A Variable Typed Logic of 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

Abstract

In this paper we introduce a variable typed logic of effects inspired by the variable type systems of Feferman for purely functional languages. VTLoE (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 à la Plotkin), and contextual assertions. The second stage extends the logic to include classes and class membership. The logic we present provides an expressive language for defining and studying properties of programs including program equivalences, in a uniform framework. The logic combines the features and benefits of equational calculi as well as program and specification logics. In addition to the usual first-order formula constructions, we add *contextual assertions.* 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. We use the logic to establish the validity of the Meyer Sieber examples in an operational setting. The theory allows for the construction of inductively defined sets and derivation of the corresponding induction principles. We hope that classes may serve as a starting point for studying semantic notions of type. Naïve attempts to represent ML types as classes fail in sense that ML inference rules are not valid.

1. Introduction

It is well known that the addition of (unrestricted) 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. We give two versions of such a combinator, the first is written in the simply typed lambda calculus with ML reference primitives. The second is an untyped version written in the language we study in the sequel. These combinators are essentially identical to the one suggested by Landin (Landin, 1964).

Version 1: Suppose that in the type environment Γ we can establish $\Gamma \rhd g : \sigma \to \tau$. Then the desired functional is

$$\mathtt{Y} = \lambda\, F : (\sigma \to \tau) \to (\sigma \to \tau)\ .\ \mathtt{let}\ z := \mathtt{ref}\, g\ \mathtt{in}\ z := \lambda\, x : \sigma\ .\ (F(!\, z)(x))\ ;\ !\, z$$

and $\Gamma \rhd \mathtt{Y} : \sigma \to \tau$. Note that in the typed case we need an arbitrary element, g, of $\sigma \to \tau$ to be able to allocate a reference cell. A complication that does not arise in the untyped case:

Version 2: Suppose $\mathtt{mk}(v)$ allocates a cell with contents v, $\mathtt{get}(z)$ gets the current contents of the cell z, and $\mathtt{set}(z, v)$ sets the contents of the cell z to be v. Then the desired functional is

$$\mathtt{Y} = \lambda f.\mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\}\mathtt{seq}(\mathtt{set}(z, \lambda x.f(\mathtt{get}(z), x)), \mathtt{get}(z))$$

In addition, references are problematic for polymorphic type systems (Damas, 1985; Tofte, 1988; Tofte, 1990; Leroy and Wies, 1990). Even when references are only allowed to contain numbers they are troublesome from a denotational point of view as illustrated by the absence of fully abstract models. For example, in (Meyer and Sieber, 1988) they give a series of seven 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 (O'Hearn and Tennent, 1992; O'Hearn and Tennent, 1993b) 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, (Halpern et al., 1984; Halpern et al., 1983; Sieber, 1985; Olderog, 1984; Manna and Waldinger, 1981; O'Hearn and Tennent, 1992; O'Hearn and Tennent, 1993b), these equivalences, if expressible, must be independent of these systems. More recently (O'Hearn and Tennent, 1993a) gives a denotational model that handles the Meyer-Sieber examples and their variations correctly. This model uses a categorical notion called relational parametricity to capture locality. Full abstraction for this model remains an open problem.

In recent years various systems for reasoning about properties of programs written in general programming languages have been proposed, most notably Hoare's logic (Apt, 1981), Dynamic logic (Harel, 1984), Reynolds Specification Logic (Reynolds, 1982), Moggi's metalanguage for computational monads (Moggi, 1991), and Pitt's Evaluation Logic (Pitts, 1990). All are program logics of the *exogenous* kind, i.e. programs appear in formulas.

Hoare's logic is quite weak in the sense that it cannot express termination of programs, nor their equivalence. Dynamic logic can express termination but not equivalence, while the opposite is true for Specification logic. Hoare and Dynamic logic also suffer from the fact that they assign to bound variables, making variable instantiation problematic. Evaluation logic extends Moggi's metalanguage for computational monads 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 both of computation processes and of logical tools. Also VtloE is an exogenous logic, but while we try to axiomatize the properties of one canonical semantics for a very rich but specific language, trying to capture the reasoning principles peculiar to it, both Moggi's and Pitt's system are more in the LCF tradition and originate from recent advances in constructive Domain Theory. These systems are general systems which reduce 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. On the other hand our system concentrates on precisely one semantics, the natural operational one, and take the *operational equivalence* induced by that semantics as the primitive predicate. We seek simplicity of reasoning and expressivity of descriptions and justify our system on the basis of its power in deriving properties of one canonical semantics rather than on its general applicability. Finally we use classical logic, the systems of Moggi and Pitts are based on constructive and categorical Logic. Similarly, as Tennett points out, Reynolds has given examples showing that, Specification Logic must be intuitionistic, i.e. the classical law for double negation forces models of variables and assignment to be trivial.

1.1. Overview

In this paper we introduce a variable typed logic of effects 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 (Feferman, 1975; Feferman, 1979) and later applied to the study of purely functional languages (Feferman, 1985; Feferman, 1990). A similar extension incorporating non-local control effects was introduced in (Talcott, 1993). VTLoE (Variable Type 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. The second stage extends the logic to include classes and class membership.

The logic we present provides an expressive language for defining constraints and for studying properties and program equivalences, in a uniform framework. Our atomic formulas express the (operational or observational) equivalence of programs à la Plotkin (Plotkin, 1975). 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 logic combines the features and benefits of equational calculi and program and specification logics. The theory 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 (Tofte, 1990). Other effects can also be represented within the system. These include read/write effects (Lucassen, 1987; Lucassen and Gifford, 1988; Jouvelot and Gifford, 1991) and various forms of interference (Reynolds, 1978; Reynolds, 1982).

The paper is divided into three parts.

(I) The first part describes the syntax and operational semantics of our language, and establishes the basic results concerning the corresponding notion of operational equivalence. We also discuss some of the subtleties that arise due to the presence of mutable data. Most of the results in this part are from (Mason and Talcott, 1991a).

(II) In part two we describe the first order aspects of our logic of effects, and provide examples of their use. These examples include proofs within VTLoE of the validity of the examples of Meyer and Sieber.

(III) In the third part we enlarge our logic to allow for the manipulation of classes. We give an extended example of the use of VTLoE in specifying and verifying a program.

Part II is independent of part III, while much of part III is orthogonal to part II.

Earlier versions of part II of this paper appeared as (Mason and Talcott, 1992b). Earlier versions of part III of this paper appeared as (Honsell et al., 1993).

1.1.1. Part I

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.

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 operations and basic constants or atoms $\mathbb{A}$, (such as the Lisp booleans `t` and `nil` as well as the natural numbers $\mathbb{N}$). We can think of this language as an untyped dialect of ML.

The semantics of expressions is a call-by-value evaluation relation given by a reduction relation on syntactic entities. These syntactic entities represent the state of an abstract machine. A state has three components: the current instruction, the current continuation, and the current state of memory. Their syntactic counterparts are *redexes*, *reduction contexts*, and *memory contexts* respectively.

We define the operational equivalence relation and study its general properties. Our definition extends the equivalence relations defined in (Morris, 1968) and (Plotkin, 1975). In general it is very difficult to establish the operational equivalence of expressions. Thus it is desirable to have a simpler characterization of operational equivalence, one that limits the class of contexts (or observations) that must be considered. In (Mason and Talcott, 1991a) 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.

In the presence of effects several notions split into spectrums of variations. For example, in the presence of effects there are several degrees of *definedness*. 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 an expression evaluating to a value, without altering (but possibly enlarging) memory. An even stronger notion of definedness is that of an 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:

$$\mathtt{mk}(x) \qquad \mathtt{seq}(\mathtt{mk}(x), 1) \qquad \mathtt{get}(x) \qquad 1.$$

None of these notions are equivalent in the sense that terms of one kind are not in general operationally equivalent to terms of the other kind. These notions are formalizable in our theory of classes, see §5.4. Such distinctions mean that care must be taken in generalizing notions such as extensionality, η-conversion and fixed-point operators.

1.1.2. Part II

The first order fragment of VTLoE is a minor generalization of classical first order logic. The atomic formulas of our language assert the operational equivalence of expressions. These expressions are just terms in our call-by-value lambda calculus. In addition to the usual first-order formula constructions, we add *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 and evaluation 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 (Mason and Talcott, 1992a) where the corresponding rules had complicated side-conditions.

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. The underlying logic is classical. Consequently we concentrate on the properties of contextual assertions and their uses.

1.1.3. Part III

Using methods of (Feferman, 1975; Feferman, 1990) and (Talcott, 1993), we extend our theory to include a general theory of classifications (classes for short). We extend the syntax to include class terms (either class variables, class constants, or comprehension terms). We extend the set of formulas to include class membership and quantification over classes. Class variables range over sets of values closed under operational equivalence.

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. We show that by using classes one can specify and verify properties of non-trivial programs.

Just as in the case of definedness, in the presence of effects 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 $X_1, \ldots, X_n \overset{\mu}{\to} Y$ with arguments in $X_1, \ldots, X_n$ 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 (Lucassen, 1987;

Lucassen and Gifford, 1988; Jouvelot and Gifford, 1991). 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.

It was hoped that classes would serve as a starting point for studying semantic notions of type. (Feferman, 1990) 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 $(\forall X, Y)([X \to Y] \to [X \to Y] \to [X \to Y])$ 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. Naïve attempts to represent ML types as classes fails in sense that ML inference rules are not valid. Its 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. Our analysis indicates that ML types are therefore more syntactic than semantic.

1.2. 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. $Y_0 \xrightarrow{f} Y_1$ 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, ... range over $\mathbb{N}$.

Part I: Semantic Foundation

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 (atoms) and primitive operations. The semantics is given operationally in terms of a single step reduction relation. We shall be somewhat terse in this section. For a more leisurely treatment see (Mason and Talcott, 1991a).

2.1. Syntax of Terms

We fix a countably infinite set of variables, $\mathbb{X}$, a countable set of atoms, $\mathbb{A}$, and a family of operation symbols $\mathbb{F} = \{\mathbb{F}_n \mid n \in \mathbb{N}\}$ ($\mathbb{F}_n$ is a set of n-ary operation symbols) with $\mathbb{X}$, $\mathbb{A}$, $\mathbb{F}_n$ for $n \in \mathbb{N}$. All these sets are assumed to be pairwise disjoint. We assume $\mathbb{A}$ contains two distinct elements playing the role of booleans, `t` for *true* and `nil` for *false*. From the given sets we define expressions, value expressions, contexts, and value substitutions.

Definition ($\mathbb{L}$ $\mathbb{V}$ $\mathbb{P}$ $\mathbb{S}$ $\mathbb{E}$): The set of λ-abstractions, $\mathbb{L}$, the set of value expressions, $\mathbb{V}$, the set of immutable pairs, $\mathbb{P}$, the set of value substitutions, $\mathbb{S}$, and the set of expressions, $\mathbb{E}$, are defined, mutually recursively, as the least sets satisfying the following equations:

$$\mathbb{L} = \lambda\mathbb{X}.\mathbb{E}$$

$$\mathbb{V} = \mathbb{X} + \mathbb{A} + \mathbb{L} + \mathbb{P}$$

$$\mathbb{P} = \mathtt{pr}(\mathbb{V}, \mathbb{V})$$

$$\mathbb{S} = \mathbb{X} \xrightarrow{\mathrm{f}} \mathbb{V}$$

$$\mathbb{E} = \mathbb{V} + \mathtt{app}(\mathbb{E}, \mathbb{E}) + \mathbb{F}_n(\mathbb{E}^n)$$

We let a range over $\mathbb{A}$, x, y, z range over $\mathbb{X}$, v range over $\mathbb{V}$, ρ range over $\mathbb{L}$, σ range over $\mathbb{S}$, and e range over $\mathbb{E}$. Note that the structured data, $\mathbb{P}$, are taken to be values. λ is a binding operator and free and bound variables of expressions are defined as usual. Two expressions are considered equal if they are the same up to renaming of bound variables. $\mathrm{FV}(e)$ is the set of free variables of e. $e^{\{x:=e'\}}$ is the result of substituting e' for x in e taking care not to trap 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 *closed expression* is an expression with no free variables. Thus $\mathbb{E}_\emptyset$ is the set of all closed expressions. We write $\{x_i := v_i \mid i < n\}$ for the substitution σ with domain $\{x_i \mid i < n\}$ such that $\sigma(x_i) = v_i$ for $i < n$. e^σ is the result of simultaneous substitution of free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$, again taking care not to trap variables.

The operations, $\mathbb{F}$, are partitioned into memory operations and operations that are independent of memory.[1] A memory operation may modify memory, and its result may depend on state of memory when it is executed. The memory operations are:

$$\{\mathtt{get},\mathtt{mk}\} \subseteq \mathbb{F}_1 \quad \{\mathtt{set}\} \subseteq \mathbb{F}_2.$$

The remaining operations neither affect the memory, nor are affected by the memory. In this paper we explicitly include operations for dealing with immutable pairs, a branching operation, and assorted predicates.

$$\{\mathtt{atom},\mathtt{cell},\mathtt{fst},\mathtt{snd},\mathtt{ispr},\mathtt{isnat},\mathtt{+1},\mathtt{-1}\} \subseteq \mathbb{F}_1 \qquad \{\mathtt{eq},\mathtt{pr}\} \subseteq \mathbb{F}_2 \qquad \{\mathtt{br}\} \subseteq \mathbb{F}_3$$

In addition there are operations on numbers which we shall not enumerate explicitly. As usual we use the syntactic sugar `let`, `if`, `seq` (a sequencing construct akin to `progn`, `begin` or `;`) to make programs more readable. We also will sometimes write $e_0(e_1)$ instead of $\mathtt{app}(e_0, e_1)$. For example,

$e_0(e_1)$ abbreviates $\mathtt{app}(e_0, e_1)$

$\mathtt{let}\{x := e_0\}e_1$ abbreviates $\mathtt{app}(\lambda x.e_1, e_0)$

$\mathtt{seq}(e_0, e_1)$ abbreviates $\mathtt{app}(\mathtt{app}(\lambda z.\lambda x.x, e_0), e_1)$

$\mathtt{if}(e_0, e_1, e_2)$ abbreviates $\mathtt{app}(\mathtt{br}(e_0, \lambda z.e_1, \lambda z.e_2), \mathtt{nil})$ for z fresh

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

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

We let C range over $\mathbb{C}$. $C[e]$ denotes the result of replacing any hole in C by e. Free variables of e may become bound in this process, we let $\mathrm{Traps}(C)$ be those variables which may be trapped.

Definition (Traps):

$$\mathrm{Traps}(C) = \begin{cases} \emptyset & \text{if } C \in \mathbb{X} \cup \mathbb{A} \cup \{\bullet\} \\ \mathrm{Traps}(C_0) \cup \{z\} & \text{if } C = \lambda z.C_0 \\ \mathrm{Traps}(C_0) \cup \mathrm{Traps}(C_1) & \text{if } C = \mathtt{app}(C_0, C_1) \\ \mathrm{Traps}(C_1) \cup \ldots \cup \mathrm{Traps}(C_n) & \text{if } C = \vartheta(C_1, \ldots, C_n) \text{ and } \vartheta \in \mathbb{F}_n \end{cases}$$

[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 affected by and can affect computation state.

2.2. Semantics of Terms

The operational semantics of expressions is given by a reduction relation $\overset{*}{\mapsto}$ on a syntactic representation of the state of an abstract machine, referred to as *computation descriptions*. A state has three components: the current instruction, the current continuation, and the current state of memory. Their syntactic counterparts are *redexes*, *reduction contexts*, and *memory contexts* respectively. Redexes describe the primitive computation steps. A primitive step is either a β_{value}-reduction or the application of a primitive operation to a sequence of value expressions.

Definition ($\mathbb{E}_{\mathrm{r}}$): The set of redexes, $\mathbb{E}_{\mathrm{r}}$, is defined as

$$\mathbb{E}_{\mathrm{r}} = \mathtt{app}(\mathbb{V}, \mathbb{V}) + \bigcup_{n \in \mathbb{N}} (\mathbb{F}_n(\mathbb{V}^n) - \mathbb{P})$$

Reduction contexts identify the subexpression of an expression that is to be evaluated next, they correspond to the standard reduction strategy (left-first, call-by-value) of (Plotkin, 1975) and were first introduced in (Felleisen and Friedman, 1986).

Definition ($\mathbb{R}$): The set of reduction contexts, $\mathbb{R}$, is the subset of $\mathbb{C}$ defined by

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

We let R range over $\mathbb{R}$. An expression is either a value expression or decomposes uniquely into a redex placed in a reduction context.

Lemma (Decomposition): If $e \in \mathbb{E}$ then either $e \in \mathbb{V}$ or e can be written uniquely as $R[e']$ where R is a reduction context and $e' \in \mathbb{E}_{\mathrm{r}}$.

Definition ($\mathbb{M}$): The set of memory contexts, $\mathbb{M}$, is the set of contexts Γ of the form

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

where $z_i \neq z_j$ when $i \neq j$. We include the possibility that $n = 0$, in which case $\Gamma = \bullet$. We let Γ range over $\mathbb{M}$.

We have divided the memory 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 can view memory contexts as finite maps from variables to value expressions. Hence we define the domain of Γ (as above) to be $\mathrm{Dom}(\Gamma) = \{z_1, \ldots, z_n\}$, and $\Gamma(z_i) = v_i$ for $1 \leq i \leq n$. Two memory contexts are considered the same if they are the same when viewed as functions. Viewing memory contexts and finite maps, we define the modification of memory contexts, $\Gamma\{z := \mathtt{mk}(v)\}$, and the union of two memory contexts, $(\Gamma_0 \cup \Gamma_1)$, in the obvious way. Γ^{σ} is the result of applying σ to each value in the range of Γ.

Definition ($\mathbb{D}$): The set of computations descriptions (briefly descriptions), $\mathbb{D}$, is defined to be the set $\mathbb{M} \times \mathbb{E}$. Thus a description is a pair with first component a memory context and second component an arbitrary expression. We do not require that the free variables of the expression be contained in the domain of the memory context. This allows us to define reductions uniformly in parameters that are not touched by the reduction step, and hence to provide a form of symbolic evaluation. We let $\Gamma; e$ range over $\mathbb{D}$. A *closed* description is a description of the form $\Gamma; e$ where $e \in \mathbb{E}_{\mathrm{Dom}(\Gamma)}$. *Value descriptions* are descriptions whose expression component is a value expression, i.e. a description of the form $\Gamma; v$. $(\Gamma; e)^{\sigma} = \Gamma^{\sigma}; e^{\sigma}$.

Definition ($\overset{*}{\mapsto}$): The reduction relation $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$. The clauses are:

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

$$(\text{atom}) \quad \Gamma; R[\mathtt{atom}(v)] \mapsto \begin{cases} \Gamma; R[\mathtt{t}] & \text{if } v \in \mathbb{A} \\ \Gamma; R[\mathtt{nil}] & \text{if } v \in \mathbb{L} \cup \mathbb{P} \cup \mathrm{Dom}(\Gamma) \end{cases}$$

$$(\text{ispr}) \quad \Gamma; R[\mathtt{ispr}(v)] \mapsto \begin{cases} \Gamma; R[\mathtt{t}] & \text{if } v \in \mathbb{P} \\ \Gamma; R[\mathtt{nil}] & \text{if } v \in \mathbb{A} \cup \mathbb{L} \cup \mathrm{Dom}(\Gamma) \end{cases}$$

$$(\text{cell}) \quad \Gamma; R[\mathtt{cell}(v)] \mapsto \begin{cases} \Gamma; R[\mathtt{t}] & \text{if } v \in \mathrm{Dom}(\Gamma) \\ \Gamma; R[\mathtt{nil}] & \text{if } v \in \mathbb{L} \cup \mathbb{P} \cup \mathbb{A} \end{cases}$$

$$(\text{eq}) \quad \Gamma; R[\mathtt{eq}(v_0, v_1)] \mapsto \begin{cases} \Gamma; R[\mathtt{t}] & \text{if } v_0 = v_1,\ v_0, v_1 \in \mathrm{Dom}(\Gamma) \cup \mathbb{A}, \\ \Gamma; R[\mathtt{nil}] & \text{if the above fails, and } \mathrm{FV}(v_0, v_1) \subseteq \mathrm{Dom}(\Gamma). \end{cases}$$

$$(\text{fst}) \quad \Gamma; R[\mathtt{fst}(\mathtt{pr}(v_1, v_2))] \mapsto \Gamma; R[v_1]$$

$$(\text{snd}) \quad \Gamma; R[\mathtt{snd}(\mathtt{pr}(v_1, v_2))] \mapsto \Gamma; R[v_2]$$

$$(\text{br}) \quad \Gamma; R[\mathtt{br}(v_0, v_1, v_2)] \mapsto \begin{cases} \Gamma; R[v_1] & \text{if } v_0 \in (\mathbb{A} - \{\mathtt{nil}\}) \cup \mathbb{L} \cup \mathbb{P} \cup \mathrm{Dom}(\Gamma) \\ \Gamma; R[v_2] & \text{if } v = \mathtt{nil} \end{cases}$$

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

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

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

Note that in the `atom` and `cell` rules if one of the arguments is a variable not in the domain of the memory context then the primitive reduction step is not determined. This is also the case in the `eq`, `br`, `get`, and `set` rules.

Definition ($\downarrow \updownarrow$): A closed description, $\Gamma; e$ is *defined* (written $\downarrow \Gamma; e$) if it evaluates to a value description.

$$\downarrow(\Gamma; e) \;\Leftrightarrow\; (\exists \Gamma'; v')(\Gamma; e \overset{*}{\mapsto} \Gamma'; v')$$

For closed expressions e, we write $\downarrow e$ to mean $\downarrow \emptyset; e$ and $e_0 \updownarrow e_1$ to mean that $\downarrow e_0$ iff $\downarrow e_1$.

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):

(i) $\Gamma_0[e_0] = \Gamma_1[e_1]$ $\qquad$ if $\Gamma; e \mapsto \Gamma_i; e_i$ for $i < 2$

(ii) $R[\Gamma[e]] \overset{*}{\mapsto} \Gamma; R[e]$ $\qquad$ if $\mathrm{FV}(R) \cap \mathrm{Dom}(\Gamma) = \emptyset$.

(iii) $\Gamma; e \mapsto \Gamma'; e' \;\Rightarrow\; (\Gamma; e)^{\sigma} \mapsto (\Gamma'; e')^{\sigma}$ $\qquad$ if $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\sigma) = \emptyset$.

(iv) $\Gamma; e \mapsto \Gamma'; e' \;\Rightarrow\; (\Gamma_0 \cup \Gamma); e \mapsto (\Gamma_0 \cup \Gamma'); e'$ $\qquad$ if $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_0) = \emptyset$.

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

2.2.1. A Little History

The fact that one can present a syntactic reduction system for this, and related, imperative λ-calculi was discovered independently by three people in 1986-1987: Carolyn Talcott (Mason and Talcott, 1991a), Matthias Felleisen and Robert Hieb (Felleisen and Hieb, 1992). As well as being conceptually elegant, it has also provided the necessary tools for several key results and proofs. It provided the basis for an elegant revision of (Felleisen, 1987; Felleisen and Friedman, 1989) that was later published in (Felleisen and Hieb, 1992). Other notable successful uses of the technique is the type soundness proof, via subject reduction, of the imperative ML type system (Felleisen and Wright, 1991). The analysis of parameter passing in Algol (Crank and Felleisen, 1991; Weeks and Felleisen, 1993). The analysis of reduction calculi for Scheme-like languages (Sabry and Felleisen, 1993; Fields and Sabry, 1993). In 1987 Ian Mason realized that it provided the ideal notion of a normal form and symbolic evaluation needed in the completeness result presented in (Mason and Talcott, 1992a).

The approach Felleisen et al take is a reduction system on expressions of the form $\Gamma[e]$, while we adopt a slightly more complex reduction system on pairs of the form $\Gamma; e$. There is a technical reason for this: notice that letting

$$e_0 ::= \mathtt{if}(\mathtt{and}(\mathtt{cell}(x), \mathtt{eq}(\mathtt{get}(x), 0)), x, \mathtt{mk}(0))$$

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

we have, for any closing Γ, that $\Gamma[e_0]$ and $\Gamma[e_1]$ have a common reduct. They (e_o and e_1) are not operationally equivalent. The key distinction between $\Gamma; e$ and $\Gamma[e]$ is that renaming of variables in $\mathrm{Dom}(\Gamma)$ is allowed in $\Gamma[e]$ but not in $\Gamma; e$. This distinction, though somewhat technical, is important for the statement and proof of a number of results.

2.3. Operational Equivalence of Terms

In this section we define the operational equivalence relation and study its general properties. Operational[2] equivalence formalizes the notion of equivalence as black-boxes. Treating programs as black boxes requires only observing what effects and values they produce, and not how they produce them. Our definition extends the extensional equivalence relations defined by (Morris, 1968) and (Plotkin, 1975) to computation over memory structures. As shown by (Abramsky, 1990; Abramsky, 1991; Bloom, 1990; Egidi et al., 1992; Howe, 1989; Mason, 1986b; Mason and Talcott, 1991a; Jim and Meyer, 1991; Milner, 1977; Ong, 1988; Pitts and Stark, 1993; Smith, 1992; Talcott, 1985) operational equivalence and approximation can be characterized in various ways.

Definition ($\cong$): 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.

$$e_0 \cong e_1 \Leftrightarrow (\forall C \in \mathbb{C} \mid C[e_0], C[e_1] \in \mathbb{E}_\emptyset)(C[e_0] \updownarrow C[e_1])$$

[2] we use the term *operational* rather than *observational* because the latter suggests a class of equivalence relations (depending on the type of observations being made), whereas the former suggests the intrinsic computational nature of the relation.

2.3.1. Caveats

The operational equivalence is not trivial since the inclusion of branching implies that `t` and `nil` are not equivalent. By definition operational equivalence is a congruence relation on expressions:

$$\textbf{Congruence}: \quad e_0 \cong e_1 \;\Rightarrow\; (\forall C \in \mathbb{C})(C[e_0] \cong C[e_1])$$

However it is not necessarily the case that substitution instances of equivalent expressions are equivalent even if the instantiating expressions always returns a value. As a counter-example we have $\mathtt{if}(\mathtt{cell}(x), \mathtt{eq}(x, x), \mathtt{t}) \cong \mathtt{t}$ but $\mathtt{if}(\mathtt{cell}(\mathtt{mk}(\mathtt{t})), \mathtt{eq}(\mathtt{mk}(\mathtt{t}), \mathtt{mk}(\mathtt{t})), \mathtt{t}) \cong \mathtt{nil}$. The reason underlying this is that in the case of programs with effects, returning a value is not an appropriate characterization of definedness. In particular returning a value is not the same as being operationally equivalent to a value. This is in contrast to the purely functional case and is due to the presence of *effects*. For example, each of the following expressions always returns a value

$$\mathtt{mk}(x) \qquad \mathtt{if}(\mathtt{cell}(x), \mathtt{set}(x, y), x) \qquad \mathtt{if}(\mathtt{cell}(x), \mathtt{get}(x), x)$$

but none is equivalent to a value, i.e. for no value expression v do we have $e \cong v$ for any of the above three expressions. The first has an *allocation effect*. The second has a *write effect*. The third has a *read effect*. Similarly an expression of the form $\Gamma[\lambda x.e]$ is in general not operationally equivalent to a value.

Lemma (ex): Let e be $\mathtt{let}\{z := \mathtt{mk}(0)\}\lambda x.z$. Then e is not operationally equivalent to a value.

Proof (ex): This proof also serves to demonstrate the methods used in reasoning operationally. The principles we use will be justified shortly. Suppose to the contrary that $\mathtt{let}\{z := \mathtt{mk}(0)\}\lambda x.z \cong v$ for some $v \in \mathbb{V}$. Then by considering the context

$$\mathtt{eq}(\mathtt{app}(\bullet, 1), \mathtt{app}(\bullet, 1))$$

we have

$$\begin{aligned}
&\mathtt{eq}(\mathtt{app}(\mathtt{let}\{z := \mathtt{mk}(0)\}\lambda x.z, 1), \mathtt{app}(\mathtt{let}\{z := \mathtt{mk}(0)\}\lambda x.z, 1)) \\
&\quad \cong \mathtt{eq}(\mathtt{let}\{z := \mathtt{mk}(0)\}\mathtt{app}(\lambda x.z, 1), \mathtt{let}\{z := \mathtt{mk}(0)\}\mathtt{app}(\lambda x.z, 1)) \\
&\quad \cong \mathtt{eq}(\mathtt{let}\{z := \mathtt{mk}(0)\}z, \mathtt{let}\{z := \mathtt{mk}(0)\}z) \\
&\quad \cong \mathtt{eq}(\mathtt{mk}(0), \mathtt{mk}(0)) \cong \mathtt{nil}
\end{aligned}$$

Consequently by (**congruence**) $\mathtt{eq}(\mathtt{app}(v, 1), \mathtt{app}(v, 1)) \cong \mathtt{nil}$. But by considering the context

$$\mathtt{let}\{y := \bullet\}\mathtt{eq}(\mathtt{app}(y, 1), \mathtt{app}(y, 1))$$

we have

$$\begin{aligned}
&\texttt{let}\{y := e\}\texttt{eq}(\texttt{app}(y,1),\texttt{app}(y,1))\\
&\quad\cong \texttt{let}\{z := \texttt{mk}(0)\}\texttt{let}\{y := \lambda x.z\}\texttt{eq}(\texttt{app}(y,1),\texttt{app}(y,1)) \qquad \text{by evaluation}\\
&\quad\cong \texttt{let}\{z := \texttt{mk}(0)\}\texttt{let}\{y := \lambda x.z\}\texttt{eq}(z,z) \qquad \text{by evaluation}\\
&\quad\cong \texttt{let}\{z := \texttt{mk}(0)\}\texttt{eq}(z,z) \qquad \text{by } \beta_{\text{value}}\\
&\quad\cong \texttt{let}\{z := \texttt{mk}(0)\}\texttt{t} \qquad \text{by evaluation}\\
&\quad\cong \texttt{t} \qquad \text{by garbage collection}
\end{aligned}$$

On the other hand

$$\begin{aligned}
&\texttt{let}\{y := v\}\texttt{eq}(\texttt{app}(y,1),\texttt{app}(y,1))\\
&\quad\cong \texttt{eq}(\texttt{app}(v,1),\texttt{app}(v,1)) \qquad \text{by } \beta_{\text{value}}\\
&\quad\cong \texttt{nil} \qquad \text{by above}
\end{aligned}$$

This is a contradiction. $\square_{\mathbf{ex}}$

This distinction means that care must be taken in generalizing notions such as η-conversion and fixed-point operators.

The η rule for the pure lambda calculus has the form $e \cong \lambda x.e(x)$ if x is not free in e. In an applied calculus where there are objects that are not functions we need the additional restriction that e must denote a function. In the presence of memory objects, if we interpret *e denotes a function* as $e \cong \Gamma[\lambda x.e_0]$ for some memory context Γ, then the restricted η rule is not valid. If we interpret *e denotes a function* as $e \cong \lambda x.e_0$, then the restricted η rule is valid.

Lemma ($\neg\eta$): In general $\lambda x.(\Gamma[\lambda x.e])x$ is not operationally equivalent to $\Gamma[\lambda x.e]$.

Proof ($\neg\eta$): As a counter-example we have

$$\texttt{let}\{z := \texttt{mk}(0)\}\bullet;\lambda x.\texttt{let}\{y := \texttt{get}(z)\}\texttt{seq}(\texttt{set}(z,x),y).$$

$\square$

2.3.2. Closed Instantiations and Uses

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. In the simple world of the call-by-value lambda calculus it can be shown that two terms are operationally equivalent iff all closed instantiations are operationally equivalent. If we define a *closed use* of an expression to be the placement of the expression in a closed reduction context, then all closed instantiations of two expressions are operationally equivalent iff all uses of all closed uses are equidefined. In symbols:

$$\begin{aligned}
e_0 \cong e_1 &\Leftrightarrow (\forall \sigma \in \mathbb{S} \mid e_0^\sigma, e_1^\sigma \in \mathbb{E}_\emptyset)(e_0^\sigma \cong e_1^\sigma)\\
&\Leftrightarrow (\forall \sigma \in \mathbb{S} \mid e_0^\sigma, e_1^\sigma \in \mathbb{E}_\emptyset)(\forall R \in \mathbb{R}_\emptyset)(R[e_0^\sigma] \updownarrow R[e_1^\sigma])
\end{aligned}$$

In a world with state the notion of a closed instantiation must be suitably generalized. It is no longer appropriate to instantiate the free variables of an expression merely by closed values. For this reason we redefine 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. However the analogous characterization of operational equivalence is false. There exist two non-operationally equivalent expressions whose closed instantiations (even in the generalized sense) are all operationally equivalent. Two examples of this phenomenon are:

$$\mathtt{seq}(\mathtt{set}(z, \lambda x.x), \lambda x.x) \quad \text{and} \quad \mathtt{seq}(\mathtt{set}(z, \lambda x.x), \lambda x.\mathsf{app}(\mathtt{get}(z), x))$$

$$\mathtt{mk}(\mathtt{get}(x)) \quad \text{and} \quad \mathtt{seq}(\mathtt{mk}(\mathtt{get}(x)), x)$$

The key observation is that arbitrary contexts may create memory that is shared by expressions and their uses. Thus we can obtain a characterization of operational equivalence by considering closed instantiations of uses. A *use* of an expression e is the placement of e into a reduction context, not necessarily closed. We then have the result that two expressions are operationally equivalent just if all closed instantiations of all uses are equidefined. This latter property is called (**ciu**) equivalence and is a weak form of extensionality. [3]

Theorem (ciu): $e_0 \cong e_1 \Leftrightarrow (\forall \Gamma, \sigma, R)(\mathrm{FV}(\Gamma[R[e_i^{\sigma}]]) = \emptyset \Rightarrow (\Gamma[R[e_0^{\sigma}]] \updownarrow \Gamma[R[e_1^{\sigma}]]))$

A proof of (**ciu**) appears in (Mason and Talcott, 1991a), below we give a simpler proof. Using this theorem we can easily establish, for example, the validity of the let-rules of the lambda-c calculus (Moggi, 1989) (see also (Talcott, 1993) where these laws are established for a language with control abstractions).

Corollary ($\mathtt{let}_c$):

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

(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.

Another nice property that is easily established using (**ciu**) is that reduction preserves operational equivalence:

Lemma (eval): $\Gamma; e \mapsto \Gamma'; e' \Rightarrow \Gamma[e] \cong \Gamma'[e']$.

This property is the basis of the calculi found in (Felleisen and Hieb, 1992).

2.3.3. The Proof of the (ciu) Theorem

Defining

$$e_0 \cong^{ciu} e_1 \Leftrightarrow (\forall \Gamma, \sigma, R)(\mathrm{FV}(\Gamma[R[e_i^{\sigma}]]) = \emptyset \Rightarrow (\downarrow \Gamma[R[e_0^{\sigma}]] \Leftrightarrow \downarrow \Gamma[R[e_1^{\sigma}]])),$$

the (**ciu**) theorem may be restated as

Theorem (ciu): $e_0 \cong e_1 \Leftrightarrow e_0 \cong^{ciu} e_1$.

[3] Notice that in the absence of memory a closed instantiation of a use of an expression is the same as closed use of a closed instantiation since $R[e]^{\sigma} = R^{\sigma}[e^{\sigma}]$. This equivalence fails in the world with memories.

We give a proof that synthesizes ideas from proofs in (Smith, 1992; Mason and Talcott, 1991a; Howe, 1989). We first give an informal overview of the proof. The ($\Rightarrow$) direction is not difficult, since $\cong^{ciu}$ has a smaller collection of contexts to distinguish expressions than $\cong$ has. ($\Leftarrow$) is the difficult direction. This proof uses the observation that it suffices to show $\cong^{ciu}$ is a congruence, $e_0 \cong^{ciu} e_1 \Rightarrow C[e_0] \cong^{ciu} C[e_1]$, because $\cong^{ciu}$ is stronger than equitermination. To show congruence, we prove lemmas that establish congruence for single-constructors: operators $\mathbb{F}$, app, and λx may be placed around the e_i's and preserve $\cong^{ciu}$. The first lemma (**Op ciu**) proves this property for $\mathbb{F}$ and app, and (**Lambda ciu**) proves the lambda case. C is then constructed by induction on its size, proving (**ciu**).

Lemma (Op ciu): $e_0 \cong^{ciu} e_1 \Rightarrow \delta(\bar{e}_a, e_0, \bar{e}_b) \cong^{ciu} \delta(\bar{e}_a, e_1, \bar{e}_b)$ for any $\delta \in \mathbb{F} \cup \{\mathsf{app}\}$, $\bar{e}_a, \bar{e}_b$.

Proof (Op-ciu): Pick arbitrary $\delta, \Gamma, R, \sigma$ such that $\Gamma[R[(\delta(\bar{e}_a, e_j, \bar{e}_b))^\sigma]] \in \mathbb{E}_\emptyset$ for $j < 2$. Since δ does not bind, σ may be factored in, so it suffices to show

$$\downarrow\Gamma; R[\delta(\bar{e}_a^\sigma, e_0^\sigma, \bar{e}_b^\sigma))] \Rightarrow \downarrow\Gamma; R[\delta(\bar{e}_a^\sigma, e_1^\sigma, \bar{e}_b^\sigma))]$$

(the converse is symmetric). Proceed by induction on the length of the computation of the assumption. Since $\bar{e}_a$ and $\bar{e}_b$ are arbitrary we drop the σ and assume $\mathrm{FV}(\bar{e}_a) \subseteq \mathrm{Dom}(\Gamma)$ and $\mathrm{FV}(\bar{e}_b) \subseteq \mathrm{Dom}(\Gamma)$.

Assume the conclusion is true for all $\Gamma, \bar{e}_a, \bar{e}_b$ with shorter computations. Proceed by cases on whether all elements $\bar{e}_a$ are values. Suppose so (or if $\bar{e}_a$ is empty): then, define $R_0 = R[\delta(\bar{e}_a, \bullet, \bar{e}_b))]$, and the conclusion follows directly by assumption. Suppose not. Then there is some $e_{a,i}$ such that $e_{a,i} \notin \mathbb{V}$ and $e_{a,k} \in \mathbb{V}$ for $k < i$. This means we have reduction context $R_0 = R[\delta(e_{a,0}, \ldots, e_{a,i-1}, \bullet, e_{a,i+1}, \ldots, e_{a,n}, e_0^\sigma, e_b)]$, and by (**cr.v**),

$$\begin{aligned}&\Gamma; R[\delta(e_{a,0}, \ldots, e_{a,i-1}, e_{a,i}, e_{a,i+1}, \ldots, e_{a,n}, e_j^\sigma, e_b)] \mapsto\\ &\Gamma'; R[\delta(e_{a,0}, \ldots, e_{a,i-1}, e'_{a,i}, e_{a,i+1}, \ldots, e_{a,n}, e_j^\sigma, e_b)], \quad j < 2,\end{aligned}$$

so by induction hypothesis the conclusion is direct. $\Box_{\mathbf{Op-ciu}}$

Lemma (Lambda ciu): $e_0 \cong^{ciu} e_1 \Rightarrow \lambda x.e_0 \cong^{ciu} \lambda x.e_1$.

Proof (lambda ciu): Given arbitrary Γ, σ, R such that $\Gamma[R[(\lambda x.e_i)^\sigma]] \in \mathbb{E}_\emptyset$, show

$$\downarrow\Gamma[R[\lambda x.e_0^\sigma]] \Rightarrow \downarrow\Gamma[R[\lambda x.e_1^\sigma]];$$

the converse is symmetric. Note we may assume $x \notin \mathrm{Dom}(\sigma)$, $x \notin \mathrm{FV}(\mathrm{Rng}(\sigma))$, and bring the substitutions inside the λ's. Generalize this statement to

$$\downarrow(\Gamma; e)^{\{z:=\lambda x.e_0^\sigma\}} \Rightarrow \downarrow(\Gamma; e)^{\{z:=\lambda x.e_1^\sigma\}},$$

where $z \notin \mathrm{Dom}(\Gamma)$, $z \notin \mathrm{FV}(\mathrm{Rng}(\sigma))$. The original goal follows by letting $e = R[z]$. Proceed by induction on the length of the computation of the assumption. Consider whether $\Gamma; e$ is uniform in z, i.e. whether

$$\Gamma; e \mapsto \Gamma; e'$$

for some e'. If it is uniform, then by (**cr.iii**),

$$(\Gamma; e)^{\{z:=\lambda x.e_i^\sigma\}} \mapsto (\Gamma'; e')^{\{z:=\lambda x.e_i^\sigma\}}, \quad i < 2,$$

and the result follows directly by induction hypothesis.

Consider then the case where the description $\Gamma; e$ is stuck, i.e. does not reduce. Since $\downarrow(\Gamma; e)^{\{z:=\lambda x.e_0^\sigma\}}$, the description does not get stuck when a λ-value is substituted for z. By inspection of the rules, replacing z with a λ-value causes a stuck computation to become un-stuck in only one case, the case where $e = R[\mathsf{app}(z, v)]$ for some v. Consider this case. By inspection of the (**app**) rule, we have the following uniformity property:

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

for all expressions e'. Therefore, by (**cr.iii**) and picking e' to be e_0^σ and e_1^σ,

$$(\Gamma; R[\mathsf{app}(\lambda x.e_i^\sigma, v)])^{\{z:=\lambda x.e_i^\sigma\}} \mapsto (\Gamma; R[e_i^{\sigma\{x:=v\}}])^{\{z:=\lambda x.e_i^\sigma\}}, \quad i < 2.$$

It thus suffices to show

$$\downarrow(\Gamma; R[e_1^{\sigma\{x:=v\}}])^{\{z:=\lambda x.e_1^\sigma\}}.$$

By the induction hypothesis,

$$\downarrow(\Gamma; R[e_0^{\sigma\{x:=v\}}])^{\{z:=\lambda x.e_1^\sigma\}}.$$

Then by assumption $e_0 \cong^{ciu} e_1$, e_0 above can be replaced by e_1 (take the substitution in the definition of $\cong^{ciu}$ to be $\sigma\{x := v\}$), giving what was desired. $\square_{\mathbf{lambda-ciu}}$

We may now prove (**ciu**), $e_0 \cong e_1 \Leftrightarrow e_0 \cong^{ciu} e_1$.

Proof (ciu):

$(\Rightarrow)$ Pick C such that $C[e_i] \overset{*}{\mapsto} \Gamma; R[e_i^\sigma]$; one such C is

$$C = \Gamma[\mathsf{app}(\ldots\mathsf{app}(\lambda x_1, \ldots, \lambda x_n.R[\bullet], v_1)\ldots, v_n)],$$

where $\mathrm{Dom}(\sigma) = \{x_1, \ldots x_n\}$ and $\sigma(x_i) = v_i$ for $i < n$.

$(\Leftarrow)$ Show $(\forall C)(C[e_0] \cong^{ciu} C[e_1])$, the congruence of $\cong^{ciu}$. From this the result is direct by picking $\Gamma = \emptyset$, $R = \bullet$, $\mathrm{Dom}(\sigma) = \emptyset$. Proceed by induction on the size of C. For the base case, $C = \bullet$ or $C = v$ for $v \in \mathbb{A}$; for the former the result follows by assumption, and for the latter by reflexivity. Inductively assume true for contexts smaller than C. We break C into two cases, $C = \lambda x.C_0$, and $C = \delta(C_0, \ldots, C_n)$ for $\delta \in \{\mathsf{app}\} \cup \mathbb{F}$.

$(C = \lambda x.C_0)$ By induction hypothesis $C_0[e_0] \cong^{ciu} C_0[e_1]$, so $\lambda x.(C_0[e_0]) \cong^{ciu} \lambda x.(C_0[e_1])$ by (**Lambda ciu**). And since hole filling captures, $(\lambda x.C_0)[e_0] \cong^{ciu} (\lambda x.C_0)[e_1]$.

$(C = \delta(C_0, \ldots, C_n))$ By induction hypothesis, $C_i[e_0] \cong^{ciu} C_i[e_1]$, for $i \leq n$. Then, by (**Op ciu**),

$$\begin{aligned}
&\delta(C_0[e_0], C_1[e_0] \ldots C_n[e_0]) \\
&\quad \cong^{ciu} \delta(C_0[e_1], C_1[e_0] \ldots C_n[e_0]) \\
&\quad \cong^{ciu} \delta(C_0[e_1], C_1[e_1] \ldots C_n[e_0]) \\
&\quad \vdots \qquad \vdots \\
&\quad \cong^{ciu} \delta(C_0[e_1], C_1[e_1] \ldots C_n[e_1]).
\end{aligned}$$

$\square_{\mathbf{ciu}}$

Part II: The First-Order Theory of Individuals

3. The Syntax and Semantics of the First-Order Theory

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 $\Gamma \models \Phi[\sigma]$.

3.1. Syntax of Formulas

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 reaches 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 ($\mathbb{U}$):

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

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

Definition ($\mathbb{W}$):

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

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 `mk`:

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

3.2. Evaluation of Contextual Assertions

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

Definition ($\Gamma;\, U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma';\, U'[\![\sigma']\!]$)**:** Let $U \in \mathbb{U}$ be such that $\mathrm{Traps}(U) = \{x_1, \ldots x_n\}$, $\Gamma \in \mathbb{M}$, $\sigma \in \mathbb{S}$, $(\mathrm{Dom}(\Gamma) \cup \mathrm{Dom}(\sigma)) \cap \mathrm{Traps}(U) = \emptyset$ and let z be a fresh variable. We write

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

to mean,

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

and $\mathrm{Dom}(\sigma') = \mathrm{Dom}(\sigma) \cup (\mathrm{Traps}(U) - \mathrm{Traps}(U'))$. Note that σ and σ' will agree on the domain of σ. To avoid problems with shadowing of trapped variables, such as $\mathtt{let}\{x := e_0\}\mathtt{let}\{x := e_1\}\bullet$, before carrying out the reduction $\Gamma;\, (U[\mathsf{app}(\ldots\mathsf{app}(z, x_1), \ldots, x_n)])^{\sigma}$ we assume the shaddowed trapped variables of U are renamed to something fresh, and the part of the accumulated substitution corresponding to these variables is carried out, leaving only original traps in the domain of σ'. This is just one way of doing the bookkeeping for the symbolic evaluation. This kind of hygiene can also be taken care of automatically by introducing schematic variables with associated substitutions (c.f. (Talcott, 1991)). This symbolic or parametric reduction of contexts is explored in more detail in (Agha et al., 1993).

The following two lemmas are key in developing methods for reasoning about contextual assertions, and illustrate the basic ideas in reasoning about computation with contextual assertions.

Lemma (u-red): Let Γ, σ, U be such that $\mathrm{Dom}(\sigma) \cap \mathrm{Traps}(U) = \emptyset$, and $\Gamma;\, U[\mathtt{nil}]^{\sigma}$ is closed. If $\Gamma;\, U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_0;\, U_0[\![\sigma_0]\!]$ (with $\mathrm{Dom}(\sigma_0) = \mathrm{Dom}(\sigma) \cup (\mathrm{Traps}(U) - \mathrm{Traps}(U_0))$) and if $\mathrm{Traps}(U_1) \cap (\mathrm{Dom}(\sigma) \cup \mathrm{Traps}(U)) = \emptyset$, then $\Gamma;\, (U[U_1])[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_0;\, (U_0[U_1])[\![\sigma_0]\!]$

Proof : By induction on the length of the computation. We consider cases according to the construction of U.

If $U = \bullet$, then we are done, since the reduction is length 0.

If $U = \mathtt{let}\{x := e\}U'$, then $\Gamma; e^{\sigma} \overset{*}{\mapsto} \Gamma'; v$ and $\Gamma;\, U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma';\, U'[\![\sigma']\!] \overset{*}{\mapsto} \Gamma_0;\, U_0[\![\sigma_0]\!]$ where $\sigma' = \sigma\{x := v\}$. Also, $\Gamma;\, (U[U_1])[\![\sigma]\!] \overset{*}{\mapsto} \Gamma';\, U'[U_1][\![\sigma']\!]$, and by induction hypothesis $\Gamma';\, U'[U_1][\![\sigma']\!] \overset{*}{\mapsto} \Gamma_0;\, (U_0[U_1])[\![\sigma_0]\!]$

The other cases are similar.

$\Box_{\mathbf{u-red}}$

Lemma (u-red-cmps): Let $U = U_0[U_1]$, with $\mathrm{Traps}(U_0) \cap \mathrm{Traps}(U_1) = \emptyset$, and let σ be such that $\mathrm{Dom}(\sigma) \cap \mathrm{Traps}(U) = \emptyset$. Then $\Gamma;\, U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_1;\, R[\![\sigma_1]\!]$ iff there are $R_0, R_1, \Gamma_0, \sigma_0$ such that $R = R_0[R_1]$ and

(0) $\Gamma;\, U_0[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_0;\, R_0[\![\sigma_0]\!]$

(1) $\Gamma_0;\, U_1[\![\sigma_0]\!] \overset{*}{\mapsto} \Gamma_1;\, R_1[\![\sigma_1]\!]$

Proof : The backward implication follows from (**cr**). For the forward implication, assume $\Gamma;\, (U_0[U_1])[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_1;\, R[\![\sigma_1]\!]$. We find the required $R_0, R_1, \Gamma_0, \sigma_0$ by induction on the length of the computation, considering cases on the structure of U_0.

If $U_0 = \bullet$, then take $R_0 = \bullet$, $\Gamma_0 = \Gamma$, $\sigma_0 = \sigma$, and $R_1 = R$.

If $U_0 = \mathtt{let}\{x := e\}\, U_0'$, then $U = \mathtt{let}\{x := e\}\, U'$ where $U' = U_0'[U_1]$. Also, $\Gamma;\, e^{\sigma} \overset{*}{\mapsto} \Gamma';\, v$ for some $\Gamma';\, v$, and letting $\sigma' = \sigma\{x := v\}$:

(a) $\Gamma;\, U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma';\, U'[\![\sigma']\!] \overset{*}{\mapsto} \Gamma_1;\, R[\![\sigma_1]\!]$; and

(b) $\Gamma;\, U_0[\![\sigma]\!] \overset{*}{\mapsto} \Gamma';\, U_0'[\![\sigma']\!]$.

By the induction hypothesis we can find $R_0, R_1, \Gamma_0, \sigma_0$ such that $R = R_0[R_1]$

(0′) $\Gamma';\, U_0'[\![\sigma']\!] \overset{*}{\mapsto} \Gamma_0;\, R_0[\![\sigma_0]\!]$

(1′) $\Gamma_0;\, U_1[\![\sigma_0]\!] \overset{*}{\mapsto} \Gamma_1;\, R_1[\![\sigma_1]\!]$

By (b) we are done.

The remaining cases are similar.

$\Box_{\mathrm{u-red-cmps}}$

3.3. Semantics of Formulas

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 $\mathrm{FV}(\Phi^{\sigma}) \cup \mathrm{FV}(e_j^{\sigma}) \subseteq \mathrm{Dom}(\Gamma)$ for $j < 2$ we define satisfaction:

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

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

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

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

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

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

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

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

$\square_{\textbf{valid}}$

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.

Although evaluation is a relation for contextual assertions, as it is for ordinary expressions, the following lemmas shows that the difference in the results can essentially be ignored.

Lemma (unique-ev): Let $U \in \mathbb{U}$ be such that $\mathrm{Traps}(U) = \{x_1, \ldots x_n\}$, $\Gamma \in \mathbb{M}$, $\sigma \in \mathbb{S}$, $\mathrm{Dom}(\sigma) \cap \mathrm{Traps}(U) = \emptyset$. If $\Gamma;\, U\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma_0; R_0\llbracket\sigma_0\rrbracket$ and $\Gamma;\, U\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma_1; R_1\llbracket\sigma_1\rrbracket$, then $\Gamma_0 \models \Phi[\sigma_0] \Leftrightarrow \Gamma_1 \models \Phi[\sigma_1]$ for any formula closed by $\Gamma_j; \sigma_j$.

Proof : This is because the only difference between $\Gamma_0; R_0\llbracket\sigma_0\rrbracket$ and $\Gamma_1; R_1\llbracket\sigma_1\rrbracket$ is in the choice names for cells in $\mathrm{Dom}(\Gamma_j) - \mathrm{Dom}(\Gamma)$. In particular, $\mathrm{Dom}(\sigma_j) = \mathrm{Dom}(\sigma) \cup \{x_1, \ldots x_n\}$. $\square$

Contextual assertions do interact nicely with evaluation.

Lemma (eval):

$$\text{If} \quad \Gamma_0;\, U_0\llbracket\sigma_0\rrbracket \overset{*}{\mapsto} \Gamma_1;\, U_1\llbracket\sigma_1\rrbracket, \quad \text{then} \quad \Gamma_0 \models U_0\llbracket\Phi\rrbracket[\sigma_0] \quad \text{iff} \quad \Gamma_1 \models U_1\llbracket\Phi\rrbracket[\sigma_1]$$

Proof (eval): If $\Gamma_0;\, U_0\llbracket\sigma_0\rrbracket \overset{*}{\mapsto} \Gamma_1;\, U_1\llbracket\sigma_1\rrbracket$, then $\Gamma_0;\, U_0\llbracket\sigma_0\rrbracket \overset{*}{\mapsto} \Gamma_0';\, R\llbracket\sigma_0'\rrbracket$ iff $\Gamma_1;\, U_1\llbracket\sigma_1\rrbracket \overset{*}{\mapsto} \Gamma_1';\, R'\llbracket\sigma_1'\rrbracket$ where $\Gamma_0';\, R\llbracket\sigma_0'\rrbracket$ and $\Gamma_1';\, R'\llbracket\sigma_1'\rrbracket$ differ only in choice of names for newly allocated cells. An the equivalence follows from (**unique-ev**). $\square_{\textbf{eval}}$

3.4. Modalities

The contextual assertions allow various modalities to be defined. Two examples are $\square\Phi$ (read true in all expansions of the current memory) and $\boxdot\Phi$ (read true in all reachable memories). The expected meanings of $\square$ and $\boxdot$ are given by the semantic satisfaction clauses

$$\text{(nec)} \quad \Gamma \models \square\Phi[\sigma] \Leftrightarrow (\forall \Gamma' \supseteq \Gamma)(\Gamma' \models \Phi[\sigma])$$

$$\text{(snec)} \quad \Gamma \models \boxdot\Phi[\sigma] \Leftrightarrow (\forall \Gamma' \mid \mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma'))(\Gamma' \models \Phi[\sigma])$$

Taxonomically the $\square$ modality lies in the S4.2 realm, although it possesses some non-standard properties. For example

$$\models \Phi \Rightarrow \square\Phi \qquad \text{for } \Phi \text{ an atomic or negated atomic formula}$$

To define $\square$ we introduce three procedures. `len` computes the length of a list. `alloc` is a procedure which takes a number n, allocates n new cells and returns them in a list. `assign` is a procedure that takes a list of cells, and a list of values of the same length, and stores the values in corresponding cells. We assume `+1`/`-1` are unary arithmetic operations that add/subtract one from a number.

Definition (`len`, `alloc`, `assign`)**:**

$$\mathtt{len} = \mathtt{Y}(\lambda l.\lambda x.\mathtt{if}(\mathtt{ispr}(x), \mathtt{+1}(l(\mathtt{snd}(x))), 0)$$

$$\mathtt{alloc} = \mathtt{Y}(\lambda a.\lambda x.\mathtt{if}(\mathtt{eq}(x, 0), \mathtt{nil}, \mathtt{pr}(\mathtt{mk}(\mathtt{nil}), a(\mathtt{-1}(x)))))$$

$$\mathtt{assign} = \mathtt{Y}(\lambda a.\lambda x, y.\mathtt{if}(\mathtt{eq}(x, \mathtt{nil}), \mathtt{nil}, \mathtt{seq}(\mathtt{set}(\mathtt{fst}(x), \mathtt{fst}(y)), a(\mathtt{snd}(x), \mathtt{snd}(y)))))$$

Using these procedures we can create U-contexts that yield arbitrary extensions of memory.

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

$$(\forall x)(\mathtt{isnat}(x) \cong \mathtt{t} \ \Rightarrow\ \mathtt{let}\{z := \mathtt{alloc}(x)\} \\ \llbracket(\forall y)(\mathtt{len}(y) \cong x \ \Rightarrow\ \mathtt{seq}(\mathtt{assign}(z, y), \llbracket\Phi\rrbracket))\rrbracket)$$

assuming x, y do not occur free in Φ.

Lemma (nec): The semantic and syntactic definitions of $\Box$ are equivalent:

$$\Gamma \models \Box\Phi[\sigma] \ \Leftrightarrow\ (\forall\Gamma' \supseteq \Gamma)(\Gamma' \models \Phi[\sigma])$$

Proof (nec): For the forward implication assume $\Gamma \models \Box\Phi[\sigma]$ (according to the abbreviation) and $\Gamma \subseteq \Gamma'$. Let $\Gamma' = \Gamma\{z_i := v_i \mid 1 \le i \le n\}$ where $[z_1, \ldots, z_n]$ are not in the domain of Γ, and let $v = \mathtt{pr}(v_1, \ldots, \mathtt{pr}(v_n, \mathtt{nil})\ldots)$. Let $\sigma_0 = \sigma\{x := n\}$, and let $\sigma_2 = \sigma_0\{y := v\}$. Then there are Γ_1, v_1, σ_1 such that $\Gamma; \mathtt{let}\{z := \mathtt{alloc}(x)\}\llbracket\sigma_0\rrbracket \overset{*}{\mapsto} \Gamma_1; \llbracket\sigma_1\rrbracket$ and $\Gamma_1; \mathtt{seq}(\mathtt{assign}(z, y), \llbracket\sigma_1\rrbracket \overset{*}{\mapsto} \Gamma'; \llbracket\sigma_2\rrbracket$. By hypothesis $\Gamma' \models \Phi[\sigma_2]$, and by the free variable assumption $\Gamma' \models \Phi[\sigma]$.

For the backward implication assume $(\forall\Gamma' \supseteq \Gamma)(\Gamma' \models \Phi[\sigma])$. Let $\sigma_0 = \sigma\{x := n\}$ and assume that $n \in \mathbb{N}$. We must show that

$$\Gamma \models \mathtt{let}\{z := \mathtt{alloc}(x)\}\llbracket\forall y.\mathtt{len}(y) \cong x \ \Rightarrow\ \mathtt{seq}(\mathtt{assign}(z, y), \llbracket\Phi\rrbracket)\rrbracket[\sigma_0].$$

Let $[z_1, \ldots, z_n]$ be fresh, $\sigma_1 = \sigma_0\{z := \mathtt{pr}(z_1, \ldots, \mathtt{pr}(z_n, \mathtt{nil})\ldots)\}$, and let $\Gamma_1 = \Gamma\{z_i := \mathtt{mk}(\mathtt{nil}) \mid 1 \le i \le n\}$, then we must show that

$$\Gamma_1 \models (\forall y)(\mathtt{len}(y) \cong x \ \Rightarrow\ \mathtt{seq}(\mathtt{assign}(z, y), \llbracket\Phi\rrbracket))[\sigma_1].$$

Let $\sigma_2 = \sigma_1\{y := v\}$ and assume that $\Gamma_1 \models \mathtt{len}(y) \cong x[\sigma_2]$. Then we must show that $\Gamma_1 \models \mathtt{seq}(\mathtt{assign}(z, y), \llbracket\Phi\rrbracket)[\sigma_2]$. By construction, $\Gamma_1; \mathtt{seq}(\mathtt{assign}(z, y), \llbracket\sigma_2\rrbracket) \overset{*}{\mapsto} \Gamma'; \bullet\llbracket\sigma_2\rrbracket$, and since $\Gamma' \supseteq \Gamma$ we are done. $\Box_{\mathrm{nec}}$

$\boxdot$ essentially expresses validity, and is easily defined using U-contexts as follows.

Definition ($\boxdot$**):** $\boxdot\Phi$ abbreviates:

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

Lemma (snec): The semantic and syntactic definitions of $\boxdot$ are equivalent:

$$\Gamma \models \boxdot\Phi[\sigma] \;\Leftrightarrow\; (\forall\Gamma' \;\big|\; \mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma'))(\Gamma' \models \Phi[\sigma])$$

Proof (snec): For the forward implication assume $\Gamma \models \boxdot\Phi[\sigma]$ and $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$. Let e be such that $\Gamma; e^{\sigma} \overset{*}{\mapsto} \Gamma'; \mathtt{nil}$. For example

$$e = \mathtt{let}\{z := \mathtt{mk}(\mathtt{nil})\}_{z \in \mathrm{Dom}(\Gamma') - \mathrm{Dom}(\Gamma)} \mathtt{seq}(\mathtt{set}(z, \Gamma'(z))_{z \in \mathrm{Dom}(\Gamma')}, \mathtt{nil}).$$

Then, taking $f = \lambda d.e$ and $x = \mathtt{nil}$ where d is fresh, we have $\Gamma' \models \Phi[\sigma]$ by definition of $\boxdot$.

For the backward implication assume

$$(\forall\Gamma' \;\big|\; \mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma'))(\Gamma' \models \Phi[\sigma])$$

and let $\sigma' = \sigma\{f := v_f, x := v_x\}$ for arbitrary v_f, v_x with free variables in $\mathrm{Dom}(\Gamma)$. If $\Gamma; \mathtt{seq}(\mathtt{app}(f, x), \bullet)[\![\sigma']\!] \overset{*}{\mapsto} \Gamma'; \bullet[\![\sigma']\!]$ then $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$ and by the backward assumption, and the requirement that f, x are not free in Φ, $\Gamma' \models \Phi[\sigma']$. Hence $\Gamma \models \boxdot\Phi[\sigma]$.

$\square_{\text{snec}}$

To define the satisfaction clause for $\forall$, a decision must be made as to the domain of quantification. We chose the set of values existing in the memory. Another possible choice would have been the set of values existing in all possible extensions of the memory. This defines a strong quantifier $\forall_{\mathrm{s}}$ with the following satisfaction clause:

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

Note that strong quantification neatly decomposes into $\square\forall$. We will examine it in more detail shortly (§3.10). It was first studied in (Mason, 1986a).

3.5. Caveats

3.5.1. The Invisible Set Problem

We would like the following property to hold:

(mem.eq) If $z \in \mathrm{Dom}(\Gamma)$, $v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)}$, $\Gamma \models (\mathtt{get}(z) \cong v)[\emptyset]$, and $\Gamma' = \Gamma\{z := \mathtt{mk}(v)\}$, then

$$\Gamma \models \Phi[\sigma] \quad \text{iff} \quad \Gamma' \models \Phi[\sigma].$$

This property fails as can be seen by the following counterexample.

$$v_z = \lambda d.\lambda x.x \qquad v = \lambda d.\mathtt{let}\{y := \mathtt{mk}(\lambda x.x)\}\lambda x.\mathtt{app}(\mathtt{get}(y), x)$$

$$\Gamma = \mathtt{let}\{z := v_z\}\,\bullet \qquad \Gamma' = \mathtt{let}\{z := v\}\bullet$$

$$U = \mathtt{let}\{x := \mathtt{app}(\mathtt{get}(z), \mathtt{nil})\}\,\bullet \qquad \Phi = U[\![x \cong \lambda x.x]\!]$$

Then $\Gamma \models \Phi[\emptyset]$, but $\Gamma' \not\models \Phi[\emptyset]$. To see that Γ' fails to satisfy Φ note that $\Gamma'; U[\![\emptyset]\!] \overset{*}{\mapsto} \Gamma_1; \bullet[\![\sigma_1]\!]$ where $\sigma_1 = \{x := \lambda x.\mathtt{app}(\mathtt{get}(y), x)\}$, and $\Gamma_1 = \Gamma'\{y := \mathtt{mk}(\lambda x.x)\}$. Taking $R = \mathtt{let}\{x := \bullet\}\mathtt{seq}(\mathtt{set}(y, \lambda x.\mathtt{app}(\mathtt{nil}, \mathtt{nil})), \mathtt{app}(x, \mathtt{nil}))$ it is easy to see that $\Gamma_1 \not\models (x \cong \lambda x.x)[\sigma_1]$ The problem is that the definition of satisfaction for contextual assertions allows the private store created by the application of v_1 to be accessed by an observing reduction context. The use of the contextual assertion is critical to the construction of the counterexample. As the following lemma shows, the property (**mem.eq**) does hold when Φ contains no contextual assertions.

Theorem (mem.eq.no-ca): If $z \in \mathrm{Dom}(\Gamma)$, $v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)}$, $\Gamma \models (z \cong v)[\emptyset]$, and $\Gamma' = \Gamma\{z := \mathtt{mk}(v)\}$, and Φ contains no contextual assertions, then

$$\Gamma \models \Phi[\sigma] \quad \text{iff} \quad \Gamma' \models \Phi[\sigma].$$

Proof : Assume $z \in \mathrm{Dom}(\Gamma)$, $v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)}$, $\Gamma \models (z \cong v)[\emptyset]$, and $\Gamma' = \Gamma\{z := \mathtt{mk}(v)\}$ then we have the following

$(\dagger)\quad \Gamma; e \updownarrow \Gamma'; e \quad \text{for} \quad e \in \mathbb{E}_{\mathrm{Dom}(\Gamma)}$.

To see this note that

$$\begin{aligned} \Gamma; e &\updownarrow \Gamma; \mathtt{seq}(\mathtt{set}(z, z), e) && \text{by computation} \\ &\updownarrow \Gamma; \mathtt{seq}(\mathtt{set}(z, v), e) && \text{by hypothesis} \\ &\updownarrow \Gamma'; e && \text{by computation} \end{aligned}$$

Assume further that Φ contains no contextual assertions. We argue by induction on the structure of Φ. There are three cases.

Case: Suppose that Φ is $e_0 \cong e_1$. Let $R \in \mathbb{R}_{\mathrm{Dom}(\Gamma)}$. Then by the above observation, $\Gamma; R[e_j^\sigma] \updownarrow \Gamma'; R[e_j^\sigma]$. □

In particular $\Gamma' \models z \cong v$, hence by symmetry we need only consider the forward implication. Thus we assume

$(*)\quad \Gamma \models \Phi[\sigma]$

and consider the remaining two cases for the construction of Φ.

Case: Suppose Φ is $\Phi_0 \Rightarrow \Phi_1$. Then

$$\begin{aligned} \Gamma \models \Phi_0[\sigma] &\Rightarrow \Gamma \models \Phi_0[\sigma] && \text{IH} \\ &\Rightarrow \Gamma \models \Phi_1[\sigma] && (*) \\ &\Rightarrow \Gamma' \models \Phi_1[\sigma] && \text{IH} \end{aligned}$$

□

Case: Suppose that Φ is $(\forall x)\Phi_0$. Let $v_0 \in \mathbb{V}_{\mathrm{Dom}(\Gamma)}$. Then

$$\begin{aligned} &\Gamma \models \Phi_0[\sigma\{x := v_0\}] && (*) \\ &\Gamma' \models \Phi_0[\sigma\{x := v_0\}] && \text{IH} \end{aligned}$$

□

3.5.2. Visible Garbage

The weak quantifier allows us to express details concerning the current state of memory. For example that there are no cells in memory, and that there is exactly one cell in memory:

$$\Phi_{0\text{-cell}} \Leftrightarrow (\forall x)(\mathtt{cell}(x) \cong \mathtt{nil})$$

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

The ability to express such facts about the state of memory provide counterexamples to the validity of the following schema

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

because the formula, Φ, can detect the production of garbage.

3.5.3. Extensionality

We now investigate what form of extensionality is true. The naïve version is false:

Lemma (non-ext):

$$\neg(\models \Box(\forall x)(\mathsf{app}(\rho_0, x) \cong \mathsf{app}(\rho_1, x))) \Rightarrow \lambda x.\mathsf{app}(\rho_0, x) \cong \lambda x.\mathsf{app}(\rho_1, x))$$

Proof : A counterexample is:

$$\rho_0 :=: \lambda z.\mathtt{if}(\mathtt{get}(y), 0, 1)$$

$$\rho_1 :=: \lambda z.0$$

$$\Gamma :=: \mathtt{let}\{y := \mathtt{mk}(\mathtt{t})\}\bullet$$

$$\sigma :=: \{y := y\}$$

Then clearly we have that $\Gamma \models \Box(\forall x)(\mathsf{app}(\rho_0, x) \cong \mathsf{app}(\rho_1, x))[\sigma]$ but not $\Gamma \models \rho_0 \cong \rho_1[\sigma]$.□

We can modify the naïve version slightly to obtain a valid principle:

Lemma (ext): Suppose that $\rho_i \in \mathbb{L}$ and $\mathrm{FV}(\rho_i) \subseteq \bar{z}$ for $i < 2$. Then

$$\models \Box(\forall \bar{z})(\forall x)(\mathsf{app}(\rho_0, x) \cong \mathsf{app}(\rho_1, x)) \Rightarrow \lambda x.\mathsf{app}(\rho_0, x) \cong \lambda x.\mathsf{app}(\rho_1, x))$$

3.6. Contextual Assertion Principles

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

(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

$$\Phi \Rightarrow U\llbracket\Phi\rrbracket$$

holds, a simple counterexample being

$$\mathtt{get}(x) \cong 2 \Rightarrow \mathtt{let}\{x := \mathtt{mk}(3)\}\llbracket\mathtt{get}(x) \cong 2\rrbracket$$

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

$$\mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{let}\{y := \mathtt{mk}(0)\}[x] \cong \mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{let}\{y := \mathtt{mk}(0)\}[y]$$

but

$$\mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{let}\{y := \mathtt{mk}(0)\}\llbracket\neg(x \cong y)\rrbracket$$

Note that (**ca.iii**) is false (or nonsensical) for general contexts; a simple counterexample is when $C_0 = \mathtt{app}(\bullet, v)$, $C_1 = \lambda x.\bullet$ and $\Phi = \mathtt{t} \cong \mathtt{nil}$.

Proof (i): Assume $\models \Phi$. We want to show that $\Gamma \models U\llbracket\Phi\rrbracket[\sigma]$ for any closing $\Gamma;\sigma$. For this purpose, assume $\Gamma;\, U\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma';\, R\llbracket\sigma'\rrbracket$. Then $\Gamma' \models \Phi[\sigma']$ by the assumption that $\models \Phi$, and we are done by definition of $\models$ for contextual assertions. $\square_{\mathbf{i}}$

Proof (ii): We must show that $\Gamma \models U\llbracket e_0 \cong e_1\rrbracket \Rightarrow U[e_0] \cong U[e_1][\sigma]$ for any closing $\Gamma;\sigma$. Assume (†): $\Gamma \models U\llbracket e_0 \cong e_1\rrbracket[\sigma]$. We must show that $\Gamma; R[U\llbracket e_0\rrbracket^{\sigma}] \updownarrow \Gamma; R[U\llbracket e_1\rrbracket^{\sigma}]$ for any R with free variables in $\mathrm{Dom}(\Gamma)$. If $\Gamma; R[U\llbracket e_0\rrbracket^{\sigma}]\downarrow$, then we can find $\Gamma'; R'$ and σ' such that $\Gamma;\, U\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma';\, R'\llbracket\sigma'\rrbracket$ and $\Gamma;\, R[U\llbracket e_j\rrbracket^{\sigma}] \updownarrow \Gamma';\, R[R'[e_j]^{\sigma'}]$ for $j < 2$. By (†) we have $\Gamma' \models e_0 \cong e_1[\sigma']$ hence $\Gamma'; R[R'[e_0]^{\sigma'}] \updownarrow \Gamma'; R[R'[e_1]^{\sigma'}]$ and (by the semantics of $\Rightarrow$) we are done. $\square_{\mathbf{ii}}$

Proof (iii): For the forward implication, assume $\Gamma \models U_0\llbracket U_1\llbracket\Phi\rrbracket\rrbracket[\sigma]$ and show $\Gamma \models (U_0[U_1])\llbracket\Phi\rrbracket[\sigma]$. For this purpose, assume $\Gamma; (U_0[U_1])\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma_1; R\llbracket\sigma_1\rrbracket$. By (**u-red-cmps**) there are $R_0, R_1, \sigma_0, \Gamma_0$ such that $R = R_0[R_1]$, $\Gamma;\, U_0\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma_0; R_0\sigma_0$, and $\Gamma_0;\, U_1\llbracket\sigma_0\rrbracket \overset{*}{\mapsto} \Gamma_1; R_1\sigma_1$. Thus, by the forward assumption, $\Gamma_0 \models U_1\llbracket\Phi\rrbracket[\sigma_0]$ and $\Gamma_1 \models \Phi[\sigma_1]$.

For the backward implication, assume $\Gamma \models (U_0[U_1])\llbracket\Phi\rrbracket[\sigma]$ and show $\Gamma \models U_0\llbracket U_1\llbracket\Phi\rrbracket\rrbracket[\sigma]$. For this purpose, assume $\Gamma;\, U_0\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma_0; R_0\llbracket\sigma_0\rrbracket$ and show $\Gamma_0 \models U_1\llbracket\Phi\rrbracket[\sigma_0]$. For this purpose, assume $\Gamma_0;\, U_1\llbracket\sigma_0\rrbracket \overset{*}{\mapsto} \Gamma_1;\, R_1\sigma_1$. By (**u-red-cmps**) $\Gamma; (U_0[U_1])\llbracket\sigma\rrbracket \overset{*}{\mapsto} \Gamma_1; (R_0[R_1])\llbracket\sigma_1\rrbracket$ and by the backward assumption $\Gamma_1 \models \Phi[\sigma_1]$. $\square_{\mathbf{iii}}$

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) $U\llbracket\mathtt{False}\rrbracket \Rightarrow U\llbracket\Phi\rrbracket$

(not) $U\llbracket\neg\Phi\rrbracket \Leftrightarrow (U\llbracket\mathtt{False}\rrbracket \vee \neg U\llbracket\Phi\rrbracket)$

(imp) $U\llbracket\Phi_0 \Rightarrow \Phi_1\rrbracket \Leftrightarrow (U\llbracket\Phi_0\rrbracket \Rightarrow U\llbracket\Phi_1\rrbracket)$

Proof (triv): Assume $\Gamma; U[\![\mathtt{False}]\!][\sigma]$. If $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma']\!]$, then $\Gamma' \models \mathtt{False}[\sigma]$ which is a contradiction. Hence $U[\![\Phi]\!]$ is vacuously true. $\Box_{\mathbf{triv}}$

Proof (not): For the forward implication assume $\Gamma \models U[\![\neg\Phi]\!][\sigma]$. If $\Gamma \models U[\![\mathtt{False}]\!][\sigma]$, then we are done by (**triv**). So, assume $\Gamma \models \neg(U[\![\mathtt{False}]\!])[\sigma]$. Thus $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma']\!]$ for some $\Gamma'; R$ and σ'. By the forward assumption, $\Gamma' \models \neg\Phi[\sigma']$. Hence $\Gamma \models \neg U[\![\Phi]\!][\sigma]$.

For the backward implication, assume $\Gamma \models (U[\![\mathtt{False}]\!] \vee \neg U[\![\Phi]\!])[\sigma]$. If $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma']\!]$ then $\Gamma' \models \neg\Phi[\sigma']$. Hence $\Gamma \models U[\![\neg\Phi]\!][\sigma]$.

$\Box_{\mathbf{not}}$

Proof (imp): For the forward implication assume $\Gamma \models U[\![\Phi_0 \Rightarrow \Phi_1]\!][\sigma]$, and $\Gamma \models U[\![\Phi_0]\!][\sigma]$. If $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma']\!]$ then by the forward assumption, $\Gamma' \models \Phi_0 \Rightarrow \Phi_1[\sigma']$ and $\Gamma' \models \Phi_0[\sigma']$. Hence $\Gamma' \models \Phi_1[\sigma']$ and $\Gamma \models \Phi_1[\sigma]$.

For the backward implication, assume $\Gamma \models (U[\![\Phi_0]\!] \Rightarrow U[\![\Phi_1]\!])[\sigma]$ and $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma']\!]$. If $\Gamma' \models \Phi_0[\sigma']$ then by the backward assumption $\Gamma' \models \Phi_1[\sigma']$. Hence $\Gamma' \models (\Phi_0 \Rightarrow \Phi_1)[\sigma']$ and $\Gamma \models U[\![\Phi_0 \Rightarrow \Phi_1]\!][\sigma]$. $\Box_{\mathbf{imp}}$

Corollary (con.prop): As for $\Rightarrow$, contextual assertions commute with $\vee$, $\wedge$, and $\Leftrightarrow$.

The corollary follows directly from the theorem (**con.prop**), if we take $\vee$, $\wedge$, and $\Leftrightarrow$ to be abbreviations.

The case of the quantifier is a little less simple.

Lemma (con.∀):

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

The converse to ($\forall$) is easily shown to be false by considering U to be $\mathtt{let}\{y := \mathtt{mk}(\mathtt{t})\}\bullet$ and Φ to be $\neg(x \cong y)$.

Proof (∀): Assume $\Gamma \models U[\![\forall x\Phi]\!][\sigma]$ and let $v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)}$. We must show that $\Gamma \models U[\![\Phi]\!][\sigma']$ where $\sigma' = \sigma\{x := v\}$. For this, assume $\Gamma; U[\![\sigma']\!] \overset{*}{\mapsto} \Gamma_1; R[\![\sigma_1']\!]$. Since x is not free in U, $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_1; R[\![\sigma_1]\!]$ where $\sigma_1' = \sigma_1\{x := v\}$. By hypothesis, $\Gamma_1 \models (\forall x\Phi)[\sigma_1]$ and since $\mathbb{V}_{\mathrm{Dom}(\Gamma)} \subseteq \mathbb{V}_{\mathrm{Dom}(\Gamma_1)}$ $\Gamma_1 \models \Phi[\sigma_1']$. $\Box_{\forall}$

The rule (**ca.i**) can be replaced by an implication using the modality $\boxdot$. This, together with (**con.prop.imp**) gives us a principle for replacing a formula by an equivalent one inside a contextual assertion.

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

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

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

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

Proof (i): Assume $\Gamma \models \boxdot((\forall x_1 \ldots x_n)\Phi)[\sigma]$, and $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma \cup \sigma']\!]$ where $\mathrm{Dom}(\Gamma') \supseteq \mathrm{Dom}(\Gamma)$, $\mathrm{Dom}(\sigma') = \mathrm{Traps}(U)$ and $\mathrm{Rng}(\sigma') \subseteq \mathbb{V}_{\mathrm{Dom}(\Gamma')}$. Thus by the initial assumption $\Gamma' \models \Phi[\sigma \cup \sigma']$. $\Box_{\mathbf{i}}$

Proof (ii): Assume $\Gamma \models \boxdot((\forall x_1 \ldots x_n)(\Phi_0 \Rightarrow \Phi_1))[\sigma]$ and $\Gamma \models \boxdot(U[\![\Phi_0]\!])[\sigma]$. Let $\mathrm{Dom}(\Gamma'') \supseteq \mathrm{Dom}(\Gamma)$ and $\Gamma''; U[\sigma] \overset{*}{\mapsto} \Gamma'; R[\![\sigma \cup \sigma']\!]$ as in the proof of (**i**). We must show $\Gamma' \models \Phi_1[\sigma \cup \sigma']$. By the second assumption $\Gamma' \models \Phi_0[\sigma \cup \sigma']$. and by the first assumption $\Gamma' \models (\Phi_0 \Rightarrow \Phi_1)[\sigma \cup \sigma']$. $\Box_{\mathbf{ii}}$

Proof (iii): similar to (ii) $\Box_{\mathbf{iii}}$

Some simple, but by no means exhaustive facts concerning divergence are:

Lemma ($\Uparrow$):

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

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

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

Proof (i): Assume $\Gamma \models \Uparrow e_0[\sigma]$. For the forward implication assume $\Gamma \models \Uparrow e_1[\sigma]$. Then $\neg(\downarrow \Gamma; R[e_j^\sigma])$ for $j < 2$ and hence $\Gamma \models (e_0 \cong e_1)[\sigma]$. For the backward implication assume $\Gamma \models (e_0 \cong e_1)[\sigma]$. Thus $\Gamma; e_0^\sigma \updownarrow \Gamma; e_1^\sigma$, and by the initial assumption $\Gamma \models \Uparrow e_1[\sigma]$. $\Box_{\mathbf{i}}$

Proof (ii): Assume e closed, and $\Gamma \models \Uparrow e[\emptyset]$. Then $\neg(\downarrow \Gamma; R[e])$ for any R (by (**cr**)). hence $\neg(\downarrow \Gamma; U[e])$. $\Box_{\mathbf{ii}}$

Proof (iii): Let $e = \mathtt{Y}(\lambda y.\lambda x.\mathtt{app}(y, x))(\mathtt{nil})$. Then $e \overset{*}{\mapsto} e$ non-trivially, hence $\neg(\downarrow \Gamma; e)$ for any Γ. $\Box_{\mathbf{iii}}$

3.7. Axioms for Proofs

We established in §2 the validity of the let-rules of the lambda-c calculus (Moggi, 1989) as valid equations concerning operational equivalence. Thus they are also valid formulas of the logic:

Lemma ($\mathtt{let}_c$ axioms):

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

(ii) $R[e] \cong \mathtt{let}\{x := e\}R[x]$ $\quad x$ not free in R

(iii) $R[\mathtt{let}\{x := e_0\}e_1] \cong \mathtt{let}\{x := e_0\}R[e_1]$ $\quad x$ not free in R

Lemma (implies axioms):

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

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

(**implies.i, implies.ii**) are important laws for manipulating contexts under assumptions. (**implies.i**) follows from (**sca**) and usual logical reasoning, (**implies.ii**) is direct from the definitions.

Here are some basic principles for the atomic portion of the language used in the examples in Section 6.

Lemma (atomic axioms):

(i) $x \cong y \wedge \texttt{atom}(x) \cong \texttt{t} \Leftrightarrow \texttt{eq}(x, y) \cong \texttt{t}$

(ii) $(\neg x \cong y) \wedge \texttt{atom}(x) \cong \texttt{t} \Leftrightarrow \texttt{eq}(x, y) \cong \texttt{nil}$

(iii) $\texttt{if}(\texttt{nil}, e_0, e_1) \cong e_1$

(iv) $\neg(y \cong \texttt{nil}) \Rightarrow \texttt{if}(y, e_0, e_1) \cong e_0$

(v) $(\texttt{atom}(x_0) \cong \texttt{t} \wedge \texttt{atom}(x) \cong \texttt{nil}) \Leftrightarrow \neg(x_0 \cong x_1)$

3.8. Memory Operation Principles

This logic extends and improves the complete first order system presented in (Mason and Talcott, 1989; Mason and Talcott, 1992a). 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`, `set` and `get` are as follows:

3.8.1. `mk` axioms

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. In a world with control effects evaluation of e_0 must be free of them for this principle to be valid (Felleisen, 1993).

Lemma (mk axioms):

(i) $\texttt{let}\{x := \texttt{mk}(v)\}[\![\neg(x \cong y) \wedge \texttt{cell}(x) \cong \texttt{t} \wedge \texttt{get}(x) \cong v]\!]$ $\quad x$ fresh

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

(iii) $\Downarrow \texttt{mk}(z)$

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

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

3.8.2. `set` axioms

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.

Lemma (set axioms):

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

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

(iii) $\mathtt{cell}(z) \Rightarrow \Downarrow \mathtt{set}(z,x)$

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

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

(vi) $\mathtt{let}\{z := \mathtt{mk}(x)\}\mathtt{seq}(\mathtt{set}(z,w),e) \cong \mathtt{let}\{z := \mathtt{mk}(w)\}e \qquad z$ not free in w

3.8.3. get axioms

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. The fact that calls to get do not alter memory can be generalized to the following lemma. It states that expressible observations are not affected by the execution of get.

Lemma (get axioms):

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

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

(iii) $\mathtt{cell}(x) \Leftrightarrow (\exists y)(\mathtt{get}(x) \cong y)$

Lemma (non-effects (get)):

$$\Phi \Rightarrow \mathtt{let}\{x := \mathtt{get}(y)\}[\![\Phi]\!] \qquad x \notin \mathrm{FV}(\Phi)$$

3.9. Soundness of Memory Principles

We demonstrate the soundness of the above memory principles (**mk**) and (**set**).

Proof (soundness): The soundness of these principles provides some simple examples of operational reasoning, we do a representative selection.

(mk.i) The soundness of (**mk.i**) is easily established, choose Γ and σ (with $x \notin \mathrm{Dom}(\Gamma)$). Now

$$\Gamma; \mathtt{let}\{x := \mathtt{mk}(v^\sigma)\}[\![\sigma]\!] \mapsto \Gamma\{x := \mathtt{mk}(v^\sigma)\}; [\![\sigma]\!]$$

Thus it suffices to show that

$$\Gamma\{x := \mathtt{mk}(v^\sigma)\} \models \Psi[\sigma]$$

$$\Psi = (\neg(x \cong y) \wedge \mathtt{cell}(x) \cong \mathtt{t} \wedge \mathtt{get}(x) \cong v)$$

So choose any $R \in \mathbb{R}_{\mathrm{Dom}(\Gamma)}$, let $\Gamma' = \Gamma\{x := \mathtt{mk}(v^\sigma)\}$, and observe that by (**cr**)

$$\Gamma'[R[\mathtt{cell}(x)]] \overset{*}{\mapsto} \Gamma'; R[\mathtt{cell}(x)] \mapsto \Gamma'; R[\mathtt{t}]$$

and

$$\Gamma'[R[\mathtt{t}]] \overset{*}{\mapsto} \Gamma'; R[\mathtt{t}].$$

Thus $\Gamma' \models (\mathtt{cell}(x) \cong \mathtt{t})[\sigma]$. Similarly $\Gamma' \models (\mathtt{get}(x) \cong v)[\sigma]$. Now let $R = \mathtt{eq}(\bullet, y)$. Then

$$\Gamma'[R[x]] \overset{*}{\mapsto} \Gamma'; R[x] \overset{*}{\mapsto} \Gamma'; \mathtt{nil}$$

and

$$\Gamma'[R[y]] \overset{*}{\mapsto} \Gamma'; R[y] \overset{*}{\mapsto} \Gamma'; \mathtt{t}$$

Thus $\Gamma' \models (x \cong y \;\Rightarrow\; \mathtt{t} \cong \mathtt{nil})[\sigma]$. Since it is not the case that $\Gamma' \models \mathtt{False}$ (**mk.iii**) is also valid $\square_{\mathbf{mk.i}}$

(**mk.iv**) We use (**ciu**), let e_{lhs} and e_{rhs} be the obvious expressions and choose Γ, R, and σ that close the expressions e_{lhs} and e_{rhs}. Without loss of generality we may assume that $x \notin \mathrm{Dom}(\Gamma)$. We consider two cases depending on whether or not $\Gamma; e_0^{\sigma}$ diverges. Suppose that $\Gamma; e_0^{\sigma} \overset{*}{\mapsto} \Gamma'; v_0$, and $\sigma' = \sigma\{y := v_0\}$. Note that $v^{\sigma} = v^{\sigma'}$ and so by (**cr**)

$$\begin{aligned} &\Gamma; R[e_{\mathrm{lhs}}^{\sigma}] \overset{*}{\mapsto} \Gamma'; R[(\mathtt{let}\{x := \mathtt{mk}(v)\}e_1)^{\sigma'}] \\ &\quad \mapsto \Gamma'\{x := v^{\sigma}\}; R[e_1^{\sigma'}] \\ &\Gamma; R[e_{\mathrm{rhs}}^{\sigma}] \overset{*}{\mapsto} \Gamma\{x := v^{\sigma}\}; R[(\mathtt{let}\{y := e_0\}e_1)^{\sigma}] \\ &\quad \mapsto \Gamma'\{x := v^{\sigma}\}; R[e_1^{\sigma'}] \end{aligned}$$

and so are equidefined. The case when $\Gamma; e_0^{\sigma}$ diverges is similar. $\square_{\mathbf{mk.iv}}$

($\mathtt{set}$.**vi**) To verify ($\mathtt{set}$.**vi**) let e_{lhs} and e_{rhs} be the obvious expressions and choose Γ, R, and σ that close them, and assume $z \notin (\mathrm{Dom}(\Gamma) \cup \mathrm{FV}(R))$. Then by (**cr.ii**) and (**cr.iii**)

$$\begin{aligned} &\Gamma; R[e_{\mathrm{lhs}}^{\sigma}] \mapsto \Gamma\{z := \mathtt{mk}(x^{\sigma})\}; R[(\mathtt{seq}(\mathtt{set}(z, w), e))^{\sigma}] \\ &\quad \overset{*}{\mapsto} \Gamma\{z := \mathtt{mk}(w^{\sigma})\}; R[e^{\sigma}] \\ &\Gamma; R[e_{\mathrm{rhs}}^{\sigma}] \mapsto \Gamma\{z := \mathtt{mk}(w^{\sigma})\}; R[e^{\sigma}] \end{aligned}$$

Hence the two expressions are equidefined. $\square_{\mathbf{set.vi}}$ $\square_{\mathbf{soundness}}$

3.10. The Strong Quantifier Fragment

In this subsection we examine the properties of the logic obtained by choosing the strong version of the $\forall$ quantifier. Recall that the satisfaction clause for $\forall_s$ was given by:

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

Also recall that it $\forall_s$ neatly decomposes into $\square\forall$. We define the strong quantifier fragment as follows:

Definition ($\mathbb{W}_s$)**:**

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

One reason given for preferring weak over strong $\forall$ was the desire for *classical* behavior. I.e. properties such as $\forall x.\Phi \;\Leftrightarrow\; \Phi$ if $x \notin \mathrm{FV}(\Phi)$. It turns out that this holds for $\forall_s$ in the following sense, writting $\Diamond(\Phi)$ for $\neg\square\neg(\Phi)$ as usual in modal logic.

Theorem (strong):

(n) $\models (\Phi \Rightarrow \Box(\Phi))$

(p) $\models (\Diamond(\Phi) \Rightarrow \Phi)$

Proof (strong): By induction on the complexity of Φ.

Case: Φ is $e_0 \cong e_1$ then we are done by previous results for atomic and negated atomic formulae.

Case: Φ is $\Phi_0 \Rightarrow \Phi_1$.

For (**n**) assume that $\Gamma \models \Phi[\sigma]$, $\Gamma' \supseteq \Gamma$, and $\Gamma' \models \Phi_0[\sigma]$. We show $\Gamma' \models \Phi_1[\sigma]$. Since $\Gamma' \models \Phi_0[\sigma]$ and $\Gamma' \supseteq \Gamma$ we infer that $\Gamma \models \Diamond\Phi_0[\sigma]$. Thus by IH(**p**) we conclude that $\Gamma \models \Phi_0[\sigma]$. Thus by hypothesis $\Gamma \models \Phi_1[\sigma]$, and by IH(**n**) $\Gamma' \models \Phi_1[\sigma]$.$\Box_{\mathbf{n}}$

For (**p**) assume $\Gamma' \supseteq \Gamma$, $\Gamma' \models \Phi[\sigma]$ and $\Gamma \models \Phi_0[\sigma]$. Then by this latter assumption and IH(**n**) we infer $\Gamma' \models \Phi_0[\sigma]$. Thus $\Gamma' \models \Phi_1[\sigma]$, and by IH(**p**) $\Gamma \models \Phi_1[\sigma]$.$\Box_{\mathbf{p}}$

Case: Φ is $U[\![\Phi_0]\!]$. Assume $\Gamma' \supseteq \Gamma$ with $\Gamma' = \Gamma \cup \Gamma_1$.

For (**n**) assume $\Gamma \models \Phi[\sigma]$ and $\Gamma'; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_0'; R[\sigma_0]$. Then by (**cr**) $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_0; R[\sigma_0]$ with $\Gamma_0' = \Gamma_0 \cup \Gamma_1$. Thus $\Gamma_0 \models \Phi_0[\sigma_0]$ and by IH(**n**) $\Gamma_0' \models \Phi_0[\sigma_0]$.$\Box_{\mathbf{n}}$

For (**p**), assume $\Gamma' \models \Phi[\sigma]$ and $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_0; R[\sigma_0]$ (without loss of generality $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_0) = \mathrm{Dom}(\Gamma)$). Then $\Gamma'; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma_0 \cup \Gamma_1; R[\sigma_0]$, and $\Gamma_0 \cup \Gamma_1 \models \Phi_0[\sigma_0]$. By IH(**p**) $\Gamma_0 \models \Phi_0[\sigma_0]$.$\Box_{\mathbf{p}}$

Case: Φ is $(\forall_s x)\Phi_0$.

For (**n**) we use the definition of $\forall_s$ and the fact that $\Box\forall_s x\Phi$ is equivalent to $\forall_s x\Phi$.$\Box_{\mathbf{n}}$

For (**p**), assume $\Gamma' \supseteq \Gamma$, $\Gamma' \models \Phi[\sigma]$. To show $\Gamma \models \Phi[\sigma]$, let $\Gamma_0 \supseteq \Gamma$ and let $v \in \mathbb{V}_{\mathrm{Dom}(\Gamma_0)}$. Without loss of generality we may assume $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma_0) = \mathrm{Dom}(\Gamma)$ Thus we can form $\Gamma_1 = \Gamma_0 \cup \Gamma' \supseteq \Gamma'$ and by IH(**n**) $\Gamma_1 \models \Phi_0[\sigma\{x := v\}]$. Hence by IH(**p**) $\Gamma_0 \models \Phi_0[\sigma\{x := v\}]$.$\Box_{\mathbf{p}}$

$\Box$

In the strong fragment `get` and `mk` both enjoy the property that expressible observations are not effected by their execution. We have already stated the property for `get`, here we state and prove it in the case of `mk`. It is false in the weak quantifier version.

Lemma (non-effects (`mk`)):

$$\Phi_s \Rightarrow \mathtt{let}\{x := \mathtt{mk}(y)\}[\![\Phi_s]\!] \qquad \Phi_s \in \mathbb{W}_s,\ x \notin \mathrm{FV}(\Phi_s)$$

Proof (mk): Let Φ_s be built from atomic equivalences via implication, contextual assertions, and $\forall_s$. We proceed by induction on the construction of Φ_s.

If Φ_s is $e_0 \cong e_1$ then we are done by the property that execution is not effected by unreachable memory, (**cr**).

If Φ_s is $\Phi_0 \Rightarrow \Phi_1$ then we are done by the induction hypothesis and (**con.prop.imp**).

If Φ is $(\forall_s z)\Phi_0$ then $\Gamma \models \Phi_s[\sigma]$ iff $\Gamma' \models \Phi_0[\sigma\{z := v\}]$ for all $\Gamma' \supseteq \Gamma$ and $v \in \mathbb{V}_{\mathrm{Dom}(\Gamma')}$, and $\Gamma \models \mathtt{let}\{x := \mathtt{mk}(y)\}\Phi_s[\sigma]$ iff $\Gamma^* \models \Phi_0[\sigma\{z := v\}]$ for all $\Gamma^* \supseteq \Gamma\{x := \sigma(y)\}$ and all

$v \in \mathbb{V}_{\mathrm{Dom}(\Gamma^*)}$. The forward implication is easy, since any Γ^* extending Γ' can also serve as an Γ' extending Γ. For the reverse implication, choose Γ' extending Γ. Without loss of generality we may assume $x \notin \mathrm{Dom}(\Gamma')$ then by induction hypothesis we are done.

If Φ_s is $U[\![\Phi_0]\!]$ then $\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; R[\![\sigma']\!]$, with $x \notin \mathrm{Dom}(\Gamma')$, just if $\Gamma; \mathtt{let}\{x := \mathtt{mk}(y)\}\, U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma^*; R[\![\sigma']\!]$, with $\Gamma^* = \Gamma'\{x := \sigma(y)\}$. By the induction hypothesis we are done. $\square_{\mathbf{mk}}$

4. The Meyer-Sieber Examples

In this section we present the Meyer-Sieber examples in the original Algol-like version, give the representation in our language, and prove the desired equivalences, interpreting equivalence as operational equivalence.

In the Algol-like notation P, Q refer to parameterless procedures declared outside the scope of the block in which they appear, and $\bot$ is some procedure that always diverges. In our language calls to externally declared procedures, P, are represented as applications of free (function) variables, p, to a dummy argument, say `nil`. Declarations of a local variable x in a block, **begin new** $x; \ldots$ **end**, is represented by $\mathtt{let}\{x := \mathtt{mk}(\mathtt{t})\}e$ where e represents the block body. The *contents* operation of (Meyer and Sieber, 1988) is $\mathtt{get}(x)$, the assignment operation, $x := v$, is $\mathtt{set}(x, v)$, and equality on program variables is `eq`.

4.1. Example 1

The following block is equivalent to P.

begin new x; P **end**

The intuitive justification for this is simple. The static scope of local variables in Algol entails that x must be inaccessible to the procedure P. Consequently allocating and then deallocating x if and when P returns has no discernable effect on the call to P. In our language the example and the equivalence translates to:

Definition ($\mathtt{ex}_1$):

$$\mathtt{ex}_1 = \mathtt{let}\{x := \mathtt{mk}(\mathtt{t})\}\mathtt{app}(p, \mathtt{nil})$$

Theorem (1): $\mathtt{ex}_1 \cong \mathtt{app}(p, \mathtt{nil})$

This equivalence is an instance of a more general garbage collection principle isolated in the system (Mason and Talcott, 1992a). This principle allows for the elimination of garbage — cells no longer accessible from the program text.

Garbage collection rule (g.c).

$$\models \Gamma_0[e] \cong e \qquad \text{provided } \mathrm{Dom}(\Gamma_0) \cap \mathrm{FV}(e) = \emptyset$$

Proof (g.c): The soundness of (**g.c**) is justified using (**ciu**) and (**cr**). Choose any Γ, σ, and R which close the expressions $\Gamma_0[e]$ and e. Without loss of generality we may assume that $\mathrm{Dom}(\Gamma_0) \cap \mathrm{FV}(R) = \emptyset = \mathrm{Dom}(\Gamma_0) \cap \mathrm{Dom}(\Gamma)$. So by (**cr.ii**) (with $R = \bullet$)

$$\Gamma[R[e^\sigma]] \overset{*}{\mapsto} \Gamma; R[e^\sigma] \quad \text{and} \quad \Gamma[R[\Gamma_0[e]^\sigma]] \overset{*}{\mapsto} \Gamma; R[\Gamma_0[e]^\sigma]$$

But also by (**cr.iii**) and (**cr.ii**) we have that $R[\Gamma_0[e]^\sigma] \stackrel{*}{\mapsto} \Gamma_0^\sigma; R[e^\sigma]$, so by (**cr.iv**) we have that

$$\Gamma[R[\Gamma_0[e]^\sigma]] \stackrel{*}{\mapsto} (\Gamma \cup \Gamma_0^\sigma); R[e^\sigma]$$

and the result now follows easily by (**cr.iv**) $\Box_{\mathrm{g.c}}$

Proof (1): by (**g.c**) $\Box_1$

4.2. Example 2

The following block is equivalent to $\bot$.

$$\textbf{begin new}\, x;\ x := 0;\ P;\ \textbf{if}\ \mathit{contents}(x) = 0\, \textbf{then}\, \bot\, \textbf{fi}\ \textbf{end}$$

Since P has no control effects, it either diverges or returns. If it does return, then the static scoping implies that the contents x will be unchanged. Thus x will contain 0 and the block as a whole will diverge. The translation of the programs and equivalences are:

Definition ($\mathtt{ex}_2$)**:**

$$\mathtt{ex}_2 = \mathtt{let}\{x := \mathtt{mk}(\mathtt{t})\}\mathtt{seq}(\mathtt{set}(x, 0), \mathtt{app}(p, \mathtt{nil}), \mathtt{if}(\mathtt{eq}(\mathtt{get}(x), 0), \mathtt{ex}_\bot, \mathtt{nil}))$$

$$\mathtt{ex}_\bot = \mathtt{app}(\mathtt{Y}(\lambda y.\lambda x.\mathtt{app}(y, x)), \mathtt{nil})$$

where `Y` is a suitable fixed-point combinator.

Theorem (2): $\mathtt{ex}_2 \cong \mathtt{ex}_\bot$

Proof (2): Two principles concerning `mk` and `set` are relevant for this example: (**mk.iv**) and (**set.vi**). We can apply these principles to obtain:

$$\begin{aligned}
\mathtt{ex}_2 \ &\cong \mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{seq}(\mathtt{app}(p, \mathtt{nil}), \mathtt{if}(\mathtt{eq}(\mathtt{get}(x), 0), \mathtt{ex}_\bot, \mathtt{nil})) \\
&\qquad \text{by } (\mathtt{set}\textbf{.vi}) \\
&\cong \mathtt{seq}(\mathtt{app}(p, \mathtt{nil}), \mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{if}(\mathtt{eq}(\mathtt{get}(x), 0), \mathtt{ex}_\bot, \mathtt{nil})) \\
&\qquad \text{by } (\mathtt{mk}\textbf{.iv}) \text{ and the definition of } \mathtt{seq} \\
&\cong \mathtt{seq}(\mathtt{app}(p, \mathtt{nil}), \mathtt{if}(\mathtt{eq}(\mathtt{get}(\mathtt{mk}(0)), 0), \mathtt{ex}_\bot, \mathtt{nil})) \\
&\qquad \text{by } (\textbf{ca.i}), (\textbf{ca.ii}) \text{ and } (\mathtt{let}_c\textbf{.ii}) \\
&\cong \mathtt{seq}(\mathtt{app}(p, \mathtt{nil}), \mathtt{if}(\mathtt{eq}(0, 0), \mathtt{ex}_\bot, \mathtt{nil})) \\
&\qquad \text{by } (\textbf{ca}) \text{ and properties of } \mathtt{get}\ ((\mathtt{mk}\textbf{.i})\ \&\ (\textbf{ca.ii})) \\
&\cong \mathtt{seq}(\mathtt{app}(p, \mathtt{nil}), \mathtt{ex}_\bot) \\
&\qquad \text{by } (\textbf{ca}) \text{ and properties of } \mathtt{if} \\
&\cong \mathtt{ex}_\bot \\
&\qquad \text{by } (\Uparrow\textbf{.i}), (\Uparrow\textbf{.ii}), \text{ and } (\Uparrow\textbf{.iii})
\end{aligned}$$

$\Box_2$

4.3. Example 3

The third example has a similar theme to example 2. Here two blocks are equivalent because the order of allocation is irrelevant, as is the order of assignment, provided that the assignment is to distinct locations. In our system this is justified using the set permutation principle, (**set.iv**), and principle (**mk.i**) together with the contextual assertion principles, (**ca**).

The following two blocks are equivalent.

$$\textbf{begin new}\, x;\ \textbf{new}\, y;\ x := 0;\ y := 0;\ Q(x, y)\, \textbf{end}$$

$$\textbf{begin new}\, x;\ \textbf{new}\, y;\ x := 0;\ y := 0;\ Q(y, x)\, \textbf{end}$$

This example and equivalence translates to:

Definition ($\mathtt{ex}_3$):

$$\mathtt{ex}_{3.1} = \mathtt{let}\{x := \mathtt{mk(t)}\}\mathtt{let}\{y := \mathtt{mk(t)}\}\mathtt{seq}(\mathtt{set}(x, 0), \mathtt{set}(y, 0), \mathsf{app}(\mathsf{app}(q, x), y))$$

$$\mathtt{ex}_{3.2} = \mathtt{let}\{x := \mathtt{mk(t)}\}\mathtt{let}\{y := \mathtt{mk(t)}\}\mathtt{seq}(\mathtt{set}(x, 0), \mathtt{set}(y, 0), \mathsf{app}(\mathsf{app}(q, y), x))$$

Theorem (3): $\mathtt{ex}_{3.1} \cong \mathtt{ex}_{3.2}$

This proof is yet again a simple derivation using the set permutation principle, (**set.iv**), and principle (**mk.i**) together with the contextual assertion principles, (**ca**).

Proof (3):

$$\mathtt{ex}_{3.1} = \mathtt{let}\{y := \mathtt{mk(t)}\}\mathtt{let}\{x := \mathtt{mk(t)}\}\mathtt{seq}(\mathtt{set}(y, 0), \mathtt{set}(x, 0), \mathsf{app}(\mathsf{app}(q, y), x))$$

by α

$$\cong \mathtt{let}\{x := \mathtt{mk(t)}\}\mathtt{let}\{y := \mathtt{mk(t)}\}\mathtt{seq}(\mathtt{set}(y, 0), \mathtt{set}(x, 0), \mathsf{app}(\mathsf{app}(q, y), x))$$

by (**mk.iv**)

$$\cong \mathtt{ex}_{3.2}$$

by (**set.iv**), (**ca.i**) and (**mk.i**)

$\square_3$

4.4. Example 4

The fourth example is again of a similar flavor. In this example a cell is allocated and assigned prior to the execution of a procedure declaration and a procedure call. It is inaccessible to both these pieces of program text, and so remains unchanged by them. This is justified by using (**mk.iv**), (**set.vi**) and β_{value}-conversion and then proceeds as in the proof of (**2**).

The following block is equivalent to $\perp$.

```
begin
    new x; new y;
    procedure Twice;
        begin y := 2 * contents(y) end;
    x := 0; y := 0;
    Q(Twice);
    if contents(x) = 0 then ⊥ fi end
```

The translation and the equivalence are:

Definition ($\mathtt{ex}_4$)**:**

$$\begin{aligned}\mathtt{ex}_4 = \mathtt{let}\{x := \mathtt{mk(t)}\}&\mathtt{let}\{y := \mathtt{mk(t)}\}\\ &\mathtt{let}\{\mathit{Twice} := \lambda d.\mathtt{set}(y, *(2, \mathtt{get}(y)))\}\\ &\quad\mathtt{seq}(\mathtt{set}(x, 0), \mathtt{set}(y, 0), \mathtt{app}(q, \mathit{Twice}), \mathtt{if}(\mathtt{eq}(\mathtt{get}(x), 0), \mathtt{ex}_\perp, \mathtt{nil}))\end{aligned}$$

Here and elsewhere d will be used to denote a dummy (fresh) variable.

Theorem (4): $\mathtt{ex}_4 \cong \mathtt{ex}_\perp$

This equivalence is established by precisely the same reasoning as the previous example, although the manipulations differ slightly.

Proof (4): By (**mk.iv**) and (**set.vi**) we have

$$\begin{aligned}\mathtt{ex}_4 \cong \mathtt{let}\{y := \mathtt{mk(t)}\}&\\ &\mathtt{let}\{\mathit{Twice} := \lambda d.\mathtt{set}(y, *(2, \mathtt{get}(y)))\}\\ &\quad\mathtt{seq}(\mathtt{set}(y, 0),\\ &\qquad\mathtt{app}(q, \mathit{Twice}),\\ &\qquad\mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{if}(\mathtt{eq}(\mathtt{get}(x), 0), \mathtt{ex}_\perp, \mathtt{nil}))\end{aligned}$$

Now by β_{value}-conversion and the same reasoning as in the proof of (**2**) this simplifies to

$$\mathtt{ex}_4 \cong \mathtt{let}\{y := \mathtt{mk}(0)\}\mathtt{seq}(\mathtt{app}(q, \lambda d.\mathtt{set}(y, *(2, \mathtt{get}(y)))), \mathtt{ex}_\perp)$$

So by ($\Uparrow$) we may conclude that $\mathtt{ex}_4 \cong \mathtt{ex}_\perp$ $\Box_4$

4.5. Example 5

The fifth example exhibits some novel aspects. Here a cell is allocated and used by the declared procedure Add_2 as local store. The procedure is then passed to a previously declared procedure, which as a result has no access to the cell x (other than indirectly via procedure calls to Add_2). Consequently the contents on the cell x can only be modified by

Add_2, and hence must satisfy any invariant that such calls preserve. The following block is equivalent to $\bot$.

```
begin
   new $x$;
   procedure $Add_2$;
         begin $x := contents(x) + 2$ end;
   $x := 0$;
   $Q(Add_2)$;
   if $contents(x) \bmod 2 = 0$ then $\bot$ fi end
```

Definition $(\mathtt{ex}_5)$**:**

$$\begin{aligned}
\mathtt{ex}_5 = {} & \mathtt{let}\{x := \mathtt{mk}(\mathtt{t})\} \\
& \qquad\quad \mathtt{let}\{\rho_5 := \lambda d.\mathtt{set}(x, +(\mathtt{get}(x), 2))\} \\
& \qquad\qquad \mathtt{seq}(\mathtt{set}(x, 0), \mathtt{app}(q, \rho_5), \mathtt{if}(\mathtt{eq}(\mathtt{mod}(\mathtt{get}(x), 2), 0), \mathtt{ex}_\bot, \mathtt{nil}))
\end{aligned}$$

Theorem (5): $\mathtt{ex}_5 \cong \mathtt{ex}_\bot$

Let ρ_5 be $\lambda d.\mathtt{set}(x, +(\mathtt{get}(x), 2))$. The key to the proof of (**5**) is the invariance lemma: $\mathtt{let}\{x := \mathtt{mk}(u)\}\rho_5$ preserves the invariant Φ_5 where Φ_5 is the formula $\mathtt{eq}(\mathtt{mod}(u, 2), 0) \cong \mathtt{t}$.

Lemma (5.1):

$$\Phi_5 \;\Rightarrow\; \mathtt{let}\{x := \mathtt{mk}(u)\}\mathtt{seq}(\mathtt{app}(q, \rho_5), \mathtt{let}\{u := \mathtt{get}(x)\}[\![\Phi_5]\!])$$

Lambda abstractions with local store, such as ρ_5, are *objects* in the sense of (Mason and Talcott, 1991b). In that paper several principles for establishing operational equivalence were presented and used. The invariance theorem (**inv**) is typical of these principles. It provides a general method for establishing invariant properties. The statement of the theorem relies on an important notion, that of a function (with local store) depending only on the *first-order* part of its argument. We say that two values have the same first-order part (written $x =_{\mathrm{fo}} y$) if they are both functions, both cells, both pairs with with the same first-order components, or are the same atom.

Definition $(x =_{\mathrm{fo}} y)$**:** $x =_{\mathrm{fo}} y$ abbreviates $\mathtt{foeq}(x, y) \cong \mathtt{t}$, where $\mathtt{foeq}$ is the function defined as follows.

$$\mathtt{isfun} = \lambda x.\mathtt{and}(\mathtt{not}(\mathtt{atom}(x)), \mathtt{not}(\mathtt{ispr}(x)), \mathtt{not}(\mathtt{cell}(x))))$$

$$\begin{aligned}
\mathtt{foeq} = \mathtt{Y}(\lambda f.\lambda x, y.\mathtt{if}(& \mathtt{isfun}(x), \\
& \mathtt{isfun}(y), \\
& \mathtt{if}(\mathtt{cell}(x), \\
& \quad \mathtt{cell}(y), \\
& \quad \mathtt{if}(\mathtt{ispr}(x), \\
& \qquad \mathtt{and}(\mathtt{ispr}(y), \mathtt{foeq}(\mathtt{fst}(x), \mathtt{fst}(y)), \mathtt{foeq}(\mathtt{snd}(x), \mathtt{snd}(y))), \\
& \qquad \mathtt{if}(\mathtt{atom}(x), \mathtt{and}(\mathtt{atom}(y), \mathtt{eq}(x, y)), \\
& \qquad\quad \mathtt{nil})))))
\end{aligned}$$

Functions with local store that depend only on the first-order part of their argument can not apply any functions accessible from their argument, hence if they always return atoms as values they cannot leak access to their local store.

Theorem (fo): Assume $(\forall x, x')(x =_{\mathrm{fo}} x' \Rightarrow \mathtt{let}\{z := \mathtt{mk}(u)\}\llbracket \mathtt{app}(\rho, x) \cong \mathtt{app}(\rho, x')\rrbracket$ is valid and $\Gamma\{z := \mathtt{mk}(u)\}; \mathtt{app}(\rho, v) \overset{*}{\mapsto} \Gamma'\{z := \mathtt{mk}(u')\}; v'$, for some $v \in \mathbb{V}_\Gamma$. Let $\Gamma; v$ as $(\widehat{\Gamma}; \widehat{v})^\sigma$ where $\widehat{\Gamma}; \widehat{v}$ have each lambda abstraction occurrence replaced by a fresh variable, and σ maps these variables to the corresponding lambda abstractions. Then we can find $\widehat{\Gamma'}; \widehat{v'}$ and a $\widehat{u'}$ such that $\Gamma'; v' = (\widehat{\Gamma'}; \widehat{v'})^\sigma$, $u' = \widehat{u'}^\sigma$, and $\widehat{\Gamma}\{z := \mathtt{mk}(u)\}; \mathtt{app}(\rho, \widehat{v}) \overset{*}{\mapsto} \widehat{\Gamma}'\{z := \mathtt{mk}(\widehat{u'})\}; \widehat{v'}$ uniformly in the domain of σ.

Proof (fo): If not then by inspection of the reduction rules, we see that

$$\widehat{\Gamma}\{z := \mathtt{mk}(u)\}; \mathtt{app}(\rho, \widehat{v}) \overset{*}{\mapsto} \widehat{\Gamma}'\{z := \mathtt{mk}(\widehat{u'})\}; \widehat{R}[\mathtt{app}(p, \widehat{x})]$$

for some variable p in $\mathrm{Dom}(\sigma)$. If p comes from $\widehat{v}$ then we can replace p by a diverging lambda abstraction $\rho_\bot$ and get a contradiction. If p comes from $\widehat{\Gamma}$ then $\widehat{v}$ must contain a cell through which p is accessed. Thus we can replace σ by $\sigma' = \sigma\{p := \rho_\bot\}$, and again get a contradiction. $\square_{\mathbf{fo}}$

If Φ is a property of atoms (i.e. all quantifiers are restricted to range over $\mathbb{A}$), then the following three conditions guarantee that Φ is preserved by the object $\Gamma; \rho$ (i.e. any use of $\Gamma; \rho$ preserves Φ viewed as a property of the contents of Γ): **(0)** Φ implies that its free variable is an atom; **(1)** $\Gamma; \rho$ depends only on the first-order part of its argument; and **(2)** any application of ρ in the context Γ is equivalent to updating the contents of Γ and returning some atom. This is made precise in the following theorem, where for simplicity we consider single cell local store.

Theorem (inv): If Φ is a property of atoms with at most u free, ρ has at most x free and

(0) $\quad \Phi \Rightarrow \mathtt{atom}(u)$

(1) $\quad (\forall y, y')(y =_{\mathrm{fo}} y' \Rightarrow \mathtt{let}\{x := \mathtt{mk}(u)\}\llbracket \mathtt{app}(\rho, y) \cong \mathtt{app}(\rho, y')\rrbracket)$

(2) $\quad \Phi \Rightarrow (\forall y)(\exists v, w)\mathtt{let}\{x := \mathtt{mk}(u)\}\llbracket \Psi_2 \wedge \Psi_3\rrbracket$

$$\Psi_2 = \mathtt{app}(\rho, y) \cong \mathtt{seq}(\mathtt{set}(x, v), w)$$

$$\Psi_3 = \mathtt{atom}(w) \wedge \Phi\{u := v\}$$

are valid, then

(3) $\quad \Phi \Rightarrow \mathtt{let}\{x := \mathtt{mk}(u)\}\mathtt{seq}(e^{\{p:=\rho\}}, \mathtt{let}\{u := \mathtt{get}(x)\}\llbracket\Phi\rrbracket)$

is valid for $e \in \mathbb{E}$ not containing x free.

Proof (inv): Assume $\Gamma \models \Phi[\sigma]$, $a = \sigma(u)$. We want to show that for $\widehat{\Gamma}; \widehat{e}$ such that $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\widehat{\Gamma})$ and frees $\widehat{\Gamma}[\widehat{e}] \subseteq \{p\}$ and $\widehat{\Gamma}\{z := \mathtt{mk}(a)\}; \widehat{e}$ with $p = \rho$ steps to $\Gamma'\{z := \mathtt{mk}(v')\}; w'$ then $\Gamma'\{z := \mathtt{mk}(v')\} \models \Phi[\{u := v'\}]$. This is by computation induction. The only interesting case is where $\widehat{e}$ has the form $\widehat{R}[\mathtt{app}(p, \widehat{y})]$. Using (**fo**) and (**1,2**) this steps uniformly to $\widehat{\Gamma}\{z := \mathtt{mk}(b)\}; \widehat{R}[w]$ for some b such that $\Gamma'\{z := \mathtt{mk}(b)\} \models \Phi \wedge \mathtt{atom}(w)[\{u := b\}]$. $\square_{\mathbf{inv}}$

Proof (5.1): By (**inv**) we need only establish (**0**,**1**,**2**) taking Φ to be Φ_5 and ρ to be ρ_5. (**0**) is trivial. (**1**) follows from the fact that ρ_5 makes no use of its argument. In particular by β_{value}-conversion $\mathtt{app}(\rho_5, y) \cong \mathtt{set}(x, +(\mathtt{get}(x), 2))$. To establish (**2**), assume Φ_5 and let y be arbitrary. Let $w = \mathtt{nil}$ and $v = u + 2$. Then clearly

$$\mathtt{let}\{x := \mathtt{mk}(u)\}[\![\Psi_4 \ \wedge \ \Psi_5]\!]$$

$$\Psi_4 = \mathtt{app}(\rho_5, y) \cong \mathtt{seq}(\mathtt{set}(x, v), w)$$

$$\Psi_5 = \mathtt{atom}(w) \ \wedge \ \Phi_5\{u := v\}$$

$\square_{5.1}$

Proof (5): We reason as follows. Let U be the context

$$\mathtt{let}\{x := \mathtt{mk}(u)\}\mathtt{seq}(\mathtt{app}(q, \rho_5), \mathtt{let}\{u := \mathtt{get}(x)\}\bullet).$$

Then

$$\Phi_5 \ \Rightarrow \ U[\![\Phi_5]\!] \qquad \text{by } (\mathbf{5.1})$$

and

$$U[\![\Phi_5 \ \Rightarrow \ \mathtt{if}(\mathtt{eq}(\mathtt{mod}(\mathtt{get}(x), 2), 0), \mathtt{ex}_\bot, \mathtt{nil}) \cong \mathtt{ex}_\bot]\!]$$

by `if` rules and (**ca.i**)

$$\Phi_5 \ \Rightarrow \ U[\![\mathtt{if}(\mathtt{eq}(\mathtt{mod}(\mathtt{get}(x), 2), 0), \mathtt{ex}_\bot, \mathtt{nil}) \cong \mathtt{ex}_\bot]\!]$$

by (**con.prop**)

$$\mathtt{ex}_5 \cong \mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{seq}(\mathtt{app}(q, \rho_5), \mathtt{ex}_\bot)$$

Taking $u = 0$ and using (**ca.ii**), now by ($\Uparrow$) we are done.

$\square_5$

4.6. Example 6

The following block is equivalent to $\bot$.

```
begin
  new x;
  procedure AlmostAdd_2(z);
      begin if z = x
            then x := 1
            else x := contents(x) + 2 end fi;
      end;
  x := 0;
  Q(AlmostAdd_2);
  if contents(x) mod 2 = 0 then ⊥ fi
end
```

Definition ($\mathtt{ex}_6$)**:**

$$\begin{aligned}\mathtt{ex}_6 = \mathtt{let}\{x := \mathtt{mk}(\mathtt{t})\}&\\ &\mathtt{let}\{\rho_6 := \lambda z.\mathtt{if}(\mathtt{eq}(x,z),\mathtt{set}(x,1),\mathtt{set}(x,+(\mathtt{get}(x),2)))\}\\ &\quad\mathtt{seq}(\mathtt{set}(x,0),\mathtt{app}(q,\rho_6),\mathtt{if}(\mathtt{eq}(\mathtt{mod}(\mathtt{get}(x),2),0),\mathtt{ex}_\perp,\mathtt{nil}))\end{aligned}$$

Theorem (6): $\quad \mathtt{ex}_6 \cong \mathtt{ex}_\perp$

Proof (6): The proof is the same as the proof of (**5**) once we establish the invariance lemma (**6.1**). $\square_6$

Let ρ_6 be $\lambda z.\mathtt{if}(\mathtt{eq}(x,z),\mathtt{set}(x,1),\mathtt{set}(x,+(\mathtt{get}(x),2)))$ then $\mathtt{let}\{x := \mathtt{mk}(u)\}\rho_6$ preserves the invariant $\Phi_6 = \Phi_5$.

Lemma (6.1):

$$\Phi_6 \;\Rightarrow\; \mathtt{let}\{x := \mathtt{mk}(u)\}\mathtt{seq}(\mathtt{app}(q,\rho_6),\mathtt{let}\{u := \mathtt{get}(x)\}[\![\Phi_6]\!])$$

Proof (6.1): By (**inv**) we need only establish (**0,1,2**) taking Φ to be Φ_6 and ρ to be ρ_6. (**0**) is trivial. To establish (**1**) and (**2**) note that

$$\neg(y \cong x) \;\Rightarrow\; \mathtt{app}(\rho_6, y) \cong \mathtt{set}(x,+(\mathtt{get}(x),2))$$

by properties of `if`

$$\mathtt{let}\{x := \mathtt{mk}(u)\}[\![\neg(y \cong x)]\!] \qquad \text{by } (\mathbf{mk.i})$$

Now (**1**) follows from (**ca.i**) and (**con.prop**), and (**2**) follows by the same reasoning as in (**5.1**). $\square_{6.1}$

4.7. Example 7

The following two blocks are equivalent.

```
begin new x;
  procedure Add_1;
      begin x := contents(x) + 1 end;
  Q(Add_1)
end
begin new x;
  procedure Add_2;
      begin x := contents(x) + 2 end;
  Q(Add_2)
end
```

where Q is a procedure declared outside the scope of this block.

Definition ($\mathtt{ex}_7$)**:**

$$\mathtt{ex}_7^n = \mathtt{let}\{x := \mathtt{mk}(0)\}\mathtt{let}\{\rho := \lambda d.\mathtt{set}(x, +(\mathtt{get}(x), n))\}\mathtt{seq}(\mathtt{app}(q, \rho), \mathtt{nil})$$

Theorem (7): $\mathtt{ex}_7^1 \cong \mathtt{ex}_7^2$

The proof of (**7**) is based on the fact that $\mathtt{ex}_7^n \cong \mathtt{seq}(\mathtt{app}(q, \lambda d.\mathtt{nil}), \mathtt{nil})$ for any $n \in \mathbb{N}$. This follows from the equivalence

(†) $\lambda d.\mathtt{nil} \cong \mathtt{let}\{x := \mathtt{mk}(0)\}\lambda d.\mathtt{set}(x, +(\mathtt{get}(x), n))$

for any number n using `let` rules. Equivalences such as †, are established by using (**abstract**). This rule allows one to establish equivalence of functions with local store by showing that the functions depend only on the *first-order* part of their arguments (**1**) and finding an invariant property of atoms (**0**) relating the contents of the local stores and such that applications (calls) of the functions preserve the invariant, and return the same atomic value, (**2**). This is made precise in the following theorem, where for simplicity we consider single cell local store.

Theorem (abstract): If Φ is a property of atoms with at most u_0, u_1 free, ρ_j has at most x free for $j < 2$, and

(0) $\Phi \Rightarrow \mathtt{atom}(u_j)$

(1) $\bigwedge_{j<2} (\forall y, y')(y =_{\mathrm{fo}} y' \Rightarrow \mathtt{let}\{x := \mathtt{mk}(u_j)\}[\![\mathtt{app}(\rho_j, y) \cong \mathtt{app}(\rho_j, y')]\!])$

(2) $\Phi \Rightarrow (\forall y)(\exists v_j, w)(\Phi\{u_j := v_j\}_{j<2} \wedge$
$\qquad\mathtt{atom}(w) \wedge$
$\qquad\bigwedge_{j<2} \mathtt{let}\{x := \mathtt{mk}(u_j)\}[\![\mathtt{app}(\rho_j, y) \cong \mathtt{seq}(\mathtt{set}(x, v_j), w)]\!])$

are valid, then

(3)

$$\Phi \Rightarrow \mathtt{let}\{x := \mathtt{mk}(u_0)\}\rho_0 \cong \mathtt{let}\{x := \mathtt{mk}(u_1)\}\rho_1$$

is valid.

Proof (abstract): Assume $\Gamma \models \Phi[\sigma]$ then we want to show (assuming (**0,1,2**)) that

$$\Gamma; R[\mathtt{let}\{x := \mathtt{mk}(u_0)\}\rho_0]\downarrow \quad \text{iff} \quad \Gamma; R[\mathtt{let}\{x := \mathtt{mk}(u_1)\}\rho_1]\downarrow$$

for any $R \in \mathbb{R}_\Gamma$. We show by computation induction that

$$(\widehat{\Gamma}\{x := \mathtt{mk}(u_0)\}; \widehat{e})^{\{p:=\rho_0\}}\downarrow \quad \text{iff} \quad (\widehat{\Gamma}\{x := \mathtt{mk}(u_1)\}; \widehat{e})^{\{p:=\rho_1\}}\downarrow$$

for any $\widehat{\Gamma}[\widehat{e}]$ with at most p free, p distinct from x. As usual the only interesting case is when $\widehat{e}$ has the form $\widehat{R}[\mathtt{app}(p, \widehat{v})]$. Then as in the invariance lemma we have

$$\widehat{\Gamma}\{x := \mathtt{mk}(u_j)\}; \widehat{R}[\mathtt{app}(\rho_j, \widehat{v})] \overset{*}{\mapsto} \widehat{\Gamma}'\{x := \mathtt{mk}(v_j)\}; \widehat{R}[w]$$

uniformly in p for some v_0, v_1 satisfying Φ and some atom w. $\square_{\mathbf{abstract}}$

Proof (†): By (**abstract**) it is sufficient to establish (**0,1,2**) taking Φ to be $\mathtt{isnat}(u_0) \cong \mathtt{t}$, ρ_0 to be $\lambda d.\mathtt{set}(x, +(\mathtt{get}(x), n))$, and ρ_1 to be $\lambda d.\mathtt{nil}$. (**0**) is trivial and (**1**) follows since neither function depends on its argument. To establish (**2**) take $v_0 = u_0 + n$, $v_1 = u_1$, $w = \mathtt{nil}$. $\square_\dagger$

Part III: Classes

5. Adding Classes to the Language

Using methods of (Feferman, 1975; Feferman, 1990) and (Talcott, 1993), 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 maybe required to provide a formal semantics. Even here classes are likely to play an important role.

5.1. Syntax of Classes

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

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

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 ($\mathbb{W}$):

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

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.

5.2. Semantics of Classes

To give semantics to the extended language, we extend the satisfaction relation as follows. Firstly we let $\mathbb{K}_{\mathrm{Dom}(\Gamma)}$, the set of class values over Γ, be the set of subsets of $\mathbb{V}_{\mathrm{Dom}(\Gamma)}$ closed under $\cong$.[4] We extend value substitutions to map class variables to class values. This is used to define $[K]^{\sigma}_{\Gamma}$, 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

[4] Another possible choice for the set of class values is the set of definable sets, i.e. the set of class code extensions (cf. (Feferman, 1979; Talcott, 1993)).

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.[5]

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

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

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:

$$\Gamma \models e \in K[\sigma] \;\Leftrightarrow\; (\exists v \in \mathbb{V}_{\mathrm{Dom}(\Gamma)})(\Gamma; e^{\sigma} \overset{*}{\mapsto} \Gamma; v \;\wedge\; v \in [K]^{\sigma}_{\Gamma})$$
$$\Gamma \models (\forall X)\Phi[\sigma] \;\Leftrightarrow\; (\forall C \in \mathbb{K}_{\mathrm{Dom}(\Gamma)})(\Gamma \models \Phi[\sigma\{X := C\}])$$

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 (Tofte, 1988; Tofte, 1990). [6]

5.3. Simple Examples

We define (extensional) equality and subset relations on classes in the usual manner.

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

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

Lemma (class):

(neq) $\neg(e_0 \cong e_1 \wedge \mathtt{let}\{x := e_0\}[\![x \in K]\!] \Rightarrow \mathtt{let}\{x := e_1\}[\![x \in K]\!])$

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

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

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

Note also that the usual form of (**allE**) is false:

$$\neg((\forall X)\Phi[X] \Rightarrow \Phi[K])$$

[5] 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 $[K]^{\sigma}_{\Gamma}$.

[6] This is the only form of non-expansiveness allowed in Tofte's type system, but in general non-expansive means without allocation effect, but possibly with read/write effects.

A counterexample will be given below after some additional notation has been introduced.

We introduce the following class constants. $\emptyset$ is the empty class. **Val** is the class of all values. **Nil** is the class containing the single element `nil`. **Bool** is the class containing two elements: `t` and `nil`. **Nat** is the class containing the natural numbers $n \in \mathbb{N}$. **Cell** is the class of memory cells.

$$\emptyset = \{x \mid x \not\cong x\}$$

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

$$\mathbf{Nil} = \{x \mid x \cong \mathtt{nil}\}$$

$$\mathbf{Bool} = \{\mathtt{t}, \mathtt{nil}\} = \{x \mid x \cong \mathtt{t} \lor x \cong \mathtt{nil}\}$$

$$\mathbf{Cell} = \{x \mid \mathtt{cell}(x) \cong \mathtt{t}\}$$

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

Note that the interpretation of $\emptyset$ and **Nil** is independent of memory contexts, while $[\mathbf{Val}]_\Gamma = \mathbb{V}_{\mathrm{Dom}(\Gamma)}$ and $[\mathbf{Cell}]_\Gamma = \mathrm{Dom}(\Gamma)$. 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.

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

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

Definition ($\to \overset{\mathrm{p}}{\to} \overset{\mu}{\to}$): There are a number of function spaces in our world. The three simplest are total, partial and memory.

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

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

$$X_1, \ldots, X_n \overset{\mu}{\to} Y = \{f \mid (\forall x_1 \in X_1, \ldots, x_n \in X_n)(\mathtt{let}\{y := \mathtt{app}(f, x_1, \ldots, x_n)\}[\![y \in Y]\!])\}$$

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

(mk) $(\forall X)(\mathtt{mk} \in X \overset{\mu}{\to} \mathbf{Cell}[x])$

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

(set) $\mathtt{set} \in \mathbf{Cell}, \mathbf{Val} \overset{\mu}{\to} \mathbf{Nil}$

However (as we shall see later) a more detailed statement concerning `set`s functionality/effects is false:

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

We can elaborate the notion of memory functionality by making explicit effects such as the class of the arguments after application of the function.

We may give the promised counter example for classical (allE). Let $K = \{x \mid \Phi_{\text{1-cell}}\}$, and $\Phi = (\forall x \in X)(\mathtt{let}\{z := \mathtt{mk}(x)\}[\![z \in \mathbf{Cell}[X]]\!])$. Note that Φ is the body of the $\mathtt{mk}$ functionality formula. Now $(\forall X)\Phi$ holds, but $\Phi[K]$ fails for any memory with singleton domain.

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

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

Then $\Phi_{\neg\text{expand}}(e)$ says that execution of e does non expand the memory, although it might modify contents of existing cells.

5.4. Caveats

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

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

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

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

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

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 (as has been demonstrated in §2.3.1) 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 $\mathbf{Val} \overset{\mu}{\rightharpoonup} \mathbf{Cell}[\mathbf{Nil}]$, and $\Phi_{\neg\text{write}}[\mathbf{Cell}](e)$ be as defined below. Then

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

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

$$\mathtt{let}\{x := e_0\}[\![x \in T]\!] \quad \text{for} \quad 0 \leq j \leq 3$$

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

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

Another example of functionality can be found in the recent work of Cardelli and Nelson. There they use a particular notion of of function space in the formal specification of the semantics of Modula 3. This space is the set

$$X, \Phi_0 \overset{\mu}{\rightharpoonup}_S Y, \Phi_1$$

of all non-expansive procedures with argument type X and result type Y which satisfy pre-condition Φ_0 and post-condition Φ_1 and which can only modify cells in the set S. In our framework we cannot express that an expression e only modifies cells in a certain class, we can only express that the expression only modify cells in the set S up to operational equivalence. Modulo this point, this function space can be represented as

$$\begin{aligned}\{f \mid (\forall x \in X)(\Phi_0(x) \Rightarrow & \\ & \mathtt{let}\{y := f(x)\}[\![y \in Y \wedge \Phi_1(y)]\!]) \wedge \\ & \Phi_{\neg\mathrm{expand}}(\mathtt{app}(f, x)) \wedge \\ & \Phi_{\neg\mathrm{write}}[S](\mathtt{app}(f, x))\}\end{aligned}$$

where

$$\begin{aligned}\Phi_{\neg\mathrm{write}}[S](e) \Leftrightarrow (\exists X_1)(X_1 \equiv \mathbf{Cell}[\mathbf{Val}] \wedge & \\ (\forall x \in X)(\forall z \in \mathbf{Val})(\forall y \in (X_1 - S)) & \\ & (\mathtt{get}(y) \cong z \Rightarrow \mathtt{let}\{w := f(x)\}[\![\mathtt{get}(y) \cong z]\!]))\}\end{aligned}$$

5.5. Classes vs Types

5.5.1. The Functional Case

(Feferman, 1990) 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 too 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 (Smith, 1988; Talcott, 1993).

Theorem (FixTypeFails): Let $\mathtt{Y}_v$ by any fixed-point combinator (such that $f(\mathtt{Y}_v(f)) = \mathtt{Y}_v(f)$). Then it is not the case that

$$f \in (C \to C) \Rightarrow \mathtt{Y}_v(f) \in C$$

for all function classes C ($C \subseteq A \overset{\mathrm{P}}{\to} B$ for some classes A, B).

Proof (FixTypeFails): Define the P to be the class of *strictly* partial maps from **Nat** to **Nat**:

$$P = \{g \in \mathbf{Nat} \overset{\mathrm{P}}{\to} \mathbf{Nat} \mid (\exists n \in \mathbf{Nat})(\neg \Downarrow g(n))\}$$

Let

$$f = \lambda p.\lambda n.\mathtt{if}(\mathtt{eq}(n, 0), n, p(n-1))$$

Then we can prove

(1) $\quad f \in P \to P$

(2) $\quad \mathtt{Y}_v(f) \in \mathbf{Nat} \to \mathbf{Nat}$

(**1**) follows by simple properties of `if`, `eq` and arithmetic (**2**) follows by induction on **Nat** using the fixed point property of Y. Consequently, $\neg(\mathtt{Y}_v(f) \in P)$ □

5.5.2. The Imperative Case

The situation becomes more problematic when references are added, even in the simply typed (or monomorphic) case. Naïve attempts to represent ML types as classes fails in sense that 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,. not just of the text being executed, but also of the contents of any cell in memory. This requirement is a strong form of subject reduction. One that does not seem to be expressible using classes (quantifying over types, whatever they may be, seems problematic). Our analysis indicates that ML types are therefore 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. (Tofte, 1988; Tofte, 1990; Leroy and Wies, 1990; Felleisen and Wright, 1991)). In this system types are built from base types, **Nat** and **Nil**, using function space and reference constructions.

$$\mathbb{T} = \mathbb{N} + \{\mathtt{nil}\} + (\mathbb{T} \overset{\lambda}{\to} \mathbb{T}) + \mathbf{ref}(\mathbb{T})$$

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

$$\mathtt{mk}_\tau : \tau \overset{\lambda}{\to} \mathbf{ref}(\tau)$$

$$\mathtt{get}_\tau : \mathbf{ref}(\tau) \overset{\lambda}{\to} \tau$$

$$\mathtt{set}_\tau : \mathbf{ref}(\tau) \overset{\lambda}{\to} \tau \overset{\lambda}{\to} \{\mathtt{nil}\}$$

which will be encoded by the corresponding η-ized operations

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

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 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 $\underline{\mathbb{N}} = \mathbf{Nat}$ and $\underline{\{\mathtt{nil}\}} = \mathbf{Nil}$. We also require that belonging to a class via $\Phi_:$ implies membership in that class:

$$\Phi_:(e, \underline{\tau}) \ \Rightarrow\ \mathtt{let}\{z := e\}[\![z \in \underline{\tau}]\!].$$

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

(1a) $\Phi_{:}(x, \underline{\tau}) \Rightarrow \Phi_{:}(\mathtt{mk}(x), \underline{\mathbf{ref}(\tau)})$

(1b) $\Phi_{:}(x, \underline{\mathbf{ref}(\tau)}) \Rightarrow \Phi_{:}(\mathtt{get}(x), \underline{\tau})$

(1c) $(\Phi_{:}(x, \underline{\mathbf{ref}(\tau)}) \wedge \Phi_{:}(y, \underline{\tau})) \Rightarrow \Phi_{:}(\mathtt{set}(x, y), \mathbf{Nil})$

(2) $\bigwedge_{i<n} \Phi_{:}(x_i, \underline{\tau_i}) \Rightarrow ((\Phi_{:}(x, \underline{\tau_a}) \Rightarrow \Phi_{:}(e, \underline{\tau_b})) \Rightarrow \Phi_{:}(\lambda x.e, \underline{\tau_a \overset{\lambda}{\to} \tau_b}))$

(3) $\bigwedge_{i<n} \Phi_{:}(x_i, \underline{\tau_i}) \Rightarrow ((\Phi_{:}(e, \underline{\tau_a \overset{\lambda}{\to} \tau_b}) \wedge \Phi_{:}(e_a, \underline{\tau_a})) \Rightarrow (\Phi_{:}(\mathtt{app}(e, e_a), \underline{\tau_b})$

The principle of *type faithfulness* (Abadi et al., 1991)

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

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

$$\underline{ref(\tau)} = \mathbf{Cell}[\underline{\tau}] \quad \underline{\tau_a \overset{\lambda}{\to} \tau_b} = \underline{\tau_a} \overset{\mu}{\to} \underline{\tau_b} \quad \Phi_{:}(e, \underline{\tau}) = \mathtt{let}\{z := e\}[\![z \in \underline{\tau}]\!]$$

However this encoding is not sound. One source of trouble is that in (**3**) the evaluation of e_a may invalidate the the assumptions that $\bigwedge_{i<n} \Phi_{:}(x_i, \underline{\tau_i})$. A counterexample to this is the following:

$$y \in \mathbf{Cell}[\mathbf{Nat}] \Rightarrow \mathtt{let}\{x := \mathtt{set}(y, \mathtt{t})\}[\![x \in \mathbf{Nil}]\!]$$

$$y \in \mathbf{Cell}[\mathbf{Nat}] \Rightarrow \mathtt{let}\{z := \lambda w.\mathtt{get}(y)\}[\![z \in \mathbf{Nil} \to \mathbf{Nat}]\!]$$

But the resulting conclusion

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

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:

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

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:

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

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

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

Consequently

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

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

But also note that

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

These issues are currently under scrutiny.

6. Using Classes

In this section we give an extended example of the use of VTLoE in specifying and verifying a program.

6.1. Class Fixed Points and Induction

The definitions and results in this subsection are taken from (Talcott, 1993) where more details, proofs, and examples can be found. Here we focus on minimum fixed points. A similar development can be carried our for maximum fixed points. For any formula $\Phi[X]$ we can form the intersection of all classes X satisfying $\Phi[X]$.

$$\bigcap_X \Phi[X] = \{x \mid (\forall X)(\Phi[X] \Rightarrow x \in X)\}$$

If $\Phi[X]$ is preserved by intersection over non-empty sets of classes and is satisfiable, then $\bigcap_X \Phi[X]$ is the least class satisfying $\Phi[X]$. This notion is not formalizable in our theory in full generality. However there is an important special case which can be formalized. Namely, formulas representing closure conditions.

We say a class operator T is monotone if $X \subseteq Y$ implies $T[X] \subseteq T[Y]$. Let $T[X]$ be a monotone operator on classes. Then the formula $T[X] \subseteq X$ is preserved by intersection over non-empty sets of classes and is satisfied by **Val**. Hence $\bigcap_X (T[X] \subseteq X)$ is the smallest class X satisfying $T[X] \subseteq X$ and by monotonicity the smallest fixed point of $T[X]$. This is formalized by the following definitions and theorems (where we assume T is a class operator with distinguished variable X).

Definition (min):

$$T^{\cap} = \bigcap_X (T[X] \subseteq X)$$

Theorem (min): If $T[X]$ is a monotone operator on classes, then (min.fix) $T^{\cap} \equiv T[T^{\cap}]$

(min) $T[Y] \subseteq Y \Rightarrow T^{\cap} \subseteq Y$

Using the least fixed-point construction we define $\mathbb{N}$, the class of natural numbers, and A^*, the class of sequences from A. $\mathbb{N}$ is the least class containing 0 and closed under `+1`.

Definition (Nat):

$$T_{\mathbb{N}}[X] = \{0\} \cup \{x \mid (\exists y \in X)(x \cong \mathtt{+1}(y))\}$$
$$\mathbb{N} = T_{\mathbb{N}}{}^{\cap}$$

From this definition we have the usual induction principle for $\mathbb{N}$. For any formula $\Phi[x]$ to show that $x \in \mathbb{N} \Rightarrow \Phi[x]$ we need only show that $\Phi[0]$ and $\Phi[x] \Rightarrow \Phi[\mathtt{+1}(x)]$.

We define the operator Z^* which constructs from any class Z the finite sequences from Z by forming the least class containing `nil` and closed under pairing with elements from Z

Definition (Fseq):

$$T_*[Z][X] = \{\mathtt{nil}\} \cup \{x \mid (\exists z \in Z, y \in X)(x \cong \mathtt{pr}(z, y))\}$$
$$Z^* = T_*[Z]^{\cap}$$

We have the fixed point and induction principles for finite sequences.

Theorem (Fseq):

(fix) $x \in A^* \Leftrightarrow (x \cong \mathtt{nil} \vee (\exists z \in Z, y \in X)(x \cong \mathtt{pr}(z, y)))$

$\Leftrightarrow x \cong \mathtt{nil} \vee (\mathtt{ispr}(x) \cong \mathtt{t} \wedge \mathtt{fst}(x) \in A \wedge \mathtt{snd}(x) \in A^*)$

(ind) $\mathtt{nil} \in X \wedge (\forall x)(\mathtt{ispr}(x) \cong \mathtt{t} \wedge \mathtt{fst}(x) \in A \wedge \mathtt{snd}(x) \in X \Rightarrow x \in X) \Rightarrow A^* \subseteq X$

6.2. Representing S-expressions

We represent binary cells as unary cells containing pairs and define S-expression operations in terms of pairing and unary-cell operations.

Definition (S-expression operations): A *cons* cell is a (unary) cell containing a pair.

$$\mathtt{cons} = \lambda x, y.\mathtt{mk}(\mathtt{pr}(x, y))$$
$$\mathtt{cons?} = \lambda z.\mathtt{and}(\mathtt{cell}(z), \mathtt{ispr}(\mathtt{get}(z)))$$
$$\mathtt{car} = \lambda z.\mathtt{fst}(\mathtt{get}(z))$$
$$\mathtt{cdr} = \lambda z.\mathtt{snd}(\mathtt{get}(z))$$
$$\mathbf{Cons}[X, Y] = \{x \mid \mathtt{cons?}(x) \cong \mathtt{t} \wedge \mathtt{car}(x) \in X \wedge \mathtt{cdr}(x) \in Y\}$$
$$\mathtt{setcar} = \lambda z, y.\mathtt{set}(z, \mathtt{pr}(y, \mathtt{cdr}(z)))$$
$$\mathtt{setcdr} = \lambda z, y.\mathtt{set}(z, \mathtt{pr}(\mathtt{car}(z), y))$$

The operations `cons`, `cons?`, `car`, `cdr` have the expected functionality. However assigning functionality to the modifiers `setcar`, `setcdr` is problematic.

Lemma (S-expression functionality):

$$\mathtt{cons} \in (X, Y \xrightarrow{\mu} \mathbf{Cons}[X,Y])$$

$$\mathtt{car} \in (\mathbf{Cons}[X,Y] \xrightarrow{\mu} X)$$

but

$$\neg(\forall XYZ)(\forall x \in X)(\forall w \in \mathbf{Cons}[Z,Y])$$
$$\mathtt{let}\{z := \mathtt{setcar}(w,x)\}\llbracket z \in \mathbf{Nil} \land w \in \mathbf{Cons}[X,Y]\rrbracket$$

Proof : To see the problem for `setcar` let X, Y, Z be the class of well-founded S-expressions over $\mathbb{N}$. If x, w are the same cons cell, then after assignment there is a cycle starting from w and hence the class of w has changed. □

The usual (first-order) laws for S-expression operations hold (Mason and Talcott, 1991a; Mason and Talcott, 1992a). We use $x = [x_a, x_d]$ to abbreviate the formula $\mathtt{cons?}(x) \cong \mathtt{t} \land \mathtt{car}(x) \cong x_a \land \mathtt{car}(x) \cong x_d$.

Lemma (cons axioms):

(i) $\mathtt{let}\{x := \mathtt{cons}(x_a, x_d)\}\llbracket\neg(x \cong y) \land x = [x_a, x_d]\rrbracket$ $\quad x$ fresh

(ii) $z = [z_a, z_d] \Rightarrow \mathtt{let}\{x := \mathtt{cons}(x_a, x_d)\}\llbracket z = [z_a, z_d]\rrbracket$

(iii) $\Downarrow \mathtt{cons}(x, y)$

(iv) $\mathtt{let}\{y := e_0\}\mathtt{let}\{x := \mathtt{cons}(x_a, x_d)\}e_1 \cong \mathtt{let}\{x := \mathtt{cons}(x_a, x_d)\}\mathtt{let}\{y := e_0\}e_1$

$\quad x \notin \mathrm{FV}(e_0)$, $y \notin \mathrm{FV}(x_a) \cup \mathrm{FV}(x_d)$

(v) $\mathtt{cons?}(z) \Rightarrow (\exists z_a, z_d)(z = [z_a, z_d])$

(vi) $\mathtt{cons?}(x) \cong \mathtt{t} \land \mathtt{cons?}(y) \cong \mathtt{nil} \Rightarrow \neg(x \cong y)$

(vii) $\mathtt{atom}(x) \Rightarrow \neg(\mathtt{cons?}(x))$

The assertion, (**cons.i**), describes the allocation effect of a call to `cons`. While (**cons.ii**) expresses what is unaffected by a call to `cons` The assertion, (**cons.iii**), expresses the totality of `cons`. The `cons` delay axiom, (**cons.i**), asserts that the time of allocation has no discernable effect on the resulting cons cell.

Lemma (setcar axioms):

(i) $\mathtt{cons?}(z) \Rightarrow \mathtt{let}\{x := \mathtt{setcar}(z, y)\}\llbracket\mathtt{car}(z) \cong y \land x \cong \mathtt{nil}\rrbracket)$

(ii) $(y \cong \mathtt{car}(z) \land \neg(w \cong z)) \Rightarrow \mathtt{let}\{x := \mathtt{setcar}(w, v)\}\llbracket y \cong \mathtt{car}(z)\rrbracket$

(iii) $y \cong \mathtt{cdr}(z) \Rightarrow \mathtt{let}\{x := \mathtt{setcar}(w, v)\}\llbracket y \cong \mathtt{cdr}(z)\rrbracket$

(iv) $\mathtt{cons?}(z) \Rightarrow \Downarrow \mathtt{setcar}(z, x)$

(v) $\neg(x \cong y) \Rightarrow \mathtt{seq}(\mathtt{setcar}(x, z), \mathtt{setcar}(y, w)) \cong \mathtt{seq}(\mathtt{setcar}(y, w), \mathtt{setcar}(x, z))$

(vi) $\mathtt{seq}(\mathtt{setcar}(x, y_0), \mathtt{setcar}(x, y_1)) \cong \mathtt{setcar}(x, y_1)$

(vii) $\mathtt{let}\{z := \mathtt{cons}(x, y)\}\mathtt{seq}(\mathtt{setcar}(z, w), e) \cong \mathtt{let}\{z := \mathtt{cons}(w, y)\}e$

$\quad z$ not free in w

(viii) $\mathtt{seq}(\mathtt{setcar}(x, z), \mathtt{setcdr}(y, w)) \cong \mathtt{seq}(\mathtt{setcdr}(y, w), \mathtt{setcar}(x, z))$

We have only included the axioms for `setcar`, the axioms for `setcdr` are analogous.

6.2.1. Caveats

An alternative representation of *cons* cells would be as a pair of distinct unary cells. As an implementation this makes modification of cons cells more efficient (finer grain access). However many of the standard properties of S-expressions operations fail to hold for this representation. An example of the failure is the following. In the Lisp S-expressions world, if two cons cells are not identical then they are disjoint. Assigning to the `car` of one cell can not change the contents of the `cdr` of that cell or any other. Consequently, the operations `setcar` and `setcdr` commute. In the `cons` pairs world the situation is more complicated. Although the two cells of a cons pair are required to be distinct, two cons pairs x, y can have disjoint component cells or they can share in a number of ways. For example:

$$\mathtt{fst}(x) \cong \mathtt{fst}(y) \ \wedge\ \mathtt{snd}(x) \cong \mathtt{snd}(y)$$

$$\mathtt{fst}(x) \cong \mathtt{fst}(y) \ \wedge\ \neg(\mathtt{snd}(x) \cong \mathtt{snd}(y))$$

For example let a, b be unary cells containing `t`, and let $x = \mathtt{pr}(a, b)$, $y = \mathtt{pr}(b, a)$. Then

$$\mathtt{cdr}(y) \cong \mathtt{t} \ \wedge\ \mathtt{seq}(\mathtt{setcar}(x, \mathtt{nil}), [\![\mathtt{cdr}(y) \cong \mathtt{nil}]\!])$$

$$\neg(\mathtt{seq}(\mathtt{setcar}(x, 0), \mathtt{setcdr}(y, 1)) \cong \mathtt{seq}(\mathtt{setcdr}(y, 1), \mathtt{setcar}(x, 0)))$$

6.3. Well-founded S-expressions and Lists

Traditionally the set of (well-founded) S-expressions over some set A is defined to be the least set containing A and closed under the `cons` operation. This corresponds to the set of binary trees with leaves in A. Using minimal fixed points we can define this set in our theory. The only point that requires attention is that we need to formulate the closure operation in terms of existing cons cells. Thus we define the S-expression closure operation for a class of atoms A, $T_{\mathbf{Sexp}}[A]$, and the class of S-expressions with atoms in A as follows.

$$T_{\mathbf{Sexp}}[A][X] = A \cup \{x \mid \mathtt{cons?}(x) \ \wedge\ \mathtt{car}(x) \in X \ \wedge\ \mathtt{cdr}(x) \in X\}$$

$$\mathbf{Sexp}[A] = T_{\mathbf{Sexp}}[A]^{\cap}$$

The S-expression induction principle derived from this definition is

Theorem (Sexp):

$$\text{(fix)} \quad (\forall x)(x \in \mathbf{Sexp}[A] \Leftrightarrow (x \in A) \ \vee\ (\mathtt{cons?}(x) \cong \mathtt{t} \ \wedge\ \mathtt{car}(x) \in \mathbf{Sexp}[A] \ \wedge\ \mathtt{cdr}(x) \in \mathbf{Sexp}[A]))$$

$$\text{(ind)} \quad (\forall X)(A \subset X \ \wedge\ (\forall x)(\mathtt{cons?}(x) \cong \mathtt{t} \ \wedge\ \mathtt{car}(x) \in X \ \wedge\ \mathtt{cdr}(x) \in X \Rightarrow x \in X) \Rightarrow \mathbf{Sexp}[A] \subseteq X)$$

One of the properties that should hold of well-founded S-expressions is that there are no cycles – the only `car`-`cdr` path that leads from a well-founded S-expression to itself is the empty path. This can be proved using S-expression induction. For this we define the locator function `loc` that takes an S-expression and a sequence of `t`s and `nil`s (representing a `car`-`cdr` path) and returns the S-expression located by the path if it lies within the S-expression, and returns a newly created cell otherwise.

$$\begin{array}{l}
\mathtt{loc} = \mathtt{Y}(\lambda l.\lambda x.\lambda p.\mathtt{if}(p, \\
\qquad\qquad\qquad\quad \mathtt{if}(\mathtt{cons?}(x), \\
\qquad\qquad\qquad\qquad l(\mathtt{if}(\mathtt{eq}(\mathtt{fst}(p), 0), \mathtt{car}(x)\mathtt{cdr}(x)), \mathtt{snd}(p)) \\
\qquad\qquad\qquad\qquad \mathtt{mk}(\mathtt{nil})), \\
\qquad\qquad\qquad\quad x)
\end{array}$$

Theorem (Wf-Sexp):

$$(\forall x \in \mathbf{Sexp}[A])(\forall p \in \mathbf{Bool}^*)(\mathtt{loc}(x, p) \cong x \;\Rightarrow\; p \cong \mathtt{nil})$$

This is easily proved by S-expression induction. We have restricted attention to S-expressions over Lisp atoms. We could allow other cells, but we must omit cells containing pairs from the base set as there is no way to distinguish these cons cells from S-expression cons cells.

A (mutable) list is either the empty list (`nil`) or a cons cell whose `cdr` is a list. We define the class **List**, of acyclic lists, as the least fixed point of the operator $T_{\mathbf{List}}$ where

Definition (List):

$$T_{\mathbf{List}}[X] = \mathbf{Nil} \cup \{x \mid \mathtt{cons}(x) \cong \mathtt{t} \wedge \mathtt{cdr}(x) \in X\}$$

$$\mathbf{List} = {T_{\mathbf{List}}}^{\cap}$$

Theorem (List):

(fix) $\quad (\forall x)(x \in \mathbf{List} \;\Leftrightarrow\; (x \cong \mathtt{nil} \vee (\mathtt{cons?}(x) \cong \mathtt{t} \wedge \mathtt{cdr}(x) \in \mathbf{List})))$

(ind) $\quad (\forall X)(\mathtt{nil} \in X \wedge (\forall x)(\mathtt{cons?}(x) \cong \mathtt{t} \wedge \mathtt{cdr}(x) \in X \;\Rightarrow\; x \in X) \;\Rightarrow\; \mathbf{List} \subseteq X)$

We formalize the acyclic property of lists as follows.

Definition (iterate,length):

$$\mathtt{iterate} = \lambda f.\mathtt{Y}(\lambda i.\lambda x.\lambda n.\mathtt{if}(\mathtt{eq}(n, 0), x, i(f(x), n - 1)))$$

$$\mathtt{len} = \mathtt{Y}(\lambda l.\lambda x.\mathtt{if}(\mathtt{cons?}(x), 1 + l(\mathtt{cdr}(x)), 0))$$

Lemma (length):

$$(\forall x \in \mathbf{List})(\exists n \in \mathbb{N})(\mathtt{iterate}(\mathtt{cdr})(x, n) \cong \mathtt{nil})$$

$$(\forall x \in \mathbf{List})(\mathtt{len}(x) \in \mathbb{N})$$

$$(\forall x \in \mathbf{List})(\mathtt{iterate}(\mathtt{cdr})(x, \mathtt{len}(x)) \cong \mathtt{nil})$$

6.4. Iterative List Traversal

In this example we deal with two programs for appending lists. The first is the traditional pure program, append, that concatenates its first argument with its second, copying the top level list structure of its first argument in the process. This example was first dealt with in depth in (Mason, 1986b; Mason, 1988)

Definition (append):

$$\texttt{append}(x, y) \leftarrow \texttt{if}(\texttt{eq}(x, \texttt{nil}), y, \texttt{cons}(\texttt{car}(x), f(\texttt{cdr}(x), y)))$$

The problem with this definition of append is that to perform the cons in the non-trivial case we must first compute the result of append-ing the cdr of the first argument onto the second. This is easily seen to entail that append will use up stack proportional to the length of its first argument. The second program is an iterative version written using setcdr. It utilizes the destructive operations in the following way. Instead of waiting around for the result of doing the append of the cdr of the first argument before it can do the cons, it performs the cons with a, possibly, dummy cdr value and later on in the computation rectifies this haste. The result is a program that need not use any stack.

Definition (iterative.append):

```
iterative.append($x, y$) ←
    if(eq($x$, nil), $y$, let{$w$ := cons(car($x$), $y$)}seq(it.app(cdr($x$), $w$), $w$))
it.app($x, w$) ←
    if(eq($x$, nil),
       $x$,
       let{$z$ := cons(car($x$), cdr($w$))}seq(setcdr($w, z$), it.app(cdr($x$), $z$)))
```

The following result could and should be taken as verification of the correctness of the iterative.append program, since we are reducing its behavior to that of a very simple program.

Theorem (append): $x \in$ **List** $\Rightarrow$ $\texttt{iterative.append}(x, y) \cong \texttt{append}(x, y)$

Before proving (**append**) we prove following lemma. It demonstrates that one can postpone setting the cdr of a newly created cell until the cell is referenced. This is the key property used in the optimization of append to iterative.append. An analogous result holds for the car.

Lemma (delaying assignment): If $w \notin \mathrm{FV}(e)$ and $w \neq z$ then

$$\texttt{let}\{w := \texttt{cons}(x, y)\}[\![\texttt{seq}(\texttt{setcdr}(w, z), e, e') \cong \texttt{seq}(e, \texttt{setcdr}(w, z), e')]\!]$$

Proof (delaying assignment):

$$\mathtt{let}\{w := \mathtt{cons}(x, y)\}\mathtt{seq}(\mathtt{setcdr}(w, z), e, e') \cong \mathtt{let}\{w := \mathtt{cons}(x, z)\}\mathtt{seq}(e, e')$$

by (**setcdr.vii**)

$$\cong \mathtt{seq}(e, \mathtt{let}\{w := \mathtt{cons}(x, z)\}e') \qquad \text{by } (\textbf{cons.iv})$$

$$\cong \mathtt{seq}(e, \mathtt{let}\{w := \mathtt{cons}(x, y)\}\mathtt{seq}(\mathtt{setcdr}(w, z), e'))$$

by (**setcdr.vii, ca.i, ca.ii**)

$$\cong \mathtt{let}\{w := \mathtt{cons}(x, y)\}\mathtt{seq}(e, \mathtt{setcdr}(w, z), e') \qquad \text{by } (\textbf{cons.iv})$$

□

Proof (append): Since $x \in \textbf{List} \Rightarrow x \cong \mathtt{nil} \vee \mathtt{cons?}(x) \cong \mathtt{t}$ by (**list.fix**), we argue by cases.

($x \cong \mathtt{nil}$)

$$x \cong \mathtt{nil} \Rightarrow \mathtt{iterative.append}(x, y)$$

$$\cong \mathtt{if}(\mathtt{eq}(x, \mathtt{nil}), y, \mathtt{let}\{w := \mathtt{cons}(\mathtt{car}(x), y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{cdr}(x), w), w))$$

by the definition of `iterative.append`

$$\cong \mathtt{if}(\mathtt{t}, y, \mathtt{let}\{w := \mathtt{cons}(\mathtt{car}(\mathtt{car}(x)), y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{cdr}(\mathtt{nil}), w), w))$$

by (**atomic.v, implies.ii**)

$$\cong y \qquad \text{by } (\textbf{atomic.viii, atomic.ix})$$

$$\cong \mathtt{append}(x, y) \qquad \text{by identical reasoning}$$

□

($\mathtt{cons?}(x) \cong \mathtt{t}$) In this case we use the following two lemmas (proved below)

Lemma (A):

$$x = [x_a, x_d] \Rightarrow \mathtt{cons}(x_a, \mathtt{append}(x_d, y)) \cong \mathtt{append}(x, y)$$

Lemma (B):

$$z \in \textbf{List} \Rightarrow$$

$$(\forall x, y)(\mathtt{cons}(x, \mathtt{append}(z, y)) \cong \mathtt{let}\{w := \mathtt{cons}(x, y)\}\mathtt{seq}(\mathtt{it.app}(z, w), w)$$

By (**cons.v**) we have $(\exists x_a, x_d)(x = [x_a, x_d])$, so by classical logic it suffices to reason under the additional assumption $x = [x_a, x_d]$. Thus, we may make the assumption $x \in \textbf{List} \wedge x = [x_a, x_d]$. The first three steps unfold and simplify the definition of

$\mathtt{iterative.append}$. The next two steps evaluate $\mathtt{car}(x)$ and $\mathtt{cdr}(x)$ relative to the assumptions.

$$x \in \textbf{List} \wedge x = [x_a, x_d] \Rightarrow \mathtt{iterative.append}(x, y)$$

$$\cong \mathtt{if}(\mathtt{eq}(x, \mathtt{nil}), y, \mathtt{let}\{w := \mathtt{cons}(\mathtt{car}(x), y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{cdr}(x), w), w))$$

unfolding $\mathtt{iterative.append}$

$$\cong \mathtt{if}(\mathtt{nil}, y, \mathtt{let}\{w := \mathtt{cons}(\mathtt{car}(x), y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{cdr}(x), w), w))$$

by (**cons.vii, cons.vi, implies.ii**)

$$\cong \mathtt{let}\{w := \mathtt{cons}(\mathtt{car}(x), y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{cdr}(x), w), w)) \qquad \text{by } (\textbf{atomic.vii})$$

$$\cong \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{cdr}(x), w), w)) \qquad \text{by } (\textbf{implies.ii})$$

$$\cong \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{it.app}(x_d, w), w)) \qquad \text{by } (\textbf{cons.ii, implies.ii, ca.ii})$$

$$\cong \mathtt{cons}(x_a, \mathtt{append}(x_d, y)) \qquad \text{by } (\textbf{B})$$

$$\cong \mathtt{append}(x, y) \qquad \text{by } (\textbf{A})$$

□

Proof (A): This is left as an exercise. $\square_{\mathbf{A}}$

Proof (B): The proof is by List induction, using (**class.allE**) to instantiate (**List.ind**) with

$$Z = \{z \mid$$
$$(\forall x_a, y)(\mathtt{cons}(x_a, \mathtt{append}(z, y)) \cong \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{it.app}(z, w), w)\}$$

We thus must show $\mathtt{nil} \in Z$ and $(\forall x)(\mathtt{cons?}(x) \cong \mathtt{t} \wedge \mathtt{cdr}(x) \in Z \Rightarrow x \in Z)$; from this and the instantiated (**List.ind**) we obtain **List** $\subseteq Z$, so by (**class.ca**), (**B**) directly follows.

Show $\mathtt{nil} \in Z$.

$$\mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{nil}, w), w))$$

$$\cong \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{nil}, w)$$

by the definition of $\mathtt{it.app}$, and (**atomic.v, atomic.vii, ca.i, ca.ii**)

$$\cong \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}w \qquad \text{by } (\textbf{atomic.iv, ca.i})$$

$$\cong \mathtt{cons}(x_a, y) \qquad \text{by } (\mathtt{let}_c\textbf{.i})$$

$$\cong \mathtt{cons}(x_a, \mathtt{append}(z, y)) \qquad \text{as in the } x \cong \mathtt{nil} \text{ case of } (\textbf{append}), (\textbf{implies.ii})$$

□

Show $\mathtt{cons?}(z) \cong \mathtt{t} \wedge \mathtt{cdr}(z) \in Z \Rightarrow z \in Z$. By (**cons.v**) we strengthen the assumptions to show $z = [z_a, z_d] \wedge \mathtt{cdr}(z) \in Z \Rightarrow z \in Z$. Then,

$$
\begin{aligned}
& z = [z_a, z_d] \wedge w = [x_a, y] \Rightarrow \\
& \quad \mathtt{it.app}(z, w) \\
& \qquad \cong \mathtt{let}\{u := \mathtt{cons}(\mathtt{car}(z), \mathtt{cdr}(w))\}\mathtt{seq}(\mathtt{setcdr}(w, u), \mathtt{it.app}(\mathtt{cdr}(z), u)) \\
& \qquad\qquad \text{by } (\textbf{cons.vii, cons.vi, atomic.vi, atomic.vii, implies.ii}) \\
& \qquad \cong \mathtt{let}\{u := \mathtt{cons}(z_a, y)\}\mathtt{seq}(\mathtt{setcdr}(w, u), \mathtt{it.app}(\mathtt{cdr}(z), u), w) \\
& \qquad\qquad \text{by } (\textbf{implies.ii}) \text{ applied twice}
\end{aligned}
$$

Now we introduce the context $\mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\bullet, w)$ using (**cons.i, cons.ii, implies.i**), and permute the conses using (**cons.iv**) to obtain

$$
\begin{aligned}
& z = [z_a, z_d] \Rightarrow \\
& \quad \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{it.app}(z, w), w) \\
& \qquad \cong \mathtt{let}\{u := \mathtt{cons}(z_a, y)\} \\
& \qquad\qquad \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{setcdr}(w, u), \mathtt{it.app}(\mathtt{cdr}(z), u), w).
\end{aligned}
$$

Using (**delaying assignment**) we have

$$
\begin{aligned}
& \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{setcdr}(w, u), \mathtt{it.app}(\mathtt{cdr}(z), u), w) \\
& \quad \cong \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{it.app}(\mathtt{cdr}(z), u), \mathtt{setcdr}(w, u), w).
\end{aligned}
$$

Using (**ca.i**) we introduce $\mathtt{let}\{u := \mathtt{cons}(z_a, y)\}\bullet$, use (**cons.iv**) permute the conses, and use (**cons.ii, implies.ii, ca.ii**) to evaluate $\mathtt{cdr}(z)$ to z_d obtaining

$$
\begin{aligned}
& z = [z_a, z_d] \Rightarrow \\
& \quad \mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{it.app}(z, w), w) \\
& \qquad \cong \mathtt{let}\{w := \mathtt{cons}(x_a, y)\} \\
& \qquad\qquad \mathtt{let}\{u := \mathtt{cons}(z_a, y)\}\mathtt{seq}(\mathtt{it.app}(z_d, u), \mathtt{setcdr}(w, u), w).
\end{aligned}
$$

Rearranging and applying the induction hypothesis we have

$$
\begin{aligned}
& z = [z_a, z_d] \wedge w = [x_a, y] \wedge \mathtt{cdr}(z) \in Z \Rightarrow \\
& \mathtt{let}\{u := \mathtt{cons}(z_a, y)\}\mathtt{seq}(\mathtt{it.app}(z_d, u), \mathtt{setcdr}(w, u), w) \\
& \quad \cong \mathtt{let}\{u := \mathtt{cons}(z_a, y)\}\mathtt{seq}(\mathtt{setcdr}(w, \mathtt{seq}(\mathtt{it.app}(z_d, u), u)), w) \\
& \qquad \text{by } (\mathtt{let}_c\textbf{.ii}) \\
& \quad \cong \mathtt{seq}(\mathtt{setcdr}(w, \mathtt{let}\{u := \mathtt{cons}(z_a, y)\}\mathtt{seq}(\mathtt{it.app}(z_d, u), u)), w) \\
& \qquad \text{by } (\mathtt{let}_c\textbf{.iii}) \\
& \quad \cong \mathtt{seq}(\mathtt{setcdr}(w, \mathtt{cons}(z_a, \mathtt{append}(z_d, y))), w) \\
& \qquad \text{by } (\textbf{implies.ii}) \text{ and applying the induction hypothesis using } (\textbf{class.ca}) \\
& \quad \cong \mathtt{let}\{w_d := \mathtt{append}(z, y)\}\mathtt{seq}(\mathtt{setcdr}(w, w_d), w) \\
& \qquad \text{by } (\mathtt{let}_c\textbf{.iii, A, implies.ii})
\end{aligned}
$$

Finally, by (**cons.i. cons.ii, implies.i**), and permuting conses using (**cons.iv**) we have

$$
\begin{aligned}
& z = [z_a, z_d] \wedge \mathtt{cdr}(z) \in Z \Rightarrow \\
& \mathtt{let}\{w := \mathtt{cons}(x_a, y)\} \\
& \quad \mathtt{let}\{u := \mathtt{cons}(z_a, y)\}\mathtt{seq}(\mathtt{it.app}(z_d, u), \mathtt{setcdr}(w, u), w) \\
& \quad \cong \mathtt{let}\{w_d := \mathtt{append}(z, y)\}\mathtt{let}\{w := \mathtt{cons}(x_a, y)\}\mathtt{seq}(\mathtt{setcdr}(w, w_d), w) \\
& \quad \cong \mathtt{let}\{w_d := \mathtt{append}(z, y)\}\mathtt{cons}(x_a, w_d) \qquad \text{by } (\textbf{setcdr.vii, ca.i, ca.ii}) \\
& \quad \cong \mathtt{cons}(x_a, \mathtt{append}(z, y)) \qquad \text{by } (\mathtt{let}_c\textbf{.iii})
\end{aligned}
$$

□

$\square_{\mathbf{B}}\ \square_{\mathbf{append}}$

7. Issues and Conclusions

In this paper we have presented a logic, VTLoE, for 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 `mk`, `set`, `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 allows us to express a wide variety of useful notions including presence of effects, functionality, and structural induction principles.

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

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

This principle is false even for quantifier free Φ. One counterexample is:

$$e_0 = \lambda y.y$$
$$e_1 = \mathtt{let}\{z := \mathtt{mk}(\lambda y.y)\}\lambda w.\mathsf{app}(\mathtt{get}(z), w)$$
$$\Phi = (x \cong \lambda y.y)$$

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 (although the plethora of choices is somewhat confusing).

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

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

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 allowable trapped variables. It may also be productive to examine both the system and the results obtained concerning L4 in the paper (German et al., 1989).

(C) It may be productive to study contextual assertions in the simpler simply typed λ calculus making use of Milner's context lemma.

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.

(D) 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.

(E) 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

The authors wish to thank Matthias Felleisen, John Greiner, Takayasu Ito, Arthur Lent and the two anonymous referees for there thoughtful comments, corrections and criticisms of an earlier draft of this paper. We also thank Carl Gunter for pointing out (Meyer and Sieber, 1988) to us when he visited Stanford in 1992. Finally we would also like to aknowledge a debt to Solomon Feferman.

This research was partially supported by DARPA contract NAG2-703, NSF grants CCR-8917606, CCR-8915663, and CCR-9109070, and Italian grant CNR-Stanford N.89.00002.26.

8. References

Abadi, M., Pierce, B., and Plotkin, G. (1991). Faithful ideal models for recursive polymorphic types. *International Journal of Foundations of Computer Science*, 2(1):1–21.

Abramsky, S. (1990). The lazy lambda calculus. In Turner, D., editor, *Research Topics in Functional Programming*. Addison-Wesley.

Abramsky, S. (1991). Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1):1–77.

Agha, G., Mason, I. A., Smith, S. F., and Talcott, C. L. (1993). A foundation for actor computation. submitted to the journal of functional programming.

Apt, K. (1981). Ten years of Hoare's logic: A survey–part I. *ACM Transactions on Programming Languages and Systems*, 4:431–483.

Bloom, B. (1990). Can LCF be topped? *Information and Computation*, 87:264–301.

Crank, E. and Felleisen, M. (1991). Parameter-passing and the lambda-calculus. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 233–245.

Damas, L. (1985). *Type Assignment in Programming Languages.* PhD thesis, Edinburgh University.

Egidi, L., Honsell, F., and Ronchi della Rocca, S. (1992). Operational, denotational and logical descriptions: a case study. *Fundamenta Informaticae*, 16(2):149–170.

Feferman, S. (1975). A language and axioms for explicit mathematics. In *Algebra and Logic*, volume 450 of *Springer Lecture Notes in Mathematics*, pages 87–139. Springer Verlag.

Feferman, S. (1979). Constructive theories of functions and classes. In *Logic Colloquium '78*, pages 159–224. North-Holland.

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

Feferman, S. (1990). 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.

Felleisen, M. (1987). *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University.

Felleisen, M. (1993). Personal communication.

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

Felleisen, M. and Friedman, D. P. (1989). A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287.

Felleisen, M. and Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271.

Felleisen, M. and Wright, A. K. (1991). A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Rice University Computer Science Department.

Fields, J. and Sabry, A. (1993). Reasoning about explicit and implicit representations of state. In *ACM Sigplan Workshop on State in Programming Languages.* YaleU/DCS/RR-968.

German, S., Clarke, E., and Halpern, J. (1989). Reasoning about procedures as parameters in the language L4. *Information and Computation*, 83(3).

Halpern, J., Meyer, A. R., and Trakhtenbrot, B. A. (1983). The semantics of local storage, or what makes the free-list free? In *11th ACM Symposium on Principles of Programming Languages*, pages 245–257.

Halpern, J., Meyer, A. R., and Trakhtenbrot, B. A. (1984). From denotational to operational and axiomatic semantics for ALGOL-like languages: An overview. In Clarke, E. and Kozen, D., editors, *Logics of Programs, Proceedings 1983*, volume 164 of *Lecture Notes in Computer Science.* Springer, Berlin.

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

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

Howe, D. (1989). Equality in the lazy lambda calculus. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE.

Jim, T. and Meyer, A. (1991). Full abstraction and the context lemma. In *Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science volume 526. Springer-Verlag.

Jouvelot, P. and Gifford, D. K. (1991). Algebraic reconstruction of types and effects. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310.

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

Leroy, X. and Wies, P. (1990). Polymorphic type inference and assignment. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages*, pages 291–302. ACM.

Lucassen, J. M. (1987). *Types and Effects, towards the integration of functional and imperative programming.* PhD thesis, MIT. Also available as LCS TR-408.

Lucassen, J. M. and Gifford, D. K. (1988). Polymorphic effect systems. In *16th annual ACM Symposium on Principles of Programming Languages*, pages 47–57.

Manna, Z. and Waldinger, R. (1981). Problematic features of programming languages. *Acta Informatica*, 16:371–426.

Mason, I. A. (1986a). Equivalence of first order Lisp programs: Proving properties of destructive programs via transformation. In *First Annual Symposium on Logic in Computer Science.* IEEE.

Mason, I. A. (1986b). *The Semantics of Destructive Lisp.* PhD thesis, Stanford University. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

Mason, I. A. (1988). Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10:177–210.

Mason, I. A. and Talcott, C. L. (1989). Axiomatizing operational equivalence in the presence of side effects. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE.

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

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

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

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

Meyer, A. R. and Sieber, K. (1988). Towards fully abstract semantics for local variables: Preliminary report. In *15th ACM Symposium on Principles of Programming Languages*, pages 191–208.

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

Moggi, E. (1989). Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE.

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

Morris, J. H. (1968). *Lambda calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology.

O'Hearn, P. and Tennent, R. (1992). Semantics of Local Variables. Technical Report ECS-LFCS-92-192, Laboratory for foundations of computer science, University of Edinburgh.

O'Hearn, P. and Tennent, R. (1993a). Relational Parametricity and Local Variables. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 171–184.

O'Hearn, P. and Tennent, R. (1993b). Semantic Analysis of Specification Logic, Part 2. *Information and Computation*, ?:?–?

Olderog, E. (1984). Hoare's logic for programs with procedures–what has been accomplished. In Clarke, E. and Kozen, D., editors, *Logics of Programs, Proceedings 1983*, volume 164 of *Lecture Notes in Computer Science*. Springer, Berlin.

Ong, C.-H. (1988). *The Lazy Lambda Calculus: An investigation into the Foundations of Functional Programming*. PhD thesis, Imperial College, University of London.

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

Pitts, A. M. and Stark, I. (1993). On the observable properties of higher order functions that dynamically create local names. In *ACM Sigplan Workshop on State in Programming Languages*. YaleU/DCS/RR-968.

Plotkin, G. (1975). Call-by-name, call-by-value and the lambda-v-calculus. *Theoretical Computer Science*, 1:125–159.

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

Reynolds, J. C. (1978). Syntactic control of interference. In *Conference record of the 5th annual ACM Symposium on Principles of Programming Languages*, pages 39–46.

Sabry, A. and Felleisen, M. (1993). Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):287–358.

Sieber, K. (1985). A partial correctness logic for programs (in an Algol-like language). In Parikh, R., editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*. Springer, Berlin.

Smith, S. F. (1988). *Partial Objects in Type Theory*. PhD thesis, Cornell University. Available as TR 88-938.

Smith, S. F. (1992). From operational to denotational semantics. In *MFPS 1991*, volume 598 of *Lecture Notes in Computer Science*, pages 54–76. Springer-Verlag.

Talcott, C. (1991). A theory of binding structures and its applications to rewriting. *Theoretical Computer Science*, to appear.

Talcott, C. L. (1985). *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation*. PhD thesis, Stanford University.

Talcott, C. L. (1993). A theory for program and data specification. *Theoretical Computer Science*, 104:129–159.

Tofte, M. (1988). *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Edinburgh University.

Tofte, M. (1990). Type inference for polymorphic references. *Information and Computation*, 89:1–34.

Weeks, S. and Felleisen, M. (1993). On the orthogonality of assignments and procedures in Algol. In *Proceedings 20th ACM Symposium on Principles of Programming Languages*, pages 57–70.

9. Index of Notations

Symbol	Description	§
$\mathtt{Y}$	Fixed point combinators	1
$\mathbb{N}$	The natural numbers, $i, j, \ldots, n \in \mathbb{N}$	1
Y^n	Sequences from Y of length n, $\bar{y} = [y_1, \ldots, y_n] \in Y^n$	1
Y^*	Finite sequences from Y	1
$[]$	The empty sequence	1
$u * v$	The concatenation of the sequences u and v	1
$\mathbf{P}_\omega(Y)$	Finite subsets of Y	1
$Y_0 \xrightarrow{\mathrm{f}} Y_1$	Finite maps from Y_0 to Y_1	1
$Y_0 \to Y_1$	Total functions from Y_0 to Y_1	1
$\mathrm{Dom}(f)$	The domain of the function f	1
$\mathrm{Rng}(f)$	The range of the function f	1
$f\{y := y'\}$	An extension to, or alteration of, the function f	1
$\mathbb{X}$	A countably infinite set of variables, $x, y, z \in \mathbb{X}$	2.1
$\mathbb{A}$	Atoms	2.1
$\mathtt{t}, \mathtt{nil}$	Atoms playing the role of booleans	2.1
$\mathbb{F}$	Operations, $\delta \in \mathbb{F}$	2.1
$\mathbb{F}_n$	n-ary operation symbols	2.1
	$\mathbb{F}_1 \supseteq \{\mathtt{get}, \mathtt{mk}, \mathtt{isatom}, \mathtt{cell}, \mathtt{fst}, \mathtt{snd}, \mathtt{ispr}, \mathtt{isnat}, \mathtt{+1}, \mathtt{-1}\}$	2.1
	$\mathbb{F}_2 \supseteq \{\mathtt{set}, \mathtt{eq}, \mathtt{pr}\}$	2.1
	$\mathbb{F}_3 \supseteq \{\mathtt{br}\}$	2.1
$\mathbb{L}$	λ-abstractions, $\lambda x.e \in \mathbb{L}$	2.1
$\mathbb{V}$	Value expressions, $v \in \mathbb{V}$	2.1
$\mathbb{E}$	Expressions, $e \in \mathbb{E}$	2.1
$\mathbb{P}$	Immutable pairs, $\mathtt{pr}(v_0, v_2) \in \mathbb{P}$	2.1
$\mathbb{S}$	Value substitutions, $\sigma \in \mathbb{S}$	2.1
$\lambda x.e$	Abstractions	2.1
$\mathtt{app}(e_0, e_1)$	Application	2.1
$\delta(\bar{e})$	Application of operations	2.1
$\mathtt{if}(e_0, e_1, e_2)$	Conditional branching	2.1
$\mathtt{let}\{x := e_0\}e_1$	Lexical variable binding	2.1
$\mathtt{seq}(e_1, \ldots, e_n)$	Sequencing construct	2.1
$\mathrm{FV}(e)$	The free variables of the expression e	2.1
$e^{\{x:=e'\}}$	The result of substituting e' for x in e	2.1
e^σ	The substituting $\sigma(x)$ for x in e, $\forall x \in \mathrm{Dom}(\sigma)$	2.1
Y_X	Elements of Y whose free variables are among X	2.1

Symbol	Description	§
$\mathbb{C}$	Contexts, $C \in \mathbb{C}$	2.1
$\bullet$	The hole in contexts	2.1
$C[e]$	The result of filling the holes in C with e	2.1
$\mathrm{Traps}(C)$	The variables which may be trapped	2.1
$\mathbb{E}_{\mathrm{r}}$	The set of redexes	2.2
$\mathbb{R}$	The set of reduction contexts, $R \in \mathbb{R}$	2.2
$\mathbb{M}$	The set of memory contexts, $\Gamma \in \mathbb{M}$	2.2
Γ^{σ}	Applying σ to each value in the range of Γ	2.2
$\mathbb{D}$	Computation descriptions, $\Gamma; e \in \mathbb{D}$	2.2
$\Gamma; v$	A value description	2.2
$\mapsto$	The single step reduction relation, $\mapsto \subset \mathbb{D} \times \mathbb{D}$	2.2
$\overset{*}{\mapsto}$	The reduction relation, $\overset{*}{\mapsto} \subset \mathbb{D} \times \mathbb{D}$	2.2
$\downarrow \Gamma; e$	A defined description	2.2
$\downarrow e$	A defined expression	2.2
$e_0 \updownarrow e_1$	Equidefined expressions	2.2
$e_0 \cong e_1$	Operational equivalence between expressions	2.3
$e_0 \cong^{ciu} e_1$	By (**ciu**) operational equivalence	2.3.3
$\mathbb{U}$	Univalent contexts, $U \in \mathbb{U}$	3.1
$\mathbb{W}$	Formulas, $\Phi \in \mathbb{W}$	3.1
$e_0 \cong e_1$	Atomic formula of $\mathbb{W}$	3.1
$\Phi_0 \Rightarrow \Phi_1$	Implications of $\mathbb{W}$	3.1
$U[\![\Phi]\!]$	Contextual assertions of $\mathbb{W}$	3.1
$(\forall x)\Phi$	Universal generalizations of $\mathbb{W}$	3.1
`False`	An unsatisfiable assertion, such as $\mathtt{t} \cong \mathtt{nil}$, of $\mathbb{W}$	3.1
$\neg\Phi$	Negations of $\mathbb{W}$	3.1
$\Downarrow e$	Definedness assertions of $\mathbb{W}$	3.1
$\Phi_0 \wedge \Phi_1$	Conjunctions of $\mathbb{W}$	3.1
$\Phi_0 \vee \Phi_1$	Disjunctions of $\mathbb{W}$	3.1
$\Phi_0 \Leftrightarrow \Phi_1$	Biconditionals of $\mathbb{W}$	3.1
$\Gamma; U[\![\sigma]\!] \overset{*}{\mapsto} \Gamma'; U'[\![\sigma']\!]$	Context Reduction	3.2
$\Gamma \models \Phi[\sigma]$	Tarskian satisfaction	3.3
$\models \Phi$	Validity	3.3
$\Box\Phi$	A modality of $\mathbb{W}$	3.4
$\boxdot\Phi$	A modality of $\mathbb{W}$	3.4
`len`	Length function	3.4
`alloc`	Allocation function	3.4
`assign`	Assignment function	3.4
$(\forall_{\mathrm{s}} x)\Phi$	Strong quantification of $\mathbb{W}$	3.4
$\Phi_{\text{0-cell}}$	Expresses that there are no cells in memory	3.5.2
$\Phi_{\text{1-cell}}$	Expresses that there is one cell in memory	3.5.2

Symbol	Description	§
$\mathbb{W}_s$	The strong quantifier fragment	3.10
P, Q	Parameterless proceedures	4
$\perp$	A diverging proceedure	4
begin new $x; \ldots$ **end**	Local variable declarations	4
contents	Dereferencing	4
$x := v$	Assignment	4
$x =_{\mathrm{fo}} y$	First order equivalence	4
`isfun`	Recongnizer of abstractions	4.5
`foeq`	Recognizer of first order equivalence	4.5
$\mathbb{X}^c$	Class variables, $A, B, \ldots X, Y, Z \in \mathbb{X}^c$	5.1
$\mathbb{A}^c$	Class constants, letters in this **Font** $\in \mathbb{A}^c$	5.1
$\{x \mid \Phi\}$	Class comprehension terms	5.1
$\mathbb{K}$	Class terms, $K \in \mathbb{K}$	5.1
$\mathbb{W}$	Formulas extended to classes	5.1
$e \in K$	Atomic formula of $\mathbb{W}$	5.1
$(\forall X)\Phi$	Class quantification	5.1
$[K]^{\sigma}_{\Gamma}$	Evaluation of class terms	5.2
$\Gamma \models \Phi[\sigma]$	Satisfaction extended to classes	5.2
$K_0 \subseteq K_1$	Class containment	5.3
$K_0 \equiv K_1$	Class equivalence	5.3
$\emptyset$	The empty class	5.3
Val	The class of values	5.3
Nil	The singleton class containg `nil`	5.3
Bool	The class of booleans	5.3
Cell	The class of cells	5.3
Nat	The class of natural numbers	5.3
$T[K]$	Class operators	5.3
Cell$[Z]$	The class of cells containing elements of Z	5.3
$\rightarrow$	The class of total functions	5.3
$\xrightarrow{\mathrm{p}}$	The class of partial functions	5.3
$\xrightarrow{\mu}$	The class of memory operations	5.3
$\Phi_{\neg\mathrm{expand}}(e)$	e does not expand memory	5.3
$X, \Phi_0 \xrightarrow{\mu}_S Y, \Phi_1$	A more complicated function space	5.4
$\Phi_{\neg\mathrm{write}}[S](e)$	e does not visibly write in the region S	5.4
$\mathbb{T}$	Monomorphic ML-like types	5.5.2
$\mathbb{N}$	The type of natural numbers	5.5.2
$\{\mathtt{nil}\}$	The type of `nil`	5.5.2
$(\mathbb{T} \xrightarrow{\lambda} \mathbb{T})$	The type of the function space	5.5.2
$\mathbf{ref}(\mathbb{T})$	The type of references	5.5.2
$\{x_i : \tau_i \mid i < n\} \vdash e : \tau$	The typing judgement	5.5.2

Symbol	Description	§
$\texttt{mk}_\tau$	Reference allocation	5.5.3
$\texttt{get}_\tau$	Reference dereferencing	5.5.3
$\texttt{set}_\tau$	Reference assignment	5.5.3
$\Phi_{:}(e, \underline{\tau})$	An encoding of the typing judgement	5.5.3
$\bigcap_X \Phi[X]$	The intersection of a family of classes	6.1
$T^\cap$	The least fixed point of a class operator	6.1
`cons`	Lisp cell allocation	6.2
`cons?`	Lisp cell recognition	6.2
`car`	Lisp cell car destructuring	6.2
`cdr`	Lisp cell cdr destructuring	6.2
$\mathbf{Cons}[X, Y]$	Lisp cells with `cars` in X, `cdrs` in Y	6.2
`setcar`	Lisp cell car assignment	6.2
`setcdr`	Lisp cell cdr assignment	6.2
`loc`	Lisp path dereferencing	6.3
$\mathbf{Sexp}[A]$	Lisp S-expressions with leaves from A	6.3
`iterate`	Lisp list iteration	6.3
`len`	Lisp list length	6.3
`append`	Lisp list append operation	6.4
$\texttt{iterative.append}(x, y)$	Iterative Lisp list append operation	6.4

Contents