Establishing a General Context Lemma in PVS
Jonathan Ford & Ian A. Mason
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 analogous 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 $X$. $P_\omega(X)$ is the set of finite subsets of $X$. $F\text{map}[X_0, X_1]$ 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 $\text{app}(f, x)$ for the application $f(x)$ of the function $f$ to the argument $x$. We write $\text{Dom}(f)$ for the domain of a function and $\text{Rng}(f)$ for its range. Thus if $f \in F\text{map}[X_0, X_1]$, then $\text{Dom}(f) \in P_\omega(X_0)$. For any function $f$, $f\{x \mapsto x'\}$ is the function $f'$ such that $\text{Dom}(f') = \text{Dom}(f) \cup \{x\}$, $f'(x) = x'$, and $f'(z) = f(z)$ for $z \neq x, z \in \text{Dom}(f)$. Also $f[X]$ is the restriction of $f$ to $X$: the function $f'$ such that $\text{Dom}(f') = \text{Dom}(f) \cap X$ and $f'(x) = f(x)$ for $x \in \text{Dom}(f')$. $N = \{0, 1, 2, \ldots\}$ is the set of natural numbers and $i, j, n, n_0, \ldots$ range over $N$. 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 $\bigcup_{n \in N} F_n(Z^n)$. For example if $\Omega$ is a ranked set of operator symbols, then the terms over $\Omega$ can be defined inductively by (as the least solution to) the equation: $T_\Omega = \Omega_n(T_\Omega^n)$. Unabbreviated, this equation reads:

$$T_\Omega = \bigcup_{n \in N} \{ \omega(t_1, \ldots, t_n) \mid \omega \in \Omega_n \land t_1 \in T_\Omega \land \ldots \land t_n \in T_\Omega \land 1 \leq i \leq n \}.$$  

Finally we use the traditional $\forall$ 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 $\lambda$s. 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 $\lambda$ 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, $X$, of variables, and $P$ of parameters:

$$X = \{x_i \mid i \in \mathbb{N}\} \quad P = \{X_i \mid i \in \mathbb{N}\}$$

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

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

**Definition 2.1 (E, L, S, F):**

The set of expressions, $E$, and the set of $\lambda$-abstractions, $L$, the set of value substitutions, $S$, and the set of parameter substitutions (fillings), $F$ are defined as the least sets satisfying the following equations:

$$E = X \cup P^S \cup A \cup L \cup O_n(E^n) \quad L = \lambda X.E \quad S = F\text{map}[X,V] \quad F = F\text{map}[P,E]$$

We let a range over $A$, $x, y, z$ range over $X, X, Y, Z$ range over $P$, $e$ range over $E$, $\varphi$ range over $L$, $\sigma$ range over $S$, and $\Sigma$ range over $F$.

$P^S$ is the set of parameters, annotated or decorated by value substitutions. Value substitutions, $S$, are finite maps from variables to value expressions. The domain of a substitution is written as $\text{Dom}(\sigma) \subseteq X$, 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, $\sigma$, 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, $\Sigma$, 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 $\sigma$ and $\Sigma$. Also the entire development is parametric in a set of atoms and operations (which includes the binary $\text{app}$, together with the arities of those operations. This theory is appropriately named $\text{Landin}$, and can be found in FLL - version - 03 :11 Implementation of the syntax within PVS is a straightforward use of the $\text{datatype with subtypes}$ 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 natural 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 contracted.

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 definitions. 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 omitting 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 syntactic 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 auxiliary 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 facility 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** (\(\text{rank}(e)\)):

\[
\text{rank}(e) = \begin{cases} 
0 & \text{if } e \in \mathbf{X} \cup \mathbf{A}, \\
1 + \text{rank}(\alpha_0) & \text{if } e = \lambda x.\alpha_0, \\
2 + \max\{\text{rank}(e_i) \mid 1 \leq i \leq n\} + n & \text{if } e = \theta(e_1, \ldots, e_n), \\
1 + \max\{\text{rank}(\sigma(x)) \mid x \in \text{Dom}(\sigma)\} & \text{if } e = X^\sigma
\end{cases}
\]

### 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** (\(\text{FV}(e), \text{FP}(e)\)):

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

\[
\text{FV}(X^\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} \text{FV}(x) \quad \text{FP}(X^\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} \text{FP}(x) \cup \{i\}
\]

The free variables \(\text{FV}(e)\) are defined in \(\text{vars} : 1-95\), while the parameters, \(\text{FP}(e)\), are defined in \(\text{vars} : 38\). We extend these to substitutions in the obvious fashion:

\[
\text{FV}(\Delta) = \bigcup_{\delta \in \text{Dom}(\Delta)} \text{FV}(\Delta(\delta)) \quad \text{FP}(\Delta) = \bigcup_{\delta \in \text{Dom}(\Delta)} \text{FP}(\Delta(\delta)) \quad \text{for } \Delta \in \mathbf{S} \cup \mathbf{F}.
\]

One last derived piece of notation concerning annotated parameters appearing in expressions. The set of trapped variables, \(\text{Traps}(e)\), 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** (\(\text{Traps}(e)\)):

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

\[
\text{Traps}(X^\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} \text{Traps}(\sigma(x)) \cup \text{Dom}(\sigma)
\]

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 \(e^{\{x := y\}}\). 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:

\[
e^{\{x := y\}} : \{e_0 \in \mathbf{E} \mid \text{rank}(e_0) = \text{rank}(e)\}
\]
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 $e^{\{x\mapsto y\}}$):**

$(\lambda z.e)^{\{x\mapsto y\}} = \lambda z.((e^{\{x\mapsto \nu}\})^{\{x\mapsto y\}})$ for $\nu$ fresh, i.e. $\nu \not\in \text{FV}(e) \cup \{x, y\}$.

Note that we always rename the 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):**

$\text{new} : \mathcal{P}_{\omega}(\mathbb{N}) \mapsto \mathbb{N}$ satisfying $(\forall W \in \mathcal{P}_{\omega}(\mathbb{N}))(\text{new}(W) \not\in W)$

**2.2.7 Substitution**

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

**Definition 2.7 (Substitution $e^\sigma$):**

$e^\sigma$ is the result of simultaneous substitution of free occurrences of $x \in \text{Dom}(\sigma)$ 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.

$(\lambda z.e)^\sigma = \lambda z.((e^{\{x\mapsto \nu\}})^\sigma)$ for $\nu$ fresh, i.e. $\nu \not\in \text{FV}(e) \cup \text{FV}(\sigma)$.

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

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):** $e^{\{x\mapsto y\}} = e^{\{x\mapsto y\}}$

**2.2.8 Filling**

**Definition 2.9 (Filling $e^\Sigma$):**

$e^\Sigma$ is the result of simultaneous substitution of decorated occurrences of $X \in \text{Dom}(\Sigma)$ 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, $(X^\sigma)^\Sigma = X^{(\sigma^\Sigma)}$ if $X \in \text{Dom}(\Sigma)$, and $X^\sigma^\Sigma$
otherwise, where $\sigma^\Sigma$ is defined point-wise: $\sigma_0^\Sigma = \{x \mapsto \sigma(x)^\Sigma \mid x \in \text{Dom}(\sigma_0)\}$. In the case of λ-abstractions, we
define parameter substitution exactly as we would value substitution:

$(\lambda x.e)^\Sigma = \lambda x.((e^{\{x\mapsto \nu\}})^\Sigma)$ for $\nu$ fresh, i.e. $\nu \not\in \text{FV}(e) \cup \text{FV}(\Sigma)$.

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

**2.2.9 $\alpha$ 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 $\equiv$ [13]:

**Definition 2.10 (\equiv):**

\[
\begin{align*}
\text{(Base)} & \quad \frac{\alpha \equiv \nu}{\nu \equiv \nu} \quad \text{provided } \nu \in A \cup X \\
\text{(Op)} & \quad \frac{e_0 \equiv e_0' \quad \ldots \quad e_n \equiv e_n'}{\vartheta(e_0, \ldots, e_n) \equiv \vartheta(e_0', \ldots, e_n')} \\
\text{(Subst)} & \quad \frac{\text{Dom}(\Delta_0) = \text{Dom}(\Delta_1) \quad \Delta_0(\delta) \equiv \Delta_1(\delta) \quad \forall \delta \in \text{Dom}(\Delta_1)}{\Delta_0 \equiv \Delta_1}
\end{align*}
\]

$\Delta_0, \Delta_1 \in S$ or $\Delta_0, \Delta_1 \in F$
There is substantial leeway in the precise way one formalizes $\alpha$. 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 $\text{Alpha:1-450}$ where we actually formalized the notion of being a proof in the above system, and defined $\equiv$ by
\[
eq_0 \equiv e_1 \triangleq (\exists \Gamma \text{ a proof in the above system })(\Gamma \vdash e_0 \equiv e_1)
\]

Establishing facts such as transitivity, symmetry, and reflexivity required the formal manipulation of proof objects. The weak form was developed in $\text{WeakAlpha:1-48}$ 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 $\text{WeakAlpha:85}$. 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 ($\alpha$):**
1. $e_0 \equiv e_1 \Rightarrow \text{rank}(e_0) = \text{rank}(e_1)$.
2. $\equiv$ is reflexive.
3. $\equiv$ is symmetric.
4. $\equiv$ is reflexive.
5. $e_0 \equiv e_1 \Rightarrow \text{FV}(e_0) = \text{FV}(e_1)$.

**2.2.10 Substitution preserves $\alpha$ Equivalence**

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

**Lemma 2.12 (AlphaSubst):**

\[
eq_0 \equiv e_1 \& \sigma_0 \equiv \sigma_1 \Rightarrow \neq_0^{\sigma_0} \equiv \neq_1^{\sigma_1}
\]

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):**

1. $(e_0 \equiv e_1 \& e_0^{\nu} \equiv e_1^{\nu}) \Rightarrow e_0^{\{x \mapsto e_0^{\nu}\}} \equiv e_0^{\{x \mapsto e_1^{\nu}\}}$
2. $(\nu_0 \notin \text{FV}(e) \& \nu_0 \neq \nu_1) \Rightarrow (e_0^{\nu_0 \mapsto \nu_1})^{\{x \mapsto e\}} \equiv (e_0^{\nu_0 \mapsto \nu_1})^{\{x \mapsto e\}}$
3. $(e_0^{\{x \mapsto \nu\}})^{\nu \mapsto \nu_1} \equiv e_0^{\{x \mapsto \nu_1\}}$ for $\nu \notin \text{FV}(e_0)$

However the actual proof first must establish (by simultaneous induction) the corresponding three properties for renaming rather than substitution $\text{AlphaSubst:AlphaPropLemma:70}$. 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 $\alpha$ congruence.

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

**Lemma 2.14 (SubstProps):**

1. $(\text{Dom}(\sigma_0) \setminus \text{Dom}(\sigma_1)) \cap \text{FV}(e) = \emptyset \Rightarrow (e^{\sigma_0})^{\sigma_1} \equiv e^{(\sigma_0 \sigma_1)}$
2. $x \notin \text{FV}(e) \Rightarrow (e^{\{y \mapsto x\}})^{\{x \mapsto e_0\}} \equiv e^{\{y \mapsto e_0\}}$
3. $\text{FV}(e) \cap \text{Dom}(\sigma) = \emptyset \Rightarrow e \equiv e'$

**Proof:**
2.2.11 Filling preserves $\alpha$ Equivalence

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

**Lemma 2.15 (AlphaFill):** 
\[
\begin{align*}
\; & e_0 \equiv e_1 \land \Sigma_0 \equiv \Sigma_1 \Rightarrow e_0^{\Sigma_0} \equiv e_1^{\Sigma_1}. \\
\end{align*}
\]

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

**Lemma 2.16 (FillSwap):** 
\[
\begin{align*}
\forall x \notin \text{FV}(\Sigma) \Rightarrow (e^\Sigma)^{\{x \rightarrow y\}} & \equiv (e^{\{x \rightarrow y\}})^\Sigma. \\
\end{align*}
\]

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):**

1. $\text{Dom}(\Sigma) \cap \text{FP}(e) = \emptyset \Rightarrow e \equiv e^{\Sigma}$.
2. $e_0 \equiv e_1 \Rightarrow \text{FP}(e_0) = \text{FP}(e_1)$.
3. $(\bigwedge_{i < 2} \text{FP}(e_i) = \emptyset \land X_0 \neq X_1) \Rightarrow (e^{\{X_0 \rightarrow \epsilon \epsilon_0\}})^{\{X_1 \rightarrow \epsilon \epsilon_1\}} \equiv e^{\{X_1 \rightarrow \epsilon \epsilon_1\}} |_{i < 2}$
4. $(\bigwedge_{i < 2} \text{FP}(e_i) = \emptyset \land X_0 \neq X_1) \Rightarrow e^{\{X_0 \rightarrow \epsilon \epsilon_0\}} \cup \{X_1 \rightarrow \epsilon \epsilon_1\} \equiv e^{\{X_1 \rightarrow \epsilon \epsilon_1\}} |_{i < 2}$

**Definition 2.18 (Term, Closed):**

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 $\alpha$ equivalence, and that they are both preserved under filling (since they remain unchanged).

2.2.12 Values

**Definition 2.19 (Value Expressions (V)):**

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

1. (triv) $\mathbf{V}$ is closed under $\equiv$
2. (vsub) $v \in \mathbf{V} \Rightarrow v^\theta \in \mathbf{V}$
3. (inst) $v \in \mathbf{V} \Rightarrow v^\Sigma \in \mathbf{V}$
4. (dich) $e^\Sigma \in \mathbf{V} \Rightarrow (e \in \mathbf{V}) \lor (e = X^\theta \land (X) \in \mathbf{V})$

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

We can now define $\text{Values : NewE : 37}$ 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 $\text{NewE}$. Once defined we must also show that renaming, value substitution, and filling all preserve the defining property of $\text{NewE}$. 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):**

\[
\begin{align*}
\; & e^\theta \in \mathbf{V} \Rightarrow e \in \mathbf{V} \\
\end{align*}
\]

**Proof:** Pick $\alpha_0$ such that $e_0^\theta \in \mathbf{V}$, and let $X$ be a fresh parameter. Put $e = X^\theta$, $\Sigma = \{X \mapsto \alpha_0\}$. Then

\[
\begin{align*}
\; & e^\Sigma = (X^\theta)^\Sigma = \Sigma(X)^{\theta^\Sigma} = \Sigma(X)^{\theta} = e_0^\theta \in \mathbf{V} \\
\end{align*}
\]

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

\[
\square
\]
3 Semantics of Terms

3.1 Operational Semantics

In a $\lambda$-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 $\sigma$ 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 ($Z$)):**

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

- (triv) $Z$ is closed under $\equiv$
- (par) $\zeta \in Z \Rightarrow o \in \text{FP}(\zeta)$
- (vsub) $\zeta \in Z \Rightarrow \zeta^o \in Z$ assuming $o \not\in \text{FP}(\sigma)$
- (inst) $\zeta \in Z \Rightarrow \zeta^2 \in Z$ assuming $o \not\in (\text{Dom}(\Sigma) \cup \text{FP}(\Sigma))$

$\zeta$ ranges over $Z$ and $o \in Z$ is the empty state expression.

**Definition 3.2 (Computation States (CS)):**

CS $\widehat{=} Z : E$ is the set of computation states. We let $S$ range over CS and let $\zeta : e$ be the state with state context $\zeta$ and expression $e$. The computation state $\zeta : e$ is said to be a value state iff $e \in V$. Given a state $\zeta : e$, we associate a corresponding expression by filling the state parameter in the context with the expression, i.e. $\zeta^{(o+\sigma)}$. A state is closed just if its corresponding expression is closed, in other words if $\zeta^{(o+\sigma)}$ has no free parameters or variables. Application of value and parameter substitutions to states is defined by in the obvious way: $(\zeta : e)^o = \zeta^o : e^o$ and $(\zeta : e)^2 = \zeta^2 : e^2$. Note that by (vsub) (inst) these are only meaningful if $o \not\in \text{Dom}(\sigma)$ and $o \not\in (\text{Dom}(\Sigma) \cup \text{FP}(\Sigma))$.

**Definition 3.3 (Reduction ($\longrightarrow$, $\longrightarrow^*$) and Definedness ($\downarrow$)):**

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

- Definedness:

  $$ (\zeta : e) \downarrow \iff (e \in V) \lor (\zeta : e \longrightarrow^* \zeta' : v) $$

- Approximation:

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

- Equidefined:

  $$ (\zeta_0 : e_0) \succeq (\zeta_1 : e_1) \iff ((\zeta_0 : e_0) \downarrow \Rightarrow (\zeta_1 : e_1) \downarrow) $$

- Equivalued:

  $$ (\zeta_0 : e_0) \simeq (\zeta_1 : e_1) \iff (\zeta_0 : e_0) \succeq (\zeta_1 : e_1) \wedge (\zeta_0 : e_0) \succeq (\exists v \in V, \zeta \in Z)(\bigwedge_{j<2} (\zeta_j : e_j) \longrightarrow^* \zeta : v) $$

- Length:

  $|\zeta : e|$ is the least $n \in N$ 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 $\lambda$-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 $\theta(v_1, \ldots, v_n)$ may be value expressions.

**Definition 3.4 (Redexes ($E_r$)):**

The set of redexes, $E_r$, is defined by:

$$ E_r = O_n(V^n) - V $$
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):** Redexes satisfy the following uniformity conditions:

1. **(triv)** \( E_r \) is closed under \( \equiv \)
2. **(vsub)** \( r \in E_r \Rightarrow r^\sigma \in E_r \)
3. **(inst)** \( r \in E_r \Rightarrow r^{\Sigma} \in E_r \)

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)):**

The set of reduction contexts, \( R \), is the subset of \( E \) defined by

\[
R = \{ \bullet \} \cup O_{m+n+1}(\{v \in V \mid v \not\in \text{FP}(v)\})^m, R, \{e \in E \mid e \not\in \text{FP}(e)\}^n)
\]

We let \( R \) range over \( R \). We adopt the convention of writing \( R[e] \) instead of \( R^{\bullet + e_1} \).

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 \( \lambda \)-language to another. Also note that \( R \) will satisfy a similar set of uniformity conditions as those satisfied by states (Definition 3.1):

**Lemma 3.7 (Rcx Uniformity):**

Reduction contexts satisfy the following uniformity conditions:

1. **(triv)** \( R \) is closed under \( \equiv \)
2. **(par)** \( R \in R \Rightarrow \bullet \in \text{FP}(R) \)
3. **(vsub)** \( R \in R \Rightarrow R^\sigma \in R \) assuming \( \bullet \not\in \text{FP}(\sigma) \)
4. **(inst)** \( R \in R \Rightarrow R^\Sigma \in R \) assuming \( \bullet \not\in (\text{Dom}(\Sigma) \cup \text{FP}(\Sigma)) \)

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):** For any \( \lambda \)-language, if \( e \in E \) then either \( e \in V \) or \( e \) can be written uniquely as either

1. \( R[r] \) where \( R \) is a reduction context and \( r \in E_r \), or else
2. \( R[X^\sigma] \) where \( R \) is a reduction context, and \( X^\sigma \in P^S \) 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 uniqueness aspect of this lemma. A simple counterexample is the following:

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

### 3.2 Uniform Semantics

We now specify what we mean by a \( \lambda \)-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)):**

A \( \lambda \)-language is said to have *uniform semantics* if it satisfies the following:
(i) Functional modulo $\alpha$ and implicit bindings:

$$\left( \bigwedge_{j < 2} (\zeta_j : e_j) \equiv (\zeta'_j : e'_j) \right) \Rightarrow \left( (\zeta_0 : e_0 \rightarrow \zeta_1 : e_1) \Rightarrow (\zeta'_0 : e'_0 \rightarrow \zeta'_1 : e'_1) \right)$$

$$\left( \bigwedge_{j < 2} \zeta : e \rightarrow \zeta_i : e_i \right) \Rightarrow \left( \zeta'_0 \equiv_{\alpha \rightarrow e} \zeta'_1 \right).$$

(ii) Uniform in value substitutions:

$$\zeta : e \rightarrow \zeta' : e' \Rightarrow (\zeta : e)^\sigma \rightarrow (\zeta' : e')^\sigma$$

provided $\text{Dom}(\sigma) \cap (\text{Traps}(\zeta) \cup \text{Traps}(\zeta')) = \emptyset$ and $\sigma \not\in \text{FP}(\sigma)$.

(iii) State evaluation:

$$\zeta : e \uparrow \sigma : \zeta'$$

(iv) Well-founded:

$$\left( \zeta : e \rightarrow \zeta' : e' \wedge \zeta : e \downarrow \right) \Rightarrow \left( \zeta' : e' \downarrow \right) \text{ for any } \Sigma \in \text{F} \text{ with } \sigma \alpha \neq (\text{Dom}(\Sigma) \cup \text{FP}(\Sigma))$$

(v) Parametric:

$$\zeta_0 : e_0 \rightarrow \zeta_1 : e_1 \Rightarrow (\zeta_0 : e_0) \Sigma \rightarrow (\zeta_1 : e_1) \Sigma$$

for any $\Sigma \in \text{F}$ with $\sigma \not\in \text{FP}(\Sigma)$.

(vi) Dichotomy:

Assuming $\sigma \not\in \text{Dom}(\Sigma)$, if $\zeta : e \Sigma \rightarrow \zeta' : e'$ then either

$$\zeta : e \text{ touches a parameter in the domain of } \Sigma, \text{ or}$$

$$\zeta : e \rightarrow \zeta_1 : e_1, \text{ for some } \zeta_1 : e_1 \text{ such that } \zeta' : e' \equiv \alpha (\zeta_1 : e_1).$$

(vii) Closure:

$$\left( \zeta_0 : e_0 \rightarrow \zeta_1 : e_1 \right) \Rightarrow \text{FP}(\zeta_1) \subseteq \text{FP}(\zeta_0) \wedge (\sigma \not\in \text{FP}(e_0) \Rightarrow \sigma \not\in \text{FP}(e_1))$$

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 $\alpha$ and $\bullet$.

We let $C$ range over expressions with $X$ as the only free parameter. Such expressions play the role of traditional $\lambda$-calculus contexts, and we extend our convention, stated in definition 3.6, of sometimes writing $C[ e ]$ instead of $C[ ^\alpha X \rightarrow e ]$. Note however that for example the traditional context $\lambda z \cdot \text{app}(y, \lambda x \cdot \text{app}(1, z))$ does not correspond to $\lambda z \cdot \text{app}(y, \lambda x \cdot \text{app}(X, z))$ but rather to one where the trappings have been made explicit at the occurrence of the $X$ parameter: $\lambda z \cdot \text{app}(y, \lambda x \cdot \text{app}(X[ ^\alpha X \rightarrow z \rightarrow e ] z )$.

**Definition 3.10 (Approximation $e_0 \sqsubseteq e_1$, Equivalence $e_0 \cong e_1$):**

For terms $e_0$, $e_1$ define

$$e_0 \sqsubseteq e_1 \Leftrightarrow (\forall C \quad C \quad C[ e_0 ] \quad C[ e_1 ] \text{ closed} \quad (\sigma : C[ e_0 ] \preceq \sigma : C[ e_1 ]))$$

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

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):**

For a \( \lambda \)-language with uniform semantics:

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

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

CIU is an acronym for closed instances of uses, since the value substitution, \( \sigma \), closes the \( e_i \), while the reduction context, \( R \), represents a use of the value returned, in the appropriate state, \( \zeta \). Thus only the value of the expressions, \( e_i \), are observed. Although the theorem holds without the added assumption that \( \text{FV}(e_0) = \text{FV}(e_1) \), 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):**

Suppose \( e \) satisfies \( \text{FV}(e) = \emptyset \). \( \text{FP}(e) \subseteq \{X_0, X_1\} \), \( \text{FV}(e_0) = \text{FV}(e_1) \), \( e^\{X_0 \rightarrow e_0, X_1 \rightarrow e_1\} \) is closed, and \( e^\{X_0 \rightarrow e_0, X_1 \rightarrow e_1\} \) is also closed.

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

\[
e = \lambda y_0. \lambda y_1. X_0^\{y_1 \rightarrow y_1, y_0 \rightarrow x_1\}^\{y_0 \rightarrow y_0, y_1 \rightarrow y_1, y' \}
\]

\[
e^\{X_0 \rightarrow e_0, X_1 \rightarrow e_1\} = \lambda y_0. \lambda y_1. y_0, \quad e^\{X_0 \rightarrow y_1, X_1 \rightarrow y_1\} = \lambda y_0. \lambda y_1. y_1, \quad \text{but} \quad e^\{X_0 \rightarrow y_0, X_1 \rightarrow y_1\} = \lambda y_0. \lambda y_1. e'.
\]

To see that the first three are all necessary consider:

\[
e = \lambda y_1. X_0^\{y_1 \rightarrow y_1, y_0 \rightarrow y_2\}^\{y_0 \rightarrow y_2, y_1 \rightarrow \lambda y_0. y_0\}
\]

\[
e^\{X_0 \rightarrow e_0, X_1 \rightarrow e_1\} = \lambda y_1. \lambda y_2. y_1, \quad e^\{X_0 \rightarrow y_1, X_1 \rightarrow y_1\} = \lambda y_1. y_1, \quad \text{but} \quad e^\{X_0 \rightarrow y_2, X_1 \rightarrow y_1\} = \lambda y_1. \lambda y_2. \lambda y_3. y_0.
\]

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):**

If \( e \) is a term, then \( R\{X^\sigma \}^\{X \rightarrow e\} = R\{e^\sigma \}^\{X \rightarrow e\} \).

4.2 The CIU Proof

The fact that one can present a syntactic reduction system for imperative \( \lambda \)-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 there. 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 \( \Rightarrow \))

The (CIU \( \Rightarrow \)) direction is relatively simple, and requires producing a context \( C \) such that \( C\{e_j\} \) evaluates to \( \zeta : R\{e_j^\sigma\} \). Suppose that \( e_0 \subseteq e_1 \), and choose \( \zeta, R, \sigma \) 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 \( \zeta \) is \( \sigma \), and that \( \sigma(x) \) is a term, for each \( x \in \text{Dom}(\sigma) \). Extend \( \sigma \) to \( \hat{\sigma} \) by mapping each \( x \in \text{Trap}(\zeta) \) to \( \text{Dom}(\sigma) \) to itself. Note that \( e_j^\hat{\sigma} = e_j^\sigma \). Choose an new parameter \( X \) and consider the expression \( C = \zeta\{x \rightarrow R\{x \rightarrow X^\sigma\}\} \).

**Lemma 4.4:**

\[
C\{e_j\} = (\zeta\{x \rightarrow R\{x \rightarrow X^\sigma\}\})\{X \rightarrow e_j\} = \zeta\{x \rightarrow R\{x \rightarrow e_j^\sigma\}\}
\]
Now by this lemma $\sigma : C_\epsilon \sqsubseteq C_\epsilon$. In fact, by definition 3.10 we have that $\sigma : C_\epsilon \sqsubseteq C_\epsilon$. Therefore $\sigma : C_\epsilon \sqsubseteq C_\epsilon$ by lemma 4.4, and (Unif.iii). Thus $\zeta : R_\epsilon \zeta \sqsubseteq R_\epsilon$ as desired. □

**Proof:** (CIU ⇔

Assume that

(\text{ciu}) \quad (\forall \zeta, R, \sigma \mid \bigwedge_{j<2} \zeta : R_\epsilon \zeta \text{ closed})(\zeta : R_\epsilon \zeta \zeta : R_\epsilon \zeta).

We prove

(\forall \zeta, e \mid \bigwedge_{j<2} \zeta : e \{X_\rightarrow e_j\} \text{ closed})(\zeta : e \{X_\rightarrow e_0\} \zeta : e \{X_\rightarrow e_1\})

by induction on the length of the computation of $(\zeta : e)\{X_\rightarrow e_0\}$.

The proof is relatively straightforward modulo the hard case. The hard case is when $e_0$ is a value, and $(\zeta : e)\{X_\rightarrow e_0\}$ 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 $\lambda$-expression, or in the range of a substitution annotating a parameter.

(\text{ciu}) \quad \text{Base Case:}

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

$$(\zeta : e)\{X_\rightarrow e_0\} = \zeta\{X_\rightarrow e_0\} : e_0^{\sigma}\{X_\rightarrow e_0\} = \zeta_0 : e_0^{\sigma}$$

is a value state, and hence

$$\zeta\{X_\rightarrow e_1\} : e_0^{\sigma}\{X_\rightarrow e_1\} = \zeta_1 : e_0^{\sigma_1}$$

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 $\zeta_1 : e_0^{\sigma_1}$ is closed. Thus we may use the (ciu) hypothesis with $\zeta = \zeta_1$, $R = \bullet$, and $\sigma = \sigma_1$ to conclude that

$$\zeta\{X_\rightarrow e_1\} : e_0^{\sigma}\{X_\rightarrow e_1\} = \zeta_1 : e_0^{\sigma_1} \zeta_1 : e_0^{\sigma_1} = \zeta_1 : e_0^{\sigma_1}$$

Now simply observe that

$$\zeta\{X_\rightarrow e_1\} = \zeta\{X^\sigma\} \{X_\rightarrow e_1\} = \zeta\{X_\rightarrow e_1\} : e_0^{\sigma}\{X_\rightarrow e_1\} = \zeta_1 : e_0^{\sigma_1}.$$
since by construction \( e(e_0, e_0, \ldots, e_0) \models X, X, \ldots, X \) 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

\[
(\zeta_1 : e(e_0, e_0, \ldots, e_0) \models e_1, e_1, \ldots, e_1) \downarrow
\]

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

\[
(\zeta_1 : e(e_0, e_0, \ldots, e_0 \models e_1, e_1, \ldots, e_1)) \preceq (\zeta_1 : e(e_1, e_1, \ldots, e_1 \models e_1, e_1, \ldots, e_1))
\]

The base case is trivial, since if there are no parameters to the left of the \( \downarrow \), 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 \( e(X, X, \ldots, X \models e_1, e_1, \ldots, e_1) \) by one.

We do this by considering the expression:

\[
e(e_0, e_0, \ldots, e_0, X \models e_1, e_1, \ldots, e_1)
\]

obtained by filling all but the right most non-nested occurrence of \( X \) in \( e(X, X, \ldots, X \models e_1, e_1, \ldots, e_1) \). By construction this expression is touching the parameter \( X \) and thus can be written as \( R'[X^0] \). Thus we may use (ciu) to conclude

\[
(\zeta_1 : R'[e_0^\sigma]) \preceq (\zeta_1 : R'[e_1^\sigma])
\]
It then only remains to show that $\epsilon(X, x, \ldots, x, e_1 \mid e_1, e_2, \ldots, e_i)$ 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 $\epsilon' = \epsilon(X, x, \ldots, x, e_1 \mid e_1, e_2, \ldots, e_i)$ 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 $C' = \epsilon(X, X, \ldots, X, R_1 \mid e_1, e_2, \ldots, e_i)$ and $\epsilon' = r_1$ 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 an 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 4x550 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


