Reasoning about Object Systems in VTLoE

Ian Mason
Department of Applied Computing and Mathematics
University of Tasmania at Launceston
Launceston, Tasmania 7250, Australia

and

Carolyn Talcott
Computer Science Department, Stanford University
Stanford, CA 94305, USA

ABSTRACT

VTLoE (Variable Type Logic of Effects) is a logic for reasoning about imperative functional programs inspired by the variable type systems of Feferman. The underlying programming language, $\lambda_{\mathtt{mk}}$, extends the call-by-value lambda calculus with primitives for arithmetic, pairing, branching, and reference cells (mutable data). In VTLoE one can reason about program equivalence and termination, input/output relations, program contexts, and inductively (and co-inductively) define data structures. In this paper we present a refinement of VTLoE. We then introduce a notion of object specification and establish formal principles for reasoning about object systems within VTLoE. Objects are self-contained entities with local state. The local state of an object can only be changed by action of that object in response to a message. In $\lambda_{\mathtt{mk}}$ objects are represented as closures with mutable data bound to local variables. A semantic principle called simulation induction was introduced in our earlier work as a means of establishing equivalence relations between streams, object behaviors, and other potentially infinite structures. These are formulated in VTLoE using the class apparatus. The use of these principles is illustrated by validating a variety of basic tranformation rules.

Keywords: functional, imperative, Object, simulation induction, contextual assertion, VTLoE

1. Introduction

Imperative functional languages are programming languages that combine the higher-order aspects of functional languages with the ability to manipulate mutable data and with other facilities that have *effects*. Traditional examples include Lisp, Scheme, ML, and object-oriented languages. More recently there has been much interest in enriching functional programming languages with capabilities to manipulate state. There is a practical need to interface functional programming languages with non-functional languages, and to permit functional programs to affect their environment. A comprehensive semantic theory is needed to support more advanced implementation technology both for purely functional languages and for imperative functional languages such as ML, Scheme, and Lisp dialogues. It is also important to develop specification logics for such languages.

VTLoE is a logic for reasoning about imperative functional programs, inspired by the variable type systems of Feferman. These systems are two sorted theories of operations and classes initially developed for the formalization of constructive mathematics [3,4] and later applied to the study of purely functional languages [5,6]. An

extension incorporating non-local control effects was introduced in [23]. In [16] we presented the first-order part of VTLoE, and showed how the Meyer-Sieber examples can be treated in this logic. The full system [10,11] extends the first-order theory by incorporating a theory of classes. This extension provides a more expressive formalism including the ability to construct inductively defined sets and derive the corresponding induction principles.

VTLoE goes well beyond traditional programming logics, such as Hoare's logic [2] and Dynamic logic [9] by treating a richer language and expressing more properties. It is close in spirit to Specification Logic [21] and to Evaluation Logic [20]. These logics all incorporate a full first order theory of data, and the ability to express program equivalence, and the ability to assert and nest Hoare-like triples (called *contextual assertions* in VTLoE). In the case of Specification logic, the underlying programming languages are quite different: Specification Logic is about Algol-like programs that are strongly typed, can store only first-order data, and obey call-by-name semantics. In contrast VTLoE is about untyped ML- or Scheme-like languages that can store arbitrary values, and obey call-by-value semantics. Evaluation logic extends Moggi's metalanguage for computational monads [19] to a full constructive predicate logic which permits formulation of statements about evaluation of computations to values, and constitutes a framework for expressing reasoning principles. It is inspired by categorical descriptions of computation processes. Evaluation logic is a general system which reduces reasoning about programs to reasoning about mathematical structures for programming semantics, which are not necessarily fully-abstract with respect to a canonical semantics. These systems are justified by completeness results over classes of categorical interpretations.

The underlying programming language of VTLoE, $\lambda_{\mathtt{mk}}$, is based on the call-by-value lambda calculus extended by the reference primitives `mk`, `set`, `get`. Atoms, references and lambda abstractions are all first class values – they can be bound to lambda variables, stored, and returned from procedures. The logic combines the features and benefits of equational calculi as well as program and specification logics. There are three layers. The foundation is the syntax and semantics of $\lambda_{\mathtt{mk}}$, the underlying term/program language. The second layer is a first-order theory built on assertions of program equivalence and program modalities called *contextual assertions*. The third layer extends the logic to include class terms, class membership, and quantification over class variables.

A simple contextual assertion has the form $\mathtt{let}\{x := e\}[\![\Phi]\!]$ meaning that after execution of e and binding the result to x the formula Φ holds. One important principle for contextual reasoning is to be able to replace e by any operationally equivalent expression without changing the semantics of the contextual assertion. This principle fails in the simple semantics of VTLoE [11], which does not fully protect private local state from observation. In this paper we present a modified semantics for formulas of VTLoE that solves the privacy problem.

Objects are self-contained entities with local state. The local state of an object can only be changed by action of that object in response to a message. In $\lambda_{\mathtt{mk}}$-like languages objects are represented as closures with mutable data bound to local

variables [22,1]. A semantic principle called simulation induction was introduced in [13] for establishing equivalence relations between streams, object behaviors, and other potentially infinite structures. In [14] informal methods based on simulation induction are used to derive an optimized specialized window editor from generic specifications of its components. In this paper we introduce a formal notion of object specification and establish formal principles based on simulation induction for reasoning about object systems within VTLoE. The formulation of these principles relies heavily on the class apparatus of VTLoE.

The remainder of this paper is organized as follows. In §2 we summarize the syntax and semantics of $\lambda_{\mathtt{mk}}$, the same subject matter is treated in far greater detail in [11]. In §3 we give the syntax and modified semantics of first-order formulas. Central to the new localized semantics is a notion of a set of visible values. We discuss two possible definitions of visibility in §4. In §5 we state and prove two theorems concerning the local semantics. In §6 we introduce and discuss persistence properties. In §7 we establish a set of principles for reasoning about behaviors and objects. In §8 we introduce a notion of object specification and establish a general theorem relating specified behaviors and objects. In §9 we give several examples of object transformations. In §10 we summarize and suggest directions for future research. A useful set of laws established in earlier work are summarized in the appendix.

1.1. Notation

We conclude the introduction with a summary of notation. Let X, Y, Y_0, Y_1 be sets. We specify meta-variable conventions in the form: let x range over X, which should be read as: the meta-variable x and decorated variants such as x', x_0, ..., range over the set X. We use the usual notation for set membership and function application. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y. $\bar{y} = [y_1, \ldots, y_n]$ is the sequence of length n with ith element y_i. $\mathbf{P}_\omega(Y)$ is the set of finite subsets of Y. $\mathrm{Fmap}[Y_0, Y_1]$ is the set of finite maps from Y_0 to Y_1. $[Y_0 \to Y_1]$ is the set of total functions f with domain Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. For any function f, $f\{y := y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$. $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the natural numbers and $i, j, n, n_0, \ldots$ range over $\mathbb{N}$.

2. The Syntax and Semantics of Terms

2.1. Syntax

The syntax of the terms of $\lambda_{\mathtt{mk}}$ is a simple extension of the lambda calculus to include basic constants or atoms $\mathbb{A}$, (such as the Lisp booleans `t` and `nil` as well as the integers $\mathbb{Z}$), together with a collection of primitive operations, $\mathbb{F} = \bigcup_{n \in \mathbb{N}} \mathbb{F}_n$, where $\mathbb{F}_n$ is the (possibly empty) set of n-ary operations. The primitive operations include the memory operations (`get`, `set`, `mk`) and immutable pairs (`pr`, `fst`, `snd`, `ispr`?), in

addition to the usual operations for branching and arithmetic. We assume an infinite set of variables, $\mathbb{X}$ and use these to define, by mutual induction, the set of λ-abstractions, $\mathbb{L}$, the set of value expressions, $\mathbb{V}$, the set of expressions, $\mathbb{E}$, and the set of contexts, $\mathbb{C}$, as the least sets satisfying the following equations:

Lambda Expressions	$\mathbb{L} = \lambda\mathbb{X}.\mathbb{E}$
Immutable Pairs	$\mathbb{P} = \mathtt{pr}(\mathbb{V}, \mathbb{V})$
Value Expressions	$\mathbb{V} = \mathbb{X} + \mathbb{A} + \mathbb{L} + \mathbb{P}$
Value Substitutions	$\mathbb{S} = \mathrm{Fmap}[\mathbb{X}, \mathbb{V}]$
Expressions	$\mathbb{E} = \mathbb{V} + \mathtt{app}(\mathbb{E}, \mathbb{E}) + \mathbb{F}_n(\mathbb{E}^n)$
Contexts	$\mathbb{C} = \{\bullet\} + \mathbb{X} + \mathbb{A} + \lambda\mathbb{X}.\mathbb{C} + \mathtt{app}(\mathbb{C}, \mathbb{C}) + \mathbb{F}_n(\mathbb{C}^n)$

We let a range over $\mathbb{A}$, x, y, z range over $\mathbb{X}$, v range over $\mathbb{V}$, $\lambda x.e$ range over $\mathbb{L}$, σ range over $\mathbb{S}$, e range over $\mathbb{E}$, and C range over $\mathbb{C}$. Note that the structured data (pairs), $\mathbb{P}$, are taken to be values. λ is a binding operator and free and bound variables of expressions are defined as usual. $\mathrm{FV}(e)$ is the set of free variables of e. A *value substitution* is a finite map σ from variables to value expressions, we let σ range over value substitutions. e^σ is the result of simultaneous substitution of free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$. We represent the function which maps x to v by $\{x := v\}$. Thus $e^{\{x:=v\}}$ is the result of replacing free occurrences of x in e by v (avoiding the capture of free variables in v). Contexts are expressions with holes. We use $\bullet$ to denote a hole. $C[e]$ denotes the result of replacing any holes in C by e. Free variables of e may become bound in this process. $\mathrm{Traps}(C)$ is the set of variables that can actually be trapped in the process of filling the holes in C.

2.2. Notation and Abbreviations

For any syntactic domain Y and set of variables X we let Y_X be the elements of Y with free variables in X. For example $\mathbb{E}_\emptyset$ is the set of closed expressions, and $\mathbb{V}_{\{z\}}$ is the set of value expressions v such that $\mathrm{FV}(v) \subseteq \{z\}$. Furthermore, for any syntactic domain Y and set V of values we let Y_V be those elements of Y of the form y^σ for some $y \in Y_Z$ and $\sigma \in Z \to V$ where Z is a finite set of variables. For example $\mathbb{V}_{\{\lambda x.\mathtt{set}(z,x)\}}$ is the set of value expressions of the form $v^{\{z:=\lambda x.\mathtt{set}(z,x)\}}$ for $v \in \mathbb{V}_{\{z\}}$.

In order to make programs easier to read, we introduce some abbreviations.

$\texttt{let}\{x := e_0\}e_1$	abbreviates	$\texttt{app}(\lambda x.e_1, e_0)$
$\texttt{seq}(e)$	abbreviates	e
$\texttt{seq}(e_0, \ldots, e_n)$	abbreviates	$\texttt{let}\{d := e_0\}$ $\texttt{seq}(e_1, \ldots, e_n)$ $\quad d \notin \mathrm{FV}(e_i)$ for $i \leq n$
$\texttt{Y}$	abbreviates	$\lambda f.\texttt{let}\{z := \texttt{mk}(\texttt{nil})\}$ $\texttt{seq}(\texttt{set}(z, \lambda x.\texttt{app}(\texttt{app}(f, \texttt{get}(z)), x)),$ $\texttt{get}(z))$
$\texttt{islam?}$	abbreviates	$\lambda x.\texttt{and}(\texttt{not}(\texttt{isatom?}(x)),$ $\texttt{not}(\texttt{ispr?}(x)),$ $\texttt{not}(\texttt{iscell?}(x))))$
$\texttt{cond}()$	abbreviates	$\texttt{nil}$
$\texttt{cond}([e_0 \Rightarrow e_0'], [e_1 \Rightarrow e_1'], \ldots, [e_n \Rightarrow e_n'])$	abbreviates	$\texttt{if}(e_0, e_0', \texttt{cond}([e_1 \Rightarrow e_1'], \ldots, [e_n \Rightarrow e_n']))$

Note that Y is a fixed-point combinator essentially identical to the one suggested by Landin [12]. When applied to a functional F of the form $\lambda f.\lambda x.e$, Y creates a private local cell, z, with contents $G = \lambda x.\texttt{app}(\texttt{app}(F, \texttt{get}(z)), x)$, and returns G. By privacy of z, G is equivalent to $F(G)$ (cf. [13]).

2.3. Operational Semantics

The operational semantics of expressions is given by a reduction relation $\overset{*}{\mapsto}$ on a syntactic representation of the state of an abstract machine, called *descriptions*. A state has three components: the current state of memory, the current continuation, and the current instruction. Their syntactic counterparts are *memory contexts*, *reduction contexts* and *redexes* respectively. Redexes describe the primitive computation steps (β-reduction or the application of a primitive operation to a sequence of value expressions). Reduction contexts, $\mathbb{R}$, (called evaluation contexts in [8]) identify the subexpression of an expression that is to be evaluated next.

$$\mathbb{R} = \{\bullet\} + \texttt{app}(\mathbb{R}, \mathbb{E}) + \texttt{app}(\mathbb{V}, \mathbb{R}) + \mathbb{F}_{m+n+1}(\mathbb{V}^m, \mathbb{R}, \mathbb{E}^n)$$

R ranges over $\mathbb{R}$. The crucial fact to note is that an arbitrary expression is either a value expression, or *decomposes uniquely* into a redex placed in a reduction context. We represent the state of memory using memory contexts. A memory context Γ is a context of the form

$$\texttt{let}\{z_1 := \texttt{mk}(\texttt{nil})\} \ldots \texttt{let}\{z_n := \texttt{mk}(\texttt{nil})\}\texttt{seq}(\texttt{set}(z_1, v_1), \ldots, \texttt{set}(z_n, v_n), \bullet)$$

where $z_i \neq z_j$ when $i \neq j$. We have divided the context into allocation, followed by assignment to allow for the construction of cycles. Thus, any state of memory is constructible by such an expression. We let Γ range over memory contexts. We

can view memory contexts as *finite maps from variables to value expressions.* Thus we refer to their domain, $\mathrm{Dom}(\Gamma)$; modify them, $\Gamma\{z := \mathtt{mk}(v)\}$, when $z \in \mathrm{Dom}(\Gamma)$; and extend them, $\Gamma\{z := \mathtt{mk}(v)\}$, when $z \notin \mathrm{Dom}(\Gamma)$.

A *description* is a *pair*, $\Gamma; e$, with first component a memory context and second component an arbitrary expression. *Value descriptions* are descriptions whose expression is a value expression, $\Gamma; v$. If $\mathrm{Dom}(\sigma) \cap \mathrm{Dom}(\Gamma) = \emptyset$, then $(\Gamma; e)^\sigma$ is $(\Gamma^\sigma; e^\sigma)$ where Γ^σ is the result of applying σ to each element in the range of Γ (replacing v_i in the above form by v_i^σ). The reduction relation $\stackrel{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$ (defined in [11]). The interesting clauses are:

(beta) $\Gamma; R[\mathtt{app}(\lambda x.e, v)] \mapsto \Gamma; R[e^{\{x:=v\}}]$

(mk) $\Gamma; R[\mathtt{mk}(v)] \mapsto \Gamma\{z := \mathtt{mk}(v)\}; R[z] \qquad z \notin \mathrm{Dom}(\Gamma) \cup \mathrm{FV}(R[v])$

(get) $\Gamma; R[\mathtt{get}(z)] \mapsto \Gamma; R[v] \qquad$ assuming $z \in \mathrm{Dom}(\Gamma)$ and $\Gamma(z) = v$

(set) $\Gamma; R[\mathtt{set}(z, v)] \mapsto \Gamma\{z := \mathtt{mk}(v)\}; R[\mathtt{nil}] \qquad$ assuming $z \in \mathrm{Dom}(\Gamma)$

A description, $\Gamma; e$ is *defined* (written $\downarrow \Gamma; e$) if it evaluates to a value description. We say two closed expressions are *equidefined*, $e_0 \updownarrow e_1$, to mean that $(\downarrow e_0)$ iff $(\downarrow e_1)$ ($\downarrow e$ abbreviates $\downarrow \emptyset; e$ for closed e).

2.4. Uniformity of Reductions

Some simple consequences of the computation rules are that reduction is functional modulo alpha conversion, memory contexts may be pulled out of reduction contexts, and computation is uniform in free variables, unreferenced memory and reduction contexts.

Lemma (cr [11]):

(i) $\Gamma_0[e_0] = \Gamma_1[e_1]$

if $\Gamma; e \mapsto \Gamma_i; e_i$ for $i < 2$

(ii) $R[\Gamma[e]] \stackrel{*}{\mapsto} \Gamma; R[e]$

if $\mathrm{FV}(R) \cap \mathrm{Dom}(\Gamma) = \emptyset$.

(iii) $\Gamma; e \mapsto \Gamma'; e' \;\Rightarrow\; (\Gamma; e)^\sigma \mapsto (\Gamma'; e')^\sigma$

if $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\sigma) = \emptyset = \mathrm{FV}(\mathrm{Rng}(\sigma)) \cap (\mathrm{Dom}(\Gamma') - \mathrm{Dom}(\Gamma))$.

(iv) $\Gamma; e \mapsto \Gamma'; e' \;\Rightarrow\; (\Gamma_0 \cup \Gamma); e \mapsto (\Gamma_0 \cup \Gamma'); e'$

if $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_0) = \emptyset$.

(v) $\Gamma; R[e] \mapsto \Gamma'; R[e'] \;\Rightarrow\; \Gamma; R'[e] \mapsto \Gamma'; R'[e']$

if $(\mathrm{Dom}(\Gamma') \cap \mathrm{FV}(R')) \subseteq \mathrm{Dom}(\Gamma)$

In **(cr.i)** $=$ is the usual notion of alpha equivalence. It makes explicit the fact that arbitrary choice in cell allocation is the same phenomenon as arbitrary choice of names of bound variables.

2.5. Operational Equivalence

Two expressions are *operationally equivalent*, written $e_0 \cong e_1$, if for any closing context C, $C[e_0]$ is defined iff $C[e_1]$ is defined. In general it is very difficult to establish the operational equivalence of expressions. Thus it is desirable to have a simpler characterization of $\cong$, one that limits the class of contexts (or observations) that must be considered. A generalization of Milner's context lemma [18] provides the desired characterization. This theorem is the key to giving a semantics to VTLoE formulas.

Theorem (Generalized Context Lemma [13,11]):

$$e_0 \cong e_1 \;\Leftrightarrow\; (\forall \Gamma, \sigma, R)(\bigwedge_{i<2} \mathrm{FV}(\Gamma[R[e_i^\sigma]]) = \emptyset \;\Rightarrow\; (\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]]))$$

3. Syntax and Semantics of VTLoE

We first present the original semantics for VTLoE, restricting our attention to the first-order fragment, and illustrate the visibility problem of this semantics. We then refine the notation and semantics to solve the visibility problem.

3.1. The Syntax of the First Order Fragment

The first order fragment of VTLoE is a minor generalization of classical first order logic. The atomic formulas assert the operational equivalence of expressions. In addition to the usual first-order formula constructions, we add *contextual assertions:* if Φ is a formula and U is a `let` *context*, then $U[\![\Phi]\!]$ is a formula. The formula, $U[\![\Phi]\!]$, expresses the fact that the assertion Φ holds at the point in the program text, U, when and if the hole requires evaluation. The set $\mathbb{U}$ of `let` *contexts*, is defined as follows.

Definition ($\mathbb{U}$):

$$\mathbb{U} = \{\bullet\} + \texttt{let}\{\mathbb{X} := \mathbb{E}\}\mathbb{U}$$

Note that each `let` context has a *unique* hole (hence the notation U). The well-formed formulas, $\mathbb{W}$, of (the first order part of) our logic are defined as follows:

Definition ($\mathbb{W}$ – First-Order):

$$\mathbb{W} = (\mathbb{E} \cong \mathbb{E}) + (\mathbb{W} \Rightarrow \mathbb{W}) + (\mathbb{U}[\![\mathbb{W}]\!]) + (\forall \mathbb{X})(\mathbb{W})$$

3.2. Non-Local Semantics

In the original formulation of VTLoE, the meaning of formulas was given by a Tarskian satisfaction relation $\Gamma \models \Phi[\sigma]$.

Definition (Non-Local Satisfaction, $\Gamma \models \Phi[\sigma]$ – First Order): Assume $\Gamma, \sigma, \Phi, e_j$ are such that $\mathrm{FV}(\Phi^\sigma) \cup \mathrm{FV}(e_j^\sigma) \subseteq \mathrm{Dom}(\Gamma)$ for $j < 2$. Then we define the satisfaction relation $\Gamma \models \Phi[\sigma]$ by induction on the structure of Φ:

$$\Gamma \models (e_0 \cong e_1)[\sigma] \quad \text{iff} \quad (\forall R \in \mathbb{R}_{\mathrm{Dom}(\Gamma)})(\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]])$$

$$\Gamma \models (\Phi_0 \Rightarrow \Phi_1)[\sigma] \quad \text{iff} \quad (\Gamma \models \Phi_0[\sigma]) \quad \text{implies} \quad (\Gamma \models \Phi_1[\sigma])$$

$$\Gamma \models U[\![\Phi]\!][\sigma] \quad \text{iff} \quad (\forall \Gamma', \sigma')((\Gamma; U[\![\sigma]\!] \stackrel{*}{\mapsto} \Gamma'; [\![\sigma']\!]) \quad \text{implies} \quad \Gamma' \models \Phi[\sigma'])$$

$$\Gamma \models (\forall x)\Phi[\sigma] \quad \text{iff} \quad (\forall v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)})(\Gamma \models \Phi[\sigma\{x := v\}])$$

In the contextual assertion clause computation involving contexts is defined as follows.

Definition ($\Gamma; U[\![\sigma]\!] \stackrel{*}{\mapsto} \Gamma'; [\![\sigma']\!]$)**:** If $U = \mathtt{let}\{x_i := e_i\}_{1 \le i \le k}\bullet$, Γ is a memory context, σ is a value substitution over Γ, and $(\mathrm{Dom}(\Gamma) \cup \mathrm{Dom}(\sigma)) \cap \{x_1, \ldots, x_k\} = \emptyset$, then

$$\Gamma; U[\![\sigma]\!] \stackrel{*}{\mapsto} \Gamma'; [\![\sigma']\!]$$

means there are Γ_i, v_i, σ_i such that $\Gamma_0 = \Gamma$, $\sigma_0 = \sigma$, $\sigma_{i+1} = \sigma_i\{x_{i+1} := v_{i+1}\}$, $\Gamma_i; e_{i+1}^{\sigma_i} \stackrel{*}{\mapsto} \Gamma_{i+1}; v_{i+1}$, and $\Gamma'; [\![\sigma']\!] = \Gamma_k; [\![\sigma_k]\!]$.

3.3. Validity

We say that a formula is *valid*, written $\models \Phi$, if $\Gamma \models \Phi[\sigma]$ for Γ, σ such that $\mathrm{FV}(\Phi^\sigma) \subseteq \mathrm{Dom}(\Gamma)$. Following the usual convention we will often write Φ as an assertion that Φ is valid, omitting the $\models$ sign. Note that the underlying logic is completely classical.

Lemma (valid): $\models (e_0 \cong e_1)$ iff the meta-statement $(e_0 \cong e_1)$ is true.

Proof (valid): The atomic case asserts that all *uses* of the expressions (relative to Γ and σ) are equidefined. This definition is motivated by the fact that if

$$(\forall \Gamma, \sigma)(\Gamma \models e_0 \cong e_1[\sigma]) \qquad \text{(i.e. } \models e_0 \cong e_1\text{)}$$

then

$$(\forall \Gamma, \sigma)(\forall R \in \mathbb{R}_{\mathrm{Dom}(\Gamma)})(\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]]).$$

By the (**ciu**) theorem this amounts to

$$(\forall C)(C[e_0] \updownarrow C[e_1]). \qquad \text{(i.e. } e_0 \cong e_1\text{)}$$

$\Box$valid

Moreover with this definition we have that the natural reading of the hypothetical assertion

$$\text{(R)} \quad e_0 \cong e_1 \Rightarrow R[e_0] \cong R[e_1]$$

is valid (i.e. true for all Γ and σ). Also note that if $\Gamma \models (e_0 \cong e_1)[\sigma]$, then $\Gamma[e_0^\sigma] \cong \Gamma[e_1^\sigma]$, but not conversely.

3.4. Examples and Abbreviations

Negation is definable, $\neg\Phi$ is just $\Phi \Rightarrow \mathtt{False}$, where `False` is any unsatisfiable assertion, such as $\mathtt{t} \cong \mathtt{nil}$. Similarly conjunction, $\wedge$, and disjunction, $\vee$ and the

biconditional, $\Leftrightarrow$, are all definable in the usual manner. Given a particular U, for example $\mathtt{let}\{x := \mathtt{mk}(v)\}\bullet$, we will often abuse notation and write $\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\Phi]\!]$ rather than $(\mathtt{let}\{x := \mathtt{mk}(v)\}\bullet)[\![\Phi]\!]$. Thus we can express the computational definedness of expressions by the following assertion: $\neg\mathtt{seq}(e, [\![\mathtt{False}]\!])$ rather than $\neg(\mathtt{seq}(e, \bullet)[\![\mathtt{False}]\!])$. We let $\Downarrow e$ abbreviate $\neg(\mathtt{seq}(e, \bullet)[\![\mathtt{False}]\!])$ and $\Uparrow e$ abbreviate its negation.

Note that the context U will in general bind free variables in Φ. A simple example is the axiom which expresses the effects of $\mathtt{mk}$:

$$(\forall y)(\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\neg(x \cong y) \wedge \mathtt{iscell?}(x) \cong \mathtt{t} \wedge \mathtt{get}(x) \cong v]\!])$$

For simplicity we have omitted certain possible contexts from the definition of $\mathbb{U}$. However those left out may be considered abbreviations. Two examples are:
(1) $\mathtt{if}(e_0, [\![\Phi_0]\!], [\![\Phi_1]\!])$ abbreviates

$$\mathtt{let}\{z := e_0\}[\![(z \cong \mathtt{nil} \Rightarrow \Phi_1) \wedge (\neg(z \cong \mathtt{nil}) \Rightarrow \Phi_0)]\!] \qquad z \text{ fresh.}$$

(2) $\vartheta(e_0, \ldots, e_n, U[\![\Phi]\!], e_{n+1}, \ldots)$ abbreviates $\mathtt{seq}(e_0, \ldots, e_n, U[\![\Phi]\!])$

3.5. Violation of Privacy

One seemingly desirable logical principle for contextual reasoning is to be able to replace the e by any operationally equivalent expression without changing the semantics of the contextual assertion $\mathtt{let}\{x := e\}[\![\Phi]\!]$. In other words the following principle seems desirable:

(Ueq) $\quad e_0 \cong e_1 \Rightarrow (\mathtt{let}\{x := e_0\}[\![\Phi]\!] \Leftrightarrow \mathtt{let}\{x := e_1\}[\![\Phi]\!])$

There are several ways in which this can fail in the above version of VTLoE. For example e_0 may produce some garbage that e_1 does not, and this garbage may be detectable via Φ. For example let

$$e_0 = \mathtt{seq}(\mathtt{mk}(0), \mathtt{mk}(0))$$

$$e_1 = \mathtt{mk}(0)$$

$$\Phi = (\exists y_0)(\exists y_1)(\mathtt{iscell?}(y_0) \cong \mathtt{t} \wedge \mathtt{iscell?}(y_1) \cong \mathtt{t} \wedge \mathtt{eq}(y_0, y_1) \cong \mathtt{nil})$$

Then

(a) $\quad \models e_0 \cong e_1$

(b) $\quad \emptyset \models \mathtt{let}\{x := e_0\}[\![\Phi]\!][\emptyset]$

(c) $\quad \neg(\emptyset \models \mathtt{let}\{x := e_1\}[\![\Phi]\!][\emptyset])$

and hence (**Ueq**) is not valid.

Another more troublesome counterexample relies on the fact that e_0 and e_1 are equivalent due to the privacy of certain cells, however their privacy is not respected

by the contextual assertion. A simple example of this is:

$e_0 = \lambda x_0.x_0$

$e_1 = \mathtt{let}\{z := \mathtt{mk}(\lambda x_0.x_0)\}\lambda w.\mathtt{app}(\mathtt{get}(z), w)$

$\Phi = x \cong \lambda y.y$

A simple induction on the length of computations (similar to those found in [13]) establishes that e_0 and e_1 are operationally equivalent, and hence $e_0 \cong e_1$ is valid in VTLoE. The essential observation is that the cell z is local to the value/object returned by e_1 and thus invisible and its contents unalterable outside this scope. However it is not the case that

$$\models \mathtt{let}\{x := e_0\}[\![\Phi]\!] \Leftrightarrow \mathtt{let}\{x := e_1\}[\![\Phi]\!]$$

Clearly $\mathtt{let}\{x := e_0\}[\![\Phi]\!]$ is valid, being an instance of the more general valid principle: $\mathtt{let}\{x := v\}[\![x \cong v]\!]$. The problem is that it is *not* the case that

$$\emptyset \models \mathtt{let}\{x := e_1\}[\![\Phi]\!][\emptyset].$$

To see this observe that

$$\emptyset; \mathtt{let}\{x := e_1\}[\![\emptyset]\!] \stackrel{*}{\mapsto} \mathtt{let}\{z := \mathtt{mk}(\lambda x_0.x_0)\}\bullet; [\![\{x := \lambda w.\mathtt{app}(\mathtt{get}(z), w)\}]\!].$$

Consequently it suffices to see that the following fails:

$$\mathtt{let}\{z := \mathtt{mk}(\lambda x_0.x_0)\}\bullet \models x \cong \lambda y.y[\{x := \lambda w.\mathtt{app}(\mathtt{get}(z), w)\}].$$

Choosing R to be the context $\mathtt{let}\{u := \bullet\}\mathtt{seq}(\mathtt{set}(z, 1), \mathtt{app}(u, 1))$ a simple computation shows that

$$\mathtt{let}\{z := \mathtt{mk}(\lambda x_0.x_0)\}[R[\lambda w.\mathtt{app}(\mathtt{get}(z), w)]]$$

diverges while

$$\mathtt{let}\{z := \mathtt{mk}(\lambda x_0.x_0)\}[R[\lambda y.y]]$$

converges to $\mathtt{let}\{z := \mathtt{mk}(\lambda x_0.x_0)\}\bullet; 1$. The crux of the problem is that the privacy of the cell z is violated by the reduction context R. This in turn is traced back to the semantics of the contextual assertion itself: No cell that is newly created (in the course of evaluating the context) is treated as private. Similarly in the first counterexample the problem may be traced to the ability of the existential quantifier to range over values that are otherwise invisible.

3.6. Visibility and Local Semantics - First Order

One approach to solving the locality problem is to define a notion of visibility relative to a memory context Γ and a set of values V over that memory ($V \subseteq \mathbb{V}_{\mathrm{Dom}(\Gamma)}$). Let $Vis(\Gamma, V)$ denote the *visible values* relative to Γ and V, $Vis(\Gamma, V) \subseteq$

$\mathbb{V}_{\mathrm{Dom}(\Gamma)}$. This set should contain V and should be closed under simple constructions that do not effect (but may read visible parts of the memory) Γ. We will give two examples of possible candidates shortly. The question of whether or not either candidate is the correct one remains an open question.

Given a suitable definition of *Vis*, we can define a localized satisfaction relation. The key is the atomic case. Here reduction contexts can only mention visible values over the memory, not arbitrary values. Also we restrict the quantifiers to range over *visible* values. We define satisfaction relative to a set of visible values.

Definition (Localized Satisfaction, $\Gamma \models_V \Phi[\sigma]$ – First Order): Assume $V \subseteq \mathbb{V}_{\mathrm{Dom}(\Gamma)}$, and $\sigma \in \mathrm{Fmap}[\mathbb{X}, \mathit{Vis}(\Gamma, V)]$. Then

$\Gamma \models_V (e_0 \cong e_1)[\sigma]$ iff $(\forall R \in \mathbb{R}_{\mathit{Vis}(\Gamma, V)})(\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]])$

$\Gamma \models_V U[\![\Phi]\!][\sigma]$ iff

$(\forall \Gamma', \sigma')((\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; \sigma')$ implies $\Gamma' \models_{V \cup \mathrm{Rng}(\sigma')} \Phi[\sigma'])$

$\Gamma \models_V (\Phi_0 \Rightarrow \Phi_1)[\sigma]$ iff $(\Gamma \models_V \Phi_0[\sigma])$ implies $(\Gamma \models_V \Phi_1[\sigma])$

$\Gamma \models_V (\forall x)\Phi[\sigma]$ iff $(\forall v \in \mathit{Vis}(\Gamma, V))(\Gamma \models_V \Phi[\sigma\{x := v\}])$

where $\mathbb{R}_{\mathit{Vis}(\Gamma, V)}$ is the set of reduction contexts built from values in $\mathit{Vis}(\Gamma, V)$ instead of arbitrary values, and in the U clause we assume $\mathrm{Traps}(U) \cap \mathrm{Dom}(\sigma) = \emptyset$.

3.7. The Syntax and Semantics of Classes

Using methods of [3,6] and [23], we extend our theory to include a general theory of classifications (classes for short).

We extend the syntax to include class terms. Class terms are either class variables, $\mathbb{X}^c$, class constants, $\mathbb{A}^c$, or comprehension terms, $\{x \mid \Phi\}$. We also extend the set $\mathbb{W}$ of formulas to include class membership and quantification over class variables. The definition of expressions remains unchanged.

Definition ($\mathbb{K}$,$\mathbb{W}$): The set $\mathbb{K}$ of class terms and $\mathbb{W}$ of formulas are defined by

$\mathbb{K} = \mathbb{X}^c \cup \mathbb{A}^c \cup \{\mathbb{X} \mid \mathbb{W}\}$

$\mathbb{W} = (\mathbb{E} \cong \mathbb{E}) \cup (\mathbb{V} \in \mathbb{K}) \cup (\mathbb{W} \Rightarrow \mathbb{W}) \cup (\forall \mathbb{X})\mathbb{W} \cup (\forall \mathbb{X}^c)\mathbb{W} \cup \mathbb{U}[\![\mathbb{W}]\!]$

We let $A, B, C, \ldots X, Y, Z$ range over $\mathbb{X}^c$ and K range over $\mathbb{K}$. We will use identifiers beginning with an upper case letter in **This** font (for example **Val**) for class constants.

To define the semantics of this extension we first have to say what class variables range over, i.e. the range of an extended σ. The only reasonable interpretation seems to be sets of visible values. These sets should be closed under visibility restricted operational equivalence. That is, if v is in the Γ, V-class, K, and $\Gamma \models_V v \cong v'$, then v' is also in K. We let $\mathbb{K}_{\Gamma, V}$ denote the collection of such sets. We restrict membership formulas to value expressions. Note that $e \in K$ can be thought of as an abbreviation for $\mathtt{let}\{x := e\}[\![x \in K]\!]$.

Now we are ready to give the revised class semantics. Note that the valuation of class terms must be indexed by V as well.

Assume, in the next two definitions, that $V \subseteq \mathbb{V}_{\mathrm{Dom}(\Gamma)}$ and σ is a finite map with domain in $\mathbb{X} \cup \mathbb{X}^c$ that maps $\mathbb{X}$ to $Vis(\Gamma, V)$ and $\mathbb{X}^c$ to $\mathbb{K}_{\Gamma,V}$.

Definition ($[K]^\sigma_{\Gamma,V}$):

$$[X]^\sigma_\Gamma = \sigma(X)$$

$$[\{x \mid \Phi\}]^\sigma_{\Gamma,V} = \{v \in Vis(\Gamma, V) \mid \Gamma \models_V \Phi[\sigma\{x := v\}]\}$$

Definition (Localized Satisfaction, $\Gamma \models_V \Phi[\sigma]$ – Classes): The new clauses in the inductive definition of satisfaction are:

$$\Gamma \models_V v \in K[\sigma] \quad \text{iff} \quad v^\sigma \in [K]^\sigma_{\Gamma,V}$$

$$\Gamma \models_V (\forall X)\Phi[\sigma] \quad \text{iff} \quad (\forall C \in \mathbb{K}_{\Gamma,V})(\Gamma \models_V \Phi[\sigma\{X := C\}])$$

In the localized semantics, we say a formula, Φ, is valid just if $\Gamma \models_V \Phi[\sigma]$ for all appropriate Γ, V, σ.

For a full development of the theory of classes in VTLoE the reader is referred to [3,6,23,11]. Here we recall a few properties and classes used in later examples.

Lemma (Class [11]):

(allE) $(\forall X)\Phi[X] \Rightarrow \Phi[K]$ where Φ contains no contextual assertions

(ca) $(\forall x)(x \in \{x \mid \Phi\} \Leftrightarrow \Phi)$

Some useful classes are:

$$\mathbf{Val} = \{x \mid x \cong x\}$$

$$\mathbf{Nil} = \{\mathtt{nil}\}$$

$$\mathbf{Nat} = \{x \mid \mathtt{isnat?}(x) \cong \mathtt{t}\}$$

$$\mathbf{Atom} = \{x \mid \mathtt{isatom?}(x) \cong \mathtt{t}\}$$

$$\mathbf{Val} \to \mathbf{Val} = \{f \mid (\forall x \in X)(\exists y \in Y)\mathtt{app}(f, x) \cong y\}$$

An $k-ary$ class operator (scheme) $T[Z_1, \ldots, Z_k]$ is a class term T with free variables among $[Z_1, \ldots, Z_k]$. For example

$$X_1, \ldots, X_n \to Y = \{f \mid (\forall x_1 \in X_1, \ldots, x_n \in X_n)(\exists y \in Y)\mathtt{app}(f, x_1, \ldots, x_n) \cong y\}$$

is an $n+1$-ary class operator $F_n[X_1, \ldots, X_n, Y]$ whose range is classes that are function spaces. Note that $\mathbf{Val} \to \mathbf{Val} = F_1[\mathbf{Val}, \mathbf{Val}]$. A class operator $[X]$ is *monotonic* if $X \subseteq Y$ implies $T[X] \subseteq T[Y]$. Such operators can be used to define classes inductively (co-inductively) by taking minimal (maximal) fixed points.

4. Candidates for Visibility

We now turn to the problem of defining visibility. Given a memory context Γ and a set of values V over that memory ($V \subseteq \mathbb{V}_{\mathrm{Dom}(\Gamma)}$) we want to define the set of *visible values* relative to Γ and V, $Vis(\Gamma, V)$.

$Vis(\Gamma, V)$ should contain V, and should be closed under simple constructions that do not effect (but may read) Γ. There are two obvious candidates for visibility, which we shall call Vis_u and Vis_e (u for uniformly constructible, and e for evaluation constructible).

4.1. Evaluation Constructible

The evaluation constructible values are simply those values which may be constructed from expressions, parameterized by V, whose evaluation neither modifies nor enlarges memory. Namley: $Vis_e(\Gamma, V)$ is the set of values v returned by computations of the form $\Gamma; e^\sigma \overset{*}{\mapsto} \Gamma; v$ where the range of σ is contained in V.

Definition ($Vis_e(\Gamma, V)$):

$$Vis_e(\Gamma, V) = \{v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)} \mid (\exists X \in \mathbf{P}_\omega(\mathbb{X}))(\exists e \in \mathbb{E}_X)(\exists \sigma \in [X \to V])(\Gamma; e^\sigma \overset{*}{\mapsto} \Gamma; v)\}$$

4.2. Uniformly Constructible

The uniformly constructible values form a somewhat smaller set. Namely: $Vis_u(\Gamma, V)$ is the least set of values containing V and closed under: substitution into value expressions containing no variables bound by Γ (for example atoms); selection of components of pairs; and getting contents of those elements that are cells.

Definition ($Vis_u(\Gamma, V)$): $Vis_u(\Gamma, V)$ is the least set of values satisfying the following conditions (1–4):

(1) $V \subset Vis_u(\Gamma, V)$

(2) $\mathbb{V}_{Vis_u(\Gamma, V)} \subset Vis_u(\Gamma, V)$

(3) $\mathtt{pr}(v_0, v_1) \in Vis_u(\Gamma, V)$ implies $v_0, v_1 \in Vis_u(\Gamma, V)$

(4) $v \in Vis_u(\Gamma, V) \cap \mathrm{Dom}(\Gamma)$ implies $\Gamma(v) \in Vis_u(\Gamma, V)$

Note that (**2**) may be expanded to: $v \in \mathbb{V}_X$ and $\sigma \in [X \to Vis_u(\Gamma, V)]$ implies $v^\sigma \in Vis_u(\Gamma, V)$. To cast this definition into a form similar to the definition of evaluation constructible values, define the following collection of expressions:

Definition ($\mathbb{E}^{\mathbf{u}}_X$):

$$\mathbb{E}^{\mathbf{u}}_X = \mathbb{V}_X + \{\mathtt{get}, \mathtt{fst}, \mathtt{snd}\}(\mathbb{E}^{\mathbf{u}}_X)$$

The following lemma provides an equivalent definition of the uniformly constructible values.

Lemma (Vis_u):

$$Vis_u(\Gamma, V) = \{v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)} \mid (\exists X \in \mathbf{P}_\omega(\mathbb{X}))(\exists e \in \mathbb{E}^{\mathbf{u}}_X)(\exists \sigma \in [X \to V])(\Gamma; e^\sigma \overset{*}{\mapsto} \Gamma; v)\}$$

We should point out that there is a reasonable amount of leeway in the definition of $\mathbb{E}^{\mathbf{u}}_X$, we have given the smallest set needed, a larger set would be

$$\mathbb{E}^{\mathbf{u}}_X = \mathbb{V}_X + \mathbb{F}_n((\mathbb{E}^{\mathbf{u}}_X)^n)$$

4.3. Simple Consequences

The following three lemmas are simple consequences of the two definitions above, and so we omit their proofs.

Lemma (*Vis*-1): For both versions of visibility the following conditions hold:

(2a) $\mathbb{A} \subset \mathit{Vis}(\Gamma, V)$

(2pr) $v_1, v_2 \in \mathit{Vis}_u(\Gamma, V)$ implies $\mathtt{pr}(v_1, v_2) \in \mathit{Vis}(\Gamma, V)$

(2lam) $\lambda x.e \in \mathbb{L}_X$ and $\sigma \in [X \to \mathit{Vis}(\Gamma, V)]$ implies $(\lambda x.e)^\sigma \in \mathit{Vis}(\Gamma, V)$

Lemma (*Vis*-2): $\mathit{Vis}_u(\Gamma, V) \subseteq \mathit{Vis}_e(\Gamma, V)$ and in general the containment is proper. For example let $\Gamma = \{z := \mathtt{mk}(1)\}$ and let $V = \{\lambda x.z\}$. Then $z \in \mathit{Vis}_e(\Gamma, V)$ but $z \notin \mathit{Vis}_u(\Gamma, V)$.

Lemma (*Vis*-3): For both versions of visibility, under the conditions of the definition of $\Gamma \models_V \Phi[\sigma]$, the following are equivalent

1. $(\forall R \in \mathbb{R}_{\mathit{Vis}(\Gamma, V)})(\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]])$
2. $(\forall X)(\forall \sigma' \in \mathrm{Fmap}[X, V])(\forall R \in \mathbb{R}_X)(\Gamma[R^{\sigma'}[e_0^\sigma]] \updownarrow \Gamma[R^{\sigma'}[e_1^\sigma]])$

Thus we could equally well use either as the definition of $\Gamma \models_V (e_0 \cong e_1)[\sigma]$.

The theory developed in the remainder of this paper, is independent of which notion of visibility we choose. Further investigation is required to determine the right notion of visibility for our purposes.

5. Properties of Local Semantics

In this section we establish two results. The first result is a preservation theorem which gives sufficient conditions under which a formula valid in the non-local semantics remains valid in the new local semantics. The second result establishes that in the new local semantics the desired principle (**Ueq**) is indeed valid. Both results are independent of which definition of visibility used.

In general the validity of a statement in the non-local semantics does not imply its validity in the local semantics. However for a large class of formulas this inference is correct. These formulas correspond to what are traditionally called the first order positive formulas.

Definition ($\mathbb{W}^+$): The set of first order positive formulas, $\mathbb{W}^+$, is defined inductively as:

$$\mathbb{W}^+ = (\mathbb{E} \cong \mathbb{E}) + (\mathbb{W}^+ \wedge \mathbb{W}^+) + (\mathbb{W}^+ \vee \mathbb{W}^+) + (\mathbb{U}[\![\mathbb{W}^+]\!]) + (\forall \mathbb{X})(\mathbb{W}^+)$$

Theorem (Preservation): Assume that $\Phi \in \mathbb{W}^+$, $V \subseteq \mathbb{V}_{\mathrm{Dom}(\Gamma)}$, and $\sigma \in \mathrm{Fmap}[\mathbb{X}, \mathit{Vis}(\Gamma, V)]$. Then

$$(\Gamma \models \Phi[\sigma]) \quad \text{implies} \quad (\Gamma \models_V \Phi[\sigma])$$

The proof of this theorem is a simple induction on the complexity of Φ. Note that the counterexamples to (**Ueq**) for the non-local semantics given in §3.5 also demonstrate that the converse of (**Presevation**) and its generalization to arbitrary formulas are both false. In the appendix we have collected a plethora of principles established for the non-local semantics in [11]. Those that are positive remain valid

for the local semantics in virtue of the above theorem. There are other connections between the two semantics that are not accounted for by (**Preservation**) for example if Φ does not contain any contextual assertions then

$$(\Gamma \models \Phi[\sigma]) \quad \text{iff} \quad (\Gamma \models_{\mathrm{Dom}(\Gamma)} \Phi[\sigma])$$

is another obviously valid inference. Similarly $\Gamma \models_V \Phi[\sigma]$ is monotonic in V for first order positive formulas, Φ. The second result of this section is that regardless of which of the two candidates for the set *Vis* we choose, the principle (**Ueq**) is valid.

Theorem (Ueq): For $e_i \in \mathbb{E}$ and $V \subset \mathbb{V}_{\mathrm{Dom}(\Gamma)}$

$$\Gamma \models_V e_0 \cong e_1 \Rightarrow (\mathtt{let}\{x := e_0\}[\![\Phi]\!] \Leftrightarrow \mathtt{let}\{x := e_1\}[\![\Phi]\!])$$

Before proving (**Ueq**) we establish some useful notation. Assume that $\Gamma \models_V e_0 \cong e_1[\sigma]$, and without loss of generality that $\Gamma; e_j^\sigma \overset{*}{\mapsto} \Gamma_j; v_j$ (since if $\Gamma; e_j^\sigma$ is undefined the result is vacuously true). Also let $\sigma_j = \sigma\{x := v_j\}$, and $V_j = V \cup \{v_j\}$ for $j < 2$. Under these assumptions we define a notion of similarity $\sim$ between visible values, sets of visible values, value substitutions (of visible values) in the respective memories Γ_i as follows:

0. $v_0 \sim v_1$
1. $u \sim u$ for all $u \in \mathit{Vis}_{\{u,e\}}(\Gamma, V)$
2. $\sigma_0' \sim \sigma_1'$ iff $\mathrm{Dom}(\sigma_0') = \mathrm{Dom}(\sigma_1')$ and $(\forall x \in \mathrm{Dom}(\sigma_i'))(\sigma_0'(x) \sim \sigma_1'(x))$
3. If $\sigma_0' \sim \sigma_1'$, $e \in \mathbb{E}_{\mathrm{Dom}(\sigma_i')}$ in the Vis_e case, $e \in \mathbb{E}^{\{v\}}_{\mathrm{Dom}(\sigma_i')}$ in the Vis_u case , and $\Gamma_i; e^{\sigma_i'} \overset{*}{\mapsto} \Gamma_i; u_i$, then $u_0 \sim u_1$.
4. $X_0 \sim X_1$ iff $(\forall x_0 \in X_0)(\exists x_1 \in X_1)(x_0 \sim x_1) \wedge (\forall x_1 \in X_1)(\exists x_0 \in X_0)(x_0 \sim x_1)$

Note that the only clauses sensitive to the choice of *Vis* are reflexivity, (**1.**), and generation (**3.**). Some simple facts concerning this notion of similarity are the following:

Lemma (∼):

0. If $\sigma_0' \sim \sigma_1'$, then $v^{\sigma_0} \sim v^{\sigma_1}$ for $v \in \mathbb{V}_{\mathrm{Dom}(\sigma_i')}$
1. $\mathit{Vis}(\Gamma_0, V_0) \sim \mathit{Vis}(\Gamma_1, V_1)$
2. If $\sigma_0 \sim \sigma_1$, then $\Gamma_0; e^{\sigma_0} \updownarrow \Gamma_1; e^{\sigma_1}$
3. If $\sigma_0 \sim \sigma_1$, then $[K]^{\sigma_0}_{\Gamma_0, V_0} \sim [K]^{\sigma_1}_{\Gamma_1, V_1}$

Finally we prove the theorem.

Proof (Ueq): The proof is by induction on the complexity of Φ. Pick $e_0, e_1 \in \mathbb{E}$ and continue with the notation and assumptions above. Assume that $\Gamma_0 \models_{V_0} \Phi[\sigma_0]$.

Case: Φ is $e_a \cong e_b$. We need to show that $\Gamma_1 \models_{V_1} e_a \cong e_b[\sigma_1]$. Now by (*Vis*-**3**) we need only show that for any $\sigma_1' \in \mathrm{Fmap}[X, V_1]$ and any $R \in \mathbb{R}_{\mathrm{Dom}(\sigma')}$

$$(\Gamma_1[R^{\sigma_1'}[e_a^{\sigma_1}]] \updownarrow \Gamma_1[R^{\sigma_1'}[e_b^{\sigma_1}]])$$

Now:

$$\begin{array}{lll}
\Gamma_1[R^{\sigma_1'}[e_a^{\sigma_1}]] & \updownarrow & \Gamma_1[R[e_a]^{\sigma_1' \cup \sigma_1}] \qquad \text{assuming simple hygiene conditions} \\
& \updownarrow & \Gamma_0[R[e_a]^{\sigma_0' \cup \sigma_0}] \\
& & \quad \text{by } (\sim\!\mathbf{.2}) \text{ and } \sigma_0' \text{ is obtained by replacing } v_1 \text{ by } v_0 \text{ in } \sigma_1' \\
& \updownarrow & \Gamma_0[R^{\sigma_0'}[e_a^{\sigma_0}]] \\
& \updownarrow & \Gamma_0[R^{\sigma_0'}[e_b^{\sigma_0}]] \\
& \updownarrow & \Gamma_0[R[e_b]^{\sigma_0' \cup \sigma_0}] \\
& \updownarrow & \Gamma_1[R[e_b]^{\sigma_1' \cup \sigma_1}] \\
& \updownarrow & \Gamma_1[R^{\sigma_1'}[e_b^{\sigma_1}]]
\end{array}$$

Case: Φ is $\Phi_0 \Rightarrow \Phi_1$. Assume $\Gamma_1 \models_{V_1} \Phi_0[\sigma_1]$ and show $\Gamma_1 \models_{V_1} \Phi_1[\sigma_1]$.

$$\Gamma_1 \models_{V_1} \Phi_0[\sigma_1] \Rightarrow \Gamma_0 \models_{V_0} \Phi_0[\sigma_0] \Rightarrow \Gamma_0 \models_{V_0} \Phi_1[\sigma_0] \Rightarrow \Gamma_1 \models_{V_1} \Phi_1[\sigma_1]$$

Case: Φ is $(\forall y)\Phi_0$. Assume $v^1 \in \mathit{Vis}(\Gamma_1, V_1)$ and show $\Gamma_1 \models_{V_1} \Phi_0[\sigma_1\{y := v^1\}]$. Now, $v^1 \in \mathit{Vis}(\Gamma_1, V_1)$ implies that there is a $v^0 \in \mathit{Vis}(\Gamma_0, V_0)$ such that $v^0 \sim v^1$ by $(\sim\!\mathbf{.1})$. By hypothesis $\Gamma \models_{V_0} \Phi_0[\sigma_0\{y := v^0\}]$ and $\sigma_0\{y := v^0\} \sim \sigma_1\{y := v^1\}$. Thus the result follows by $(\sim\!\mathbf{.2})$.

Case: Φ is $\mathtt{let}\{y := e\}[\![\Phi_0]\!]$. In this case we use a simple trick.

$$\begin{array}{ll}
& \Gamma_0 \models_{V_0} \Phi[\sigma_0] \\
\Leftrightarrow & \Gamma \models_V \mathtt{let}\{x := e_0\}\mathtt{let}\{y := e\}[\![\Phi_0]\!][\sigma] \\
\Leftrightarrow & \Gamma \models_V \mathtt{let}\{z := \mathtt{let}\{x := e_0\}\mathtt{pr}(x, e)\}[\![\Phi_0']\!] \\
& \qquad \text{where } \Phi_0' = \Phi_0^{\{x:=\mathtt{fst}(z), y:=\mathtt{snd}(z)\}} \\
\Leftrightarrow & \Gamma \models_V \mathtt{let}\{z := e_0'\}[\![\Phi_0']\!] \qquad \text{where } e_0' = \mathtt{let}\{x := e_0\}\mathtt{pr}(x, e) \\
\Leftrightarrow & \Gamma \models_V \mathtt{let}\{z := e_1'\}[\![\Phi_0']\!] \qquad \text{by the IH where } e_1' = \mathtt{let}\{x := e_1\}\mathtt{pr}(x, e) \\
\Leftrightarrow & \Gamma \models_V \mathtt{let}\{x := e_1\}\mathtt{let}\{y := e\}[\![\Phi_0]\!][\sigma] \\
\Leftrightarrow & \Gamma_1 \models_{V_1} \Phi[\sigma_1]
\end{array}$$

Case: Φ is $v \in K$. This case is immediate from $(\sim\!\mathbf{.3})$

Case: Φ is $(\forall X)\Phi_0$. This case is identical to the individual variable case.

□**Ueq**

6. Persistence

The contextual assertions allow various modalities to be defined. Of interest here is the modality $\Box\Phi$ (read true in all reachable memories). $\Box$ essentially expresses validity, and is easily defined using contextual assertions as follows.

Definition ($\Box$): $\Box\Phi$ abbreviates:

$$(\forall f, x)\mathtt{seq}(\mathtt{app}(f, x), [\![\Phi]\!]) \qquad \text{where } f, x \text{ not free in } \Phi$$

We say a formula Φ is persistent if $\Phi \Rightarrow \boxdot(\Phi)$. Some examples of persistent atomic formulae are:

$$\vartheta(x_1, \ldots, x_n) \cong z$$

where $\vartheta \in \{\texttt{isatom?}, \texttt{ispr?}, \texttt{iscell?}, \texttt{eq}, \texttt{mk}\}$. The recognizers are easily seen to be persistent because no operation can change an atom into a pair, or a pair into an atom. Similarly `eq` is persistent since no operation can change the identity of values. In fact the only properties of values that can be altered by operations are the contents of cells. Thus the formula $\texttt{get}(x) \cong \texttt{nil}$ is not persistent, since the cell denoted by x may have its contents altered. Note that (free occurences of) x itself will always refer to the same cell. Also, $\texttt{get}(x) \cong y$ iff $\texttt{set}(x, y) \cong \texttt{nil}$, thus `set` is not persistent. The fact that $\texttt{mk}(x) \cong y$ is persistent is simply due to the fact that it is persistently false.

Lemma (Sca): Let $\mathrm{Traps}(U) \subseteq \{x_1, \ldots, x_n\}$. Then

(i) $\boxdot((\forall x_1 \ldots x_n)\Phi) \Rightarrow \boxdot(U[\![\Phi]\!])$

(ii) $\boxdot((\forall x_1 \ldots x_n)(\Phi_0 \Rightarrow \Phi_1)) \Rightarrow ((\boxdot(U[\![\Phi_0]\!]) \Rightarrow \boxdot(U[\![\Phi_1]\!]))$

(iii) $\boxdot((\forall x_1 \ldots x_n)(\Phi_0 \Leftrightarrow \Phi_1)) \Rightarrow (\boxdot(U[\![\Phi_0]\!]) \Leftrightarrow \boxdot(U[\![\Phi_1]\!]))$

Persistence is closed under conjunction.

An n-ary relation is a class term S such that $S \subseteq \mathbf{Val}^n$. Where

$$\mathbf{Val}^n = \{x \mid (\exists y_1, \ldots, y_n \in \mathbf{Val})(x = \texttt{pr}(y_1, \ldots, y_n))\}.$$

If $\Phi(x_1, \ldots, x_n)$ is a formula, then the n-ary relation S defined by the formula is:

$$\texttt{pr}(y_1, \ldots, y_n) \in S \Leftrightarrow \Phi(y_1, \ldots, y_n)$$

Definition (Persistent Relation): An n-ary relation, S, is persistent if

$$S(y_1, \ldots, y_n) \Rightarrow \boxdot(S(y_1, \ldots, y_n)).$$

Lemma (Persistent Relation): A relation defined by persistent formulas is persistent.

7. Principles for Reasoning about Behaviors and Objects

We consider two general principles: one for establishing equivalence of essentially functional behaviors (abstractions whose application has no visible effects), and one for establishing equivalence of objects. In the latter case we restrict attention to objects with statically allocated local store and first-order interactions. Informally, an object has first-order interactions if it handles only first order messages, ignoring any cell and function components, and returns only first-order replies. These principles formalize simple cases of the general simulation induction principle given in [13]. Combined with the theorem (**b2o**) of the next section relating behaviors and objects these principles provide tools for reasoning about an important class of objects and transformations.

7.1. Behavior Simulation Induction

An essentially functional behavior is an abstraction whose application has no visible effects on memory. The space of essentially functional behaviors is a definable class:

$$\{f \mid (\forall x \in \mathbf{Val})(\downarrow \mathtt{app}(f,x) \Rightarrow (\exists y \in \mathbf{Val})(\mathtt{app}(f,x) \cong y))\}$$

and is larger than $\mathbf{Val} \to \mathbf{Val}$ which was defined in §3.7. To prove that essentially functional behaviors are equivalent we introduce the notion of behavior simulation and the corresponding induction principle (**BSI**). We first present these notions informally, then we indicate how they can be expressed formally within VTLoE, using classes.

Definition (Similarity - $\sim_S$): Let $S \subseteq \mathbb{L} \times \mathbb{L}$. Define the induced relation on values $\sim_S$ as the least equivalence relation on value expressions such that

(1) $\rho_0 S \rho_1$ implies $\rho_0 \sim_S \rho_1$, for $\rho_0, \rho_1 \in \mathbb{L}$;

(2) $v_i \sim_S v_i'$ for $i < 2$ implies $\mathtt{pr}(v_0, v_1) \cong \mathtt{pr}(v_0', v_1')$; and

(3) $v_0 \sim_S v_1$ and $f \in \mathbf{Val} \to \mathbf{Val}$ implies $f(v_0) \sim_S f(v_1)$.

Definition (Behavior Simulation): A relation $S \subseteq \mathbb{L} \times \mathbb{L}$ is a behavior simulation if for any ρ_0, ρ_1, v_0, v_1 such that $S(\rho_0, \rho_1) \wedge v_0 \sim_S v_1$ one of the following two conditions holds:

(u) $\Uparrow \mathtt{app}(\rho_j, v_j)$ for $j < 2$

(d) there are v_0', v_1' such that $v_0' \sim_S v_1'$, and $\mathtt{app}(\rho_j, v_j) \cong v_j'$ for $j < 2$

Theorem (BSI): If $S \subseteq \mathbb{L} \times \mathbb{L}$ is a behavior simulation, then

$$S(\rho_0, \rho_1) \Rightarrow \rho_0 \cong \rho_1$$

Observe that $\sim_S$ can be defined within VTLoE using class operators (cf. [11]), as follows. We begin by assuming that S is a subclass of

$$\{x \mid \mathtt{ispr}?(x) \cong \mathtt{t} \wedge \mathtt{islam}?(\mathtt{fst}(x)) \cong \mathtt{t} \wedge \mathtt{islam}?(\mathtt{fst}(\mathtt{snd}(x))) \cong \mathtt{t}\}.$$

Then $\sim_S$ will be the smallest class Z which contains S, satisfies the conditions for an equivalence relation, and satisfies:

$$\mathtt{pr}(x_0, y_0) \in Z \wedge \mathtt{pr}(x_1, y_1) \in Z \Rightarrow \mathtt{pr}(\mathtt{pr}(x_0, y_0), \mathtt{pr}(x_1, y_1)) \in Z$$

$$\mathtt{pr}(x_0, y_0) \in Z \wedge f \in \mathbf{Val} \to \mathbf{Val} \Rightarrow \mathtt{pr}(f(v_0), f(v_1)) \in Z$$

Similarly both the definition of behavior simulation, and the theorem (**BSI**) can be expressed within VTLoE using the class apparatus. A trivial example of the use of this principle is to establish the equivalence of ρ_0 and ρ_1:

$$\rho_0 = \lambda x.\mathtt{app}(I, x) \quad \text{and} \quad \rho_1 = \lambda x.\mathtt{seq}(\mathtt{app}(I, x), \mathtt{app}(I, x))$$

where I is the identity $\lambda z.z$

In this case we simply let $S = \{\mathtt{pr}(\rho_0, \rho_1)\}$ and invoke (**BSI**).

7.2. First-orderness

A value is first-order (written $IsFo(x)$) if it is an atom or a pair of first-order values. Two values have the same first-order part (written $x =_{\text{fo}} y$) if they are both functions, both cells, both pairs with the same first-order components, or are the same atom. The first-order projection of a value $\mathtt{fop}(x)$ is obtained by replacing all occurrences of cells and functions by the atom `nil`.

Definition ($IsFo(x)$, $x =_{\text{fo}} y$, fop):

$$\mathtt{isfo?} = \mathtt{Y}(\lambda f.\lambda x.\mathtt{if}(\mathtt{ispr?}(x), \mathtt{and}(f(\mathtt{fst}(x)), f(\mathtt{snd}(x))), \mathtt{isatom?}(x)))$$

$$\mathtt{fop} = \mathtt{Y}(\lambda f.\lambda x.\mathtt{if}(\mathtt{ispr?}(x), \mathtt{pr}(f(\mathtt{fst}(x)), f(\mathtt{snd}(x))), \mathtt{if}(\mathtt{isatom?}(x), x, \mathtt{nil})))$$

$$\begin{aligned}\mathtt{foeq} = \mathtt{Y}(\lambda f.\lambda x, y.\\ &\mathtt{cond}([\mathtt{and}(\mathtt{islam?}(x), \mathtt{islam?}(y)) \Rightarrow \mathtt{t}],\\ &\quad [\mathtt{and}(\mathtt{iscell?}(x), \mathtt{iscell?}(y)) \Rightarrow \mathtt{t}],\\ &\quad [\mathtt{and}(\mathtt{ispr?}(x),\\ &\qquad \mathtt{ispr?}(y),\\ &\qquad f(\mathtt{fst}(x), \mathtt{fst}(y)),\\ &\qquad f(\mathtt{snd}(x), \mathtt{snd}(y))) \Rightarrow \mathtt{t}],\\ &\quad [\mathtt{and}(\mathtt{isatom?}(x), \mathtt{isatom?}(y)) \Rightarrow \mathtt{eq}(x, y)],\\ &\quad [\mathtt{t} \Rightarrow \mathtt{nil}]))\end{aligned}$$

$$IsFo(x) \Leftrightarrow \mathtt{isfo?}(x) \cong \mathtt{t}$$

$$x =_{\text{fo}} y \Leftrightarrow \mathtt{foeq}(x, y) \cong \mathtt{t}$$

Note that $\mathtt{foeq}(x, y) \cong \mathtt{t}$ implies that $\mathtt{fop}(x) \cong \mathtt{fop}(y)$, however the converse is false, since information concerning whether something is a cell or abstraction is lost.

7.3. FO Object Simulation Induction

We begin with a simple version of the object induction principle that treats objects with k-ary store. An object with k-ary store is a value O such that

$$O \cong \lambda z_1, \ldots z_k.\lambda m.(O(z_1, \ldots z_k)(m))$$

The idea is that $z_1, \ldots z_k$ name cells of the local store of the object, and m is the message parameter. Let $\mathbf{z}$ be a sequence of distinct variables and let $\mathbf{v}$ be a sequence of value expressions of the same length. We write $A^{\mathbf{z}}_{\mathbf{v}}$ to abbreviate the memory context $\{z_i := \mathtt{mk}(v_i) \mid 1 \leq i \leq k\}$.

Intuitively two objects are equivalent if when started in similar states, they always give the same reply to a message and move to similar states, for some appropriate notion of similar state. As a simple case we define First Order Object simulation (FOO simulation). Let O_0, O_1 be objects with k_j-ary store for $j < 2$. Informally we say that a $k_0 + k_1$-ary relation S is a FOO simulation for O_0, O_1 if when started in S-similar stores and sent any message (applied to any argument)

either both objects diverge, or both objects use only the first order part of the argument, and both return the same (first-order) value, and their updated stores remain S-similar.

Definition (FOO Simulation): Let O_j be objects with k_j-ary store for $j < 2$. Let $\mathbf{y}_j$, $\mathbf{z}_j$ be sequences of distinct variables of length k_j for $j < 2$. An $k_0 + k_1$-ary relation S is a simulation relation for O_0, O_1 if S is persistent, and the following holds (for first-order values of any free variables):

$$
\begin{array}{l}
S(\mathbf{y}_0, \mathbf{y}_1) \;\Rightarrow\; (\forall x)(\exists \mathbf{y}_0', \mathbf{y}_1', r) \\
\quad S(\mathbf{y}_0', \mathbf{y}_1') \wedge \\
\quad \mathit{IsFo}(r) \wedge \\
\quad\quad \wedge_{j<2} \Uparrow A^{\mathbf{z}_j}_{\mathbf{y}_j'}[O_j(\mathbf{z}_j)(x)] \vee \\
\quad\quad\quad \wedge_{j<2} A^{\mathbf{z}_j}_{\mathbf{y}_j}[\![O_j(\mathbf{z}_j)(x) \cong O_j(\mathbf{z}_j)(\mathtt{fop}(x))]\!] \wedge \\
\quad\quad\quad \wedge_{j<2} A^{\mathbf{z}_j}_{\mathbf{y}_j}[\![O_j(\mathbf{z}_j)(x) \cong \mathtt{seq}(\mathtt{set}(\mathbf{z}_j, \mathbf{y}_j'), r)]\!]
\end{array}
$$

Theorem (FOO Simulation Induction (FOOSI)): Let O_j be objects with k_j-ary store, $\mathbf{y}_j$, $\mathbf{z}_j$ be sequences of distinct variables of length k_j for $j < 2$. Let S be an $k_0 + k_1$-ary relation. If S is a simulation relation for O_0 and O_1, then (for first-order values of any free variables)

$$S(\mathbf{y}_0, \mathbf{y}_1) \;\Rightarrow\; A^{\mathbf{z}_0}_{\mathbf{y}_0}[O_0(\mathbf{z}_0)] \cong A^{\mathbf{z}_1}_{\mathbf{y}_1}[O_1(\mathbf{z}_1)]$$

As a simple example of the use of (**FOOSI**) we establish the equivalence of two different *fibonacci* objects, $\mathtt{F}_0$ and $\mathtt{F}_1$. The first fibonacci object $\mathtt{F}_0$ has unary store while the second fibonacci object has binary store. Their actual definitions are:

$$
\begin{array}{l}
\mathtt{F}_0 = \lambda z.\lambda m.\mathtt{let}\{x := \mathtt{get}(z)\}\mathtt{seq}(\mathtt{set}(z, x+1), \mathtt{fib}(x)) \\
\mathtt{F}_1 = \lambda z_1, z_2 \lambda m.\mathtt{let}\{x_1 := \mathtt{get}(z_1)\}\mathtt{let}\{x_2 := \mathtt{get}(z_2)\} \\
\qquad\qquad \mathtt{seq}(\mathtt{set}(z_1, x_2), \mathtt{set}(z_2, x_1 + x_2), \mathtt{get}(z_2))
\end{array}
$$

We will show that for $n > 2$ that

$$
\begin{array}{l}
\mathtt{let}\{z := \mathtt{mk}(n)\}\mathtt{F}_0(z) \cong \mathtt{let}\{z_1 := \mathtt{mk}(\mathtt{fib}(n-2))\} \\
\qquad\qquad\qquad\qquad \mathtt{let}\{z_2 := \mathtt{mk}(\mathtt{fib}(n-1))\}\mathtt{F}_1(z_1, z_2)
\end{array}
$$

where $\mathtt{fib}$ is the usual fibonacci operation $\mathbb{N} \to \mathbb{N}$ given by

$$\mathtt{fib}(n) = \begin{cases} 1 & \text{if } 0 \le n \le 1 \\ \mathtt{fib}(n-2) + \mathtt{fib}(n-1) & \text{otherwise.} \end{cases}$$

The first step is to define the the 3-ary relation S:

$$S = \{\mathtt{pr}(n, \mathtt{fib}(n-2), \mathtt{fib}(n-1)) \;\big|\; n \in \mathbb{N} \wedge 2 \le n\}.$$

Clearly S is persistent. To see that it is a simulation relation for $\mathtt{F}_0$ and $\mathtt{F}_1$ observe that:

$\mathtt{pr}(n, n_1, n_2) \in S \;\Rightarrow\; \mathtt{pr}(n+1, n_2, n_1+n_2) \in S$

$IsFo(n_1 + n_2)$

$A^z_n[\![\mathtt{F}_0(z)(x) \cong \mathtt{F}_0(z)(\mathtt{nil}) \cong \mathtt{F}_0(z)(\mathtt{fop}(x))]\!]$

$A^{z_1,z_2}_{n_1,n_2}[\![\mathtt{F}_1(z_1,z_2)(x) \cong \mathtt{F}_1(z)_1, z_2(\mathtt{nil}) \cong \mathtt{F}_1(z_1,z_2)(\mathtt{fop}(x))]\!]$

$A^z_n[\![\mathtt{F}_0(z)(x) \cong \mathtt{seq}(\mathtt{set}(z, n+1), \mathtt{fib}(n))]\!]$

$A^{z_1,z_2}_{n_1,n_2}[\![\mathtt{F}_1(\mathbf{z}_j)(x) \cong \mathtt{seq}(\mathtt{set}(z_1, n_2), \mathtt{set}(z_2, n_1+n_2), n_1+n_2)]\!]$

Thus the desired result follows from (**FOOSI**).

8. Specifications, Behaviors, and Objects

We specify an object by a set of local parameters, a message parameter, and a sequence of message handlers. A message handler consists of a test function, a reply function and a list of updating functions (one for each parameter). The functions take as arguments the message and current value of the local parameters. Upon receipt of a message, the first handler whose test is true is invoked. The local parameters are updated according to the update expressions and the reply is computed by the reply function. A specification S with k local parameters $\mathbf{y} = y_1, \ldots, y_k$, message parameter m, and ith message handler with test function t_i, reply function r_i, and updating functions $u_{i,j}$ for $1 \le j \le k$ is written in the form

$\mathrm{S} = (\mathbf{y})(m)$

[

$\ldots$

$t_i(\mathbf{y}, m) \Rightarrow r_i(\mathbf{y}, m), u_{i,1}(\mathbf{y}, m), \ldots, u_{i,k}(\mathbf{y}, m)$

$\ldots$

]

For this to be an admissible object specification, we require that the test, updating, and reply functions are total and have no (visible) effect. For specification of objects with first-order behavior, we further require that these functions depend only on the first-order projection of the message and that the reply function returns first-order values. An alternative would be to force first-order interpretation by wrapping projections around messages and return values in the associated programs.

We associate to each specification S two programs: the local behavior function beh_{S}, and the canonical specified object, obj_{S}. The local behavior corresponding to S is purely functional. It is a closure with local parameters corresponding to those of the specification. When applied to a message, the behavior function corresponding to the updated local parameters is returned along with the reply to the message. If there is shared behavior then the current state of the shared behavior must be passed

as an argument along with the message proper, and the updated shared behavior must be returned as well. The object specified by S has the local parameters stored in its local memory. When applied to a message, the object updates the local parameter memory and returns only the reply.

Definition (beh_{S}):

$$
\begin{array}{l}
beh_{\mathrm{S}} = \mathtt{Y}(\lambda b.\lambda \mathbf{y}.\lambda m. \\
\quad \mathtt{cond}(\\
\qquad \ldots \\
\qquad [t_i(\mathbf{y}, m) \Rightarrow \mathtt{pr}(b(u_{i,1}(\mathbf{y}, m), \ldots, u_{i,k}(\mathbf{y}, m)), r_i(\mathbf{y}, m))] \\
\qquad \ldots \\
\qquad [\mathtt{t} \Rightarrow \mathtt{pr}(b(\mathbf{y}), \mathtt{nil})] \\
\qquad)\,)
\end{array}
$$

Definition (obj_{S}):

$$
\begin{array}{l}
obj_{\mathrm{S}} = \lambda \mathbf{z}.\lambda m. \\
\quad \mathtt{let}\{y_1 := \mathtt{get}(z_1)\} \ldots \mathtt{let}\{y_k := \mathtt{get}(z_k)\} \\
\qquad \mathtt{cond}(\\
\qquad\quad \ldots \\
\qquad\quad [t_i(\mathbf{y}, m) \Rightarrow \mathtt{seq}(\mathtt{set}(z_1, u_{i,1}(\mathbf{y}, m)), \ldots, \mathtt{set}(z_k, u_{i,k}(\mathbf{y}, m)), r_i(\mathbf{y}, m))] \\
\qquad\quad \ldots \\
\qquad\quad [\mathtt{t} \Rightarrow \mathtt{nil}] \\
\qquad\quad)
\end{array}
$$

There is a protocol transforming operation *b2o* (behavior-to-object) that maps the behavior corresponding to S to the object specified by S. *b2o* allocates a cell and stores the behavior function there. When applied to a message it looks up the behavior, applies it to the message, stores the new behavior, and returns the reply. Behavior functions and objects generalize the notions of reusable and onetime streams studied in [13]. The point of having two forms is that one can often compose behaviors and reason about them more easily than the corresponding objects. Using the connections established by the abstract specification and the protocol transformation one can obtain objects corresponding to transformed behaviors. Methods developed in [13,17] can be extended to further simplify and optimize the resulting objects. The point is that different representations are better suited for carrying out different sorts of transformations, and one needs to have appropriate representations at hand and be able to move from one representation to another in a semantically sound manner.

Definition (*b2o*):

$$b2o = \lambda beh.b2ox(\mathtt{mk}(beh))$$

$$b2ox = \lambda z.\lambda m.\mathtt{let}\{\mathtt{pr}(beh, r) := \mathtt{get}(z)(m)\}\mathtt{seq}(\mathtt{set}(z, beh), r)$$

where $\mathtt{let}\{\mathtt{pr}(beh, r) := \mathtt{get}(z)(m)\}$ is binding by ML-style pattern matching. The relation between objects and behaviors, corresponding to the same specification, is captured by the following theorem.

Theorem (B2o): If S is an object specification, as above, then

$$b2o(beh_{\mathrm{S}}(\mathbf{y})) \cong \mathtt{let}\{z_1 := \mathtt{mk}(y_1)\} \ldots \mathtt{let}\{z_k := \mathtt{mk}(y_k)\} obj_{\mathrm{S}}(\mathbf{z})$$

Proof (B2o): The proof is by (**FOOSI**). First note that $b2o(beh_{\mathrm{S}}(\mathbf{y})) \cong \mathtt{let}\{z_0 := beh_{\mathrm{S}}(\mathbf{y})\} b2ox(z_0)$. The relation $S(y_0, \mathbf{y})$ is defined to be $\Box(y_0 \cong beh_{\mathrm{S}}(\mathbf{y}))$. The precise assumptions on the test, reply, and updating functions are the following:

$t_i(\mathbf{y}, m) \cong t_i(\mathbf{y}, \mathtt{fop}(m)) \in \mathbf{Val}$

$r_i(\mathbf{y}, m) \cong r_i(\mathbf{y}, \mathtt{fop}(m)) \in \mathbf{Atom}$

$u_{i,j}(\mathbf{y}, m) \cong u_{i,j}(\mathbf{y}, \mathtt{fop}(m)) \in \mathbf{Val}$

These assumptions provide values for the existential variables and assure dependence only on the first-order part of m. Assume $t_i(\mathbf{y}, m)$ is true. Let r be (the value of) $r_i(\mathbf{y}, m)$, and $\mathbf{y}'$ be $[u_{i,j}(\mathbf{y}, m) \mid 1 \leq j \leq k]$. Then by equational reasoning inside the combined memory contexts

$$b2ox(z_0)(m) \cong \mathtt{seq}(\mathtt{set}(z_0, beh_{\mathrm{S}}(\mathbf{y}')), r)$$

$$obj_{\mathrm{S}}(\mathbf{z}) \cong \mathtt{seq}(\mathtt{set}(z_j, y'_j)_{1 \leq j \leq k}, r)$$

$\Box$**B2o**

9. Examples of Object Transformations

In this section we give examples illustrating a variety of basic transformations on objects. We have chosen very simple objects, so as not to obscure the mechanisms by a lot of detail. A more substantial example that can be treated by these methods is composition and specialization of a window editor described in [14].

9.1. Constant Lifting

$Bp(d, p)$ is the behavior of a bounded point, for simplicity we consider the one-dimensional case. Bp maintains the invariants $p, d \in \mathbb{N}$ and $0 \leq p \leq d$. It accepts messages `L` (move left), `R` (move right), and any other message is treated as a request for the current position. Op defines the corresponding object, and Op' is obtained from Op by the transformation that replaces a local cell with constant contents by that contents.

$$\begin{aligned}
Bp = \mathtt{Y}(\lambda b.\lambda d, p.\lambda m.\\
&\mathtt{if}(\mathtt{eq}(m, \mathtt{L}),\\
&\quad \mathtt{if}(0 < p, \mathtt{pr}(b(d, p-1), \mathtt{t}), \mathtt{pr}(b(d, p), \mathtt{nil})),\\
&\mathtt{if}(\mathtt{eq}(m, \mathtt{R}),\\
&\quad \mathtt{if}(p < d, \mathtt{pr}(b(d, p+1), \mathtt{t}), \mathtt{pr}(b(d, p), \mathtt{nil})),\\
&\quad \mathtt{pr}(b(d, p), p))))
\end{aligned}$$

$$
\begin{aligned}
Op = \lambda z_d, z_p.\lambda m.&\texttt{let}\{d := \texttt{get}(z_d)\}\texttt{let}\{p := \texttt{get}(z_p)\}\\
&\texttt{if}(\texttt{eq}(m, \texttt{L}),\\
&\quad \texttt{if}(0 < p, \texttt{seq}(\texttt{set}(z_p, p-1), \texttt{t}), \texttt{nil}),\\
&\texttt{if}(\texttt{eq}(m, \texttt{R}),\\
&\quad \texttt{if}(p < d, \texttt{seq}(\texttt{set}(z_p, p+1), \texttt{t}), \texttt{nil}),\\
&\quad p))\\
Op' = \lambda d, z_p.\lambda m.&\texttt{let}\{p := \texttt{get}(z_p)\}\\
&\texttt{if}(\texttt{eq}(m, \texttt{L}),\\
&\quad \texttt{if}(0 < p, \texttt{seq}(\texttt{set}(z_p, p-1), \texttt{t}), \texttt{nil}),\\
&\texttt{if}(\texttt{eq}(m, \texttt{R}),\\
&\quad \texttt{if}(p < d, \texttt{seq}(\texttt{set}(z_p, p+1), \texttt{t}), \texttt{nil}),\\
&\quad p))
\end{aligned}
$$

Theorem (Constant Lifting):

$$b2o(Bp(d,p)) \cong A^{z_d,z_p}_{d,p}[Op(z_d, z_p)] \cong A^{z'_p}_{d,p}[Op'(d, z'_p)]$$

Proof : The first equivalence is by (**b2o**), and simplification using $(\forall m)(e \cong e') \Rightarrow \lambda m.e \cong \lambda m.e'$. For the second equivalence we can use FOSI-I. Define $S(y_{0,0}, y_{0,1}, y_{1,0})$ by $d = y_{0,0} \in \mathbb{N}$, $p = y_{0,1} \in \mathbb{N}$, $y_{0,1} = y_{1,0}$), and $0 \leq p \leq d$. The existential variables are determined by cases on m and p

	$y'_{0,0}$	$y'_{0,1}$	$y'_{1,0}$	r
$m = \texttt{L} \wedge p > 0$	d	$p-1$	$p-1$	$\texttt{t}$
$m = \texttt{L} \wedge p = 0$	d	p	p	$\texttt{nil}$
$m = \texttt{R} \wedge p < d$	d	$p+1$	$p+1$	$\texttt{t}$
$m = \texttt{L} \wedge p = d$	d	p	p	$\texttt{nil}$
otherwise	d	p	p	p

Now we must show the following.

(1) S is persistent and invariant – $S(y_{0,0}, y_{0,1}, y_{1,0}) \Rightarrow A^{z_d,z_p,z'_p}_{d,p,p}[\![S(y'_{0,0}, y'_{0,1}, y'_{1,0})]\!]$

(2) first-orderness in argument
$A^{z_d,z_p}_{d,p}[\![Op(z_d, z_p)(m) \cong Op(z_d, z_p)(\texttt{fop}(m))]\!]$
$A^{z'_p}_{d,p}[\![Op'(d, z'_p)(m) \cong Op'(d, z'_p)(\texttt{fop}(m))]\!]$

(3) similar effects and values – if $S(y_{0,0}, y_{0,1}, y_{1,0})$, then
$A^{z_d,z_p}_{d,p}[\![Op(z_d, z_p)(m) \cong \texttt{seq}(\texttt{set}(z_d, y'_{0,0}), \texttt{set}(z_y, y'_{0,1}), r)]\!]$
$A^{z'_p}_{d,p}[\![Op'(d, z'_p)(m) \cong \texttt{seq}(\texttt{set}(z'_y, y'_{1,1}), r)]\!]$

For (1) we use the general property that membership in $\mathbb{N}$ and arithmetic relations are persistent. The case analysis on m gives the invariance of S.
For (2) we use $x \in \mathbb{A}$ implies $\texttt{eq}(m, x) \cong \texttt{eq}(\texttt{fop}(m), x)$ which is an easy consequence of pure $\cong$ reasoning.

For (3) we consider the case $m = \mathtt{L}$ and $p > 0$. The remaining cases are similar. We use persistence of S to move it inside the memory context, and facts about memory operations, (**ca,R**), to evaluate `let`s, and `if`s inside the memory context.

□

9.2. First-Order Streams

A first-order stream is a behavior that, for any message returns a pair consisting of a first-order stream and a first-order element. This can be formalized in VTLoE using the maximal fixed point operator. Define

$$\mathbf{FOV} = \{x \mid \mathit{IsFo}(x)\}.$$

The class of first-order streams is the largest class satisfying

$$\mathbf{FoStr} \equiv \mathbf{Nil} \to [\mathbf{FoStr} \times \mathbf{FOV}].$$

More precisely, using the maximum-fixed-point operator.
Definition (First Order Streams):

$$K_0 \to K_1 = \{o \mid (\forall x \in K_0)(o(x) \in K_1)\}$$

$$T_{\mathbf{FoStr}}[X] = [\mathbf{Nil} \to [X \times \mathbf{FOV}]]$$

$$\mathbf{FoStr} = T_{\mathbf{FoStr}}[X]^{\cup}$$

Where $T[X]^{\cup}$ is the maximal fixed point of the monotonic class operator $T[X]$ [23,11].

9.3. Protocol Transformation

`Bc` is the behavior of an object that increments a counter for each non-`nil` message, then sends `nil` to its local object, assumed to be a stream. It treats `nil` as a request for the current value of the local counter. `Oc` is the corresponding object code. `Ocx` is obtained from `Oc` by a protocol transformation – replacing a behavior by a corresponding object. Having carried out this transformation, the object parameter can be replaced by any equivalent object.

$$\begin{aligned}
&\mathtt{Bc} = \mathtt{Y}(\lambda b.\lambda q, s.\lambda m.\\
&\qquad \mathtt{if}(m,\\
&\qquad\quad \mathtt{let}\{\mathtt{pr}(s,r) := \mathtt{app}(s,\mathtt{nil})\}\mathtt{pr}(b(q+1,s),r),\\
&\qquad\quad \mathtt{pr}(b(q,s),q)))\\
&\mathtt{Oc} = \lambda z_q, z_s.\lambda m.\\
&\qquad\quad \mathtt{let}\{q := \mathtt{get}(z_q)\}\mathtt{let}\{s := \mathtt{get}(z_s)\}\\
&\qquad\quad\quad \mathtt{if}(m,\\
&\qquad\quad\quad\quad \mathtt{let}\{\mathtt{pr}(s,r) := \mathtt{app}(s,\mathtt{nil})\}\\
&\qquad\quad\quad\quad\quad \mathtt{seq}(\mathtt{set}(z_q, q+1), \mathtt{set}(z_s, s), r),\\
&\qquad\quad\quad\quad q)
\end{aligned}$$

$$\begin{array}{l}\texttt{Ocx} = \lambda z_q, o_s.\lambda m.\\ \qquad\qquad \texttt{let}\{q := \texttt{get}(z_q)\}\\ \qquad\qquad\quad \texttt{if}(m,\\ \qquad\qquad\qquad \texttt{seq}(\texttt{set}(z_q, q+1), \texttt{app}(o_s, \texttt{nil})),\\ \qquad\qquad\qquad q)\end{array}$$

Theorem (Protox): If s is (persistently) a first-order stream, and $q \in \mathbb{N}$ then

$$\begin{array}{l} b2o(\texttt{Bc}(q,s))\\ \quad \cong \texttt{let}\{z_q := \texttt{mk}(q)\}\texttt{let}\{z_s := \texttt{mk}(s)\}\texttt{Oc}(z_q, z_s)\\ \quad \cong \texttt{let}\{z_q := \texttt{mk}(q)\}\texttt{let}\{o_s := b2o(s)\}\texttt{Ocx}(z_q, o_s)\end{array}$$

Note that, we could omit the FO requirement on the stream if we apply `fop` to answer returned.

Proof : Again the first equivalence is by (**b2o**) and for the second equivalence we can use FOSI-I. Define $S(y_{0,0}, y_{0,1}, y_{1,0}, y_{1,1})$ by $q = y_{0,0} = y_{1,0} \in \mathbb{N}$, $s = y_{0,1} = y_{1,1}$ is persistently a first-order stream.

9.4. Specialization

Continuing with the `Bc` example, we specialize the stream s to the stream, $\texttt{f2b}(f, n)$, generated by a function f. $\texttt{Bc}_e$ is the specialized behavior and $\texttt{Oc}_e$ is the specialized object description.

$$\begin{array}{l}\texttt{f2b} = \lambda f.\texttt{Y}(\lambda b.\lambda n.\lambda m.\texttt{pr}(b(n+1), f(n)))\\ \texttt{Bc}_e = \lambda f, d.\texttt{Y}(\lambda b.\lambda n.\lambda m.\\ \qquad\qquad \texttt{if}(m,\\ \qquad\qquad\quad \texttt{pr}(b(n+1), f(n)),\\ \qquad\qquad\quad \texttt{pr}(b(n), d+n)))\\ \texttt{Oc}_e = \lambda f, d.\lambda z_n.\lambda m.\\ \qquad\qquad \texttt{let}\{n := \texttt{get}(z_n)\}\\ \qquad\qquad\quad \texttt{if}(m,\\ \qquad\qquad\qquad \texttt{seq}(\texttt{set}(z_n, n+1), f(n)),\\ \qquad\qquad\qquad d+n)\end{array}$$

Theorem (Specialize): If f is a pure function on $\mathbb{N}$ and $q, n \in \mathbb{N}$ then

$$b2o(\texttt{Bc}(q, \texttt{f2b}(f, n))) \cong \texttt{let}\{z_n := \texttt{mk}(n)\}\texttt{Oc}_e(f, q-n)(z_n)$$

Proof : This can be proved directly by (**FOOSI**). A simpler proof is by establishing the following

$$\texttt{Bc}(q, \texttt{f2b}(f, n)) \cong \texttt{Bc}_e(f, q-n)(n) \qquad \text{using (BSI)}$$

$$b2o(\texttt{Bc}_e(f, q-n)(n)) \cong \texttt{let}\{z_n := \texttt{mk}(n)\}\texttt{Oc}_e(f, q-n)(z_n) \qquad \text{using (FOOSI)}$$

9.5. Rerepresentation

The following examples illustrate transformations on the shape of memory used by an object. D_j stores a pair of atoms. The left or right atom can be set or retrieved. We assume $\texttt{putL}, \texttt{putR}, \texttt{getL}, \texttt{getR}$ are message type recognizers that map arbitrary values to booleans without any effect on memory. `atof` maps a value to an atom. D_0 stores the pair as a cons cell contained in its local store z, D_1 stores the pair simply as a cons cell c, D_2 stores the pair as a pair of cells, p, D_3 stores the pair in two local variables z_l, z_r. The Lisp operations `cons`, `car`, `cdr`, `setcar`, `setcdr` used in these examples are defined as follows:

$\texttt{cons} = \lambda x, y.\texttt{mk}(\texttt{pr}(\texttt{mk}(x), \texttt{mk}(y)))$

$\texttt{car} = \lambda x.\texttt{fst}(\texttt{get}(x))$

$\texttt{cdr} = \lambda x.\texttt{snd}(\texttt{get}(x))$

$\texttt{setcar} = \lambda x, y.\texttt{set}(x, \texttt{pr}(y, \texttt{cdr}(x)))$

$\texttt{setcdr} = \lambda x, y.\texttt{set}(x, \texttt{pr}(\texttt{car}(x), y))$

$D_0 = \lambda z.\lambda m.$
$\quad \texttt{let}\{c := \texttt{get}(z)\}$
$\quad\quad \texttt{cond}([\texttt{putL}(m) \Rightarrow \texttt{seq}(\texttt{setcar}(c, \texttt{atof}(m)), \texttt{nil})]$
$\quad\quad\quad [\texttt{putR}(m) \Rightarrow \texttt{seq}(\texttt{setcdr}(c, \texttt{atof}(m)), \texttt{nil})]$
$\quad\quad\quad [\texttt{getL}(m) \Rightarrow \texttt{car}(c)]$
$\quad\quad\quad [\texttt{getR}(m) \Rightarrow \texttt{cdr}(c)]$
$\quad\quad\quad [\texttt{t} \Rightarrow \texttt{nil}]\ )$

$D_1 = \lambda c.\lambda m.$
$\quad \texttt{cond}([\texttt{putL}(m) \Rightarrow \texttt{seq}(\texttt{setcar}(c, \texttt{atof}(m)), \texttt{nil})]$
$\quad\quad [\texttt{putR}(m) \Rightarrow \texttt{seq}(\texttt{setcdr}(c, \texttt{atof}(m)), \texttt{nil})]$
$\quad\quad [\texttt{getL}(m) \Rightarrow \texttt{car}(c)]$
$\quad\quad [\texttt{getR}(m) \Rightarrow \texttt{cdr}(c)]$
$\quad\quad [\texttt{t} \Rightarrow \texttt{nil}]\ )$

$D_2 = \lambda p.\lambda m.$
$\quad \texttt{cond}([\texttt{putL}(m) \Rightarrow \texttt{seq}(\texttt{set}(\texttt{fst}(p), \texttt{atof}(m)), \texttt{nil})]$
$\quad\quad [\texttt{putR}(m) \Rightarrow \texttt{seq}(\texttt{set}(\texttt{snd}(p), \texttt{atof}(m)), \texttt{nil})]$
$\quad\quad [\texttt{getL}(m) \Rightarrow \texttt{get}(\texttt{fst}(p))]$
$\quad\quad [\texttt{getR}(m) \Rightarrow \texttt{get}(\texttt{snd}(p))]$
$\quad\quad [\texttt{t} \Rightarrow \texttt{nil}]\ )$

$$D_3 = \lambda z_l, z_r.\lambda m.$$
$$\begin{aligned}
&\texttt{cond}([\texttt{putL}(m) \Rightarrow \texttt{seq}(\texttt{set}(z_l, \texttt{atof}(m)), \texttt{nil})]\\
&\quad [\texttt{putR}(m) \Rightarrow \texttt{seq}(\texttt{set}(z_r, \texttt{atof}(m)), \texttt{nil})]\\
&\quad [\texttt{getL}(m) \Rightarrow \texttt{get}(z_l)]\\
&\quad [\texttt{getR}(m) \Rightarrow \texttt{get}(z_r)]\\
&\quad [\texttt{t} \Rightarrow \texttt{nil}]\ )
\end{aligned}$$

Theorem (Xcons):

$$\begin{aligned}
&\texttt{let}\{z := \texttt{mk}(\texttt{cons}(x, y))\}D_0(z)\\
&\quad \cong \texttt{let}\{c := \texttt{cons}(x, y)\}D_1(c)\\
&\quad \cong \texttt{let}\{p := \texttt{pr}(\texttt{mk}(x), \texttt{mk}(y)))\}D_2(p)\\
&\quad \cong \texttt{let}\{z_l := \texttt{mk}(x)\}\texttt{let}\{z_r := \texttt{mk}(y)\}D_3(z_l, z_r)
\end{aligned}$$

To prove this theorem we generalize the FOO simulation induction principle to objects with arbitrary, but static store shape. A memory shape M is a triple $(\mathbf{y})(\mathbf{z}, \mathbf{v})$ where $\mathbf{y}, \mathbf{z}$ are sequences of distinct variables, $\mathbf{v}$ is a sequence of values expressions in $\mathbb{V}_{\mathbf{y},\mathbf{z}}$ of the same length as $\mathbf{z}$. If $\mathbf{w}$ is a sequence of value expressions of the same length as $\mathbf{y}$, not containing any elements of $\mathbf{z}$, then $M(\mathbf{w})$ denotes the memory context $\{\mathbf{z} := \mathbf{v}^\sigma\}$ where σ maps elements of $\mathbf{y}$ to corresponding elements of $\mathbf{w}$.

Definition (Generalized FOO Simulation): Let $\mathbf{y}_j$ be sequences of distinct variables of length k_j for $j < 2$. Let $M_j = (\mathbf{y}_j)(\mathbf{z}_j, \mathbf{v}_j)$ be memory shapes. Let O_j be objects with m_j-ary store for $j < 2$ and let $\mathbf{c}_j$ be subsequences of $\mathbf{z}_j$ of length m_j for $j < 2$. An $k_0 + k_1$-ary relation S is a simulation relation for $M_j(\mathbf{y}_j)O_j(\mathbf{c}_j)$ if S is persistent, and the following holds (for first-order values of any free variables):

$$\begin{aligned}
&S(\mathbf{y}_0, \mathbf{y}_1) \Rightarrow (\forall m)(\exists \mathbf{y}_0', \mathbf{y}_1', r)\\
&\quad S(\mathbf{y}_0', \mathbf{y}_1') \wedge\\
&\quad \mathit{IsFo}(r) \wedge\\
&\quad\quad \wedge_{j<2} \Uparrow M_j(\mathbf{y}_j)O_j(\mathbf{c}_j)(m) \vee\\
&\quad\quad\quad \wedge_{j<2} M_j(\mathbf{y}_j)[\![O_j(\mathbf{c}_j)(m) \cong O_j(\mathbf{c}_j)(\texttt{fop}(m))]\!] \wedge\\
&\quad\quad\quad \wedge_{j<2} M_j(\mathbf{y}_j)[\![O_j(\mathbf{c}_j)(m) \cong \texttt{seq}(\texttt{set}(\mathbf{z}_j, \mathbf{v}_j^{\{\mathbf{y}_j := \mathbf{y}_j'\}}), r)]\!]
\end{aligned}$$

Theorem (Generalized FOO Object Simulation Induction (GFOOSI)): Let M_j, O_j be as in the definition of generalized FOO simulation. Let S be an $k_0 + k_1$-ary relation. If S is a simulation relation for $M_j(\mathbf{y}_j)O_j(\mathbf{c}_j)$ then (for first-order values of any free variables)

$$S(\mathbf{y}_0, \mathbf{y}_1) \Rightarrow M_0(\mathbf{y}_0)O_0(\mathbf{c}_0) \cong M_1(\mathbf{y}_1)O_1(\mathbf{c}_1)$$

Proof : The proof now is a simple application of (**GFOOSI**). The four memory shapes are:

$M_0 = (x, y)([z, c][c, \mathtt{pr}(x, y)])$
$M_1 = (x, y)([c][\mathtt{pr}(x, y)])$
$M_2 = (x, y)([z_l, z_r][x, y])$
$M_3 = (x, y)([z_l, z_r][x, y])$

The simulation relation for equivalence of D_j, D_k, $S(x_j, y_j, x_k, y_k)$, is equivalent to $x_j \cong x_k$, $y_j \cong y_k$, and $x_j, y_j \in \mathbb{A}$. The proof that S is a simulation is similar to the argument used in (**constant lifting**). Note that the usual equational properties of Lisp operations can be derived from the corresponding properties of `mk`, `get`, `set` and properties of pairing. □

10. Conclusions

In this paper we have continued our development of a logic for reasoning about imperative functional programs. The main results of this paper are

1. The refinement of the semantics given in [10,11] so as to be more faithful to the local nature of portions of memory. One criteria for a semantic theory is that it provides an accurate notion of program equivalence. A key issue in the case of imperative functional programs is the correct treatment of access to both private and shared mutable data. In this paper we give several notions of visibility or privacy that provide a logical semantics faithful to the underlying operational one.
2. The use of classes to express *similarity* relations, which in turn are used to express powerful proof principles for establishing equivalences between imperative functional programs.

Even though the work presented here is at an early stage of development, we feel that much progress has been made towards practical application of formal methods to program development. Future work includes: applying these methods to more substantial programming examples that arise in practice; development of a proof calculus for VTLoE and establishing completeness results for the zero-order fragment in the spirit of [15]; and formalization of VTLoE in a proof development system.

11. Collected Laws of VTLoE

11.1. Lambda Laws, Equational and Definedness

Lemma ($\mathtt{let}_c$ [11]):

(i) $\mathtt{app}(\lambda x.e, v) \cong e^{\{x:=v\}} \cong \mathtt{let}\{x := v\}e$

(ii) $R[e] \cong \mathtt{let}\{x := e\}R[x]$

(iii) $R[\mathtt{let}\{x := e_0\}e_1] \cong \mathtt{let}\{x := e_0\}R[e_1]$

where in (ii) and (iii) we require x not free in R.

Lemma ($\Uparrow$ [11]):

(i) $\Uparrow e_0 \Rightarrow (\Uparrow e_1 \Leftrightarrow e_0 \cong e_1)$

(ii) $\Uparrow e \Rightarrow \Uparrow U[e]$ e closed

(iii) $\Uparrow \mathtt{Y}(\lambda y.\lambda x.\mathtt{app}(y,x))(\mathtt{nil})$

11.2. Contextual Assertions

The theorem (**ca**) provides several principles for reasoning about contextual assertions: a simple principle concerning reduction contexts; a general principle for introducing contextual assertions (akin to the rule of necessitation in modal logic); a principle for propagating contextual assertions through equations; and a principle for composing contexts (or collapsing nested contextual assertions); a principle for manipulating contexts; three principles demonstrating that contextual assertions interact nicely with the propositional connectives, if we take proper account of assertions that are true for the trivial reason that during execution, the point in the program text marked by the context hole is never reached; and a principle (whose converse is false) concerning the quantifier. Finally a lemma demonstrating that contextual assertions interact nicely with evaluation.

Theorem (ca [11]):

(i) $\Phi \Rightarrow R[\![\Phi]\!]$

(ii) $\models \Phi$ implies $\models U[\![\Phi]\!]$

(iii) $U[\![e_0 \cong e_1]\!] \Rightarrow U[e_0] \cong U[e_1]$

(iiii) $\models U_0[\![U_1[\![\Phi]\!]]\!] \Leftrightarrow (U_0[U_1])[\![\Phi]\!]$

(iv) $\mathtt{let}\{x := R[e]\}U[\![\Phi]\!] \Leftrightarrow \mathtt{let}\{y := e\}\mathtt{let}\{x := R[y]\}U[\![\Phi]\!]$ y fresh

(v) $U[\![\mathtt{False}]\!] \Rightarrow U[\![\Phi]\!]$

(vii) $U[\![\neg\Phi]\!] \Leftrightarrow (U[\![\mathtt{False}]\!] \vee \neg U[\![\Phi]\!])$

(viii) $U[\![\Phi_0 \Rightarrow \Phi_1]\!] \Leftrightarrow (U[\![\Phi_0]\!] \Rightarrow U[\![\Phi_1]\!])$

(viiii) $U[\![\forall x \Phi]\!] \Rightarrow \forall x\, U[\![\Phi]\!]$ where x not free in U

11.3. Memory Operations

The contextual assertions and laws involving **mk**, **set** and **get** are given below. The assertion, (**mk.i**), describes the allocation effect of a call to **mk**, while (**mk.ii**) expresses what is unaffected by a call to **mk**. The assertion, (**mk.iii**), expresses the totality of **mk**. The **mk** delay law, (**mk.iv**), asserts that the time of allocation has no discernable effect on the resulting cell. In a world with control effects evaluation of e_0 must be free of them for this principle to be valid [7]. The first three contextual assertions regarding **set** are analogous to those of **mk**. They describe what is returned and what is altered, what is not altered as well as when the operation is defined. The remaining three principles involve the commuting, cancellation, absorption of calls to **set**. For example the **set** absorption principle, (**set.vi**), expresses that under certain simple conditions allocation followed by assignment may

be replaced by a suitably altered allocation. The contextual assertions regarding **get** follow the above pattern. They describe what is altered and returned, what is not altered as well as when the operation is defined.

Lemma (Memory operation laws [11]):

(**mk**.i) $\mathtt{let}\{x := \mathtt{mk}(v)\}[\![\neg(x \cong y) \wedge \mathtt{iscell?}(x) \cong \mathtt{t} \wedge \mathtt{get}(x) \cong v]\!]$

x fresh

(**mk**.ii) $y \cong \mathtt{get}(z) \Rightarrow \mathtt{let}\{x := \mathtt{mk}(v)\}[\![y \cong \mathtt{get}(z)]\!]$

(**mk**.iii) $\Downarrow \mathtt{mk}(z)$

(**mk**.iv) $\mathtt{let}\{y := e_0\}\mathtt{let}\{x := \mathtt{mk}(v)\}e_1 \cong \mathtt{let}\{x := \mathtt{mk}(v)\}\mathtt{let}\{y := e_0\}e_1$

$x \notin \mathrm{FV}(e_0)$, $y \notin \mathrm{FV}(v)$

(**set**.i) $\mathtt{iscell?}(z) \Rightarrow \mathtt{let}\{x := \mathtt{set}(z, y)\}[\![\mathtt{get}(z) \cong y \wedge x \cong \mathtt{nil}]\!])$

(**set**.ii) $(y \cong \mathtt{get}(z) \wedge \neg(w \cong z)) \Rightarrow \mathtt{let}\{x := \mathtt{set}(w, v)\}[\![y \cong \mathtt{get}(z)]\!]$

(**set**.iii) $\mathtt{iscell?}(z) \Rightarrow \Downarrow \mathtt{set}(z, x)$

(**set**.iv) $\neg(x_0 \cong x_2) \Rightarrow$
$\mathtt{seq}(\mathtt{set}(x_0, x_1), \mathtt{set}(x_2, x_3)) \cong \mathtt{seq}(\mathtt{set}(x_2, x_3), \mathtt{set}(x_0, x_1))$

(**set**.v) $\mathtt{seq}(\mathtt{set}(x, y_0), \mathtt{set}(x, y_1)) \cong \mathtt{set}(x, y_1)$

(**set**.vi) $\mathtt{get}(y) \cong z \Rightarrow \mathtt{set}(y, z) \cong \mathtt{nil}$ $\quad z$ not free in w

(**get**.i) $\mathtt{let}\{x := \mathtt{get}(y)\}[\![x \cong \mathtt{get}(y)]\!]$

(**get**.ii) $y \cong \mathtt{get}(z) \Rightarrow \mathtt{let}\{x := \mathtt{get}(w)\}[\![y \cong \mathtt{get}(z)]\!]$

(**get**.iii) $\mathtt{iscell?}(x) \Leftrightarrow (\exists y)(\mathtt{get}(x) \cong y)$

Acknowledgements

We would like to thank four anonymous referees for helpful comments on an earlier draft. This research was partially supported by DARPA contract NAG2-703, and NSF grants CCR-8917606, and CCR-8915663.

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, McGraw-Hill Book Company, 1985.
2. K.R. Apt. Ten years of Hoare's logic: A survey–part I. *ACM Transactions on Programming Languages and Systems*, 4:431–483, 1981.
3. S. Feferman. A language and axioms for explicit mathematics. In *Algebra and Logic*, volume 450 of *Springer Lecture Notes in Mathematics*, pages 87–139. Springer Verlag, 1975.
4. S. Feferman. Constructive theories of functions and classes. In *Logic Colloquium '78*, pages 159–224. North-Holland, 1979.

5. S. Feferman. A theory of variable types. *Revista Colombiana de Matématicas*, 19:95–105, 1985.

6. S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In *Logic and Computation*, volume 106 of *Contemporary Mathematics*, pages 101–136. A.M.S., Providence R. I., 1990.

7. M. Felleisen, 1993. Personal communication.

8. M. Felleisen and D.P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

9. D. Harel. Dynamic logic. In D. Gabbay and G. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II*, pages 497–604. D. Reidel, 1984.

10. F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A theory of classes for a functional language with effects. In *Proceedings of CSL92*, volume 702 of *Lecture Notes in Computer Science*, pages 309–326. Springer, Berlin, 1993.

11. F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 1994. to appear.

12. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

13. I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

14. I. A. Mason and C. L. Talcott. Program transformation for configuring components. In *ACM/IFIP Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1991.

15. I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.

16. I. A. Mason and C. L. Talcott. References, local variables and operational reasoning. In *Seventh Annual Symposium on Logic in Computer Science*, pages 186–197. IEEE, 1992.

17. I. A. Mason and C. L. Talcott. Program transformation via contextual assertions. In N. D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language, and Computation: Festschrift in Honor of Satoru Takasu*, number 792 in Lecture Notes in Computer Science, pages 225–254. Springer-Verlag, 1994.

18. R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

19. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

20. A. M. Pitts. Evaluation logic. In *IVth Higher-Order Workshop, Banff*, volume 283 of *Workshops in Computing*. Springer-Verlag, 1990.

21. J.C. Reynolds. Idealized ALGOL and its specification logic. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

22. G. L. Steele and G. J. Sussman. Scheme, an interpreter for extended lambda calculus. Technical Report Technical Report 349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1975.

23. C. L. Talcott. A theory for program and data type specification. *Theoretical Computer Science*, 104:129–159, 1992.