Formal Foundations of Operational Semantics

Jonathan Ford and Ian A. Mason
School of Mathematics, Statistics, & Computing Science
University of New England
Armidale, NSW 2351 Australia
{jford,iam} @turing.une.edu.au

Abstract. In this paper we report on the results of a sophisticated and substantial use of PVS to establish a recent result in operational semantics. The result we establish is a context lemma for operational equivalence for very wide class of programming languages, known as the CIU theorem. The proof uses the annotated holes technique to represent contexts and compute with them. Thus this paper demonstrates that that it is possible to use PVS as a tool in the development of modern operational techniques, and a productive tool at that. The process of formalizing the CIU theorem revealed several gaps in published proof. The proof of the CIU theorem in PVS took approximately six months to develop. The actual machine checked proof involves the proving of around one thousand facts, and takes PVS slightly less than three hours of CPU time running on a Linux machine configured with 2 GBytes of main memory and four 550 MHz Xeon PIII processors.

Keywords: λ-calculus, contexts, operational semantics, theorem proving, PVS.

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

1. Introduction

In this paper we describe the use of a computer-based proof assistant to verify the results presented in a foundational paper [26] dealing with the semantics of a wide class of programming languages. The results presented here lay the groundwork for developing and implementing a logic for specifying and reasoning about programs written in this general class of programming languages; the actual logic is described in more detail in [19].

The work described here actually represents part of the development of this logic prior to its actual implementation. The formal proof assistant that we use is the Prototype Verification System (PVS) developed at SRI [2]. This paper reports 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 required 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. The results presented here exemplify the possible benefit of serious formalization in contrast to standard mathematical practice, which typically leaves much unsaid.

This paper has several, complementary, purposes. On the one hand it is a detailed presentation of the proof of the results found in [26], on the other hand it is a road map for the actual mechanized proof [6], and the issues raised in its development. Hopefully it may also be of assistance as a how-to-guide for others who undertake similar formalization projects. We also try to 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 [26, 19], to contrast it with the word *formal* that refers to the analogous notion, as formalized in the corresponding PVS development.

1.1. The Subject Matter

Starting from Landin's original proposal [14] that a programming language consists of the λ-calculus augmented by primitive operations, Carolyn Talcott and the second author set about studying the nature of operational equivalence in such languages. Operational equivalence is a natural equivalence between programs in a particular programming language. Two programs/phrases are equivalent in this sense if they cannot be distinguished by textually replacing one, in a larger surrounding program, by the other. In other words two pieces of operationally equivalent code may be used interchangeably, without affecting the behaviour of the enclosing program. The enclosing program is usually thought of as a context, thus operational equivalence is usually paraphrased as being equivalent in all enclosing contexts.

Much work has been done to develop methods for reasoning about operational equivalence. Methods developed 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 [22], 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.

In 1986-1987 syntactic reduction systems for imperative λ-calculi were discovered independently in [18] and [4]. As well as being conceptually elegant, these systems have also provided the necessary basis for several key results and proofs. In addition to eliminating messy isomorphism considerations, to deal with arbitrary choice of names of newly allocated structures, this syntactic presentation of the semantics was a key step leading to the formulation of a context lemma for languages with effects: the so-called CIU theorem, first presented in [17]. CIU is an acronym for **C**losed **I**nstances of **U**ses. There are now several proofs of this result in the literature. The theorem first appeared in [18] 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 [26]. A second, distinct, proof was presented in [11] that simply shows that the CIU relation is a congruence. This proof also appears in a more general setting in [16].

We concentrate on the most general form of the lemma that appears in [26], and the proof that we formalize is the one presented there. The lemma states that for two expressions to be operationally equivalent it is sufficient that the values returned by evaluating substitution instances of the expressions are indistinguishable by further use. In other words we need only look at whether or not the values of the expressions are distinguishable, assuming that we can replace their free variables by values.

1.2. THE APPROACH

In our formalization effort we have taken great pains to formalize the actual theoretical treatment, rather than adapting it to the tastes of the formal system we are using, in this case PVS. We use named variables, and formalize the notion of being equivalent modulo the renaming of bound variables: α-equivalence. In this regard we were almost completely successful, only on two occasions was it necessary to deviate, slightly, from the exact theoretical treatment. We will mention them in the narrative. In this sense we made no use of the *tricky* representations that McKinna and Pollack discuss [21]. In our earlier work [8, 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 also be seen as an attempt to reconcile informal mathematical/theoretical 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. 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 as a benchmark, see [8] for a survey.

1.3. THE SIGNIFICANCE

We should also point out that the formalization process is part of a grander scheme: Producing a tool for reasoning, in a very general sense, about programs written in λ-languages. We are currently building upon this formalization effort with an implementation, using Rewriting Logic, of the logic described in [19]. The results of this effort will be described elsewhere.

1.4. 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 a 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. 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:

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

We also use the notation $A \stackrel{\triangle}{=} B$ to emphasize that A is *defined to be equal to* B, rather than being equal as a consequence of the natures of A and B. In the body of the paper we use the notation `file` :<line number> to refer to the particular line in the file, whose name is `file.pvs`, that may be downloaded and unpacked from [6]. So for example `CIU` :511 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 `Computation` : `Rcx_Fill_Rcx` :281 refers to the lemma `Rcx_Fill_Rcx` in the file `Computation.pvs` that lies on line 281 of the unpacked version of [6].

Finally we use Don Knuth's dangerous bend symbol [12] to indicate where the formal development uncovered gaps in the theoretical development.

2. An Overview of PVS

PVS is a verification system developed by SRI and draws on almost 20 years of experience in designing and implementing such systems. PVS is a classical logic formulation of (a generalization of) higher order logic. PVS has a very sophisticated type system, which includes predicate subtypes, dependent types, parameterized theories, a mechanism for defining abstract data types, numbers (both real and integral), ordinals, and forms of induction up to ϵ_0. This power, of course, comes at a price. In this case the price is that typechecking is undecidable. As a consequence the typechecker generates TCCs (Type-Correctness Conditions) that need to be discharged either by the PVS system or its user. The advantage of the PVS type system is that preconditions and postconditions can be incorporated into a function's type. A precondition is incorporated by declaring a more restrictive parameter type, while a postcondition is incorporated by declaring a more restrictive return type. A perfect example of this technique can be found in the definition of the variable renaming operation presented in §3.3.3.

With complicated specifications in PVS, it is possible to get overwhelmed by TCCs. Accordingly, a mechanism is provided to alleviate the buildup of TCCs via judgements that make available more specific information to the typechecker. Judgements come in two varieties. Constant judgements state that a particular constant has a more specific type than its declared type, while subtype judgements state that one type is a subtype of another. We point out in the proof where we make use of the judgement mechanism.

A background collection of theories is provided in the PVS prelude. Included are theories for numbers, set operations, finite sets, ordinals, functions, induction schemes and abstract data types, including a list definition. The prelude also contains a number of judgements.

The PVS prover accepts commands via a Lisp-like interface (embedded in Emacs). These commands consist of high-level commands called strategies and a number of more specific commands known as rules. Strategies are designed to tackle a broad range of problems and ideally finish proofs automatically. Rules, on the other hand, give the user much more control over the proof, although the actions taken are generally more atomic. For example, the *split* rule splits the current proof into a number of subproofs, while the *prop* strategy splits the proof and then applies propositional flattening and simplification. It is generally a good idea to attempt proofs using the higher level strategies first, resorting to lower level commands only when necessary. In addition to increasing the level of automation, this approach produces proofs that are more resiliant to changes in the specifications.

PVS provides a mechanism for defining abstract data types (ADTs) inductively using a list of constructors. From these constructors, a complete set of axioms is automatically generated which contains: *extensionality axioms* (two objects constructed from the same components using the same constructor are identical); *eta axioms* (an object constructed from the same components of X is identical to X); *accessor axioms* (a component of a constructed object is the appropriate argument to the constructor); *induction schemes* (for induction on the structure of the ADT); and *recursive combinators* (for defining rank functions either for the natural numbers, or the set of ordinals). In addition to ADT axioms, PVS automatically generates induction schemes for all inductive definitions.

A PVS specification is split up into theories and data type definitions. Each theory consists of a (possibly empty) list of parameters, importing and exporting statements, type definitions, constant definitions, function definitions, judgements, and lemmas. The parameters can be types, subtypes, or constants. Exporting statements are used to specify the names that are made visible to theories that are importing the current theory. Importing statements specify a list of theories to be imported and can be either parametrized or unparameterised.

3. Syntax

3.1. Background

In this section we present the general framework as described in [26], and more recently in [19]. Concurrently we will describe how these notions are interpreted into PVS. We will illustrate the concepts and definitions that are introduced in this paper by means of a, hopefully familiar, running example. This running example is a simplified version of Scheme, which we will call *simple scheme* and denote by λ_{sch}. λ_{sch} will have as atoms: integers, $\mathbb{Z}$, and booleans, $\mathbb{B}$. The remaining more complex data consists of mutable binary pairs (i.e. *cons cells*), and of course *closures*. It also has a branching construct based on the built in booleans, which is used to generate the usual *Scheme*-like `if` construct. Note that λ_{sch} does not have any *control* operations such as `call/cc`, nor does it have variable assignment as provided by `set!`(x, v), nor does it even have the usual *empty list* `()`.

Example 3.1 (The running example of λ_{sch}): The primitive data operations of λ_{sch} consist of the following sets of operations. The boolean valued predicates, consisting of `integer?`, `cons?`, `boolean?`, and `procedure?`. The arithmetical operations, consisting of the usual binary operations on the integers, $\mathbb{Z}$, namely `+`, `-`, and `*`, as well as a, boolean valued, equality binary predicate, `=`, on integers. The traditional Lisp primitives, consisting of `cons`, `car`, and `cdr`. The binary memory operation `cons` allocates new mutable pairs. Evaluation of `cons`(v_a, v_d) results in the allocation of a new binary pair, a *cons cell*, whose first and second components are initialized to v_a, and v_d respectively. For historical reasons the first component is called the *car*, whilst the second is called the *cdr*. The unary operations `car` and `cdr` retrieve the *car* and *cdr* components of an existing *cons cell*. These operations are not defined if they are applied to anything other than a cons cell. The destructive memory operations, consisting of `setcar!` and `setcdr!`. The binary, memory modification primitives `setcar!` and `setcdr!` expect a cons cell as their first argument; they update memory, so that the new car (in the case of `setcar!`) or cdr (in the case of `setcdr!`) of that cons cell is the second argument. The value returned by these operations is unimportant, though we have chosen the integer 0. Finally we include the strict ternary branching operation, `br`. `br` branches depending on whether its first argument is true or not. We adopt the usual Lisp/Scheme convention that any value other than the boolean `false` is true.

An example of a λ-language with uniform semantics that has control operations can be found in [26].

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.

Example 3.2 (States of λ_{sch} informally): To illustrate the idea behind states, consider the following context

$$\begin{array}{l}\mathtt{let}\{y_2 := \mathtt{cons}(0,0)\}\,\mathtt{let}\,\{y_1 := \mathtt{cons}(0,0)\} \\ \quad \mathtt{seq}(\mathtt{setcar!}\,(y_1, v_{1a}), \mathtt{setcdr!}\,(y_1, v_{1d}), \\ \qquad \mathtt{setcar!}\,(y_2, v_{2a}), \mathtt{setcdr!}\,(y_2, v_{2d}), \\ \qquad [\,]\,)\end{array}$$

of our, as yet informal, running example. It describes the state of memory where there are 2 cells, y_1 and y_2, with each particular cell y_i having as its `car` the value v_{ia} and as its `cdr` the value v_{id} for $1 \leq i \leq 2$. Because such contexts are just a part of a slightly generalized syntax, it allows us the benefit of providing a semantics for λ_{sch} without having to introduce additional semantic baggage.

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, possibly conditional, 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 said to be *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 main result then states that the CIU context lemma holds for any λ-language with uniform semantics.

The uniformity requirements are that each reduction rule hold not only for traditional expressions, but also for expressions containing parameters. Parameters are somewhat akin to meta-variables, in that free variables of the instantiating expression may be captured when the parameters are instantiated by those expressions. This parametric notion of computation is best presented using the idea of a context and the treatment here follows the general theory presented in [15], and used in [17, 18].

3.1.1. *Contexts as Expressions*

Central to our whole approach is the notion of a *context*, an expression with, possibly many distinct, holes in it. Hole are places where another expression can be inserted, and whose free variables may become bound by the enclosing expression or *context*. We use contexts to: represent the *current state* in a computation; represent the *continuation* in a computation; define *operational equivalence*; define the *uniformity conditions* necessary for CIU to hold; prove the usual *CIU context lemma* (by computing with them); define the *atomic satisfaction* clauses of the logic; and finally as a *formula constructor* in the logic. Thus we build them into the language right from the start. Contexts can be formalized by adding parameters to the syntax of expressions. The novelty of our approach is that we decorate parameters with information recording pending substitutions which are to be applied when the parameter is instantiated. This has several desirable consequences. One is that alpha conversion is valid for contexts, unlike the traditional lambda-calculus contexts. The other is that symbolic execution commutes with parameter instantiation, which simplifies reasoning about program equivalence.

An example of a traditional context with two occurrences of the same hole:

$$\mathtt{app}(\lambda x.\lambda y.\mathtt{app}(y, \mathtt{app}(x, \texttt{[ ]})), \lambda y.\texttt{[ ]})$$

where app is a binary operation representing λ-application described in §1.4, and `[ ]` is the hole. We will abbreviate this expression by C:

$$C \stackrel{\triangle}{=} \mathtt{app}(\lambda x.\lambda y.\mathtt{app}(y, \mathtt{app}(x, \texttt{[ ]})), \lambda y.\texttt{[ ]})$$

In the traditional approach filling is textual replacement. Traditionally filling is represented by the notation $C\,\texttt{[}\,e\,\texttt{]}$ where C is the context and e is the expression being inserted into the hole:

$$C\,\texttt{[}\mathtt{app}(x, y)\texttt{]} \stackrel{\triangle}{=} \mathtt{app}(\lambda x.\lambda y.\mathtt{app}(y, \mathtt{app}(x, \mathtt{app}(x, y))), \lambda y.\mathtt{app}(x, y))$$

This has the consequence that one cannot α-convert contexts, and hence cannot compute with them.

In our approach holes are parameters annotated by substitutions, thus the traditional context, C, above is represented as:

$$C \ \stackrel{\triangle}{=}\ \mathtt{app}(\lambda x.\lambda y.\mathtt{app}(y, \mathtt{app}(x, X^{\{x\mapsto x, y\mapsto y\}})), \lambda y.X^{\{y\mapsto y\}})$$

In this approach, filling is normal capture avoiding substitution, followed by the annotating substitution.

$$C\,[\,\mathtt{app}(x,y)\,] \ \stackrel{\triangle}{=}\ C^{\{X\mapsto \mathtt{app}(x,y)\}}$$

In this approach it is simple to define α-congruence, for example:

$$C \stackrel{\alpha}{\equiv} \mathtt{app}(\lambda\nu_x.\lambda\nu_y.\mathtt{app}(\nu_y, \mathtt{app}(\nu_x, X^{\{x\mapsto\nu_x, y\mapsto\nu_y\}})), \lambda\nu_y.X^{\{y\mapsto\nu_y\}})$$

for new and distinct ν_x and ν_y. Thus

$$\begin{aligned}
&C\,[\,\mathtt{app}(x,y)\,]\\
&\quad\stackrel{\alpha}{\equiv} \mathtt{app}(\lambda\nu_x.\lambda\nu_y.\mathtt{app}(\nu_y, \mathtt{app}(\nu_x, \mathtt{app}(x,y)^{\{x\mapsto\nu_x, y\mapsto\nu_y\}})),\\
&\qquad\qquad \lambda\nu_y.\mathtt{app}(x,y)^{\{y\mapsto\nu_y\}})\\
&\quad\stackrel{\alpha}{\equiv} \mathtt{app}(\lambda\nu_x.\lambda\nu_y.\mathtt{app}(\nu_y, \mathtt{app}(\nu_x, \mathtt{app}(\nu_x,\nu_y))), \lambda\nu_y.\mathtt{app}(x,\nu_y)).\\
&\quad\stackrel{\alpha}{\equiv} \mathtt{app}(\lambda x.\lambda y.\mathtt{app}(y, \mathtt{app}(x, \mathtt{app}(x,y))), \lambda y.\mathtt{app}(x,y)).
\end{aligned}$$

It then also becomes a simple matter to compute with these more general expressions:

$$\begin{aligned}
&\mathtt{app}(\lambda x.\lambda y.\mathtt{app}(y, \mathtt{app}(x, X^{\{x\mapsto x, y\mapsto y\}})), \lambda y.X^{\{y\mapsto y\}})\\
&\quad\stackrel{\alpha}{\equiv}\quad \mathtt{app}(\lambda x.\lambda\nu.\mathtt{app}(\nu, \mathtt{app}(x, X^{\{x\mapsto x, y\mapsto\nu\}})), \lambda y.X^{\{y\mapsto y\}})\\
&\quad\stackrel{\beta}{\longrightarrow}\quad \lambda\nu.\mathtt{app}(\nu, \mathtt{app}(\lambda y.X^{\{y\mapsto y\}}, X^{\{x\mapsto\lambda y.X^{\{y\mapsto y\}}, y\mapsto\nu\}})).
\end{aligned}$$

3.2. 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.

3.2.1. *The* Theoretical *Treatment of Syntax*

Fix two disjoint countably infinite sets, $\mathbf{X}$, of variables, and $\mathbf{P}$, of parameters. 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, $\mathbf{V}$, a subset of expressions 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 [19].

Definition 3.3 (E, L, S, F): `FLL-version-03` :1–156

The set of expressions, $\mathbf{E}$, 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:

$$\mathbf{E} = \mathbf{X} \cup \mathbf{P}^{\mathbf{S}} \cup \mathbf{A} \cup \mathbf{L} \cup \mathbf{O}_n(\mathbf{E}^n) \quad \mathbf{L} = \lambda \mathbf{X}.\mathbf{E}$$

$$\mathbf{S} = \mathrm{Fmap}[\mathbf{X}, \mathbf{V}] \quad \mathbf{F} = \mathrm{Fmap}[\mathbf{P}, \mathbf{E}]$$

To reiterate on our naming conventions described in section 1.4 we point out that the above definitions take up the first one hundred and fifty six lines of the file `FLL-version-03.pvs` downloadable from [6]. 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 the syntactic counterpart to the usual semantic notion of *environments*. They 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. Hence our example context presented above in §3.1.1

$$\mathtt{app}(\lambda x.\lambda y.\mathtt{app}(y, \mathtt{app}(x, X^{\{x \mapsto x, y \mapsto y\}})), \lambda y.X^{\{y \mapsto y\}})$$

is a typical example of 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.

Example 3.4 (Syntax of $\lambda_{\rm sch}$): We can now provide our running example with a firmer syntactic foundation.

$$
\begin{array}{lcl}
\mathbf{A} & \triangleq & \mathbb{Z} \cup \mathbb{B} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \cup \{\texttt{true}, \texttt{false}\} \\
\mathbf{O}_1 & \triangleq & \{\texttt{integer?}, \texttt{cons?}, \texttt{boolean?}, \texttt{procedure?}, \texttt{car}, \texttt{cdr}\} \\
\mathbf{O}_2 & \triangleq & \{\texttt{app}, \texttt{cons}, \texttt{setcar!}, \texttt{setcdr!}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{=}\} \\
\mathbf{O}_3 & \triangleq & \{\texttt{br}\}
\end{array}
$$

Example 3.5 (Extended Syntax of $\lambda_{\rm sch}$): In order to make programs in $\lambda_{\rm sch}$ easier to read, we introduce some abbreviations.

$$
\begin{array}{lcll}
f(x) & \triangleq & \texttt{app}(f, x) & \\
\texttt{let}\,\{x := e_0\}e_1 & \triangleq & \texttt{app}(\lambda x.e_1, e_0) & \\
\texttt{seq}(e) & \triangleq & e & \\
\texttt{seq}(e_0, \ldots, e_n) & \triangleq & \texttt{let}\,\{z := e_0\}\texttt{seq}(e_1, \ldots, e_n) & \bigwedge_{i \leq n} z \notin \mathrm{FV}(e_i) \\
\texttt{if}(e_0, e_1, e_2) & \triangleq & \texttt{app}(\texttt{br}(e_0, \lambda z.e_1, \lambda z.e_2), 0) & \bigwedge_{i \leq 2} z \notin \mathrm{FV}(e_i)
\end{array}
$$

Example 3.6 (Programs of $\lambda_{\rm sch}$): Since the $\lambda_{\rm sch}$-calculus extends the call-by-value λ-calculus, the usual call-by-value fixed point combinator is a term in the language. A somewhat different fixed point combinator, which makes use of the mutable cons cells is possible:

$$
\begin{array}{lcl}
\texttt{Y} & \triangleq & \lambda y.\,\texttt{let}\{z := \texttt{cons}(0, 0)\} \\
 & & \qquad\quad \texttt{seq}(\texttt{setcar!}(z, \lambda x.\texttt{app}(\texttt{app}(y, \texttt{car}(z)), x)), \\
 & & \qquad\qquad\quad \texttt{car}(z)) \\
F & \triangleq & \lambda f.\lambda x.\texttt{if}(x = 0, 1, f(x - 1) * x) \\
\texttt{Fact} & \triangleq & \texttt{Y}(F)
\end{array}
$$

`Y` is a version of the fixed-point combinator, essentially identical to the one suggested by Landin [13]. Here we have used it to define the usual factorial operation, `Fact`.

3.2.2. *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. Finite maps in PVS are best represented as dependent record types

$$
\mathrm{Fmap}[A, B] = \prod_{\delta \in \mathbf{P}_\omega(A)} (\delta \longrightarrow B)
$$

The actual PVS definition uses a parametric theory:

```
fmap[domain, range: TYPE]: THEORY
 BEGIN
  fmap: TYPE =  [# dom: finite_set[domain],
                   sub: [(dom) -> range]  #]
 END fmap
```

which defines the parametric theory `fmap[domain, range]` whose sole purpose is the definition of the dependent type `fmap` of finite functions whose domain is a finite subset of `domain` and whose range is a subset of `range`. The two type parameters `domain` and `range` are arbitrary types, while the defined type `fmap` is defined using the dependent product construct of PVS:

`[# ... #].`

Also the entire development is parametric in a set of atoms and operations, that includes the binary `app`, together with the arities of those operations. Since the operations are a non-empty collection, we use PVS's `TYPE+` mechanism to represent them as a non-empty type. This theory is appropriately named `Landin`, and can be found in `FLL−version−03` :8:

```
Landin: THEORY
 BEGIN
  At: TYPE
  Op: TYPE+
  arity: [Op -> nat]
  app: Op
  appAx: AXIOM arity(app) = 2
 END Landin
```

Implementation of the syntax within PVS is a straightforward use of the *datatype with subtypes* facility, that we will describe in more detail momentarily. 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 is naturally a subtype of the set of expressions. However they appear negatively in the domain (to the finite set constructor) of annotating substitutions, 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 obvious identification of a variable and the index (in this case a natural number) from which it was constructed.

The second technical issue is how to deal with the argument lists to the operations in the language. On the advice of Shankar [25] 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 to, 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. Subsequent research has uncovered an even better encoding [10]. In this new approach, not adopted here, an application constructor `app` takes two arguments: a function symbol field `func` and an arguments field `args`. The arguments field is formalized as a *dependent* type `[below(arity(func)) -> syntax]` which represents an *array* of `syntax` in the arity of the function symbol of the term. The type `below(num)` for a natural number `num` is the (possibly empty) subrange $0, \ldots, \mathtt{num} - 1$. The array-based formalization has some important advantages: terms are well-formed by construction thus avoiding the need for cumbersome proof obligations. Also operations on terms can be defined by a simple structural recursion without the use of mutual recursion on terms and lists of terms.

Finally we made a design decision to fully develop the syntax prior to the introduction of the notion of values. There are both obvious and not so obvious reasons for this. The obvious one 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 one is that if we restrict the range of annotating substitutions to values at this stage, then it 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 the appropriate syntactic class by recursion, as a subtype of expressions with the desired feature.

```
Syntax: DATATYPE WITH SUBTYPES E, P, D
 BEGIN
  IMPORTING Landin, fmap
  A(a: At): A?: E
  X(i: nat): X?: E
  L(x: (X?), b: E): L?: E
  O(op: Op, args: D): O?: E
  D(arglist: list[E]): D?: D
  H(p: P, s: fmap[nat, E]): H?: E
  P(i: nat): P?: P
 END Syntax
```

In this definition we declare a datatype `Syntax`, with subtypes `E`, `P`, and `D` representing the expressions **E**, parameters **P**, and the argument lists of expressions that have no theoretical counterpart, as mentioned above. The constructors of the datatype are the operators `A`, `X`, `L`, `O`, `D`, `H`, and `P`. In the case of `A`, `X`, `D`, and `P` they can be thought of as injections of the sets `At`, `nat`, `list[E]`, and `nat` into the respective subtypes. We think of `A(a)` as representing the atom a, `X(i)` as representing the variable x_i, and `P(i)` as representing the parameter X_i. The recognizers of the objects created by these constructors are the operators `A?`, `X?`, `L?`, `O?`, `D?`, `H?`, and `P?`. The actual subtypes to which the constructed objects belong is specified by the last entry in each line/declaration (each line is called a constructor specification in PVS). Thus `A`, `X`, `L`, `O`, and `H` construct expression objects (i.e, objects belonging to the subtype `E`), `D` constructs argument list objects (i.e objects belonging to the subtype `D`), while `P` constructs parameters (i.e, objects belonging to the subtype `P`). The arguments to each constructor have been given explicit names, these are the accessors. Thus in the case of λ-expressions, if `l` is an object constructed by the operation `L`, then the λ bound variable is `x(l)`, while the body of the λ-expression is `b(l)`. As we have briefly mentioned, from this declaration PVS generates the following:

- An extentionality axiom for each constructor. E.g.

  $$(\forall \mathtt{l}_0, \mathtt{l}_1 \in \mathtt{L?})(\mathtt{x}(\mathtt{l}_0) = \mathtt{x}(\mathtt{l}_1) \land \mathtt{b}(\mathtt{l}_0) = \mathtt{b}(\mathtt{l}_1)) \Rightarrow \mathtt{l}_0 = \mathtt{l}_1.$$

- An eta axiom for each constructor (that has accessors). E.g

  $$(\forall \mathtt{l} \in \mathtt{L?})(\mathtt{l} = \mathtt{L}(\mathtt{x}(\mathtt{l}), \mathtt{b}(\mathtt{l}))).$$

- For each accessor, an axiom that says that the accessor composed with the corresponding constructor returns the correct value. E.g. an axiom stating that $\mathtt{x}(\mathtt{L}(\mathtt{y}, \mathtt{e})) = \mathtt{y}$ and $\mathtt{b}(\mathtt{L}(\mathtt{y}, \mathtt{e})) = \mathtt{e}$.

- An inclusive axiom stating that every element of the datatype belongs to at least one recognizer subtype.

- Two induction schemes for proving properties of the datatype.

- Subterm, and proper subterm relations, together with an axiom stating the well-foundedness of the proper subterm relation.

- Recursive combinators, such as `reduce_nat`, for defining derived notions like size, depth or rank of an element of the datatype.

- Several other things that need not concern us in this paper.

3.3. Auxilliary Syntactic Notions

One of the real differences between the 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 people 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 variables (α equivalence). We must also show that these operations are functional modulo this equivalence relation, as well as establishing that α equivalence is a congruence. Though, on the positive side, we should point out that these theories, once developed, may be reused. Indeed their structure already builds upon our earlier work establishing the Church-Rosser theorem for a named variable presentation of the call-by-value λ-calculus [8].

3.3.1. *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 formal notions of rank do not coincide. In the formal treatment an application consists of an operation paired with a list of expressions. The list of expressions belongs to a bonafide syntactic subcategory, `D`. This has the consequence that the members of the list of expressions have rank two less than the application, not one as they would have in the theoretical treatment. Thus the actual rank we implement is:

Definition 3.7 ($\mathrm{rank}(e)$)**:** `Rank` :1–87

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

The PVS code that defines this uses `reduce_nat`

```
rank(e): nat =
```

```
reduce_nat((LAMBDA a: 0),
           (LAMBDA i: 0),
           (LAMBDA i1, i2: i2 + 1),
           (LAMBDA op, i: i + 1),
           (LAMBDA ln:
             list_max(ln) + length(ln) + 1),
           (LAMBDA i1, s:
             max(add(0,
                     {i: nat |
                       (EXISTS (d: (dom(s))):
                         sub(s)(d) = i)}))
                  + 1),
           (LAMBDA i: 0)
           )(e)
```

3.3.2. *Freeness and Traps*

The first notions 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 3.8 ($\mathrm{FV}(e)$**,** $\mathrm{FP}(e)$)**:** `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` :38, while the parameters, $\mathrm{FP}(e)$, are defined in `Vars` :72. We extend these to substitutions in the obvious fashion:

$$\mathrm{FV}(\Delta) = \bigcup_{\delta \in \mathrm{Dom}(\Delta)} \mathrm{FV}(\Delta(\delta)) \qquad \mathrm{FP}(\Delta) = \bigcup_{\delta \in \mathrm{Dom}(\Delta)} \mathrm{FP}(\Delta(\delta))$$

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 3.9 ($\mathrm{Traps}(e)$)**:** `Traps` :1-36
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.

Example 3.10 ($\mathrm{FV}(e)$**,** $\mathrm{FP}(e)$**, and** $\mathrm{Traps}(e)$ **in** λ_{sch})**:** In what follows let

$$\begin{aligned}\zeta \stackrel{\triangle}{=} & \ \texttt{let}\{y_2 := \texttt{cons}(0,0)\}\ \texttt{let}\,\{y_1 := \texttt{cons}(0,0)\} \\ & \quad \texttt{seq}(\texttt{setcar!}\,(y_1, z_1), \texttt{setcdr!}\,(y_1, \texttt{true}), \\ & \qquad \texttt{setcar!}\,(y_2, \lambda w.y_1), \texttt{setcdr!}\,(y_2, \lambda x.\texttt{app}(\texttt{car}(y_2), x)), \\ & \qquad X^{\{z_1 \mapsto y_1, z_2 \mapsto y_2\}})\end{aligned}$$

where $z_1 \neq z_2$ and $y_i \neq z_j$ for all $1 \leq i \leq j \leq 2$. Then

$$\mathrm{FV}(\zeta) = \{z_1\}, \quad \mathrm{FP}(\zeta) = \{X\}, \quad \mathrm{Traps}(\zeta) = \{z_1, z_2\}.$$

3.3.3. *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 [8], 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:

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

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.[1] 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 3.11 (Renaming $e^{\{x \mapsto y\}}$**):** `Subst` :26

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

$$(X^\sigma)^{\{x \mapsto y\}} = X^{\sigma^{\{x \mapsto y\}}}$$

$$\sigma^{\{x \mapsto y\}} = \lambda \nu \in \mathrm{Dom}(\sigma).\sigma(\nu)^{\{x \mapsto y\}}$$

[1] Note that we only need state this for the subtypes `E`, `D`, and `P`. For the other syntactic categories it is a consequence of the definition.

Note that we always rename the λ bound variables, regardless of whether or not a clash might have otherwise occurred.

Example 3.12 (Renaming in λ_{sch}): Let ζ be as in example 3.10. Then

$$\begin{aligned}\zeta^{\{z_1 \mapsto y_2\}} = {} & \mathtt{let}\{\nu_2 := \mathtt{cons}(0,0)\}\,\mathtt{let}\,\{\nu_1 := \mathtt{cons}(0,0)\} \\ & \quad \mathtt{seq}(\mathtt{setcar!}\,(\nu_1, y_2), \mathtt{setcdr!}\,(\nu_1, \mathtt{true}), \\ & \qquad \mathtt{setcar!}\,(\nu_2, \lambda w.\nu_1), \mathtt{setcdr!}\,(\nu_2, \lambda x.\mathtt{app}(\mathtt{car}(\nu_2), x)), \\ & \qquad X^{\{z_1 \mapsto \nu_1, z_2 \mapsto \nu_2\}})\end{aligned}$$

where ν_1 and ν_2 are distinct fresh variables.

```
Rename(e, oldvar, newvar): RECURSIVE
 {e0: Syntax |
   rank(e0) = rank(e) AND
   (E(e0) IFF E(e))   AND
   (D(e0) IFF D(e))   AND
   (P(e0) IFF P(e))   AND
   ((D(e) AND D(e0))
      IMPLIES
        length(arglist(e)) = length(arglist(e0)))}
= CASES  e
      OF A(a_var): e,
         X(i_var): IF i(oldvar) = i_var
                   THEN newvar
                   ELSE e ENDIF,
         L(x_var, b_var):
           LET x: (X?) =
                 X(New(add(i(newvar),
                           add(i(oldvar), Vars(e)))))
             IN L(x, Rename(Rename(b_var, x_var, x),
                            oldvar, newvar)),
         O(op_var, args_var):
           O(op_var, Rename(args_var, oldvar, newvar)),
         D(arglist_var):
           CASES arglist_var OF
             null: D(null),
             cons(a, d):
                D(cons(Rename(a, oldvar, newvar),
                       arglist(Rename(D(d),
                                      oldvar,
                                      newvar))))
           ENDCASES,
         H(p_var, s_var):
```

```
          H(p_var,
             (# dom := dom(s_var),
                sub
                  := (LAMBDA (i: (dom(s_var))):
                        Rename(sub(s_var)(i),
                               oldvar, newvar)) #)),
        P(i_var): e
     ENDCASES
    MEASURE (LAMBDA (e, oldvar, newvar): rank(e));
```

To produce such *fresh* variables we posit a function new that pulls them, or at least their indices, out of a hat:

Definition 3.13 (new): FLL−version−03 :109

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

```
New: THEORY
 BEGIN
  z: VAR finite_set[nat]
  New: [finite_set[nat] -> nat]
  NewAx: AXIOM NOT member(New(z), z)
 END New
```

3.3.4. *Substitution*

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

Definition 3.14 (Substitution e^σ): Subst :118–144

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:

Lemma 3.15 (Renaming): Subst : Rename_Subst :158

$e^{\{x \mapsto y\}} = e^{\{x \mapsto y\}}$ where the left hand side is renaming, while the right hand side is substitution.

Although overloading is allowable in PVS, the actual formulation and statement of this result in PVS does not use it:

```
Rename_Subst: LEMMA
 Rename(e, X(i), X(j))
 = Subst(e, MakeSubst(X(i), X(j)))
```

3.3.5. *Filling*

Definition 3.16 (Filling e^Σ): `Fill` :17–46

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. 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 filling – the substitution of parameters – 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.

Example 3.17 (Filling in λ_{sch}): Let ζ be as in example 3.10. Then

$$\zeta^{\{X\mapsto \mathtt{setcar!}(z_1,\mathtt{cons}(y_1,z_2))\}} =$$

$$\begin{aligned}&\mathtt{let}\{\nu_2 := \mathtt{cons}(0,0)\}\,\mathtt{let}\,\{\nu_1 := \mathtt{cons}(0,0)\}\\ &\quad \mathtt{seq}(\mathtt{setcar!}\,(\nu_1, z_1), \mathtt{setcdr!}\,(\nu_1, \mathtt{true}),\\ &\qquad \mathtt{setcar!}\,(\nu_2, \lambda w.\nu_1), \mathtt{setcdr!}\,(\nu_2, \lambda x.\mathtt{app}(\mathtt{car}(\nu_2), x)),\\ &\qquad \mathtt{setcar!}\,(\nu_1, \mathtt{cons}(y_1, \nu_2)))\end{aligned}$$

where ν_1 and ν_2 are distinct fresh variables.

3.3.6. α *Equivalence*

The notion of being equivalent modulo the renaming of bound variables easily extends to this more general setting [15] 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 $\overset{\alpha}{\equiv}$ [15]:

Definition 3.18 ($\overset{\alpha}{\equiv}$): `WeakAlpha` :13–49

$$(\text{Base}) \quad \frac{}{v \overset{\alpha}{\equiv} v} \qquad \text{provided } v \in \mathbf{A} \cup \mathbf{X}$$

$$(\text{Op}) \quad \frac{e_0 \overset{\alpha}{\equiv} e_0' \quad \ldots \quad e_n \overset{\alpha}{\equiv} e_n'}{\vartheta(e_0, \ldots, e_n) \overset{\alpha}{\equiv} \vartheta(e_0', \ldots, e_n')}$$

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

for $\Delta_0, \Delta_1 \in \mathbf{S}$ or $\Delta_0, \Delta_1 \in \mathbf{F}$

$$\text{(Param)}\quad \frac{\sigma_0 \overset{\alpha}{\equiv} \sigma_1}{X^{\sigma_0} \overset{\alpha}{\equiv} X^{\sigma_1}}$$

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

There is substantial leeway in the precise way one formalizes $\overset{\alpha}{\equiv}$. These issues are discussed at length in [21] so we shall not dwell on them here. The main point is that we attempted two formalizations, in keeping with the well-tested principle 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

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

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 [8]. 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 CIU theorem.

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

Lemma 3.19 ($\overset{\alpha}{\equiv}$): `WeakAlpha` :76–104

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 transitive. `WeakAlpha` : `WeakTransitive` :97
5. $e_0 \overset{\alpha}{\equiv} e_1 \;\Rightarrow\; \mathrm{FV}(e_0) = \mathrm{FV}(e_1)$. `WeakAlpha` : `Alpha_FV` :104

3.3.7. *Substitution preserves α Equivalence*

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

Lemma 3.20 (AlphaSubst): `AlphaSubst` : `AlphaSubst_Theorem` :132

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

As pointed out by Mason in [15], the proof of this is somewhat delicate; actually it is even more delicate than suggested there. This is because the proof presented there does not stand up to the test of formalizing in PVS. In [15] the suggested proof requires establishing, directly, the following three properties by simultaneous induction.

Lemma 3.21 (Alpha): `AlphaSubst` : `AlphaSubst_Swap` :128

1. $(e_0 \stackrel{\alpha}{\equiv} e_1 \;\wedge\; e_0' \stackrel{\alpha}{\equiv} e_1') \;\Rightarrow\; {e_0}^{\{x \mapsto e_0'\}} \stackrel{\alpha}{\equiv} {e_1}^{\{x \mapsto e_1'\}}$
2. $(\nu_0 \notin \mathrm{FV}(e) \wedge \nu_0 \neq x \neq \nu_1) \;\Rightarrow\; ({e_0}^{\{\nu_0 \mapsto \nu_1\}})^{\{x \mapsto e\}} \stackrel{\alpha}{\equiv} ({e_0}^{\{x \mapsto e\}})^{\{\nu_0 \mapsto \nu_1\}}$
3. $({e_0}^{\{x \mapsto \nu\}})^{\{\nu \mapsto y\}} \stackrel{\alpha}{\equiv} {e_0}^{\{x \mapsto y\}}$ 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 3.21 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 3.22 (SubstProps):

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

3.3.8. *Filling preserves α Equivalence*

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

Lemma 3.23 (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 3.24 (FillSwap): `AlphaFill` : `AlphaFill_Swap` :47
$x \notin \mathrm{FV}(\Sigma) \;\Rightarrow\; (e^{\Sigma})^{\{x \mapsto y\}} \stackrel{\alpha}{\equiv} (e^{\{x \mapsto y\}})^{\Sigma}$

Definition 3.25 (Term, Closed): `Closed` :1-40
We adopt the following naming convention: an expression with no parameters is called a *term*; a *term* with no free variables is *closed*. Note that according to this naming convention, 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: For example being a term is preserved by α equivalence, as is being closed. They are also both preserved under filling (since they remain unchanged).

3.3.9. *Values*

Values are expressions that no longer need to be evaluated. They are thus a special type of expression. In the call-by-value world that this paper deals with, a lambda expression applied to another expression can only be reduced (via β-reduction) after the argument expression has been reduced to a value. We are now in a position to define the class of values.

Definition 3.26 (Value Expressions (V)): `Values` :1-189
The set of value expressions, $\mathbf{V}$, contains all variables, atoms, and λs. It may in addition contain expressions of the form $\vartheta(v^n)$:

(range) $\quad \mathbf{X} \cup \mathbf{A} \cup \mathbf{L} \subseteq \mathbf{V} \subseteq \mathbf{X} \cup \mathbf{A} \cup \mathbf{L} \cup \mathbf{O}_n(\mathbf{V}^n)$

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) $\quad \mathbf{V}$ is closed under $\stackrel{\alpha}{\equiv}$ `Values` :82

(vsub) $\quad v \in \mathbf{V} \;\Rightarrow\; v^{\sigma} \in \mathbf{V}$ `Values` :88

(inst) $\quad v \in \mathbf{V} \;\Rightarrow\; v^{\Sigma} \in \mathbf{V}$ `Values` :91

There is one closure condition remaining. It describes precisely under what conditions filling an expression can produce a value. There are exactly three cases where filling an expression can result in a value. The simplest is when the expression is, itself, a value. The next simplest is when the expression is a parameter, and it is being replaced by a value. Note that the annotating substitution does not cause problems because of (**vsub**). The third case is somewhat more complex to state and requires two auxiliary definitions. We need to define the notion of the *top-level* rank of a parameter in an expression.

$$\mathrm{rk_t}(e, X) = \begin{cases} 0 & \text{if } e \in \mathbf{X} \cup \mathbf{A} \cup \mathbf{L} \text{ or } e = Y^\sigma \text{ and } Y \neq X, \\ 1 & \text{if } e = X^\sigma, \\ \sum_{i=0}^{n} \mathrm{rk_t}(e_i, X) & \text{if } e = \vartheta(e_1, \ldots, e_n) \end{cases}$$

The *top-level* parameters are then simply those parameters in $\mathrm{FP}(e)$ that have non-zero rank.

$$\mathrm{TLP}(e) = \{X \in \mathrm{FP}(e) \mid \mathrm{rk_t}(e, X) > 0\}$$

In other words: an occurrence of X in e is said to be *top-level* if it is not in the scope of a λ-expression, or in the range of a substitution annotating a parameter.

Finally to state trichotomy we need to define the third alternative, the set of pseudo-constructed values $\mathbf{Val}^{\mathrm{s}}$.

$$\mathbf{Val}^{\mathrm{s}} = \mathbf{O}_n((\mathbf{Val} \cup \mathbf{P}^{\mathbf{S}} \cup \mathbf{Val}^{\mathrm{s}})^n)$$

The trichotomy axiom can now be expressed as, `Values` :94:

$$\text{(trich)} \quad e^\Sigma \in \mathbf{V} \;\Rightarrow\; \begin{array}{l} (e \in \mathbf{V}) \vee \\ (e = X^\sigma \,\wedge\, \Sigma(X) \in \mathbf{V}) \vee \\ (e \in \mathbf{Val}^{\mathrm{s}} \,\wedge\, (\forall X \in \mathrm{TLP}(e))(\Sigma(X) \in \mathbf{V})) \end{array}$$

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

Example 3.27 (V of λ_{sch}): In λ_{sch} there are no constructors, thus

$$\mathbf{V} \stackrel{\triangle}{=} \mathbf{X} \cup \mathbf{A} \cup \mathbf{L}$$

In the original presentation [26] the only explicit closure condition on values, other than (**range**), was that it be closed under value substitutions, i.e. (**vsub**). The (**triv**) principle is required in several places and is clearly just common-sense. The requirement that it be closed under filling, (**inst**), was revealed as a TCC in the specification of the set of values. The somewhat more subtle principle (**trich**) was discovered in the actual formal proof of the CIU theorem. It replaces an early more restrictive dichotomy principle also discovered in the actual proof of the CIU theorem [7, 19]. It is not that mysterious though: If, after instantiating the parameters, the resulting expression is a value, then one of three possibilities must be true. The first possibility is that the original expression was a value, and hence all occurrences of parameters must lie in the scope of an enclosing λ-expression, and thus be irrelevant from the point of view of the result of instatiantion being a value. The second possibility is that the original expression was a parameter, which in this case, for the result to be a value, must have been instantiated by a value. The final case is that the original expression is a constructed value modulo a few top-level occurrences of parameters, which consequently must have been instantiated by values.

We can now define `Values` : `NewE` :37, the subtype of expressions that we will restrict our attention to in the sequel. This consists of those expressions whose parameter annotations are solely value substitutions. A value substitution is a substitution whose ranges 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 is a good example of some of the subtle consequences of the above closure conditions on values.

Lemma 3.28 (inv): `Values` : `ValuesSubst_Lemma` :116
$e_0{}^{\sigma} \in \mathbf{V} \;\Rightarrow\; e_0 \in \mathbf{V}$

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 (**trich**) to conclude that $e_0 \in \mathbf{V}$. □

4. Semantics

4.1. Operational Semantics

In a λ-language the computation state is represented as a class of expressions. Each particular language will possess its 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 4.1 (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 $\stackrel{\alpha}{\equiv}$ `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) $\zeta \in \mathbf{Z} \Rightarrow \zeta^{\Sigma} \in \mathbf{Z}$ `Computation` :37

assuming $\circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{FP}(\Sigma))$

ζ ranges over $\mathbf{Z}$ and $\circ \in \mathbf{Z}$ is the empty state expression. We adopt the usual convention of writing $\zeta\,[\,e\,]$ instead of $\zeta^{\{\circ \mapsto e\}}$.

Example 4.2 (Z of λ_{sch}): In λ_{sch} the set of state contexts, $\mathbf{Z}$, is the set of contexts, ζ, of the form

$$
\begin{aligned}
&\mathtt{let}\{y_n := \mathtt{cons}(0,0)\} \ldots \mathtt{let}\,\{y_1 := \mathtt{cons}(0,0)\}\\
&\quad \mathtt{seq}(\mathtt{setcar!}\,(y_1, v_{1a}), \mathtt{setcdr!}\,(y_1, v_{1d}),\\
&\qquad \ldots,\\
&\qquad \mathtt{setcar!}\,(y_n, v_{na}), \mathtt{setcdr!}\,(y_n, v_{nd}),\\
&\qquad \circ^{\{z_1 \mapsto y_1, \ldots, z_n \mapsto y_n\}})
\end{aligned}
$$

where $z_i \neq z_j$ and $y_i \neq z_j$ for all $1 \leq i < j \leq n$. In the special case when $n = 0$ we take ζ to be simply $\circ$. Note that the $\{y_i \mid 1 \leq i \leq n\}$ are bound in ζ and may be α-converted when necessary. We have divided the state contexts into allocation, followed by assignment to allow for the construction of cycles. Thus, any state of memory is constructible by such an expression.

In the original presentation [26] there were no explicit closure conditions on states. The need for states to be closed under value substitution and filling was revealed as TCC's. The other conditions are necessary in the development of the uniformity conditions.

Definition 4.3 (Computation States (CS)): `Computation` :46

$\mathbf{CS} \stackrel{\triangle}{=} \mathbf{Z} : \mathbf{E}$ 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. $\zeta\,[\,e\,]$, which according to our convention mentioned in definition 4.1 is just $\zeta^{\{\circ \mapsto e\}}$. A state is *closed* just if its corresponding expression is closed, in other words if $\zeta\,[\,e\,]$ has no free parameters or variables. Application of value and parameter substitutions to states is defined in the obvious way: $(\zeta : e)^{\sigma} = \zeta^{\sigma} : e^{\sigma}$ and $(\zeta : e)^{\Sigma} = \zeta^{\Sigma} : e^{\Sigma}$. Note that by (**vsub**), and (**inst**) these are only meaningful if $\circ \notin \mathrm{Dom}(\sigma)$ and $\circ \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{FP}(\Sigma))$.

Definition 4.4 ($\longrightarrow\!\!\!\!\succ$, $\downarrow$, $\preceq$, $\updownarrow$, and $\sim$): Given a reduction relation for a λ-language: $\zeta : e \longrightarrow \zeta' : e'$, the following definitions are standard, where $\longrightarrow\!\!\!\!\succ$ is the *transitive closure* of $\longrightarrow$

Definedness: `Semantics` :38

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

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

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

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 [8]. 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 operator applied to the appropriate number of value expressions, provided that the operator is not a constructor. 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 4.5 (Redexes ($\mathbf{E}_{\rm r}$)): `Computation` :65
The set of redexes, $\mathbf{E}_{\rm r}$, is defined by:

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

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.

Example 4.6 ($\mathbf{E}_{\rm r}$ of $\lambda_{\rm sch}$): Since there are no constructors in $\lambda_{\rm sch}$ the set of redexes, $\mathbf{E}_{\rm r}$, is:

$$\mathbf{E}_{\rm r} \triangleq \mathbf{O}_n(\mathbf{V}^n)$$

So typical examples of redexes are:

$$\texttt{app}(\lambda x.x, \lambda x.x) \quad \texttt{br}(\lambda x.x, 7, 13) \quad 4+5 \quad \texttt{car}(x) \quad \texttt{setcdr!}(9, 11)$$

Note that whether or not a redex is actually reducible depends on the precise semantics given. In the semantics we provide for $\lambda_{\rm sch}$, the fifth redex above will not be reducible.

The set of redexes is closed under the following uniformity conditions.

Lemma 4.7 (Redex Uniformity): Redexes satisfy the following uniformity conditions:

(triv) $\mathbf{E}_{\mathrm{r}}$ is closed under $\stackrel{\alpha}{\equiv}$ `Computation` :70

(vsub) $r \in \mathbf{E}_{\mathrm{r}} \Rightarrow r^{\sigma} \in \mathbf{E}_{\mathrm{r}}$ `Computation` :92

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

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 [24] and were first introduced in [3].

Definition 4.8 (Reduction Contexts (R)): `Computation` :74
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 usual convention of writing $R[e]$ instead of $R^{\{\bullet \mapsto e\}}$.

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.

Example 4.9 (R of λ_{sch}): In λ_{sch} the set of reduction contexts, $\mathbf{R}$, is defined by

$$\mathbf{R} = \{\ \bullet\ \} \cup \mathbf{O}_{m+n+1}(\mathbf{V}^m, \mathbf{R}, \mathbf{E}^n)$$

Typical examples of reduction contexts that we will use in the rest of the paper are:

$\mathtt{let}\,\{x := \bullet\}\mathtt{seq}(e_0, \ldots, e_n)$

$\mathtt{if}(\bullet, e_1, e_2)$

$\mathtt{app}(\mathtt{app}(\lambda z.e_0, \bullet), e_1$

$\mathtt{seq}(\bullet, e_1, \ldots, e_n)$

Also note that $\mathbf{R}$ will satisfy a similar set of uniformity conditions as those satisfied by states (Definition 4.1), a fact that was discovered during the formal treatment:

Lemma 4.10 (Rcx Uniformity): `Computation` :1-319
Reduction contexts satisfy the following uniformity conditions:

(triv) $\mathbf{R}$ is closed under $\stackrel{\alpha}{\equiv}$ `Computation` :166

(par) $R \in \mathbf{R} \Rightarrow \bullet \in \mathrm{FP}(R)$ `Computation` :151

(vsub) $R \in \mathbf{R} \Rightarrow R^{\sigma} \in \mathbf{R}$ `Computation` :241

assuming $\bullet \notin \mathrm{FP}(\sigma)$

(inst) $R \in \mathbf{R} \Rightarrow R^{\Sigma} \in \mathbf{R}$ `Computation` :276

assuming $\bullet \notin (\mathrm{Dom}(\Sigma) \cup \mathrm{FP}(\Sigma))$

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 4.11 (Decomposition): `Computation` : `Rcx_Decomposition_Theorem` :231
For any λ-language, if $e \in \mathbf{E}$ with $\bullet \notin \mathrm{FP}(e)$, 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 uniqueness aspect of this lemma. A simple counterexample is the following:

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

Numerous facts concerning reduction contexts must be proved to carry out the CIU proof in PVS. One obvious, and important, property is that they are *closed under composition*:

Lemma 4.12 (Composition): `Computation` : `Rcx_Fill_Rcx` :281

$$R_0\,[\,R_1\,] = R_0^{\{\bullet \mapsto R_1\}} \in \mathbf{R}.$$

Example 4.13 (Operations on the Z of λ_{sch}): In order to define the semantics of the memory operations of λ_{sch} — `cons`, `car`, `cdr`, `setcar!`, and `setcdr!` — we need corresponding operations on the state contexts $\mathbf{Z}$ of λ_{sch}. In other words we need to be able to:

– obtain the `car` or `cdr` of a variable representing a cell in a state context ζ, i.e. the `car` or `cdr` of a variable in the $\mathrm{Traps}(\zeta)$;

– update a memory context ζ, corresponding to the effects of a `setcar!` or `setcdr!` operation, to obtain a modified ζ';

– enlarge a memory context ζ, corresponding to the effects of a `cons` operation, to obtain a modified ζ' that includes the newly allocated cell;

As a result we will define three operations:

$$\mathtt{car}(\zeta, x) = v, \quad \zeta_{\mathtt{setcdr!}(x,v)} = \zeta', \quad \text{and} \quad \zeta_{\mathtt{cons}(v_a,v_d)} = \zeta''.$$

To begin, fix ζ as in example 4.2. For brevity let

$$\sigma \stackrel{\triangle}{=} \{z_1 \mapsto y_1, \ldots, z_n \mapsto y_n\}$$

be the substitution annotating $\circ$ in ζ. Note that σ is a renaming, and as a consequence so is it's inverse $\sigma^{-1} \stackrel{\triangle}{=} \{y_1 \mapsto z_1, \ldots, y_n \mapsto z_n\}$. We can view λ_{sch} state contexts as finite maps from variables to pairs of value expressions, the domain of ζ as a map corresponding to the set $\mathrm{Traps}(\zeta)$.

For $z_i \in \mathrm{Traps}(\zeta)$ we define operations corresponding to `car` and `cdr` with respect to the state ζ as follows:

$$\mathtt{car}(\zeta, z_i) = {v_{ia}}^{\sigma^{-1}} \quad \mathtt{cdr}(\zeta, z_i) = {v_{id}}^{\sigma^{-1}}$$

for $1 \leq i \leq n$.

If $z \in \mathrm{Traps}(\zeta)$, say $z = z_i$ for simplicity, and $v \in \mathbf{V}$ is such that $\mathrm{FV}(v) \cap \{y_i \mid 1 \leq i \leq n\} = \emptyset$, then we define two related λ_{sch} state contexts $\zeta_{\mathtt{setcar!}(z_i,v)}$ and $\zeta_{\mathtt{setcdr!}(z_i,v)}$ to be the following modifications to ζ:

$$\begin{aligned}
\zeta_{\mathtt{setcar!}(z_i,v)} \stackrel{\triangle}{=} & \mathtt{let}\{y_n := \mathtt{cons}(0,0)\} \ldots \mathtt{let}\,\{y_1 := \mathtt{cons}(0,0)\} \\
& \quad \mathtt{seq}(\ldots, \mathtt{setcar!}\,(y_i, v^\sigma), \ldots, \circ^\sigma) \\
\zeta_{\mathtt{setcdr!}(z_i,v)} \stackrel{\triangle}{=} & \mathtt{let}\{y_n := \mathtt{cons}(0,0)\} \ldots \mathtt{let}\,\{y_1 := \mathtt{cons}(0,0)\} \\
& \quad \mathtt{seq}(\ldots, \mathtt{setcdr!}\,(y_i, v^\sigma), \ldots, \circ^\sigma)
\end{aligned}$$

If $v_a, v_d \in \mathbf{V}$ are such that $(\mathrm{FV}(v_a) \cup \mathrm{FV}(v_d)) \cap \{y_i \mid 1 \leq i \leq n\} = \emptyset$, then we define the λ_{sch} state context $\zeta_{\mathtt{cons}(v_a,v_d)}$ to be the following modification to ζ:

$$
\begin{array}{l}
\zeta_{\mathtt{cons}(v_a,v_d)} \stackrel{\triangle}{=} \mathtt{let}\{y := \mathtt{cons}(0,0)\} \\
\qquad\qquad \mathtt{let}\{y_n := \mathtt{cons}(0,0)\} \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\quad \mathtt{let}\{y_1 := \mathtt{cons}(0,0)\} \\
\qquad\qquad\qquad \mathtt{seq}(\mathtt{setcar!}(y_1, v_{1a}), \mathtt{setcdr!}(y_1, v_{1d}), \\
\qquad\qquad\qquad\quad \ldots, \\
\qquad\qquad\qquad\quad \mathtt{setcar!}(y_n, v_{na}), \mathtt{setcdr!}(y_n, v_{nd}), \\
\qquad\qquad\qquad\quad \mathtt{setcar!}(y, {v_a}^{\sigma}), \mathtt{setcdr!}(y, {v_d}^{\sigma}), \\
\qquad\qquad\qquad\quad \circ^{\sigma\{z \mapsto y\}})
\end{array}
$$

where $z \notin \mathrm{FV}(v_a) \cup \mathrm{FV}(v_d)$ and $z, y \notin \mathrm{Traps}(\zeta) \cup \{y_i \mid 1 \leq i \leq n\}$. A somewhat simpler description of $\zeta_{\mathtt{cons}(v_a,v_d)}$ can be given in terms of the filling operation. Let

$$
\begin{array}{lcl}
\sigma' & \stackrel{\triangle}{=} & \sigma\{z \mapsto y\} = \{z_1 \mapsto y_1, \ldots, z_n \mapsto y_n, z \mapsto y\} \\
\zeta_{\mathrm{alloc}} & \stackrel{\triangle}{=} & \mathtt{let}\,\{y := \mathtt{cons}(0,0)\}\mathtt{seq}(\circ^{\{z \mapsto y\}}) \\
\zeta_{\mathrm{assign}} & \stackrel{\triangle}{=} & \mathtt{seq}(\mathtt{setcar!}(z, v_a), \mathtt{setcdr!}(z, v_d), \circ^{\sigma'})
\end{array}
$$

Then

$$
\zeta_{\mathtt{cons}(v_a,v_d)} \stackrel{\alpha}{\equiv} \zeta_{\mathrm{alloc}}\,[\![\,\zeta\,[\![\,\zeta_{\mathrm{assign}}\,]\!]\,]\!]
$$

4.2. Uniform Semantics

We now specify what we mean by a λ-language having a *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 4.14 (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` :28 & 70

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

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

(**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

$\zeta : e \updownarrow \circ : \zeta^{\{\circ\mapsto e\}} \updownarrow \circ : \zeta\,[\![e]\!]$.

(**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

$\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1 \;\Rightarrow\; (\zeta_0 : e_0)^{\Sigma} \longrightarrow (\zeta_1 : e_1)^{\Sigma}$

for any $\Sigma \in \mathbf{F}$ 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 $\zeta' : e' \overset{\alpha}{\equiv} (\zeta_1 : e_1)^{\Sigma}$.

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

(i) $(\zeta_0 : e_0 \longrightarrow \zeta_1 : e_1) \;\Rightarrow\; \mathrm{FV}(\zeta_1\,[\![e_1]\!]) \subseteq \mathrm{FV}(\zeta_0\,[\![e_0]\!])$

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

In the original presentation [26] the definition of uniformity lacked both the closure requirements (**U.vii**), and the more trivial of the functionality conditions, i.e the first condition of (**U.i**). The first clause of the (**U.i**) requirement is really a statement of convention: when we write a reduction rule, we really are writing a family of such rules, where we allow all α equivalent variants of both the left hand side and the right hand of the reduction rule.

In the languages we consider (**U**) holds for the following reasons. The second clause in the first requirement (**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.

Example 4.15 ($\longrightarrow$ of λ_{sch}): In what follows let

$$\mathbf{V}_\zeta \stackrel{\triangle}{=} \mathbf{A} \cup \mathbf{L} \cup \mathrm{Traps}(\zeta)$$

The clauses are:

(β_{v}) $\quad \zeta : R\,[\,\mathtt{app}(\lambda x.e, v)\,] \longrightarrow \zeta : R\,[\,e^{\{x \mapsto v\}}\,]$

(int) $\quad \zeta : R\,[\,\mathtt{integer?}(v)\,] \longrightarrow \begin{cases} \zeta : R\,[\,\mathtt{true}\,] & \text{if } v \in \mathbb{Z} \\ \zeta : R\,[\,\mathtt{false}\,] & \text{if } v \in \mathbf{V}_\zeta - \mathbb{Z} \end{cases}$

(bool) $\quad \zeta : R\,[\,\mathtt{boolean?}(v)\,] \longrightarrow \begin{cases} \zeta : R\,[\,\mathtt{true}\,] & \text{if } v \in \mathbb{B} \\ \zeta : R\,[\,\mathtt{false}\,] & \text{if } v \in \mathbf{V}_\zeta - \mathbb{B} \end{cases}$

(pair) $\quad \zeta : R\,[\,\mathtt{cons?}(v)\,] \longrightarrow \begin{cases} \zeta : R\,[\,\mathtt{true}\,] & \text{if } v \in \mathrm{Traps}(\zeta) \\ \zeta : R\,[\,\mathtt{false}\,] & \text{if } v \in \mathbf{V}_\zeta - \mathrm{Traps}(\zeta) \end{cases}$

(proc) $\quad \zeta : R\,[\,\mathtt{procedure?}(v)\,] \longrightarrow \begin{cases} \zeta : R\,[\,\mathtt{true}\,] & \text{if } v \in \mathbf{L} \\ \zeta : R\,[\,\mathtt{false}\,] & \text{if } v \in \mathbf{V}_\zeta - \mathbf{L} \end{cases}$

(eq) $\quad \zeta : R\,[\,v_0 = v_1\,] \longrightarrow \begin{cases} \zeta : R\,[\,\mathtt{true}\,] & \text{if } v_0 = v_1,\ v_0, v_1 \in \mathbb{Z}, \\ \zeta : R\,[\,\mathtt{false}\,] & \text{otherwise} \end{cases}$

assuming both $v_0, v_1 \in \mathbf{V}_\zeta$

(binop) $\quad \zeta : R\,[\,v_0 \boxdot v_1\,] \longrightarrow \zeta : R\,[\,v\,]$

if $v_0, v_1 \in \mathbb{Z}$ and $v = (v_0 \boxdot v_1)$ for $\boxdot \in \{\, \mathtt{+}\,,\ \mathtt{-}\,,\ \mathtt{*}\,\}$

(br) $\quad \zeta : R\,[\,\mathtt{br}(v_0, v_1, v_2)\,] \longrightarrow \begin{cases} \zeta : R\,[\,v_1\,] & \text{if } v_0 \in \mathbf{V}_\zeta - \{\mathtt{false}\} \\ \zeta : R\,[\,v_2\,] & \text{if } v_0 = \mathtt{false} \end{cases}$

(cons) $\quad \zeta : R\,[\,\mathtt{cons}(v_a, v_d)\,] \longrightarrow \zeta_{\mathtt{cons}(v_a, v_d)} : R\,[\,z\,]$

where $\{z\} = \mathrm{Traps}(\zeta_{\mathtt{cons}(v_a, v_d)}) - \mathrm{Traps}(\zeta)$

(car) $\quad \zeta : R\,[\,\mathtt{car}(z)\,] \longrightarrow \zeta : R\,[\,v\,]$

if $z \in \mathrm{Traps}(\zeta)$ and $\mathtt{car}(\zeta, z) = v$

(cdr) $\zeta : R[\,\mathtt{cdr}(z)\,] \longrightarrow \zeta : R[\,v\,]$

if $z \in \mathrm{Traps}(\zeta)$ and $\mathtt{cdr}(\zeta, z) = v$

(setcar) $\zeta : R[\,\mathtt{setcar!}(z, v)\,] \longrightarrow \zeta_{\mathtt{setcar!}(z,v)} : R[\,0\,]$

if $z \in \mathrm{Traps}(\zeta)$

(setcdr) $\zeta : R[\,\mathtt{setcdr!}(z, v)\,] \longrightarrow \zeta_{\mathtt{setcdr!}(z,v)} : R[\,0\,]$

if $z \in \mathrm{Traps}(\zeta)$

Note that reduction is not restricted to computation states that are closed. However in the integer? and cons? rules, if one of the arguments is a variable not in the domain of the λ_{sch} state context then there is no appropriate primitive reduction step. This is also the case in the $=$, br, car, cdr, setcar!, and setcdr! rules. Also note that car, cdr, setcar!, and setcdr! are also undefined if their first argument is not a pair, i.e. a variable in the domain of the λ_{sch} state context.

Example 4.16 (Control effects in the λ_{sch}): Observe that in each reduction rule in example 4.15 the enclosing reduction context on the right-hand side of the rule is the same as the one on left-hand side. This together with the lemma 4.12 allows one to prove by induction on the length of computations the following result, stating that λ_{sch} has no control effects.

$$(\forall \zeta_{\mathrm{L}}, \zeta_{\mathrm{R}} \in \mathbf{Z})(\forall e_{\mathrm{L}}, e_{\mathrm{R}} \in \mathbf{E})(\forall R \in \mathbf{R})$$
$$((\zeta_{\mathrm{L}} : e_{\mathrm{L}} \longrightarrow\!\!\!\!\succ \zeta_{\mathrm{R}} : e_{\mathrm{R}}) \Rightarrow (\zeta_{\mathrm{L}} : R[\,e_{\mathrm{L}}\,] \longrightarrow\!\!\!\!\succ \zeta_{\mathrm{R}} : R[\,e_{\mathrm{R}}\,]))$$

Example 4.17 (Computation in the λ_{sch}): For typographical reasons let ζ' denote $\zeta_{\mathtt{cons}(0,0)}$, and ζ'' denote $\zeta'_{\mathtt{setcar!}(z_1, \lambda w.\mathtt{app}(\mathtt{app}(F, \mathtt{car}(z_1)), w))}$.

Then

$$\begin{aligned}
\zeta : \mathtt{Fact} \quad &= \quad \zeta : \mathtt{Y}(F) \\
\longrightarrow \quad & \zeta : \mathtt{let}\{z := \mathtt{cons}(0,0)\} \\
& \qquad \mathtt{seq}(\mathtt{setcar!}(z, \lambda w.\mathtt{app}(\mathtt{app}(F, \mathtt{car}(z)), w)), \mathtt{car}(z)) \\
\longrightarrow \quad & \zeta' : \mathtt{let}\{z := z_1\} \\
& \qquad \mathtt{seq}(\mathtt{setcar!}(z, \lambda w.\mathtt{app}(\mathtt{app}(F, \mathtt{car}(z)), w)), \mathtt{car}(z)) \\
\longrightarrow \quad & \zeta' : \mathtt{seq}(\mathtt{setcar!}(z_1, \lambda w.\mathtt{app}(\mathtt{app}(F, \mathtt{car}(z_1)), w)), \mathtt{car}(z_1)) \\
\longrightarrow \quad & \zeta'' : \mathtt{seq}(0, \mathtt{car}(z_1)) \\
\longrightarrow \quad & \zeta'' : \mathtt{car}(z_1) \\
\longrightarrow \quad & \zeta'' : \lambda w.\mathtt{app}(\mathtt{app}(F, \mathtt{car}(z_1)), w)
\end{aligned}$$

Now let G denote the expression $\lambda w.\mathtt{app}(\mathtt{app}(F, \mathtt{car}(z_1)), w)$, and observe that

$$\begin{array}{lcl} \zeta'' : \mathtt{app}(F, G) & = & \mathtt{app}(\lambda f.\lambda x.\mathtt{if}(x = 0, 1, f(x - 1) \star x), G) \\ \quad \longrightarrow & & \zeta'' : \lambda x.\mathtt{if}(x = 0, 1, G(x - 1) * x) \end{array}$$

Thus by simple computation we have that:

$$\begin{array}{lcl} \zeta : \mathtt{Fact}(n) & \longrightarrow & \zeta'' : \mathtt{app}(G, n) \\ \quad \longrightarrow & & \zeta'' : \mathtt{app}(\mathtt{app}(F, \mathtt{car}(z_1)), n) \\ \quad \longrightarrow & & \zeta'' : \mathtt{app}(\mathtt{app}(F, G), n) \\ \quad \longrightarrow & & \zeta'' : \mathtt{app}(\lambda x.\mathtt{if}(x = 0, 1, G(n - 1) \star x), n) \\ \quad \longrightarrow & & \zeta'' : \mathtt{if}(n = 0, 1, G(n - 1) \star n) \end{array}$$

Example 4.18 ($\mathtt{Fact}(n)$ reduction in λ_{sch}): It is now a simple matter to prove by induction that

$$(\forall \zeta \in \mathbf{Z})(\forall n \in \mathbb{Z})((n \geq 0) \;\Rightarrow\; \zeta : \mathtt{Fact}(n) \twoheadrightarrow \zeta'' : n!).$$

4.3. Approximation and Equivalence

In this subsection we define the operational approximation and equivalence relations. Operational approximation ($\sqsubseteq$) is a pre-ordering relation determined by the operational semantics. Operational equivalence ($\cong$) is the corresponding equivalence relation obtained by intersecting operational approximation with its inverse. Modulo operational equivalence, operational approximation is a partial ordering with respect to definedness. Operational equivalence formalizes the notion of equivalence as black-boxes. Treating programs as black boxes requires only observing what effects and values they produce, and not how they produce them. Our definition extends the extensional equivalence relations defined by Morris [23] and Plotkin [24] to this more general setting.

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 definitions 4.1 and 4.8, 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 4.19 ($e_0 \sqsubseteq e_1$, $e_0 \cong e_1$): `Approx` :1–76
For terms e_0, e_1 define *approximation* by:

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

and *equivalence* by:

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

using the relation $\preceq$ defined in definition 4.4.

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.

5. The CIU Theorem and it's Proof

5.1. The CIU Theorem

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

Theorem 5.1 (CIU): `CIU` : `CIU` :511
For a λ-language with uniform semantics:

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

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

As mentioned earlier 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 examined in the states resulting from their execution.

Example 5.2 (CIU in the λ_{sch}): As a simple application of the CIU theorem we show:

$$(\forall n \in \mathbb{Z})((n \geq 0) \;\Rightarrow\; \mathtt{Fact}(n) \cong n!).$$

Firstly observe that $\mathrm{FV}(\mathtt{Fact}(n)) = \mathrm{FV}(n!) = \emptyset$ Pick an arbitrary non-negative $n \in \mathbb{Z}$, and choose any ζ, R, and σ such that $\zeta : R\,[\,\mathtt{Fact}(n)^{\sigma}\,]$ and $\zeta : R\,[\,n!^{\sigma}\,]$ are closed. We show that

$$\zeta : R\,[\,\mathtt{Fact}(n)^{\sigma}\,] \updownarrow \zeta : R\,[\,n!^{\sigma}\,]\,.$$

Which by their lack of free variables, is equivalent to

$$\zeta : R\,[\mathtt{Fact}(n)] \updownarrow \zeta : R\,[n!]\,.$$

Now by example 4.18 we have that

$$\zeta : \mathtt{Fact}(n) \longrightarrow\!\!\!\succ \zeta'' : n!.$$

Thus, since in the λ_{sch}, by example 4.16, there are no control effects, we have that

$$\zeta : R\,[\mathtt{Fact}(n)] \longrightarrow \zeta'' : R\,[n!]$$

and we are done.

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.[2]

Lemma 5.3 (Closed): $\mathtt{CIU} : \mathtt{Ciu\_Closed} : 104$
Suppose e satisfies $\mathrm{FV}(e) = \emptyset$, $\mathrm{FP}(e) \subseteq \{X_0, X_1\}$, $\mathrm{FV}(e_0) = \mathrm{FV}(e_1)$, $e^{\{X_0 \mapsto e_0, X_1 \mapsto e_0\}}$ 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.

That such a closure condition is necessary was neither considered in the original treatment [26], nor in the original presentation of this proof technique [17, 18]. To see that the first two conditions (without the third) are necessary let:

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

Then $e^{\{X_0 \mapsto y_0, X_1 \mapsto y_0\}} \stackrel{\alpha}{\equiv} \lambda y_0.\lambda y_1.y_0$, and $e^{\{X_0 \mapsto y_1, X_1 \mapsto y_1\}} \stackrel{\alpha}{\equiv} \lambda y_0.\lambda y_1.y_1$

are both closed, but $e^{\{X_0 \mapsto y_0, X_1 \mapsto y_1\}} \stackrel{\alpha}{\equiv} \lambda y_0.\lambda y_1.e'$ is not.

To see that the third condition is also necessary, consider the case where e_0 and e_1 are distinct variables, y_0 and y_1. Let

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

Then $e^{\{X_0 \mapsto y_0, X_1 \mapsto y_0\}} \stackrel{\alpha}{\equiv} \lambda y_1.\lambda y_2.y_1$, and $e^{\{X_0 \mapsto y_1, X_1 \mapsto y_1\}} \stackrel{\alpha}{\equiv} \lambda y_1.y_1$,

are both closed, but $e^{\{X_0 \mapsto y_0, X_1 \mapsto y_1\}} \stackrel{\alpha}{\equiv} \lambda y_1.\lambda y_2.\lambda y_3.y_0$ is not.

[2] We may be able to eliminate this formulation of the lemma, and replace it with the assumption that $\mathrm{FV}(e_i) \subseteq \mathrm{Dom}(e)$, and show that computation (starting from traditional *closing* contexts) preserves this property.

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

Lemma 5.4 (Delay): `CIU : Ciu_Delay :121`
If e is a term, then $R\,[\,X^{\sigma}\,]^{\{X \mapsto e\}} = R\,[\,e^{\sigma}\,]^{\{X \mapsto e\}}$.

5.2. An Overview of the CIU Proof

Proof: (CIU $\Rightarrow$) `CIU : CIUR :163`
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 $e_i{}^{\hat{\sigma}} = e_i{}^{\sigma}$. Let X be the distinguished parameter used to represent traditional contexts and consider the expression $C = \zeta^{\{\circ \mapsto R^{\{\bullet \mapsto X^{\hat{\sigma}}\}}\}}$.

Lemma 5.5: `CIU : CIUR_Helper2 :145`
$C\,[\,e_j\,] = (\zeta^{\{\circ \mapsto R^{\{\bullet \mapsto X^{\hat{\sigma}}\}}\}})^{\{X \mapsto e_j\}} = \zeta^{\{\circ \mapsto R^{\{\bullet \mapsto e_j{}^{\hat{\sigma}}\}}\}}$

Now by this lemma, each $\circ : C\,[\,e_j\,]$ is closed. Thus by definition 4.19 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 5.5, and (**Unif.iii**). Thus $\zeta : R\,[\,e_0{}^{\sigma}\,] \preceq \zeta : R\,[\,e_1{}^{\sigma}\,]$ as desired. □

Proof: (CIU $\Leftarrow$) `CIU : CIUL :507`
Assume that

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

In what follows we let Σ_i denote $\{X \mapsto e_i\}$, for $0 \leq i < 2$.

We prove the more general result that:

$$(\forall \zeta, e \mid \bigwedge_{j<2} (\zeta : e)^{\Sigma_j} \text{ closed})((\zeta : e)^{\Sigma_0} \preceq (\zeta : e)^{\Sigma_1})$$

by induction on the length of the computation of $(\zeta : e)^{\Sigma_0}$.

The proof is relatively straightforward except for two harder cases, which require the use of a trick. Both harder cases occur when e_0 is a value. The easier of these is when $(\zeta : e)^{\Sigma_0}$ is also a value, and this makes up the core of the base case. The more complex of these two cases occurs when $(\zeta : e)^{\Sigma_0}$ non-trivially reduces to a value, and this makes up the core of the induction step. Both of these cases must be proved by another induction. In the base case this induction is on the number of *top-level occurrences* (see the definition of values: 3.26) of X that occur in e. While, in the induction step this induction is on the number of *top-level occurrences* of X that occur to the left, or below, where the computation is currently taking place.

(CIU $\Leftarrow$) Base Case: `CIU : CIUL_Base` :298

Suppose that $(\zeta : e)^{\Sigma_0}$ is a value state. Thus e^{Σ_0} must be a value. Then by (**trich**) of definition 3.26, we must consider three separate cases:

(i) $e \in \mathbf{V}$

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

(iii) $e \in \mathbf{Val}^{\mathbf{s}} \wedge (\forall X \in \mathrm{TLP}(e))(\Sigma(X) \in \mathbf{V})$

We prove the first two directly, the third is then proved by induction on the number of top-level occurrences of X in e.

(CIU $\Leftarrow$) Base Case (i): `CIU : CIUL_Base_Helper` :166

In the case where e is a value, we have that e^{Σ_1} is also a value by (**inst**) of definition 3.26, and hence $(\zeta : e)^{\Sigma_1}$ is also value state (note that $\zeta^{\Sigma_1} \in \mathbf{Z}$ is implicit in the hypothesis).

(CIU $\Leftarrow$) Base Case (ii): `CIU : CIUL_Base_Helper` :166

So suppose that e_0 is a value, and that e is of the form X^{σ} for some σ. Also let $\zeta_j = \zeta^{\Sigma_j}$ and $\sigma_j = \sigma^{\Sigma_j}$ for $0 \leq j < 2$. Then in this case

$$(\zeta : e)^{\Sigma_0} = \zeta^{\Sigma_0} : {e_0}^{\sigma^{\Sigma_0}} = \zeta_0 : {e_0}^{\sigma_0}$$

is a value state, and hence

$$\zeta^{\Sigma_1} : {e_0}^{\sigma^{\Sigma_1}} = \zeta_1 : {e_0}^{\sigma_1}$$

must also be a value state by (**inst**) of definition 3.26. We also claim that $\zeta_1 : {e_0}^{\sigma_1}$ is closed by lemma 5.3. 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

$$\zeta^{\Sigma_1} : {e_0}^{\sigma^{\Sigma_1}} = \zeta_1 : {e_0}^{\sigma_1} \preceq \zeta_1 : {e_1}^{\sigma_1} = \zeta^{\Sigma_1} : {e_1}^{\sigma^{\Sigma_1}}$$

Now simply observe that

$$(\zeta : e)^{\Sigma_1} = (\zeta : X^{\sigma})^{\Sigma_1} = \zeta^{\Sigma_1} : {e_1}^{\sigma^{\Sigma_1}} = \zeta_1 : {e_1}^{\sigma_1}.$$

So $(\zeta : e)^{\Sigma_1}\downarrow$.

(CIU $\Leftarrow$) Base Case (iii): `CIU : CIUL_Base_Helper_2` :287

In this case we know that e_0 is a value, and $(\zeta : e)^{\Sigma_0}$ is a value state. From this it is not hard to show the following fact. Let e' be obtained from e by replacing all non top-level occurrences of X by e_1. Then $(\zeta : e')^{\Sigma_0}$ is also a value state. Thus we can restrict our attention to the case when e only has top-level occurrences of X.

We prove a more general result. Suppose e has only top-level occurrences of the parameter X, and that $(\zeta : e)^{\Sigma_0}$ is defined. Then we show that $(\zeta : e)^{\Sigma_1}$ is also defined. As mentioned above we do this by induction on the number of occurrences of the parameter X. The base case of this induction is immediate. So suppose we have established the result for all expressions with fewer than $n+1$ occurrences of the parameter X, and that e is an expression with exactly $n+1$ occurrences. We write

$$e(\underbrace{X, X, \ldots, X, X}_{\text{exactly } n+1 \ X\text{'s}})$$

to indicate each occurrence of X, in the correct order of appearance within e. In other words we go from the leftmost occurrence to the rightmost occurrence. Let Y be a new parameter and consider the expression $e(X, X, \ldots, X, Y)$ obtained by replacing the rightmost occurrence of X by Y. By construction we have that $e(X, X, \ldots, X, Y)^{\Sigma_0}$ is of the form $R\,[\,Y^{\sigma}\,]$, where R is the reduction context obtained by filling all but the rightmost occurrence of X by the value e_0, and replacing the single annotated occurrence of Y by the reduction hole (here σ will be the substitution annotating the right-most occurrence of X in e). Thus using the CIU hypothesis we have that

$$(\zeta : R\,[\,{e_0}^{\sigma}\,] \ \preceq \ \zeta : R\,[\,{e_1}^{\sigma}\,])$$

and since by our initial hypothesis $R\,[\,{e_0}^{\sigma}\,]$ is a value, we conclude that $\zeta : R\,[\,{e_1}^{\sigma}\,]$ is defined. We now apply the induction hypothesis to the expression $e(X, X, \ldots, X, Y)^{\{Y \mapsto e_1\}}$ to conclude that

$$\zeta : (e(X, X, \ldots, X, Y)^{\{Y \mapsto e_1\}})^{\Sigma_1} = \zeta : e^{\Sigma_1}$$

is defined.

(CIU $\Leftarrow$) Induction Case: `CIU : CIUL_Helper` :376

Now suppose that $(\zeta : e)^{\Sigma_0}\downarrow$ non-trivially. Thus there is a $(\zeta^* : e^*)$ such that $(\zeta : e)^{\Sigma_0} \longrightarrow (\zeta^* : e^*)$ and $|(\zeta^* : e^*)| < |(\zeta : e)^{\Sigma_0}|$. Hence by **(Unif.vi)** either

- **(i)** there is a state $(\zeta' : e')$ such that $(\zeta : e) \longrightarrow (\zeta' : e')$ and $(\zeta' : e')^{\Sigma_0} = (\zeta^* : e^*)$, or else
- **(ii)** $(\zeta : e)$ touches the parameter X.

(CIU $\Leftarrow$) Induction Case (i): `CIU : CIUL_Helper` :376

In (**i**) we have by the induction hypothesis that $(\zeta' : e')^{\Sigma_0} \preceq (\zeta' : e')^{\Sigma_1}$, and thus by hypothesis $(\zeta' : e')^{\Sigma_1}\downarrow$. This case is completed by observing that since $(\zeta : e) \longrightarrow (\zeta' : e')$, by (**Unif.v**) we have that $(\zeta : e)^{\Sigma_1} \longrightarrow (\zeta' : e')^{\Sigma_1}$. Thus $(\zeta : e)^{\Sigma_1}\downarrow$ 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) ($e_0 \notin \mathbf{V}$): `CIU` : `CIUL_Helper` :376

Since $e_0 \notin \mathbf{V}$ we have by (**Decomposition**) (i.e. lemma 4.11) 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**)

$$(\zeta : R\,[\,X^{\sigma}\,])^{\Sigma_0} = (\zeta : R\,[\,{e_0}^{\sigma}\,])^{\Sigma_0} = (\zeta : R\,[\,R_0\,[\,r_0\,]^{\sigma}\,])^{\Sigma_0}$$
$$= (\zeta : R\,[\,{R_0}^{\sigma}\,[\,{r_0}^{\sigma}\,]\,])^{\Sigma_0}.$$

Now since $R\,[\,{R_0}^{\sigma}\,]$ is a reduction context, and ${r_0}^{\sigma}$ is still a redex by (**inv**) (i.e. lemma 3.28), 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**) $(\zeta : R\,[\,{e_0}^{\sigma}\,])^{\Sigma_0} \longrightarrow (\zeta' : e')^{\Sigma_0}$ and hence $(\zeta' : e')^{\Sigma_0}\downarrow$, so by the induction hypothesis $(\zeta' : e')^{\Sigma_1}\downarrow$. Now also by (**Unif.v**) $(\zeta : R\,[\,{e_0}^{\sigma}\,])^{\Sigma_1} \longrightarrow (\zeta' : e')^{\Sigma_1}$, consequently we may conclude that $(\zeta : R\,[\,{e_0}^{\sigma}\,])^{\Sigma_1}\downarrow$. Now put $\zeta_1 = \zeta^{\Sigma_1}$, $R_1 = R^{\Sigma_1}$, and $\sigma_1 = \sigma^{\Sigma_1}$. Then by lemma 5.3 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 $(\zeta : e)^{\Sigma_1}\downarrow$ since they are identical.

(CIU $\Leftarrow$) Induction Case (ii) ($e_0 \in \mathbf{V}$): `CIU` : `CIUL_HH` :483

Since we are assuming that $(\zeta : e)^{\Sigma_0}\downarrow$ non-trivially, we know that the term e^{Σ_0} is not a value, and so must decompose uniquely into $R^*\,[\,r^*\,]$. Furthermore since e_0 is a value we can find a D containing both X and $\bullet$, and a d such that:

$$R^* = D^{\Sigma_0} \quad r^* = d^{\Sigma_0} \quad e = D^{\{\bullet \mapsto d\}}$$

Although d need not be a redex, and D need not be a reduction context, since in general both could contain occurrences of X where values are actually required.

We say a top-level occurrence of X is *touched* in e if it occurs either in d, or to the left of the $\bullet$ in D. In what follows we shall generalize the notation we introduced in the base case, and write

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

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 other words the leftmost touched occurrence corresponds to the first X in this list, while the rightmost touched occurrence corresponds to the X just before the $\mid$.

Now the induction hypothesis can be used to show that

$$(\zeta : e(e_0, e_0, \ldots, e_0 \mid X, X, \ldots, X))^{\Sigma_0}$$
$$\preceq (\zeta : e(e_0, e_0, \ldots, e_0 \mid X, X, \ldots, X))^{\Sigma_1}$$

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 D and d. The partial filling of D and d 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 $\zeta_1 = \zeta^{\Sigma_1}$ 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 left-hand 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 *rightmost top-level 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:

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

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 top-level 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 $D' = e(X, X, \ldots, X, R_1 \mid e_1, e_1, \ldots, e_1)$ and $d' = r_1$ suffices. □

5.3. THE CIU PROOF IN MORE DETAIL

One of the complications that is glossed over in the theoretical treatment is the need to keep careful track of the free variables and parameters occurring in the expressions under manipulation, as mentioned in the discussion surrounding lemma 5.3. Most of the necessary facts needed for the CIU proof are collected in the theory `Frees` found in `Frees.pvs`. For example, the process of filling an expression can cause the free variables and parameters that occur in its annotations to be lost. One of the main results proved in this theory is the `Frees_Fill_Theorem` which is used in CIU to proved that the closedness of expressions is maintained throughout the proof.

```
Frees_Fill_Theorem: THEOREM
   (dom(s) = dom(s1) AND
    (FORALL (i:(dom(s))):
         FV(sub(s)(i)) = FV(sub(s1)(i)) AND
         FP(sub(s)(i)) = FP(sub(s1)(i)))) IMPLIES
     (FV(Fill(e, s)) = FV(Fill(e, s1)) AND
      FP(Fill(e, s)) = FP(Fill(e, s1)))
```

The hard case in the induction step requires formalizing the idea of an expression being almost a reduction context. That is an expression that may contain parameters (other than the reduction hole), but has the property that if one fills the top-level occurrences of these holes, that occur to the left of the reduction hole, with values the result is indeed a reduction context. These are called pseudo-reduction contexts in the formal treatment, and are denoted by $\mathbf{R}^{\mathrm{s}}$. They are defined, and their basic properties established in `Rcx_Hole.pvs`. It is also in this theory that we define the hole rank of an expression. This is the notion of rank used in the induction hypothesis of the CIU proof, and in the third case of the base case. Some of the facts concerning this notion of rank, and its definition can be found in `tlfprank.pvs`. In addition we develop some machinery needed for manipulating the right-most top-level occurrence of a parameter, such as replacing it by another, new parameter, and filling such a parameter. Some of this machinery can be found in `blSwap.pvs`.

The actual proof of CIU in PVS involved proving over one thousand distinct facts (1200 to be precise). It takes PVS 10577 seconds (176 minutes) of CPU time running on a Linux machine configured with 2GBytes of main memory and 4×550 MHz Xeon PIII processors. The proof of CIU itself can be broken down into several groups of lemmas which are listed below, along with statistics.

- CIU Right, and helpers (6 lemmas, 184 seconds)
- CIU Left base case for Values and Holes (1 lemma, 102 seconds)
- CIU Left base case for $\mathbf{Val}^{\mathsf{s}}$ (11 lemmas, 797 seconds)
- CIU Left induction case (21 lemmas, 994 seconds)

6. Conclusions

The main aim of this work was to provide a formally verifed foundation for the development of operational tools for reasoning about higher-order programs with imperative/stateful effects. One of the most important aspects of the work reported here is the way it used PVS to put the paper [26] through a fine tooth comb. The final product [19] is certainly the better for it. 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 are necessary. Unstated but necessary uniformity requirements were needed for the proof to be carried out. Finally some extra conditions on the terms or contexts considered in the actual CIU proof were needed to ensure that replacing one expression preserves the *closedness* of the computation states involved. Another aspect of the work presented here is that it demonstrates 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 plausible that tools like PVS will, in the future, play an important part in language design, implementation, and even program development itself. More general conclusions concerning formal developments within PVS can be found in [9].

The formalization of the *annotated holes* technique is also a highlight of the work, and provides the novel technique with a rock solid foundation.

6.1. PVS Statistics

The dump file containing all the PVS definitions, facts, and proofs is 19 MBytes and is available from `http://mcs.une.edu.au/~pvs/` [6], though we should point out that there is a substantial amount of whitespace in the S-expressions that PVS uses to represent proofs. In fact the gzipped dump file is only 800 KBytes. Even so, the size of the result is a consequence of the fact that the development started from square one. It thus represents roughly eight times as much work as was required to prove the Church–Rosser theorem in PVS, as reported in [8]. The actual PVS proof resides in over one hundred files, and the presentation of the results in this paper made use of the names of these files, and line numbers within these files. This system of presentation works well for the statements of the results contained in the development, but not for the formal proofs.

Prior to this work and the earlier work reported in [8], 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 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 [1] and we thank them for their effort. We also like to thank Carolyn Talcott for her careful comments on the final draft, and for encouraging our use of PVS in the style presented here. We would also like to thank David Basin and the anonymous referees for helpful comments that resulted, hopefully, in a better paper.

References

1. 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`.
2. J. Crow, S. Owre, J. Rushby, N. Shankar and M. Srivas. A Tutorial Introduction to PVS. Technical report, SRI International, 1995. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida.
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 (approximately 17 Megabytes) available at `http://mcs.une.edu.au/~pvs/`.
7. J. Ford and I. A. Mason. Establishing a General Context Lemma in PVS. In G Antoniou and G Governatori (editors), *Proceedings of the 2nd Australasian Workshop on*

Computational Logic, AWCL'01 , pages 75–91, 2001. Available as postscript from `http://mcs.une.edu.au/~iam/Data/Papers/01awcl.ps`.

8. J. Ford and I. A. Mason. Operational Techniques in PVS – A Preliminary Evaluation. In C. J. Fidge (editor), *Proceedings of the Australasian Theory Symposium, CATS '01*, Volume 42 of *Electronic Notes in Theoretical Computer Science*, pages 124–142. Elsevier Science, 2001. Available as postscript from `http://mcs.une.edu.au/~iam/Data/Papers/01cats.ps`.
9. J. Ford, I. A. Mason and N Shankar. Lessons Learned from Formal Developments in PVS., 2002. presented at LICS'02 (within FLoC'02), Copenhagen.
10. J. Ford and N Shankar. Verifying Shostak. In *The proceedings of CADE-18 (within FLoC'02), Copenhagen.*, 2002.
11. 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.
12. Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1984.
13. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, Volume 6, pages 308–320, 1964.
14. P. J. Landin. The next 700 programming languages. *Comm. ACM*, Volume 9, pages 157–166, 1966.
15. I. A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, Volume 12, pages 171–201, 1999.
16. 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.
17. 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.
18. I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, Volume 1, pages 287–327, 1991.
19. I. A. Mason and C. L. Talcott. Feferman–Landin Logic. In W. Sieg, R. Sommer and C. Talcott (editors), *Reflections – A symposium honoring Solomon Feferman on his 70th birthday*. *Lecture Notes in Logic*, 2001.
20. 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.
21. J. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, Volume 23, 1999. An abridged version appeared as [20].
22. R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, Volume 4, pages 1–22, 1977.
23. J. H. Morris. *Lambda calculus models of programming languages*. Ph.D. thesis, Massachusetts Institute of Technology, 1968.
24. G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, Volume 1, pages 125–159, 1975.
25. N. Shankar. Personal communication, July 2000.
26. C. L. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1996.