Establishing a General Context Lemma in PVS

Jonathan Ford & Ian A. Mason
School of Mathematics, Statistics, & Computing Science, U.N.E
Armidale, NSW 2351 Australia
{jford, iam} @turing.une.edu.au

In theory there is no difference between theory and practice; but in practice there is. . . . Anon.

1 Introduction

In this paper we report on the results of a sophisticated and substantial use of PVS to establish a recent result in operational semantics. This is of interest not only because it requires the substantial development of current higher order techniques in operational semantics, but also because it exposed several gaps in the published presentation of the result. Thus this paper exemplifies the possible benefit of serious formalization in contrast to standard mathematical practice which typically leaves much unsaid. We also take great pains to formalize the actual theoretical treatment, rather than adapting it to the tastes of either the machine, and PVS. In this regard we were almost completely successful, only on two occasions was it necessary to deviate, slightly, from the exact formal treatment. We will mention them in the narrative. In this sense we made no use of the *tricky* representations that McKinna and Pollack discuss [19]. In our earlier work [7, 5] we carried out in PVS, for the first time, a *named variable proof* of the Church–Rosser result for Landin's call-by-value Iswim, without eliminating α congruence by tricky encodings. Thus our work can be seen as an attempt to reconcile theory, with formal verification in practice. We should also point out that prior to this earlier work very little use of PVS's inductive abstract data types had been made. Thus this work also represents the first use of these aspects of PVS to verify a non-trivial recent result, as opposed to a classic result that has been used somewhat as a benchmark, see [7] for a survey.

Thus this paper has several, hopefully complementary, purposes. On the one hand it is a detailed presentation of the proof of the CIU theorem for uniform λ-languages, on the other hand it is a road map for the actual mechanized proof [6], and the issued raised in its development. We also try and address some of the issues that are raised in presenting both a theoretical and the corresponding formal development. We will use the word *theoretical* to refer to the treatment of the subject matter as it normally appears in journal publications such as [24, 17], to contrast it with the word *formal* that refers to the analagous notion, as formalized in the corresponding PVS development. Also in the body of the paper we use the notation `file`:<line number> to refer to the particular line in the unpacked file, whose name is `file.pvs`, of [6]. So for example `CIU`:210 is the actual statement of the main theorem presented in this paper. We also extend this notation to include the name of the theorem, lemma, or definition when this is of interest. Thus `CIU` : `CIU`:210 refers to the theorem `CIU` in the file `CIU.pvs` that lies on line 210 of the unpacked version of [6]. This system of presentation works well for the statements of the results contained in the development, but not for the formal proofs. An area of PVS that needs more attention.

1.1 Historical Background

Much work has been done to develop methods for reasoning about operational approximation and equivalence. Methods developed for reasoning about operational approximation and equivalence include: general schemes for establishing equivalence; context lemmas (alternative characterizations that reduce the number of contexts to be considered); and (bi)simulation relations (alternative characterizations or approximations based on co-inductively defined relations). An early example is Robin Milner's context lemma [20] which greatly simplifies the proof of operational equivalence in the case of the typed λ calculus by reducing the contexts to be considered to a simple chain of applications. Carolyn Talcott [23] studies general notions of equivalence for languages based on the call-by-value λ calculus, and develops several schemes for establishing properties of such relations. Doug Howe [9] develops a schema for proving congruence for a class of languages with a particular style of operational semantics. This schema succeeds in capturing many simple functional programming language features. Building on this work, Howe [10] uses an approach similar to the idea of uniform computation to define structured evaluation systems in which the form of the evaluation rules guarantees that (bi)simulation relations are congruences. The form of the rules is specified using meta variables with arities and higher-order substitutions. This syntax enrichment is very similar to the notions of place-holder and filling used here to specify uniform semantics. The idea of using such meta terms to specify classes of rules giving rise to reduction relations with special properties was used by Peter

Aczel in [1] to prove a general Church-Rosser theorem and in Klop [11] to develop the theory of Combinatory Reduction Systems. Meta terms are also used in describing a unification procedure for higher-order patterns by Tobias Nipkow in [21]. Mason and Talcott [15, 16] introduced the CIU characterization of operational equivalence which is a form of context lemma for imperative languages. This lemma was then generalized by Carolyn Talcott to a very wide class of programming languages in [24]. It is this form of the lemma that we concentrate on in this paper.

Notation

We conclude the introduction with a summary of our notation conventions. Let X, X_0, X_1 be sets. We specify meta-variable conventions in the form: let x range over X, which should be read as: the meta-variable x and decorated variants such as x', $x_0, \ldots$, range over the set X. $\mathbf{P}_\omega(X)$ is the set of finite subsets of X. $\mathrm{Fmap}[X_0, X_1]$ is the set of finite maps from X_0 to X_1. To emphasize application as an binary operation, and to unify syntax, we often write $\mathtt{app}(f, x)$ for the application $f(x)$ of the function f to the argument x. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. Thus if $f \in \mathrm{Fmap}[X_0, X_1]$, then $\mathrm{Dom}(f) \in \mathbf{P}_\omega(X_0)$. For any function f, $f\{x \mapsto x'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{x\}$, $f'(x) = x'$, and $f'(z) = f(z)$ for $z \neq x, z \in \mathrm{Dom}(f)$. Also $f\lceil X$ is the restriction of f to X: the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cap X$ and $f'(x) = f(x)$ for $x \in \mathrm{Dom}(f')$. $\mathbf{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers and $i, j, n, n_0, \ldots$ range over $\mathbf{N}$. In the defining equations for various syntactic classes we use two notational conventions: pointwise lifting of syntax operations to syntax classes; and the Einstein summation convention that a phrase of the form $F_n(Z^n)$ abbreviates $\bigcup_{n\in\mathbf{N}} F_n(Z^n)$. For example if Ω is a ranked set of operator symbols, then the terms over Ω can be defined inductively by (as the least solution to) the equation: $T_\Omega = \Omega_n(T_\Omega^n)$. Unabbreviated, this equation reads:

$$T_\Omega = \bigcup_{n\in\mathbf{N}} \{\omega(t_1, \ldots, t_n) \mid \omega \in \Omega_n \wedge t_i \in T_\Omega \quad \text{for} \quad 1 \leq i \leq n\}.$$

Finally we use the traditional  symbol to indicate where the formal development uncovered gaps in the theoretical development.

2 Syntax of Terms

2.1 Background

In this section we present both the general framework as described in [24], and more recently in [17]. Concurrently we will describe how these notions are interpreted into PVS, as well as any interesting observations concerning their formalization.

The operational theory is a small-step operational semantics, and is obtained by defining a notion of state and a single step reduction relation on states. States consist of an expression and a state context. A state context often describes dynamically created entities such as memory cells, arrays, files, etc. The form of state contexts needed depends on the choice of primitive operations. There is an empty state context, and for each state there is an associated expression representing that state. Value expressions are a subset of the set of expressions used to represent semantic values. These include variables, atoms, and λs. If the expression component of a state is a value, then the state is a value state and no reduction steps are possible. Otherwise, the expression decomposes uniquely into a redex placed in a reduction context. A (call-by-value) redex is a primitive operator applied to a list of values. There is one reduction rule for each primitive operator, and the single-step reduction relation on states is determined by the reduction rule for the redex operator. Of course it may happen that a redex is ill-formed (a runtime error) and no reduction step is possible. A state is defined just if it reduces (in a finite number of steps) to a value state. Using these basic notions we define the operational approximation and equivalence relations in the usual way in terms of definedness in all program contexts. This is the basic semantic framework, independent of the choice of primitive operations. Within this framework we define the notion of uniform semantics.

The uniformity requirements are that each reduction rule hold not only for traditional expressions, but also for expressions containing parameters or meta variables. This parametric notion of computation is best presented using the idea of a context and the treatment here follows the general theory presented in [13].

2.2 The Syntax of Expressions

A particular λ language requires a set of atoms, and a set of operations, to define the syntax. It then requires the specification of the values and states. These however are just suitably uniform subsets of the basic syntax.

2.2.1 The *Theoretical* Treatment of Syntax

Fix two disjoint countably infinite sets, $\mathbf{X}$, of variables, and $\mathbf{P}$ of parameters:

$$\mathbf{X} = \{x_i \mid i \in \mathbf{N}\} \qquad \mathbf{P} = \{X_i \mid i \in \mathbf{N}\}$$

The basic syntax of a λ-language is then determined by specifying three sets: a countable set of atoms, $\mathbf{A}$, disjoint from $\mathbf{X}$ and $\mathbf{P}$; a family of operation symbols $\mathbf{O} = \{\mathbf{O}_n \mid n \in \mathbf{N}\}$ ($\mathbf{O}_n$ is a set of n-ary operation symbols) disjoint from $\mathbf{X} \cup \mathbf{A} \cup \mathbf{P}$; and the set of value expressions, a subset of expressions, $\mathbf{V}$, that we will specify in more detail immediately after the definition of the syntax.

We assume that $\mathbf{O}$ contains at least the binary operation `app` (lambda application). By taking $\mathbf{A} = \{\ \}$ and $\mathbf{O} = \{\mathtt{app}\}$ we obtain the expressions of the pure call-by-value λ calculus, Λ_v. Examples of ML style references and escape operators such as the Scheme `call/cc` can be found in [17].

Definition 2.1 (E, L, S, F): $\mathtt{FLL-version-03}$:1–160
The set of expressions, $\mathbf{E}$, and the set of λ-abstractions, $\mathbf{L}$, the set of value substitutions, $\mathbf{S}$, and the set of parameter substitutions (fillings), $\mathbf{F}$ are defined as the least sets satisfying the following equations:

$$\mathbf{E} = \mathbf{X} \cup \mathbf{P}^{\mathbf{S}} \cup \mathbf{A} \cup \mathbf{L} \cup \mathbf{O}_n(\mathbf{E}^n) \quad \mathbf{L} = \lambda\mathbf{X}.\mathbf{E} \quad \mathbf{S} = \mathrm{Fmap}[\mathbf{X}, \mathbf{V}] \quad \mathbf{F} = \mathrm{Fmap}[\mathbf{P}, \mathbf{E}]$$

We let a range over $\mathbf{A}$, x, y, z range over $\mathbf{X}$, X, Y, Z range over $\mathbf{P}$, e range over $\mathbf{E}$, φ range over $\mathbf{L}$, σ range over $\mathbf{S}$, and Σ range over $\mathbf{F}$.

$\mathbf{P}^{\mathbf{S}}$ is the set of parameters, annotated or decorated by value substitutions. Value substitutions, $\mathbf{S}$, are finite maps from variables to value expressions. The domain of a substitution is written as $\mathrm{Dom}(\sigma)$, and is defined in the usual way. Parameter substitutions are finite maps from parameters to expressions. We write $\{x_i \mapsto v_i \mid i < n\}$ for the value substitution, σ, with domain $\{x_i \mid i < n\}$ such that $\sigma(x_i) = v_i$ for $i < n$. Similarly we write $\{X_i \mapsto e_i \mid i < n\}$ for the parameter substitution, Σ, with domain $\{X_i \mid i < n\}$ such that $\Sigma(X_i) = e_i$ for $i < n$. Note that a parameter decorated by a value substitution is an expression. This allows us to compute parametrically with partially specified expressions, and thus our expressions generalize the usual notion of context. In this more general setting we must be somewhat more careful to define certain basic notions.

2.2.2 The *Formal* Treatment of Syntax

In keeping with our attempt to be faithful to the theoretical treatment, we use finite maps in PVS to represent the finite maps σ and Σ. Also the entire development is parametric in a set of atoms and operations (which includes the binary `app`, together with the arities of those operations. This theory is appropriately named `Landin`, and can be found in $\mathtt{FLL-version-03}$:11 Implementation of the syntax within PVS is a straightforward use of the *datatype with subtypes* facility. There are really only two technical issues that arise, and one design issue. We will look at the technical issues first, then discuss the design issue.

The first technical issue arises because the set of variables are naturally a subtype of the set of expressions. However they appear negatively in the domain (to the finite set constructor) of annotating substitution, and thus some way of avoiding this must be found. The solution is relatively simple, and relatively painless in that it does not cause a great divergence between the theoretical treatment and the formal one. We annotate parameters by maps from finite sets of natirual numbers rather than variables. We then make use of the natural identification of a variable and the index (in this case a natural number) from which it was contructed.

The second technical issue is how to deal with the argument lists to the operations in the language. On the advice of Shankar we adopted the style of having a separate subtype for lists of expressions, and explicitly recursed on this structure in all our defintions. This provides the PVS typechecker with additional information, that it would not normally be entitled too, and as a result substantially reduces the number of TCC's generated. We should point out though, that adopting this strategy means that subsequent syntactic notions will reflect this minor difference.

Finally we made a design decision to fully develop the syntax prior to the introduction of the notion of values. There are obvious and not so obvious reasons for this. The obvious is that by ommiting the value restrictions we are developing a more general framework that one day may be put to good use. The not so obvious is that by adding the restriction, at this stage, that the range of annotating substitutions be restricted to values would substantially complicate the development of the basic syntatic operations such as renaming, substitution, and filling, especially in the generated TCCs. Thus we chose to ignore values at this stage, and only later in the development (`Values` :19) define them by recursion, as a subtype of expressions with the desired feature.

2.2.3 Auxilliary Syntactic Notions

One of the real differences between theoretical and formal treatments now takes place. What can be glossed over in a page or two in a theoretical treatment, can now take several person weeks and significant amounts of patience and ingenuity.

Given the above implementation of the basic syntax, we must now develop the more basic auxilliary notions that are glossed over rather tersely in any theoretical treatment. We must define the rank of an expression, so as to be able to define by induction and recursion more complex notions. We must define derived notions such as the free variables of an expression, and the set of parameters that occur in it. These sets will be finite, and this fact itself must be verified, usually as TCCs.

We must define substitution, which itself entails developing the simpler notion of renaming. Filling similarly must be defined, as must the notion of being equivalent modulo the renaming of bound variable (α equivalence). We must then also show that, the simple as well as, these important operations are functional modulo this equivalence relation. As well as establishing that α equivalence is a congruence, and that it has all the properties that we assume it to have (i.e. it is indeed a congruence: an equivalence relation preserved by the syntactic constructions).

2.2.4 Rank

The rank of an expression is defined using the `reduce_nat` facilty that is automatically generated from the representation of the syntax using PVS's datatype with subtypes facility. Because of the particular choices made in implementing the syntax, the theoretical and practical notions of rank do not coincide. The actual rank we implement is:

Definition 2.2 ($\mathrm{rank}(e)$**):** `Rank` :1–87

$$\mathrm{rank}(e) = \begin{cases} 0 & \text{if } e \in \mathbf{X} \cup \mathbf{A}, \\ 1 + \mathrm{rank}(e_0) & \text{if } e = \lambda x.e_0, \\ 2 + \max(\{\mathrm{rank}(e_i) \mid 1 \leq i \leq n\}) + n & \text{if } e = \vartheta(e_1, \ldots, e_n), \\ 1 + \max(\{\mathrm{rank}(\sigma(x)) \mid x \in \mathrm{Dom}(\sigma)\}) & \text{if } e = X^\sigma \end{cases}$$

2.2.5 Derived Syntactic Notions

The first notion needed in the formal development are the set of variables, free variables, and parameters that occur in an expression. They are all simple recursive definitions, and in the actual formal treatment are sets of natural numbers. Thus they are the indexes of the variables, free variables, and parameters respectively. We must also establish rather obvious facts, such as that the free variables are a subset of the variables.

Definition 2.3 ($\mathrm{FV}(e)$**,** $\mathrm{FP}(e)$**):** `Vars` :1-95

The free variables, $\mathrm{FV}(e)$, and the parameters, $\mathrm{FP}(e)$, (which are always free) of an expression e are defined inductively. The novel clauses are:

$$\mathrm{FV}(X_i^\sigma) = \bigcup_{x \in \mathrm{Dom}(\sigma)} \mathrm{FV}(\sigma(x)) \qquad \mathrm{FP}(X_i^\sigma) = \bigcup_{x \in \mathrm{Dom}(\sigma)} \mathrm{FP}(\sigma(x)) \cup \{i\}$$

The free variables $\mathrm{FV}(e)$ are defined in `Vars` :14 , while the parameters, $\mathrm{FP}(e)$, are defined in `Vars` :38 . We extend these to substitutions is the obvious fashion:

$$\mathrm{FV}(\Delta) = \bigcup_{\delta \in \mathrm{Dom}(\Delta)} \mathrm{FV}(\Delta(\delta)) \qquad \mathrm{FP}(\Delta) = \bigcup_{\delta \in \mathrm{Dom}(\Delta)} \mathrm{FP}(\Delta(\delta)) \qquad \text{for } \Delta \in \mathbf{S} \cup \mathbf{F}.$$

One last derived piece of notation concerning annotated parameters appearing in expressions. The set of *trapped* variables, $\mathrm{Traps}(e)$, is defined to be the smallest set of variables that contains the domains of any substitution that annotates an occurrence of a parameter in e.

Definition 2.4 ($\mathrm{Traps}(e)$**):** `Traps` :1-35

These amount to a simple inductive definition, the interesting clause being:

$$\mathrm{Traps}(X^\sigma) = \bigcup_{x \in \mathrm{Dom}(\sigma)} \mathrm{Traps}(\sigma(x)) \cup \mathrm{Dom}(\sigma) \qquad \texttt{Traps} :20$$

The set of traps, like the other notions described above, are the set of indexes of the appropriate variables, and they satisfy simple properties, like being preserved under renamings.

2.2.6 Renaming

As a prelude to defining (capture avoiding) substitution, filling, and the companion notion of renaming of bound variables (a.k.a α-conversion), as we pointed out in [7], careful treatments of the λ-calculus will define, by structural recursion, the notion of a variable renaming $e^{\{x \mapsto y\}}$. One nice property possessed by variable renamings is that it preserves rank, unlike substitution, and to establish its totality we must build that fact into its type:

$$e^{\{x \mapsto y\}} : \{e_0 \in \mathbf{E} \mid \mathrm{rank}(e_0) = \mathrm{rank}(e)\}$$

Actually we must build much more information into its type. For example we must assert that renaming preserves the subtypes used in defining the basic syntax. In other word it maps λ expressions to λ expression, applications to applications etc. Even the simpler notion of renaming generates over twenty TCCs that must be verified. Renaming itself is of interest because of the nested recursion that takes place in the λ clause.

Definition 2.5 (Renaming $e^{\{x\mapsto y\}}$): `Subst` :26

$$(\lambda z.e)^{\{x\mapsto y\}} = \lambda\nu.((e^{\{z\mapsto\nu\}})^{\{x\mapsto y\}}) \qquad \text{for } \nu \text{ fresh, i.e. } \nu \notin \mathrm{FV}(e) \cup \{x, y\}.$$

Note that we always rename the λ bound variables, regardless of whether or not a clash might have otherwise occurred. To produce such *fresh* variables we posit a function new that pulls them, or at least their indices, out of a hat:

Definition 2.6 (new): `FLL-version-03` :112

$$new : \mathbf{P}_\omega(\mathbf{N}) \mapsto \mathbf{N} \quad \text{satisfying} \quad (\forall W \in \mathbf{P}_\omega(\mathbf{N}))(new(W) \notin W)$$

2.2.7 Substitution

We can now turn our attention to the *glamour* operations: substitution, and filling.

Definition 2.7 (Substitution e^σ): `Subst` :110–182

e^σ is the result of simultaneous substitution of free occurrences of $x \in \mathrm{Dom}(\sigma)$ in e by $\sigma(x)$, taking care not to trap variables. Taking care not to trap variables amounts to defining simultaneous substitution into a λ expression by the following scheme, which makes use of the previously defined notion of renaming,

$$(\lambda z.e)^\sigma = \lambda\nu.((e^{\{z\mapsto\nu\}})^\sigma) \qquad \text{for } \nu \text{ fresh, i.e. } \nu \notin \mathrm{FV}(e) \cup \mathrm{FV}(\sigma).$$

In the case of decorated parameters we define simultaneous substitution as follows, $(X^{\sigma_0})^\sigma = X^{(\sigma_0^\sigma)}$, where $\sigma_0^\sigma = \{x \mapsto \sigma_0(x)^\sigma \mid x \in \mathrm{Dom}(\sigma_0)\}$.

We also prove that our notation is not misleading; in the following lemma the left hand side is renaming, while the right hand side is substitution.

Lemma 2.8 (Renaming): $e^{\{x\mapsto y\}} = e^{\{x\mapsto y\}}$ `Subst` : `Subst_Rename` :166

2.2.8 Filling

Definition 2.9 (Filling e^Σ): `Fill` :1–73

e^Σ is the result of simultaneous substitution of decorated occurrences of $X \in \mathrm{Dom}(\Sigma)$ in e by $\Sigma(X)$ instantiated by the (suitably substituted) decoration, again taking care not to trap variables (other than those in the range of the decoration). For decorated parameters it is defined as follows, $(X^\sigma)^\Sigma = \Sigma(X)^{\sigma^\Sigma}$ if $X \in \mathrm{Dom}(\Sigma)$, and X^{σ^Σ} otherwise, where σ^Σ is defined point-wise: $\sigma^\Sigma = \{x \mapsto \sigma(x)^\Sigma \mid x \in \mathrm{Dom}(\sigma)\}$. In the case of λ-abstractions, we define parameter substitution exactly as we would value substitution:

$$(\lambda x.e)^\Sigma = \lambda\nu.((e^{\{x\mapsto\nu\}})^\Sigma) \qquad \text{for } \nu \text{ fresh, i.e. } \nu \notin \mathrm{FV}(e) \cup \mathrm{FV}(\Sigma).$$

which again makes use of the previously defined notion of renaming.

2.2.9 α Equivalence

While the notion of being equivalent modulo the renaming of bound variables easily extends to this more general setting [13] by simply clarifying what can and cannot be bound: parameters are *never* bound; variables in the domain of an annotating substitution are *never* bound; variables in the range of an annotating substitution *may* be bound. We begin by presenting the (proof) theoretical definition of $\overset{\alpha}{\equiv}$ [13]:

Definition 2.10 ($\overset{\alpha}{\equiv}$): `WeakAlpha` :1–108

$$\text{(Base)}\ \frac{}{v \overset{\alpha}{\equiv} v} \quad \text{provided } v \in \mathbf{A} \cup \mathbf{X} \qquad \text{(Op)}\ \frac{e_0 \overset{\alpha}{\equiv} e_0' \quad \dots \quad e_n \overset{\alpha}{\equiv} e_n'}{\vartheta(e_0, \dots, e_n) \overset{\alpha}{\equiv} \vartheta(e_0', \dots, e_n')}$$

$$\text{(Subst)}\ \frac{\mathrm{Dom}(\Delta_0) = \mathrm{Dom}(\Delta_1) \qquad \Delta_0(\delta) \overset{\alpha}{\equiv} \Delta_1(\delta) \quad \forall \delta \in \mathrm{Dom}(\Delta_i)}{\Delta_0 \overset{\alpha}{\equiv} \Delta_1} \quad \Delta_0, \Delta_1 \in \mathbf{S} \text{ or } \Delta_0, \Delta_1 \in \mathbf{F}$$

$$\text{(Param)}\quad \frac{\sigma_0 \overset{\alpha}{\equiv} \sigma_1}{X^{\sigma_0} \overset{\alpha}{\equiv} X^{\sigma_1}} \qquad \text{(Lambda)}\quad \frac{e_0^{\{x_0 \mapsto \nu\}} \overset{\alpha}{\equiv} e_1^{\{x_1 \mapsto \nu\}} \quad \text{for } \nu \text{ fresh.}}{\lambda x_0.e_0 \overset{\alpha}{\equiv} \lambda x_1.e_1}$$

There is substantial leeway in the precise way one formalizes $\overset{\alpha}{\equiv}$. These issues are discussed at length in [19] so we shall not dwell on them here. The main point to make is that we attempted two formalizations, in keeping with the well tested rule of having weak introduction principles and strong elimination principles. The strong form was developed in `Alpha` :1-450 where we actually formalized the notion of being a proof in the above system, and defined $\overset{\alpha}{\equiv}$ by

$$e_0 \overset{\alpha}{\equiv} e_1 \overset{\triangle}{=} (\exists \Gamma \text{ a proof in the above system })(\Gamma \vdash e_0 \overset{\alpha}{\equiv} e_1) \qquad \texttt{Alpha}:185$$

Establishing facts such as transitivity, symmetry, and reflexivity required the formal manipulation of proof objects. The weak form was developed in `WeakAlpha` :1-48 and was a simple inductive definition along the lines we used to prove the Church–Rosser theorem in [7]. The two forms were proved equivalent in `WeakAlpha` :85 . However it was the weaker form that proved the most useful in the subsequent development of the CUI theorem.

Both the formal and theoretical treatment require the development of a large library of facts concerning this relation. The most basic facts necessary are collected here together in a lemma.

Lemma 2.11 ($\overset{\alpha}{\equiv}$):

1. $e_0 \overset{\alpha}{\equiv} e_1 \Rightarrow \mathrm{rank}(e_0) = \mathrm{rank}(e_1)$. `WeakAlpha` : `WeakRank` :76
2. $\overset{\alpha}{\equiv}$ is reflexive. `WeakAlpha` : `WeakReflexive` :88
3. $\overset{\alpha}{\equiv}$ is symmetric. `WeakAlpha` : `WeakSymmetric` :94
4. $\overset{\alpha}{\equiv}$ is reflexive. `WeakAlpha` : `WeakTransitive` :97
5. $e_0 \overset{\alpha}{\equiv} e_1 \Rightarrow \mathrm{FV}(e_0) = \mathrm{FV}(e_1)$. `WeakAlpha` : `Alpha_FV` :104

2.2.10 Substitution preserves α Equivalence

The main result concerning the relationship between α equivalence and substitution is that the latter preserves the former:

Lemma 2.12 (AlphaSubst): `AlphaSubst` : `AlphaSubst_Theorem` :132

$e_0 \overset{\alpha}{\equiv} e_1 \wedge \sigma_0 \overset{\alpha}{\equiv} \sigma_1 \Rightarrow e_0^{\sigma_0} \overset{\alpha}{\equiv} e_1^{\sigma_1}$

As pointed out by Mason in [13] the proof of this is somewhat delicate, actually it is even more delicate than suggested there. Since the proof presented there does not stand up to the test of formalizing in PVS. In [13] the suggested proof requires establishing, directly, the following three properties by simultaneous induction.

Lemma 2.13 (Alpha): `AlphaSubst` : `AlphaSubst_Swap` :132

1. $(e_0 \overset{\alpha}{\equiv} e_1 \wedge e_0' \overset{\alpha}{\equiv} e_1') \Rightarrow e_0^{\{x \mapsto e_0'\}} \overset{\alpha}{\equiv} e_0^{\{x \mapsto e_0'\}}$
2. $(\nu_0 \notin \mathrm{FV}(e) \wedge \nu_0 \neq x \neq \nu_1) \Rightarrow (e_0^{\{\nu_0 \mapsto \nu_1\}})^{\{x \mapsto e\}} \overset{\alpha}{\equiv} (e_0^{\{x \mapsto e\}})^{\{\nu_0 \mapsto \nu_1\}}$
3. $(e_0^{\{x \mapsto \nu\}})^{\{\nu \mapsto y\}} \overset{\alpha}{\equiv} e_0^{\{x \mapsto y\}} \qquad \text{for } \nu \notin \mathrm{FV}(e_0)$

However the actual proof first must establish (by simultaneous induction) the corresponding three properties for renaming rather than substitution `AlphaSubst` : `Alpha_Prop_Lemma` :70 . Then using these facts concerning renaming establish the first two properties above, again by simultaneous induction. Note that the third property of lemma 2.13 is actually a property of renaming. It is now relatively simple to prove that substitution preserves α congruence.

The formal development and proof of the CIU theorem also requires some other relatively simple properties of substitution and α congruence. They are summarized in the following lemma.

Lemma 2.14 (SubstProps):

1. $(\mathrm{Dom}(\sigma_0) - \mathrm{Dom}(\sigma_1)) \cap \mathrm{FV}(e) = \emptyset \Rightarrow (e^{\sigma_0})^{\sigma_1} \overset{\alpha}{\equiv} e^{(\sigma_0^{\sigma_1})}$ `AlphaSubst` : `AlphaSubst_Inner` :136
2. $x \notin \mathrm{FV}(e) \Rightarrow (e^{\{y \mapsto x\}})^{\{x \mapsto e_0\}} \overset{\alpha}{\equiv} e^{\{y \mapsto e_0\}}$ `AlphaSubst` : `AlphaSubst_Contract` :140
3. $\mathrm{FV}(e) \cap \mathrm{Dom}(\sigma) = \emptyset \Rightarrow e \overset{\alpha}{\equiv} e^{\sigma}$ `AlphaSubst` : `Subst_FV` :149

2.2.11 Filling preserves α Equivalence

We must now establish similar properties of filling, the principle being that filling also preserves α congruence.

Lemma 2.15 (AlphaFill): `AlphaFill : AlphaFill_Theorem :51`

$e_0 \stackrel{\alpha}{\equiv} e_1 \wedge \Sigma_0 \stackrel{\alpha}{\equiv} \Sigma_1 \Rightarrow e_0^{\Sigma_0} \stackrel{\alpha}{\equiv} e_1^{\Sigma_1}$

This is proved by simultaneous induction along with the following fact.

Lemma 2.16 (FillSwap): `AlphaFill : AlphaFill_Swap :47`

$x \notin \mathrm{FV}(\Sigma) \Rightarrow (e^{\Sigma})^{\{x \mapsto y\}} \stackrel{\alpha}{\equiv} (e^{\{x \mapsto y\}})^{\Sigma}$

As in the case of substitution the development and proof of the CIU theorem requires establishing several simple facts concerning filling. These we collect together in the following lemma.

Lemma 2.17 (FillProps): `AlphaFill :65–82`

1. $\mathrm{Dom}(\Sigma) \cap \mathrm{FP}(e) = \emptyset \Rightarrow e \stackrel{\alpha}{\equiv} e^{\Sigma}$.
2. $e_0 \stackrel{\alpha}{\equiv} e_1 \Rightarrow \mathrm{FP}(e_0) = \mathrm{FP}(e_1)$.
3. $(\bigwedge_{i<2} \mathrm{FP}(e_i) = \emptyset \wedge X_0 \neq X_1) \Rightarrow (e^{\{X_0 \mapsto e_0\}})^{\{X_1 \mapsto e_1\}} \stackrel{\alpha}{\equiv} e^{\{X_i \mapsto e_i \mid i<2\}}$
4. $(\bigwedge_{i<2} \mathrm{FP}(e_i) = \emptyset \wedge X_0 \neq X_1) \Rightarrow e^{\{X_0 \mapsto e_0\} \cup \{X_1 \mapsto e_1\}} \stackrel{\alpha}{\equiv} e^{\{X_1 \mapsto e_1\} \cup \{X_0 \mapsto e_0\}}$

Definition 2.18 (Term, Closed): `Closed :1-27`

We adopt the convention that an expression with no parameters is called a *term*. Furthermore a *term* with no free variables is *closed*. Thus being closed implies having no parameters.

These notions are used heavily in the statement and proof of the CIU theorem, and consequently we must also establish simple facts concerning them, such as that they are preserved by α equivalence, and that they are both preserved under filling (since they remain unchanged).

2.2.12 Values

Definition 2.19 (Value Expressions (V)): `Values :1-122`

The set of value expressions, $\mathbf{V}$, contains all variables, atoms, and λs. It may in addition contain expressions of the form $\vartheta(v^n)$. Operators, ϑ, that produce value expressions, are called *constructors*. A binary (non-mutable) pairing operation is a prototypical constructor. $\mathbf{V}$ must also satisfy:

(triv) $\mathbf{V}$ is closed under $\stackrel{\alpha}{\equiv}$ `Values :75`

(vsub) $v \in \mathbf{V} \Rightarrow v^{\sigma} \in \mathbf{V}$ `Values :81`

(inst) $v \in \mathbf{V} \Rightarrow v^{\Sigma} \in \mathbf{V}$ `Values :84`

(dich) $e^{\Sigma} \in \mathbf{V} \Rightarrow (e \in \mathbf{V}) \vee (e = X^{\sigma} \wedge \Sigma(X) \in \mathbf{V})$ `Values :87`

We let v range over $\mathbf{V}$.

We can now define `Values : NewE :37` the subtype of expressions that we will restrict our attention to in the sequel. Those expressions that only contain parameters annotated by value substitutions, substitutions whose range is a subset of $\mathbf{V}$. We will continue to denote this new set by $\mathbf{E}$, though in the formal treatment it is called `NewE`. Once defined we must also show that renaming, value substitution, and filling all preserve the defining property of `NewE`. This is done via judgements to enable the PVS typechecker to use them in the typechecking process.

The following lemma simply points out a simple consequence of the closure conditions on values.

Lemma 2.20 (inv): `Values : ValuesSubst_Lemma :99`

$e^{\sigma} \in \mathbf{V} \Rightarrow e \in \mathbf{V}$

Proof: Pick e_0 such that $e_0^{\sigma} \in \mathbf{V}$, and let X be a fresh parameter. Put $e = X^{\sigma}$, $\Sigma = \{X \mapsto e_0\}$. Then

$$e^{\Sigma} = (X^{\sigma})^{\Sigma} = \Sigma(X)^{(\sigma^{\Sigma})} = \Sigma(X)^{\sigma} = e_0^{\sigma} \in \mathbf{V}$$

since $X^{\sigma} \notin \mathbf{V}$ we can use (**dich**) to conclude that $e_0 \in \mathbf{V}$. □

3 Semantics of Terms

3.1 Operational Semantics

In a λ-language computation state is represented as a class of expressions. Each particular language will possess it's own class of state expressions, reflecting the nature of the primitive operations that it is based on. In what follows we fix a distinguished parameter $\circ$ to designate the position at which effects are to be observed in a context representing a computation state. We call this the *state* parameter.

Definition 3.1 (State expressions (Z)): `Computation`:19

For a particular λ-language $\mathbf{Z}$ is assumed to be a subset of $\mathbf{E}$. We call $\mathbf{Z}$ the set of state expressions. $\mathbf{Z}$ is assumed to satisfy the following uniformity conditions:

(triv) $\mathbf{Z}$ is closed under $\overset{\alpha}{\equiv}$ `Computation`:27

(par) $\zeta \in \mathbf{Z} \Rightarrow \circ \in \mathrm{FP}(\zeta)$ `Computation`:31

(vsub) $\zeta \in \mathbf{Z} \Rightarrow \zeta^{\sigma} \in \mathbf{Z}$ assuming $\circ \notin \mathrm{FP}(\sigma)$ `Computation`:34

(inst) $\zeta \in \mathbf{Z} \Rightarrow \zeta^{\Sigma} \in \mathbf{Z}$ assuming $\circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{FP}(\Sigma))$ `Computation`:37

ζ ranges over $\mathbf{Z}$ and $\circ \in \mathbf{Z}$ is the empty state expression.

Definition 3.2 (Computation States (CS)): `Computation`:46

$\mathbf{CS} \triangleq \mathbf{Z} : \mathbf{E}$ is the set of computation states. We let S range over $\mathbf{CS}$ and let $\zeta : e$ be the state with state context ζ and expression e. The computation state $\zeta : e$ is said to be a *value state* iff $e \in \mathbf{V}$. Given a state $\zeta : e$, we associate a corresponding expression by filling the state parameter in the context with the expression, i.e. $\zeta^{\{\circ \mapsto e\}}$. A state is *closed* just if its corresponding expression is closed, in other words if $\zeta^{\{\circ \mapsto e\}}$ has no free parameters or variables. Application of value and parameter substitutions to states is defined by in the obvious way: $(\zeta : e)^{\sigma} = \zeta^{\sigma} : e^{\sigma}$ and $(\zeta : e)^{\Sigma} = \zeta^{\Sigma} : e^{\Sigma}$. Note that by **(vsub)** **(inst)** these are only meaningful if $\circ \notin \mathrm{Dom}(\sigma)$ and $\circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{FP}(\Sigma))$.

Definition 3.3 (Reduction ($\longrightarrow$, $\longrightarrow\!\!\!\!\succ$) and Definedness ($\downarrow$)): `Semantics`:28

Given a reduction relation for a λ-language: $\zeta : e \longrightarrow \zeta' : e'$, the following definitions are standard. The *transitive* closure of $\longrightarrow$ is $\longrightarrow\!\!\!\!\succ$

Definedness: `Semantics`:38

$$(\zeta : e)\downarrow \;\Leftrightarrow\; (e \in \mathbf{V}) \vee (\zeta : e \longrightarrow\!\!\!\!\succ \zeta' : v)$$

Approximation: `Semantics`:42

$$(\zeta_0 : e_0) \preceq (\zeta_1 : e_1) \;\Leftrightarrow\; (\zeta_0 : e_0)\downarrow \Rightarrow (\zeta_1 : e_1)\downarrow$$

Equidefined: `Semantics`:45

$$(\zeta_0 : e_0) \updownarrow (\zeta_1 : e_1) \;\Leftrightarrow\; ((\zeta_0 : e_0)\downarrow \Leftrightarrow (\zeta_1 : e_1)\downarrow)$$

Equivalued: `Semantics`:48

$$(\zeta_0 : e_0) \sim (\zeta_1 : e_1) \;\Leftrightarrow\; (\zeta_0 : e_0) \updownarrow (\zeta_1 : e_1) \wedge (\zeta_0 : e_0)\downarrow \Rightarrow (\exists v \in \mathbf{V}, \zeta \in \mathbf{Z})(\bigwedge_{j<2} (\zeta_i : e_i) \longrightarrow\!\!\!\!\succ \zeta : v)$$

Length: `Semantics`:55

$|\zeta : e|$ is the least $n \in \mathbf{N}$ such that $\zeta : e$ reduces to a value state in n steps, if $\zeta : e\downarrow$.

The formal treatment of the transitive closure of a relation, and its rank (used in defining the length of a computation) are theories that we were able to reuse from our proof of the Church–Rosser theorem [7]. They are treated in `rel`:1–210 . To define reduction rules for general λ-languages, and formulate the central properties of reduction and equivalence, we introduce the notions of redex and reduction context. Since evaluation is call-by-value, a redex is simply an non-constructor operator applied to the appropriate number of value expressions. Redexes and value expressions must be disjoint, thus we must account for the fact that some expressions of the form $\vartheta(v_1, \ldots, v_n)$ may be value expressions.

Definition 3.4 (Redexes ($\mathbf{E}_{\mathrm{r}}$)): `Computation`:65

The set of redexes, $\mathbf{E}_{\mathrm{r}}$, is defined by:

$$\mathbf{E}_{\mathrm{r}} = \mathbf{O}_n(\mathbf{V}^n) - \mathbf{V}$$

Note that redexes in our framework may or may not reduce. The point is that they are simply expressions of a particular shape, in other words: candidates for reduction. The set of redexes is closed under the following uniformity conditions.

Lemma 3.5 (Redex Uniformity): Computation :??
Redexes satisfy the following uniformity conditions:

(triv) $\mathbf{E}_{\mathrm{r}}$ is closed under $\overset{\alpha}{\equiv}$ Computation :??

(vsub) $r \in \mathbf{E}_{\mathrm{r}} \Rightarrow r^{\sigma} \in \mathbf{E}_{\mathrm{r}}$ Computation :88

(inst) $r \in \mathbf{E}_{\mathrm{r}} \Rightarrow r^{\Sigma} \in \mathbf{E}_{\mathrm{r}}$ Computation :??

We use the distinguished parameter $\bullet$ to denote the *evaluation* parameter (or hole), and we define the notion of a reduction context, R, accordingly. Reduction contexts (also called evaluation contexts in the literature) identify the subexpression of an expression in which reduction to a value must occur next. They themselves represent the remainder of the computation, i.e the *continuation*. In our approach they correspond to the left-first, call-by-value reduction strategy of [22] and were first introduced by [3].

Definition 3.6 (Reduction Contexts (R)): Computation :70
The set of reduction contexts, $\mathbf{R}$, is the subset of $\mathbf{E}$ defined by

$$\mathbf{R} = \{\bullet\} \cup \mathbf{O}_{m+n+1}(\{v \in \mathbf{V} \mid \bullet \notin \mathrm{FP}(v)\}^m, \mathbf{R}, \{e \in \mathbf{E} \mid \bullet \notin \mathrm{FP}(e)\}^n)$$

We let R range over $\mathbf{R}$. We adopt the convention of writing $R[\,e\,]$ instead of $R^{\{\bullet \mapsto e_1\}}$.

Observe that both the definition of redex, and the definition of reduction contexts depend on the particular choice of values, and thus vary from one λ-language to another. Also note that $\mathbf{R}$ will satisfy a similar set of uniformity conditions as those satisfied by states (Definition 3.1):

Lemma 3.7 (Rcx Uniformity): Computation :1-233
Reduction contexts satisfy the following uniformity conditions:

(triv) $\mathbf{R}$ is closed under $\overset{\alpha}{\equiv}$ Computation :109

(par) $R \in \mathbf{R} \Rightarrow \bullet \in \mathrm{FP}(R)$ Computation :94

(vsub) $R \in \mathbf{R} \Rightarrow R^{\sigma} \in \mathbf{R}$ assuming $\bullet \notin \mathrm{FP}(\sigma)$ Computation :189

(inst) $R \in \mathbf{R} \Rightarrow R^{\Sigma} \in \mathbf{R}$ assuming $\bullet \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{FP}(\Sigma))$ Computation :224

A term, an expression without parameters, is either a value expression or decomposes uniquely into a redex placed in a reduction context. This generalizes to the present situation in the following fashion.

Lemma 3.8 (Decomposition): Computation : Rcx_Decomposition_Theorem :179
For any λ-language, if $e \in \mathbf{E}$ then either $e \in \mathbf{V}$ or e can be written uniquely as either

(i) $R[\,r\,]$ where R is a reduction context and $r \in \mathbf{E}_{\mathrm{r}}$, or else

(ii) $R[\,X^{\sigma}\,]$ where R is a reduction context, and $X^{\sigma} \in \mathbf{P}^{\mathbf{S}}$ is a decorated parameter.

In the latter case we say that the expression is *touching* the parameter, while in the former we say that the expression *may be reducible*. The requirement that the evaluation parameter does not occur in either the leading value expressions, or the trailing expressions is necessary for the uniquenss aspect of this lemma. A simple counterexample is the following:

$$R_0 = \vartheta(\bullet, \bullet) \quad R_1 = \vartheta(\bullet, r) \quad R_0[\,r\,] = \vartheta(r, r) = R_1[\,r\,].$$

3.2 Uniform Semantics

We now specify what we mean by a λ-language having *uniform semantics*. The key requirement is that reduction steps that do not touch a parameter are uniformly independent of what the parameter might stand for. In addition, we require that: single step reduction is essentially deterministic; reduction is preserved by value substitution; a state, and its associated expression started in the empty state context, are equi-defined; and if one state reduces to another then the two states are equi-defined and the reduct has shorter computation length, if defined.

Definition 3.9 (Uniformity (U)): Semantics :1–121
A λ-language is said to have *uniform semantics* if it satisfies the following:

(i) Functional modulo $\stackrel{\alpha}{\equiv}$ and implicit bindings: `Semantics` :70

$$(\bigwedge_{j<2}(\zeta_i : e_i) \stackrel{\alpha}{\equiv} (\zeta_i' : e_i')) \;\Rightarrow\; ((\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1) \;\Rightarrow\; (\zeta_0' : e_0' \longrightarrow \zeta_1' : e_1'))$$

$$(\bigwedge_{j<2}\zeta : e \longrightarrow \zeta_i : e_i) \;\Rightarrow\; \zeta_0^{\{\circ\mapsto e_0\}} \stackrel{\alpha}{\equiv} \zeta_1^{\{\circ\mapsto e_1\}}.$$

(ii) Uniform in value substitutions: `Semantics` :75

$$\zeta : e \longrightarrow \zeta' : e' \;\Rightarrow\; (\zeta : e)^\sigma \longrightarrow (\zeta' : e')^\sigma$$

provided $\mathrm{Dom}(\sigma) \cap (\mathrm{Traps}(\zeta) \cup \mathrm{Traps}(\zeta')) = \emptyset$ and $\circ \notin \mathrm{FP}(\sigma)$.

(iii) State evaluation: `Semantics` :79

$$\zeta : e \updownarrow \circ : \zeta^{\{\circ\mapsto e\}}$$

(iv) Well-founded: `Semantics` :82

$$(\zeta : e \longrightarrow \zeta' : e' \wedge \zeta : e\downarrow) \;\Rightarrow\; (\zeta' : e'\downarrow \wedge |\zeta' : e'| < |\zeta' : e'|)$$

(v) Parametric: `Semantics` :86

$$\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1 \;\Rightarrow\; (\zeta_0 : e_0)^\Sigma \longrightarrow (\zeta_1 : e_1)^\Sigma \qquad \text{for any } \Sigma \in \mathbf{F} \text{ with } \circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{FP}(\Sigma)).$$

(vi) Dichotomy: `Semantics` :90
Assuming $\circ \notin \mathrm{Dom}(\Sigma)$, if $(\zeta : e)^\Sigma \longrightarrow \zeta' : e'$ then either

$\zeta : e$ touches a parameter in the domain of Σ, or

$\zeta : e \longrightarrow \zeta_1 : e_1$, for some $\zeta_1 : e_1$ such that $\zeta' : e' \stackrel{\alpha}{\equiv} (\zeta_1 : e_1)^\Sigma$.

(vii) Closure: `Semantics` :95 & 100

(i) $(\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1) \;\Rightarrow\; \mathrm{FV}(\zeta_1^{\{\circ\mapsto e_1\}}) \subseteq \mathrm{FV}(\zeta_0^{\{\circ\mapsto e_0\}})$

(ii) $(\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1) \;\Rightarrow\; \mathrm{FP}(\zeta_1 : e_1) \subseteq \mathrm{FP}(\zeta_0 : e_0) \wedge (\circ \notin \mathrm{FP}(e_0) \;\Rightarrow\; \circ \notin \mathrm{FP}(e_1))$

In the languages we consider **(U)** holds for the following reasons. **(U.i)** holds because the only non-determinism in a reduction step is the choice of names used in the state context. **(U.ii)** holds because reductions that do not depend on the values of free variables, are parametric in the values that those variables take. **(U.iii)** holds because reduction of $\circ : \zeta^{\{\circ\mapsto e\}}$ essentially recreates the state context ζ. **(U.iv)** follows since if a state is defined, then any reduction makes progress. Clearly if the reduct state is defined, then the original state is defined. **(U.v)** and **(U.vi)** formalizes the uniformity requirement for reduction steps. **(U.vi)** states that either computation touches a parameter or is parametric. These are satisfied by reduction rules that treat the reduction context as an abstract entity, and that depend on the kind of construction of a redex argument, but not on any information about subparts. This is easily expressed using the parameters. Finally, **(U.vii)** holds because computation neither introduces new parameters, nor new free variables. Furthermore the state parameter does not propagate into the expression being evaluated.

3.3 Approximation and Equivalence

Now we define operational approximation and equivalence on terms and lay the ground work for studying properties of these relations. In what follows we fix a particular distinguished parameter, X, distinct from $\circ$ and $\bullet$. We let C range over expressions with X as the only free parameter. Such expressions play the role of traditional λ-calculus contexts, and we extend our convention, stated in definition 3.6, of sometimes writing $C[\,e\,]$ instead of $C^{\{X\mapsto e\}}$. Note however that for example the traditional context $\lambda z.\mathtt{app}(y, \lambda x.\mathtt{app}([\,], z))$ does not correspond to $\lambda z.\mathtt{app}(y, \lambda x.\mathtt{app}(X, z))$ but rather to one where the trappings have been made explicit at the occurrence of the X parameter: $\lambda z.\mathtt{app}(y, \lambda x.\mathtt{app}(X^{\{x\mapsto x, z\mapsto z\}}, z))$.

Definition 3.10 (Approximation $e_0 \sqsubseteq e_1$, Equivalence $e_0 \cong e_1$): `Approx` :1–76
For terms e_0, e_1 define

$$e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall C \;\big|\; C[\,e_0\,], C[\,e_1\,]\text{closed})(\circ : C[\,e_0\,] \preceq \circ : C[\,e_1\,])$$

$$e_0 \cong e_1 \;\Leftrightarrow\; e_0 \sqsubseteq e_1 \wedge e_1 \sqsubseteq e_0$$

Note that we are restricting our attention to terms, rather than arbitrary expressions. It is easy to see that operational approximation is a congruence on terms: if $e_0 \sqsubseteq e_1$, then $C[\,e_0\,] \sqsubseteq C[\,e_1\,]$. Similarly for operational equivalence.

4 The CIU Theorem and it's Proof

4.1 The CIU Theorem

We may now state the main result concerning languages with uniform semantics, the CIU theorem.

Theorem 4.1 (CIU): CIU : CIU :210

For a λ-language with uniform semantics:

$$e_0 \sqsubseteq e_1 \;\Leftrightarrow\; (\forall \zeta, R, \sigma \mid \bigwedge_{j<2} \zeta : R[\![e_j^\sigma]\!] \text{ closed})(\zeta : R[\![e_0^\sigma]\!] \preceq \zeta : R[\![e_1^\sigma]\!])$$

under the assumption that $\mathrm{FV}(e_0) = \mathrm{FV}(e_1)$.

CIU is an acronym for *closed instances of uses*, since the value substitution, σ, closes the e_i, while the reduction context, R, represents a use of the value returned, in the appropriate state, ζ. Thus only the value of the expressions, e_i, are observed. Although the theorem holds without the added assumption that $\mathrm{FV}(e_0) = \mathrm{FV}(e_1)$, we have as yet been unable to verify this using PVS. This is the subject of ongoing work, and we hope to be able to remove this assumption shortly.

This assumption is used in the following lemma, which guarantees that we may preserve closedness by replacing some number of occurrences of e_0 by e_1.

Lemma 4.2 (Closed): CIU : Ciu_Closed :83

Suppose e satifies $\mathrm{FV}(e) = \emptyset$, $\mathrm{FP}(e) \subseteq \{X_0, X_1\}$, $\mathrm{FV}(e_0) = \mathrm{FV}(e_1)$, $e^{\{X_0 \mapsto e_0, X_1 \mapsto e_0\}}$ is closed, and $e^{\{X_0 \mapsto e_1, X_1 \mapsto e_1\}}$ is closed. Then $e^{\{X_0 \mapsto e_0, X_1 \mapsto e_1\}}$ is also closed.

To see that the first two conditions (without the third) are necessary consider the following:

$$e = \lambda y_0.\lambda y_1.X_0^{\{y_1 \mapsto y_1, y_0 \mapsto X_1^{\{y_0 \mapsto y_0, y_1 \mapsto e'\}}\}}$$

$$e^{\{X_0 \mapsto y_0, X_1 \mapsto y_0\}} = \lambda y_0.\lambda y_1.y_0, \quad e^{\{X_0 \mapsto y_1, X_1 \mapsto y_1\}} = \lambda y_0.\lambda y_1.y_1, \quad \text{but} \quad e^{\{X_0 \mapsto y_0, X_1 \mapsto y_1\}} = \lambda y_0.\lambda y_1.e'.$$

To see that the first three are all necessary consider:

$$e = \lambda y_1.X_0^{\{y_1 \mapsto y_1, y_0 \mapsto \lambda y_2.X_1^{\{y_0 \mapsto y_1, y_1 \mapsto \lambda y_3.X_0\}}\}}$$

$$e^{\{X_0 \mapsto y_0, X_1 \mapsto y_0\}} = \lambda y_1.\lambda y_2.y_1, \quad e^{\{X_0 \mapsto y_1, X_1 \mapsto y_1\}} = \lambda y_1.y_1, \quad \text{but} \quad e^{\{X_0 \mapsto y_0, X_1 \mapsto y_1\}} = \lambda y_1.\lambda y_2.\lambda y_3.y_0.$$

Another useful lemma in the PVS proof allows us, ahead of time, to replace a parameter, by what it is about to be filled with.

Lemma 4.3 (Delay): CIU : Ciu_Delay :100

If e is a term, then $R[\![X^\sigma]\!]^{\{X \mapsto e\}} = R[\![e^\sigma]\!]^{\{X \mapsto e\}}$.

4.2 The CIU Proof

The fact that one can present a syntactic reduction system for imperative λ-calculi was discovered independently in 1986-1987 in [16], and in [4]. As well as being conceptually elegant, it has also provided the necessary tools for several key results and proofs. In addition to eliminating messy isomorphism considerations, to deal with arbitrary choice of names of newly allocated structures, it also was a key step leading to the formulation of the CIU theorem, first presented in [15]. There are now several proofs of this result in the literature. The theorem first appeared in [16] and the proof (sketch) presented there used techniques similar to those developed here. A somewhat more detailed and general version of this same technique appeared in [24]. A second, distinct, proof was presented in [8] that simply shows that the CIU relation is a congruence. This proof also appears in a more general setting in [14].

Proof: (CIU $\Rightarrow$) CIU : CIUR :153

The (CIU $\Rightarrow$) direction is relatively simple, and requires producing a context C such that $C[\![e_i]\!]$ evaluates to $\zeta : R[\![e_1^\sigma]\!]$. Suppose that $e_0 \sqsubseteq e_1$, and choose ζ, R, σ such that $\zeta : R[\![e_j^\sigma]\!]$ is closed for $0 \leq j < 2$. Since these expressions are closed, we may assume that the only free parameter in ζ is $\circ$, and that $\sigma(x)$ is a term, for each $x \in \mathrm{Dom}(\sigma)$. Extend σ to $\hat{\sigma}$ by mapping each $x \in \mathrm{Traps}(\zeta) - \mathrm{Dom}(\sigma)$ to itself. Note that $e_i^{\hat{\sigma}} = e_i^\sigma$. Choose an new parameter X and consider the expression $C = \zeta^{\{\circ \mapsto R^{\{\bullet \mapsto X^{\hat{\sigma}}\}}\}}$.

Lemma 4.4: CIU : CIUR_Helper2 :129

$$C[\![e_j]\!] = (\zeta^{\{\circ \mapsto R^{\{\bullet \mapsto X^{\hat{\sigma}}\}}\}})^{\{X \mapsto e_j\}} = \zeta^{\{\circ \mapsto R^{\{\bullet \mapsto e_i^{\hat{\sigma}}\}}\}}$$

Now by this lemma $\circ : C[\,e_j\,]$ are closed, thus by definition 3.10 we have that

$$\circ : C[\,e_0\,] \preceq \circ : C[\,e_1\,].$$

Then $\circ : C[\,e_j\,] \updownarrow \zeta : R[\,e_j^\sigma\,]$ by lemma 4.4, and (**Unif.iii**). Thus $\zeta : R[\,e_0^\sigma\,] \preceq \zeta : R[\,e_1^\sigma\,]$ as desired. □

Proof: (CIU $\Leftarrow$) `CIU : CIUL :206`

Assume that

$$\text{(ciu)} \quad (\forall \zeta, R, \sigma \mid \bigwedge_{j<2} \zeta : R[\,e_j^\sigma\,] \text{ closed})(\zeta : R[\,e_0^\sigma\,] \preceq \zeta : R[\,e_1^\sigma\,]).$$

We prove

$$(\forall \zeta, e \mid \bigwedge_{j<2} (\zeta : e)^{\{X \mapsto e_j\}} \text{ closed})((\zeta : e)^{\{X \mapsto e_0\}} \preceq (\zeta : e)^{\{X \mapsto e_1\}})$$

by induction on the length of the computation of $(\zeta : e)^{\{X \mapsto e_0\}}$.

The proof is relatively straightforward modulo the hard case. The hard case is when e_0 is a value, and $(\zeta : e)^{\{X \mapsto e_0\}}$ non-trivially reduces to a value. This case itself must be proved by another induction. This induction is on the number of *non-nested occurrences* of X that occur to the left, or below, where the computation is currently taking place. An occurrence of X in e is said to be *non-nested* if it is not in the scope of a λ-expression, or in the range of a substitution annotating a parameter.

(CIU $\Leftarrow$) Base Case: `CIU : CIUL_Base :156`

Suppose that $(\zeta : e)^{\{X \mapsto e_0\}}$ is a value state. Thus $e^{\{X \mapsto e_0\}}$ must be a value. Thus by (**dich**) of definition 2.19, either e is a value or else it is of the form X^σ for some σ, and e_0 itself must be a value. In the case where e is a value, we have that $e^{\{X \mapsto e_1\}}$ is also a value by (**fill**) of definition 2.19, and hence $(\zeta : e)^{\{X \mapsto e_1\}}$ is also value state (note that $\zeta^{\{X \mapsto e_1\}} \in \mathbf{Z}$ is implicit in the hypothesis). So suppose that e_0 is a value, and that e is of the form X^σ for some σ. Also let $\zeta_j = \zeta^{\{X \mapsto e_j\}}$ and $\sigma_j = \sigma^{\{X \mapsto e_j\}}$ for $0 \leq j < 2$. Then in this case

$$(\zeta : e)^{\{X \mapsto e_0\}} = \zeta^{\{X \mapsto e_0\}} : e_0^{\sigma^{\{X \mapsto e_0\}}} = \zeta_0 : e_0^{\sigma_0}$$

is a value state, and hence

$$\zeta^{\{X \mapsto e_1\}} : e_0^{\sigma^{\{X \mapsto e_1\}}} = \zeta_1 : e_0^{\sigma_1}$$

must also be a value state by (**fill**) of definition 2.19. We also claim that $\zeta_1 : e_0^{\sigma_1}$ is closed by lemma 4.2. By assumption $\zeta_1 : e_1^{\sigma_1}$ is closed. Thus we may use the (**ciu**) hypothesis with $\zeta = \zeta_1$, $R = \bullet$, and $\sigma = \sigma_1$ to conclude that

$$\zeta^{\{X \mapsto e_1\}} : e_0^{\sigma^{\{X \mapsto e_1\}}} = \zeta_1 : e_0^{\sigma_1} \preceq \zeta_1 : e_1^{\sigma_1} = \zeta^{\{X \mapsto e_1\}} : e_1^{\sigma^{\{X \mapsto e_1\}}}$$

Now simply observe that

$$(\zeta : e)^{\{X \mapsto e_1\}} = (\zeta : X^\sigma)^{\{X \mapsto e_1\}} = \zeta^{\{X \mapsto e_1\}} : e_1^{\sigma^{\{X \mapsto e_1\}}} = \zeta_1 : e_1^{\sigma_1}.$$

So $(\zeta : e)^{\{X \mapsto e_1\}}\downarrow$.

(CIU $\Leftarrow$) Induction Case: `CIU : CIUL_IH :166`

Now suppose that $(\zeta : e)^{\{X \mapsto e_0\}}\downarrow$ non-trivially. Thus there is a $(\zeta^* : e^*)$ such that $(\zeta : e)^{\{X \mapsto e_0\}} \longrightarrow (\zeta^* : e^*)$ and $|(\zeta^* : e^*)| < |(\zeta : e)^{\{X \mapsto e_0\}}|$. Hence by (**Unif.vi**) either

- **(i)** there is a state $(\zeta' : e')$ such that $(\zeta : e) \longrightarrow (\zeta' : e')$ and $(\zeta' : e')^{\{X \mapsto e_0\}} = (\zeta^* : e^*)$, or else
- **(ii)** $(\zeta : e)$ touches the parameter X.

(CIU $\Leftarrow$) Induction Case (i): `CIU : CIUL :221`

In (**i**) we have by the induction hypothesis that $(\zeta' : e')^{\{X \mapsto e_0\}} \preceq (\zeta' : e')^{\{X \mapsto e_1\}}$, and thus by hypothesis $(\zeta' : e')^{\{X \mapsto e_1\}}\downarrow$. This case is completed by observing that since $(\zeta : e) \longrightarrow (\zeta' : e')$, by (**Unif.v**) we have that $(\zeta : e)^{\{X \mapsto e_1\}} \longrightarrow (\zeta' : e')^{\{X \mapsto e_1\}}$. Thus $(\zeta : e)^{\{X \mapsto e_1\}}\downarrow$ as desired.

We are left with case (**ii**) where $(\zeta : e)$ touches the parameter X. Without loss of generality let $e = R[\,X^\sigma\,]$. We consider two cases depending on whether or not $e_0 \in \mathbf{V}$:

(CIU $\Leftarrow$) Induction Case (ii) ($e_0 \notin \mathbf{V}$):

Since $e_0 \notin \mathbf{V}$ we have by (**Decomposition**) (i.e. lemma 3.8) that either e_0 uniquely decomposes into $R_0[\,r_0\,]$, or else $R_0[\,X_0^{\sigma_0}\,]$. However the latter case is ruled out since e_0 is a term. So $e_0 = R_0[\,r_0\,]$. Now by lemma (**delay**)

$$(\zeta : R[\,X^\sigma\,])^{\{X \mapsto e_0\}} = (\zeta : R[\,e_0^\sigma\,])^{\{X \mapsto e_0\}} = (\zeta : R[\,R_0[\,r_0\,]^\sigma\,])^{\{X \mapsto e_0\}} = (\zeta : R[\,R_0^\sigma[\,r_0^\sigma\,]\,])^{\{X \mapsto e_0\}}$$

Now since $R[\,R_0^\sigma\,]$ is a reduction context, and r_0^σ is still a redex by (**inv**) (i.e. lemma 2.20), we have that $(\zeta : R[\,e_0^\sigma\,])$ does not touch the parameter X and so reduces uniformly by (**Unif.vi**). Thus $(\zeta : R[\,e_0^\sigma\,]) \longrightarrow (\zeta' : e')$ for some $(\zeta' : e')$. Thus by (**Unif.v**) $(\zeta : R[\,e_0^\sigma\,])^{\{X \mapsto e_0\}} \longrightarrow (\zeta' : e')^{\{X \mapsto e_0\}}$ and hence $(\zeta' : e')^{\{X \mapsto e_0\}}\downarrow$, so by the induction hypothesis $(\zeta' : e')^{\{X \mapsto e_1\}}\downarrow$. Now also by (**Unif.v**) $(\zeta : R[\,e_0^\sigma\,])^{\{X \mapsto e_1\}} \longrightarrow (\zeta' : e')^{\{X \mapsto e_1\}}$, consequently we may conclude that $(\zeta : R[\,e_0^\sigma\,])^{\{X \mapsto e_1\}}\downarrow$. Now put $\zeta_1 = \zeta^{\{X \mapsto e_1\}}$, $R_1 = R^{\{X \mapsto e_1\}}$, and $\sigma_1 = \sigma^{\{X \mapsto e_1\}}$. Then by lemma 4.2 we may instantiate (**ciu**) and conclude

$$\zeta_1 : R_1[\,e_0^{\sigma_1}\,] \preceq \zeta_1 : R_1[\,e_1^{\sigma_1}\,].$$

Consequently $\zeta_1 : R_1[\,e_1^{\sigma_1}\,]\downarrow$, as is $(\zeta : e)^{\{X \mapsto e_1\}}\downarrow$ since they are identical.

(CIU $\Leftarrow$) Induction Case (ii) ($e_0 \in \mathbf{V}$): CIU : CIUL_Value :182

Since we are assuming that $(\zeta : e)^{\{X \mapsto e_0\}}\downarrow$ non-trivially, we know that the term $e^{\{X \mapsto e_0\}}$ is not a value, and so must decompose uniquely into $R^*[\,r^*\,]$. Furthermore since e_0 is a value we can find a C containing both X and $\bullet$, and an c such that:

$$R^* = C^{\{X \mapsto e_0\}} \quad r^* = c^{\{X \mapsto e_0\}} \quad e = C^{\{\bullet \mapsto c\}}$$

Although c need not be a redex, and C need not be a reduction context, since in general both could contain occurrences of X.

We say an non-nested occurrence of X is *touched* in e if it occurs either in c, or to the left of the $\bullet$ in C. In what follows we shall write

$$e(\underbrace{X, X, \ldots, X}_{\text{all } \textit{touched} \text{ occurrences}} \mid \overbrace{X, X, \ldots, X}^{\text{all other occurrences}})$$

to indicate the occurrences that are touched and those that are not. We also assume that the touched occurrences are correctly ordered from left to right. In that the leftmost touched occurrence corresponds to the first X in this list, while the right most touched occurrence corresponds to the X just before the $\mid$.

Now the induction hypothesis can be used to show that

$$(\zeta : e(e_0, e_0, \ldots, e_0 \mid X, X, \ldots, X))^{\{X \mapsto e_0\}} \preceq (\zeta : e(e_0, e_0, \ldots, e_0 \mid X, X, \ldots, X))^{\{X \mapsto e_1\}}$$

since by construction $e(e_0, e_0, \ldots, e_0 \mid X, X, \ldots, X)$ does not touch a hole, but rather decomposes into the partially filled C and c. The partial filling of C and c are now, also by construction, a reduction context and a redex. Thus both sides will reduce uniformly by a single step, and the induction hypothesis applies to these reduced expressions.

Thus putting $\zeta_1 = \zeta^{\{X \mapsto e_1\}}$ we may conclude that

$$(\zeta_1 : e(e_0, e_0, \ldots, e_0 \mid e_1, e_1, \ldots, e_1))\downarrow$$

We now prove, by induction on the number occurrences of parameters in $e(X, X, \ldots, X \mid e_1, e_1, \ldots, e_1)$, that

$$(\zeta_1 : e(e_0, e_0, \ldots, e_0 \mid e_1, e_1, \ldots, e_1)) \preceq (\zeta_1 : e(e_1, e_1, \ldots, e_1 \mid e_1, e_1, \ldots, e_1))$$

The base case is trivial, since if there are no parameters to the left of the $\mid$, the lefthand side state is identical to the right hand side state. Thus we need only show how we may reduce the number occurrences of parameters in $e(X, X, \ldots, X \mid e_1, e_1, \ldots, e_1)$ by one.

We do this by considering the expression:

$$e(e_0, e_0, \ldots, e_0, X \mid e_1, e_1, \ldots, e_1)$$

obtained by filling all but the *right most non-nested occurrence* of X in $e(X, X, \ldots, X \mid e_1, e_1, \ldots, e_1)$. By construction this expression is touching the parameter X and thus can be written as $R'[\,X^\sigma\,]$. Thus we may use (**ciu**) to conclude

$$(\zeta_1 : R'[\,e_0^\sigma\,]) \preceq (\zeta_1 : R'[\,e_1^\sigma\,])$$

Or in other words:

$$(\zeta_1 : e(e_0, e_0, \ldots, e_0, e_0 \mid e_1, e_1, \ldots, e_1)) \preceq (\zeta_1 : e(e_1, e_1, \ldots, e_0, e_1 \mid e_1, e_1, \ldots, e_1))$$

It then only remains to show that $e(X, X, \ldots, X, e_1 \mid e_1, e_1, \ldots, e_1)$ satisfies the induction hypothesis, and has one less occurrence of the parameter X. Clearly the number of non-nested parameters decreases by one. To see that $e' = e(X, X, \ldots, X, e_1 \mid e_1, e_1, \ldots, e_1)$ has the desired decomposition, and reduces non-trivially we consider two cases, depending on whether or not e_1 is a value or not. When e_1 is a value the same decomposition remains valid, and as the computation is uniform, non-triviality follows. If e_1 is not a value, then since it is a term it must be of the form: $R_1[r_1]$. Then $C' = e(X, X, \ldots, X, R_1 \mid e_1, e_1, \ldots, e_1)$ and $c' = r_1$ suffices. □

5 Conclusions

The most obvious conclusion to draw from the work reported here is that it is possible to use PVS as a tool in the development of modern operational techniques, and a productive tool at that. It is not hard to see that tools like PVS will, in the future, play an important part in language design, implementation, and even program development itself.

The formalization of the *annotated holes* technique is also a highlight of the work. Providing the unusual technique with a unquestionable basis. However the most important aspect of the work reported here is the way it used PVS to put the paper [24] through a fine tooth comb. The final product [17] is certainly the better for it. To recap: the process of formalizing the CIU theorem revealed three major categories of problems with the published theoretical versions. Unstated closure conditions on the set of values, and set of states. Unstated but necessary uniformity requirements needed for the proof to be carried out. Finally some extra conditions on the terms or contexts considered in the actual CIU proof to ensure that replacing one expression preserves the *closedness* of the computation states involved.

5.1 PVS Statistics

The actual proof of CIU in PVS took the authors (mainly the first one) approximately four months (July 00 through early November 00). The actual machine checked proof involves the proving of two hundred and sixty six (266) distinct facts, and takes PVS two thousand six hundred and sixty six (2662) seconds (44 minutes) of CPU time running on a Linux machine configured with 2 GBytes of main memory and 4×550 MHz Xeon PIII processors. The dump file containing all the PVS definitions, facts, and proofs is 8.362 MBytes and is available from `http://mcs.une.edu.au/~pvs/` [6]. It thus represents roughly four times as much work as was required to prove the Church–Rosser theorem in PVS, as reported in [7].

5.2 Acknowledgements

Prior to this work and the earlier work reported in [7], little use had been made of the abstract datatype facility in PVS. In the course of our work we uncovered several bugs in its implementation. We wish to thank explicitly Sam Owre and Shankar at SRI in Menlo Park for promptly fixing these bugs, providing timely advice, and encouragement, and thus allowing our work to reach fruition.

Our work also made heavy use of the finite set libraries in PVS that have been made freely available by Ricky W. Butler et al [2] and we thank them for their effort.

We also like to thank Carolyn Talcott for encouraging our use of PVS in the style presented here.

References

[1] P. Aczel. A generalized church-rosser theorem, July 1978.

[2] R. W. Butler, B. Dutertre, D. Jamsek, S. Owre and D. Griffioen. Pvs finite set library, 1997. available at `http://pvs.csl.sri.com/pvs/libraries/finite_sets.dmp`.

[3] 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.

[4] 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.

[5] J. Ford. The Church–Rosser theorem in PVS, 2000. PVS dump file (2.4 Megabytes) available at `http://mcs.une.edu.au/~pvs/`.

[6] J. Ford. The CIU theorem in PVS, 2000. PVS dump file (?.? Megabytes) available at `http://mcs.une.edu.au/~pvs/`.

[7] J. Ford and I. A. Mason. Operational Techniques in PVS – A Preliminary Evaluation. In *Proceedings of the Australasian Theory Symposium, CATS '01*, 2001.

[8] 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, 1995.

[9] D. J. Howe. Equality in the lazy lambda calculus. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.

[10] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, Volume 124, Number 2, pages 103–112, 1996.

[11] J. Klop. *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1980.

[12] I. A. Mason. Parametric Computation. In James Harland (editor), *Proceedings of the Australasian Theory Symposium, CATS '97*, pages 103 – 112, 1997.

[13] I. A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, Volume 12, pages 171–201, 1999. An abridged version appears as [12].

[14] I. A. Mason, S. F. Smith and C. L. Talcott. From Operational Semantics to Domain Theory. *Information and Computation*, Volume 128, Number 1, pages 26–47, 1996.

[15] I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, Volume 372 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 1989.

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

[17] I. A. Mason and C. L. Talcott. Feferman–Landin Logic. In W. Sieg, R. Sommer and C. Talcott (editors), *Reflections – A symposium honoring Solomon Feferman on his 70th birthday*. to appear in *Lecture Notes in Logic*, 2000.

[18] J. McKinna and R. Pollack. Pure Type Systems Formalized. In M. Bezem and J. F. Groote (editors), *Typed Lambda Calculi and Applications*, Volume 664 of *Lecture Notes in Computer Science*, pages 289 – 305. Springer Verlag, 1993.

[19] J. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, Volume 23, 1999. An abridged version appeared as [18].

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

[21] T. Nipkow. Higher-order critical pairs. In *Sixth Annual Symposium on Logic in Computer Science*. IEEE, 1991.

[22] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, Volume 1, pages 125–159, 1975.

[23] C. L. Talcott. *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation*. Ph.D. thesis, Stanford University, 1985.

[24] C. L. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1996.