid \mathtt{if}\ M\ \mathtt{then}\ M_1\ \mathtt{else}\ M_2 \mid \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{in}_i(x_i^{\alpha_i}).M_i\}_{i\in\{1,2\}}$$

We use standard boolean/arithmetic constants and operations, resp. ranged over by `c` and `op`. Types can carry reference types: hence procedures, references and data structures may pass/return/store reference names, leading to general forms of aliasing as discussed in the Introduction. We freely use obvious shorthands like $M;N$ and $\mathtt{let}\ x = M\ \mathtt{in}\ N$.

A *basis* $\Gamma;\Delta$ is a pair of finite maps one from names to non-reference types (Γ, ..., called *environment basis*) and the other from names to reference types (Δ, ..., called *reference basis*). $\Theta, \Theta', \ldots$ combine two kinds of bases. The typing rules are standard and omitted [35]. We write $\Gamma;\Delta \vdash M : \alpha$ when M has type α under $\Gamma;\Delta$.

A program is *semi-closed* if its environment basis is empty, written $\Delta \vdash M : \alpha$. A *store* $(\sigma, \ldots)$ is a finite map from reference names to semi-closed values, to which the typing extends in the obvious way. Using *configurations* of the form (M, σ) with semi-closed M and store σ typable under a common basis, the call-by-value, one step reduction, written $(M,\sigma) \longrightarrow (M',\sigma')$, is defined in the standard way [18, 35]. We write

$$(M,\sigma)\Downarrow \qquad \text{if} \quad \exists V,\sigma'.\, (M,\sigma) \longrightarrow^* (V,\sigma') \not\longrightarrow .$$

Henceforth we only consider well-typed programs and configurations.

2.2 Models.

We introduce a class of models which concisely represent computational situations of interest. We follow our previous work [24] except for the additional use of *distinctions* to describe aliasing, an innovation coming from the π-calculus [32]. Our models are immediately faithful to the observable behaviour of programs, which is important for our logic's observational completeness, established later.

A *distinction over* Δ $(\mathcal{D}, \ldots)$ is a type-respecting equivalence relation over $\mathsf{dom}(\Delta)$. The equivalence classes of a distinction are called its *identicals* $(\mathbf{i}, \mathbf{j}, \ldots)$. The point of using distinctions is to have a modular way of specifying in the model which references are aliased (those in the same identical) and which are not. Let $\Delta \vdash M : \alpha$ and let $\mathcal{D}$ be a distinction over Δ. Regarding identicals of $\mathcal{D}$ as names, we can substitute a $\mathcal{D}$-identical $\mathbf{i}$ for each name $x \in \mathbf{i}$ in M, which we write $M\mathcal{D}$. Intuitively, $M\mathcal{D}$ is a program whose names are coalesced following $\mathcal{D}$. $M\mathcal{D}$ is typed by $\Delta\mathcal{D}$, which is defined similarly. For example, given $M \stackrel{\text{def}}{=} \mathtt{if}\ x = y\ \mathtt{then}\ 0\ \mathtt{else}\ 1$, if $\mathcal{D}$ only equates x and y then we have

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

Note $M\mathcal{D}$ reduces as $(M\mathcal{D},\sigma) \longrightarrow (0,\sigma)$ which is quite different from M itself, showing that distinctions affect observable behaviour of programs.

A *typed context* $C[\,\cdot\,]_{\Gamma;\Delta;\alpha}$ is a context with a hole typed with α under $\Gamma;\Delta$. A typed context is *semi-closing* if it does not λ-abstract any reference name in the hole and the resulting program is semi-closed. Δ is *complete* if, whenever $\mathsf{Ref}(\alpha)$ occurs in a type in Δ's range, there is a name of that type in Δ. Let Δ be complete and $\mathcal{D}$ be a distinction over Δ and assume $\Gamma;\Delta \vdash M_{1,2} : \alpha$. Then we write

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

for all semi-closing $C[\,\cdot\,]_{\Gamma;\Delta;\alpha}$ and well-typed stores σ. The standard contextual congruence [35], which we denote by $\cong$, coincides with the closure of $\cong_{\mathcal{D}}$ under arbitrary distinctions.

An *abstract value of type* $(\mathcal{D};\Delta;\alpha)$ is a $\cong_{\mathcal{D}}$-congruence class of semi-closed values which are typed as α under Δ. We let $\mathbf{v}$, $\mathbf{w}$,... range over abstract values. In short, abstract values are semi-closed values taken modulo the typed congruence relative to a given distinction. Since reference names are values, identicals are also abstract values (of appropriate types). We write $[V]^{\mathcal{D};\Delta;\alpha}$ for an abstract value whose representative is V, and $[\![\alpha]\!]^{\Delta}_{\mathcal{D}}$ for the set of all abstract values of type $(\mathcal{D};\Delta;\alpha)$. We can now define a model.

Definition 1 *A* model of type $\Gamma;\Delta$, *written* $\mathcal{M}^{\Gamma;\Delta}$, *is a triple* $(\mathcal{D},\xi,\sigma)$ *where*

- *$\mathcal{D}$ is a distinction on Δ;*
- *ξ, called* environment, *is a finite map from $\mathsf{dom}(\Gamma,\Delta)$ to abstract values which is type-respecting in the sense that each $x \in \mathsf{dom}(\Gamma,\Delta)$ is mapped to an abstract value of type $\mathcal{D};\Delta;\Gamma(x)$;*
- *σ, called* (abstract) store, *is a finite map from the identicals of $\mathcal{D}$ to abstract values which is type-respecting in the sense that each $\mathbf{i} \in \mathcal{D}$ is mapped to an abstract value of type $\mathcal{D};\Delta;\alpha$ assuming $(\Delta\mathcal{D})(\mathbf{i}) = \mathsf{Ref}(\alpha)$.*

2.3 Syntax of Assertions.

The logical language is standard first-order logic with equality [31, § 2.8], extended with assertions for evaluation with side effects [24] and quantifications over store content. Let $\star \in \{\wedge,\vee,\supset\}$ and $Q \in \{\forall,\exists\}$. We highlight changes from [24].

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

The first set of expressions (ranged over by $e, e', \ldots$) are *terms* while the second set are *formulae* (ranged over by $A,B,C,C'\ldots$). The constants include unit, numerals and booleans, while $\mathsf{op}(\tilde{e})$ ranges over first-order operations, both coming from the underlying programming language. We also have pairing, projection and injection, again corresponding to those in the target programming language. The final term $!e$ dereferences e. Unlike in [24], quantification can abstract variables of all types including references. We also use truth T (definable as $1 = 1$) and falsity F (which is $\neg\mathsf{T}$). Finally, $x \neq y$ stands for $\neg(x = y)$.

The formula $\{C\}\ e \bullet e' \ =\ x\ \{C'\}$ is called *evaluation formula* [24], where the name x binds its free occurrences in C'. Intuitively, $\{C\}\ e \bullet e' \ =\ x\ \{C'\}$ asserts on the evaluation of an application with pre/post conditions, and can be read:

> *an invocation of e with an argument e' under hypothetical initial state C (pre-condition) terminates with a final state and a resulting value, the latter named x, both described by C' (post-condition).*

The pre/post conditions are about hypothetical state since we often need to describe imperative behaviour independent from a current state. For example,

$$!x = 1 \ \wedge\ \forall i.\forall j.\{!x = i\}\ f \bullet j \ =\ z\ \{z = !x = i + j\}$$

asserts that (1) the current content of x is 1; and (2) if, hypothetically, the content of x is i and f is invoked with j, then the return value and the resulting content of x are both $i + j$. Note that $e \bullet e' = x$ in $\{C\}\ e \bullet e' \ =\ x\ \{C'\}$ is asymmetric and $\bullet$ is not commutative. [1] Content quantifications $\langle !e\rangle$ and $[!e]$ are illustrated through examples later. $\mathsf{fv}(C)$ denotes the set of free variables in C. $C^{\text{-}\tilde{x}}$ indicates $\mathsf{fv}(C) \cap \{\tilde{x}\} = \emptyset$. Binding in formulae is induced only by standard quantifiers, $\forall$, $\exists$, and by evaluation formulae. In particular, $\mathsf{fv}(\langle !e\rangle C) = \mathsf{fv}([!e]C) = \mathsf{fv}(e) \cup \mathsf{fv}(C)$. Formulae are taken up to α-convertibility (some care is needed to avoid name capture, as illustrated in [2, §5.2], though all concrete examples in this paper can be read without this concern).

Starting from variables, each term can be typed inductively. Using typed terms is not strictly necessary but contributes to clarity

[1] In [24], we wrote $\{C\}\ e \bullet e' \searrow x\ \{C'\}$ instead of $\{C\}\ e \bullet e' \ =\ x\ \{C'\}$.

and understandability. We write $\Theta \vdash e : \alpha$ when e has type α under Θ. We also write $\Theta \vdash C$ when terms in C are well-typed under Θ. *Henceforth we only treat well-typed terms and formulae.* Type annotations for variables are often omitted in examples.

Logical substitution plays an important role in the present logic. We define, with m fresh:

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

Intuitively $C\{\!\!\{e_2/!e_1\}\!\!\}$ describes the situation where a model satisfying C is updated at a memory cell referred to by e_1 (of a reference type) with a value e_2 (of its content type), with $e_{1,2}$ interpreted in the current model. Through the help of axioms discussed later, logical substitution interacts with content quantification just as syntactic substitution does with conventional quantification. For example, $[!x]C \supset C\{\!\!\{e/!x\}\!\!\}$ for any x, e and C, which corresponds to the familiar $\forall x.C \supset C[e/x]$. $C\{\!\!\{e/!x\}\!\!\} \supset \langle !x\rangle C$ also holds, corresponding to the standard entailment $C[e/x] \supset \exists x.C$.

Convention Logical connectives are used with 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 $(C_1 \supset C_2) \wedge (C_2 \supset C_1)$. Similarly,

- $\{C\}e \bullet e'\{C'\}$ stands for $\{C\}e \bullet e' = x\{x = () \wedge C'\}$ with $x \notin \mathsf{fv}(C')$; and
- $\{C\}e \bullet e' = e''\{C'\}$ for $\{C\}e \bullet e' = x\{x = e'' \wedge C'\}$ with $x \notin \mathsf{fv}(C') \cup \mathsf{fv}(e'')$ and e'' not a variable.

Formulae are often called *assertions*.

2.4 Semantics of Assertions.

The interpretation of terms, written $[\![e]\!]_{\mathcal{M}}$, is straightforward and omitted. The defining clauses for the satisfaction relation is standard [31] ($e_1 = e_2$ is interpreted by the identity; connectives are interpreted classically [23, 22, 24]), except: $\mathcal{M} \models \{C\}e \bullet e' = x\{C'\}$ is given following [24] (to wit: "if the given environment and any hypothetical state together satisfy C, then the application of $[\![e]\!]_{\mathcal{M}}$ to $[\![e']\!]_{\mathcal{M}}$ converges to a value (named x) and a state which together satisfy C'"); whereas standard and content quantifications are interpreted as:

- $\mathcal{M} \models \forall x^{\alpha}.C$ 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 model $\mathcal{M}'$ such that $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$.
- $\mathcal{M} \models [!e^{\mathsf{Ref}(\alpha)}].C$ if $[\![e]\!]_{\mathcal{M}} = [\![x]\!]_{\mathcal{M}}$ and for each $V \in [\![\alpha]\!]^{\Delta}_{\mathcal{D}}$ we have $\mathcal{M}[x \mapsto V] \models C$.
- $\mathcal{M} \models \langle !e^{\mathsf{Ref}(\alpha)}\rangle.C$ if $[\![e]\!]_{\mathcal{M}} = [\![x]\!]_{\mathcal{M}}$ and some $V \in [\![\alpha]\!]^{\Delta}_{\mathcal{M}}$ exists with $\mathcal{M}[x \mapsto V] \models C$.

Above $\mathcal{M} \leq_{x:\alpha} \mathcal{M}'$ means that $\mathcal{M}'$ is exactly like $\mathcal{M}$, except that the former has an additional entry for x (in detail: one new entry for x is added to the environment; if x has a reference type, it may either be adjoined to an existing identical or form a new identical for which a new entry is added to the store). $\mathcal{M}[x \mapsto V]$, with x a reference name in $\mathcal{M}$, is exactly like $\mathcal{M}$ except that it stores $[V]$ at the identical containing x.

2.5 Examples of Assertions

We illustrate usage of our assertion language through simple examples. Throughout we assume x, y, z are typed as $\mathsf{Ref}(\mathsf{Nat})$, while i, j are typed as Nat, unless otherwise stated.

1. (dereference) The assertion $x = 2$ with x typed as Nat, says that a (functional) variable has the value 2. The assertion $!x = 2$ says the content of a reference named x is 2. Finally the assertion $!!x = 2$ with x typed as $\mathsf{Ref}(\mathsf{Ref}(\mathsf{Nat}))$, says that the content of a reference which is itself the content of a reference, the last one named x, is 2.

2. (content quantification, 1) A simple formula using existential content quantification is $\langle !x\rangle !x = 3$. It is equivalent to T because all it says is that x can possibly store 3, which is surely true regardless of its current value (just as $\exists i.i = 3$ is always true). Dually $[!x]!x = 3$ is equivalent to F since it claims that x stores 3 whatever value x may store, which is impossible regardless of the current content of x (just as $\forall i.i = 3$ is a contradictory statement).

3. (content quantification, 2) Consider $\langle !x\rangle !y = 3$. Since if x and y are equal the content of both references are hidden, and because $!y = 3$ is equivalent to $(x = y \wedge !x = 3) \vee (x \neq y \wedge !y = 3)$, the assertion $\langle !x\rangle !y = 3$ is equivalent to $x = y \vee (x \neq y \wedge !y = 3)$ hence to $x \neq y \supset !y = 3$. Next consider $[!x]!y = 3$. It says whatever natural number a reference named x may store, the number stored in y is 3. For this to hold, it is sufficient and necessary that x and y name distinct memory cells and that the content of y is 3. Thus the assertion is logically equivalent to $x \neq y \wedge !y = 3$. In general, the assertion $\langle !e\rangle C$ claims C holds for the content of a reference qualified in C *if* that reference is distinct from e; whereas $[!e]C$ claims C holds *and* any reference whose content is discussed in C is distinct from e.

4. (swap, 1) Recall $\mathtt{swap} \stackrel{\text{def}}{=} \lambda(x,y).\mathtt{let}\ z = !x\ \mathtt{in}\ (x := !y; y := z)$ from the Introduction. The behaviour of this program, named u, can be described by the following assertion.
$$\forall xyij.\{!x = i \wedge !y = j\}u \bullet (x,y)\{!x = j \wedge !y = i\} \qquad (5)$$
Above and henceforth we use an evaluation formula with multiple arguments for readability.

5. (swap, 2) $\mathtt{swap}$ above in fact works for a pair of references of an arbitrary type, and is indeed typable as such in polymorphic programming languages like ML and Haskell. Following [23], we can capture its polymorphic behaviour by adding $\forall \mathrm{X}.C$ (and dually $\exists \mathrm{X}.C$) to the assertion language, with the grammar of types extended with type variables $(\mathrm{X}, \mathrm{Y}, \ldots)$ and quantifiers ($\forall \mathrm{X}.\alpha$ and $\exists \mathrm{X}.\alpha$). With this extension, we can refine (5).
$$\begin{array}{l}\forall \mathrm{X}.\forall x^{\mathsf{Ref}(\mathrm{X})}.\forall y^{\mathsf{Ref}(\mathrm{X})}.\forall i^{\mathrm{X}}.\forall j^{\mathrm{X}}.\\ \qquad \{!x = i \wedge !y = j\}u \bullet (x,y)\{!x = j \wedge !y = i\}\end{array} \qquad (6)$$
The assertion should be readable naturally. Types in assertions are interpreted syntactically, incorporating a map from type variables to closed types to the environment part of models [23].

6. (swap 3) The assertions (5) and (6) may not fully capture the behaviour of $\mathtt{swap}$ in that they do not say $\mathtt{swap}$ only modifies the content of references it receives as arguments (which can be crucial if we are to use $\mathtt{swap}$ as part of a larger program). To capture this property, we may assert, refining (5):
$$\begin{array}{l}\forall \mathrm{Y}.\forall z^{\mathsf{Ref}(\mathrm{Y})}.\forall h^{\mathrm{Y}}.\forall xyij.\ (z \neq xy \supset\\ \{!x = i \wedge !y = j \wedge !z = h\}\ u \bullet (x,y)\ \{!x = j \wedge !y = i \wedge !z = h\})\end{array} \qquad (7)$$
Above "$z \neq xy$" stands for "$z \neq x \wedge z \neq y$", similarly henceforth. z, j are polymorphically typed since we wish to say any reference of any type except xy are left unmodified. The assertion (7) now captures the whole observable behaviour of (monomorphic) $\mathtt{swap}$ in the sense that any program satisfying the assertion is observationally congruent to $\mathtt{swap}$ under arbitrary distinctions. Since (7) is slightly verbose, we may wish to use a shorthand, writing:
$$\forall xyij.\ \{!x = i \wedge !y = j\}u \bullet (x,y)\{!x = j \wedge !y = i\}\ @\ xy \qquad (8)$$

which formally stands for (7) (note the translation is mechanical). The general form of this construction is:

$$\{C\}e \bullet e' = x\{C'\} \ @ \ \{e_0, e_1, \ldots, e_{n-1}\} \tag{9}$$

where $\{e_0, e_1, \ldots, e_{n-1}\}$ (usually written as a sequence, as in (8)) is a finite set of terms of reference types, called *write set*, in which dereferences should not occur as subterms. The shorthanded form (9) is called *located assertion* and used extensively from now on.

7. (double) Let $\mathtt{double}^? \stackrel{\text{def}}{=} \lambda(x,y).(x := !x + !x;\, y := !y + !y)$. Obviously $\mathtt{double}^?$ will double the content of each of its two argument only if x and y are distinct. We give a located assertion for $\mathtt{double}^?$, named u.

   $\forall xyih.$

   $$\{!x = i \wedge\, !y = j \wedge x \neq y\}\ u \bullet (x,y)\ \{!x = 2 \cdot i \wedge\, !y = 2 \cdot j\} \ @ \ xy. \tag{10}$$

   This specification doesn't talk about the case $x = y$. A full specification of $\mathtt{double}^?$ is given as, with $A \stackrel{\text{def}}{=} \,!x = i \wedge\, !y = j$:

   $\forall xyij.$

   $$\{A\}u \bullet (x,y)\{(x = y \wedge !x = 4i) \vee (x \neq y \wedge !x = 2i \wedge !y = 2j)\} @ xy \tag{11}$$

   Specification (11) suggests how we may refine this program so that it becomes robust w.r.t aliasing. Let:

   $$\begin{aligned}\mathtt{double}^! \stackrel{\text{def}}{=} \ \ & \lambda(x,y).\mathtt{if}\ x = y\ \mathtt{then}\ x := !x + !x \\ & \qquad\qquad\quad \mathtt{else}\ x := !x + !x;\, y := !y + !y.\end{aligned}$$

   The program now meets the "expected" specification obtained by deleting "$x \neq y$" from the precondition of (10). The relationship between semantics of a program and its specification is clarified by observational completeness discussed in Section 4.

3. Proof Rules and Axioms

3.1 Judgement and its Validity.

A judgement consists of a program and a pair of formulae following Hoare [20], augmented with a fresh name called *anchor* [23, 22, 24], written $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$. This sequent is used for both validity and provability. If we wish to be specific, we prefix it with either $\vdash$ (for provability) or $\models$ (for validity). In $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$:

- u is the *anchor* of the judgement, which should *not* be in $\mathsf{dom}(\Gamma,\Delta) \cup \mathsf{fv}(C)$; and
- C is the *pre-condition* and C' is the *post-condition*.

An anchor is used to name the value resulting from M, and specifies its behaviour. As in Hoare logic, the distinction between primary and auxiliary names plays an important role in our logic. In $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$, the *primary names* are $\mathsf{dom}(\Gamma,\Delta) \cup \{u\}$, while the *auxiliary names* are those free names in C and C' which are not primary. *Henceforth we assume judgements are always well-typed*, in the sense that, in $\{C\}\ M^{\Gamma;\Delta;\alpha} :_u \{C'\}$,

> $\Gamma,\Delta,\Theta \vdash C$ and $\Gamma, u : \alpha, \Delta, \Theta \vdash C'$ such that $\mathsf{dom}(\Theta) \cap (\mathsf{dom}(\Gamma,\Delta) \cup \{u\}) = \emptyset$.

As a convenient notation, when $\alpha = \mathsf{Unit}$, we write $\{C\}M^{\Gamma;\Delta;\alpha}\{C'\}$ for $\{C\}M^{\Gamma;\Delta;\alpha} :_u \{C'\}$ with $u \notin \mathsf{fv}(C')$, recovering a Hoare triple [20]. Validity of judgement is given by the following clause.

Definition 2 (validity of judgement) We say $\{C\}\ M^{\Gamma;\Delta} :_m \{C'\}$ is *valid*, written $\models \{C\}\ M^{\Gamma;\Delta} :_m \{C'\}$, if, whenever $(\mathcal{D}, \xi, \sigma)^{\Gamma';\Delta'} \models C$, we have $(M\xi, \sigma) \Downarrow (\mathbf{v}, \sigma')$ such that $(\mathcal{D}, \xi \cdot m : \mathbf{v}, \sigma') \models C'$, for each Γ' and Δ' which respectively extend (i.e. are supersets of) Γ and Δ.

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

$$[In_1]\ \frac{\{C\}\ M :_v \{C'[\mathtt{in}_1(v)/u]\}}{\{C\}\ \mathtt{in}_1(M) :_u \{C'\}} \qquad [Proj_1]\ \frac{\{C\}\ M :_m \{C'[\pi_1(m)/u]\}}{\{C\}\ \pi_1(M) :_u \{C'\}}$$

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

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

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

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

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

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

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

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

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

Figure 1. Main proof rules. The difference from the rules in [24] is highlighted.

Thus a valid judgement demands termination. By using arbitrary models under arbitrary extensions of bases, the validity is robust with respect to arbitrary aliasing and weakening. Total correctness was chosen to facilitate comparison with [24]. It is straightforward to obtain partial correctness: only the recursion rule changes.

3.2 Main Proof Rules.

Figure 1 presents the main compositional proof rules. There is one rule for each language construct following its typing. In addition, there are *structural rules* which simply manipulate formulae. Some of the main structure rules are listed in Figure 2. For each rule we stipulate:

- Free $i, j, \ldots$ range over auxiliary names. Further no primary names in the premise(s) occur as auxiliary names in the conclusion (this may be considered as a variant of the bound name convention).
- $A, A', B, B', \ldots$ range over *stateless formulae*, i.e. those formulae which do not contain active dereferences (a dereference $!e$ is *active* if it does not occur in pre/post conditions of evaluation formulae nor under the scope of content quantification of $!e$: for example, $!x$ is active in $!x = 2$ but neither in $\langle !x\rangle(C \wedge !x = 2)$ nor in $\{!x = 0\}u \bullet ()\{!x = 1\}$).

Despite the added complexity of models due to aliasing, the only essential textual change in the proof rules from the logic for alias-free imperative higher-order functions in [24], is the replacement of syntactic with logical substitution in [*Assign*] (highlighted), demonstrating clean, rigorous stratification of logics. Below we discuss two rules for imperative constructs, using simple inferences for illustration.

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

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

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

$$[\wedge\text{-}\mathit{Post}]\ \frac{\{C\} M :_u \{C_1\} \quad \{C\} M :_u \{C_2\}}{\{C\} M :_u \{C_1 \wedge C_2\}} \qquad [\mathit{Aux}_\forall]\ \frac{\{C^{-i}\}\ M :_u \{C'\}}{\{C\}\ M :_u \{\forall i.C'\}}$$

$$[\mathit{Aux}_\exists]\ \frac{\{C\}\ M :_u \{C'^{-i}\}}{\{\exists i.C\}\ M :_u \{C'\}} \qquad [\mathit{Invariance}]\ \frac{\{C\}\ M^{\Gamma;\Delta;\alpha} :_m \{C'\}}{\{C \wedge A\}\ M :_m \{C' \wedge A\}}$$

Figure 2. Structural rules.

$[\mathit{Deref}]$ infers the property of $!M$ for an arbitrary program M of a reference type, saying:

> *If we wish to have C' as a result of dereference $!M$ named u starting from the initial state C, we should assume the same thing about M (to be evaluated into a reference) named x, substituting $!x$ for u in C.*

A simple example of using $[\mathit{Deref}]$ follows.

$\{\top\}\ x :_z \{z = x\}$	(Var, Conseq)
$\{\top\}\ \lambda x.x :_m \{\forall x.\{\top\} m \bullet x = z\{z = x\}\}$	(Abs, Aux$_\forall$)
$\{\forall x.\{\top\} m \bullet x = z\{z = x\}\}\ y :_n \{n = y \wedge \{\top\} m \bullet n = z\{z = y\}\}$	
$\{\top\}\ (\lambda x.x)y :_z \{!z = !y\}$	(App, Conseq)
$\{\top\}\ !((\lambda x.x)y) :_u \{u = !y\}$	(Deref)

In the second line we use a structural rule $[\mathit{Aux}_\forall]$. The third and fourth lines use standard equality laws (e.g. $z = y \supset !z = !y$).

The rule $[\mathit{Assign}]$ treats assignment of an arbitrary expression (of type α) to an arbitrary expression (of type $\mathsf{Ref}(\alpha)$), both possibly inducing side effects. It reads:

> *If the result after executing $M := N$ should satisfy C' starting from C, then, starting from the same state C, M named m should terminate to reach some C_0, and, in turn, N named n evaluates from C_0 to reach C', with its occurrences of n substituted for $!m$.*

Note the rule assumes the left-right evaluation order (the rule which assumes the right-left evaluation order, or no order at all, can be easily formulated). The next example starts from Line 4 in the previous inference.

1. $\{\top\}\ (\lambda x.x)y :_m \{m = y\}$	above
2. $\{m = y \wedge 1 = 1\}\ 1 :_n \{m = y \wedge n = 1\}$	(Const)
3. $(m = y \wedge n = 1) \quad \supset \quad (!y = 1)\{\!\mid n/!m \mid\!\}$	$(\star)$
4. $\{m = y \wedge 1 = 1\}\ 1 :_n \{(!y = 1)\{\!\mid n/!m \mid\!\}\}$	(2, 3, Conseq)
5. $\{\top\}\ (\lambda x.x)y := 1 \{!y = 1\}$	(1, 4, Assign)

Line 3, which involves semantic update, is derived later. The case with side effects can be reasoned similarly.

$[\mathit{Assign}]$ treats the most general form of assignment. From this rule, we can derive specialised assignment rules which offer more efficient reasoning. For example, if both sides of the assignment are simultaneously both logical terms and programs, we have the following simplified rule.

$$[\mathit{AssignS}] \quad \{C\{\!\mid e_2/!e_1 \mid\!\}\}\ e_1 := e_2\ \{C\}$$

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

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

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

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

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

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

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

Figure 3. Derived proof rules for located assertions (other rules directly follow Fig.1 and 2).

The rule is directly derivable from $[\mathit{Assign}]$ and $\{C[e/u]\}e :_u \{C\}$ (which is also derivable).

3.3 Structured Reasoning for Programs with Aliasing.

One of the central problems in large-scale software development is to prevent inadvertent interference between programs through shared variables, especially in the presence of aliasing. The located assertions in §2.5 address this concern by delineating part of the store a program may affect. Below we extend this idea to judgements.

$$\{C\} M :_u \{C'\}@\tilde{e} \overset{\text{def}}{\equiv} \{C \wedge y \neq \tilde{e} \wedge !y = i\} M :_u \{C' \wedge y \neq \tilde{e} \wedge !y = i\}$$

where y and i are fresh and distinct (to be precise, y and i are respectively typed as $\mathsf{Ref}(\mathrm{X})$ and X for a fresh X) and $\tilde{e}$ is a write set as for located assertions, cf. §2.5 (6). For example $\{!x = i\}\ x := !x + 1\ \{!x = i + 1\}@x$ says the command increments the content of x and does nothing else.

Valid located judgements are derivable by the proof rules for non-located judgements by translating located judgements to non-located ones. A more efficient method is to use compositional proof rules which are derivable in the original system but which are tailored for located judgements, the main ones of which are listed in Figure 3. The initial four rules should be naturally read (note $[\mathit{Assign}]$ demands the assigned reference to be among a write effect, while $[\mathit{Deref}]$ does not have such a condition). In $[\mathit{Invariance}]$, we say C is *$!e$-free* when $[!e]C \equiv C$. Since $!e$-freedom of C is (up to $\equiv$) equivalent to C having the shape $[!e]C'$ or $\langle !e\rangle C'$ for some C', the rule is in fact equipotent to each one of the following rules:

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

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

$[\mathit{Invariance}]$ and its variants improve the standard invariance rules in Hoare logics in that they need no extra-logical side condition (which says "M does not modify variables in C_0"). The next rule $[\mathit{SeqI}]$ ("I" for independent) is directly derivable from $[\mathit{InvUniv}]$, $[\mathit{InvExist}]$ and the standard sequencing rule. The rule looks lopsided, but its meaning is operationally transparent:

Assume (1) $\{C_1\}M_1\{C_1'\}@\tilde{e_1}$ and (2) $\{C_2\}M_2\{C_2'\}@\tilde{e_2}$. Suppose C_1 and C_2 initially hold, the latter regardless of the content of $\tilde{e}_1$. Let first M_1 run: then by (1), C_1' holds. Since M_1 only modifies $\tilde{e}_1$, C_2 still holds, so that if M_2 runs next, we reach C_2' by (2). This next run only modifies $\tilde{e}_2$, hence if *C_1' does not talk about $\tilde{e}_2$,* then *it should continue to hold in the final state.*

The rule directly infers a judgement for a sequenced pair of programs from independent judgements for the component programs. Here we show a very simple usage of this rule.

1	$\{\mathsf{T}\}\ x := 2\ \{!x = 2\}@x$	(AssignS)
2	$\{\mathsf{T}\}\ y :=!z\ \{!y = !z\}@y$	(AssignS)
3	$\{\mathsf{T}\}\ x := 2; y :=!z\ \{\langle !y\rangle !x = 2\ \wedge\ !y = !z\}@xy$	(SeqI)

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

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

Finally, [*Weakening*] is easily understood, while [*Thinning*] recovers extensionality (for example the judgement $\{\mathsf{T}\}\ x :=!x\ \{\mathsf{T}\}@\emptyset$ becomes derivable).

3.4 Laws of Content Quantification

Content quantification is introduced because aliasing cuts off the unique bond between a reference name and its content. Hence (hypothetical) properties of content need to be described independently from names. For verification, content quantification offers tractable reasoning on aliased references through succinct logical laws. These laws are deduced starting from axioms and applying inference rules in the standard way [31]. The axiom system includes the standard axioms and rules of first-order logic with equality [31, §2.8], formal number theory, as well as axioms for evaluation formulae, data types and content quantification. Here we focus on content quantification (other proper axioms follow our previous work [24], as detailed in [2, §6.3/6.4]).

Axioms for $[!x]$ and $\langle !x\rangle$ may be given following either those of standard quantifiers [31, §2.3] or those of modal operators [6]. Here we take the former approach, which is more concise. First we regard $\langle !x\rangle C$ as standing for $\neg[!x](\neg C)$. Then there are three axioms.

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

In (CA1), $C^{\text{-}!x}$ indicates C is *syntactically* $!x$*-free*. We generate the set of syntactically $!x$-free formulae, $\mathcal{S}^{\text{-}!x}$ by: (1) $[!x]C \in \mathcal{S}^{\text{-}!x}$; and (2) $C \wedge \bigwedge_i e_i \neq x \in \mathcal{S}^{\text{-}!x}$ where $\{e_i\}$ exhaust all active dereferences (cf.§3.2) in C; (3) the result of applying any logical connective (including negation) or standard/content quantifier except $\forall x, \exists x$ to formulae in $\mathcal{S}^{\text{-}!x}$ is again in $\mathcal{S}^{\text{-}!x}$. (CA1) corresponds to familiar $\forall x.(C_1 \supset C_2) \supset (C_1 \supset \forall x.C_2)$ with $x \notin \mathsf{fv}(C_1)$. (CA2) is a degenerate form of $\forall x.C \supset C[e/x]$. (CA3) says two ways to represent logical substitutions coincide, which is important to recover all properties of semantic update as studied in [7, 9, 10, 33], as discussed in the next section. Finally, to the rules of inference, we add the following analogue of standard generalisation.

$$\text{(CGen)}\quad C\ \Rightarrow\ [!x]C$$

In a deduction with non-trivial assumptions, we demand assumptions to be syntactically $!x$-free if the deduction uses (CGen) for $!x$. By the standard argument, we obtain the deduction theorem [31, §2.4].

Let us list some of the useful laws (focussing on existentials for our later convenience), all deducible from the axiom system.

$$\begin{array}{ll} \text{(L1)} & C \supset \langle !x\rangle C \\ \text{(L2)} & \langle !x\rangle\langle !x\rangle C \supset \langle !x\rangle C \\ \text{(L3)} & \langle !x\rangle !x = e \\ \text{(L4)} & \exists m.\langle !x\rangle C \equiv \langle !x\rangle \exists m.C \quad (m \neq x) \\ \text{(L5)} & \langle !x\rangle(C_1 \vee C_2) \equiv \langle !x\rangle C_1 \vee \langle !x\rangle C_2 \\ \text{(L6)} & [!x]C_1 \wedge \langle !x\rangle C_2 \supset \langle !x\rangle(C_1 \wedge C_2) \\ \text{(L7)} & \langle !x\rangle(C_1 \wedge C_2) \equiv C_1 \wedge \langle !x\rangle C_2 \quad (C_1\ !x\text{-free}) \\ \text{(L8)} & \text{if } C \equiv C_0^{\text{-}!x} \text{ then } C \text{ is } x\text{-free.} \end{array}$$

All are analogues of the well-known laws for existential quantifiers and the "May" modality. (L7) implies $(C_1 \wedge C_2)\{\!|e/!x|\!\} \equiv C_1 \wedge (C_2\{\!|e/!x|\!\})$ when C_1 is $!x$-free, suggesting the use of content quantification for realising the following locality principle which has been missing so far (a similar point is observed by Bornat [7], after his extensive exploration of logical calculations involving semantic update using Morris's method):

> *"If part of a formula does not concern the content of x, then an update (substitution) at x by some value should not affect that part."*

As a simple application of these laws, we derive $C\{\!|\mathsf{c}/!x|\!\}\{\!|e/!x|\!\} \equiv C\{\!|\mathsf{c}/!x|\!\}$ mentioned in the Introduction. Let $C' \overset{\text{def}}{=} C\{\!|\mathsf{c}/!x|\!\}$ and m be fresh. Writing (fol) indicates the use of first-order logic with equality [31, §2.8].

$$\begin{array}{lll} \multicolumn{3}{l}{\exists m.(\langle !x\rangle(C' \wedge !x = m) \wedge m = e)} \\ & \equiv \exists m.(C'\ \wedge\ \langle !x\rangle(\,!x = m)\ \wedge\ m = e) & \text{(L8, L7)} \\ & \equiv C'\ \wedge\ \exists m.\langle !x\rangle(\,!x = m\ \wedge\ m = e) & \text{(fol, L8)} \\ & \equiv C'\ \wedge\ \langle !x\rangle \exists m.(\,!x = m\ \wedge\ m = e) & \text{(L4)} \\ & \equiv C'\ \wedge\ \langle !x\rangle\ !x = e & \text{(fol)} \\ & \equiv C' & \text{(L3, fol)} \end{array}$$

Note the inference never touches C' (as it should not). As another example, we infer ($\star$), Line 3, from the inference in Page 9.

$$\begin{array}{llll} (m{=}y \wedge n{=}1) & \equiv & [!m](m{=}y \wedge n{=}1) & \text{(L8)} \\ & \equiv & [!m](m{=}y \wedge n{=}1)\ \wedge\ \langle !m\rangle !m{=}n & \text{(L3)} \\ & \supset & \langle !m\rangle(m{=}y \wedge n{=}1 \wedge !m{=}n) & \text{(L6)} \\ & \supset & (!y = 1)\{\!|n/!m|\!\}. & \text{(fol)} \end{array}$$

As noted in the Introduction, the proposed framework effectively subsumes the known calculation methods based on semantic update, which are often useful. Below we list simple ones (all are easily justifiable by the laws given above). Some others are also discussed in the next section. Below, in (S1-a) (resp. (S1-b)), we assume e_1 (resp. e) and e' do not contain dereferences.

(S0) $C\{\!|e'/!e|\!\} \equiv C$, assuming C is $!e$-free.

(S1-a) $(e' = !e_1)\{\!|e''/!e_2|\!\} \equiv ((e_1 = e_2 \wedge e' = e'') \vee (e_1 \neq e_2 \wedge e' = !e_2))$ or, as its special instance:

(S1-b) $(e' = !e)\{\!|e''/!e|\!\} \equiv e' = e''$.

4. Technical Results

This section records key theoretical properties of the logic, starting with soundness.

Theorem 1 (soundness for the proof rules) *If* $\vdash \{C\}\ M :_u \{C'\}$ *then we have* $\models \{C\}\ M :_u \{C'\}$.

PROOF: We show [*Deref*,*Assign*]. Write $(M\xi, \sigma) \Downarrow_m (\xi{\cdot}m : \mathbf{v}, \sigma')$ for $(M\xi, \sigma) \Downarrow (\mathbf{v}, \sigma')$. For [*Deref*]:

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

For $[Assign]$ we reason, with $\xi_0 = \xi \cdot m : \mathbf{i}$:

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

The last line is by the logical equivalence between $\mathcal{M} \models C'\{\!|n/!m|\!\}$ and $\mathcal{M}[\, [\![m]\!]_{\mathcal{M}} \mapsto [\![n]\!]_{\mathcal{M}}\,] \models C'$, which is immediate. See [24, §5.5] and [2, §8.1] for other cases. □

Proposition 1 (soundness of the axioms) (1) *(CA1–3) in § 3.4 are valid.* (2) *(CGen) is sound in the sense that if C is valid then so is $[!x]C$.*

For the proofs, see [2, §8.1]. Since proof rules in Figure 3 are derivable by those in Figures 1 and 2 (as well as Kleymann's strengthened consequence rule for (Thinning) [27]), we also know:

Corollary 1 *If $\vdash \{C\}\ M :_u \{C'\}\ @\ \tilde{e}$ by the rules in Figure 3, then $\models \{C\}\ M :_u \{C'\}\ @\ \tilde{e}$.*

4.1 Elimination of Content Quantification.

We show any formula containing content quantification can be transformed into a formula without them, up to logical equivalence. This is closely related with the decomposition result in [10]. In the course of the proof, we also establish mutual representability between content quantification and Cartwright-Oppen/Morris's semantic update.

Proposition 2 *1. We have $[!e]C \equiv \forall m.C\{\!|m/!e|\!\}$ with m be fresh. Dually, $\langle !e\rangle C \equiv \exists m.C\{\!|m/!e|\!\}$.*

2. (following [10]) Let $\star \in \{\wedge, \vee, \supset\}$, $Q \in \{\forall, \exists\}$ and $z \notin \{x, y\}$.

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

3. If C has no content quantification we can rewrite C up to $\equiv$ as $\exists \tilde{r}\tilde{c}.(\ (\wedge_i c_i = !r_i) \wedge C')$, where $\tilde{r}\tilde{c}$ are fresh and C' has no active dereference.

PROOF: Mechanical using the axioms, see [2, §5.4]. □

Proposition 2 (1, 2), which depend on (CA3) in §3.4, establish a direct connection between content quantification and semantic update, allowing us to restore the latter's reasoning methods from [10, 9, 33]. Now transform $(!u = z)\{\!|m/!x|\!\}$ into: $\langle !x\rangle(!u = z \wedge !x = m)$ (with m fresh) which is equivalent to $(x = u \wedge m = z) \vee (x \neq u \wedge !u = z)$. Write $[\![(!u = z)\{\!|m/!x|\!\}]\!]$ for the formula on the right. Using Proposition 2,

$$\langle !x\rangle C \quad \equiv \quad \exists m.(\exists \tilde{r}\tilde{c}.((\wedge_i [\![(!r_i = c_i)\{\!|m/!x|\!\}]\!]) \wedge C'))$$

with C' without content quantification and m etc. fresh. Performing this transformation repeatedly, we obtain:

Theorem 2 (elimination) *For each C, there exists C' s.t. $C \equiv C'$ and no content quantification occurs in C'.*

4.2 Observational Completeness

A central property of our logic is its precise correspondence with the observational congruence, in the sense that two programs are contextually equivalent iff they satisfy the same set of assertions. This offers foundations of modular software engineering, where replacement of one module with another with the same specification does not violate the observable behaviour of the whole software, up to the latter's global specification.

For the proof we extend the method we used in [24], which we now outline. We first define a subset of programs which represent a limited class of behaviours, called *finite canonical forms* (FCFs, ranged over by $F, F', \ldots$), whose construction comes from game semantics [3]. Now let us say (C, C') is a *characteristic assertion pair* (or a *CAP*) of F at u when F is the least behaviour w.r.t. $\sqsubseteq$ s.t. $\models \{C\}M :_u \{C'\}$, where $\sqsubseteq$ is the preorder counterpart of $\cong$. We then derive CAPs for FCFs by introducing tailored proof rules which refine those in [24] with located judgements (cf. Figure 3). For writing down CAPs and constructing proof rules, we need a small extension of terms in our assertion language which stands for vectors of variable. Once we know there are CAPs for all FCFs, we can translate discernibility of finite contexts, hence FCFs, into the latter's CAPs, hence we know any discerning finite contexts can be represented by logical assertions, concluding the proof. Following [24] we work with total correctness assertions (TCAs) which represent properties closed upwards w.r.t. $\sqsubseteq$, which is enough (see [2, §8.3] for details). Writing $\vdash_{\mathsf{char}} \{C\}\ F :_u \{C'\}$ when $\{C\}\ F :_u \{C'\}$ is provable with the derived rules, we obtain:

Proposition 3 *If $\vdash_{\mathsf{char}} \{C\}\ F :_u \{C'\}$, then (C, C') is a CAP of F at u.*

Such (C, C') gives the *most general formula* for total correctness in the sense of [4], so that we observe:

Corollary 2 (relative completeness for FCFs) *If $\models \{C\}F :_u \{C'\}$ such that (C, C') are TCAs, then $\vdash \{C\}F :_u \{C'\}$.*

We conjecture that Corollary 2 extends to general programs by a similar argument. Now write $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$ whenever: $\models \{C\}M_1^{\Gamma;\Delta;\alpha} :_u \{C'\}$ iff $\models \{C\}M_2^{\Gamma;\Delta;\alpha} :_u \{C'\}$. We conclude:

Theorem 3 (observational completeness) $\Gamma; \Delta \vdash M_1 \cong M_2 : \alpha$ *iff* $\Gamma; \Delta \vdash M_1 \cong_{\mathcal{L}} M_2 : \alpha$.

PROOF: "Only if" is direct from the definitions. For "if" suppose $M_1 \cong_{\mathcal{L}} M_2$ but $M_1 \not\cong M_2$. By abstraction, we assume $M_{1,2}$ are semi-closed. By construction there is semi-closed FCF F and $\tilde{F}'$ such that $(FM_1, \tilde{r} \mapsto \tilde{F}') \Downarrow$ and $(FM_2, \tilde{r} \mapsto \tilde{F}') \Uparrow$. By Proposition 3, there are assertions which characterise F and $\tilde{F}'$. Let such formula for F at f be written $[\![F]\!](f)$. With $A \stackrel{\text{def}}{=} [\![F]\!](f) \wedge (\wedge_i [\![F_i']\!](!r_i))$ we now reason:

$$\begin{array}{lcl}
(FM_1,\ \tilde{r} \mapsto \tilde{F}') \Downarrow & \Rightarrow & f : [F]{\cdot}m : [M_1] \models \{\wedge_i [\![F_i']\!](!r_i)\}\ f \bullet m = z\ \{\mathsf{T}\} \\
 & \Rightarrow & \models \{\mathsf{T}\}\ M_1 :_m \{\forall f.\{A\}\ f \bullet m\ = z\ \{\mathsf{T}\}\}
\end{array}$$

But $\not\models \{\mathsf{T}\}\ M_2 :_m \{\forall f.\{A\} f \bullet m = z\{\mathsf{T}\}\}$, that is $M_1 \not\cong_{\mathcal{L}} M_2$. □

5. Reasoning Examples

This section illustrates the use of our logic for verifying correctness of programs, starting from simple examples and finishing with highlights from the derivation for a higher-order generic Quicksort [26], extracted from the full derivation in [2, §9]. Along the way we also show how easily our logic can accommodate generalisations in type structures. Throughout we use the rules for the located assertions from Figure 3.

5.1 Double and Swap

Double We begin with reasoning about $\mathsf{double}^?$ from § 2.5 (6), which exhibits different behaviour under different distinctions. We derive the specification (10). The key point is a use of [*SeqI*] in Figure 3 for a short inference, focussing on the extensional property of each part, setting $A \overset{\text{def}}{\equiv} x \neq y \wedge\ !x = i \wedge\ !y = j$.

$$\begin{array}{l}
\{!x=i\}\ x := !x+!x\ \{!x=2i\}@x \\ \hline
\{!y=j\}\ y := !y+!y\ \{!y=2j\}@y \\ \hline
\{!x=i \wedge [!x]!y=j\}\ x := !x+!x\ ;\ y := !y+!y\ \{\langle !y\rangle !x=2i \wedge\ !y=2j\}@xy \\ \hline
\{A\}\ x := !x+!x\ ;\ y := !y+!y\ \{x \neq y \supset (!x=2i \wedge\ !y=2j)\}@xy \\ \hline
\{A\}\ x := !x+!x\ ;\ y := !y+!y\ \{!x=2i \wedge\ !y=2j\}@xy \\ \hline
\{\mathsf{T}\}\,\mathsf{double}^? :_u \{\ \forall x,y.\ \{A\}\,u \bullet (x,y)\,\{!x=2i \wedge !x=2j\}\,@xy\ \}@\emptyset
\end{array}$$

where the first two lines are instantly derivable by (S1-b) in §3.4 and (AssignS), the third uses (SeqI), the next (Consequence), then ($\supset$-$\wedge$) and the last (Abs, $\mathsf{Aux}_\forall$).

Swap: Located Reasoning The `swap` procedure is a classical example for reasoning about aliased programs [10, 9, 12]. We can either use [*SeqI*] as done for $\mathsf{double}^?$ above, or use the traditional decomposition method via our result in §4.1. Next we verify that `swap` satisfies (8), § 2.5. We start with extensional reasoning for the two assignments and combine them using [*SeqI*]. Let $A \overset{\text{def}}{=} x = y \supset i = j$.

$$\begin{array}{ll}
\{!y = j\}\ x :=\ !y\ \{!x = j\}@x & \text{(AssignS)} \\ \hline
\{z = i\}\ y := z\ \{!y = i\}@y & \text{(AssignS)} \\ \hline
\{!y = j \wedge [!x]z = i\}\ x :=\ !y\ ;\ y := z\ \{\langle !y\rangle !x = j \wedge\ !y = i\}@xy & \text{(SeqI)} \\ \hline
\{!x = i \wedge !y = j \wedge z = i\} & \\
\qquad x :=\ !y\ ;\ y := z & \text{(Conseq)} \\
\{(x \neq y \supset !x = j) \wedge\ !y = i\}@xy & \\ \hline
\{A \wedge !x = i \wedge !y = j \wedge z = i\} & \\
\qquad x :=\ !y\ ;\ y := z & \text{(Invariance)} \\
\{A \wedge (x \neq y \supset !x = j) \wedge\ !y = i\}@xy & \\ \hline
\{!x = i \wedge !y = j \wedge z = i\}\ x :=\ !y\ ;\ y := z\ \{x = j \wedge y = i\}@xy & \text{(Conseq)} \\ \hline
\{!x = i \wedge !y = j\}\ !x :_z \{!x = i \wedge !y = j \wedge z = i\}@\emptyset & \text{(Deref)} \\ \hline
\{!x = i \wedge !y = j\} & \\
\qquad \mathtt{let}\ z =\ !x\ \mathtt{in}\ (x :=\ !y\ ;\ y := z) & \text{(Let)} \\
\{!x = j \wedge !y = i\}@xy & \\ \hline
\{\mathsf{T}\}\ \mathtt{swap} :_u \{\mathsf{Swap}_u\}@\emptyset & \text{(Abs)}
\end{array}$$

In the fifth line, A is stateless. In the sixth line, we used $!x = i \wedge\ !y = j$ entails A. The rest is immediate. We can further universally abstract the program, using [*TAbst*] (which appears at the end of §5.3).

Swap: Reasoning by Traditional Method For comparison, we now show reasoning based on the traditional method.

$$\begin{array}{ll}
\{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}\} & \\
\qquad x :=\ !y & \text{(AssignS)} \\
\{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\}@x & \\ \hline
\{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\}\ y := z\ \{!x = j \wedge !y = i\}@y & \text{(AssignS)} \\ \hline
\{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}\} & \\
\qquad x :=\ !y\ ;\ y := z & \text{(Seq)} \\
\{!x = j \wedge !y = i\}@xy & \\ \hline
(!x = i \wedge !y = j \wedge z = i)\ \supset\ (!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\} & (\star\star\star) \\ \hline
\{!x = i \wedge !y = j \wedge z = i\}\ x :=\ !y\ ;\ y := z\ \{!x = j \wedge !y = i\}@xy & \text{(Conseq)} \\ \hline
\{!x = i \wedge !y = j\}\ !x :_z \{!x = i \wedge !y = j \wedge z = i\}@\emptyset & \text{(Deref)} \\ \hline
\{!x = i \wedge !y = j\} & \\
\qquad \mathtt{let}\ z =\ !x\ \mathtt{in}\ (x :=\ !y\ ;\ y := z) & \text{(Let)} \\
\{!x = j \wedge !y = i\}@xy & \\ \hline
\{\mathsf{T}\}\ \mathtt{swap} :_u \{\mathsf{Swap}_u\} & \text{(Abs)}
\end{array}$$

In the sixth line we used the following derived rule, using the encoding $\mathtt{let}\ x = M\ \mathtt{in}\ N \overset{\text{def}}{=} (\lambda x.N)M$.

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

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

$$\begin{array}{lll}
\multicolumn{2}{l}{(!x = j \wedge !y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\}} & \\
\equiv & (!x = j)\{\!|z/!y|\!\}\{\!|!y/!x|\!\} \wedge (!y = i)\{\!|z/!y|\!\}\{\!|!y/!x|\!\} & (\text{Pro.2 (2)}) \\
\equiv & ((x = y \supset z = j) \wedge (x \neq y \supset !x = j))\{\!|!y/!x|\!\} \wedge & \\
 & (z = i)\{\!|!y/!x|\!\} & (S1) \\
\equiv & (x = y \supset z = j)\{\!|!y/!x|\!\} \wedge & \\
 & (x \neq y \supset !x = j)\{\!|!y/!x|\!\} \wedge (z = i)\{\!|!y/!x|\!\} & (\text{Pro.2 (2)}) \\
\equiv & (x = y \supset z = j) \wedge (x \neq y \supset (!x = j\{\!|!y/!x|\!\})) \wedge z = i & (\text{L7}) \\
\equiv & (x = y \supset z = j) \wedge (x \neq y \supset !y = j) \wedge z = i & (\text{S1}) \\
\subset & !x = i \wedge\ !y = j \wedge z = i & (\text{fol})
\end{array}$$

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

5.2 Circular References

Next we consider $x :=\, !!x$, an example of a circular data structure. Typing this program requires recursive types, which we outline first. Taking the equi-isomorphic approach [35] where recursively defined types are equated iff their representation as regular trees are isomorphic, the grammar of types is extended as follows, for both the programming language and for the assertion language.

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

The typing rules do not change except for taking types up to tree isomorphism. Accordingly no change is needed in the axioms and proof rules. We wish to prove the following judgement.

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

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

$$(!x = y \wedge\ !y = x) \supset !!x = x \equiv (!x = x)[!!x/!x] \equiv (!x = x)\{\!|!!x/!x|\!\}\ (\star\star)$$

Given $(\star\star)$, the inference is immediate:

$$\begin{array}{lll}
1. & \{(!x = x)\{\!|!!x/!x|\!\}\}\ x :=\, !!x\ \{!x = x\} & \text{(AssignS)} \\ \hline
2. & \{!x = y \wedge\ !y = x\}\ x :=\, !!x\ \{!x = x\} & (1, (\star\star), \text{Conseq})
\end{array}$$

Similarly we can easily derive

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

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 $!x = (1, \mathtt{inr}(x))$ says x stores a pair of 1 and the right injection of a reference to itself, precisely capturing graphical structure of the datum. We can also assert and reason about stored procedures including programs with Landin's recursion (e.g. $x := \lambda z.\mathtt{if}\ z = 0\ \mathtt{then}\ 1\ \mathtt{else}\ z \times (!x)(z-1)$, after whose run $(!x)n$ computes n's factorial), see [24] for details.

5.3 A Polymorphic, Higher-Order Procedure: Quicksort

Hoare's Quicksort is an efficient algorithm for sorting arrays using recursion. Apart from recursive calls to itself, Quicksort calls Partition, a procedure which permutes elements of an array so that they are divided into two contiguous parts, the left containing elements less than a "pivot value" *pv* and the right those greater than *pv*. The pivot value *pv* is one of the array elements which may ideally be their mean value. In the following we specify and derive a full specification of one instance of the algorithm, directly taken from its well-known C version [26, §5.1.1]. First we present the code, assuming a generic swapping procedure from §2.5 (4-6). We use indentation for scoping.

```
quicksort $\overset{\text{def}}{=}$
$\mu q.\ \lambda(a,c,l,r).$
    if $l < r$ then
        let $p' =$ par$(a,c,l,r)$ in
            $q(a,c,l,p'-1)$;
            $q(a,c,p'+1,r)$
```

```
par $\overset{\text{def}}{=}$
$\lambda(a,c,l,r)$
    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$(a[r], a[!p])$; $!p$
```

In these programs we omit type annotations for variables, the main ones of which (for both programs) are:

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

$\mathrm{X}[\,]$ is the type of a generic array, see below. 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.

- Its correctness crucially relies on the extensional behaviour of each part: when recursively calling itself twice in the last two lines of the Quicksort code, 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" [34, 37] through the use of logical primitives of general nature rather than those introduced for that specific purpose.
- The program makes essential use of a higher-order procedure, receiving as its argument a comparison procedure which is used for permuting elements.
- 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 [13], we believe a rigorous verification of Quicksort's extensional behaviour with higher-order procedures and polymorphism is given here for the first time.

Preparation: Arrays We first introduce arrays, both for programs and assertions. The presentation may offer basic ideas on how a new data type can be systematically incorporated in the presented logic. We add:

(types)	α	$::=$	$\ldots \mid \alpha[\,]$	$[Array]\ \dfrac{\Gamma\vdash M:\alpha[\,] \quad \Gamma\vdash N:\mathsf{Nat}}{\Gamma\vdash M[N]:\mathsf{Ref}(\alpha)}$
(programs)	M	$::=$	$\ldots \mid M[N]$	
(Terms)	e	$::=$	$\ldots \mid e[e'] \mid \mathsf{size}(e)$	

An array consists of a sequence of references: selecting an entry will return the corresponding reference which can then be dereferenced (we can equally treat less analytical presentation). In models an array is regarded as a finite partial injection from Nat to references.

Any data type is equipped with axioms characterising its behaviour. The main ones for arrays are: *Each array of size n is made up of n distinct references*:

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

Two arrays are equal iff their size and component references coincide.

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

Two distinct arrays never overlap (not applicable in some languages).

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

We also need proof rules for arrays, one introduction rule for array identifiers and one elimination rule for indexing. Since the former is common to all variables, we only add the latter.

$$[Array]\ \frac{\begin{array}{l}\{C\}\,M :_m \{C_0\} @ \tilde{e} \\ \{C_0\}\,N :_n \{C'[m[n]/u]\} @ \tilde{e'} \qquad C'[m[n]/u]\ \supset\ 0 \le n < \mathsf{size}(m)\end{array}}{\{C\}\ M[N] :_u \{C'\}\ @\ \tilde{e}\tilde{e'}}$$

The rightmost premise prevents out-of-bound errors (treatment of errors is further discussed in [2]).

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

$$\{\mathsf{T}\}\ \mathtt{qsort} :_u \{\forall \mathrm{X}.\mathsf{Qsort}_u\} @ \emptyset. \tag{12}$$

where we set, omitting types, Qsort_u is the predicate

$$\forall abclr. \begin{pmatrix} \{\mathsf{Equal}(ablr) \wedge \mathsf{Order}(c)\} \\ \quad u \bullet (a,c,l,r) \\ \{\mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)\} @ a[l\ldots r]ip \end{pmatrix} \tag{13}$$

Here $a[l\ldots r]ip$ is short for $a[l], \ldots, a[r], i, p$. The variable b is auxiliary and is of the same array type as a, denoting the initial copy of a, so that we can specify the change of a in the post-condition is only in the ordering of its elements. Each predicate used in (13) has the following meaning. For the precondition:

- $\mathsf{Equal}(ablr)$ 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.
- $\mathsf{Order}(c)$ says: *c calculates a total order without side effects.* Formally, it is the conjunction of:
  1. $\forall xy.\,(c \bullet (x,y) = \mathsf{T}\ \vee\ c \bullet (x,y) = \mathsf{F})$. In this assertion "$c \bullet (x,y) = e$" stands for "$\{\mathsf{T}\} c \bullet (x,y) = z\{z = e\} @ \emptyset$" ("the comparison terminates and has no side effects");
  2. $\forall xy.(x \neq y\ \supset\ (c \bullet (x,y) = \mathsf{T} \vee c \bullet (y,x) = \mathsf{T}))$ ("two distinct elements are always ordered"); and
  3. $(c \bullet (x,y) = \mathsf{T} \wedge c \bullet (y,z) = \mathsf{T}) \supset c \bullet (x,z) = \mathsf{T}$ ("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 $\mathsf{Equal}(ablr)$.
- $\mathsf{Sorted}(alrc)$ says: *the content of a in the range from l to r are sorted w.r.t. the total order implemented by c.* Formally: $\mathsf{Sorted}(aclr) \overset{\text{def}}{\equiv} \forall i,j.(l \le i < j \le r\ \supset\ c \bullet (!a[i], !a[j]) = \mathsf{T})$.

So Qsort_u in (13) as a whole says:

> Initially we assume two distinct arrays, a and b, of the same content from l to r ($\mathsf{Equal}(ablr)$), 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 from a full derivation of the judgement (12) recorded in [2, §9].

Reasoning (1): Sorting Disjoint Subarrays First we focus on the last two lines of `quicksort` which sort subarrays by recursive calls. The reasoning demonstrate how the use of our refined invariance rule offers quick inference by combining two local, extensional specifications. Concretely our aim is to establish:

$$\{C_1\}\ \mathsf{q}(a,c,l,p'-1)\ ;\ \mathsf{q}(a,c,p'+1,r)\ \{C_1'\}@a[l...r]ip \qquad (14)$$

where

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

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

Two newly introduced predicates are illustrated below.

$\mathsf{QsortBounded}(qj)$ with j of Nat type is used as an inductive hypothesis for recursion. It is the same as Qsort_q, given in (13), Page 10, except that it only works for a range no more than j and that it replaces "$\mathsf{Equal}(ablr)$" in the precondition of (13) with "$\mathsf{Perm}(ablr)$", which is necessary for the induction to go through. $\mathsf{Parted}(aclrk)$ says the subarray of a from l to r is partitioned at an intermediate index k w.r.t. the order defined by c. Formally it is given as:

$$l\le k\le r \ \wedge\ \forall j.(l\le j\le k \supset c\bullet(!a[j],!a[k])=\mathsf{T})$$
$$\wedge\ \forall j.(k\le j\le r \supset c\bullet(!a[k],!a[j])=\mathsf{T})$$

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 $\tilde{e_2} \overset{\text{def}}{=} a[l..p'-1]pi$ and $\tilde{e_3} \overset{\text{def}}{=} a[p'+1..r]pi$ (which are the parts touched by the first/second calls, respectively). We now derive:

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

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

$$C_2 \overset{\text{def}}{=} \mathsf{Equal}(ab'l(p'-1)) \wedge \mathsf{Order}(c) \wedge \forall j<k.\mathsf{QsortBounded}(qj) \wedge p'-1-l<k$$

$$C_2' \overset{\text{def}}{=} \mathsf{Perm}(ab'l(p'-1)) \wedge \mathsf{Sorted}(acl(p'-1))$$

$$C_3 \overset{\text{def}}{=} \mathsf{Equal}(ab'(p'+1)r) \wedge \mathsf{Order}(c) \wedge \forall j<k.\mathsf{QsortBounded}(qj) \wedge r-(p'+1)<k$$

$$C_3' \overset{\text{def}}{=} \mathsf{Perm}(ab'(p'+1)r) \wedge \mathsf{Sorted}(ac(p'+1)r)$$

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 the following derived rules (the first one is due to Kleymann).

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

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

Using these rules and [*SeqI*], (R.1/2/3/5) are immediate. The remaining step is the derivation of (R.4), the condition for [*Conseq-Aux*]. For this, the first step is to see:

$$(\mathtt{Dist} \wedge !a[p'] = !b'[p'] \wedge \mathsf{Perm}(bb'lr) \wedge \mathsf{Parted}(bclrp')) \supset ((C_2' \wedge C_3') \supset C_1') \qquad (15)$$

is valid, which is elementary ($\mathtt{Dist}$ says a and b' are distinct and do not overlap with p and i). Then by (CGen) (cf. §3.4) we can universally content quantify $!\tilde{e}_2\tilde{e}_3$ over (15). By $\mathtt{Dist}$, the antecedent of (15) is $!\tilde{e}_2\tilde{e}_3$-robust, hence we can apply (CA1) (cf. §3.4) to reach (R.4).

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.

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

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

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

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

$$(\mathsf{Invar}\wedge r-!i>0) \supset \left(\begin{array}{l}\{\mathsf{Invar}\wedge r-!i>0\} \\ \qquad c\bullet(!a[!i],pv)=z \\ \{c\bullet(!a[!i],pv)=z\wedge\mathsf{Invar}\wedge r-!i>0\}@\emptyset\end{array}\right)$$

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

$$\{C_{then}\}\ \mathtt{ifbody}\ \{\mathsf{Invar}\{\!|!i+1/!i|\!\}\wedge r-!i\le j)\}@a[l...r-1]ip \quad (\text{omitted})$$

$$C_{\neg then} \supset (\mathsf{Invar}\{\!|!i+1/!i|\!\}\wedge r-!i\le j)$$

$$\begin{array}{l}\{C_{inloop}\} \\ \qquad \mathtt{if}\ c(!a[!i],pv)\ \mathtt{then\ ifbody} \\ \{\mathsf{Invar}\{\!|!i+1/!i|\!\}\wedge r-!i\le j)\}@a[l...r-1]pi\end{array} \quad (\text{IfThen})$$

Thus reasoning about a conditional branch which involves a call to a received procedure is no more difficult than treating first-order expressions. Above we used the following simplification of [*If*].

$$[\textit{IfThen}]\ \frac{\{C\}\ M:_m\{C_0\}@\tilde{e} \quad \{C_0[\mathsf{T}/m]\}\ N\ \{C'\}@\tilde{e'} \quad C_0[\mathsf{F}/m]\supset C'}{\{C\}\ \mathtt{if}\ M\ \mathtt{then}\ N\ \{C'\}@\tilde{e}\tilde{e'}}$$

The rest of the verification for Partition is mechanical, so that we reach the following natural judgement:

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

Reasoning (3): Polymorphism We are now ready to derive the whole specification of Quicksort (12). 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 [23]). We also list the rule for "let" which is easily derivable from [*Abs*] and [*App*] through the standard encoding. Below, $\mathsf{ftv}(\Theta)$

indicates the type variables in Θ, similarly for $\mathsf{ftv}(C)$.

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

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

We now present the derivation. For brevity we use the following abbreviations: $C_\star \stackrel{\text{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Sorted}(aclr)$, $B' \stackrel{\text{def}}{=} \mathsf{Perm}(ablr) \wedge \mathsf{Order}(c) \wedge \forall j < k.\mathsf{QsortBounded}(qj) \ \wedge\ r - l \leq k$, and $B \stackrel{\text{def}}{=} B' \ \wedge\ l < r$. We also write $\mathtt{qsort}'$ for $\mathtt{qsort}$ in Page 9 without the first line (i.e. without μ/λ-abstractions), M for $\mathtt{q}(a,c,l,p'-1)\ ;\ \mathtt{q}(a,c,p'+1,r)$ and N for $\mathtt{q}(l,p'-1)\ ;\ \mathtt{q}(p'+1,r)$.

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

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

6. Conclusion

This paper introduced a program logic for imperative higher-order functions with general forms of aliasing, presented its basic theory, and explored its use for specification and verification through simple but non-trivial examples. Distinguishing features of the proposed program logic include a general treatment of imperative higher-order functions and aliasing; its precise correspondence with observational semantics [15, 19]; 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 which can fully capture behaviours 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 guarantee of programs.

The logic is built on our earlier work [24], where we introduced a logic for imperative higher-order functions without aliasing. In [24], a reference type in both the programming and assertion languages, is never carried in another type, which leads to the lack of aliasing: operationally, a procedure never receives or returns (and a reference never stores) references, while logically, equating two distinct reference names is 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 the logic in [24]. In the following we conclude the paper with discussions on remaining topics and related work.

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 [36]. 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). In this case, we do not have to change the assertion language but have only to 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.

$$\frac{\{C^{\text{-}x}\}\,N :_n \{C_0\} \quad \{([!x]C_0)[!x/n]\}\,M^{\Gamma;\Delta\cdot x:\mathsf{Ref}(\alpha);\beta} :_m \{C'^{\text{-}x}\}}{\{C\}\ \mathtt{new}\ x := N\ \mathtt{in}\ M^{\Gamma;\Delta;\beta} :_u \{C'\}} \tag{16}$$

which says that, when inferring for M, we can safely assume that the newly generated x is distinct from existing reference names, and that the description of the resulting state and value, C', should not mention this new reference (for further illustration, see [2, §10]). It is notable that this rule and its refinement for the restricted form of local references allow us to treat the standard parameter passing mechanism in procedural languages such as C and Java through the following simple translation: a procedure definition "$\mathtt{f}(\mathtt{x},\mathtt{y})\ \{...\}$" 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 (16). Thus the (lack of) aliasing in stack variables can be analysed as a special case of aliasing in general references, allowing uniform understanding.

In the fully general form of local references, a newly generated reference can be exported to the outside of its original scope (reminiscent of scope extrusion in the π-calculus [32]) and can live longer than the generating procedure. A procedure can now have local state, possibly changing behaviour each time it runs, 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. Among others we need to treat a newly generated reference which is exported to the outside, which means, in the context of (16), that C' should now be able to talk about the freshly generated reference. How this can be done with a clean and minimal extension to the present logic will be discussed in a forthcoming exposition.

Related Work.

A detailed historical survey of the program logics and reasoning methods which treat aliasing, is given in [2, §10], where the main efforts in this genre in the last three decades are discussed. Below we focus on some of the directly related Hoare-like program logics which treat aliasing. Janssen and van Emde Boas [25] 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 [10], Morris [33] and Trakhtenbrot, Halpern and Meyer [39]. The work by Cartwright and Oppen [10] presented a (relative) completeness result for a language with aliasing and procedures. Morris [33] gives extensive reasoning examples. Bornat [7] further explored Morris's reasoning method. Trakhtenbrot et al. [39] also presents 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 [16, 29, 4] contain logical analyses of aliasing (which goes back to [30]). More recently Kulczycki et al. [12] study possible ways to reason about aliasing induced by call-by-reference procedure calls.

A different approach to the logical treatment of aliasing, based on Burstall's early work, is *Separation Logic* by Reynolds, O'Hearn and others [37, 34]. 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 exhibits an interesting contrast with our approach, both philosophically and technically. Their logic starts from a resource-aware assignment rule [37]: $\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}$ 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 is available at address e, demonstrating the resource-oriented nature of the logic (motivated from reasoning for low-level code such as assemblers). 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 [20] and ours.

On the basis of these resource-oriented proof rules, [37, 34] propose a variant of the invariance rule.

$$\frac{\{C\}\ P\ \{C'\} \quad \mathsf{fv}(C_0) \cap \mathsf{modify}(P) = \emptyset}{\{C * C_0\}\ P\ \{C' * C_0\}} \tag{17}$$

The second premise is the standard side condition ($\mathsf{modify}(P)$ is the set of all stack-allocated variables which P may write to). Apart from this side condition, soundness of this rule hinges on the resource-oriented assignment/dereference rules described above, by which all the variables (addresses) in the heap which P may write to are explicitly mentioned in C. Like the standard invariance rule, this rule is intended to serve as an aid for modular verification of program correctness.

Separation Logic's ability to reason about aliased references crucially depends on its resource-oriented nature, the separating conjunction $*$ and a special predicate $\mapsto$ to represent content of memory cells. In contrast, the present work aims at a precise logical articulation of observational meaning of programs in the traditions of both Hennessy-Milner logic and Hoare logic, as exemplified by Theorem 3. Another difference is that our logic aims to make the best of the standard logical apparatus of first-order logic with equality to represent general aliasing situations. These differences come to life for example in the [*Invariance*] rule of §3, which plays a role similar to (17). Our rule relies on purely compositional reasoning about observable behaviour, which, as examples in the previous section may suggest, contributes to tractability in reasoning. Concrete examples will serve to elucidate the difference. The following shows a possible inference for $x := 2; y := !z$ through a direct application of (17, Assign, Inv, Seq, Consequence).

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

For the same program, a direct application of our invariance rule gives:

$$\begin{array}{lll}1 & \{\mathsf{T}\}\ x := 2\ \{!x = 2\}@x & \text{(Assign)}\\ \hline 2 & \{\mathsf{T}\}\ y := !z\ \{!y = !z\}@y & \text{(Assign)}\\ \hline 3 & \{\mathsf{T}\}\ x := 2; y := !z\ \{\langle !y\rangle !x = 2\ \wedge\ !y = !z\}@xy & \text{(SeqI)}\end{array}$$

Reflecting observational nature, the pre-condition simply stays empty. Note also that $\langle !y\rangle !x = 2\ \wedge\ !y = !z$ is equivalent to $(x \neq y \supset !x = 2)\ \wedge\ !y = !z$, which is more general than $x \mapsto 2\ *\ \exists i.(y \mapsto i\ \wedge\ z \mapsto i)$. Intuitively this is because the content quantification (here $\langle !y\rangle$) offers a more refined form of protection from sharing/aliasing.

These examples suggest a gain in generality in using the proposed logical framework for representation of sharing and disjointness of data structures. While $C_1 * C_2$ is practically embeddable as $[!\tilde{e}_2]C_1 \wedge [!\tilde{e}_1]C_2$ where $\tilde{e}_i$ exhausts active dereferences of C_i, the examples suggest 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 (for example a predicate $\mathsf{allocated}(e)$ may say e of a reference type is allocated). Because of differences in orientation, we also expect a fruitful interplay between ideas from Separation Logic and those from the proposed logic and its ramifications. As one such instance, our long version [2, §10] reports a generalisation of a refined version of (17) studied by O'Hearn, Yang and Reynolds [34].

We also note elimination procedures similar to our Theorem 2 are studied by Lozes [28] and Calcagno et al. [8].

Acknowledgements We thank the anonymous referees for their useful comments. This work was partially supported by EPSRC grants GR/R03075/01, GR/T04236/01, GR/S55538/01, GR/T04724/01 and GR/T03208/01.

References

[1] C– home page. http://www.cminusminus.org.

[2] A full version of the present paper. Available for download at www.dcs.qmul.ac.uk/˜kohei/logics.

[3] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *LICS'98*, pages 334–344, 1998.

[4] K R. Apt. Ten Years of Hoare Logic: a survey. *TOPLAS*, 3:431–483, 1981.

[5] Friedrich L. Bauer, Edsger W. Dijkstra, and Tony Hoare, editors. *Theoretical Foundations of Programming Methodology, Lecture Notes of an International Summer School*. Reidel, 1982.

[6] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.

[7] Richard Bornat. Proving pointer programs in hoare logic. In *Conf. on Mathematics of Program Construction*, LNCS. Springer-Verlag, 2000.

[8] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *Proc. FoSSaCs'05*, LNCS. Springer-Verlag.

[9] Robert Cartwright and Derek C. Oppen. Unrestricted procedure calls in Hoare's logic. In *POPL*, pages 131–140, 1978.

[10] Robert Cartwright and Derek C. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.

[11] Patrick Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, volume B*, pages 843–993. Elsevier, 1999.

[12] G. W. Kulczycki et al. Reasoning about procedure calls with repeated arguments and the reference-value distinction. Technical Report TR ♯02-13a, Dept. of Comp. Sci., Iowa State Univ., December 2003.

[13] Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[14] Robert W. Floyd. Assigning meaning to programs. In *Symp. in Applied Mathematics*, volume 19, 1967.

[15] Irene Greif and Albert R. Meyer. Specifying the Semantics of while Programs: A Tutorial and Critique of a Paper by Hoare and Lauer. *ACM Trans. Program. Lang. Syst.*, 3(4), 1981.

[16] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst.*, 2(4):564–579, 1980.

[17] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI'02*. ACM, 2002.

[18] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1995.

[19] Matthew Hennessy and Robin Milner. Algebraic Laws for Non-Determinism and Concurrency. *JACM*, 32(1), 1985.

[20] Tony Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.

[21] Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.

[22] Kohei Honda. From process logic to program logic. In *ICFP'04*, pages 163–174. ACM Press, 2004.

[23] Kohei Honda and Nobuko Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP'04*, pages 191–202. ACM Press, 2004.

[24] Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. In *Proc. LICS'05*, pages 270–279. IEEE, 2005.

[25] T. M. V. Janssen and Peter van Emde Boas. On the proper treatment of referencing, dereferencing and assignment. In *Proc. ICALP*, pages 282–300, 1977.

[26] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[27] Thomas Kleymann. Hoare logic and auxiliary variables. Technical report, University of Edinburgh, LFCS ECS-LFCS-98-399, October 1998.

[28] Etienne Lozes. Elimination of spatial connectives in static spatial logics. *TCS*, to appear.

[29] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Trans. Program. Lang. Syst.*, 1(2):226–244, 1979.

[30] John L. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[31] Elliot Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.

[32] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1):1–77, 1992.

[33] Joseph M. Morris. A general axiom of assignment/ assignment and linked data structures/ a proof of the Schorr-Wait algorithm. In [5], pages 25–52. Reidel, 1982.

[34] Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL'04*, 2004.

[35] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[36] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *HOOTS'98*, CUP, pages 227–273, 1998.

[37] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, 2002.

[38] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

[39] Boris Trakhtenbrot, Joseph Halpern, and Albert Meyer. From Denotational to Operational and Axiomatic Semantics for ALGOL-like languages: an overview. In *Logic of Programs*, volume 164 of *LNCS*, pages 474–500, 1984.