A first order logic of effects

Ian A. Mason*

Department of Mathematics and Computer Science, University of New England,
Armidale 2351, N.S.W, Australia

Abstract

In this paper we describe some of our progress towards an operational implementation of a
modern programming logic. The logic is inspired by the variable type systems of Feferman,
and is designed for reasoning about imperative functional programs. The logic goes well beyond
traditional programming logics, such as Hoare’s logic and Dynamic logic in its expressibility,
yet is less problematic to encode into higher order logics. The main focus of the paper is to
present an axiomatization of the base first order theory.

Keywords: Formal methods; λ-calculus; Contexts; Operational semantics; Theorem proving

1. Introduction

VTLoE [37, 24, 38, 40, 25] is a logic for reasoning about imperative functional pro-
grams, 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 [12, 13] and later applied to the study of purely functional
languages [14, 15]. VTLoE builds upon recent advances in the semantics of languages
with effects [16, 19, 30, 35, 36] and goes well beyond traditional programming logics,
such as Hoare’s logic [6] and Dynamic logic [23] by treating a richer language and
expressing more properties. It is close in spirit to Specification Logic [53] and to
Evaluation Logic [48].

The underlying programming language of VTLoE, λ_{mk}, is based on the call-by-
value lambda calculus extended by the reference primitives mk, set, get. Atoms,
references and lambda abstractions are all first class values – they can be bound to
lambda variables, stored, and returned from procedures. It can thus be thought of as
a fragment of untyped ML or a variant of Scheme. The logic combines the features
and benefits of equational calculi as well as program and specification logics. There
are three layers. The foundation is the syntax and semantics of λ_{mk}, the underlying
term/programming language. The second layer is a first-order theory built on assertions of program equivalence and program modalities called contextual assertions. The third layer extends the logic to include class terms, class membership, and quantification over class variables. In this paper we concentrate on the first two layers. The main topic of this paper is the presentation of a Hilbert style formal system for the first order fragment of VTLoE and the proof of its completeness.

Contextual assertions were first introduced in [34] as a means for expressing constraint propagation. It was quickly realized that they are an essential feature of any language for reasoning about the effects of programs. In our earlier work on axiomatizing imperative features [33, 32, 36] we presented a simple sequent system for proving equations. The introduction rules for the allocation and updating primitives were complicated by ugly side conditions, conditions so ugly as to make the implementation of the system either unpalatable or unfeasible. The main result of this paper is to generalize these previous results to a full first order language with contextual assertions. The crucial point is that contextual assertions eliminate the need for any side conditions.

1.1. Overview and notation

The remainder of this paper is organized as follows. In Section 2 we introduce the syntax and semantics of the terms of VTLoE. In Section 3 we introduce the syntax and semantics of the formulas of VTLoE. In Section 4 we present the proof theory of VTLoE. In Section 5 we relate the semantics with the proof theory.

We conclude the introduction with a summary of notation. Let X, Y, Y0, Y1 be sets. We specify meta-variable conventions in the form: x ranges over X, which should be read as: the meta-variable x and decorated variants such as x', x0, ..., range over the set X. We use the usual notation for set membership and function application. Yn is the set of sequences of elements of Y of length n. Y* is the set of finite sequences of elements of Y. v = [y1, ..., yn] is the sequence of length n with ith element yi. Po(Y) is the set of finite subsets of Y. Y0 Y 1 is the set of finite maps from Y0 to Y1. [Y0 Y 1] is the set of total functions f with domain Y0 and range contained in Y1. We write Dom(f) for the domain of a function and Rng(f) for its range. For any function f, f{y := y'} is the function f' such that Dom(f') = Dom(f) {y}, f'(y) = y', and f'(z) = f(z) for z # y, z E Dom(f). N = {0, 1, 2, ...} is the natural numbers and i,j,n,n0,... range over N.

2. The syntax and semantics of terms

2.1. Syntax

The syntax of the terms of j_\text{nat} is a simple extension of the lambda calculus to include basic atomic data A, (such as the Lisp booleans t and nil, and in practice the natural numbers \mathbb{N}), together with a collection of primitive operations, F = \bigcup_{n \in \mathbb{N}} F_n,
where \( F_n \) is the (possibly empty) set of \( n \)-ary operations.

Booleans \( \{t, \text{nil}\} \subseteq \mathbb{A} \)

Recognizers \( \mathbb{T} = \{\text{atom?}, \text{cell?}, \text{lambda?}\} \)

Unary operations \( F_1 = \{\text{mk}, \text{get}\} \cup \mathbb{T} \)

Binary operations \( F_2 = \{\text{app}, \text{eq}, \text{set}\} \)

Ternary operations \( F_3 = \{\text{br}\} \)

The primitive operations include: the memory operations (\( \text{mk}, \text{get}, \text{set} \)) for allocating, dereferencing, and updating unary cells; the usual operations for strict branching (\( \text{br} \)); the recognizing operations (\( \text{atom?}, \text{cell?}, \text{lambda?} \)) (or characteristic functions using the booleans \( t \) and \( \text{nil} \)) of their respective domains. We also treat application, \( \text{app} \), as a binary operation for the sake of uniformity.

Together with the atoms, \( \mathbb{A} \), we assume an infinite set of variables, \( \mathbb{X} \) and use these to define, by mutual induction, the set of \( \lambda \)-abstractions, \( \mathbb{L} \), the set of value expressions, \( \mathbb{V} \), the set of value substitutions, \( \mathbb{S} \), the set of expressions, \( \mathbb{E} \), and the set of contexts, \( \mathbb{C} \), as the least sets satisfying the following equations:

- Atoms \( \mathbb{A} \)
- Variables \( \mathbb{X} \)
- Lambda expressions \( \mathbb{L} = \lambda \mathbb{X}. \mathbb{E} \)
- Value expressions \( \mathbb{V} = \mathbb{X} + \mathbb{A} + \mathbb{L} \)
- Value substitutions \( \mathbb{S} = \{ \mathbb{X} \mapsto \mathbb{V} \} \)
- Expressions \( \mathbb{E} = \mathbb{V} + F_n(\mathbb{E}') \)
- Contexts \( \mathbb{C} = \{\bullet\} + \mathbb{X} + \mathbb{A} + \lambda \mathbb{X}. \mathbb{C} + F_n(\mathbb{C}') \)

\( \lambda \) is a binding operator and free and bound variables of expressions are defined as usual. \( \text{FV}(e) \) is the set of free variables of \( e \). A value substitution is a finite map \( \sigma \) from variables to value expressions, we let \( \sigma \) range over value substitutions. \( e^\sigma \) is the result of simultaneous substitution of free occurrences of \( x \in \text{Dom}(\sigma) \) in \( e \) by \( \sigma(x) \). We represent the function which maps \( x \) to \( v \) by \( \{x := v\} \). Thus \( e^{\{x := v\}} \) is the result of replacing free occurrences of \( x \) in \( e \) by \( v \) (avoiding the capture of free variables in \( v \)). Contexts are expressions with holes. We use \( \bullet \) to denote a hole. \( C[e] \) denotes the result of replacing any holes in \( C \) by \( e \). Free variables of \( e \) may become bound in this process. \( \text{Traps}(C) \) is the set of variables that can actually be trapped in the process of filling the holes in \( C \).

### 2.2. Informal semantics

We give a brief and informal guide to the more novel of the primitive operations:

\( \text{mk} \) is a memory allocation primitive: the evaluation of \( \text{mk}(v) \) results in the allocation of a new memory cell and initializes this cell so that it contains the value \( v \). The value returned by this cell to \( \text{mk} \) is the newly allocated cell. \( \text{mk} \) is total.
get is the memory access primitive: the evaluation of get(v) is defined iff v is a memory cell. If v is a memory cell, then get(v) returns the value stored in that cell. Note that there is no reason why a cell cannot store itself (or some more elaborate cycle). get is partial.

set is the memory modification primitive: the evaluation of set(v₀, v₁) is defined iff v₀ is a memory cell. If v₀ is a memory cell, then set(v₀, v₁) modifies that cell so that its new contents becomes v₁. The value returned by a call to set is somewhat arbitrary and somewhat irrelevant. In VTLoE we have chosen nil as the return value, thus if v is a cell, then set(v, v) will return nil, and more importantly modify v so that it contains itself. set is partial.

br is the strict branching primitive: the evaluation of br(v₀, v₁, v₂) returns v₂ if v₀ is the atom nil, otherwise it returns v₁. Thus any non-nil value is considered true. The usual lazy branching primitive if(e₀, e₁, e₂) is simply app(br(e₀, λz.e₁, λz.e₂), nil) for a fresh variable (see Section 2.3). br is total.

eq is the equality primitive (solely on atoms): eq(v₀, v₁) returns t if v₀ and v₁ are the same atom, otherwise it returns nil. eq is total.

2.3. Abbreviations

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

\[ f(x) \triangleq \text{app}(f, x) \]
\[ \text{let}\{x := e₀\}e₁ \triangleq \text{app}(\lambda x.e₁, e₀) \]
\[ \text{seq}(e) \triangleq e \]
\[ \text{seq}(e₀, \ldots, eₙ) \triangleq \text{let}\{z := e₀\}\text{seq}(e₁, \ldots, eₙ) \quad z \not\in \text{FV}(eᵢ) \text{ for } i < n \]
\[ \text{if}(e₀, e₁, e₂) \triangleq \text{app}(\text{br}(e₀, λz.e₁, λz.e₂), \text{nil}) \text{ for } z \not\in \text{FV}(eᵢ) \text{ for } i < 2 \]
\[ \text{not}(e) \triangleq \text{if}(e, \text{nil}, t) \]
\[ \text{and}(e₀, e₁) \triangleq \text{if}(e₀, e₁, \text{nil}) \]
\[ \text{and}(e₀, e₁, \ldots, eₙ) \triangleq \text{and}(e₀, \text{and}(e₁, \ldots, eₙ)) \]
\[ \text{or}(e₀, e₁) \triangleq \text{if}(e₀, \text{t}, e₁) \]
\[ \text{or}(e₀, e₁, \ldots, eₙ) \triangleq \text{or}(e₀, \text{or}(e₁, \ldots, eₙ)) \]

2.4. Programming examples

2.4.1. Equality on cells

To simplify matters later we have omitted equality on cells as a primitive operation. It is however easily definable [30]. To determine whether or not two cells are distinct, we first store their current contents in a safe place. We then place nil in the first cell, and then t in the second cell. The cells are distinct if and only if the first cell still contains nil. We then restore the world to the way it was prior to our
inquisition.

\[
\text{eq} \triangleq \lambda x. \lambda y. \text{if} \ (\text{and(cell?(x), cell?(y)))},
\]
\[
\text{let}\{(x_0 := \text{get}(x), y_0 := \text{get}(y)}\}
\]
\[
\text{seq(set}(x, \text{nil}),
\text{set}(y, t),
\text{let}\{z := \text{get}(x)}\}
\]
\[
\text{seq(set}(x, x_0),
\text{set}(y, y_0),
\text{z)\},
\text{nil})
\]

2.4.2. Landin's recursion operator

Since the $\lambda_{mk}$-calculus extends the call-by-value $\lambda$-calculus the usual call-by-value fixed point combinator is a term in the language. A somewhat different fixed point combinator, that makes use of the reference primitives, is possible:

\[
Y \triangleq \lambda y. \text{let} \{z := \text{mk(ni1)}\}
\]
\[
\text{seq(set}(z, \lambda x. \text{app}(\text{app}(y, \text{get}(z)), x)),
\text{get}(z))
\]

This version of the fixed point combinator is essentially identical to the one suggested by Landin [29]. When applied to a functional $F$ of the form $\lambda f. \lambda x. e$, $Y$ creates a private local cell, $z$, with contents $G = \lambda x. \text{app}(\text{app}(F, \text{get}(z)), x)$, and returns $G$. By privacy of $z$, $G$ is equivalent to $F(G)$ (cf. [35]). Note that this example is typable in the simply typed lambda calculus (for provably non-empty types (cf. [25])). Thus adding operations for manipulating references to the simply typed lambda calculus causes the failure of strong normalization as well as many other of its nice mathematical properties.

2.4.3. Integer streams

From an abstract point of view, a stream is simply a (possibly infinite) sequence of data [1]. In the $\lambda_{mk}$-calculus we can represent streams simply as functional objects. The sequence corresponding to a $\lambda_{mk}$-stream is the values returned by repeated application of the object to a fixed (and hopefully irrelevant) argument. The simplest example of a non-trivial $\lambda_{mk}$-stream is the stream of natural numbers.

\[
\text{makeStream} \triangleq \lambda m. \text{let}\{z := \text{mk}(m)\}
\]
\[
\lambda x. \text{let}\{n := \text{get}(z)\}
\]
\[
\text{seq(set}(z, n + 1),
n)\)
\]

Here makeStream applied to an integer $m$ creates a stream of integers beginning with that integer. The so-created stream when queried (applied to any value) returns the next integer in the stream.
2.4.4. The sieve of Eratosthenes

A somewhat more interesting example of a stream is the sieve of Eratosthenes [1].
We begin with the functional filter which expects an integer, \( n \), and a stream, \( s \) and
then creates a new stream. This new stream when queried repeatedly calls the stream
argument, \( s \), until an integer not divisible by the number argument, \( n \), is returned. This
number is then returned as the answer to the query.

\[
\text{filter} \triangleq \lambda n. \lambda s. \lambda x. \text{let} \{ m := s(\text{nil}) \} \text{ if } \text{divides?}(n, m) \text{ then filter}(n, s)(\text{nil}) \text{ else } m
\]

\text{sieve} is an expression which when evaluated creates a new sieve of Eratosthenes.
This new stream is a stream of the prime numbers. Each time the stream is queried it
returns the current prime and updates its local stream to filter with this prime.

\[
\text{sieve} \triangleq \text{let} \{ \text{sc} := \text{mk}(\text{makeStream}(2)) \} \lambda x. \text{let} \{ s := \text{get}(\text{sc}) \} \text{let} \{ p := s(\text{nil}) \} \text{ seq } \text{set}(\text{sc}, \text{filter}(p, s)),
\]

2.5. Semantics of terms

The operational semantics of expressions is given by a reduction relation \( \rightarrow \) 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 con-
tinuation, 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 \( \beta_c \)-reduction or the application of
a primitive operation to a sequence of value expressions. The set of redexes, \( \mathbb{E}_r \), is
defined as

\[
\mathbb{E}_r = \mathbb{F}_n(\forall^n)
\]

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 [50]
and were first introduced in [18]. The set of reduction contexts, \( \mathbb{R} \), is the subset of \( \mathbb{C} \)
defined by

\[
\mathbb{R} = \{ \bullet \} + \mathbb{F}_{m+n+1}(\forall^m, \mathbb{R}, \mathbb{E}^n)
\]

In the sequel \( R \) ranges over \( \mathbb{R} \). An expression is either a value expression or decom-
poses uniquely into a redex placed in a reduction context. An easy structural induction
establishes that if \( e \in \mathbb{E} \), then either \( e \in \forall \) or \( e \) can be written uniquely as \( R[e'] \) where
\( R \in \mathbb{R} \) and \( e' \in \mathbb{E}_r \). The set of memory contexts, \( \mathbb{M} \), is the set of contexts \( \Gamma \) of the form

\[
\text{let} \{ z_1 := \text{mk}(\text{nil}) \} \ldots \text{let} \{ z_n := \text{mk}(\text{nil}) \} \text{ seq } \text{set}(z_1, v_1), \ldots, \text{set}(z_n, v_n), \bullet
\]
where \( z_i \neq z_j \) when \( i \neq j \). We include the possibility that \( n = 0 \), in which case \( \Gamma = \cdot \). Subsequently \( \Gamma \) ranges over \( \mathcal{M} \).

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 \( \Gamma \) (as above) to be \( \text{Dom}(\Gamma) = \{ z_1, \ldots, z_n \} \), 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, \( \Gamma \{ z := \text{mk}(v) \} \), and the union of two memory contexts, \( (\Gamma_0 \cup \Gamma_1) \), in the obvious way. If \( \text{Dom}(\Gamma) \cap \text{Dom}(\sigma) = \emptyset \), then \( \Gamma^\sigma \) is the result of applying \( \sigma \) to each value in the range of \( \Gamma \).

The set of computation descriptions (briefly descriptions), \( \mathbb{D} \), is defined to be the set \( \mathcal{M} \times \mathbb{E} \). 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 \( \Gamma; e \) ranges over \( \mathbb{D} \). A closed description is a description of the form \( \Gamma; e \) where \( \Gamma[e] \) is closed. Value descriptions are descriptions whose expression component is a value expression, i.e. a description of the form \( \Gamma; v \). Substitution into descriptions is defined pointwise: \( (\Gamma; e)^\sigma = \Gamma^\sigma; e^\sigma \), provided \( \text{Dom}(\Gamma) \cap \text{Dom}(\sigma) = \emptyset \).

**Definition (\( \rightarrow^* \)).** The reduction relation \( \rightarrow^* \) is the reflexive transitive closure of \( \rightarrow \). The clauses are:

- \((\beta)\) \( \Gamma; R[\text{app}(\lambda x. e, v)] \rightarrow \Gamma; R[e^{(x := i)}] \)
- \((\text{atom})\) \( \Gamma; R[\text{atom?}(v)] \rightarrow \begin{cases} \Gamma; R[t] & \text{if } v \in \mathbb{A} \\ \Gamma; R[\text{nil}] & \text{if } v \in \mathbb{L} \cup \text{Dom}(\Gamma) \end{cases} \)
- \((\text{cell})\) \( \Gamma; R[\text{cell?}(v)] \rightarrow \begin{cases} \Gamma; R[t] & \text{if } v \in \text{Dom}(\Gamma) \\ \Gamma; R[\text{nil}] & \text{if } v \in \mathbb{L} \cup \mathbb{A} \end{cases} \)
- \((\text{eq})\) \( \Gamma; R[\text{eq}(v_0, v_1)] \rightarrow \begin{cases} \Gamma; R[t] & \text{if } v_0 = v_1, v_0, v_1 \in \mathbb{A} \\ \Gamma; R[\text{nil}] & \text{otherwise} & \text{(provided } v_0, v_1 \in \text{Dom}(\Gamma) \cup \mathbb{L} \cup \mathbb{A}) \end{cases} \)
- \((\text{br})\) \( \Gamma; R[\text{br}(v_0, v_1, v_2)] \rightarrow \begin{cases} \Gamma; R[v_1] & \text{if } v_0 \in (\mathbb{A} - \{ \text{nil} \}) \cup \mathbb{L} \cup \text{Dom}(\Gamma) \\ \Gamma; R[v_2] & \text{if } v = \text{nil} \end{cases} \)
- \((\text{mk})\) \( \Gamma; R[\text{mk}(v)] \rightarrow \Gamma \{ z := \text{mk}(v) \}; R[z] & \text{if } z \notin \text{Dom}(\Gamma) \cup \text{FV}(R[v]) \)
- \((\text{get})\) \( \Gamma; R[\text{get}(z)] \rightarrow \Gamma; R[v] & \text{if } z \in \text{Dom}(\Gamma) \text{ and } \Gamma(z) = v \)
- \((\text{set})\) \( \Gamma; R[\text{set}(z, v)] \rightarrow \Gamma \{ z := \text{mk}(v) \}; R[\text{nil}] & \text{if } z \notin \text{Dom}(\Gamma) \)

Note that reduction is not restricted to closed descriptions. However in the atom? and cell? rules if one of the arguments is a variable not in the domain of the memory.
context then there is no appropriate primitive reduction step. This is also the case in the eq, br, get, and set rules. As mentioned in Section 2.2 get and set are also undefined if their first (and in the case of get only) argument is not a cell, i.e. a variable in the domain of the memory context.

**Definition** (↓↑↓↓). A closed description, \( \Gamma; e \) is defined (written \( \downarrow \Gamma; e \)) if it evaluates to a value description. A description is undefined (written \( \uparrow \Gamma; e \)) if it is not defined. Two descriptions, \( \Gamma; e_0 \) and \( \Gamma; e_1 \) are equivalued (written \( \Gamma; e_0 \equiv \Gamma; e_1 \)) if they are both undefined or have a common reduct (i.e. they both reduce to a particular description)

\[
\downarrow(\Gamma; e) \iff \exists \Gamma'; \nu'(\Gamma; e \rightarrow \Gamma'; \nu')
\]

\[
\uparrow(\Gamma; e) \iff \neg \downarrow(\Gamma; e)
\]

\[
\Gamma; e_0 \equiv \Gamma; e_1 \iff (((\Gamma; e_0) \land \downarrow(\Gamma; e_1)) \lor ((\exists \Gamma'; e)(\Gamma; e_0 \rightarrow \Gamma'; e \land \Gamma; e_1 \rightarrow \Gamma'; e)))
\]

For closed expressions \( e \), we write \( \downarrow e \) to mean \( \downarrow \emptyset; e \), similarly we write \( e_0 \equiv e_1 \) to mean that \( e_0 \equiv e_1 \) and, finally \( e_0 \equiv e_1 \) to mean \( \emptyset; e_0 \equiv \emptyset; e_1 \).

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) (Mason and Talcott [35]).

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

(ii) \( R[\Gamma[e]] \rightarrow \Gamma; R[e] \) if \( \text{FV}(R) \cap \text{Dom}(\Gamma) = \emptyset \).

(iii) \( \Gamma; e \rightarrow \Gamma'; \nu' \Rightarrow \Gamma; e \rightarrow (\Gamma'; \nu') \)

\[ \text{if } \ \text{Dom}(\Gamma') \cap \text{Dom}(\sigma) - \emptyset = \text{FV}(\text{Rng}(\sigma)) \cap (\text{Dom}(\Gamma') - \text{Dom}(\Gamma)). \]

(iv) \( \Gamma; e \rightarrow \Gamma'; \nu' \Rightarrow (\Gamma_0 \cup \Gamma); e \rightarrow (\Gamma_0 \cup \Gamma'); \nu' \) if \( \text{Dom}(\Gamma') \cap \text{Dom}(\Gamma_0) = \emptyset \).

(v) \( \Gamma; R[e] \rightarrow \Gamma'; R[e'] \Rightarrow \Gamma; R'[e] \rightarrow \Gamma'; R'[e'] \) if \( e \notin \mathbb{V} \) and \( (\text{Dom}(\Gamma') \cap \text{FV}(\mathbb{K})) \subseteq \text{Dom}(\Gamma) \)

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. In (cr.v) the requirement that \( e \notin \mathbb{V} \) is necessary. Consider the following counterexample (due to Soeren Lassen). Let

\[
e = e' = \lambda x. \text{app}(x, x) \quad R_0 = \text{app}(e, \bullet) \quad R_1 = \bullet
\]

Then \( \Gamma; R_0[e] \rightarrow \Gamma'; R_0[e'] \) by \( \beta_v \), but it is not the case that \( \Gamma; R_1[e] \rightarrow \Gamma'; R_1[e'] \).

2.6. **Operational equivalence of terms**

In this section we define the operational equivalence relation and study its general properties. Operational 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 [44, 50] to computation over memory structures. As shown by [2, 3, 9, 11, 26, 30, 35, 27, 41, 46, 49, 54, 55] operational equivalence and approximation can be characterized in various ways.

Definition ($\equiv$). Two expressions are operationally equivalent, written $e_0 \equiv e_1$, if for any closing context $C$, $C[e_0]$ is defined iff $C[e_1]$ is defined.

$$e_0 \equiv e_1 \iff (\forall C \in \mathcal{C} \mid \text{FV}(C[e_0]) = \text{FV}(C[e_1]) = \emptyset)(C[e_0] \Downarrow C[e_1])$$

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

**Congruence**: $e_0 \equiv e_1 \Rightarrow (\forall C \in \mathcal{C})(C[e_0] \equiv C[e_1])$

However, it is not necessarily the case that substitution instances of equivalent expressions are equivalent even if the instantiating expressions always return a value. As a counter-example we have $\text{if}(\text{cell}?(x), \text{eq}(x,x), t) \equiv t$ but $\text{if}(\text{cell}?(\text{mk}(t)), \text{eq}(\text{mk}(t), \text{mk}(t)), t) \equiv \text{nil}$. 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

$$\text{mk}(x) \quad \text{if}(\text{cell}?(x), \text{set}(x,y), x) \quad \text{if}(\text{cell}?(x), \text{get}(x), x)$$

but none is equivalent to a value, i.e. for no value expression $v$ do we have $e \equiv v$ 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.

In general it is very difficult to establish the operational equivalence of expressions. Thus it is desirable to have a simpler characterization of $\equiv$, one that limits the class of contexts (or observations) that must be considered. The main context lemma in this case is the following

Theorem (ciu) (Mason and Talcott [35]). $c_0 \equiv c_1 \iff (\forall \Gamma, \sigma, R)(\text{FV}(\Gamma[R[e_0^\sigma]]) = \emptyset \Rightarrow (\Gamma[R[e_0^\sigma]] \Downarrow \Gamma[R[e_1^\sigma]])$)

A proof of (ciu) appears in [35], and in [25].

---

1 Since writing this paper a very general proof of (ciu) that applies to a very general class of programming languages has been found [59].
3. The syntax and semantics of formulas

3.1. Syntax

The first order fragment of our logic is a minor generalization of classical first order logic. The atomic formulas assert the equivalencedness and operational equivalence of expressions. In addition to the usual first-order formula constructions, we add a \texttt{let}-assertion: if $\Phi$ is a formula, $x$ a variable, and $e$ an expression then $\texttt{let}\{x := e\}[\Phi]$ is a formula.

\textbf{Definition} ($\mathcal{W}$).

$$\mathcal{W} = (E \equiv E) + (E \equiv E) + (W \Rightarrow W) + (\texttt{let}\{X := E\}[W]) + (\forall X)(\mathcal{W})$$

For typographical convenience we will let $L$ range over the class of \texttt{let} contexts. Thus $L$ denotes a generic member of $\texttt{let}\{X := E\}$. 

3.2. Semantics

The meaning of formulas is given by a Tarskian satisfaction relation $\Gamma \models \Phi[\sigma]$.

\textbf{Definition} ($\Gamma \models \Phi[\sigma]$). Assume $\Gamma, \sigma, \Phi, e_j$ satisfy $\text{FV}(\Phi') \cup \text{FV}(e_j) \subseteq \text{Dom}(\Gamma)$ for $j < 2$, and $\text{FV}(\Gamma) = \emptyset$. Then we define the satisfaction relation $\Gamma \models \Phi[\sigma]$ by induction on the structure of $\Phi$:

- $\Gamma \models (e_0 \sim e_1)[\sigma]$ iff $\Gamma; e_0 \equiv \Gamma; e_1$

- $\Gamma \models (e_0 \equiv e_1)[\sigma]$ iff $(\forall R \in R \mid \text{FV}(R) \subseteq \text{Dom}(\Gamma))(\Gamma[R[e_0]] \models \Gamma[R[e_1]])$

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

- $\Gamma \models \texttt{let}\{x := e\}[\Phi][\sigma]$ iff $\Gamma; e^\sigma; \Gamma', e_1$ implies $\Gamma' \models \Phi[\sigma[x := e_1]]$

- $\Gamma \models (\forall x)\Phi[\sigma]$ iff $\forall v \in V \mid \text{FV}(v) \subseteq \text{Dom}(\Gamma)(\Gamma \models \Phi[\sigma[x := v]])$

As is usual in logic we define the subsidiary notions of validity and logical consequence as follows:

$$\models \Phi \quad \text{iff} \quad (\forall \Gamma, \sigma \mid \text{FV}(\Phi') \subseteq \text{Dom}(\Gamma))(\Gamma \models \Phi[\sigma])$$

$$\Phi_0 \models \Phi_1 \quad \text{iff} \quad \models \Phi_0 \Rightarrow \Phi_1$$

The requirement in the definition of satisfaction that $\text{FV}(\Gamma) = \emptyset$ was pointed out to the author by Jacob Frost [20], it is necessary for the proof that $\models e_0 \equiv e_1$ and the meta-statement $e_0 \equiv e_1$ are equivalent.

3.3. Examples, counterexamples and caveats

Negation is definable, $\neg \Phi$ is just $\Phi \Rightarrow \text{False}$, where False is any unsatisfiable assertion, such as $t \equiv \text{nil}$. Similarly, conjunction, $\wedge$, and disjunction, $\vee$ and the biconditional, $\leftrightarrow$, are all definable in the usual manner. Termination and non-termination
are simple abbreviations, there is, however, a plethora of notions of definedness that
can be expressed. We let |e abbreviate \( \neg (\text{let}\{x := e\}[\text{False}]) \) and |e abbreviate its
negation \( \text{let}\{x := e\}[\text{False}] \). A stronger notion of definedness is that of being equivalent
(either via \( \sim \) or via \( \cong \)) to a value. To see that this notion differs recall that
\( \text{mk}(v) \) is always defined in the sense that \( \text{lmk}(o) \), yet \( \text{mk}(v) \) is never equivalent to a
value (either via \( \sim \) or via \( \cong \)). These two notions of definedness will be important
later. Note that the let-assertion is a binding operator akin to \( \forall \). A simple example is
the axiom which expresses the effects of \( \text{mk} \):

\[(\forall z)(\forall y)(\text{let}\{x := \text{mk}(z)\}[\neg (x \equiv y) \land \text{cell?}(x) \equiv t \land \text{get}(x) \equiv v])\]

We use the symbol \( \simeq \) to denote either of the binary relations in our logic, \( \cong \) and
\( \sim \). It is important to note that, unlike equality in first order logic, neither of these
binary relations (\( \cong \) nor \( \sim \)) is a congruence in the sense that \( e_0 \cong e_1 \Rightarrow C[e_0] \cong C[e_1] \)
is falsifiable (even when no trapping occurs). For example,
\[
\text{get}(x) \simeq t \Rightarrow \text{app}(\text{seq}((x, \text{nil}), \lambda z. z), \text{get}(x)) \simeq \text{app}(\text{seq}((x, \text{nil}), \lambda z. z), t)
\]
is obviously not valid. Similarly, false is the related principle that \( \Phi \Rightarrow \text{let}\{x := e\}[\Phi] \).

For example,
\[
\text{get}(x) \simeq t \Rightarrow \text{let}\{x := \text{set}(x, \text{nil})\}[\text{get}(x) \simeq t]
\]
is clearly not valid. Also along these lines is the observation that while
\[
\text{let}\{x := e\}[e_0 \cong e_1] \Rightarrow \text{let}\{x := e\}[e_0] \cong \text{let}\{x := e\}[e_1]
\]
is valid, its converse is false, since
\[
\text{let}\{x := \text{mk}(0)\}\text{let}\{y := \text{mk}(0)\}[x] \cong \text{let}\{x := \text{mk}(0)\}\text{let}\{y := \text{mk}(0)\}[y]
\]
is valid, but \( \text{let}\{x := \text{mk}(0)\}\text{let}\{y := \text{mk}(0)\}[x \cong y] \) is not.

3.3.1. Violation of privacy

Rather than give the impression that everything is rosy, we point out the following
problem raised in [40]. One seemingly desirable logical principle for contextual rea-
soning is to be able to replace the \( e \) by any operationally equivalent expression without
changing the semantics of the contextual assertion \( \text{let}\{x := e\}[\Phi] \). In other words the
following principle seems desirable:
\[
e_0 \cong e_1 \Rightarrow (\text{let}\{x := e_0\}[\Phi] \Leftrightarrow \text{let}\{x := e_1\}[\Phi])
\]
However, there are several ways in which this can fail in this logic. For example, \( e_0 \)
may produce some garbage that \( e_1 \) does not, and this garbage may be detectable via
\( \Phi \). For example, letting
\[
e_0 = \text{seq}(\text{mk}(0), \text{mk}(0))
\]
\[
e_1 = \text{mk}(0)
\]
\[
\Phi = (\exists y_0)(\exists y_1)(\text{cell?}(y_0) \equiv t \land \text{cell?}(y_1) \equiv t \land \text{eq}(y_0, y_1) \equiv \text{nil})
\]
provides a counterexample.
Another more troublesome counterexample relies on the fact that $e_0$ and $e_1$ may be equivalent due to the privacy of certain cells, however their privacy is not respected by the contextual assertion. A simple example of this is:

$$e_0 = \lambda x_0 . x_0$$

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

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

A simple induction on the length of computations (similar to those found in [35]) establishes that $e_0$ and $e_1$ are operationally equivalent, and hence $e_0 \equiv e_1$ is valid. The essential observation is that the cell $z$ is local to the value/object returned by $e_1$ and thus invisible and its contents unalterable outside this scope. However it is not the case that

$$\models \text{let}\{x := e_0\}[\Phi] \iff \text{let}\{x := e_1\}[\Phi]$$

However all is not lost, we do have that the weaker principle

$$e_0 \sim e_1 \Rightarrow (\text{let}\{x := e_0\}[\Phi] \iff \text{let}\{x := e_1\}[\Phi])$$

is valid.

### 3.4. Extending the syntax of contextual assertions

For simplicity we have minimalized the syntax of contextual assertions to simple let statements. In earlier treatments [25] we dealt with a much wider class of contexts, 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, $U$, is defined as follows.

**Definition (U).**

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

The semantics is a simple generalization of the one presented here, and the curious are referred to [25] for details. The main reason for restricting our attention to let contexts, apart from the simplicity in presentation, is that those left out may be considered abbreviations:

1. $R[\Phi]$ abbreviates $\Phi$.
2. $\text{seq}(e_1, e_2, \ldots, e_n, [\Phi])$ abbreviates

$$\text{let}\{x_1 := e_1\} \ldots [\text{let}\{x_n := e_n\}[\Phi]]$$

provided $x_i$ are fresh.

3. $\text{let}\{x_1 := e_1, \ldots, x_n := e_n\}[\Phi]$ abbreviates

$$\text{let}\{x_1 := e_1\} \ldots [\text{let}\{x_n := e_n\}[\Phi]]$$

provided $x_i \not\in \text{FV}(e_j)$ for $1 \leq i < j \leq n$. 
(3) if(e₀,[Φ₀],[Φ₁]) abbreviates
\[\text{let}\{z := e₀\}(z \equiv \text{nil} \Rightarrow \Phi₁) \land (\neg(z \equiv \text{nil} \Rightarrow \Phi₀)) \quad z \text{ fresh.}\]

(4) \(\phi(e₀, \ldots, eₙ, U[\Phi], e_{n+1}, \ldots)\) abbreviates \(\text{seq}(e₀, \ldots, eₙ, U[\Phi])\)

That these abbreviations are in fact reasonable derives from Theorem (ca.iii) in [25] which states that (in this generalized semantics):

**Theorem (ca)** (Honsell et al. [25]).

(iii) \(U₀[U₁[\Phi]] \iff (U₀[U₁])[\Phi]\)

4. Proof theory

Since contextual assertions are akin to modalities, we give a Hilbert style presentation. In the long run a natural deduction style system in the style of Prawitz [51] may be more desirable.²

**Definition (\(\vdash\Phi\)).** The consequence relation, \(\vdash\), is the smallest relation on \(\mathcal{W}\) that is closed under the rules given below.

The rules are partitioned into several groups. Each group of rules is given a label, for future reference, and members of the group are numbered. For example, (E.i) refers to the first rule in the group of equivalence and evaluation rules (the second group below). A rule has a (possibly empty) set of premisses and a conclusion. In the case that the set of premisses is non-empty the rule is displayed with a horizontal bar separating the premisses from the conclusion.

**Variable convention:** We adopt Barendregt’s convention [8] that in any particular mathematical situation the bound and free variables in expressions are distinct. However we do (and must) allow free variables of expressions to coincide with bound (trappable) variables in contexts.

So for example we assume in: (E.vi) that \(x\) not free in \(R\); (E.vii) that \(x \not\in \text{FV}(e)\); (C.v) that \(x \not\in \text{FV}(\Phi)\); (Q.ii) that \(x \not\in \text{FV}(\Phi₀)\); (mk.ii) and (mk.iii) that \(x \not\in \text{FV}(\Phi₀)\); and in (S.i) that \(x \not\in \tilde{z}\). On the other hand in (mk.i) we must explicitly state that the variable \(x\) is distinct from the variables \(y\) and \(z\). This convention makes the statement of (Q.i) somewhat cumbersome.

Most axioms hold true for both equivaluedness, \(\sim\), and operational equivalence, \(\cong\), if this is the case, then rather than write out the principle twice, we use the symbol \(\cong\) to range over these two equivalence relations. One important reason for introducing \(\sim\) is that important principles fail for \(\cong\). In particular (C.iii) below fails as indicated in [40] and in Section 3.3.1.

² Since writing this paper Jacob Frost has developed a natural deduction presentation of this system [21] and implemented it in the proof assistant Isabelle.
4.1. Basic equivalence axioms and rules

The first, most basic axiom concerning operational equivalence and equivaluedness is that the booleans \( t \) and \( \text{nil} \) are not equivalent.

**Non-triviality (T).**

(i) \( \vdash \neg(t \simeq \text{nil}) \)

The second set of rules concerning equivaluedness hold true also of operational equivalence. They are equivalence relations, \((E.i, E.ii, E.iii)\). They satisfy a certain restricted form of substitutivity, \((E.iv)\). And are preserved under simple forms of evaluation, \((E.v, E.vi, E.vii)\), these last three principles are (equivalent to) the let-rules of the lambda-c calculus [42].

**Equivalence and evaluation rules (E).**

1. \( \vdash e_0 \simeq e_0 \)
2. \( \vdash (e_0 \simeq e_1 \land e_1 \simeq e_2) \Rightarrow e_0 \simeq e_2 \)
3. \( \vdash e_0 \simeq e_1 \Rightarrow e_1 \simeq e_0 \)
4. \( \vdash e_0 \simeq e_1 \Rightarrow e_0 \simeq e_1 \)

(ii) \( e_0 \simeq e_1 \Rightarrow \text{let}\{x := e_0\}e \simeq \text{let}\{x := e_1\}e \)

(vi) \( \text{app}(\lambda x.e, v) \simeq e^{[x := v]} \simeq \text{let}\{x := v\}e \)

(vii) \( \text{let}\{x := e_0\}\text{let}\{y := e_1\}e \simeq \text{let}\{y := \text{let}\{x := e_0\}e_1\}e \)

The remaining axioms rules concerning operational equivalence (other than that it is an equivalence relation) are: \((\simeq.i)\), equivaluedness implies operational equivalence; \((\simeq.ii)\), operational equivalence is preserved under the collection of garbage; \((\simeq.iii)\), operational equivalence is non-trivial on abstractions; and they agree with one another on atoms and cells, \((\simeq.iv)\).

**Operational equivalence rules (\(\simeq\)).**

1. \( \vdash e_0 \sim e_1 \Rightarrow e_0 \simeq e_1 \)
2. \( \vdash e \simeq F[e] \) provided \(\text{FV}(e) \cap \text{Dom}(F) = \emptyset\)
3. \( \vdash e_0 \simeq e_1 \quad \text{(The } \xi \text{ rule)} \)
4. \( \vdash \tau(x) \sim \tau \land \tau(y) \sim \tau \Rightarrow (x \simeq y \Leftrightarrow x \sim y) \tau \in \{\text{atom?}, \text{cell?}\} \)

4.2. Contextual axioms and rules

Contextual assertions are a modality and as such possess a rule akin to necessitation, \((C.i)\). Note that this is a rule of proof and not an implication. A simple counterexample to the implication can be found in Section 3.3. The remaining axioms concerning contextual assertions are: \((C.ii)\), contextual assertions distribute across the equivalences, again a counterexample to the converse can be found in Section 3.3; \((C.iii)\), a form of contextual assertion introduction involving equivaluedness (the corresponding principle for operational equivalence is false, Section 3.3.1); \((C.iv)\), a principle akin to \(\beta\) conversion; and \((C.v)\), a principle allowing for the manipulation of contexts.
Contextual rules (C).
(i) $\vdash \Phi$ (Context Introduction)
(ii) $L[e_0] \simeq e_1 \Rightarrow L[e_0] \simeq L[e_1]
(iii) $\vdash e_0 \simeq e_1 \Rightarrow (\mathsf{let}(x := e_0)[\Phi] \leftrightarrow \mathsf{let}(x := e_1)[\Phi])$
(iv) $\vdash \mathsf{let}(x := e_0)[\Phi] \leftrightarrow \mathsf{let}(x := e_1)[\Phi]
(v) $\vdash \mathsf{let}(x := e_0)[\Phi] \leftrightarrow \mathsf{let}(y := \mathsf{let}(x := e_0)[\Phi])$

4.3. Logical axioms and rules

The propositional rules are, in addition to the usual Hilbert style presentation of modus ponens, (P.iii), and a generating set of tautologies, (P.i) a modal axiom corresponding to $\mathbf{K}$ and its converse, (P.ii).

Propositional rules (P).
(i) $\vdash \Phi$ provided $\Phi$ is an instance of a tautology
(ii) $L[\Phi_0 \Rightarrow \Phi_1] \leftrightarrow (L[\Phi_0] \Rightarrow L[\Phi_1])$
(iii) $\vdash \Phi_0 \Rightarrow \Phi_1$ (Modus Ponens)

Similarly, the quantifier axioms are all standard [10] except for (Q.iv) and (Q.v) which assert that operations other than mk and app have no allocation effect, and that mk only allocates the value it returns.

Quantifier rules (Q).
(i) $\vdash (\forall x)\Phi \Rightarrow (\forall x)\Phi$ (Generalization $\forall$)
(ii) $\vdash (\forall x)\Phi_0 \Rightarrow \Phi_1 \Rightarrow (\forall x)\Phi_0 \Rightarrow (\forall x)\Phi_1$
(iii) $\vdash (\forall x)\Phi_0 \Rightarrow \Phi_1 [x := e_0]
(iv) $\vdash (\forall x)\mathsf{let}[z := \theta(y)][\Phi] \Rightarrow \mathsf{let}[z := \theta(y)][(\forall x)\Phi][\theta \in \mathbb{F} \setminus \{\mathsf{mk}, \mathsf{app}\}]
(v) $\vdash (\forall x)\mathsf{let}[z := \mathsf{mk}(y)][\Phi(x)] \land \mathsf{let}[z := \mathsf{mk}(y)][\Phi(z)]$

Note that the converses of these last two axioms are easily derivable.

4.4. Undefinedness principles

The most basic principle concerning undefinedness is that two undefined terms are both equivalued and operationally indistinguishable, (U.i). The rest of the principles concern the partiality of the underlying operation. Note that in the case of the memory operations mk and set, being defined is not the same as being equivalent to a value. In the other cases this is true, although we need only express the weaker form. The stronger forms are derivable.

Undefinedness rules (U).
(i) $\vdash \uparrow e_0 \Rightarrow \uparrow e_1 \Leftrightarrow e_0 \simeq e_1$
(ii) $\vdash \downarrow \mathsf{mk}(x)$
(iii) $\vdash \downarrow \mathsf{set}(z,x) \Leftrightarrow \mathsf{cell?(z)} \simeq \mathsf{t}$
(iv) $\vdash \downarrow \mathsf{get}(x) \Leftrightarrow \mathsf{cell?(x)} \simeq \mathsf{t} \Leftrightarrow (\exists y)(\mathsf{get}(x) \simeq y)$
(v) $\vdash \downarrow \theta(x)$ for $\theta \in \mathbb{T} \cup \{\mathsf{eq}, \mathsf{br}\}$
4.5. Data operation axioms and rules

We treat each operation in turn. We should point out, however, that we have grouped together a collection of principles that concern when an assertion propagates into or out of a context. They may be found after this collection, (S).

The principles concerning mk, other than the definedness principle above, (U.ii), are quite straightforward. (mk.i) describes the allocation effect of a call to mk. While (mk.ii) and (mk.iii) assert that the time of allocation has no discernable effect on the resulting call. In a world with control effects e₀ must be free of them for this principle to be valid [17].

**mk rules (mk).**

(i) \(\vdash \text{let}\{x := \text{mk}(z)\}[\neg(x \simeq y) \land \text{cell?(x)} \simeq t \land \text{get}(x) \simeq z] \quad x \text{ fresh}\)

(ii) \(\vdash \text{let}\{y := e₀\}\text{let}\{x := \text{mk}(z)\}e₁ \simeq \text{let}\{x := \text{mk}(z)\}\text{let}\{y := e₀\}e₁\)

(iii) \(\vdash \text{let}\{y := e₀\}\text{let}\{x := \text{mk}(z)\}\Phi \Leftrightarrow \text{let}\{x := \text{mk}(z)\}\text{let}\{y := e₀\}\Phi\)

The first two contextual assertions regarding set are analogous to those of (mk.i). They describe what is returned and what is altered, what is not altered. The remaining four principles involve the commuting, cancellation, absorption, and idempotence of calls to set. For example the set absorption principle, (set.v), expresses that under certain simple conditions allocation followed by assignment may be replaced by a suitably altered allocation.

**set rules (set).**

(i) \(\vdash \text{let}\{x := \text{set}(z, y)\}\text{let}\{z := \text{mk}(x)\}\text{get}(z) \simeq y \land x \simeq \text{nil}\)

(ii) \(\vdash (y \simeq \text{get}(z) \land \neg(w \simeq z)) \Rightarrow \text{let}\{x := \text{set}(w, w₀)\}[y \simeq \text{get}(z)]\)

(iii) \(\vdash \neg(x₀ \simeq x₂) \Rightarrow \text{seq}(\text{set}(x₀, x₁), \text{set}(x₂, x₃)) \simeq \text{seq}(\text{set}(x₁, x₂), \text{set}(x₀, x₃))\)

(iv) \(\vdash \text{seq}(\text{set}(z₀, x₀), \text{set}(x₀, y₀)) \simeq \text{set}(x₀, y₁)\)

(v) \(\vdash \text{let}\{z := \text{mk}(x)\}\text{seq}(\text{set}(z, w), e) \simeq \text{let}\{z := \text{mk}(w)\}e\)

(vi) \(\vdash \text{get}(x) \simeq y \Rightarrow \text{set}(x, y) \simeq \text{nil}\)

The rules concerning eq are unproblematic. eq(x, y) is either true or false, (eq.i). Note that this dichotomy will imply that a call to eq is always equivalent to a value. eq(x, y) is true only when its arguments are both atoms and are equi-valued, (eq.ii).

**eq rules (eq).**

(i) \(\vdash \text{eq}(x, y) \simeq t \lor \text{eq}(x, y) \simeq \text{nil}\)

(ii) \(\vdash \text{eq}(x, y) \simeq t \Leftrightarrow (x \simeq y \land \text{atom?}(x) \simeq t \land \text{atom?}(y) \simeq t)\)

The recognizers are similarly simple. τ(x) is either true or false, (τ.i), and hence always equivalent to a value. The recognizers are the characteristic functions of disjoint, and exhaustive sets, (τ.ii), and they correspond to the appropriate sets in question, (τ.iii).

**τ rules, (τ ∈ \(T\)) (τ).**

(i) \(\vdash τ(x) \simeq t \lor τ(x) \simeq \text{nil}\)

(ii) \(\vdash τ(x) \simeq t \Leftrightarrow \bigwedge_{τ' \in τ \setminus \{τ\}} τ'(x) \simeq \text{nil}\)

(iii) \(\vdash \text{atom?}(v) \simeq t \quad \text{provided } v \in \mathbb{A}\)

(iv) \(\vdash \text{lambda?}(v) \simeq t \quad \text{provided } v \in \mathbb{L}\)
The branching primitive is as simple as eq. If its first argument is false then it returns its third argument, (br.i). If its first argument is not false then it returns its second argument, (br.ii). These together imply that a call to br is always equivalent to a value.

**br rules (br).**

(i) \( x \simeq \text{nil} \Rightarrow \text{br}(x, y, z) \simeq z \)

(ii) \( \neg(x \simeq \text{nil}) \Rightarrow \text{br}(x, y, z) \simeq y \)

### 4.6. Constraint propagation principles

An important class of axioms are those which allow assertions to be propagated into and out of assertions. In order to be succinct we write \( \Phi[\pm \phi] \) to abbreviate the two formulas \( \Phi[\phi] \) and \( \Phi[\neg \phi] \).

**Static \( \simeq \) rules (S).**

(i) \( \vdash \text{let}\{x := \theta(\vec{x})\}[x \simeq \theta(\vec{x})] \quad \text{for} \quad \theta \in \mathbb{F} - \{\text{mk, app}\} \)

(ii) \( \vdash \pm (z_0 \simeq z_1) \Rightarrow \text{let}\{x := \theta(\vec{y})\}[\pm(z_0 \simeq z_1)] \quad \text{for} \quad \theta \in \mathbb{F} \)

(iii) \( \vdash \theta(\vec{y}) \Rightarrow (\text{let}\{x := \theta(\vec{y})\}[\pm(z_0 \simeq z_1)] \Rightarrow \pm(z_0 \simeq z_1)) \quad \text{for} \quad \theta \in \mathbb{F} \)

(iv) \( \vdash \pm (z \simeq \theta_0(\vec{y})) \Rightarrow \text{let}\{x := \theta_1(\vec{w})\}[\pm(z \simeq \theta_0(\vec{y}))] \quad \theta_0, \theta_1 \in \mathbb{F} \)

provided that if \( \theta_1 \in \{\text{set, app}\} \), then \( \theta_0 \in \mathbb{F} - \{\text{get, set, app}\} \).

(v) \( \vdash \theta_1(\vec{w}) \Rightarrow (\text{let}\{x := \theta_1(\vec{w})\}[\pm(z \simeq \theta_0(\vec{y}))] \Rightarrow \pm(z \simeq \theta_0(\vec{y}))) \quad \theta_0, \theta_1 \in \mathbb{F} \)

provided that if \( \theta_1 \in \{\text{set, app}\} \), then \( \theta_0 \in \mathbb{F} - \{\text{get, set, app}\} \).

As an aside we point out that (S.i) is provable when \( \theta = \text{set} \). In (S.iv) and (S.v) if \( \theta_1 \in \{\text{set, app}\} \) and \( \theta_0 \in \{\text{get, set, app}\} \), then the principles have simple counterexamples (see Section 4.9).

### 4.7. Observations

**Observation 0.** The only axioms and rules concerning get are those in (S), (U), (mk) and (set).

**Observation 1.** As a simple example of a proof we establish the following ubiquitous principle. Note that this is the term version of the principle (C.ii) that fails for operational equivalence.

**Lemma (R.eq).**

(R.eq) \( e_0 \simeq e_1 \Rightarrow R[e_0] \simeq R[e_1] \)

**Proof (R.eq).**

1. \( \vdash R[e_1] \simeq \text{let}\{z := e_1\}R[z] \quad \text{by (E.vi) for \( z \) fresh} \)
2. \( e_0 \simeq e_1 \Rightarrow \text{let}\{z := e_0\}R[z] \)
   \[ \simeq \text{let}\{z := e_0\}R[z] \quad \text{by (E.iv)} \]
3. \( e_0 \simeq e_1 \Rightarrow R[e_0] \simeq R[e_1] \quad \text{from (1) and (2) by (E) and (P)} \)

**Observation 2.** Some axioms above are new in the sense that they have replaced principles that appeared in the earlier treatments [37, 24, 40, 25]. These were pointed
out to me by Jacob Frost [20], in particular the following principles (essentially the \(\text{let}_c\) rules of the computational \(\lambda\)-calculus [43]) are now derivable.

**Lemma (C.r).**

(C.r.i) \(\vdash \text{let}\{x := \mathcal{R}[e]\}[\Phi] \iff \text{let}\{z := e\}\text{let}\{x := \mathcal{R}[z]\}[\Phi]\)

(C.r.ii) \(\vdash \mathcal{R}[\text{let}\{x := e_0\}e_1] \simeq \text{let}\{x := e_0\}\mathcal{R}[e_1] \quad x \text{ not free in } \mathcal{R}\)

**Proof (C.r.i).**

1. \(\vdash \mathcal{R}[e] \sim \text{let}\{z := e\}\mathcal{R}[z]\) by (E.vi)
2. \(\vdash \text{let}\{x := \mathcal{R}[e]\}[\Phi]\)
\(\iff \text{let}\{x := \text{let}\{z := e\}\mathcal{R}[z]\}[\Phi]\) by (1, C.iii) and (P)
3. \(\vdash \text{let}\{x := \mathcal{R}[z]\}[\Phi]\)
\(\iff \text{let}\{z := e\}\text{let}\{x := \mathcal{R}[z]\}[\Phi]\) by (2, C.v) and (P). □

**Proof (C.r.ii).**

1. \(\vdash \mathcal{R}[\text{let}\{x := e_0\}e_1] \simeq \text{let}\{y := \text{let}\{x := e_0\}e_1\}\mathcal{R}[y]\)
by (E.vi).
2. \(\vdash \mathcal{R}[\text{let}\{x := e_0\}e_1] \simeq \text{let}\{x := e_0\}\text{let}\{y := e_1\}\mathcal{R}[y]\)
from (1) using (E.vii, E.ii).
3. \(\vdash \mathcal{R}[\text{let}\{x := e_0\}e_1] \simeq \text{let}\{x := e_0\}\mathcal{R}[e_1]\)
from (2) using (E.vi, C.i, C.ii) and (E.ii). □

**Observation 3.** Similarly a previous quantifier principle [25]

\[(Q.p) \vdash L[(\forall x)\Phi] \Rightarrow (\forall x)L[\Phi]\]
where \(x \not\in \text{FV}(L) \cup \text{Traps}(L)\)
is now derivable, again pointed out to me by Jacob Frost [20].

**Proof (Q.p.).**

1. \(\vdash (\forall x)\Phi \Rightarrow \Phi\) by (Q.iii)
2. \(\vdash L[(\forall x)\Phi] \Rightarrow \Phi\)
from (1) using (C.i)
3. \(\vdash L[(\forall x)\Phi] \Rightarrow L[\Phi]\)
from (2) using (P.ii) and modus ponens
4. \(\vdash (\forall x)(L[(\forall x)\Phi] \Rightarrow L[\Phi])\)
from (3) using (Q.i)
5. \(\vdash L[(\forall x)\Phi] \Rightarrow (\forall x)(L[\Phi])\)
from (4) using (Q.ii) and the assumption that \(x \not\in \text{FV}(L)\) □

**Observation 4.** A useful corollary to the \((P)\) and \((C)\) rules is the following version of cut.

\[(P.\text{cut}) \quad \vdash \Phi_0 \Rightarrow L_0[\Phi_1] \quad \vdash \Phi_1 \Rightarrow \Phi_2 \quad \vdash \Phi_2 \Rightarrow L_1[\Phi_3] \quad \vdash \Phi_0 \Rightarrow L_0[L_1][\Phi_3]\]

**Proof (P.\text{cut}).**

1. \(\vdash \Phi_0 \Rightarrow L_0[\Phi_1]\) by assumption
2. \(\vdash \Phi_1 \Rightarrow \Phi_2\) by assumption

(3) ⊢ Φ₂ ⇒ L_1[Φ₃] by assumption
(4) ⊢ L₀[Φ₁ ⇒ Φ₂] from (2) using (C.i)
(5) ⊢ L₀[Φ₁] ⇒ L₀[Φ₂] from (4) using (P.ii) and modus ponens (P.iii)
(6) ⊢ L₀[Φ₂ ⇒ L₁[Φ₃]] from (3) using (C.i)
(7) ⊢ L₀[Φ₂] ⇒ L₀[L₁[Φ₃]] from (6) using (P.ii) and modus ponens (P.iii)
(8) ⊢ Φ₀ ⇒ L₀[L₁][Φ₃] from (1), (5) and (7) using (P.i) and modus ponens (P.iii)

4.8. Derived rules

Because L₁ is a modality akin to □ we do not have a deduction theorem. However, one can easily establish a weak form of the deduction theorem which is useful.³

Theorem (Weak deduction). Assume that from ⊢ Φ₀ one can establish ⊢ Φ₁, without using context introduction, (C.i); the ζ rule, (≡ .iii); or generalization on any variable free in Φ₀. Then ⊢ Φ₀ ⇒ Φ₁.

Proof (Weak deduction). This is a very simple modification on the standard argument [28] (induction on the length of proof). □

A simple corollary of this is a version of reduction ad absurdum:

\[
\frac{(⊢ Φ)}{⊢ False} \quad \frac{⊢ False}{⊢ ¬ Φ}
\]

is derivable if the derivation

\[
(⊢ Φ) \quad \frac{⊢ False}{⊢ ¬ Φ}
\]

does not use context introduction, (C.i); the ζ rule, (≡ .iii); or generalization on any variable free in Φ. Since if this is the case then in fact using (weak deduction) we have

\[- Φ ⇒ False.\]

And, consequently, by definition ⊢ ¬Φ. A similar observation reduces the strong form of (∃I)

\[
(∃E) \quad \frac{(⊢ Φ₀)}{⊢ Φ₁} \quad \frac{⊢ (∃y)Φ₀}{⊢ Φ₁} \quad y \not\in \text{FV}(Φ₁)
\]

³ We could strengthen the theorem by weakening the condition without using to without using ... on any formula depending on the assumption ⊢ Φ₀.
to the derivable form:

\[
(\exists E) \vdash \Phi_0 \Rightarrow \Phi_1 \\
(\exists y) \Phi_0 \Rightarrow \Phi_1, y \not\in \text{FV}(\Phi_1)
\]

4.9. Simple counterexamples

The following variations on the (S) principles are not valid.

1. \(\text{let} \{x := t(x(z))\} \vdash \Phi \)

2. \(\text{let} \{x := \text{get}(x)\} \vdash \text{app}(x(z))\)

3. \(\text{let} \{x := \text{set}(x)\} \vdash \text{app}(x(z))\)

5. Completeness

We say that an expression is first order, \(e \in \mathbb{E}_1\), iff it contains neither unapplied \(\lambda\)-expressions, nor non-\(\lambda\) applications. A formula is first order, \(\Phi \in \mathbb{W}_1\), iff it is built up from first order expressions. The appropriate first order syntactic subclasses are defined formally by the following mutually recursive definitions:

\[
\begin{align*}
\mathbb{V}_1 &= \mathbb{X} + \mathbb{A} \\
\mathbb{L}_1 &= \lambda \mathbb{X}.\mathbb{E}_1 \\
\mathbb{E}_1 &= \mathbb{V}_1 + \text{app}(\mathbb{L}_1, \mathbb{E}_1) + (F_n - \{\text{app}\})(\mathbb{E}_1^n) \\
\mathbb{C}_1 &= \{\ast\} + \mathbb{X} + \mathbb{A} + (F_n - \{\text{app}\})(\mathbb{C}_1^n) \\
\mathbb{W}_1 &= (\mathbb{E}_1 \simeq \mathbb{E}_1) + (\mathbb{W}_1 \Rightarrow \mathbb{W}_1) + (\text{let}\{X := \mathbb{E}_1\}[\mathbb{W}_1]) + (\forall X)(\mathbb{W}_1)
\end{align*}
\]

Definition \(\Pi \hat{\Pi}\). The set of constraints, \(\Pi\), and the set of complex constraints, \(\hat{\Pi}\), are defined as follows:

\[
\begin{align*}
\Pi &= - \pm(\mathbb{V}_1 \Rightarrow \mathbb{W}_1) \mid ((F_1 - \text{mk})(\mathbb{E}_1) \Rightarrow \mathbb{W}_1) \\
\hat{\Pi} &= \pm(\mathbb{V}_1 \Rightarrow \mathbb{W}_1) \Rightarrow ((F_1 - \text{mk})(\mathbb{E}_1) \Rightarrow \mathbb{W}_1) + (\hat{\Pi} \Rightarrow \hat{\Pi})
\end{align*}
\]

A simple constraint set, \(\pi\), is defined to be a finite subset of \(\Pi\), \(\pi \in \mathbb{P}_\omega(\Pi)\). A complex constraint, \(\hat{\pi}\), is an element of \(\hat{\Pi}\). We let \(\pi\) range over simple constraints sets, and \(\hat{\pi}\) range over complex constraints. A simple constraint is said to be static if it is a subset of \(\Pi - \{\text{get}\}(\mathbb{V}_1) \Rightarrow \mathbb{W}_1\). A complex constraint is said to be static if it
is a boolean combination of elements from $\Pi - (\{\text{get}\}(\forall^o) \sim \forall^o)$. Note that by (S), static constraints propagate through any contextual assertion. Also note that $\forall_1 - \{\text{mk}\} = T \cup \{\text{mk}\}$.

It would perhaps be more symmetric if we defined simple constraints to be conjunctions of constraints. The reason we define them to be sets of constraints is to facilitate a single definition, in particular $\pi_M$ below. Modulo this one definition, the reader may reasonably assume that $\pi$ is a finite conjunction of elements from $\Pi$. Thus any simple constraint set is equivalent to a single complex constraint. Note that a constraint set consists of formulas of the form $v_0 \sim v_1$ or $\neg(v_0 \sim v_1)$ or

$$\{\text{get, atom?, cell?, lambda?}\}(v_0) \sim v_1.$$ Negations of the latter are not needed since $\neg(\theta(v_0) \sim v_1)$ can be rewritten as $\{\theta(v_0) \sim z, \neg(z \sim v_1)\}$ for $z$ fresh. We sometimes abuse notation and identify $\pi$ with the conjunction of its members, hence treating a simple constraint set as a special type of complex constraint.

**Definition** (Cells($\pi$)). Cells($\pi$) is the subset of FV($\pi$) defined by

$$\text{Cells}(\pi) = \{x \in \text{FV}(\pi) | \pi | c \text{ell}\}(x) \sim t\}.$$ If $x \in \text{Cells}(\pi)$, then $x$ must be interpreted as a cell. The notion of satisfaction, $\models$, used here is simply the one defined in Section 3.2. To express the constraints implicit in a first order memory context $r$ we define for any $\pi$ the extension of $\pi$ by $r$ relative to a given set of variables $X$ to be $\pi^X_r$:

**Definition** ($\pi^X_r$). If $r \in \mathcal{F}_r$, $X \in \mathcal{P}_w(X - \text{Dom}(r))$ and FV($\pi$) $\cap$ Dom($r$) $=$ $\emptyset$, then we define $\pi$ as follows:

$$\pi^X_r = \pi \cup \pi_{\text{cells}} \cup \pi_{\text{contents}} \cup \pi_{\text{distinct}}$$

$$\pi_{\text{cells}} = \{\text{cell?}(z) \sim t | z \in \text{Dom}(r)\}$$

$$\pi_{\text{contents}} = \{\text{get}(z) \sim \Gamma(z) | z \in \text{Dom}(r)\}$$

$$\pi_{\text{distinct}} = \{\neg(z \sim y) | y \in \text{FV}(\pi) \cup X \cup (\text{Dom}(r) - \{z\}), z \in \text{Dom}(r)\}.$$ When $X = \emptyset$ we write $\pi_r$ rather than $\pi^X_r$.

**5.1. The first completeness theorem**

We begin by proving a quantifier free version of the main theorem. The full version is then a simple generalization.

**Theorem** (Completeness – 1). If $\Phi \in \mathcal{W}^o$ is first order and is quantifier free, then there is a complex constraint $\pi$ such that

$$\vdash \pi \iff \Phi.$$
Note that by using propositional calculus together with \((P.ii)\) i.e. the principle

\[
\vdash L[\Phi_0] \Rightarrow \Phi_1 \Leftrightarrow (L[\Phi_0] \Rightarrow L[\Phi_1])
\]

it suffices to demonstrate the theorem when \(\Phi\) is of the form

\[
\underbrace{L_1[1] \ldots L_n[e_0 \sim e_1] \ldots}
\]

For this reason we define

\[
L^* = \{\bullet\} + \text{let}\{X := E\} L^*
\]

and let \(L^*\) range over \(L^*\).

The proof of the completeness theorem involves the symbolic evaluation of arbitrary formulas and expressions, with respect to a suitable set of constraints, to a canonical form. The symbolic evaluation of an expression, with respect to a set of constraints \(\pi\), requires keeping track of three things: the newly allocated memory; the modifications to the original memory (described by \(\pi\)); and the remaining computation. The remainder of a computation is simply an expression. The newly allocated memory is simply a memory context. The modifications to the original memory are represented by another special kind of context called a modification, \(M\). We begin by defining relative to a fixed constraint set \(\pi\) a symbolic reduction relation \(\rightarrow^*\). It is defined in such a way that:

**Contexts:** \((L^*[\Phi_0] \rightarrow^* L^*[\Phi_1])\) implies \(\vdash \pi \Rightarrow (L^*[\Phi_0] \Leftrightarrow L^*[\Phi_1])\)

and

**Expressions:** \((e_0 \rightarrow^* e_1)\) implies \(\vdash \pi \Rightarrow e_0 \sim e_1\).

The definition requires the notion of a modification and the corresponding decomposition of contexts and expressions. The effects that the evaluation of an expression has on the original memory, described by constraints, are represented by contexts called modifications. They are simply sequences of assignments to variables that are not in the domain of the memory context, but are assumed to be cells.

**Definition (Modifications).** A modification, \(M\), is a first order context of the form

\[
\text{seq}(\text{set}(z_1, v_1), \ldots, \text{set}(z_n, v_n), \bullet)
\]

where \(z_i = z_j\) implies \(i = j\). We define \(\text{Dom}(M) = \{z_1, \ldots, z_n\}\) and (in analogy with \(\Gamma(x)\)) \(M(z_i) = v_i\) for \(i = 1, \ldots, n\).

**5.2. \(\pi\)-Reduction**

In analogy to the semantic reduction relations we define \(\rightarrow_{\pi}\), and \(\rightarrow^*_{\pi}\). In order to ensure that definitions are meaningful we introduce the notion of coherence. Roughly a constraint and a pair of memory and modification contexts are coherent (written
Coh(\pi, \Gamma; M) if Dom(\Gamma) \cap \text{FV}(\pi) = \emptyset, modifications in M are to elements of Cells(\pi), \pi decides equality on Cells(\pi), distinct elements of Dom(M) are provably distinct in \pi, and \pi contains at most one get assertion for any z in Cells(\pi). (The last condition is a technicality to make various definitions and proofs simpler.)

**Definition (Coherence).** If \Gamma is a first order memory context and M is a first order modification as above then we say (\pi, \Gamma; M) is coherent, written Coh(\pi, \Gamma; M), if the following five conditions hold:

1. Dom(\Gamma) \cap \text{FV}(\pi) = \emptyset
2. Dom(M) \subseteq Cells(\pi)
3. If x_0, x_1 \in Dom(M) are distinct, then \pi \models \neg(x_0 \sim x_1).
4. If x_0, x_1 \in Cells(\pi), then \pi \models (x_0 \sim x_1) or \pi \models \neg(x_0 \sim x_1).
5. If x \in Cells(\pi), then there is at most one formula (\text{get}(z) \sim v) \in \pi with \pi \models (z \sim x), and if (\text{get}(z) \sim v) \in \pi, then z \in Cells(\pi).

We write Coh(\pi, M) for Coh(\pi, \Gamma; M) when Dom(\Gamma) is empty, when Dom(M) is empty we write Coh(\pi, \Gamma) for Coh(\pi, \Gamma; M), and when Dom(\Gamma) and Dom(M) are both empty we write Coh(\pi) for Coh(\pi, \Gamma; M).

One use of the notion of coherence is to ensure the simplicity of the following definition of \pi_M. If a modification, M, and a constraint set, \pi, are coherent, then the modification of \pi implicit in M is made explicit in \pi_M. To construct \pi_M from \pi we first remove the set of all assertions in \pi concerning contents of cells that are mutated by M. The set removed is referred to as \pi_{\text{forget}} and is well defined by virtue of coherence. Then we add to \pi - \pi_{\text{forget}} the set of assertions, \pi_{\text{assign}} concerning the cells updated by M.

**Definition (\pi_M).** For Coh(\pi, M) we define \pi_M as follows:

\[ \pi_M = (\pi - \pi_{\text{forget}}) \cup \pi_{\text{assign}} \]

\[ \pi_{\text{assign}} = \{\text{get}(z) \sim v | M(z) = v, z \in \text{Dom}(M)\} \]

\[ \pi_{\text{forget}} = \{((\text{get}(x) \sim v) \in \pi | (\exists z \in \text{Dom}(M))(\pi \models x \sim z)\} \]

**Definition (M[z := m_k(v)]).** Suppose that M is a modification, Coh(\pi, M) and z \in Cells(\pi). Then \(M[z := m_k(v)]\) is defined to be the modification \(M'\) with Dom(M') = Dom(M) \cup \{z\}, and for \(z' \in \text{Dom}(M')\)

\[ M'(z') = \begin{cases} M(z') & \text{if } \pi \models \neg(z \sim z') \\ v & \text{if } \pi \models (z \sim z') \end{cases} \]

**Definition (e_0 \xymatrix{i \ar@{=>}[r] & e_1}).** Assume that Coh(\pi, \Gamma; M) and that \(\pi' = (\pi_{\Gamma})_M\). Then the reduction relation \(\xymatrix{i \ar@{=>}[r] & \ar@{=}[r] & e_1}\) on expressions is the reflexive transitive closure of \(\xymatrix{i \ar@{=>}[r] & \ar@{=}[r] & e_1}\) given by

\[ (\beta_\pi) \quad \Gamma[M[R[\text{app}(\lambda x. e, v)]]] \xymatrix{i \ar@{=>}[r] & \ar@{=}[r] & \Gamma[M[R[e^{(x := v)}]]] \]
Definition \((L^*; \Phi_0)^{L^*; \Phi_1} \). Assume that \(\text{Coh}(\pi, \Gamma; M)\). Then the reduction relation \(\rightarrow_{\pi}^{\Gamma}\) on formulas is the reflexive transitive closure of \(\rightarrow_{\pi}\) given by

\[
(\text{val}) \quad (\Gamma[M[\text{let}\{x := e\}]][\Phi]) \rightarrow_{\pi} \Gamma[M][\Phi[\{x := e\}]]
\]

\[(\text{red}) \quad (\Gamma[M[\text{let}\{x := e\}]][L^*][\Phi]) \rightarrow_{\pi} \Gamma_1[M][\text{let}\{x := e_1\}[L^*]][\Phi]
\]

provided \(\Gamma[M[e]] \rightarrow_{\pi} \Gamma_1[M][e_1]\).

Lemma (Coherence). Coherence is preserved by syntactic reduction.

The Context Modification Introduction lemma, \((\text{CMI})\), generalizes the contextual assertions concerning \(\text{mk}\) and \(\text{set}\) to arbitrary memory-modification contexts pairs.

Lemma (CMI). If \(\text{Coh}(\pi, \Gamma; M)\) and \(X = \text{FV}(\Gamma[M]) \cup \text{FV}(\pi)\) then

\[
\vdash \pi \Rightarrow \Gamma[M][\pi_X^{\pi}].
\]

Proof (CMI). Let

\[
\pi_{\text{cells}} = \{\text{cell}(z) \sim t \mid z \in \text{Dom}(\Gamma)\}
\]

\[
\pi_{\text{contents}} = \{\text{get}(z) \sim \Gamma(z) \mid z \in \text{Dom}(\Gamma)\}
\]

\[
\pi_{\text{distinct}} = \{\neg(z \sim y) \mid y \in X \cup (\text{Dom}(\Gamma) - \{z\}), z \in \text{Dom}(\Gamma)\}.
\]

Suppose without loss of generality that:

\[
\Gamma = \text{let}[z_1 := \text{mk}(\text{nil})] \ldots \text{let}[z_n := \text{mk}(\text{nil})]
\]

\[
\text{seq}(\text{set}(z_1, v_1), \ldots, \text{set}(z_n, v_n), \bullet)
\]

\[
M = \text{seq}(\text{set}(x_1, v'_1), \ldots, \text{set}(x_m, v'_m), \bullet)
\]
Then
\[ \vdash \pi \Rightarrow \text{let}\{z_1 := \operatorname{mk}(\text{nil})\} \ldots \text{let}\{z_n := \operatorname{mk}(\text{nil})\}[\pi] \]
by (S) and propositional logic (P).

\[ \vdash \pi \Rightarrow \text{let}\{z_1 := \operatorname{mk}(\text{nil})\} \ldots \text{let}\{z_n := \operatorname{mk}(\text{nil})\}[\pi_{\text{cells}} \cup \pi_{\text{distinct}}] \]
by (S), (mk.i) and propositional logic (P).

\[ \vdash \pi \cup \pi_{\text{cells}} \cup \pi_{\text{distinct}} \Rightarrow \text{seq}(\text{set}(x_1, v_1), \ldots, \text{set}(x_n, v_n), \bullet)[\pi \cup \pi_{\text{cells}} \cup \pi_{\text{distinct}} \cup \pi_{\text{contents}}] \]
by (S), (set.ii), (set.i) and propositional logic (P).

Thus by the above and (P.cut)
\[ \vdash \pi \Rightarrow I[\pi^{\lambda}] \]

Now by coherence we may split \( \pi \) into two disjoint sets \( \pi' \) and \( \pi_{\text{forget}} \) so that
(a) for any \((v \sim \text{get}(w)) \in \pi' \) we have that \( \pi' \models \neg(x \sim w) \) for every \( x \in \text{Dom}(M) = \{x_1, \ldots, x_m\} \).
(b) \( \pi_{\text{forget}} \) contains only those statements of the form \((v \sim \text{get}(w)) \) such that there is an \( x \in \text{Dom}(M) = \{x_1, \ldots, x_m\} \) such that \( \pi' \models x \sim w \). Thus
\[ \vdash \pi' \cup \pi_{\text{cells}} \cup \pi_{\text{distinct}} \Rightarrow \text{seq}(\text{set}(x_1, v'_1), \ldots, \text{set}(x_m, v'_m), \bullet)[\pi' \cup \pi_{\text{cells}} \cup \pi_{\text{distinct}}] \]
by (S), (set.ii), (set.i), (a.) and propositional logic (P).

\[ \vdash \pi' \Rightarrow \text{seq}(\text{set}(x_1, v'_1), \ldots, \text{set}(x_m, v'_m), \bullet)[\{\text{get}(x_i) \sim v_i \mid i = 1, \ldots, m\}] \]
by (S), (set.ii), (set.i), and propositional logic (P).

Thus by the above and (P.cut)
\[ \vdash \pi \Rightarrow I[M][\pi^{\lambda}] \]

A simple but useful corollary of (CMI) is the following:

**Corollary** (cmi).

*If* \( \text{Coh}(\pi, \Gamma; M) \), *and* \( \vdash (\pi_{\Gamma})_M \Rightarrow \uparrow \epsilon \), *then* \( \vdash \pi \Rightarrow \uparrow I[M[R[\epsilon]]] \).

**Proof** (cmi).

1. \( \vdash \pi \Rightarrow I[M][\pi^{\lambda}] \) by (CMI).
2. \( \vdash (\pi_{\Gamma})_M \Rightarrow \uparrow \epsilon \) by assumption.
3. \( \vdash \pi \Rightarrow I[M][\uparrow \epsilon] \) by the above two facts and (P.cut).
4. \( \vdash \pi \Rightarrow I[M][\text{seq}(\epsilon, \text{False})] \) by definition.
5. \( \vdash \pi \Rightarrow \text{seq}(I[M[R[\epsilon]]], [\text{False}]) \) by repeated application of (C.r)
6. \( \vdash \pi \Rightarrow \uparrow I[M[R[\epsilon]]] \) by definition. □
Before we state the key lemmas, we require one last set of definitions. Syntactic reduction is defined so that if \( \pi \) contains enough information concerning the nature of the free variables of \( e \), then

\[
e \xrightarrow{\pi} \Gamma[M[e']]\]

and either \( e' = v \) or else \( e' \) corresponds to a stuck state, one that cannot reduce due to simple type mismatches.

**Definition (\( n \)-stuck state).** An expression \( e \) is said to be \( n \)-stuck state if \( e \) can be written as \( \Gamma[M[R[e']]]) \) for some \( \Gamma, M, R, \) and \( e' \), such that \( \text{Coh}(\pi, \Gamma; M) \), \( e' \in \{\text{get}(v), \text{set}(v, v')\} \), and \((\pi R)_v \models \text{cell}? (v) \leadsto \text{nil}\).

An expression \( e \) is said to reduce to a \( \pi \)-stuck state if \( e \xrightarrow{\pi} e' \), and \( e' \) is a \( n \)-stuck state. Similarly a formula \( L^*[\Phi] \) is said to reduce to a \( \pi \)-stuck state if

\[
L^*[\Phi] \xrightarrow{\pi} \Gamma[M[\text{let}\{x := R[e']\}]][\Phi']
\]

\( \text{Coh}(\pi, \Gamma; M) \), and \( \Gamma[M[R[e']]]) \) is a \( \pi \)-stuck state.

In order to formalize the notion of a constraint set \( \pi \) containing enough information, we make the following definitions. A accessor chain of length \( n \) is a reduction context of the form

\[
\vartheta_1(\vartheta_2(\ldots \vartheta_n(\bullet)\ldots))
\]

where \( \vartheta_i \in \{\text{get}\} \). Note that an accessor chain of length 0 is just \( \bullet \). Finally we define the notion of \( n \)-completeness for constraint sets relative to a finite set of variables and atoms, \([X,A]\). The idea is that such a constraint set contains sufficient information to completely determine the evaluation of any expression of size less than \( n \) built from the given variables and atoms.

**Definition (\( n \)-Complete w.r.t. \([X,A]\)).** \( \pi \) is \( n \)-complete w.r.t. \([X,A]\) if for every \( \Theta, \Theta_0 \), accessor chains of length \( \leq n \), and \( y, y_0 \in \bar{x} \), if \( \pi \models \Theta(y) \sim v \) and \( \pi \models \Theta_0(y_0) \sim v_0 \), then

1. \( \pi \models \tau(v) \sim \tau \) or \( \pi \not\models \tau(v) \sim \text{nil} \) \( \tau \in \Gamma \)
2. \( \pi \models v \sim \alpha \) or \( \pi \not\models (v \sim \alpha) \) \( \alpha \in A \cup \{t, \text{nil}, v_0\} \)
3. \( \pi \not\models \text{cell}? (v) \sim t \) implies \( (\exists v_0 \in \mathbb{V}^0)(\pi \models \text{get}(v) \sim v_0) \)
4. \( \pi \not\models \text{cell}? (v) \sim \text{nil} \) implies \( (\exists v_0 \in \mathbb{V}^0)(\pi \models \text{get}(v) \sim v_0) \)

Note that if \( \text{Coh}(\pi) \), then (4) is automatically valid.

**Definition (Atoms(Z)).** If \( Z \subseteq \mathbb{E} \), then Atoms(Z) is the set of atoms occurring in Z.
5.3. The main lemmas

The following five lemmas enable a straightforward proof of the completeness theorem. Lemmas 0, 1, 3, and 4 hold for the full language, while Lemma 2 holds only for those expressions which are first order. In what follows let \( r \) be the usual notion of rank (on terms, contexts, and formulas) associated with the inductive definitions of these syntactic categories.

**Lemma 0.** If \( \bar{\pi}_0 \) and \( \bar{\pi}_1 \) are complex constraints, then

\[
\bar{\pi}_0 \models \bar{\pi}_1 \iff \vdash \bar{\pi}_0 \Rightarrow \bar{\pi}_1.
\]

**Lemma 1.**

(i) If \( e \models_{\bar{\pi}_e} e' \), then \( \vdash \pi \Rightarrow e \sim e' \).

(ii) If \( L^*[\Phi] \models_{\bar{\pi}_e} L^*[\Phi'] \), then \( \vdash \pi \Rightarrow (L^*[\Phi] \Leftrightarrow L^*[\Phi']) \).

**Lemma 2.** Assume that \( e, L^* \) are first order, \( \Phi \in \mathbb{W} \) (not necessarily first order), \( FV(e, L^*) \subseteq X \), \( \text{Atoms}(\pi, e, L^*) \subseteq A \) and that \( m \in \mathbb{N} \).

(i) If \( \pi \) is \((r(e)+m)\)-complete w.r.t. \( [X, A] \) and \( \text{Coh}(\pi) \), then either \( e \) reduces to a \( \pi \)-stuck state, or else there exists a memory context \( \Gamma \), a modification \( M \), and a \( v \) such that \( e \models_{\bar{\pi}} \Gamma[M[v]] \), \( \text{Coh}(\pi, \Gamma; M) \) and \( (\pi_\Gamma)_M \) is \( m \)-complete w.r.t. \( [X \cup \text{Dom}(\Gamma), A \cup \text{Atoms}(v)] \).

(ii) If \( \pi \) is \((r(L^*)+m)\)-complete w.r.t. \( [X, A] \) and \( \text{Coh}(\pi) \), then either \( L^*[\Phi] \) reduces to a \( \pi \)-stuck state, or else there exists a memory context \( \Gamma \), a modification \( M \) and a substitution \( \sigma \) such that \( L^*[\Phi] \models_{\bar{\pi}} \Gamma[M][\Phi^\sigma] \), \( \text{Coh}(\pi, \Gamma; M) \) and \( (\pi_\Gamma)_M \) is \( m \)-complete w.r.t. \( [X \cup \text{Dom}(\Gamma), A \cup \text{Atoms}(\text{Rng}(\sigma))] \).

**Lemma 3.** For any consistent \( \pi, X, A \in \mathbb{P}_o(A) \), and \( n \in \mathbb{N} \) there exists \( N \in \mathbb{N} \) and a family of constraint sets \( \{\pi_i\}_{i<N} \) such that

(i) Each \( \pi_i \) is \( n \)-complete w.r.t. \( [X, \text{Atoms}(\pi_i, A)] \), and \( \text{Coh}(\pi_i) \).

(ii) \( \vdash \pi \Rightarrow (\bigvee_{i<N} \pi_i) \)

**Lemma 4.** Suppose \( \bar{\pi} \) and \( \bar{\pi}_i = \Gamma_i[M_i[v_i]] \), \( i < 2 \), are such that \( \text{Coh}(\pi, \Gamma_i; M_i) \) for \( i < 2 \), and \( \pi \) is \( 1 \)-complete with respect to \([\text{FV}(e_0) \cup \text{FV}(e_1), \text{Atoms}(e_0, e_1)]\). Then there are static complex constraints \( \bar{\pi}_i \) and \( \bar{\pi}_{ii} \) such that

(i) \( \vdash \pi \Rightarrow (\bar{\pi}_i \Leftrightarrow e_0 \sim e_1) \)

(ii) \( \vdash \pi \Rightarrow (\bar{\pi}_{ii} \Leftrightarrow e_0 \simeq e_1) \)

5.4. Proof of completeness – I

**Proof (Completeness – I).** Fix a particular \( L^*[e_0 \simeq e_1] \) and let \( \text{FV}(L^*[e_0 \simeq e_1]) = X \) and \( \text{Atoms}(L^*[e_0 \simeq e_1]) = A \). By propositional logic and Lemma 3 we need only show that there is a complex constraint \( \bar{\pi} \) such that

\[
\vdash \pi \Rightarrow (\bar{\pi} \Leftrightarrow L^*[\Phi])
\]
assuming that \( \pi \) is \( m \)-complete w.r.t. \([X, \text{Atoms}(\pi, A)]\), and \( \text{Coh}(\pi) \) for suitably large \( m (m \geq 1 + r(L^*) + \max(r(e_0), r(e_1))) \).

To demonstrate this, pick a suitably large \( m (m \geq 1 + r(L^*) + \max(r(e_0), r(e_1))) \). By Lemma 3 (with \( \pi \) being \( \tau \sim \tau \)) there exists \( N \in \mathbb{N} \) and a family of constraint sets \( \{\pi_i\}_{i \in N} \) such that

(i) Each \( \pi_i \) is \( n \)-complete w.r.t. \([X, \text{Atoms}(\pi_i, A)]\), and \( \text{Coh}(\pi_i) \).

(ii) \( \vdash (\forall i < N \pi_i) \)

Thus we obtain a family of complex constraints \( \{\tilde{\pi}_i\}_{i < N} \) such that

\[ \vdash \pi_i \Rightarrow (\tilde{\pi}_i \Rightarrow L^*[\Phi]) \quad \text{for } i < N \]

Then letting

\[ \pi \text{ be defined to be } \bigwedge_{i < N} (\pi_i \Rightarrow \pi_i) \]

it is a simple matter to demonstrate that

\[ \vdash \pi \Rightarrow L^*[\Phi] \]

Now by Lemma 2(ii) either \( L^*[e_0 \simeq e_1] \) reduces to a \( \pi \)-stuck state, or else there exists a memory context \( \Gamma \), a modification \( \Delta \) and a substitution \( \sigma \) such that \( L^*[e_0 \simeq e_1] \Rightarrow \pi \Gamma[[e_0 \simeq e_1]]_\pi \), \( \text{Coh}(\pi, \Gamma; M) \) and \( (\pi M)_{(\pi M)} \) is \( (m - r(L^*)) \)-complete w.r.t. \([X \cup \text{Dom}(\Gamma), A \cup \text{Atoms}((\text{Rng}(\sigma)))\]). We consider these two cases in turn:

(1) Suppose \( L^*[e_0 \simeq e_1] \) reduces to a \( \pi \)-stuck state:

\[ L^*[\Phi] \Rightarrow_\pi \Gamma[M[\text{let}\{x := R(e')\}]]\Phi' \]

\[ \text{Coh}(\pi, \Gamma; M) \text{ and } \Gamma[M[R(e')]] \text{ is a } \pi \text{-stuck state.} \]

Hence \( e' \in \{\text{get}(v), \text{set}(v, v')\} \), and \( (\pi M) \vdash \text{cell}(v) \sim \text{nil} \). In this situation using Lemma 0, the axioms for undefinedness, (U), and corollary (cmi) we have that

\[ \vdash \pi \Rightarrow \Gamma[M[R(e')]] \]

Thus by this (with \( R = \text{let}\{x := R(\bullet)\} \bullet_2 \)) and (P) we obtain

\[ \vdash \pi \Rightarrow \Gamma[M[\text{let}\{x := R(e', L^*)\}]]\Phi' \]

\[ \vdash \pi \Rightarrow L^*[e_0 \simeq e_1] \text{ by Lemma 1(ii).} \]

Thus any tautological complex constraint \( \pi \) will suffice. \( \square \)

(2) Suppose \( L^*[e_0 \simeq e_1] \Rightarrow_\pi \Gamma[M[[e_0 \simeq e_1]]_\sigma = \Gamma[M[e_0 \simeq e_1]]_\pi \), \( \text{Coh}(\pi, \Gamma; M) \) and \( (\pi M)_{(\pi M)} \) is \( (m - r(L^*)) \)-complete w.r.t. \([X \cup \text{Dom}(\Gamma), A \cup \text{Atoms}((\text{Rng}(\sigma)))\]). Now

\[ \vdash \pi \Rightarrow L^*[e_0 \simeq e_1] \Rightarrow \Gamma[M[[e_0 \simeq \sigma_1]]_\sigma \text{ by Lemma 1(a).} \]

\[ \vdash \pi \Rightarrow \Gamma[M[(\pi M)_{M}]] \text{ by Lemma (CMI).} \]

Now let \( \pi' = (\pi M)_{M} \) then by Lemma 2(i), for \( i < 2 \), either \( e_i \) reduces to a \( \pi' \)-stuck state, or else there exists a memory context \( \Gamma_i \), a modification \( M_i \), and a \( v \), such that \( e_i \Rightarrow_\pi \Gamma_i[M_i[v]] \text{ and } \text{Coh}(\pi', \Gamma_i; M_i) \). Thus a priori there are three cases to consider:
when $e_0^\sigma$ and $e_1^\sigma$ both reduce to $\pi'$-stuck states; when exactly one of $e_0^\sigma$ and $e_1^\sigma$ reduce to a $\pi'$-stuck state; and when neither $e_0^\sigma$ nor $e_1^\sigma$ reduces to a $\pi'$-stuck state.

(2a) Suppose $e_0^\sigma$ and $e_1^\sigma$ both reduce to $\pi'$-stuck states. Thus $e_0^\sigma \leadsto_{\pi'} \Gamma_0[M_0[R[e_0^\sigma]]]$, and $e_1^\sigma \in \{get(v), set(v, v')\}$, with $(\pi'_0)_M \models cell?(v) \sim \text{nil}$. In this situation we can use Lemma (0), the axioms for undefinedness, (U), and Corollary (cmi) we have that
\[ \vdash \pi' \Rightarrow \Gamma_0[M_0[R[e_1^\sigma]]] \text{ for } i < 2. \]
So by Lemma 1(i) and the rules concerning undefinedness, (U),
\[ \vdash \pi' \Rightarrow e_0^\sigma \simeq e_1^\sigma \quad \text{Thus } \vdash \pi \Rightarrow L^*[e_0 \simeq e_1] \]
and so again any tautological complex constraint suffices. □

(2b) Suppose, without loss of generality that only $e_1$ reduces to a stuck state. Thus $e_0^\sigma \leadsto_{\pi'} \Gamma_0[M_0[R[e_0^\sigma]]]$, and $e_1^\sigma \in \{get(v), set(u, u')\}$, with $\text{cell?}(v) \sim \text{nil}$. Again we use Lemma 0, the axioms for undefinedness, (U), and (cmi) we have that
\[ \vdash \pi' \Rightarrow \Gamma_0[M_0[R[e_0^\sigma]]] \]
\[ \vdash \pi' \Rightarrow e_0^\sigma \]
Now on the other hand there exists a memory context $\Gamma_i$, a modification $M_1$, and a $v_1$ such that $e_1^\sigma \leadsto_{\pi'} \Gamma_1[M_1[v_1]]$. Thus
\[ \vdash \pi' \Rightarrow \Gamma_1[M_1[v_1]] \text{ using } Coh(\pi'_1, I_1; M_1) \text{ and (U)} \]
\[ \vdash \pi' \Rightarrow e_1^\sigma \text{ by Lemma 1(i)} \]
\[ \vdash \pi' \Rightarrow \neg(e_0^\sigma \simeq e_1^\sigma) \text{ by the above and (U.i)} \]
\[ \vdash \pi \Rightarrow L^*[(e_0^\sigma \simeq e_1^\sigma)] \text{ by the above and (P.cut)} \]
\[ \vdash \pi \Rightarrow \neg L^*[e_0 \simeq e_1] \text{ by (P) since } \vdash \pi \Rightarrow \neg L^*[\text{False}] \]
Thus any tautologically false complex constraint $\bar{\pi}$ will suffice. □

(2c) Suppose neither $e_0^\sigma$ nor $e_1^\sigma$ reduces to a $\pi'$-stuck state. So by assumption for $i < 2$ there exists a memory context $\Gamma_i$, a modification $M_i$, and a $v_i$ such that $e_i^\sigma \leadsto_{\pi'} \Gamma_i[M_i[v_i]]$ and $Coh(\pi'_i, I_i; M_i)$. Thus
\[ \vdash \pi' \Rightarrow (e_i^\sigma \simeq \Gamma_i[M_i[v_i]]) \text{ by Lemma 1(i)}. \]
\[ \vdash \pi' \Rightarrow (e_0^\sigma \simeq e_1^\sigma \Leftrightarrow (\Gamma_0[M_0[v_0]] \simeq \Gamma_1[M_1[v_1]])) \]
Thus
\[ \vdash \pi \Rightarrow (L^*[e_0 \simeq e_1] \Leftrightarrow (\Gamma[M][\Gamma_0[M_0[v_0]] \simeq \Gamma_1[M_1[v_1]])) \]
Consequently by Lemma 4 we obtain the desired $\bar{\pi}$. Note that by (S), and coherence we have
\[ \vdash \pi \Rightarrow \Gamma[M][\bar{\pi}] \Leftrightarrow \bar{\pi} \]
since the $\bar{\pi}$ provided by Lemma 4 is static. □
5.5. Proofs of the lemmas

Lemma 0. If \( \tilde{\pi}_0 \) and \( \tilde{\pi}_1 \) are complex constraints, then

\[ \tilde{\pi}_0 \models \tilde{\pi}_1 \quad \text{iff} \quad \vdash \tilde{\pi}_0 \Rightarrow \tilde{\pi}_1. \]

Proof. See Nelson and Oppen [45]. \( \square \)

Lemma 1.

(i) If \( e \equiv_{\pi} e' \), then \( \vdash \pi \Rightarrow e \sim e' \).

(ii) If \( L^*([\Phi]) \sim_{\pi} L^*([\Phi']) \), then \( \vdash \pi \Rightarrow (L^*([\Phi]) \leftrightarrow L^*([\Phi'])) \).

Proof. It suffices to show that if \( \text{Coh}(\pi, \Gamma; M) \), then

(i) \( \Gamma[M[e]] \sim_{\pi} \Gamma'[M'[e']] \) implies \( \vdash \pi \Rightarrow (\Gamma[M[e]] \sim \Gamma'[M'[e']]) \)

(ii) \( \Gamma[M][[\Phi]] \sim_{\pi} \Gamma'[M'][[\Phi']] \) implies \( \vdash \pi \Rightarrow (\Gamma[M][[\Phi]] \equiv \Gamma'[M'][[\Phi']]) \)

Let \( \pi' = (\pi_T)_M \) and observe that the proof naturally divides into cases corresponding to the definitions of \( \equiv_{\pi} \). We begin by proving Lemma 1(i). However in every case other than \( (\text{mk}) \) and \( (\text{set}) \) we actually prove the stronger result that

\[ \Gamma[M[R[e]]] \sim_{\pi} \Gamma[M[R[e']]] \quad \text{implies} \quad \vdash \pi \Rightarrow \Gamma[M[R]] \downarrow e \sim e'. \]

This is useful in the proof of the second part of the lemma.

Proof 1(i).

\( (\beta) \) Assume that \( e = R[\text{app}(\lambda x.e_0, v)] \). In this case:

\[ \vdash \text{app}(\lambda x.e_0, v) \sim e_0 \{x := v\} \]

by axiom \((\text{E.v})\).

\[ \vdash \pi' \Rightarrow \text{app}(\lambda x.e_0, v) \sim e_0 \{x := v\} \]

by axioms \((\text{P})\).

\[ \vdash \pi \Rightarrow \Gamma[M][\text{app}(\lambda x.e_0, v) \sim e_0 \{x := v\}] \]

by Lemmas \((\text{CMI})\), and \((\text{P.cut})\).

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{app}(\lambda x.e_0, v)]]] \sim \Gamma[M[R[e_0 \{x := v\}]]] \]

by \((\text{R.eq})\), \((\text{P})\), and \((\text{C.ii})\).

\( (\tau) \) Assume that \( e = R[\tau(v)] \) for \( \tau \in \mathcal{T} \). In this case:

\[ \pi' \models \tau(v) \sim b \]

by assumption, for \( b \in \{\text{t, nil}\} \).

\[ \vdash \pi' \Rightarrow \tau(v) \sim b \]

by Lemma 0.

\[ \vdash \pi \Rightarrow \Gamma[M][\tau(v) \sim b] \]

by Lemmas \((\text{CMI})\) and \((\text{P.cut})\).

\[ \vdash \pi \Rightarrow \Gamma[M[R[\tau(v)]]] \sim \Gamma[M[R[b]]] \]

by \((\text{R.eq})\), \((\text{P})\) and \((\text{C.ii})\).
(eq.t) Assume that \( e = R[eq(v_0, v_1)] \) and \( e' = R[t] \). In this case:

\[ 
\pi' \models \mathcal{P}_t(v_0, v_1) \\
\text{by assumption.} \\
\pi' \models v_0 \sim v_1 \land \text{atom?}(v_0) \sim t \land \text{atom?}(v_1) \sim t \\
\text{by definition of } \mathcal{P}_t. \\
\vdash \pi' \Rightarrow (v_0 \sim v_1 \land \text{atom?}(v_0) \sim t \land \text{atom?}(v_1) \sim t) \\
\text{by Lemma 0.} \\
\vdash \pi' \Rightarrow eq(v_0, v_1) \sim t \\
\text{by axiom (eq.ii).} \\
\vdash \pi \Rightarrow I[M][eq(v_0, v_1) \sim t] \\
\text{by Lemmas (CMI) and (P.cut).} \\
\vdash \pi \Rightarrow I[M][R[eq(v_0, v_1)]] \sim I[M][R[t]] \\
\text{by (R.eq), (P) and (C.ii).} \\
\]

(eq.nil) Assume that \( e = R[eq(v_0, v_1)] \) and \( e' = R[\text{nil}] \). In this case:

\[ 
\pi' \models \neg \mathcal{P}_t(v_0, v_1) \\
\text{by assumption.} \\
\pi' \models \neg v_0 \sim v_1 \land \text{atom?}(v_0) \sim t \land \text{atom?}(v_1) \sim t \\
\text{by definition of } \mathcal{P}_t. \\
\vdash \pi' \Rightarrow \neg (v_0 \sim v_1 \land \text{atom?}(v_0) \sim t \land \text{atom?}(v_1) \sim t) \\
\text{by Lemma 0.} \\
\vdash \pi' \Rightarrow \neg eq(v_0, v_1) \sim t \\
\text{by axiom (eq.ii).} \\
\vdash \pi' \Rightarrow eq(v_0, v_1) \sim \text{nil} \\
\text{by axioms (eq.i) and (P)} \\
\vdash \pi \Rightarrow I[M][eq(v_0, v_1) \sim \text{nil}] \\
\text{by Lemmas (CMI) and (P.cut).} \\
\vdash \pi \Rightarrow I[M][R[eq(v_0, v_1)]] \sim I[M][R[\text{nil}]] \\
\text{by (R.eq), (P) and (C.ii).} \\
\]

(br.t) Assume that \( e = R[br(v_0, v_1, v_2)] \) and that \( \pi' \models \neg (v_0 \sim \text{nil}) \). In this case:

\[ 
\pi' \models \neg(v_0 \sim \text{nil}) \\
\text{by assumption.} \\
\vdash \pi' \Rightarrow \neg(v_0 \sim \text{nil}) \\
\text{by Lemma 0.} \\
\vdash \pi' \Rightarrow br(v_0, v_1, v_2) \sim v_1 \\
\text{by axioms (br.ii) and (P).} \\
\]
\( \vdash \pi \Rightarrow \Gamma[M][br(v_0, v_1, v_2) \sim v_1] \) by Lemmas (CMI) and (P.cut).

\( \vdash \pi \Rightarrow \Gamma[M[R[br(v_0, v_1, v_2)]]] \Rightarrow \Gamma[M[R[v_1]]] \) by (R.eq), (P) and (C.ii).

**(br.nil)** Assume that \( e = R[br(v_0, v_1, v_2)] \) and that \( \pi' \models v_0 \sim \text{nil} \). In this case:

\[ \begin{align*}
\pi' & \models v_0 \sim \text{nil} \quad \text{by assumption.} \\
\vdash \pi' \Rightarrow v_0 \sim \text{nil} & \quad \text{by Lemma 0.} \\
\vdash \pi' \Rightarrow br(v_0, v_1, v_2) \sim v_2 & \quad \text{by axioms (br.i) and (P').} \\
\vdash \pi \Rightarrow \Gamma[M][br(v_0, v_1, v_2)] \sim v_2 & \quad \text{by Lemmas (CMI) and (P.cut).} \\
\vdash \pi \Rightarrow \Gamma[M[R[br(v_0, v_1, v_2)]]] \Rightarrow \Gamma[M[R[v_2]]] & \quad \text{by (R.eq), (P) and (C.ii).}
\end{align*} \]

**(mk)** Assume that \( e = R[mk(v)] \). In this case:

\[ \begin{align*}
\vdash R[mk(v)] & \sim \text{let}\{z := mk(v)\}R[z] \\
& \quad \text{for } z \text{ fresh, by axiom (E.vi).} \\
\vdash \Gamma[M[R[mk(v)]]] & \sim \text{let}\{z := mk(v)\}R[z] \\
& \quad \text{by rule (C.i).} \\
\vdash \Gamma[M[R[mk(v)]]] & \Rightarrow \Gamma[M[\text{let}\{z := mk(v)\}R[z]]] \\
& \quad \text{by axiom (C.ii).} \\
\vdash \Gamma[M[R[mk(v)]]] & \Rightarrow \Gamma[\text{let}\{z := mk(v)\}[M[R[z]]]] \\
& \quad \text{by axioms (E), (C.i), and (mk.ii).} \\
\vdash \Gamma[M[R[mk(v)]]] & \Rightarrow \Gamma[\text{seq(set(z,v),}[M[R[z]]])] \\
& \quad \text{by axioms (set.v), (C.ii) and (E).} \\
\vdash \Gamma[M[R[mk(v)]]] & \Rightarrow \Gamma[\text{set}(z,v)\text{seq(set(z,v),}[M[R[z]]])] \\
& \quad \text{by axioms (set.v), (C.ii) and (E).} \\
\vdash \pi \Rightarrow \Gamma[M[R[mk(v)]]] & \Rightarrow \Gamma[\text{set}(z,v)\text{seq(set(z,v),}[M[R[z]]])] \\
& \quad \text{by axioms (P).}
\end{align*} \]

**(get)** Assume that \( e = R[get(z)] \) and \( e' = R[v] \). In this case:

\[ \begin{align*}
\pi' & \models get(z) \sim v \quad \text{by assumption.} \\
\vdash \pi' \Rightarrow get(z) \sim v & \quad \text{by Lemma 0.} \\
\vdash \pi \Rightarrow \Gamma[M][get(z) \sim v] & \quad \text{by Lemmas (CMI) and (P.cut).} \\
\vdash \pi \Rightarrow \Gamma[M[R[get(z)]]] \Rightarrow \Gamma[M[R[v]]] & \quad \text{by (R.eq), (C.ii) and (P).}
\end{align*} \]

**(set)** Assume that \( e = \text{set}(z,v) \).

\[ \begin{align*}
\vdash \text{set}(z,v) & \sim \text{seq(set}(z,v),\text{nil}) \quad \text{by axioms (E.v), (C.ii), (set.i), and (P).}
\end{align*} \]
\[ \vdash \pi \Rightarrow \Gamma[M][\text{set}(z, u) \sim \text{seq}(\text{set}(z, v), \text{nil})] \]

by Lemmas (CM1), (P.cut), and (P).

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{set}(z, v)]]] \sim \Gamma[M[R[\text{seq}(\text{set}(z, v), \text{nil})]]] \]

by (R.eq), (C.ii), and (P).

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{set}(z, v)]]] \sim \Gamma[M[\text{seq}(\text{set}(z, v), R[\text{nil}])]] \]

by axioms (E.vi), (P), (C.i), and (E).

Now we consider two cases: \( z \in \text{Dom}(\Gamma) \); and \( z \notin \text{Dom}(\Gamma) \). In the former case we have:

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{set}(z, v)]]] \sim \Gamma[M[\text{seq}(\text{set}(z, v), R[\text{nil}])]] \]

by the above.

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{set}(z, v)]]] \sim \Gamma[\text{seq}(\text{set}(z, v), M[R[\text{nil}])]] \]

by axioms (set.iii), (P), (C.i), and (E).

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{set}(z, v)]]] \sim \Gamma[z := \text{mk}(v)](M[R[\text{nil}])] \]

by axioms (set.iii), (set.iv), (C.i), (P), and (E).

In the latter case we have:

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{set}(z, v)]]] \sim \Gamma[M[\text{seq}(\text{set}(z, v), R[\text{nil}])]] \]

by the above.

\[ \vdash \pi \Rightarrow \Gamma[M[R[\text{set}(z, v)]]] \sim \Gamma[M[z := e_1][\text{set}(z, v), \text{nil})]] \]

by axioms (set.iii), (P), (C.i), (P), and (E). \( \square \)

As mentioned earlier: in every case other than \( \text{mk} \) and \( \text{set} \) we actually prove the stronger result that

\[ \Gamma[M[R[e]]] \iff \Gamma[M[R[e']]] \] implies \( \vdash \pi \Rightarrow \Gamma[M[R]] e \sim e' \).

**Proof** (ii). Assume that \( \text{Coh}(\pi, \Gamma; M) \). Then we have two cases:

(val) \( \Gamma[M[\text{let}\{x := v\}][\Phi]] \iff \Gamma[M[\Phi^{x := v}]] \)

(red) \( \Gamma[M[\text{let}\{x := e\}][\Phi]] \iff \Gamma_1[M[\text{let}\{x := e_1\}][\Phi]] \)

where \( \Gamma[M[e]] \iff \Gamma_1[M[e_1]] \)

(val) In this case:

\[ \vdash \text{let}\{x := v\}[\Phi] \iff \Phi^{x := v} \]

by axiom (C.iv).

\[ \vdash \pi' \Rightarrow \text{let}\{x := v\}[\Phi] \iff \Phi^{x := v} \]

by axioms (P).
Here we consider three separate subcases depending on the nature of the reduction \( \Gamma[M[e]] \rightarrow_n \Gamma'[M'[e']] \). If this does not involve \( \text{(mk)} \) or \( \text{(set)} \), then by the stronger version of Lemma 1(i) we have that

\[
\vdash \pi \Rightarrow \Gamma[M][e \sim e'].
\]

Thus by axioms \((C.iii)\), \((P)\), and lemma \((P.cut)\) we obtain the desired conclusion:

\[
\vdash \pi \Rightarrow (\Gamma[M[\text{let}\{x := e\}0]][\Phi] \Leftrightarrow \Gamma[M[\text{let}\{x := e'\}0]][\Phi]).
\]

Thus we are only left with the cases when the reductions involves \( \text{(mk)} \) or \( \text{(set)} \).

**\text{(mk)}** Let \( R[\text{mk}(v), \text{mk}(u)] \) abbreviate \( \text{let}\{x := \text{mk}(v)\} \). In this case we need to show that

\[
\vdash \pi \Rightarrow (\Gamma[M[R[\text{mk}(v), \text{mk}(u)]]][\Phi] \Leftrightarrow \Gamma[\{z := \text{mk}(v)\}][M[R[z, 0]]][\Phi]) \quad \text{for } z \text{ fresh}.
\]

To this end,

\[
\vdash \Gamma[M[R[\text{mk}(v), 0]][\Phi] \Leftrightarrow \Gamma[M[\text{let}\{z := \text{mk}(v)\}][\Phi]]
\]

by axioms \((C.iii)\), \((P)\) and rule \((C.i)\).

\[
\vdash \Gamma[M[R[\text{mk}(v), 0]][\Phi] \Leftrightarrow \Gamma[\text{let}\{z := \text{mk}(v)\}][M[R[z, 0]]][\Phi]
\]

by axioms \((\text{mk.iii})\) and \((P)\).

\[
\vdash \Gamma[M[R[\text{mk}(v), 0]][\Phi] \Leftrightarrow \Gamma[\text{let}\{z := \text{mk}(\text{nil})\}][M[R[z, 0]]][\Phi]
\]

by axioms \((\text{mk.iii})\) and \((P)\) and rule \((C.i)\).

\[
\vdash \Gamma[M[R[\text{mk}(v), 0]][\Phi] \Leftrightarrow \Gamma[\{z := \text{mk}(v)\}][M[R[z, 0]]][\Phi]
\]

by \((\text{mk.iii})\) and \((P)\).

**\text{(set)}** Again let \( R[\text{set}(v), 0] \) abbreviate \( \text{let}\{x := 0\} \). In this case \( e = \text{set}(z, v) \) and we must consider two possibilities, either \( z \in \text{Dom}(\Gamma) \) or not. In the former case we must show:

\[
\vdash \pi \Rightarrow (\Gamma[M[\text{set}(z, v), 0]][\Phi] \Leftrightarrow \Gamma[\{z := \text{mk}(v)\}][M[R[\text{nil}, 0]]][\Phi])
\]

While in the latter case we must show

\[
\vdash \pi \Rightarrow (\Gamma[M[\text{set}(z, v), 0]][\Phi] \Leftrightarrow \Gamma[M[\{z := \text{mk}(v)\}, 0]][R[\text{nil}, 0]]][\Phi])
\]
In either case we begin by observing that:

\[ \vdash \text{set}(z, u) \sim \text{seq}(\text{set}(z, u), \text{nil}) \] by axioms \((E.v), (C.ii), (\text{set}.)\). and \((P)\).

\[ \vdash R[\text{set}(z, v), \bullet][\Phi] \iff R[\text{seq}(\text{set}(z, v), \text{nil}), \bullet][\Phi] \] by axiom \((C.iii)\).

\[ \vdash R[\text{set}(z, v), \bullet][\Phi] \iff \text{seq}(\text{set}(z, v), R[\text{nil}, \bullet])][\Phi] \] by axioms \((C.v)\) and \((E)\).

\[ \vdash \pi \Rightarrow M[R[\text{set}(z, v), \bullet]][\Phi] \iff \text{seq}(M[\text{set}(z, v)], R[\text{nil}, \bullet])[\Phi] \] by axioms \((P)\), \((C.i)\), and \((C.ii)\).

\[ \vdash \pi \Rightarrow M[R[\text{set}(z, v), \bullet]][\Phi] \iff \text{seq}(M[\text{set}(z, v)], R[\text{nil}, \bullet])[\Phi] \] by axioms \((P)\), and \((C.v)\).

Now the latter case is simple since:

\[ \vdash M[\text{set}(z, v)] \sim M\{z := \text{mk}(v)\}\text{[\text{nil}]} \] by the proof of Lemma 1(i).

\[ \vdash \pi \Rightarrow (M[R[\text{set}(z, v), \bullet]][\Phi] \iff \text{seq}(M\{z := \text{mk}(v)\}_z[\text{nil}], R[\text{nil}, \bullet])[\Phi]) \] by axioms \((P)\), and \((C.iii)\).

\[ \vdash \pi \Rightarrow (M[R[\text{set}(z, v), \bullet]][\Phi] \iff M\{z := \text{mk}(v)\}_z[R[\text{nil}, \bullet]][\Phi]) \] by axioms \((C.iv)\), \((P)\), and \((C.iii)\).

\[ \vdash \pi \Rightarrow (I[M[R[\text{set}(z, v), \bullet]]][\Phi] \iff I[M\{z := \text{mk}(v)\}_z[R[\text{nil}, \bullet]]][\Phi]) \] by \((P.\text{cut})\) and \((P)\).

Now in the former case we have:

\[ \vdash \pi \Rightarrow M[\text{set}(z, v)] \sim \text{seq}(\text{set}(z, v), M[\text{nil}]) \] again by the proof of Lemma 1(i).

\[ \vdash \pi \Rightarrow (M[R[\text{set}(z, v), \bullet]][\Phi] \iff \text{seq}(\text{set}(z, v), M[\text{nil}], R[\text{nil}, \bullet])[\Phi]) \] by axioms \((P)\), \((C.iii)\), \((C.i)\), \((C.v)\).

\[ \vdash \pi \Rightarrow (M[R[\text{set}(z, v), \bullet]][\Phi] \iff \text{seq}(\text{set}(z, v), M[R[\text{nil}, \bullet]])[\Phi]) \] by axioms \((C.iii)\), \((C.iv)\), \((C.i)\), \((C.v)\) and \((P)\).

\[ \vdash \pi \Rightarrow (I[M[R[\text{set}(z, v), \bullet]]][\Phi] \iff I[\text{seq}(\text{set}(z, v), M[R[\text{nil}, \bullet]])][\Phi]) \] by lemma \((P.\text{cut})\) and axioms \((C.ii)\) and \((P)\).

\[ \vdash \pi \Rightarrow (I[M[R[\text{set}(z, v), \bullet]]][\Phi] \iff I[x := \text{mk}(v)]M[R[\text{nil}, \bullet]][\Phi]) \] by axioms \((\text{mk}.)\), \((\text{set}.)\), \((\text{set}.)\) \((\text{set}.)\) and \((C.iii)\). \(\square\)

**Lemma 2.** Assume \(e, L^*[\Phi]\) are first order, \(\text{FV}(e, L^*) \subseteq X\), and \(\text{Atoms}(\pi, e, L^*) \subseteq A\).

(i) If \(\pi\) is \((r(e) + m)\)-complete w.r.t. \([X, A]\) and \(\text{Coh}(\pi)\), then either \(e\) reduces to a \(\pi\)-stuck state, or else there exists a memory context \(\Gamma\), a modification \(M\), and a \(v\) such that \(e \Rightarrow_\pi \Gamma[M[v]]\). \(\text{Coh}(\pi, \Gamma; M)\) and \((\pi_\Gamma)_M\) is \(m\)-complete w.r.t. \([X \cup \text{Dom}(\Gamma), A \cup \text{Atoms}(v)]\).
(ii) If $\pi$ is $(r(L^* + \nu) + m)$-complete w.r.t. $[X, A]$ and $\text{Coh}(\pi)$, then either $L^*[\Phi]$ reduces to a $\pi$-stuck state, or else there exists a memory context $\Gamma$, a modification $M$ and a substitution $\sigma$ such that $L^*[\Phi] \rightarrow z \Gamma[M][\Phi^\nu]$, $\text{Coh}(\pi, \Gamma; M)$ and $(\pi; \Gamma)_M$ is $m$-complete w.r.t. $[X \cup \text{Dom}(\Gamma), A \cup \text{Atoms}([\text{Rng}(\sigma)])].$

**Proof.** Both these follow from the simple observation that if $e \rightarrow^\pi e'$ and $\pi$ is $(r(e) + m)$-complete w.r.t. $[X, A]$, then $\pi$ is $(r(e') + m)$-complete w.r.t. $[X, A]$.

*Lemma 3.* For any consistent $\pi, X, A \in \Phi(A)$, and $n \in \mathbb{N}$ there exists $N \in \mathbb{N}$ and a family of constraint sets $\{\pi_i\}_{i < N}$ such that

(i) Each $\pi_i$ is $n$-complete w.r.t. $[X, \text{Atoms}(\pi_i, A)]$, and $\text{Coh}(\pi_i)$.

(ii) $\vdash \pi \leftrightarrow (\bigvee_{i < N} \pi_i)$

**Proof.** Suppose that $\pi$ is consistent but not $n$-complete w.r.t. $[X, A]$. Pick $\Theta, \Theta_0$, accessor chains of length $\leq n$, and $y, y_0 \in X$ such that $\pi \vdash \Theta[y] \sim v$ and $\pi \vdash \Theta_0[y_0] \sim v_0$ and one of the Lemmas 1–4 fails. We repair each possible failure in turn. If Lemma 1 fails then

$$\vdash \pi \leftrightarrow (\pi \cup \{v \sim t\}) \lor (\pi \cup \{v \sim \text{nil}\})$$

by ($\tau.i$), for $\tau \in \mathbb{T}$. If Lemma 2 fails then

$$\vdash \pi \leftrightarrow (\pi \cup \{v \sim z\}) \lor (\pi \cup \{-(v \sim z)\})$$

by propositional logic ($P$), for $z \in A \cup \{t, \text{nil}, v_0\}$. If Lemma 3 fails then

$$\vdash \pi \leftrightarrow (\pi \cup \{\text{get}(v) \sim z\})$$

for $z$ fresh, by ($U.\text{iv}$) and existential elimination ($\exists \text{E}$). If Lemma 4 fails, then $\pi$ is inconsistent by ($U.\text{iv}$) and ($\exists \text{I}$). This contradicts our initial assumptions. Thus in each (possible) case it is possible to enlarge $\pi$ or branch and enlarge so as to rectify this particular failure. Generating the required family $\{\pi_i | i < N\}$ is now trivial. \hfill $\square$

*Lemma 4.* Suppose $\pi$ and $e_i = \Gamma_i[M_i[v_i]]$, $i < 2$, are such that $\text{Coh}(\pi, \Gamma; M_i)$ for $i < 2$, and $\pi$ is 1-complete with respect to $[FV(e_0) \cup FV(e_1), \text{Atoms}(e_0, e_1)]$. Then there are static complex constraints $\tilde{\pi}_i$ and $\tilde{\pi}_a$ such that

(i) $\vdash \pi \Rightarrow (\tilde{\pi}_i \Rightarrow e_0 \sim e_1)$

(ii) $\vdash \pi \Rightarrow (\tilde{\pi}_a \Rightarrow e_0 \equiv e_1)$

**Proof.** Define $M^s_i$ as follows:

$$M^s_i(z) = \begin{cases} M_i(w) & \text{if } w \in \text{Dom}(M_i) \text{ and } \pi \models w \sim z \\
    w & \text{if } \pi \models w \sim \text{get}(z) \text{ and } (\forall y \in \text{Dom}(M_i))(\pi \models -(z \sim y)) \end{cases}$$

The supposition that $\text{Coh}(\pi, \Gamma; M_i)$ and $\pi$ is 1-complete with respect to $[FV(e_i), \text{Atoms}(e_i)]$ suffice to ensure that $M^s_i$ is defined on $\text{Cells}(\pi)$. Actually $M^s_i$ as defined is a relation modulo $\pi$-equivalence, but this suffices for our purposes.
4(i) Without loss of generality we may assume that \( \text{Dom}(\Gamma_0) \) and \( \text{Dom}(\Gamma_1) \) are disjoint and of the same cardinality, \(|\text{Dom}(\Gamma_0)| = |\text{Dom}(\Gamma_1)|\). Since otherwise any tautologically false static constraint would suffice. Now for each bijection 
\[
f : \text{Dom}(\Gamma_0) \rightarrow \text{Dom}(\Gamma_1)
\]
define \( \pi_f \) to be the set
\[
\{ x_i \sim f(x_i), \Gamma_0(x_i) \sim \Gamma_1(f(x_i)), M^0_i(z) \sim M^1_i(z), v_0 \sim v_1 \mid x_i \in \text{Dom}(\Gamma_0), z \in \text{Cells}(\pi) \}
\]
Observe that \( \pi_f \) is a set of static constraints. The desired static complex constraint is just
\[
\pi_i = \bigvee \{ \pi_f \mid f : \text{Dom}(\Gamma_0) \rightarrow \text{Dom}(\Gamma_1) \text{ a bijection} \}
\]

4(ii) This case is slightly more complex than the previous one. We begin by defining two increasing sequences, \( \{\delta_j \mid j \in \mathbb{N}\} \), of subsets of \( \text{Dom}(\Gamma_i) \) for \( i < 2 \) by induction.
\[
\delta^0_0 = \text{Dom}(\Gamma_0) \cap \text{FV}(M_i[v_i])
\]
and having defined \( \delta^i_j \) we define \( \delta^i_{j+1} \) by
\[
\delta^i_{j+1} = \delta^i_j \cup \{ z \in \text{Dom}(\Gamma_i) \mid (\exists x \in \delta^i_j)(z \in \text{FV}(\Gamma_i(x))) \}
\]
Put \( \delta^i = \bigcup_{j \in \mathbb{N}} \delta^i_j \). We claim that \( \text{Dom}(\Gamma_i) - \delta^i \) is precisely the garbage created by \( e_i \). Consequently, we may assume that the two sets \( \delta^0 \) and \( \delta^1 \) are disjoint and of the same cardinality, \(|\delta^0| = |\delta^1|\), since otherwise any tautologically static constraint would suffice. We now proceed in the same fashion as in the previous case. For each bijection 
\[
f : \delta^0 \rightarrow \delta^1
\]
define \( \pi_f \) to be the set
\[
\{ x_i \sim f(x_i), \Gamma_0(x_i) \cong \Gamma_1(f(x_i)), M^0_i(z) \cong M^1_i(z), v_0 \cong v_1 \mid x_i \in \text{Dom}(\Gamma_0), z \in \text{Cells}(\pi) \}
\]
Observe that \( \pi_f \) is a set of static constraints. The desired static complex constraint is just
\[
\pi_u = \bigvee \{ \pi_f \mid f : \delta^0 \rightarrow \delta^1 \text{ a bijection} \}
\]

6. The second completeness theorem

We now demonstrate that the quantifier free assumption can be eliminated from the completeness result of the previous section.

**Definition** (\( \bar{\Pi} \)). The set of quantified constraints, \( \bar{\Pi} \) is defined by
\[
\bar{\Pi} := \pm (\forall^o \cong \forall^o) + ((\forall - \{mk\})(\forall^o) \cong \forall^o) + (\bar{\Pi} \Rightarrow \bar{\Pi}) + (\forall x)\bar{\Pi}
\]
Theorem (Completeness – II). If $\Phi \in \mathcal{W}_0$ is first order, then there is a quantified constraint $\pi$ such that

$$\vdash \pi \Leftrightarrow \Phi$$

Proof. We prove this by induction on the quantifier rank of our first order $\Phi$. The only new case we need to consider in this more general situation is how to simplify a formula of the form $L[(\forall x)\Phi_0]$. Fix a particular $L^*[(\forall x)\Phi]$ and let $\text{FV}(L^*[(\forall x)\Phi]) = X$ and $\text{Atoms}(L^*[(\forall x)\Phi]) = A$. By propositional logic and Lemma 3 we need only show that there is a complex constraint $\pi$ such that

$$\vdash \pi \Rightarrow (\pi \Leftrightarrow L^*[(\forall x)\Phi])$$

assuming that $\pi$ is $m$-complete w.r.t. $[X, \text{Atoms}(\pi, A)]$, and $\text{Coh}(\pi)$ for suitably large $m$ ($m \geq 1 + r(L^*) + \max(r((\forall x)\Phi))$). Now by Lemma 2(ii) either $L^*[(\forall x)\Phi]$ reduces to a $\pi$-stuck state, or else there exists a memory context $\Gamma$, a modification $M$ and a substitution $\sigma$ such that $L^*[(\forall x)\Phi] \Rightarrow_{\pi} \Gamma[M][(\forall x)\Phi^\sigma]$, $\text{Coh}(\sigma, \Pi; M)$ and $(\Pi)_{M}$ is $(m - r(L^*))$-complete w.r.t. $[X \cup \text{Dom}(\Pi), A \cup \text{Atoms}(\text{Rng}(\sigma))]$. We should point out that we are not using a more general version of 2(ii) than the one stated, since there are no restrictions on the formula inside the contextual assertion.

The first case proceeds exactly as in the proof of the first completeness theorem. We are left to deal with the second case.

(2) Suppose $L^*[(\forall x)\Phi] \Rightarrow_{\pi} \Gamma[M][(\forall x)\Phi^\sigma]$, $\text{Coh}(\sigma, \Pi; M)$ and $(\Pi)_{M}$ is $(m - r(L^*))$-complete w.r.t. $[X \cup \text{Dom}(\Pi), A \cup \text{Atoms}(\text{Rng}(\sigma))]$. Now

$$\vdash \pi \Rightarrow (L^*[(\forall x)\Phi] \Leftrightarrow \Gamma[M][(\forall x)\Phi^\sigma])$$

by Lemma 1(a).

Also note that:

$$\vdash \Gamma[M][(\forall x)\Phi^\sigma] \Leftrightarrow \Gamma[M][(\forall x)(\Phi^\sigma)] \quad \text{given obvious hygiene assumptions.}$$

$$\vdash \Gamma[M][(\forall x)(\Phi^\sigma)] \Leftrightarrow \Gamma[(\forall x)M][(\Phi^\sigma)]$$

by repeated application of (Q.iv) and its converse.

$$\vdash \Gamma[(\forall x)M][(\Phi^\sigma)] \Leftrightarrow \Gamma[(\forall x) \bigwedge_{z \in \text{Dom}(\Pi)} (\Phi^\sigma)[x := z]]$$

by repeated application of (Q.vi) and its converse.

We are now in a position to apply the induction hypothesis to obtain:

$$\vdash \pi_0 \Leftrightarrow \Gamma[M[(\Phi^\sigma)]]$$

$$\vdash \pi_1 \Leftrightarrow \Gamma \left[ M \bigwedge_{z \in \text{Dom}(\Pi)} (\Phi^\sigma)[x := z] \right]$$

Consequently,

$$\vdash \pi \Rightarrow (L^*[(\forall x)\Phi] \Leftrightarrow ((\forall x)\pi_0 \land \pi_1))$$

and thus we reach the desired conclusion. $\square$
To obtain the desired correspondence between the proof theory and semantics we have to elaborate on the first lemma. We need an analysis of the theory of those first order structures in the language

\{get, atom?, cell?, lambda?\}

which satisfy the principles enumerated in Section 4. This theory is known to be decidable, indeed the weak monadic second order is shown to be decidable in Rabin's landmark S2S paper [52]. Consequently, we may conclude that the provability of a first order $\Phi \in W^\omega$ is decidable.

7. Conclusions, directions and future work

In this paper we have continued our investigations into a Variable Typed Logic of Effects that began in [37, 24, 38, 40, 25]. In particular, we presented an axiomatization of the base first order theory. In [22] we described an encoding of this logic into the generic proof assistant Isabelle [47]. Encoding the syntax and proof theory of the logic was a relatively painless procedure, especially when compared with the contortions required for logics of the Hoare and Dynamic ilk [39, 7]. This encoding has subsequently been improved, elaborated and utilized in [21].

Since the semantics of the underlying $\lambda_{mk}$-calculus is operational, and the semantics of the logic is defined strictly in terms of syntactic entities, it seems not unreasonable to expect an implementation to be capable of encoding it. This would allow for both proof theoretical and semantic reasoning to be carried out at the same time in the same context [61]. This would have two obvious attractions:

1. It will also allow the system to semantically verify its own proof system, an extremely attractive idea.
2. It would allow for the dynamic enrichment of the proof theory by introducing new, semantically verified, principles. Thus the logic implemented would be truly dynamic.

The only obstacle to successfully encode the semantics is the problem of encoding lambda calculus style contexts and hole filling (i.e. the corresponding notion of substitution with variable binding capture). To achieve this it may be necessary to adopt the binding structure approach developed in [56, 58]. We have recently developed a named variable version of the binding structure approach [31], what remains is to implement this in a logical framework.

While we have presented a completeness result in this paper, substantial work still remains to be done. We mention here several open problems in this area:

1. The axiom system presented here contains axioms and rules for quantifiers but not structured data (i.e. typically immutable pairs), however the question of whether these axioms and rules are complete remains open. We conjecture that the techniques presented in [60] can be modified and adapted to this framework to obtain an affirmative answer.
2. In an operational setting, the main tool for establishing principles such as structural induction, fixed-point induction, co-induction, and simulation induction is induction on the length of computation. By incorporating both semantical and proof-theoretical principles into the proof environment we solve the problem of how computation induction can be formalized within this programming logic.

3. Thus far we have studied systems with control features [57] as well as systems with imperative features. The unification of these two theories has only recently been studied in detail [59]. While that work concentrates solely on the properties of the underlying term language, the results established indicate that VTLoE like logics can be developed for a very general class of programming languages. In this particular case (languages with imperative and control features) most of the nice meta-theoretic properties such as completeness should carry over in some form or other, certain principles must be modified if they are to remain valid [17].

4. In the long term it is hoped that our work on concurrent and distributed programming [4] and [5] will result in a unified approach to all three enrichments to the underlying functional language.

Acknowledgements

This paper owes its existence to the interest shown in VTLoE by Jacob Frost. His numerous questions concerning the (possibility of an) axiomatization forced the author to sit down and work out the details. Much of the work reported here was done while the author was at the University of Tasmania @ Launceston, and partially supported by ARC grant M0008202. As usual the author would also like to thank Carolyn Talcott for providing much of the inspiration and error detection. The anonymous referee must also be thanked for many helpful comments.

References


