An Operational Logic of Effects

Jacob Frost
Department of Computer Science
Technical University of Denmark
Building 344
2800 Lyngby
Denmark
E-mail jf@id.dtu.dk

Ian A. Mason
Applied Computing & Mathematics
University of Tasmania at Launceston
Launceston
Tasmania 7250
Australia
E-mail imason@leven.appcomp.utas.edu.au

Abstract

In this paper we describe 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 too present an axiomatization of the base first-order theory, and an implementation of the logic into the generic proof assistant Isabelle. *We also indicate the directions of our current research to blend these two advances into an operational whole.*

Keywords semantics, logic, derivation, verification, specification, theorem proving.

1 Introduction

In this paper we continue the investigations into a *Variable Typed Logic of Effects* that began in [20, 11, 21, 23, 12]. In particular we present an axiomatization of the base first-order theory, and an encoding of the logic into the generic proof assistant *Isabelle* [26]. We also indicate the directions of our current research to blend these two advances into an operational whole. In the remainder of this section we give a brief overview of both the object logic, VTLoE, and Isabelle.

1.1 An Introduction to VTLoE

VTLoE is a logic for reasoning about imperative functional programs, inspired by the variable type systems of Feferman [5, 6]. VTLoE builds upon recent advances in the semantics of languages with effects [7, 9, 15, 18, 19] and goes well beyond traditional programming logics, such as Hoare's logic [2] and Dynamic logic [10] by treating a richer programming language and more expressive logical language. It is close in spirit to Specification Logic [29] and to Evaluation Logic [27].

Proceedings of CATS'96 (Computing: The Australasian Theory Symposium), Melbourne, Australia, January 29–January 30 1996.

The underlying programming language of VTLoE, $\lambda_{\mathtt{mk}}$, is based on the call-by-value lambda calculus extended by the reference primitives `mk`, `set`, `get`. It can thus be thought of as a fragment of untyped ML or a variant of Scheme. In our language atoms, cells and lambda abstractions are all first class values and as such are storable. This has several consequences. Firstly, mutation and variable binding are separate and so we avoid the problems that typically arise (e.g. in Hoare's and Dynamic logic) from the conflation of program variables and logical variables. Secondly, the equality and sharing of cells (aliasing) is easily expressed and reasoned about. Thirdly, the combination of mutable cells and lambda abstractions allows us to study object based programming within our framework. The logic combines the features and benefits of equational calculi as well as program and specification logics. There are three layers. The foundation is the syntax and semantics of $\lambda_{\mathtt{mk}}$, the underlying term/programming language. The second layer is a first-order theory built on assertions of program equivalence and program modalities called `let`-*formulas*. The third layer extends the logic to include class terms, class membership, and quantification over class variables.

1.2 An Introduction to Isabelle

Isabelle is a generic proof assistant. It can be instantiated to support reasoning in a variety of *object-logics*. At present these object-logics include: first- and higher-order logic, sequent calculus, Zermelo-Fraenkel set theory, a version of Constructive Type Theory and several modal logics. A number of experimental logics, including the one described in this paper, are also being developed.

An object-logic is encoded by extending the *meta-logic* of the *Pure* Isabelle system. This section introduces the Pure system and describes how it can be extended to support new object-logics.

1.2.1 Syntax

The language of Isabelle is typed lambda calculus. New syntax is declared by extending the basic language with new classes, types, constants etc. This section introduces the basic language and describes how it is extended with the meta-level syntax for the Pure Isabelle system. Although Isabelle's syntax is purely ASCII, mathematical symbols are used to improve readability. Isabelle's *type classes* control polymorphism and permit forms of overloading. They closely resemble Haskell's type classes [13] and are essentially sets of types. The Pure system has a build-in class logic of all logical types. It is possible to declare new types and type constructors with associated arities. Arities are constructed from classes and can be seen "types" of type constructors. The Pure system declares a type of meta-formulas (prop) and a function type constructor ($\Rightarrow$) with the following arities:

$$\mathsf{prop} :: \mathsf{logic} \qquad \Rightarrow :: (\mathsf{logic}, \mathsf{logic})\mathsf{logic}$$

The type prop is typically used as the type of judgements in object-logics. Curried function types are abbreviated $[\tau_1, \ldots, \tau_n] \Rightarrow \tau$. Terms are the usual ones. Abstraction is written $\lambda_{\mathcal{I}} x.t$ and application $t_1(t_2)$. The curried forms are abbreviated $\lambda_{\mathcal{I}} x_1 \ldots x_n.t$ and $t(t_1, \ldots, t_n)$. New constants are declared by giving their type. The Pure system declares constants for meta-implication ($\Longrightarrow$), meta-quantification ($\bigwedge$) and meta-equality ($\equiv$):

$$\begin{aligned} \Longrightarrow &:: [\mathsf{prop}, \mathsf{prop}] \Rightarrow \mathsf{prop} \\ \bigwedge &:: (\alpha :: \mathsf{logic} \Rightarrow \mathsf{prop}) \Rightarrow \mathsf{prop} \\ \equiv &:: [\alpha :: \{\}, \alpha] \Rightarrow \mathsf{prop} \end{aligned}$$

In the above, type variables (α) range over the universal sort $\{\}$ (containing all types) or logic, thus restricting polymorphism. Nested meta-implication is abbreviated $[\![\Phi_1, \ldots, \Phi_n]\!] \Longrightarrow \Phi$ and nested meta-quantification $\bigwedge x_1 \ldots x_n.\Phi$.

Isabelle has many features to extend the basic lambda calculus syntax with more readable notation. Among these features are *mixfix declarations* which can express any context-free priority grammar, and *translations* on both ASTs (abstract syntax trees) and lambda terms. Translations are applied automatically during parsing and printing. The symbol $\rightleftharpoons$ is used for *macros*. These are a particular kind of AST translations which are especially easy to read and write. Macros apply to ASTs and can therefore express syntactic translations which meta-equalities cannot.

1.2.2 Rules

The rules of Isabelle's intuitionistic higher-order meta-logic are represented in LCF style. They are essentially functions between terms of type prop. A central rule is the resolution rule described in figure 1. Here s is a higher-order unifier of ψ and ϕ_i. A substantial amount of machinery is connected with resolution and higher-order unification. This machinery includes schematic variables, lifting over both formulas and variables etc. Rules at the object-level are simply meta-formulas. A meta-formula such as $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ can be read as an object-level natural deduction rule with premises $\phi_1, \ldots, \phi_n$ and conclusion ϕ. Natural deduction notation is used accordingly.

1.2.3 Proofs

Proofs at the meta-level are carried out by applying functions representing valid meta-inferences. Proving a meta-theorem corresponds to deriving an object-rule. In particular, resolution can be used to build object-proofs by joining object-rules. This supports both forward and backward proof of theorems at the object-level.

Isabelle's *tactics* support backwards proofs at the object-level. A meta-formula $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ represents the state of a backward proof with subgoals $\phi_1, \ldots, \phi_n$ and goal ϕ. Tactics are valid meta-inferences, working on proof states by refining the subgoals. Isabelle contains simple tactics for refining a proof state by resolution, for proving subgoals by assumption etc. There are also a number of generic packages which, upon instantiation to a particular object-logic, provide tactics for simplification and proof construction using classical proof procedures.

Isabelle's *tacticals* combine tactics into new tactics. They can be used to encode powerful proof procedures as tactics. Isabelle has a number of build-in tacticals, for sequencing, alteration, searching etc.

1.3 Overview & Notation.

The paper is organized as follows. In section 2 we describe the syntax, semantics and Isabelle encoding of the of the terms of VTLoE. In section 3 we describe the syntax, semantics and encoding of the formulas of VTLoE. In section 4 we present the proof theory of VTLoE and its encoding. In section 5 we discuss our current work, conjectures, future research and present our concluding remarks. We conclude this introduction with a note concerning notation. Let Y_0, Y_1 be sets. We use the usual notation for set membership and function application. $Y_0 \xrightarrow{f} Y_1$ is the set of finite maps from Y_0 to Y_1. $[Y_0 \Rightarrow Y_1]$ is the set of total functions f with domain Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. For any function f, $f\{y := y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$.

2 The Terms of VTLoE

2.1 The Syntax of Terms

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

$$\{\mathtt{t}, \mathtt{nil}\} \subseteq \mathbb{A}$$

$$\frac{[\![\psi_1;\ldots;\psi_m]\!] \Longrightarrow \psi \quad [\![\phi_1;\ldots;\phi_n]\!] \Longrightarrow \phi}{([\![\phi_1;\ldots;\phi_{i-1};\psi_1;\ldots\psi_m;\phi_{i+1};\ldots;\phi_n]\!] \Longrightarrow \phi)s} \quad (\psi s \equiv \phi_i s) \quad 1 \leq i \leq n$$

Figure 1: The Resolution Rule

$$\mathbb{F}_1 = \{\mathtt{mk}, \mathtt{get}, \mathtt{fst}, \mathtt{snd}\} \quad \cup$$
$$\{\mathtt{atom}?, \mathtt{cell}?, \mathtt{pair}?, \mathtt{lambda}?\}$$
$$\mathbb{F}_2 = \{\mathtt{app}, \mathtt{eq}, \mathtt{set}, \mathtt{pr}\}$$
$$\mathbb{F}_3 = \{\mathtt{br}\}$$

The primitive operations include: the memory operations (`mk`,`get`,`set`) for allocating, dereferencing, and updating unary cells; the immutable pairing operations (`pr`, `fst`, `snd`); the usual operations for strict branching (`br`), equality on atoms (`eq`), and arithmetic; the recognizing operations (`atom`?, `cell`?, `pair`?, `lambda`?) (or characteristic functions using the booleans `t` and `nil`) of their respective domains. We also treat application, `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 λ-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:

$\mathbb{A}$	a ranges over $\mathbb{A}$
$\mathbb{X}$	x, y, z range over $\mathbb{X}$
$\mathbb{L} = \lambda\mathbb{X}.\mathbb{E}$	$\lambda x.e$ ranges over $\mathbb{L}$
$\mathbb{V} = \mathbb{X} + \mathbb{A} + \mathbb{L} + \mathtt{pr}(\mathbb{V}, \mathbb{V})$	v ranges over $\mathbb{V}$
$\mathbb{S} = \mathbb{X} \xrightarrow{f} \mathbb{V}$	σ ranges over $\mathbb{S}$
$\mathbb{E} = \mathbb{V} + \mathbb{F}_n(\mathbb{E}^n)$	e ranges over $\mathbb{E}$

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

C ranges over $\mathbb{C}$

Note that the structured data (pairs) are taken to be values. λ is a binding operator and free and bound variables of expressions are defined as usual. $\mathrm{FV}(e)$ is the set of free variables of e. A *value substitution* is a finite map σ from variables to value expressions, we let σ range over value substitutions. e^σ is the result of simultaneous substitution of free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$. We represent the function which maps x to v by $\{x := v\}$. Thus $e^{\{x:=v\}}$ is the result of replacing free occurrences of x in e by v (avoiding the capture of free variables in v). Contexts are expressions with holes. We use $\bullet$ to denote a hole. $C[e]$ denotes the result of replacing any holes in C by e. Free variables of e may become bound in this process. In order to make programs easier to read, we introduce some abbreviations. `br` is a strict conditional, and the usual conditional construct `if` can be considered an abbreviation following Landin [14]. `let` and `seq` are the usual syntactic sugar, `seq` being a sequencing primitive.

2.2 The Semantics of Terms

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

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

R ranges over $\mathbb{R}$.

The key property is that an arbitrary expression is either a value expression, or *decomposes uniquely* into a redex placed in a reduction context. We represent the state of memory using memory contexts. A memory context Γ is a context of the form

$$\begin{array}{l}
\mathtt{let}\{z_1 := \mathtt{mk}(\mathtt{nil})\} \\
\quad \ddots \quad \ddots \quad \ddots \\
\qquad \mathtt{let}\{z_n := \mathtt{mk}(\mathtt{nil})\} \\
\qquad\quad \mathtt{seq}(\mathtt{set}(z_1, v_1), \\
\qquad\qquad \vdots \quad \vdots \quad \vdots \\
\qquad\qquad \mathtt{set}(z_n, v_n), \\
\qquad\qquad \bullet)
\end{array}$$

where $z_i \neq z_j$ when $i \neq j$ (and if $n = 0$, then $\Gamma = \bullet$). We have divided the context into allocation, followed by assignment to allow for the construction of cycles. Thus, any state of memory is constructible by such an expression. We let Γ range over memory contexts. We can view memory contexts as *finite maps from variables to value expressions.* Thus we refer to their domain, $\mathrm{Dom}(\Gamma)$; modify them, $\Gamma\{z := \mathtt{mk}(v)\}$, when $z \in \mathrm{Dom}(\Gamma)$; and extend them, $\Gamma\{z := \mathtt{mk}(v)\}$, when $z \notin \mathrm{Dom}(\Gamma)$.

A *description* is a *pair*, $\Gamma; e$, with first component a memory context and second component an arbitrary expression. *Value descriptions* are descriptions whose expression is a value expression, $\Gamma; v$. The reduction relation $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$ (defined in [12]). The interesting clauses are:

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

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

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

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

A closed description, $\Gamma; e$ is *defined* (written $\downarrow \Gamma; e$) if it reduces 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 $\Gamma; e_0 \sim \Gamma; e_1$) if they are both undefined or have a common reduct (i.e. they both reduce to a particular description). Note that reduction is functional modulo α-conversion.

Two expressions are *operationally equivalent*, written $e_0 \cong e_1$, if for any closing context C, $C[e_0]$ is defined iff $C[e_1]$ is defined. In general it is very difficult to establish the operational equivalence of expressions. Thus it is desirable to have a simpler characterization of $\cong$, one that limits the class of contexts (or observations) that must be considered. A generalization of Milner's context lemma [24] provides the desired characterization. This theorem is the key to giving a semantics to VTLoE formulas.

Theorem (Generalized Context Lemma [18, 12]): $e_0 \cong e_1$ iff $\downarrow \Gamma[R[e_0^\sigma]] \Leftrightarrow \downarrow \Gamma[R[e_1^\sigma]]$ for all Γ, σ, R satisfying $\mathrm{FV}(\Gamma[R[e_i^\sigma]]) = \emptyset$ for $i < 2$.

2.3 The Encoding of Terms in Isabelle

VTLoE is a partial term logic with variables ranging over values and the central syntactic categories are those of terms $\mathbb{E}$ and values $\mathbb{V}$. The syntax has been simplified accordingly for the purpose of the encoding.

Values $\mathbb{V}$ can be viewed as a subset of terms $\mathbb{E}$ but Isabelle's type system does not support subtyping. The solution used in the current encoding is to declare a type of terms i corresponding to $\mathbb{E}$ and a subtype judgement Val expressing that a term is a value:

$\mathsf{i} :: \mathsf{logic} \qquad \mathsf{Val} :: \mathsf{i} \Rightarrow \mathsf{prop}$

Alternatively a separate type corresponding to $\mathbb{V}$ could be declared together with an injection function from values to terms. This would get rid of the extra judgement at the expense of a slightly more complicated syntax and an extra reduction axiom for pairs.

A new constant is declared for each term constructor of VTLoE. They have the following types:

nil, t	::	i
$\mathsf{mk}, \mathsf{get}, \mathsf{fst}$	::	$\mathsf{i} \Rightarrow \mathsf{i}$
$\mathsf{snd}, \mathsf{atom?}, \mathsf{cell?}, \mathsf{lamb?}, \mathsf{pair?}$	::	$\mathsf{i} \Rightarrow \mathsf{i}$
$\mathsf{app}, \mathsf{set}, \mathsf{pr}$	::	$[\mathsf{i}, \mathsf{i}] \Rightarrow \mathsf{i}$
br	::	$[\mathsf{i}, \mathsf{i}, \mathsf{i}] \Rightarrow \mathsf{i}$
lam'	::	$(\mathsf{i} \Rightarrow \mathsf{i}) \Rightarrow \mathsf{i}$

The variable binding term constructor lam' is handled using higher-order syntax, identifying object-level variables with meta-level variables. This eliminates the need for a separate type corresponding to $\mathbb{X}$.

The rules for Val formalize which terms belong to the subtype of values. There are four rules, one for each of the four kinds of values:

$$\mathsf{Val}(\mathsf{nil}) \qquad \mathsf{Val}(\mathsf{t}) \qquad \mathsf{Val}(\mathsf{lam}'(f))$$

$$\frac{\mathsf{Val}(a) \quad \mathsf{Val}(b)}{\mathsf{Val}(\mathsf{pr}(a, b))}$$

A number of term constructors, such as lets and ifs, are defined in terms of the basic ones. Having declared the necessary constants, let' and if, these are defined using Isabelle's meta-equality, for example:

$$\mathsf{let}'(e, f) \equiv \mathsf{app}(\mathsf{lam}'(f), e)$$

Finally Isabelle's translation mechanism is used to introduce external syntax. Only the translations for lam and let are shown below:

$$\mathsf{lam}\ x.e \quad \rightleftharpoons \quad \mathsf{lam}'(\lambda_{\mathcal{I}} x.e)$$
$$\mathsf{let}\{x := e_0\}e_1 \quad \rightleftharpoons \quad \mathsf{let}'(e_0, \lambda_{\mathcal{I}} x.e_1)$$

3 The Formulas of VTLoE

3.1 The Syntax of Formulas

The first-order fragment of our logic is a minor generalization of classical first-order logic. The atomic formulas assert the equivaluedness and operational equivalence of expressions. In addition to the usual first-order formula constructions, we add a $\mathtt{let}$-formula (called a contextual assertion in [12]): if Φ is a formula, x a variable, and e an expression then $\mathtt{let}\{x := e\}[\![\Phi]\!]$ is a formula.

Definition ($\mathbb{W}$):

$$\mathbb{W} = (\mathbb{E} \sim \mathbb{E}) + (\mathbb{E} \cong \mathbb{E}) + (\mathbb{W} \Rightarrow \mathbb{W}) \;+\; (\mathtt{let}\{\mathbb{X} := \mathbb{E}\}[\![\mathbb{W}]\!]) + (\forall \mathbb{X})(\mathbb{W})$$

3.2 The Semantics of Formulas

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

Definition ($\Gamma \models \Phi[\sigma]$): Assume $\Gamma, \sigma, \Phi, e_j$ are such that $\mathrm{FV}(\Phi^\sigma) \cup \mathrm{FV}(e_j^\sigma) \subseteq \mathrm{Dom}(\Gamma)$ for $j < 2$. Then we define the satisfaction relation $\Gamma \models \Phi[\sigma]$ by induction on the structure of Φ. As is usual in logic we also define the subsidiary notions of validity and logical consequence.

$$\Gamma \models (e_0 \sim e_1)[\sigma] \quad \text{iff} \quad \Gamma; e_0^\sigma \sim \Gamma; e_1^\sigma$$

$$\Gamma \models (e_0 \cong e_1)[\sigma] \quad \text{iff}$$
$$(\forall R \mid \mathrm{FV}(R) \subseteq \mathrm{Dom}(\Gamma))(\Gamma[R[e_0^\sigma]] \updownarrow \Gamma[R[e_1^\sigma]])$$

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

$$\Gamma \models \mathtt{let}\{x := e\}[\![\Phi]\!][\sigma] \quad \text{iff}$$
$$(\Gamma; e^\sigma \overset{*}{\mapsto} \Gamma'; v) \quad \text{implies} \quad \Gamma' \models \Phi[\sigma\{x := v\}]$$

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

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

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

Negation is definable, $\neg\Phi$ is just $\Phi \Rightarrow$ `False`, where `False` is any unsatisfiable assertion, such as `t` $\cong$ `nil`. Similarly conjunction, $\wedge$, and disjunction, $\vee$ and the biconditional, $\Leftrightarrow$, are all definable in the usual manner, as are termination and non-termination of terms. Note that the `let`-formula is a binding operator akin to $\forall$. We use the symbol $\simeq$ to denote either of the binary relations in our logic, $\cong$ and $\sim$.

3.3 The Encoding of Formulas in Isabelle

Formalizing the syntax of formulas in Isabelle is similar to formalizing the term syntax. A new type corresponding to the single syntactic category of formulas $\mathbb{W}$ is declared:

$$\mathsf{o} :: \mathsf{logic}$$

A new constant is declared for falsity and for each kind of formula in the original syntax. The six necessary constants are:

$$\begin{array}{llll}
\mathsf{False} :: \mathsf{o} & & =, =_o :: & [\mathsf{i}, \mathsf{i}] \Rightarrow \mathsf{o} \\
\longrightarrow \;::\; [\mathsf{o}, \mathsf{o}] \Rightarrow \mathsf{o} & & \mathsf{Let}' \;:: & [\mathsf{i}, \mathsf{i} \Rightarrow \mathsf{o}] \Rightarrow \mathsf{o} \\
\mathsf{All}' \;::\; (\mathsf{i} \Rightarrow \mathsf{o}) \Rightarrow \mathsf{o} & & &
\end{array}$$

The symbol $=$ corresponds to $\sim$ and $=_o$ corresponds to $\cong$. The variable binding constructs Let' and All' are again handled using higher-order syntax.

As before, a number of extra constants are declared and then defined in terms of the basic ones. These include constants representing the usual logical constants, connectives and quantifiers:

$$\begin{array}{lll}
\neg A & \equiv & (A \longrightarrow \mathsf{False}) \\
A \mid B & \equiv & (\neg A \longrightarrow B) \\
A \,\&\, B & \equiv & \neg(\neg A \mid \neg B) \\
\mathsf{Ex}'(A) & \equiv & \neg\mathsf{All}'(\lambda_{\mathcal{I}} x. \neg A(x))
\end{array}$$

Finally, macro translations introduce a suitable external syntax. The translations below are those for the two variable binding constructs All and Let, and the usual shorthand $\neq$:

$$\begin{array}{lll}
\mathsf{All}\; x.A & \rightleftharpoons & \mathsf{All}'(\lambda_{\mathcal{I}} x.A) \\
\mathsf{Let}\{x := e\}[\![A]\!] & \rightleftharpoons & \mathsf{Let}'(e, \lambda_{\mathcal{I}} x.A) \\
e_0 \neq e_1 & \rightleftharpoons & \neg(e_0 = e_1)
\end{array}$$

4 The Proof Theory of VTLoE

4.1 The Hilbert System

We give a Hilbert style presentation, although a natural deduction style system in the style of Prawitz [28] may in the long run be more desirable. We adopt Barendregt's convention [3] 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 variables in *contexts*.

Definition ($\vdash \Phi$): The consequence relation, $\vdash$, is the smallest relation on $\mathbb{W}$ that is closed under the rules alluded to below. For reasons of brevity we only explicitly include those that are either novel or important for the presentation.

For lack of space we only present, in figure 2, the salient features of the system, a full list may be found in [16]. The first, most basic axiom concerning operational equivalence and equivaluedness is that the booleans `t` and `nil` are not equivalent. Both $\sim$ and $\cong$ are equivalence relations, satisfy a certain restricted form of substitutivity, and are preserved under simple forms of evaluation. These last evaluation principles are (equivalent to) the let-rules of the lambda-c calculus [25]. The remaining axioms and rules concerning operational equivalence (other than that it is an equivalence relation) are: that equivaluedness implies operational equivalence; operational equivalence is preserved under the collection of garbage; equivaluedness is syntactic identity on abstractions, in contrast operational equivalence is non-trivial on abstractions (the ξ rule holds: abstractions are operationally equivalent if it is *valid* that their bodies are equivalent); and the two equivalence relations agree with one another on atoms and cells.

`let`-formulas 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. The remaining axioms concerning `let`-formulas are: (**C.ii**), `let`-formulas distribute across the equivalences; (**C.iii**), a form of `let`-formula introduction involving equivaluedness (the corresponding principle for operational equivalence is false); (**C.iv**), a principle akin to β conversion; and (**C.v**), a principle allowing for the manipulation of contexts. The propositional rules are, in addition to the usual Hilbert style

(C.i) $$\frac{\vdash \Phi}{\vdash \mathtt{let}\{x := e\}[\![\Phi]\!]}$$ **(Context Introduction)**

(C.ii) $\vdash \mathtt{let}\{x := e\}[\![e_0 \simeq e_1]\!] \Rightarrow \mathtt{let}\{x := e\}[e_0] \simeq \mathtt{let}\{x := e\}[e_1]$

(C.iii) $\vdash e_0 \sim e_1 \Rightarrow (\mathtt{let}\{x := e_0\}[\![\Phi]\!] \Leftrightarrow \mathtt{let}\{x := e_1\}[\![\Phi]\!])$

(C.iv) $\vdash \mathtt{let}\{x := v\}[\![\Phi]\!] \Leftrightarrow \Phi^{\{x:=v\}}$

(C.v) $\vdash \mathtt{let}\{x := e_0\}[\![\mathtt{let}\{y := e_1\}[\![\Phi]\!]]\!] \Leftrightarrow \mathtt{let}\{y := \mathtt{let}\{x := e_0\}e_1\}[\![\Phi]\!]$

(P.ii) $\vdash \mathtt{let}\{x := e\}[\![\Phi_0 \Rightarrow \Phi_1]\!] \Leftrightarrow (\mathtt{let}\{x := e\}[\![\Phi_0]\!] \Rightarrow \mathtt{let}\{x := e\}[\![\Phi_1]\!])$

(Q.iv) $\vdash (\forall x)\mathtt{let}\{z := \vartheta(\bar{y})\}[\![\Phi]\!] \Rightarrow \mathtt{let}\{z := \vartheta(\bar{y})\}[\![(\forall x)\Phi]\!]$ $\quad \vartheta \in \mathbb{F} - \{\mathtt{mk}, \mathtt{app}\}$

($\mathtt{mk}$.i) $\vdash \mathtt{let}\{x := \mathtt{mk}(z)\}[\![\neg(x \simeq y) \wedge \mathtt{cell?}(x) \simeq \mathtt{t} \wedge \mathtt{get}(x) \simeq z]\!]$

($\mathtt{set}$.i) $\vdash \mathtt{let}\{x := \mathtt{set}(z, y)\}[\![\mathtt{get}(z) \simeq y \wedge x \simeq \mathtt{nil}]\!]$

($\mathtt{set}$.ii) $\vdash (y \simeq \mathtt{get}(z) \wedge \neg(w \simeq z)) \Rightarrow \mathtt{let}\{x := \mathtt{set}(w, w_0)\}[\![y \simeq \mathtt{get}(z)]\!]$

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

(S.iii) $\vdash \downarrow \vartheta(\bar{y}) \Rightarrow (\mathtt{let}\{x := \vartheta(\bar{y})\}[\![z_0 \simeq z_1]\!] \Rightarrow (z_0 \simeq z_1))$ $\quad$ for $\vartheta \in \mathbb{F}$

(S.iv) $\vdash (z \simeq \vartheta_0(\bar{y})) \Rightarrow \mathtt{let}\{x := \vartheta_1(\bar{w})\}[\![z \simeq \vartheta_0(\bar{y})]\!]$ $\quad \vartheta_0, \vartheta_1 \in \mathbb{F}$

provided that if $\vartheta_1 \in \{\mathtt{set}, \mathtt{app}\}$, then $\vartheta_0 \in \mathbb{F} - \{\mathtt{get}, \mathtt{set}, \mathtt{app}\}$.

Figure 2: A Selection of Axioms and Rules

presentation of modus ponens and a generating set of tautologies, a modal axiom corresponding to **K** and its converse, (**P.ii**). Similarly the quantifier axioms are all standard [4] except for (**Q.iv**) which asserts that operations other than `mk` and `app` have no allocation effect. (**`mk`.i**) describes the allocation effect of a call to `mk`. The only other fact concerning `mk` is a pair of axioms stating that the time of allocation has no discernable effect on the resulting call. The first two `let`-formulas regarding `set` are analogous to those of (**`mk`.i**). They describe what is returned, what is altered, and 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. An important class of axioms are those which allow assertions to propagated into and out of `let`-formulas. Two typical examples are (**S.iii**) and (**S.iv**).

One important reason for introducing $\sim$ is that important principles fail for $\cong$. In particular (**C.iii**) above fails as indicated in [23]. It also allows for an adaptation of the proof of completeness theorem found in [19] to the current system. We say that an expression is *first-order* iff it contains neither unapplied λ-expressions, nor non-λ applications. A formula is *first-order* iff it is bulit up from first-order expressions. Then we have the following:

Theorem (Completeness [16]): If $\Phi \in \mathbb{W}$ is first-order and is quantifier free, then

$$\vdash \Phi \quad \text{iff} \quad \models \Phi.$$

This logic extends and improves the complete first-order system presented in [17, 19]. The system presented there had several defects. In particular the rules concerning the effects of `mk` and `set` had complicated side-conditions. Using `let`-formulas we can express them simply and elegantly. Some principles concerning the memory operations not mentioned in [12] were discovered in the process of constructing the completeness proof. The Hilbert system presented above is minimal by design in order to simplify the proof of soundness and completeness. The choice of rules is not necessarily the best if we want a basis that extends nicely or is useful in an encoding.

The axiom system presented here contains axioms and rules for quantifiers and structured data (in this case immutable pairs), however the question of whether these axioms and rules are complete remains open. We conjecture that the techniques presented in [32] can be modified and adapted to this framework to obtain an affirmative answer to this conjecture.

4.2 Proof Theory in Isabelle

Unlike the original Hilbert style formulation of the proof theory, the encoding is based on a sequent calculus style formulation. Just as in Gentzen's system LK the central judgement is that of a *sequent*:

$$\vdash_{\mathcal{S}} :: [\mathsf{sequence}, \mathsf{sequence}] \Rightarrow \mathsf{prop}$$

Associative sequences of formulas are handled as in the implementation of LK in Isabelle. The type sequence corresponds to an external syntax for associative lists where comma is the associative operator. Internally a higher-order representation handles associative unification.

Assumption rule

$$\Delta, A, \Theta \vdash_{\mathcal{S}} \Lambda, A, \Xi \quad \text{Asm}$$

Structural rules

$$\frac{\Delta, \Theta, \Theta, \Lambda \vdash_{\mathcal{S}} \Xi}{\Delta, \Theta, \Lambda \vdash_{\mathcal{S}} \Xi}\,\text{Cont}_{\mathcal{L}} \qquad \frac{\Delta \vdash_{\mathcal{S}} \Theta, \Lambda, \Lambda, \Xi}{\Delta \vdash_{\mathcal{S}} \Theta, \Lambda, \Xi}\,\text{Cont}_{\mathcal{R}}$$

$$\frac{\Delta, \Lambda \vdash_{\mathcal{S}} \Xi}{\Delta, \Theta, \Lambda \vdash_{\mathcal{S}} \Xi}\,\text{Weak}_{\mathcal{L}} \qquad \frac{\Delta \vdash_{\mathcal{S}} \Theta, \Xi}{\Delta \vdash_{\mathcal{S}} \Theta, \Lambda, \Xi}\,\text{Weak}_{\mathcal{R}}$$

$$\frac{\Delta, \Lambda, \Theta, \Xi \vdash_{\mathcal{S}} \Pi}{\Delta, \Theta, \Lambda, \Xi \vdash_{\mathcal{S}} \Pi}\,\text{Exch}_{\mathcal{L}} \qquad \frac{\Delta \vdash_{\mathcal{S}} \Theta, \Xi, \Lambda, \Pi}{\Delta \vdash_{\mathcal{S}} \Theta, \Lambda, \Xi, \Pi}\,\text{Exch}_{\mathcal{R}}$$

Cut rule

$$\frac{\Delta \vdash_{\mathcal{S}} \Theta, A \quad A, \Delta \vdash_{\mathcal{S}} \Theta}{\Delta \vdash_{\mathcal{S}} \Theta}\,\text{Cut}$$

Logical rules

$$\Delta, \mathsf{False}, \Theta \vdash_{\mathcal{S}} \Lambda \quad \mathsf{False}_{\mathcal{L}}$$

$$\frac{\Delta, \Theta \vdash_{\mathcal{S}} \Lambda, A \quad \Delta, B, \Theta \vdash_{\mathcal{S}} \Lambda}{\Delta, A \longrightarrow B, \Theta \vdash_{\mathcal{S}} \Lambda}\,\longrightarrow_{\mathcal{L}}$$

$$\frac{\Delta, A \vdash_{\mathcal{S}} \Theta, B, \Lambda}{\Delta \vdash_{\mathcal{S}} \Theta, A \longrightarrow B, \Lambda}\,\longrightarrow_{\mathcal{R}}$$

Figure 3: Basic Sequent Rules

The system consists of some basic rules as well as axioms and rules translated directly from the Hilbert system. The basic rules are an assumption rule, structural rules, a cut rule and logical rules for falsity and implication ($\mathsf{False}_{\mathcal{L}}$, $\longrightarrow_{\mathcal{L}}$ and $\longrightarrow_{\mathcal{R}}$). They are displayed in figure 3). It is essentially the usual rules although modified to be directly applicable in more cases. Some rules have also been generalized to work on sequences instead of single formulas. The result is that more rules are derivable as opposed to only admissible. The remaining axioms and rules of the axiomatic Hilbert system, not accounted by the basic rules, are translated directly into corresponding axioms and rules in the encoding. Some care must be exercised to ensure that this translation is correct. It is, for example, often necessary to restrict term variables to range over values. The rules in figure 4 correspond to (**C.i**), (**C.ii**), (**C.v**) and (mk.**i**). The subscript x expresses meta-quantification and the first rule is just $(\bigwedge x.\mathsf{Val}(x) \Longrightarrow \vdash_{\mathcal{S}} A(x)) \Longrightarrow \vdash_{\mathcal{S}} \mathsf{Let}'(e, A)$.

$$\frac{\begin{array}{c}[\mathsf{Val}(x)]_x \\ \vdots \\ \vdash_{\mathcal{S}} A(x)\end{array}}{\vdash_{\mathcal{S}} \mathsf{Let}'(e, A)}$$

$$\begin{array}{ll}\vdash_{\mathcal{S}} & \mathsf{Let}\{x := e\}[\![e_0(x) = e_1(x)]\!] \\ & \quad\longleftrightarrow \\ & \mathsf{let}'(e, e_0) = \mathsf{let}'(e, e_1)\end{array}$$

$$\begin{array}{ll}\vdash_{\mathcal{S}} & \mathsf{Let}\{x := e_0\}[\![\mathsf{Let}'(e_1(x), A)]\!] \\ & \quad\longleftrightarrow \\ & \mathsf{Let}'(\mathsf{let}'(e_0, e_1), A)\end{array}$$

$$\frac{\mathsf{Val}(y) \quad \mathsf{Val}(z)}{\begin{array}{l}\vdash_{\mathcal{S}} \mathsf{Let}\{x := \mathsf{mk}(z)\} \\ \qquad [\![x \neq y \;\&\; \mathsf{cell?}(x) = \mathsf{t} \;\&\; \mathsf{get}(x) = z]\!]\end{array}}$$

Figure 4: Translated Rules and Axioms

The encoding is flexible. It is possible to derive Hilbert, natural deduction, and sequent calculus style rules and axioms. The different reasoning styles can be mixed as desired.

Sequent calculus and natural deduction style rules for the `let`-formula should describe the *logical* aspects of the `let`-formula. Here the necessitation like rule (**C.i**) and the axiom (**P.ii**) are seen as describing the logical properties. With this classification in mind the rule $\mathsf{Let}_{\mathcal{LR}}$ in figure 5 describes the logical aspects of the `let`-formula. Using this rule it is possible to derive theorems corresponding to (**C.i**) and (**P.ii**). The ability to have several formulas on the right of $\vdash_{\mathcal{S}}$ appear to be essential and the rule feels inherently classical. It is hard to classify as either a left or right rule because it introduces `let`-formulas on both sides of $\vdash_{\mathcal{S}}$ and because it affects all the formulas in the sequent. The rule does not seem to compare directly with any common rule of modal logic. The fact that it is based on an atypical property in modal logic similar to the converse of (**K**), might help to explain this.

Introduction rules usually follow from right rules by restricting the right side of $\vdash_{\mathcal{S}}$ to just one formula. Similar elimination rules follow from left rules by cut and restriction of the right hand side. Applying these ideas to $\mathsf{Let}_{\mathcal{LR}}$ results in the rule $\mathsf{Let}_{\mathcal{IE}}$ in figure 5. An unfortunate consequence of restricting the right hand side to just one formula is that a theorem corresponding to the converse of (**K**) can no longer be derived. One solution is to add an extra rule corresponding to the converse of (**K**). Other solutions are to allow multiple formulas on the right of $\vdash_{\mathcal{S}}$ or to incorporate the problematic property into $\mathsf{Let}_{\mathcal{IE}}$ in some other way.

$$\dfrac{\begin{array}{c}[\mathsf{Val}(x)]_x\\ \vdots \\ A_1(x),\ldots,A_m(x) \vdash_{\mathcal{S}} B_1(x),\ldots,B_n(x)\end{array}}{\mathsf{Let}'(e,A_1),\ldots,\mathsf{Let}'(e,A_m) \vdash_{\mathcal{S}} \mathsf{Let}'(e,B_1),\ldots,\mathsf{Let}'(e,B_n)}\,\mathsf{Let}_{\mathcal{LR}} \quad m \geq 0, n \geq 1$$

$$\dfrac{\begin{array}{c}[\mathsf{Val}(x)]_x\\ \vdots \\ A_1(x),\ldots,A_m(x) \vdash_{\mathcal{S}} B(x)\end{array} \qquad \Delta \vdash_{\mathcal{S}} \mathsf{Let}'(e,A_1) \quad \cdots \quad \Delta \vdash_{\mathcal{S}} \mathsf{Let}'(e,A_m)}{\Delta \vdash_{\mathcal{S}} \mathsf{Let}'(e,B)}\,\mathsf{Let}_{\mathcal{IE}}$$

Figure 5: Logical `let`-Rules

Rules such as $\mathsf{Let}_{\mathcal{LR}}$ and $\mathsf{Let}_{\mathcal{IE}}$ cannot be represented directly in the encoding at present. There are no direct way of representing sequences such as $\mathsf{Let}'(e,A_1),\ldots,\mathsf{Let}'(e,A_i)$ or having rules with a variable number of premises such as $\mathsf{Let}_{\mathcal{IE}}$. At a later stage it might be possible to express such rules using extra judgements. Even if they were expressible they would not be derivable but only admissible in the current formulation. The reason is that they are proved by induction on the length of sequences of a sequent. Everything is however not lost: it is possible to derive each concrete instance.

5 Conclusions, Current and Future Work

Thus far we have described the current state of the implementation and axiomatization of VTLoE, it is not the final product. We now spend some time describing the future development of the implementation into a hopefully practical system.

5.1 Elaborating on the Encoding

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 [22, 1]. Since the semantics of the underlying $\lambda_{\mathtt{mk}}$-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 [33]. It will also allow the system to semantically verify its own proof system, an extremely attractive idea. It would also 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 encoding 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 [30, 31].

5.2 Tactics, Strategies and Rewriting in VTLoE

VTLoE is a computational logic and one obvious area of application is formal development of $\lambda_{\mathtt{mk}}$ programs. This section describes one particular unsolved problem encountered when trying to develop $\lambda_{\mathtt{mk}}$ programs formally using the encoding. The problem is illustrated using a very simple program development example. Although the goal is to use the encoding to *derive* $\lambda_{\mathtt{mk}}$ programs only the simpler *verification* problem will be considered here.

The `let`-formula of VTLoE can be viewed as a generalisation of Hoare triples and can be used to express program correctness in a similar way. The following correctness theorem specify that the program $\mathsf{swap}(x,y)$ swaps the contents of two cells x and y:

$$\begin{array}{ll}\vdash_{\mathcal{S}} & \mathsf{All}\ x\ y\ a\ b.\mathsf{get}(x) = a\ \&\ \mathsf{get}(y) = b \\ & \qquad \longrightarrow \\ & \mathsf{Seq}(\mathsf{swap}(x,y), [\![\mathsf{get}(x) = b\ \&\ \mathsf{get}(y) = a]\!])\end{array}$$

where

$$\begin{array}{l}\mathsf{swap}(x,y) \equiv \mathsf{let}\{c := \mathsf{get}(x)\} \\ \qquad\qquad\qquad \mathsf{seq}(\mathsf{set}(x,\mathsf{get}(y)),\mathsf{set}(y,c))\end{array}$$

The term `seq` and the formula Seq are just the corresponding `let`-term and `let`-formula where the bound variable does not occur in the body. The formula to the left of $\longrightarrow$ is referred to as the *pre-condition* and the body of the `let`-formula as the *post-condition*. The correctness theorem does not specify the program completely. A typical correctness theorem will also be concerned with termination, the value returned by the program and exclude certain effects on the store.

The post-condition generally contains much more information about what the program should do than the pre-condition. A program is therefore typically developed by working backwards from the post-condition towards the pre-condition. In other words, the problem is to eliminate the outermost `let`-formula on the right hand side of $\longrightarrow$ and prove the result from the pre-condition.

The problem of eliminating a *complex* `let`-formula can usually be simplified to a problem of eliminating *simple* `let`-formulas. A `let`-formula is classified depending on the program it takes as its first argument. A *simple* expression is a value or an operation applied to values. All other expressions are complex. It is possible to derive decomposition rules for complex `let`-formulas such as:

$$\frac{\begin{array}{c}[\mathsf{Val}(a)]_a \\ \vdots \\ A(a) \vdash_{\mathcal{S}} \mathsf{Let}'(e_1(a), B)\end{array} \qquad \Delta \vdash_{\mathcal{S}} \mathsf{Let}'(e_0, A)}{\Delta \vdash_{\mathcal{S}} \mathsf{Let}'(\mathsf{let}'(e_0, e_1), B)}$$

This rule is similar to the rule for sequential composition in Hoare logic, except that `let`-formulas also bind values.

Having applied such decomposition rules repeatedly in the swap case the first subproblem to be solved is to find a suitable $?A(x, y, a, b, c)$ such that:

$$?A(x,y,a,b,c) \vdash_{\mathcal{S}} \mathsf{Seq}(\mathsf{set}(x,c), \llbracket \mathsf{get}(x) = b \;\&\; \mathsf{get}(y) = a \rrbracket)$$

In Isabelle a question mark is used to indicate that a variable is a *schematic variable*, that is a free variable which can be instantiated during unification. The effect of the `set`-operation on the `get`-operation of the first conjunct is obvious. In the second case it is necessary to consider two cases depending on whether x and y are aliases or not. A suitable $?A(x, y, a, b, c)$ is:

$$c = b \;\&\; (x = y \longrightarrow c = a) \;\&\; (x \neq y \longrightarrow \mathsf{get}(y) = a)$$

Although it is relatively simple to handle the problem of eliminating simple `let`-formulas in the swap example it is much harder in general. It is particular hard for expressions which might interact with the store such as `set`, `get`, `mk`, `app`. Imagine what happens when the programs in the post-condition are complex expressions with read, write or allocation effects. The possibility of recursion add further complications.

As indicated above, an unsolved problem is to find a systematic method for eliminating simple `let`-formulas which is also sufficient general. In the `set` case such a method should work as a generalized Hoare style assignment rule and should be encoded in Isabelle as a tactic. This would allow proof steps of the same granularity as in Hoare logic. This is a minimum requirement if the system is to be used for practical program development.

Our investigations so far suggest that the problem of eliminating `let`-formulas are closely connected with equational reasoning in VTLoE. Facts such as (**C.ii**) and the following point in that direction:

$$\begin{array}{l}\vdash \mathtt{let}\{a := \vartheta(\bar{b})\}e_0 \sim \mathtt{let}\{a := \vartheta(\bar{b})\}e_1 \\ \quad \Rightarrow \\ \mathtt{let}\{a := \vartheta(\bar{b})\}\llbracket e_0 \sim e_1 \rrbracket \\ \quad \text{for } \vartheta \in \mathbb{F} - \{\mathtt{mk}, \mathtt{app}\}\end{array}$$

$$\begin{array}{l}\vdash \begin{array}{l}\mathtt{let}\{x := \mathtt{mk}(a)\} \\ \;\mathtt{pr}(e_0, x)\end{array} \sim \begin{array}{l}\mathtt{let}\{x := \mathtt{mk}(a)\} \\ \;\mathtt{pr}(e_1, x)\end{array} \\ \quad \Rightarrow \\ \mathtt{let}\{x := \mathtt{mk}(a)\}\llbracket e_0 \sim e_1 \rrbracket\end{array}$$

Rewriting is a very useful technique in theorem proving. It is usually based on transitivity and congruence properties of the relation in question. Both $\sim$ and $\cong$ is transitive, but neither is a congruence in the strong sense:

$$\vdash e_0 \simeq e_1 \;\Rightarrow\; C(e_0) \simeq C(e_1) \quad C \in \mathbb{C}$$

This is easy to see by considering the example:

$$\begin{array}{l}\vdash \mathtt{get}(x) \sim \mathtt{nil} \\ \quad \Rightarrow \\ \mathtt{seq}(\mathtt{set}(x, \mathtt{t}), \mathtt{get}(x)) \sim \mathtt{seq}(\mathtt{set}(x, \mathtt{t}), \mathtt{nil})\end{array}$$

which is clearly false. It is however possible to derive rules as the following:

$$\frac{\Delta \vdash_{\mathcal{S}} e_0 = e_0' \quad \Delta \vdash_{\mathcal{S}} \mathsf{Seq}(e_0', \llbracket e_1 = e_1' \rrbracket)}{\Delta \vdash_{\mathcal{S}} \mathsf{set}(e_0, e_1) = \mathsf{set}(e_0', e_1')}$$

Rewriting based on such rules once again illustrates how important it is to be able to reason systematically about `let`-formulas. The sequence Δ will typically consists of equations which should be used for rewriting. To rewrite the second argument e_1 of the `set` operation it is necessary to push the equations in Δ though e_0', ie. finding a suitable $?A$ such that $\Delta \vdash_{\mathcal{S}} \mathsf{Seq}(e_0', ?A)$. One obvious application of rewriting in VTLoE is symbolic evaluation of programs.

Acknowledgments

The work reported here was carried out while the first author was visiting the Computer Laboratory at the University of Cambridge and supported by the ESPRIT Basic Research Project 6453 “Types for Proofs and Programs” and the Danish Natural Science Research Council. The first author thanks Lawrence C. Paulson for supporting the work and giving substantial useful advice, as well as Søren T. Heilmann for reading earlier versions of this paper. Both authors would like to thank Carolyn Talcott for inspiration and numerous helpful discussions.

References

[1] A.Avron, F.Honsell, I.A. Mason and Robert Pollack. Using Typed Lambda Calculus to Implement Formal Systems on a Machine. *Journal of Automated Reasoning*, Volume 9, pages 309–354, 1992.

[2] K.R. Apt. Ten years of Hoare's logic: A survey–part I. *ACM Transactions on Programming Languages and Systems*, Volume 4, pages 431–483, 1981.

[3] H. Barendregt. *The lambda calculus: its syntax and semantics*. North-Holland, 1981.

[4] C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, Amsterdam, 1973.

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

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

[7] M. Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. Ph.D. thesis, Indiana University, 1987.

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

[9] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, Volume 103, pages 235–271, 1992.

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

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

[12] F. Honsell, I. A. Mason, S. F. Smith and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, Volume 119, Number 1, pages 55–90, May 1995.

[13] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *SIGPLAN*, Volume 27, Number 5, May 1992.

[14] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, Volume 6, pages 308–320, 1964.

[15] I. A. Mason. *The Semantics of Destructive Lisp*. Ph.D. thesis, Stanford University, 1986. Also available as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

[16] I A. Mason. A Logic of Effects, 1995. Available either via anonymous ftp from leven.appcomp.utas.edu.au (pub/imason) or via URL:http://www.appcomp.utas.edu.au:/users/imason.

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

[18] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, Volume 1, pages 287–327, 1991.

[19] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, Volume 105, Number 2, pages 167–215, 1992.

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

[21] I. A. Mason and C. L. Talcott. Program transformation via contextual assertions. In *Logic, Language and Computation. Festschrift in Honor of Satoru Takasu*, Volume 792 of *Lecture Notes in Computer Science*, pages 225–254. Springer, Berlin, 1994.

[22] I.A. Mason. Hoare's Logic in the LF. Technical Report ECS-LFCS-87-32, Laboratory for foundations of computer science, University of Edinburgh, 1987.

[23] I.A. Mason and C.L. Talcott. Reasoning about Object Systems in VTLoE. *International Journal of Foundations on Computer Science*, Volume 6, Number 3, pages 265–298, 1995.

[24] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, Volume 4, pages 1–22, 1977.

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

[26] Lawrence C. Paulson. *Isabelle, A Generic Theorem Prover*. Number 82 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.

[27] A. M. Pitts. Evaluation logic. In *IVth Higher-Order Workshop, Banff*, Volume 283 of *Workshops in Computing*. Springer-Verlag, 1990.

[28] D. Prawitz. *Natural Deduction: A Proof-theoretical Study*. Almquist and Wiksell, 1965.

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

[30] C. L. Talcott. Binding structures. In Vladimir Lifschitz (editor), *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.

[31] C. L. Talcott. A theory of binding structures and its applications to rewriting. *Theoretical Computer Science*, Volume 112, pages 99–143, 1993.

[32] R. L. Tenney. *Decidable Pairing Functions*. Ph.D. thesis, Cornell University, 1972. also appears as Computer Science TR 72-136.

[33] R. W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, Volume 13, pages 133–170, 1980.