

A Theory of Classes for a Functional Language with Effects

Furio Honsell
Udine University
honsell@uduniv.cineca.it

Ian A. Mason
Stanford University
iam@cs.stanford.edu

Scott Smith
Johns Hopkins University
scott@cs.jhu.edu

Carolyn Talcott
Stanford University
clt@sail.stanford.edu

1. Introduction

It is well known that the addition of references or other mutable data to a functional programming language complicates matters. Adding operations for manipulating references to the simply typed lambda calculus causes the failure of most of the nice mathematical properties. For example strong normalization fails because it is possible to construct a fixed-point combinator for any functional type:

$$Y_m = \lambda p. \mathbf{let}\{z := \mathbf{mk}(g)\} \mathbf{seq}(\mathbf{set}(z, \lambda x. \mathbf{app}(\mathbf{app}(p, \mathbf{get}(z)), x)), \mathbf{get}(z))$$

where $\mathbf{mk}(v)$ allocates a cell with contents v , $\mathbf{get}(z)$ gets the current contents of the cell z , and $\mathbf{set}(z, v)$ sets the contents of the cell z to be v , and g is any variable. In addition, references are problematic for polymorphic type systems [28, 29]. References are also troublesome from a denotational point of view as illustrated by the absence of fully abstract models. For example, in [17] Meyer and Sieber give a series of examples of programs that are operationally equivalent (according to the intended semantics of block-structured Algol-like programs) but which are not given equivalent denotations in traditional denotational semantics. They propose various modifications to the denotational semantics which solve some of these discrepancies, but not all. In [21, 20] a denotational semantics that overcomes some of these problems is presented. However variations on the seventh example remain problematic. Since numerous proof systems for Algol are sound for the denotational models in question, [8, 7, 25, 22, 12, 21, 20], these equivalences, if expressible, must be independent of these systems.

In this paper we introduce a variable typed logic of effects (i.e. a logic of effects where classes can be defined and quantified over) inspired by the variable type systems of Feferman [3, 4] for purely functional languages. A similar extension incorporating non-local control operations was introduced in [27]. The logic we present provides an expressive language for defining specifications and constraints and for studying properties and program equivalences, in a uniform framework. Thus it has an advantage over a plethora of systems in the literature that aim to capture solitary aspects of computation. The theory also allows for the construction of inductively defined sets and derivation of the corresponding induction principles. Classes can be used to express, inter alia, the non-expansiveness of terms [29]. Other effects can also be represented within the system. These include read/write effects and various forms of interference [24]. The first order fragment is described in [16] where it is used to resolve the denotationally problematic examples of [17].

In our language atoms, references and lambda abstractions are all first class values and as such are storable. This has several consequences. Firstly, mutation

and variable binding are separate and so we avoid the problems that typically arise (e.g. in Hoare's and dynamic logic) from the conflation of program variables and logical variables. Secondly, the equality and sharing of references (aliasing) is easily expressed and reasoned about. Thirdly, the combination of mutable references and lambda abstractions allows us to study object based programming within our framework. Our atomic formulas express the (operational or observational) equivalence of programs à la Plotkin [23]. Neither Hoare's logic nor Dynamic logic incorporate this ability, or make use of such equivalences (e.g. by replacing one piece of program text by another without altering the overall meaning).

The terms of our language are simply the terms of the call-by-value lambda calculus extended by the reference primitives `mk`, `set`, `get`. We also include a collection of operations¹ and basic constants or atoms \mathbb{A} , (such as the Lisp booleans `t` and `nil` as well as the integers \mathbb{Z}). We can think of this language as an untyped dialect of ML. The atomic formulas of our language assert class membership and the operational equivalence of expressions. In addition to the usual first-order formula constructions and quantification over class variables, we add a mechanism for annotating points in programs with formulas. Namely, *contextual assertions*: if Φ is a formula and U is a *univalent 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. Univalent contexts are the largest natural class of contexts (expressions with a unique hole) whose symbolic evaluation is unproblematic. Contextual assertions generalize Hoare's triples in that they can be nested, used as assumptions, and their free variables may be quantified. They are similar in spirit to program modalities in dynamic logic. Using contextual assertions we can express the axioms concerning the effects of `mk` and `set` simply and elegantly. This improves the complete system (for quantifier/recursion free expressions) presented in [15] where the corresponding rules had complicated side-conditions.

The semantics of expressions is a call-by-value evaluation relation given by a reduction relation on syntactic entities. In [14] we used this approach to establish a *useful* characterization of operational equivalence. This characterization reduces the number of contexts that need to be considered. The class of contexts that need to be considered correspond naturally to states of an abstract machine. The logic is a partial term logic with variables ranging over values. The characterization of operational equivalence allows for a natural notion of satisfaction of first order formulas relative to a memory state and assignment of values to variables. Our style of operational semantics naturally provides for the symbolic evaluation of contexts, which is the key to defining the semantics of contextual assertions. Classes range over sets of values closed under operational equivalence.

¹ In our work operations come in three flavors: algebraic operations which act on atomic data, and whose properties are given by algebraic equations; structural operations which act uniformly on specific kinds of data (other than atomic) such as pairs, records, finite sets; and computational operations which provide access to computation state, these include memory operations, and control operations. Algebraic and structural operations are *context free* – their action/meaning is independent of computation state, whereas the meaning of computation primitives is effected by and can effect computation state.

In the presence of effects several notions split into spectrums of variations. We give two examples:

Firstly, there are many possible notions of “function space” according to how the effects of a computation are accounted for. One example is the class of memory functions, $X_1, \dots, X_n \xrightarrow{\mu} Y$ with arguments in X_1, \dots, X_n and result in Y allowing for the possible modification of memory in the process. This can be refined by making the possible effects explicit in the spirit of [10, 11]. At the other end of the spectrum there is the function space that corresponds to those operations that return appropriate values without even enlarging memory, let alone altering existing memory.

Secondly, in the presence of effects there are several degrees of “definedness”. They are all easily expressible in our system. The weakest notion is that of computational definedness. An expression is computationally defined, if (for any assignment of free variables) it returns a value. A stronger notion is that of an expression evaluating to a value, without altering (but possibly enlarging) memory. An even stronger notion of definedness is that of an expression evaluating to a value, without altering or enlarging memory. The strongest notion of definedness is that of evaluating to a value independently of the memory. The following terms exemplify these degrees: $\mathbf{mk}(x)$, $\mathbf{seq}(\mathbf{mk}(x), 1)$, $\mathbf{get}(x)$, 1.

Feferman [4] proposes an explanation of ML types in the variable type framework. This gives a natural semantics to ML type expressions, but there are problems with polymorphism, even in the purely functional case. For example the fixed point operator can be typed in ML as $(\forall X, Y)([X \rightarrow Y] \rightarrow [X \rightarrow Y] \rightarrow [X \rightarrow Y])$ but this is false in the variable type framework as there are types (classes) not closed under “limits of chains”. The situation becomes more problematic when references are added. Naive attempts to represent ML types as classes fails in sense that ML inference rules are not valid. It seems that the essential feature of ML type system, in addition to the inference rules, is the preservation of types during the execution of well-typed programs. In this sense they are more syntactic than semantic.

The remainder of this paper is organized as follows. Section 2. reviews the syntax and semantics of the underlying computational language and summarizes the main results of previous work. The unfamiliar reader may be advised to consult [14, 16] for a more detailed treatment. VTL_{oE} (Variable Typed Logic of Effects) is introduced in two stages. The first stage is the first-order theory of individuals built on assertions of equality (operational equivalence), and contextual assertions. This is presented in Section 3. The second stage extends the logic to include classes and class membership. This is presented in Section 4. In Section 5. we present our conclusions and suggest further directions of research.

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, \dots , 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 . $\bar{y} = [y_1, \dots, y_n]$ is the sequence of length n with i th element y_i . $\mathbf{P}_\omega(Y)$ is the set of finite subsets of

Y . $[Y_0 \rightarrow Y_1]$ is the set of total functions f with domain Y_0 and range contained in Y_1 . We write $\text{Dom}(f)$ for the domain of a function and $\text{Rng}(f)$ for its range. For any function f , $f\{y := y'\}$ is the function f' such that $\text{Dom}(f') = \text{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \text{Dom}(f)$. $\mathbb{N} = \{0, 1, 2, \dots\}$ is the natural numbers and i, j, n, n_0, \dots range over \mathbb{N} .

2. The Syntax and Semantics of Terms

The syntax of the terms of our language is a simple extension of that 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} , which include the the memory operations $\{\mathbf{get}, \mathbf{set}, \mathbf{mk}\}$, and the *context free* operations $\{\mathbf{cell}, \mathbf{eq}, \mathbf{br}\}$. The branching primitive **br** is a strict version of the Lisp conditional, **if**. 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} , and the set of expressions, \mathbb{E} as the least sets satisfying the following equations:

$$\mathbb{L} = \lambda\mathbb{X}.\mathbb{E} \quad \mathbb{V} = \mathbb{X} + \mathbb{A} + \mathbb{L} \quad \mathbb{E} = \mathbb{V} + \mathbf{app}(\mathbb{E}, \mathbb{E}) + \mathbb{F}_n(\mathbb{E}^n)$$

λ is a binding operator and free and bound variables of expressions are defined as usual. $\text{FV}(e)$ is the set of free variables of e . For any syntactic domain Y and set of variables X we let Y_X be the elements of Y with free variables in X . 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 $x \in \text{Dom}(\sigma)$ 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).

We use the syntactic sugar **let**, **if**, **seq** (a sequencing construct akin to **progn**, **begin** or **;**) to make programs more readable. For example

$$\begin{aligned} \mathbf{let}\{x := e_0\}e_1 & \text{ abbreviates } \mathbf{app}(\lambda x.e_1, e_0) \\ \mathbf{seq}(e_0, e_1) & \text{ abbreviates } \mathbf{app}(\lambda z.e_1, e_0) \quad z \text{ fresh} \\ \mathbf{if}(e_0, e_1, e_2) & \text{ abbreviates } \mathbf{app}(\mathbf{br}(\lambda z.e_1, \lambda z.e_2, e_0), \mathbf{nil}) \quad z \text{ fresh} \\ \mathbf{app}(f, x_1, \dots, x_n) & \text{ abbreviates } \mathbf{app}(\dots(\mathbf{app}(f, x_1), x_2), \dots, x_n) \end{aligned}$$

Contexts are expressions with holes. We use \bullet to denote a hole. The set of contexts, \mathbb{C} , is defined by

$$\mathbb{C} = \{\bullet\} + \mathbb{X} + \mathbb{A} + \lambda\mathbb{X}.\mathbb{C} + \mathbf{app}(\mathbb{C}, \mathbb{C}) + \mathbb{F}_n(\mathbb{C}^n)$$

We let C range over \mathbb{C} . $C[e]$ denotes the result of replacing any holes in C by e . Free variables of e may become bound in this process. The finite set of variable which may be trapped by filling the context C are called its traps and denoted by $\text{Traps}(C)$.

The operational semantics of expressions is given by a reduction relation \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 identify the subexpression of an expression that is to be evaluated next. They correspond to the left-first, call-by-value reduction strategy of Plotkin [23] and were first introduced in [5].

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

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

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

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, Γ_0, \dots range over memory contexts. We can view memory contexts as *finite maps from variables to value expressions*. Thus we refer to their domain, $\text{Dom}(\Gamma)$; modify them, $\Gamma\{z := \mathbf{mk}(v)\}$, when $z \in \text{Dom}(\Gamma)$; extend them, $\Gamma\{z := \mathbf{mk}(v)\}$, when $z \notin \text{Dom}(\Gamma)$; and form the disjoint union of two of them, $(\Gamma_0 \cup \Gamma_1)$.

A *description* is a pair, $\Gamma; e$, with first component a memory context and second component an arbitrary expression (As mentioned above, this arbitrary expression is either a value expression, or *decomposes uniquely* into a redex placed in a reduction context.) *Value descriptions* are descriptions whose expression is a value expression, $\Gamma; v$. We use the convention that an expression used as a description means that the memory context is empty, thus e abbreviates $\emptyset; e$. Note that descriptions may have free variables (i.e. it need not be the case that $\text{FV}(e) \subseteq \text{Dom}(\Gamma)$).

The reduction relation \mapsto^* is the reflexive transitive closure of \mapsto . The interesting clauses are:

$$\begin{aligned} (\text{beta}) \quad & \Gamma; R[\mathbf{app}(\lambda x.e, v)] \mapsto \Gamma; R[e^{x:=v}] \\ (\text{mk}) \quad & \Gamma; R[\mathbf{mk}(v)] \mapsto \Gamma\{z := \mathbf{mk}(v)\}; R[z] \quad z \notin \text{Dom}(\Gamma) \cup \text{FV}(R[v]) \\ (\text{get}) \quad & \Gamma; R[\mathbf{get}(z)] \mapsto \Gamma; R[v] \quad \text{assuming } z \in \text{Dom}(\Gamma) \text{ and } \Gamma(z) = v \\ (\text{set}) \quad & \Gamma; R[\mathbf{set}(z, v)] \mapsto \Gamma\{z := \mathbf{mk}(v)\}; R[\mathbf{nil}] \quad \text{assuming } z \in \text{Dom}(\Gamma) \\ (\text{cell}) \quad & \Gamma; R[\mathbf{cell}(v)] \mapsto \begin{cases} \Gamma; R[\mathbf{t}] & \text{if } v \in \text{Dom}(\Gamma) \\ \Gamma; R[\mathbf{nil}] & \text{if } v \in \mathbb{L} \cup \mathbb{A} \end{cases} \end{aligned}$$

A description, $\Gamma; e$ is *defined* (written $\downarrow \Gamma; e$) if it evaluates to a value description. For closed expressions e , we write $\downarrow e$ to mean $\downarrow \emptyset; e$. We say two expressions are *eqidefined*, $e_0 \Downarrow e_1$, to mean that $(\downarrow e_0) \text{ iff } (\downarrow e_1)$. Note that in the `cell` rule if one of the arguments is a variable not in the domain of the memory context, then no reduction step is possible.

Operational (or observational) equivalence formalizes the notion of programs as black-boxes. Treating programs as black boxes requires only observing what effects and values they produce, and not how they produce them. In our framework the allowed observations are those made by closing program contexts. 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. This definition extends the extensional equivalence relations defined by Morris [19] and Plotkin [23] to computation over memory structures.

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. We define a *closed instantiation* of an expression e to be a memory context, Γ , together with a value substitution, σ , such that $\Gamma[e^\sigma]$ is closed. A *use* of an expression e is the placement of e into a reduction context. The desired result is then that two expressions are operationally equivalent just if all closed instances of all uses are equidefined.

Theorem (ciu):

$$e_0 \cong e_1 \Leftrightarrow (\forall \Gamma, \sigma, R)(\text{FV}(\Gamma[R[e_0^\sigma]]) = \emptyset \Rightarrow (\Gamma[R[e_0^\sigma]] \Downarrow \Gamma[R[e_1^\sigma]]))$$

The proof of (ciu) appears in [14]. Using this theorem we can easily establish, for example, the validity of the let-rules of the lambda-c calculus [18].

3. The Syntax and Semantics of Formulas I. — Individuals

In addition to being a useful tool for establishing laws of operational equivalence, (ciu) can be used to define a satisfaction relation between memory contexts and equivalence assertions. In an obvious analogy with the usual first-order Tarskian definition of satisfaction this can be extended to define a satisfaction relation $\Gamma \models \Phi[\sigma]$.

The atomic formulas of our language assert the operational equivalence of two expressions. In addition to the usual first-order formula constructions we add *contextual assertions*: if Φ is a formula and U is a certain type of context, then $U[\Phi]$ is a formula. This form of formula expresses the fact that the assertion Φ holds at the point in the program text marked by the hole in U , if execution of the program reached that point. The contexts allowed in contextual assertions are called *univalent contexts*, (U-contexts). They are the largest natural class of contexts whose symbolic evaluation is unproblematic. The key restriction is that we forbid the hole to appear in the scope of a (non-let) lambda, thus preventing the proliferation of holes. The class of U-contexts, \mathbb{U} , is defined as follows.

Definition (U):

$$\mathbb{U} = \{\bullet\} + \text{let}\{\mathbb{X} := \mathbb{E}\}\mathbb{U} + \text{if}(\mathbb{E}, \mathbb{U}, \mathbb{U}) + \text{app}(\mathbb{U}, \mathbb{E}) + \text{app}(\mathbb{E}, \mathbb{U}) + \mathbb{F}_{m+n+1}(\mathbb{E}^m, \mathbb{U}, \mathbb{E}^n)$$

The well-formed formulas, \mathbb{W} , of (the first order part of) our logic are defined as follows:

Definition (\mathbb{W}):

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

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

$$(\forall y)(\mathbf{let}\{x := \mathbf{mk}(v)\}[\neg(x \cong y) \wedge \mathbf{cell}(x) \cong \mathbf{t} \wedge \mathbf{get}(x) \cong v])$$

In order to define the semantics of contextual assertions, we need to extend computation to univalent contexts. The idea here is quite simple, to compute with contexts we need to keep track of the β -conversions that have taken place with the hole in the scope of the λ . To indicate that the substitution σ has taken place at the hole in U we write $U[\sigma]$. Computation is then written as $\Gamma; U[\sigma] \mapsto^* \Gamma'; U'[\sigma']$ and is defined as follows:

Definition ($\Gamma; U[\sigma] \mapsto^* \Gamma'; U'[\sigma']$): Let $U \in \mathbb{U}$ be such that $\text{Traps}(U) = \{x_1, \dots, x_n\}$, assume $\text{Dom}(\sigma) \cap \text{Traps}(U) = \emptyset$ and let z be a fresh variable. We write

$$\Gamma; U[\sigma] \mapsto^* \Gamma'; U'[\sigma']$$

to mean,

$$\Gamma; (U[\mathbf{app}(\dots \mathbf{app}(z, x_1), \dots, x_n)])^\sigma \mapsto^* \Gamma'; (U'[\mathbf{app}(\dots \mathbf{app}(z, x_1), \dots, x_n)])^{\sigma'}$$

and $\text{Dom}(\sigma') = \text{Dom}(\sigma) \cup (\text{Traps}(U) - \text{Traps}(U'))$. Note that σ and σ' will agree on the domain of σ .

The Tarskian definition of satisfaction $\Gamma \models \Phi[\sigma]$ is given by a simple induction on the structure of Φ .

Definition ($\Gamma \models \Phi[\sigma]$): $(\forall \Gamma, \sigma, \Phi, e_j)$ such that $\text{FV}(\Phi^\sigma) \cup \text{FV}(e_j^\sigma) \subseteq \text{Dom}(\Gamma)$ for $j < 2$ we define satisfaction:

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

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

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

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

The atomic clause in the definition of satisfaction is justified by the **(ciu)** theorem. Negation is definable, $\neg\Phi$ is just $\Phi \Rightarrow \mathbf{False}$, where \mathbf{False} is any unsatisfiable assertion, such as $\mathbf{t} \cong \mathbf{nil}$. We can also express the computational definedness of expressions by the following assertion $\neg\mathbf{seq}(e, [\mathbf{False}])$. We let $\downarrow e$ abbreviate this expression and $\uparrow e$ abbreviate its negation. We say that a formula is valid, written $\models \Phi$, if $\Gamma \models \Phi[\sigma]$ for Γ, σ such that $\text{FV}(\Phi^\sigma) \subseteq \text{Dom}(\Gamma)$. We define $\Phi_{\neg\text{write}}(e)$ to abbreviate:

$$\downarrow e \wedge (\forall z)(\forall y)((\mathbf{cell}(z) \cong \mathbf{t} \wedge \mathbf{get}(z) \cong y) \Rightarrow \mathbf{let}\{x := e\}[\mathbf{get}(z) \cong y]).$$

This asserts that the evaluation of the expression e does not visibly alter the contents of any pre-existing cell. In VTLoE we cannot express that an expression e does not

modify the contents of any cells. This can be seen by considering the following expression:

`if(or(atom(get(x)), cell(get(x))), x, let{y := get(x)}set(x, λz.app(y, z)))`

which makes no detectable changes to any cells. Consequently we can only express that the expression does not modify the contents of cells modulo operational equivalence.

The theorem **(ca)** provides three principles for reasoning about contextual assertions: 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).

Theorem (ca):

- (i) $\frac{\models \Phi}{\models U[\Phi]}$
- (ii) $U[e_0 \cong e_1] \Rightarrow U[e_0] \cong U[e_1]$
- (iii) $U_0[U_1[\Phi]] \Leftrightarrow (U_0[U_1])[\Phi]$

It is in general false that $\Phi \Rightarrow U[\Phi]$ holds, a simple counter-example being

`get(x) ≅ 2 ⇒ let{x := mk(3)}[get(x) ≅ 2]`

The converse of **(ca.ii)** is false, as can be seen by the following:

`let{x := mk(0)}let{y := mk(0)}[x ≅ let{x := mk(0)}let{y := mk(0)}[y]`

but `let{x := mk(0)}let{y := mk(0)}[¬(x ≅ y)]`. Note that **(ca.iii)** is false for general contexts; a simple counterexample is when $C_0 = \mathbf{app}(\bullet, v)$, $C_1 = \lambda x.\bullet$ and $\Phi = \mathbf{t} \cong \mathbf{nil}$. Contextual assertions also 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.

Lemma (con.prop):

- (triv) $U[\mathbf{False}] \Rightarrow U[\Phi]$
- (not) $U[\neg\Phi] \Leftrightarrow (U[\mathbf{False}] \vee \neg U[\Phi])$
- (imp) $U[\Phi_0 \Rightarrow \Phi_1] \Leftrightarrow (U[\Phi_0] \Rightarrow U[\Phi_1])$

The case of the quantifier is a little less simple.

Lemma (con.∀):

- (∀) $U[\forall x \Phi] \Rightarrow \forall x U[\Phi]$

The converse to (∀) is easily shown to be false by considering U to be `let{y := mk(t)}•` and Φ to be `¬(x ≅ y)`. Contextual assertions do interact nicely with evaluation.

Lemma (eval):

If $\Gamma_0; U_0[\sigma_0] \mapsto^* \Gamma_1; U_1[\sigma_1]$, then $\Gamma_0 \models U_0[\Phi][\sigma_0]$ iff $\Gamma_1 \models U_1[\Phi][\sigma_1]$

This logic extends and improves the complete first order system presented in [13, 15]. There certain reasoning principles were established as basic, and from these all others, suitably restricted, could be derived using simple equational reasoning. The system presented there had several defects. In particular the rules concerning the effects of **mk** and **set** had complicated side-conditions. Using contextual assertions we can express them simply and elegantly. Their justification is also unproblematic. The contextual assertions and axioms involving **mk**, and **set** are:

Definition (mk axioms):

- (mk.i) $\text{let}\{x := \text{mk}(v)\}[\neg(x \cong y) \wedge \text{cell}(x) \cong \text{t} \wedge \text{get}(x) \cong v]$ x fresh
- (mk.ii) $y \cong \text{get}(z) \Rightarrow \text{let}\{x := \text{mk}(v)\}[y \cong \text{get}(z)]$ x fresh
- (mk.iii) $\Downarrow \text{mk}(z)$
- (mk.iv) $\text{let}\{y := e_0\}\text{let}\{x := \text{mk}(v)\}e_1 \cong \text{let}\{x := \text{mk}(v)\}\text{let}\{y := e_0\}e_1$
 $x \notin \text{FV}(e_0), y \notin \text{FV}(v)$

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 axiom, **(mk.iv)**, asserts that the time of allocation has no discernable effect on the resulting cell.

Definition (set axioms):

- (set.i) $\text{cell}(z) \cong \text{t} \Rightarrow \text{let}\{x := \text{set}(z, y)\}[\text{get}(z) \cong y \wedge x \cong \text{nil}]$
- (set.ii) $(y \cong \text{get}(z) \wedge \neg(w \cong z)) \Rightarrow \text{let}\{x := \text{set}(w, v)\}[y \cong \text{get}(z)]$
- (set.iii) $\text{cell}(z) \cong \text{t} \Rightarrow \Downarrow \text{set}(z, x)$
- (set.iv) $\neg(x_0 \cong x_2) \Rightarrow \text{seq}(\text{set}(x_0, x_1), \text{set}(x_2, x_3)) \cong \text{seq}(\text{set}(x_2, x_3), \text{set}(x_0, x_1))$
- (set.v) $\text{seq}(\text{set}(x, y_0), \text{set}(x, y_1)) \cong \text{set}(x, y_1)$
- (set.vi) $\text{let}\{z := \text{mk}(x)\}\text{seq}(\text{set}(z, w), e) \cong \text{let}\{z := \text{mk}(w)\}e$ z not free in w

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.

4. The Syntax and Semantics of Formulas II. — Classes

Using methods of Feferman [2, 4] and Talcott [27], we extend our theory to include a general theory of classifications (classes for short). With the introduction of classes, principles such as structural induction, as well as principles accounting for the effects of an expression can easily be expressed. Classes serve as a starting point for studying semantic notions of type. As will be seen direct representation of type inference systems can be problematic, and additional notions may be required to provide a formal semantics. Even here classes are likely to play an important role.

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, $\{x \mid \Phi\}$.

Definition (\mathbb{K}): The set \mathbb{K} of class terms is defined by

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

We extend the set \mathbb{W} of formulas to include class membership and quantification over class variables. We should point out that \mathbb{K} and \mathbb{W} form a mutual recursive definition. The definition of expressions remains unchanged.

Definition (\mathbb{W}):

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

We let A, B, C, \dots, 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 give semantics to the extended language, we extend the satisfaction relation as follows. Firstly we let $\mathbb{K}_{\text{Dom}(\Gamma)}$, the set of class values over Γ , be the set of subsets of $\mathbb{V}_{\text{Dom}(\Gamma)}$ closed under \cong . We extend value substitutions to map class variables to class values. This is used to define $[K]_{\Gamma}^{\sigma}$, the value of a class term, K , relative to the given memory context, Γ , and the closing value substitution σ . In principle, the class term evaluation is relative to a valuation for class constants, but since all of our class constants are introduced by definitional extension, this can be ignored.²

Definition ($[\mathbb{K}]_{\Gamma}^{\sigma}$):

$$[X]_{\Gamma}^{\sigma} = \sigma(X)$$

$$[\{x \mid \Phi\}]_{\Gamma}^{\sigma} = \{v \in \mathbb{V}_{\text{Dom}(\Gamma)} \mid \Gamma \models \Phi[\sigma\{x := v\}]\}$$

We then extend the satisfaction relation to formulas involving class terms and quantifiers.

Definition ($\Gamma \models \Phi[\sigma]$): The new clauses in the inductive definition of satisfaction are:

$$\Gamma \models e \in K[\sigma] \Leftrightarrow (\exists v \in \mathbb{V}_{\text{Dom}(\Gamma)})(\Gamma; e^{\sigma} \mapsto^* \Gamma; v \wedge v \in [K]_{\Gamma}^{\sigma})$$

$$\Gamma \models (\forall X)\Phi[\sigma] \Leftrightarrow (\forall C \in \mathbb{K}_{\text{Dom}(\Gamma)})(\Gamma \models \Phi[\sigma\{X := C\}])$$

It is important to note that if $\Gamma \models e \in K[\sigma]$, then e evaluates (in the appropriate state) to a value without altering memory, the so-called non-expansive expressions [28, 29]. We define (extensional) equality and subset relations on classes in the usual manner.

$$K_0 \subseteq K_1 \Leftrightarrow (\forall x)(x \in K_0 \Rightarrow x \in K_1)$$

$$K_0 \equiv K_1 \Leftrightarrow K_0 \subseteq K_1 \wedge K_1 \subseteq K_0$$

² Some class *constants* are absolute (have meaning independent of memory), but most have meaning that varies with memory (even when they are closed). Thus the semantics of classes should be parameterized by a constant interpretation mapping constants to functions that map a memory to a set of values existing in that memory. As class constants only occur in our language as definitional extensions, this issue has been swept under the rug in the definition of $[K]_{\Gamma}^{\sigma}$.

As a consequence of the semantics of classifications the following are valid.

Lemma (class):

(def) $e \in K \Rightarrow \Downarrow e$

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

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

The usual form of **(allE)** is false, $\neg((\forall X)\Phi[X] \Rightarrow \Phi[K])$. A counter example will be given below after some additional notation has been introduced.

We introduce the following class constants. \emptyset is the empty class, $\emptyset = \{x \mid x \not\cong x\}$. **Val** is the class of all values, $\mathbf{Val} = \{x \mid x \cong x\}$. **Nil** is the class containing the single element **nil**, $\mathbf{Nil} = \{x \mid x \cong \mathbf{nil}\}$. **Cell** is the class of memory cells, $\mathbf{Cell} = \{x \mid \mathbf{cell}(x) \cong \mathbf{t}\}$. Note that the interpretation of \emptyset and **Nil** is independent of memory contexts, while $[\mathbf{Val}]_\Gamma = \mathbb{V}_{\text{Dom}(\Gamma)}$ and $[\mathbf{Cell}]_\Gamma = \text{Dom}(\Gamma)$. A class operator is a class term with a distinguished class variable. We write $T[X]$ making the variable explicit and $T[K]$ for the result of replacing the distinguished variable X by K (with suitable renaming of bound variables to avoid capture). We can refine the class of cells to reflect the class of their contents.

$\mathbf{Cell}[Z] = \{x \mid \mathbf{cell}(x) \cong \mathbf{t} \wedge \mathbf{get}(x) \in Z\}$

Thus $\mathbf{Cell} \equiv \mathbf{Cell}[\mathbf{Val}]$.

Definition ($\rightarrow \xrightarrow{\mathbf{v}} \xrightarrow{\mathbf{\mu}}$): There are a number of function spaces in our world. The three simplest are total, partial and memory.

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

$X \xrightarrow{\mathbf{v}} Y = \{f \mid (\forall x \in X)(\forall y)(\mathbf{app}(f, x) \cong y \Rightarrow y \in Y)\}$

$X \xrightarrow{\mathbf{\mu}} Y = \{f \mid (\forall x \in X)(\mathbf{let}\{y := \mathbf{app}(f, x)\}\llbracket y \in Y \rrbracket)\}$

The first set denotes the set of total functions which do not alter existing memory (they may enlarge the domain of the memory by producing garbage (i.e. unreachable cells), but they may not alter existing cells); the second set denotes partial functions which also do not alter existing memory (again they may produce garbage); while the third set denotes partial functions which can possibly alter memory.

Now we can characterize the functionality of operations such as **mk**, and **get**, as follows.

(mk) $(\forall X)(\forall x \in X)(\mathbf{let}\{z := \mathbf{mk}(x)\}\llbracket z \in \mathbf{Cell}[X] \rrbracket)$

(get) $(\forall X)(\forall x \in \mathbf{Cell}[X])(\mathbf{get}(x) \in X)$

However (as we shall see in the last example of §4.) the analogous fact concerning **set** is false.

(set) $\neg(\forall X)(\forall x, y)(x \in X \wedge y \in \mathbf{Cell} \Rightarrow \mathbf{let}\{z := \mathbf{set}(y, x)\}\llbracket z \in \mathbf{Nil} \wedge y \in \mathbf{Cell}[X] \rrbracket)$

The **(mk)**, **(get)** and **(set)** can be restated as

$\lambda x.\mathbf{mk}(x) \in X \xrightarrow{\mathbf{\mu}} \mathbf{Cell}[X]$

$\lambda x.\mathbf{get}(x) \in \mathbf{Cell}[X] \rightarrow X$

$\lambda xy.\mathbf{set}(x, y) \in \mathbf{Cell}[X] \rightarrow Y \xrightarrow{\mathbf{\mu}} \mathbf{Nil}$

Now we give the promised counter example for classical (allE). Let

$$K = \{x \mid \Phi_{1\text{-cell}}\}$$

$$\Phi_{1\text{-cell}} \Leftrightarrow (\exists x)(\text{cell}(x) \cong \mathbf{t}) \wedge (\forall x, y)(\text{cell}(x) \cong \mathbf{t} \wedge \text{cell}(y) \cong \mathbf{t} \Rightarrow x \cong y)$$

$$\Phi = (\forall x \in X)(\text{let}\{z := \text{mk}(x)\}[\![z \in \text{Cell}[X]\!]])$$

Note that Φ is the body of the mk functionality formula. Now $(\forall X)\Phi$ holds, but $\Phi[K]$ fails for any memory with singleton domain. In fact $(\forall X)\Phi$ holds since the meaning of the class variable is determined at the outset, and remains unchanged throughout. The meaning of the class K is instead computed twice, with respect to two different memories. The second time it denotes the empty class.

Class membership expresses a very restricted form of non-expansiveness, allowing neither expansion of memory domain nor change in contents of existing cells. Let $\Phi_{\neg\text{expand}}(e)$ stand for the formula

$$(\forall X)(X \equiv \text{Cell} \Rightarrow \text{seq}(e, [\![X \equiv \text{Cell}\!]])).$$

Then $\Phi_{\neg\text{expand}}(e)$ says that execution of e does not expand the memory, although it might modify contents of existing cells.

To illustrate some of the subtleties regarding class membership, and notions of expansiveness, consider the following expressions:

$$e_0 = \lambda x.\text{mk}(\text{nil})$$

$$e_1 = \text{let}\{z := \text{mk}(\text{nil})\}\lambda x.z$$

$$e_2 = \text{seq}(\text{if}(\text{cell}(y), \text{set}(y, \text{nil}), \text{nil}), \lambda x.\text{mk}(\text{nil}))$$

$$e_3 = \text{seq}(\text{if}(\text{cell}(y), \text{set}(y, \text{nil}), \text{nil}), \text{let}\{z := \text{mk}(\text{nil})\}\lambda x.z)$$

Then each of these expressions evaluates to a memory function mapping arbitrary values to cells containing nil . But they differ in the effects they have. e_0 is a value (and as such neither expands nor modifies memory). e_1 is not a value and is expansive (its evaluation enlarges the domain of memory) but does not modify existing memory. e_2 may modify existing memory, but does not expand it. e_3 is expansive, and it may modify existing memory. These observations can be expressed in the theory as follows. Let T be $\mathbf{Val} \xrightarrow{\mu} \mathbf{Cell}[\mathbf{Nil}]$, then

$$e_0 \in T \subseteq \mathbf{Val}$$

$$e_j \notin \mathbf{Val} \quad \text{for } 1 \leq j \leq 3$$

$$\text{let}\{x := e_j\}[\![x \in T]\!] \quad \text{for } 0 \leq j \leq 3$$

$$\Phi_{\neg\text{write}}(e_j) \quad \text{for } 0 \leq j \leq 1$$

$$\Phi_{\neg\text{expand}}(e_j) \quad \text{for } j \in \{0, 2\}$$

Feferman [4] proposes an explanation of ML types in the variable type framework. This gives a natural semantics to ML type expressions, but there are problems with polymorphism, even in the purely functional case. The collection of classes is

much to rich to be considered a type system. One problem that arises is that fixed-point combinators can not be uniformly typed over all classes. This problem arises even in the absence of memory [26, 27].

Theorem (FixTypeFails): Let Y_v by any fixed-point combinator (such that $f(Y_v(f)) \cong Y_v(f)$). Then it is not the case that

$$f \in ((A \xrightarrow{p} B) \rightarrow (A \xrightarrow{p} B)) \Rightarrow Y_v(f) \in (A \xrightarrow{p} B)$$

for all classes A, B .

Proof (FixTypeFails): Define the P to be the class of *strictly* partial maps from \mathbb{N} to \mathbb{N} , $P = \{g \in \mathbb{N} \xrightarrow{p} \mathbb{N} \mid (\exists n \in \mathbb{N})(\neg \Downarrow g(n))\}$. Let $f \cong \lambda p. \lambda n. \mathbf{if}(\mathbf{eq}(n, 0), n, p(n - 1))$. Then we can prove

$$(1) \quad f \in P \rightarrow P \quad (2) \quad Y_v(f) \in \mathbb{N} \rightarrow \mathbb{N}$$

(1) follows by simple properties of \mathbf{if} , \mathbf{eq} and arithmetic (2) follows by induction on \mathbb{N} using the fixed point property of Y . Consequently, $\neg(Y_v(f) \in P)$ \square

The situation becomes more problematic when references are added, even in the simply typed (or monomorphic) case. Naive attempts to represent ML types as classes fail in the sense that the ML inference rules are not valid. The essential feature of the ML type system, in addition to the inference rules, is the preservation of types during the execution of well-typed programs. In this sense they are more syntactic than semantic.

In the following we illustrate the problems that arise in trying to encode the monomorphic type system with higher-order functions and references (cf. [28, 29, 9, 6]). In this system types are built from base types, \mathbb{N} and \mathbf{Nil} , using the reference construction and a suitable function space constructor (provisionally denoted by $\xrightarrow{\lambda}$).

$$\mathbb{T} = \mathbb{N} + \mathbf{Nil} + (\mathbb{T} \xrightarrow{\lambda} \mathbb{T}) + \mathbf{ref}(\mathbb{T})$$

The typing judgement in this system is of the form $\{x_i : \tau_i \mid i < n\} \vdash e : \tau$ and the constants, for each $\tau \in \mathbb{T}$, have the following type

$$\mathbf{mk}_\tau : \tau \xrightarrow{\lambda} \mathbf{ref}(\tau) \quad \mathbf{get}_\tau : \mathbf{ref}(\tau) \xrightarrow{\lambda} \tau \quad \mathbf{set}_\tau : \mathbf{ref}(\tau) \xrightarrow{\lambda} \tau \xrightarrow{\lambda} \mathbf{Nil}$$

which will be encoded by the corresponding η -ized operations

$$\lambda x. \mathbf{mk}(x) \quad \lambda x. \mathbf{get}(x) \quad \lambda xy. \mathbf{set}(x, y)$$

To encode this system requires a class term $\underline{\tau}$ corresponding to the type expression τ , and a formula $\Phi:(e, \underline{\tau})$ encoding the typing judgement $e : \tau$. Using these we can represent the judgement $\{x_i : \tau_i \mid i < n\} \vdash e : \tau$ by the formula $\bigwedge_{i < n} \Phi:(x_i, \underline{\tau}_i) \Rightarrow \Phi:(e, \underline{\tau})$ To simplify matters we shall assume that base types are represented by their corresponding class constants, in other words $\underline{\mathbb{N}} = \mathbb{N}$ and $\underline{\mathbf{Nil}} = \mathbf{Nil}$. We also require that belonging to a class via Φ implies membership in that class:

$$\Phi:(e, \underline{\tau}) \Rightarrow \mathbf{let}\{z := e\}\llbracket z \in \underline{\tau} \rrbracket.$$

That this encoding is faithful amounts to requiring several conditions. These include:

- (1a) $\Phi:(x, \underline{\tau}) \Rightarrow \Phi:(\mathbf{mk}(x), \underline{\mathbf{ref}}(\tau))$
- (1b) $\Phi:(x, \underline{\mathbf{ref}}(\tau)) \Rightarrow \Phi:(\mathbf{get}(x), \underline{\tau})$
- (1c) $(\Phi:(x, \underline{\mathbf{ref}}(\tau)) \wedge \Phi:(y, \underline{\tau})) \Rightarrow \Phi:(\mathbf{set}(x, y), \mathbf{Nil})$
- (2) $\bigwedge_{i < n} \Phi:(x_i, \underline{\tau}_i) \Rightarrow ((\Phi:(x, \underline{\tau}_a) \Rightarrow \Phi:(e, \underline{\tau}_b)) \Rightarrow \Phi:(\lambda x.e, \underline{\tau}_a \xrightarrow{\lambda} \underline{\tau}_b))$
- (3) $\bigwedge_{i < n} \Phi:(x_i, \underline{\tau}_i) \Rightarrow ((\Phi:(e, \underline{\tau}_a \xrightarrow{\lambda} \underline{\tau}_b) \wedge \Phi:(e_a, \underline{\tau}_a)) \Rightarrow \Phi:(\mathbf{app}(e, e_a), \underline{\tau}_b))$

The principle of *type faithfulness* [1]

$$e_a \cong e_b \Rightarrow ((\bigwedge_{i < n} \Phi:(x_i, \underline{\tau}_i) \Rightarrow \Phi:(e_a, \underline{\tau})) \Rightarrow (\bigwedge_{i < n} \Phi:(x_i, \underline{\tau}_i) \Rightarrow \Phi:(e_b, \underline{\tau})))$$

is also a desirable property. Note that in our framework this implies that a subterm of a typable term need not be typable. This can be regarded as an advantage of a semantic approach to types over the syntactic approach.

The simplest encoding would be to take

$$\underline{\mathbf{ref}}(\tau) = \mathbf{Cell}[\underline{\tau}] \quad \underline{\tau}_a \xrightarrow{\lambda} \underline{\tau}_b = \underline{\tau}_a \xrightarrow{\mu} \underline{\tau}_b \quad \Phi:(e, \underline{\tau}) = \mathbf{let}\{z := e\}[\![z \in \underline{\tau}]\!]$$

However the usual inference rules are not sound under this encoding. One source of trouble is that in **(3)** the evaluation of e_a may invalidate the assumptions that $\bigwedge_{i < n} \Phi:(x_i, \underline{\tau}_i)$. A counterexample to this is the following:

$$\begin{aligned} y \in \mathbf{Cell}[\mathbb{N}] &\Rightarrow \mathbf{let}\{x := \mathbf{set}(y, \mathbf{t})\}[\![x \in \mathbf{Nil}]\!] \\ y \in \mathbf{Cell}[\mathbb{N}] &\Rightarrow \mathbf{let}\{z := \lambda w.\mathbf{get}(y)\}[\![z \in \mathbf{Nil} \rightarrow \mathbb{N}]\!] \end{aligned}$$

But the resulting conclusion

$$y \in \mathbf{Cell}[\mathbb{N}] \Rightarrow \mathbf{let}\{z := \mathbf{app}(\lambda w.\mathbf{get}(y), \mathbf{set}(y, \mathbf{t}))\}[\![z \in \mathbb{N}]\!]$$

is clearly false.

To ensure that this phenomenon is ruled out, one would like $\Phi:$ to have the following property: if e is typed, and its evaluation alters the contents of a cell, then any type that cell had before evaluation it has after evaluation. Similarly for values of functional type. We can express this as the following schema:

$$(\forall z)(\bigwedge_{i < n} \Phi:(x_i, \underline{\tau}_i) \Rightarrow ((\Phi:(e, \underline{\tau}) \wedge \Phi:(z, \underline{\tau}')) \Rightarrow \mathbf{let}\{w := e\}[\![\Phi:(w, \underline{\tau}) \wedge \Phi:(z, \underline{\tau}')]\!]))$$

Since this property is not true for general classes the existence of such a $\Phi:$ seems doubtful.

To see that classes are too rich to be preserved by evaluation, consider the following class:

$$A = \{x \in \mathbf{Cell} \mid (\exists n \in \mathbb{N})(\mathbf{get}^n(x) \cong \mathbf{nil})\}$$

where $\mathbf{get}^n(x)$ is the n th iterate of \mathbf{get} , definable by simple recursion. Now observe that

$$x \in \mathbf{Cell}[A] \Rightarrow x \in A$$

Consequently

$$x \in \mathbf{Cell}[A] \Rightarrow \mathbf{let}\{z := \mathbf{set}(x, x)\}[\![z \in \mathbf{Nil}]\!]$$

$$x \in \mathbf{Cell}[A] \Rightarrow \lambda y. x \in \mathbf{Nil} \rightarrow \mathbf{Cell}[A]$$

But also note that

$$(\forall x \in \mathbf{Cell}[A])(\mathbf{let}\{z := \mathbf{app}(\lambda y. x, \mathbf{set}(x, x))\}[\![z \notin \mathbf{Cell}[A]]\!])$$

5. Issues and Conclusions

In this paper we have presented a logic, VTLoE, for specifying and reasoning about programs with effects. The semantics of this logic is based on a notion of program equivalence relative to a given memory. VTLoE goes well beyond traditional programming logics, such as Hoare's and Dynamic logic:

(1) The underlying programming language is a rich language based on the call-by-value lambda calculus extended by the reference primitives \mathbf{mk} , \mathbf{set} , \mathbf{get} , as well as constants representing traditional forms of atomic and structured data.

(2) In our language atoms, references and lambda abstractions are all first class values and as such are storable.

(3) The separation of mutation and variable binding allows us to avoid the problems that typically arise (e.g. in Hoare's and dynamic logic) from the conflation of program variables and logical variables.

(4) The equality and sharing of references (aliasing) may be directly expressed and reasoned about.

(5) The combination of mutable references and lambda abstractions allows us to study object-based programming within VTLoE.

(6) Central to VTLoE is the ability to express the operational equivalence of programs, a very general notion of program equivalence.

(7) In addition to the usual first-order formula constructions and quantification over class variables, the logic includes contextual assertions. This allows for direct reasoning about changes in state, and subsumes the state-based reasoning methods possible in both Hoare's and Dynamic logic.

(8) Class membership and quantification allow us to express a wide variety of useful notions including presence of effects, functionality, and structural induction principles (the latter issue is not dealt here).

The logic presented here is perhaps best viewed as a starting point for further research rather than a final product. In particular there are at least four directions for further research.

(A) There seems to be good evidence to support a *more localized* semantics for contextual assertions. One indication is the failure of the following principle:

$$e_0 \cong e_1 \Rightarrow \mathbf{let}\{x := e_0\}[\Phi] \Leftrightarrow \mathbf{let}\{x := e_1\}[\Phi]$$

This principle is false even for quantifier free Φ . In particular operational equivalence does not preserve membership in classes. One counterexample is:

$$e_0 = \lambda y.y \quad e_1 = \mathbf{let}\{z := \mathbf{mk}(\lambda y.y)\}\lambda w.\mathbf{app}(\mathbf{get}(z), w) \quad \Phi = x \in \{x \mid x \cong \lambda y.y\}$$

The problem is that the reduction contexts allowed in the atomic clause are allowed to alter the contents of any cell in memory, regardless of whether or not that cell is local. Similarly it may be that by restricting the quantifiers to range over *visible* or *non-local* values, the resulting logic will have nicer metatheoretic properties.

(B) At present there are very simple *valid* principles that VTLoE as axiomatized herein does not establish. One simple example is:

$$\mathbf{let}\{z := \mathbf{get}(y)\}\mathbf{let}\{x := \mathbf{mk}(v)\}[\mathbf{get}(y) \cong z]$$

Even though it appears to be related to **(get.i)** and **(mk.iv)**, it does not follow from them. What is lacking is any way of reasoning about the equivalence of contexts, not just expressions. It will probably be fruitful to extend the formal system to include assertions concerning the equivalence of contexts as well as expressions. For example we could introduce a new judgement of the form

$$U_0 \cong_X U_1$$

to mean that the contexts are equivalent with respect to a set X of variables allowed trapped.

Two important concepts that appear to lie outside the realm of VTLoE are the ability to express type information, and the ability to perform some sort of computation induction.

(C) Since it appears that *types* cannot be encoded as *classes* it may be that some type structure, via a new form of judgment, should be built in from the beginning.

(D) A powerful semantic method for establishing laws of program equivalence is computation induction, induction on the length of computation. Unfortunately, by its very nature, computation induction does not yield readily to axiomatization in a formal theory that admits non-trivial equivalences. This is due to the difficulty of maintaining a dual view of programs as descriptions of computations and programs as black boxes within a single formal theory. One approach to solving this problem is to extend the logic to include an ordering $e_0 \sqsubseteq e_1$ that expresses the operational approximation of computations. Finite projection operations modeled on the finite projections of domain theory can hopefully be used to prove inductive properties of \sqsubseteq .

Aknowledgements

This research was partially supported by DARPA contract NAG2-703, NSF grants CCR-8917606, CCR-8915663, and CCR-9109007, and Italian grant CNR-Stanford n.89.00002.26.

6. References

- [1] M. Abadi, B. Pierce, and G. Plotkin. Faithful ideal models for recursive polymorphic types. *International Journal of Foundations of Computer Science*, 2(1):1–21, 1991.
- [2] 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.
- [3] S. Feferman. A theory of variable types. *Revista Colombiana de Matemáticas*, 19:95–105, 1985.
- [4] 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.
- [5] M. Felleisen. *The Calculi of Lambda- ν -cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.
- [6] M. Felleisen and A. K. Wright. A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Rice University Computer Science Department, 1991.
- [7] J.Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *11th ACM Symposium on Principles of Programming Languages*, pages 245–257, 1983.
- [8] J.Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. From denotational to operational and axiomatic semantics for ALGOL-like languages: An overview. In E. Clarke and D. Kozen, editors, *Logics of Programs, Proceedings 1983*, volume 164 of *Lecture Notes in Computer Science*. Springer, Berlin, 1984.
- [9] X. Leroy and P. Wies. Polymorphic type inference and assignment. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages*, pages 291–302. ACM, 1990.
- [10] J. M. Lucassen. *Types and Effects, towards the integration of functional and imperative programming*. PhD thesis, MIT, 1987. Also available as LCS TR-408.
- [11] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *16th annual ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.
- [12] Z. Manna and R. Waldinger. Problematic features of programming languages. *Acta Informatica*, 16:371–426, 1981.
- [13] I. A. Mason and C. L. Talcott. Axiomatizing operational equivalence in the presence of side effects. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.
- [14] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 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] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *15th ACM Symposium on Principles of Programming Languages*, pages 191–208, 1988.
- [18] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.
- [19] J. H. Morris. *Lambda calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1968.
- [20] P.W. O’Hearn and R.D. Tennent. Semantic Analysis of Specification Logic, Part 2. *Information and Computation*, ?:-?, 199?

- [21] P.W. O'Hearn and R.D. Tennent. Semantics of Local Variables. Technical Report ECS-LFCS-92-192, Laboratory for foundations of computer science, University of Edinburgh, 1992.
- [22] E. Olderog. Hoare's logic for programs with procedures—what has been accomplished. In E. Clarke and D. Kozen, editors, *Logics of Programs, Proceedings 1983*, volume 164 of *Lecture Notes in Computer Science*. Springer, Berlin, 1984.
- [23] G. Plotkin. Call-by-name, call-by-value and the lambda-v-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [24] J. C. Reynolds. Syntactic control of interference. In *Conference record of the 5th annual ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.
- [25] K. Sieber. A partial correctness logic for programs (in an Algol-like language). In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*. Springer, Berlin, 1985.
- [26] Scott Fraser Smith. Partial objects in type theory. Technical Report TR 88-938, Department of Computer Science, Cornell University, 1988. Ph. D. thesis.
- [27] C. L. Talcott. A theory for program and data specification. In *Design and Implementation of Symbolic Computation Systems, DISCO'90*, volume 429 of *Lecture Notes in Computer Science*, pages 91–100. Springer-Verlag, 1990. Full version to appear in TCS special issue.
- [28] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, 1988.
- [29] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.