Electronic Notes in Theoretical Computer Science 42 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume42.html` 19 pages

Operational Techniques in PVS — A Preliminary Evaluation

Jonathan M. Ford [2] and Ian A. Mason [1,3]

School of Mathematics, Statistics, & Computing Science, U.N.E
Armidale, NSW 2351 Australia

Abstract

In this paper we present a preliminary analysis of the suitability of using PVS as a tool for developing operational semantics and programming logics in a semi-automatic fashion. To this end we present a formalized proof of the Church–Rosser theorem for a version of the call-by-value lambda calculus in the spirit of Landin's ISWIM. The proof is developed in the PVS system, and is used as a test bed or benchmark for evaluating the applicability of that system for carrying out more complex operational arguments. Our approach is relatively unusual in that it is based on the named variable approach, and concentrates on the call-by-value version of the β rule. Although there are numerous computer-based proofs of the Church–Rosser theorem in the literature, all of the existing proofs eliminate the need to treat α conversion. The novel aspects of our approach are that: we use the PVS system, especially its built-in abstract data types facility, to verify a version of the Church–Rosser theorem; we formalize a version of the λ-calculus, as it normally appears in textbooks, rather than tailoring it to suit the machine or system; we treat an ISWIM variation on the call-by-value version of the λ-calculus, rather than the simpler traditional call-by-name version. However the main aim of the work reported here was to evaluate PVS as a tool for developing, state of the art, operational based programming logics for realistic programming languages.

1 Introduction

In this paper we present a formalized proof [4] of the Church–Rosser theorem for a version of the call-by-value lambda calculus [23] in the spirit of Landin's ISWIM [9]. The proof is developed in the PVS [2] system, and is used as a test bed or benchmark for evaluating the applicability of that system for carrying out more complex operational arguments, such as computing with

[1] The second author was partially supported by U.N.E small ARC grant No. 8202 881 499.
[2] Email: `jford@turing.une.edu.au`
[3] Email: `iam@turing.une.edu.au`

contexts [11], developing Feferman-Landin logic [15] or proving the Curry-Howard isomorphism theorem for various versions of typed lambda calculi, and the corresponding logics [28].

Our approach is relatively unusual in that it is based on the named variable approach, and concentrates on the call-by-value version of the β rule. Our desire to handle the more complex β rule is motivated by our desire to extend this work to more realistic programming languages. The proof is based on the, now standard, Tait–Martin-Löf notion of parallel reduction.

Although there are numerous computer-based proofs of the Church–Rosser theorem in the literature [26,8,25,22,17,20] (see section 5 for a brief survey), all of the existing proofs eliminate the need to treat α conversion by using reasonably standard encoding tricks. α conversion can be avoided either by eliminating the syntactic category of named variables in favour of de Bruijn indices [3], or by using the variables of the logical framework itself [6], rather than incorporating one into the encoded system.

Only the treatment by McKinna and Pollack [17] uses named variables, rather than de Bruijn indices or the variables of the logical framework itself. However McKinna and Pollack, following on in Gentzen and Prawitz's footsteps, make a rigorous syntactic distinction between free and bound variables. Our named variable approach differs from McKinna and Pollack in that we do not make such a distinction between free and bound variables. Consequently, unlike McKinna and Pollack, we must formalize α-equivalence, prior to developing the various notions of reduction. Again this desire to handle the λ-calculus as it is, rather than how the PVS system (or any other theorem prover) would prefer it to look, is motivated by our desire to extend this treatment to richer systems that may not be so easily streamlined. In a similar vein, McKinna and Pollack also use what they term *tricky* representations, that are faithful to the intuitive notion, but whose faithfulness is left unformalized. A typical example from [17] is the representation of a renaming of variables as a Lisp-style association list, i.e. a list of pairs of variables, and using a Lisp-style `assoc` operation to obtain the new name for a variable. The fact that such an alist represents a function is an *accidental* feature of `assoc`, as is the fact that `cons`ing onto a the front of an alist *shadows* any old values associated with the variable. Indeed they point out that this representation makes it very difficult to construct bijective renamings, to the point that they avoid doing so. In a similar vein McKinna and Pollack almost exclusively use lists for representations, when the natural mathematical treatments use functions. Our approach, on the other hand, elects to use the natural mathematical representation wherever possible. We will discuss this, and its consequences in more detail shortly. Thus the novel aspects of our approach are that:

- we use the PVS system, especially its built-in abstract data types facility, to verify a version of the Church–Rosser theorem;
- we formalize a version of the λ-calculus, as it normally appears in textbooks,

rather than tailoring it to suit the machine or system;

- we treat an ISWIM variation on the call-by-value version of the λ-calculus, rather than the simpler traditional call-by-name version.

However the main aim of the work reported here was to evaluate PVS as a tool for developing, state of the art, operational based programming logics for realistic programming languages.

2 A Whirlwind Tour of the Church–Rosser Theorem

Following in the footsteps of [24] we provide a whirlwind tour of the call-by-value λ-calculus and the Church–Rosser Theorem. The Church–Rosser Theorem was historically taken to be a consistency proof for a system designed to be a functional foundation of Mathematics (i.e. λ-calculus) [1]. These days the λ-calculus and the Church–Rosser Theorem is part of almost any Theoretical Computer Scientist's education.

Our treatment of the λ-calculus follows that of Landin [9] (a la ISWIM) in that we include constants and primitive operations, as well as the more usual λ-abstractions and applications. The primitive operations each posses an arity, whose existence we will suppress in the remainder of this paper. We start with an infinite set of variables, X (x, y, z range over X), a set of constants, A, and set of primitive operation symbols, O, and define by induction the set of λ-expressions Λ. For our purposes Λ is the least set satisfying:

$$\Lambda ::= X \cup A \cup \lambda X.\Lambda \cup \Lambda(\Lambda) \cup O(\Lambda, \ldots, \Lambda)$$

The inductive nature of Λ allows for a myriad of rank functions, as well as structural recursions. Simple definitions that use structural recursions are the sets of variables, free variables ($\mathrm{FV}(e)$), and bound variables occurring in an expression. As a prelude to defining (capture avoiding) substitution, $e_0[x := e_1]$, and the companion notion of renaming of bound variables (a.k.a α-conversion), careful treatments of the λ-calculus will define, by structural recursion, the notion of a variable renaming. One nice property possessed by variable renamings is that it preserves rank, unlike substitution. α-conversion and substitution are then themselves defined by structural recursion. At this point it usual to define a notion of α-equivalence, $\overset{\alpha}{\equiv}$, and either remark that λ-expressions are now only distinguished up to $\overset{\alpha}{\equiv}$, or much less frequently form the quotient $\Lambda/\overset{\alpha}{\equiv}$. The latter of course requires first establishing that $\overset{\alpha}{\equiv}$ is a congruence, and that the operations of interest (such as renaming and substitution) are functional with respect to it. The rule for deducing the $\overset{\alpha}{\equiv}$ of λ-abstractions, $\lambda x_0.e_0 \overset{\alpha}{\equiv} \lambda x_1.e_1$, reduces to showing $e_0[x_0 := y] \overset{\alpha}{\equiv} e_1[x_1 := y]$ for suitable y. From a logical point of view we have, at least, two choices: we can require that this hypothesis is true for some fresh y (i.e. $y \notin \mathrm{FV}(e_0) \cup \mathrm{FV}(e_1)$), or we can require that it is true for all such y. Even though these two

$(\beta_{\mathrm{v}})\quad (\lambda x.e)v \xmapsto{\beta_v} e[x := v] \quad$ for v a value.

$(\delta)\quad o(e_1, \ldots, e_n) \xmapsto{\beta_v} v \quad$ if $e_1, \ldots, e_n \in \mathrm{Dom}(\delta)$ and $\delta(o, e_1, \ldots, e_n) = v$.

$$(\mathrm{C}_\lambda)\quad \frac{e_0 \xmapsto{\beta_v} e_1}{\lambda x.e_0 \xmapsto{\beta_v} \lambda x.e_1}$$

$$(\mathrm{C}_{\mathrm{left}})\quad \frac{e_0 \xmapsto{\beta_v} e_1}{e(e_0) \xmapsto{\beta_v} e(e_1)} \qquad (\mathrm{C}_{\mathrm{right}})\quad \frac{e_0 \xmapsto{\beta_v} e_1}{e_0(e) \xmapsto{\beta_v} e_1(e)}$$

$$(\mathrm{C}_{\delta\mathrm{i}})\quad \frac{e \xmapsto{\beta_v} e'}{o(e_1, \ldots, e_i, e, e_{i+2}, \ldots, e_n) \xmapsto{\beta_v} o(e_1, \ldots, e_i, e', e_{i+2}, \ldots, e_n)}$$

for $0 \le i \le n$.

Fig. 1. Single Step β-value-reduction

forms will generate the same relation, the precise choice will have non-trivial consequences for the rigorous machine checked proof.

The time is now ripe to define (call-by-value) computation on λ terms. To do this one specifies the set of *values*, V, as a (sometimes inductively defined) subset of the λ terms, which in this case consists of constants, variables, and λ abstractions. The single step β-value-reduction relation, $\xmapsto{\beta_v}$, is parametric in a δ function, $\delta : O(A^n) \mapsto V$, and is generated by the rules in figure 1. As defined $\xmapsto{\beta_v}$ is neither reflexive, nor transitive.

As is standard, given a relation R, we define R^* to be the transitive reflexive closure of R. A relation R is said to have the *diamond property* written, $\Diamond(\mathrm{R})$, iff

$$(\forall e_0, e_1, e_2)(e_0 \mathrel{\mathrm{R}} e_1 \;\wedge\; e_0 \mathrel{\mathrm{R}} e_2 \;\Rightarrow\; (\exists e_3)(e_1 \mathrel{\mathrm{R}} e_3 \;\wedge\; e_2 \mathrel{\mathrm{R}} e_3))$$

A relation is said to be *Church–Rosser* or *confluent* iff $\Diamond(\mathrm{R}^*)$. The Church–Rosser theorem states that $\Diamond(\xmapsto{\beta_v}^{*})$. Since $\xmapsto{\beta_v}$ in neither reflexive nor transitive it is not the case the $\Diamond(\xmapsto{\beta_v})$. To see this consider the following two counterexamples. In each case the dotted arrow does not belong to $\xmapsto{\beta_v}$.

Reflexivity:

$$\begin{tikzcd}
 & (\lambda x.y)(\lambda w.((\lambda x.x)z)) \arrow[dl] \arrow[dr] & \\
(\lambda x.y)\lambda w.z \arrow[dr] & & y \arrow[dl, dashed] \\
 & y &
\end{tikzcd}$$

Transitivity:

$$\begin{tikzcd}
& (\lambda x.xx)(\lambda w.((\lambda x.y)z)) \arrow[dl] \arrow[dr] & \\
(\lambda x.xx)(\lambda w.y) \arrow[dr] & & \lambda w.(((\lambda x.y)z)((\lambda x.y)z)) \arrow[dl, dashed] \\
& (\lambda w.y)(\lambda w.y) &
\end{tikzcd}$$

The Tait–Martin-Löf proof of this theorem involves defining a parallel reduction relation, $\overset{p}{\longmapsto}$, which is reflexive, merges left and right application reduction into a single rule, and allows, in a single step, both the abstraction and the value to be reduced in the β_v reduction step, see figure 2 for the complete definition. The proof then follows from establishing three facts:

(i) $\Diamond(\overset{p}{\longmapsto})$ holds. This is the delicate part of the proof. Our version of the proof is a structural induction on the proof that $e_0 \overset{p}{\longmapsto} e_1$ and proceeds by case analysis on the last step in this proof. This is not the only method of proof, for example Takahashi [29] has recently published a proof that does not analyse the reductions, but rather relies on taking the *maximum* parallel reduction step (called the *complete development*). We do not follow this version of the proof.

(ii) $\Diamond(\overset{p}{\longmapsto})$ implies that $\Diamond(\overset{p}{\longmapsto}^*)$. This actually holds for any relation R, and has a nice geometric proof. It is also relatively simple to establish using a double induction.

$$\begin{tikzcd}
L \arrow[r] \arrow[d] & M_1 \arrow[r] & M_2 \arrow[r] & \cdots \arrow[r] & M \\
N_1 \arrow[d] & & & & \\
N_2 \arrow[d] & & & & \\
\vdots \arrow[d] & & & & \\
N & & & &
\end{tikzcd}$$

fills out to

$$\begin{tikzcd}
L \arrow[r] \arrow[d] & M_1 \arrow[r] \arrow[d] & M_2 \arrow[r] \arrow[d] & \cdots \arrow[r] & M \\
N_1 \arrow[r] \arrow[d] & K_{11} \arrow[r] \arrow[d] & K_{12} & \cdots & \\
N_2 \arrow[r] \arrow[d] & K_{21} & \cdots & & \\
\vdots \arrow[d] & \vdots & & & \vdots \arrow[d] \\
N & & & \cdots \arrow[r] & K
\end{tikzcd}$$

(iii) $\overset{p}{\longmapsto}^*$ is the same relation as $\overset{\beta_v}{\longmapsto}^*$. Pollack points out [24] that this step is usually considered trivial, but can cause problems for the named variable approach. In our version of the proof, neither of these observations is true. The proofs are non-trivial, but non-problematic.

2.1 A Summary of the Encoding and Proof

To summarize, our encoding of the λ-calculus, and the subsequent proof of the Church–Rosser theorem has the following shape:

$$(\mathrm{P}_{\mathrm{refl}})\quad e \overset{p}{\longmapsto} e$$

$$(\mathrm{P}_{\beta_\mathrm{v}})\quad \frac{e \overset{p}{\longmapsto} e' \quad v \overset{p}{\longmapsto} v'}{(\lambda x.e)v \overset{p}{\longmapsto} e'[x := v']} \quad \text{for } v \text{ a value.}$$

$$(\delta)\quad o(e_1,\ldots,e_n) \overset{p}{\longmapsto} v \quad \text{if } e_1,\ldots,e_n \in \mathrm{Dom}(\delta) \text{ and } \delta(o,e_1,\ldots,e_n) = v.$$

$$(\mathrm{P}_\lambda)\quad \frac{e_0 \overset{p}{\longmapsto} e_1}{\lambda x.e_0 \overset{p}{\longmapsto} \lambda x.e_1}$$

$$(\mathrm{P}_{\mathrm{app}})\quad \frac{e_0 \overset{p}{\longmapsto} e_1 \quad e_2 \overset{p}{\longmapsto} e_3}{e_0(e_2) \overset{p}{\longmapsto} e_1(e_3)} \qquad (\mathrm{P}_\delta)\quad \frac{e_i \overset{p}{\longmapsto} e_i' \quad \text{for } 0 \le i \le n.}{o(e_1,\ldots e_n) \overset{p}{\longmapsto} o(e_1',\ldots,e_n')}$$

Fig. 2. Parallel β-value-reduction

- **Syntax:**
  Define the syntax of the λ-expressions, and related notions of expression rank, free and bound variables, renaming, and substitution.
- **Alpha:**
  Define $\overset{\alpha}{\equiv}$ and establish that it is an equivalence (congruence) relation, and that renaming, and substitution are functional modulo $\overset{\alpha}{\equiv}$. Several lemmas concerning the interaction between renaming, and substitution also need to be established.
- **Quotient:**
  Formalize the notion of identifying λ-expressions only up to $\overset{\alpha}{\equiv}$.
- **β and δ:**
  Define single step β-value-reduction (parametric in a δ function)
- **Closures:**
  Develop a general mechanism for generating the transitive, reflexive closure of a relation, as well as a method for establishing facts about such closures (e.g. rank induction).
- **Parallel:**
  Define the notion of single step parallel reduction, and establish some basic facts concerning it. For example that it preserves values, and is preserved by substitution:

$$e \overset{p}{\longmapsto} e' \;\Rightarrow\; (e \in V \;\Rightarrow\; e' \in V)$$
$$(e_0 \overset{p}{\longmapsto} e_1 \wedge v_0 \overset{p}{\longmapsto} v_1) \;\Rightarrow\; e_0[x := v_0] \overset{p}{\longmapsto} e_1[x := v_1]$$

- **Proof:**
  The proof now consists of the three steps described above.
  · $\Diamond(\overset{p}{\longmapsto})$ holds
  · $\Diamond(\overset{p}{\longmapsto})$ implies that $\Diamond(\overset{p}{\longmapsto}^*)$
  · $\overset{p}{\longmapsto}^*$ is the same relation as $\overset{\beta_v}{\longmapsto}^*$

3 An Overview of PVS

PVS is a verification system developed by SRI and draws on almost 20 years of experience at designing such systems. 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.

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 in Emacs via a Lisp-like interface. 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 resistant 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 are the same if they are constructed from the same components).
- Eta Axioms (an object constructed from the same components of X is identical to X)

- Accessor axioms (a component of a constructor is the appropriate constructor argument)
- Induction Schemes (for induction on the structure of the ADT)
- 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 theorem 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.

4 A Tour of the Encoding of the λ-calculus and the Church–Rosser Proof

4.1 Syntax

The set of variables is defined as a type with the property that for every finite set of variables, there is a variable not contained within it. From this definition a *new* function can be defined on finite sets of variables, with the property that $(\forall y \in \mathrm{Fin}(X)) new(y) \notin y$.

The set of λ-expressions is defined as an abstract data type.

```
Λ[A: TYPE+, O: TYPE+, # : [O → nat]]: DATATYPE
 BEGIN
  IMPORTING X
  Var(x: X): Var?
  λ(x: X, e: Λ): λ?
  app(e: Λ, e′: Λ): app?
  K(a: A): K?
  δ(o: O, l: list[Λ]): δ?
 END Λ
```

The datatype takes three parameters, a non-empty type A for the atoms, a non-empty type O for the primitive functions, and a function # which maps each primitive function to its arity.

The rank of a λ-expression is defined using the automatically generated recursive combinator (see section 3). The rank is used throughout our specification for carrying out inductive proofs on λ-expressions. It is proved that the rank of an expression is larger than the ranks of all its subexpressions.

$$\begin{aligned}
&rk(e) = 1 \quad e \in (X \cup A)\\
&rk(\lambda x.e) = 1 + rk(e)\\
&rk(e(e_0)) = 1 + rk(e) + rk(e_0)\\
&rk(o(e_1, e_2, \ldots, e_n)) = 1 + rk(e_1) + rk(e_2) + \ldots + rk(e_n)
\end{aligned}$$

We develop the notion of free variables (FV) via an inductive definition.

$$\begin{aligned}
&\mathrm{FV}(x) = \{x\}\\
&\mathrm{FV}(a) = \{\}\\
&\mathrm{FV}(\lambda x.e) = \mathrm{FV}(e) - \{x\}\\
&\mathrm{FV}(e(e_0)) = \mathrm{FV}(e) \cup \mathrm{FV}(e_0)\\
&\mathrm{FV}(o(e_1, e_2, \ldots, e_n)) = \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \ldots \cup \mathrm{FV}(e_n)
\end{aligned}$$

We can apply the *new* function to the set of free variables of an expression to get a fresh variable and thus avoid accidental capture. The problem here is that the *new* function is defined on finite sets of X, and $\mathrm{FV}(e)$ is defined as having type $\mathit{setof}(X)$, so taking the *new* of $\mathrm{FV}(e)$ will lead to the generation of a TCC. Including a judgement, however stating that $\mathrm{FV}(e) \in \mathrm{Fin}(X)$ in our specification suppresses the production of such TCCs. The prelude contains judgements about finite sets unions, intersection and so forth (e.g. $(\forall X, Y \in \mathrm{Fin}(T))(X \cup Y) \in \mathrm{Fin}(T))$. Thus, even expressions of the form $\mathit{new}(\mathrm{FV}(e) \cup \mathrm{FV}(e_0))$ typecheck without producing TCCs.

Defining FV allows us to treat renaming and substitution. The renaming function replaces all free occurrences of one variable with another. To achieve this it may sometimes be necessary to rename the bound variables of an expression to prevent capture.

$$(\lambda y.x)[x := y] \neq \lambda y.y$$

We do this by renaming all λ bound variables.

$$(\lambda y.x)[x := y] = \lambda z.y \quad \text{for some new variable } z$$

In general for λ-abstractions, renaming is defined as

$$(\lambda x.e)[y := z] = \lambda x_0.(e[x := x_0][y := z]) \quad \text{where} \quad x_0 = \mathit{new}(\mathrm{FV}(e) \cup \{y, z\})$$

It is in defining the renaming function that we first run into trouble with TCCs. As mentioned in section 3, TCCs need to be proved by the PVS system, or the user. Unfortunately it is possible to generate unprovable TCCs, often from fairly innocuous specifications. For example, consider the lambda case of our renaming function:

```
$e[y := z]$ : Recursive $\Lambda$ =
  Cases $e$ of :
      . . .
      $\lambda x.e_0$ : let $x_0 = new(\mathrm{FV}(e) \cup \{y, z\})$ in
          $\lambda x_0.(e_0[x := x_0][y := z])$
      . . .
  Endcases
 Measure $rk(e)$
```

To prove that the function terminates, we need to show that each expression in the recursive calls is smaller than the original expression. Now clearly $rk(e_0) < rk(e)$, as e_0 is a subexpression of e, but in general PVS knows nothing about $rk(e_0[x := x_0])$. This will lead to the unprovable TCC:

$$\forall e' : rk(e') < rk(e)$$

To overcome this problem we build more information into the declared type of the renaiming function. In particular we express that the rank of its value is no greater than the rank of its argument:

$$e[y := z] : \text{Recursive}\, \{e_0 \in \Lambda \;\big|\; rk(e_0) \leq rk(e)\} = \quad \ldots$$

This gives PVS the information it needs to establish that the nested recursion in the λ case terminates.

4.2 Alpha

We formally define $\overset{\alpha}{\equiv}$ using an inductive definition. As mentioned in section 2, several such definitions are possible. Consider, for example, the first case mentioned for λ-abstractions. Then $\lambda x_0.e_0 \overset{\alpha}{\equiv} \lambda x_1.e_1$ and $\lambda x_1.e_1 \overset{\alpha}{\equiv} \lambda x_2.e_2$ if $\exists y_1, y_2$ such that $e_0[x_0 := y_1] \overset{\alpha}{\equiv} e_1[x_1 := y_1]$ and $e_1[x_1 := y_2] \overset{\alpha}{\equiv} e_2[x_2 := y_2]$. The problem with this is that transitivity is difficult to prove because y_1 and y_2 are not necessarily the same variable (subsequent lemmas prove that the choice of variables is irrelevant but rely on $\overset{\alpha}{\equiv}$ being transitive). The other case for λ-abstractions requires $e_0[x_0 := y] \overset{\alpha}{\equiv} e_1[x_1 := y]$ for all y not free in e_0 or e_1. However this definition is unwieldy in proofs where we require the new variable to be outside the free variables of some other expression. Accordingly the relation for $\overset{\alpha}{\equiv}$ is not either of the above, but relies instead on the existence of a finite set of variables. For $\lambda x_0.e_0$ and $\lambda x_1.e_1$ to be $\overset{\alpha}{\equiv}$, the rule requires that $\forall y$ outside of this finite set, and not contained within the free variables of either expression $e_0[x_0 := y_1] \overset{\alpha}{\equiv} e_1[x_1 := y_1]$. This gives us the greatest control over the new variable, and hence the greatest ease at proving theorems.

$$\frac{\exists T \in \mathrm{Fin}(X) \quad \text{where} \quad \forall x \notin T \cup \mathrm{FV}(e_0) \cup \mathrm{FV}(e_1) \quad e_0[x_0 := x] \overset{\alpha}{\equiv} e_1[x_1 := x])}{\lambda x_0.e_0 \overset{\alpha}{\equiv} \lambda x_1.e_1}$$

Interestingly enough, proving the simplest properties of $\overset{\alpha}{\equiv}$ is quite challenging. For example, the following three properties of $\overset{\alpha}{\equiv}$ are proved simultaneously by induction on the rank of expressions, an approach similar to that used in [11]:

(a) $e_0 \overset{\alpha}{\equiv} e_1 \Rightarrow e_0[x := y] \overset{\alpha}{\equiv} e_1[x := y]$

(b) $(x \neq x_1 \wedge x \neq y_1 \wedge x_1 \neq y) \Rightarrow e[x := y][x_1 := y_1] \overset{\alpha}{\equiv} e[x_1 := y_1][x := y]$

(c) $y \notin \mathrm{FV}(e) \Rightarrow e[x := y][y := z] \overset{\alpha}{\equiv} e[x := z]$

4.3 Quotient

In defining the quotient space modulo α equivalence, $\overset{\alpha}{\equiv}$, we need to build a comprehensive theory about the new type. Many of the lemmas are similar to those found in a PVS ADT file, but also require redefining such things as renaming and free variables. These are done with respect to the old functions. For example, let q be the function which maps a λ-expression to its α coset. The free variables of a λ-expression of $\Lambda/\overset{\alpha}{\equiv}$ is defined as:

$$\mathrm{FV}(E) = \{x \mid (\exists e)(q(e) = E \wedge x \in \mathrm{FV}(e))\}$$

where E is in the quotient space $\Lambda/\overset{\alpha}{\equiv}$. However $\overset{\alpha}{\equiv}$ preserves FV, and therefore:

$$q(e) = E \Rightarrow \mathrm{FV}(e) = \mathrm{FV}(E).$$

From now on e, e_i, ... will range over the newly formed quotient space. Working with the quotient space allows us to forget about $\overset{\alpha}{\equiv}$, which makes definitions and lemmas a lot more intuitive, and also makes proofs easier. The one difficulty which arises from the quotient space is that λ-abstractions now have infinitely many representations. We prove, however, the following important property about these representations:

$$y \notin \mathrm{FV}(e_0) \Rightarrow \lambda x.e_0 = \lambda y.e_1 \iff e_1 = e_0[x := y].$$

4.4 β and δ

The β and δ relations are defined on the quotient space, and are *not* inductive. The only thing of note about the β relation is that it only allows β reduction on values. The δ relation reduces primitive functions to values, and requires

$$(e_0 \mathrm{R}^* e_1 \wedge e_0 \neq e_1) \Rightarrow (\exists e)(e_0 \mathrm{R}\, e \wedge e \mathrm{R}^* e_1 \wedge rk(e, e_1) \leq rk(e_0, e_1))$$

Fig. 3. Rank property for R^*

each argument to be reduced to an atom before evaluation. This relation is also parametric in a specific δ function. A predicate for a valid δ function is also defined which requires the function only to evaluate primitive functions with the correct number of arguments. In addition, certain combinations of arguments may not produce a valid result for a certain primitive operation. For example, one would assume that dividing by zero would fail to reduce under a reasonable δ function.

4.5 Closures

In defining β and δ reduction we develop the notion of the compatible closure of a relation. This is defined as being the minimal superset of the relation that is compatible with the structure of λ-expressions. In the case of β and δ, the compatible closure allows reduction of subexpressions.

$$e_0 \mathrm{R}\, e_1 \Rightarrow \lambda x.e_0 \mathrm{R}\, \lambda x.e_1 \wedge e(e_0) \mathrm{R}\, e(e_1) \wedge e_0(e) \mathrm{R}\, e_1(e) \wedge \\ o(\ldots, e_0, \ldots) \mathrm{R}\, o(\ldots, e_1, \ldots)$$

We use a different definition to [24] for the transitive reflexive closure of a relation.

$$e \mathrm{R}^* e \quad \text{and} \quad e_0 \mathrm{R}\, e_1 \wedge e_1 \mathrm{R}^* e_2 \Rightarrow e_0 \mathrm{R}\, e_2$$

To induct on the definition of the transitive reflexive closure, we define a rank. The difficulty here is that there may be more than one way to *prove* that a pair lies in the transitive reflexive closure. A *path* between two expressions e and e_0 is a list whose first and last elements are e and e_0 respectively, and with the property that every pair of consecutive elements are in the relation R. We define a predicate $rk?(e, e_0, k)$ to be true if there is a path of length $k + 1$ between e and e_0. So, for example e_0 R e_1 implies that $rk?(e_0, e_1, 1)$ is true.

$$rk?(e, e, 0)$$
$$rk?(e_0, e_1, k) \wedge e \mathrm{R}\, e_0 \Rightarrow rk?(e, e_1, k + 1)$$

We then take the rank $rk(e, e_0)$ of two expressions to be the minimum of all such k for which $rk?(e, e_0, k)$ holds, or 0 if $\neg(e \mathrm{R}^* e_0)$. This rank gives the important result shown in figure 3, which is an integral part of all inductive proofs on the transitive, reflexive closure of a relation.

4.6 Parallel

The definition of $\xmapsto{p}$ is inductive with the only difficulty coming from the β-reduction and λ-abstraction cases. As in the case for $\overset{\alpha}{\equiv}$, we have at least two ways to define the relation. In this case we can either choose an individual representation for each λ-abstraction, or consider each possible representation. Our $\xmapsto{p}$ uses the latter approach, although it is likely that there is little difference between the two. In fact, as soon as it is established that renaming preserves $\xmapsto{p}$, the initial representation becomes irrelevant. To illustrate our handling of λ-abstractions we provide the formal definition for the β-reduction case:

$$\begin{aligned}&(\forall x \notin \mathrm{FV}(e))(\exists e_0, e_1, v_0, v_1)(\\ &\qquad (e = (\lambda x.e_0)(v_0) \wedge e' = e_1[x := v_1] \wedge e_0 \xmapsto{p} e_1 \wedge v_0 \xmapsto{p} v_1)) \Rightarrow e \xmapsto{p} e'\end{aligned}$$

Before we give a detailed analysis of the proof of $\diamond(\xmapsto{p})$ let us outline our motivations for forming the quotient space. There are many possible $\xmapsto{p}$ relations over the original (non quotient) Λ. The difficulty in defining such a relation is how to incorporate $\overset{\alpha}{\equiv}$ into it. We consider the two approaches that we attempted. Our first approach involves replacing equality, $=$, the P_{refl} axiom of figure 2 of section 2 by $\overset{\alpha}{\equiv}$ (i.e: $e \overset{\alpha}{\equiv} e_0 \Rightarrow e \xmapsto{p} e_0$). We had difficulty proving that this gave us the correct transitive reflexive closure. We also could not establish the diamond lemma for the β-reduction and λ-abstraction cases.

The other approach we consider is defining $\xmapsto{p}$ without $\overset{\alpha}{\equiv}$, and then defining another relation, say $\xmapsto{p\alpha}$, by:

$$(\exists e_0, e_1)(e \overset{\alpha}{\equiv} e_0 \wedge e' \overset{\alpha}{\equiv} e_1 \wedge e_0 \xmapsto{p} e_1) \Rightarrow e \xmapsto{p\alpha} e'$$

Unfortunately this too leads to problems in proving $\xmapsto{p}$ for applications, λ-abstractions and δ-reduction.

Ideally we would like to add $\overset{\alpha}{\equiv}$ statements into each of the six cases, so for example, the non-β application case would look something like (where e and e' are applications):

$$(\exists e_0, e_1, e_2, e_3)(e \overset{\alpha}{\equiv} e_0(e_1) \wedge e' \overset{\alpha}{\equiv} e_2(e_3) \wedge e_0 \xmapsto{p} e_2 \wedge e_1 \xmapsto{p} e_3) \Rightarrow e \xmapsto{p} e'$$

Defining $\xmapsto{p}$ like this is a messy process and subsequently proving anything about it is likely to be difficult. It is clear that some mechanism is desirable for removing $\overset{\alpha}{\equiv}$ from our definitions and lemmas, so it can be ignored except where required. We feel that the most intuitive and elegant method is to form the quotient space modulo $\overset{\alpha}{\equiv}$.

4.7 Proof

Before we prove $\diamond(\xmapsto{p})$ we need to establish an important property of $\xmapsto{p}$ that is required for the β-reduction case, namely that $\xmapsto{p}$ is preserved by substitution (see section 2.1). To see where it is used, consider the following case in the $\diamond(\overset{p}{\longmapsto})$ proof. Suppose that $e \overset{p}{\longmapsto} e_1$, $e \overset{p}{\longmapsto} e_2$, $v \overset{p}{\longmapsto} v_1$ and $v \overset{p}{\longmapsto} v_2$. Then

$$\begin{tikzcd}
 & (\lambda x.e)v \arrow[dl] \arrow[dr] & \\
e_1[x := v_1] & & e_2[x := v_2]
\end{tikzcd}$$

Now by the induction hypothesis we can find an e' and a v' so that $e_i \overset{p}{\longmapsto} e'$ and $v_i \overset{p}{\longmapsto} v'$ for $i \in \{1, 2\}$. Thus we can complete the diagram:

$$\begin{tikzcd}
 & (\lambda x.e)v \arrow[dl] \arrow[dr] & \\
e_1[x := v_1] \arrow[dr] & & e_2[x := v_2] \arrow[dl] \\
 & e'[x := v'] &
\end{tikzcd}$$

The λ-abstraction case requires only that $\overset{p}{\longmapsto}$ is preserved by *renaming.* Suppose $e \overset{p}{\longmapsto} e_i$ for $i \in \{1, 2\}$. Then

$$\begin{tikzcd}
 & (\lambda x.e) \arrow[dl] \arrow[dr] & \\
\lambda x.e_1 & & \lambda x.e_2
\end{tikzcd}$$

Thus by the induction hypothesis, there exists an e' such that $e_i \overset{p}{\longmapsto} e'$ for $i \in \{1, 2\}$. Now for any $y \notin \mathrm{FV}(e_1)$ we can complete the diagram using $\lambda y.e'[x := y]$, since:

$$\begin{tikzcd}
 & \lambda x.e \arrow[dl] \arrow[dr] & \\
\lambda y.e_1[x := y] = \lambda x.e_1 \arrow[dr] & & \lambda x.e_2 = \lambda y.e_2[x := y] \arrow[dl] \\
 & \lambda y.e'[x := y] &
\end{tikzcd}$$

The primitive operation case causes another problem as there are two ways for such an expression to reduce, namely by δ-reduction or by reduction on each of its arguments. Fortunately δ-reduction can only be performed if the arguments are all atoms. Furthermore, it is not hard to prove that under $\overset{p}{\longmapsto}$, atoms can only reduce to themselves. Suppose that $o(\bar{a}) \overset{p}{\longmapsto} v_0$ and $o(\bar{a}) \overset{p}{\longmapsto} o(\bar{b})$ where $a_i \overset{p}{\longmapsto} b_i$ for $1 \leq i \leq n$. However $a_i \in A$ implies that $a_i = b_i$. Thus $o(\bar{b}) \overset{p}{\longmapsto} v_0$.

All in all, proving the diamond lemma for $\overset{p}{\longmapsto}$ is the hardest step in our proof. The lemma is split up into different sub-lemmas (one for each case), to make editing, and revising the proofs easier.

Proving that $\xmapsto{p}{}^* = \xmapsto{\beta_v}{}^*$ is a relatively simple process in comparison. The proof consists of a number of separate lemmas which are given below:

(PBs) $e \xmapsto{p} e_0 \Rightarrow e \xmapsto{\beta_v}{}^* e_0$

(PsBs) $e \xmapsto{p}{}^* e_0 \Rightarrow e \xmapsto{\beta_v}{}^* e_0$

(BPs) $e \xmapsto{\beta_v} e_0 \Rightarrow e \xmapsto{p}{}^* e_0$

(BsPs) $e \xmapsto{\beta_v}{}^* e_0 \Rightarrow e \xmapsto{p}{}^* e_0$

The lemmas **(PBs)** and **(BPs)** are proved by induction on the relevant inductive definition (parallel reduction and compatible closure respectively). In contrast, the proofs of **(PsBs)** and **(BsPs)** are carried out by induction on the rank of the transitive reflexive closure.

5 Previous, Current and Subsequent Work

We finish of this paper with a discussion of: previous related work, the conclusions drawn from the work presented here, and finally our work with PVS subsequent to the work reported here.

5.1 Previous Work

Presumably because of its importance to the foundations of (Theoretical) Computer Science, the Church–Rosser theorem has been the subject of several machine based theorem proving studies [26,8,25,22,17,20].

The earliest treatment was by Shankar using the Boyer-Moore theorem prover [26], and later appears as a chapter in his PhD thesis [27]. The formalization of the λ-calculus uses de Bruijn indices, and the proof is the standard Tait–Martin-Löf version. One notable point about this proof is that it is carried out in a very weak logic, one that has no explicit quantifiers.

The next treatment was Huet's formal development of the theory of residuals in the λ-calculus using the Coq system [8]. He uses de Bruijn indices in his formalization and establishes Church–Rosser as a corollary to his treatment of residuals.

Rasmussen [25] ports Huet's treatment to Isabelle. The emphasis of his treatment is on the difficulties involved in translating one mechanical proof on one platform to another mechanical proof on another platform.

Nipkow [20] presents a very general and abstract treatment of Tait–Martin-Löf style proofs of Church–Rosser in Isabelle. His treatment is based on a general theory of commutating relations, and covers both β and η reduction systems. He also encodes and compares both the original proof that parallel reduction has the diamond property, as well as the more recent one due to Takahashi [29].

Pfenning in [22] presents a development of the Tait–Martin-Löf proof in his Elf implementation of the Edinburgh LF [6]. The novel aspect of this treatment is that it uses higher order abstract syntax to encode the lambda calculus. This encoding does not have a syntactic category for variables of the (object) λ calculus, but rather uses the variables of the LF framework. For example the λ constructor is modeled by a constant in the framework of the form $\lambda : (\Lambda \mapsto \Lambda) \mapsto \Lambda$, where Λ is the syntactic category corresponding to (object) λ expression.

5.2 Conclusions Concerning the Work Reported Here

Our work differs from the previous work reported above in two important ways. The first and most obvious difference is that we use the PVS system, whereas the work reported above relied on older systems. Prior to the work reported here, little use had been made of the abstract datatype facility in PVS. The work reported here helped debug these facilities of PVS, and thus helped refine the system. This refinement of the PVS system is an ongoing process, for example the prover doesn't automatically apply the correct extensionality and eta axioms, so that the specific axiom needs to be explicitly stated in the prover command. However, the bottom line is that the abstract datatype mechanism is extremely useful in encoding operational approaches to semantics, as is demonstrated by our subsequent work.

The second and more important difference is that we directly formalize and reason about α equivalence. Something that has not been done previously, to our knowledge. Indeed the main conclusion of this work, and of our subsequent work as well, is that it is indeed possible to formalize α equivalence, and remain faithful to the presentations found in text books and journals.

5.3 PVS Statistics

The actual proof of Church–Rosser in PVS took the first author approximately four months, although some of this time was spent learning PVS. Some time was also wasted attempting a direct proof of Church–Rosser without first forming the quotient space. The actual machine checked proof involves the proving of two hundred and thirty six (236) distinct facts, and takes PVS three hundred and sixty seconds (362) of CPU time running on a Linux machine configured with 2 GBytes of main memory and 4×550 MHz Xeon PIII processors. The dump file containing all the PVS definitions, facts, and proofs is 2.396 MBytes and is available from `http://mcs.une.edu.au/~pvs/` [4].

5.4 Subsequent Work

After successfully carrying out this first experiment reported here. We undertook a second sophisticated and substantial use of PVS, one that established a recent result in operational semantics. This experiment was 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. Thus this experiment exemplifies the possible benefit of serious formalization offers standard mathematical practice, which typically leaves much unsaid.

Much work has been done to develop methods for reasoning about operational approximation and equivalence. An early example is Robin Milner's context lemma [18] 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. Mason and Talcott [13,14] introduced the CIU characterization of operational equivalence which is a form of context lemma for imperative languages. This lemma was then generalized by Carolyn Talcott to a very wide class of programming languages in [30]. It is this lemma that we verified in this second experiment. This lemma is fundamental to our formalization effort since it is the corner stone upon which we define the semantics of our specification logic [15]. Again we took great pains to formalize the actual theoretical treatment, rather than adapting it to the tastes of both the machine, and PVS. The results of this experiment have been briefly discussed at [12] and appear as [5].

Acknowledgements

In the course of our work we uncovered several bugs in PVS's implementation. We wish to thank explicitly Sam Owre and Shankar at SRI in Menlo Park for promptly fixing these bugs, providing timely advice, and encouragement, and thus allowing our work to reach fruition. We would also like to thank Carolyn Talcott for providing constructive criticism, as well as encouragement. The two anonymous CATS'01 referees must also be thanked for their insightful comments, that helped improve both the veracity and presentation of this paper.

References

[1] A. Church and J.B. Rosser. Some properties of conversion. *Transactions of the AMS*, Volume 39, pages 472–482, 1936.

[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] N. G. de Bruijn. Lambda-Calculus Notation with Nameless Dummies: a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem. *Indagationes Mathematicae*, Volume 34, Number 5, pages 381–392, 1972.

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

[5] J. Ford and I. A. Mason. Establishing a General Context Lemma in PVS. In *Proceedings of the 2nd Australasian Workshop on Computational Logic, AWCL'01* , 2000. submitted.

[6] R. Harper, H. Honsell and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.

[7] G. Huet. Residual Theory in λ-Calculus: A Formal Development. Technical Report 2009, INRIA, 1993.

[8] G. Huet. Residual Theory in λ-Calculus: A Formal Development. *Journal of Functional Programming*, Volume 4, Number 3, pages 371–394, 1994. An earlier version appeared as [7].

[9] P. J. Landin. A correspondence between Algol60 and Church's lambda notation. *Comm. ACM*, Volume 8, pages 89–101, 158–165, 1965.

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

[11] I. A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, Volume 12, pages 171–201, 1999. An abridged version appears as [10].

[12] I. A. Mason. A second glance at Feferman-Landin logic, 2000. Invited talk, HOOTS '00 Montreal, Canada, September 2000.

[13] 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.

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

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

[16] 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.

[17] J. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, Volume 23, 1999. An abridged version appeared as [16].

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

[19] T. Nipkow. More Church-Rosser Proofs (in Isabelle/HOL). In M. McRobbie and J.K. Slaney (editors), *Automated Deduction – CADE-13*, Volume 1104 of *Lecture Notes in Computer Science*, pages 733–747. Springer Verlag, 1996.

[20] T. Nipkow. More Church-Rosser Proofs (in Isabelle/HOL), 2000. to appear in *Journal of Automated Reasoning.* An abridged version appears as [19].

[21] F. Pfenning. A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework. Technical Report CMU-CS-92-186, Carnegie Mellon University, 1992.

[22] F. Pfenning. A Proof of the Church-Rosser Theorem and its Representation in a Logical Framework. *Journal of Automated Reasoning (to appear)*, 2000. An earlier version appears as [21].

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

[24] R Pollack. Polishing up the Tait-Martin-Löf Proof of the Church–Rosser Theorem. Unpublished note, available from `ftp://ftp.cs.chalmers.se/pub/users/pollack/churchrosser.dvi.gz`, 1995.

[25] O. Rasmussen. The Church–Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, 1995.

[26] N. Shankar. A Mechanical Proof of the Church-Rosser Theorem. *Journal of the Association for Computing Machinery*, Volume 35, Number 3, pages 475–522, 1988.

[27] N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge University Press, 1994.

[28] M. Sorensen and P. Urzyczyn. Lectures on the Curry-Howard Isomorphism. Technical Report 14, DIKU Report, 1998.

[29] M. Takahashi. Parallel reductions in the λ-calculus. *Information and Computation*, Volume 118, pages 120–127, 1995.

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