

FORMAL VERIFICATION AND OPERATIONAL SEMANTICS

By

Jonathan Ford

`jford@turing.une.edu.au`

A thesis submitted for the degree of
Bachelor of Computing Science (Honours)
of The University of New England
May 2001

DECLARATION

I certify that the substance of this thesis has not already been submitted for any degree and is not currently being submitted for any other degree.

I certify that to the best of my knowledge, any help received in preparing this thesis, and all sources used, have been acknowledged in this thesis.

.....

Acknowledgements

I would like to thank my honours supervisor, Ian Mason, for helping me with the theoretical background of my work and for being a dedicated and patient advisor.

I would also like to thank my family, in particular my mother and father, for providing emotional support for me throughout the year without which my work could never have reached fruition.

I would also like to thank my friends for all of the computer gaming we shared together in the wee hours of the morning when I was finding respite from my studies.

Abstract

In this thesis, two theorems of operational semantics are specified and proved in the PVS theorem prover. The first theorem is the Church–Rosser theorem for an untyped, call-by-value λ -calculus with primitive operations. The novel aspect of the proof is that α -equivalence is specifically defined without resorting to the use of a *tricky* representation such as De Bruijn indices. The proof also takes advantage of the little used PVS abstract data type (ADT) mechanism.

This feature of PVS allows the specification of complex inductive structures, and is particularly suited to syntactical definitions. In addition to providing a quick and easy method for such definitions, PVS generates many of the required definitions and axioms automatically such as rank definitions and induction schemes.

The second theorem is the CIU theorem for uniform λ -languages and as such represents the first use of PVS for proving a recent result. The process of specifying and proving the CIU theorem uncovered several gaps in existing theory. As with the Church–Rosser proof, the PVS ADT mechanism is used for the specification of the syntax of λ -languages.

Contents

| | |
|---|------------|
| Acknowledgements | iii |
| Abstract | iv |
| 1 Introduction | 1 |
| 2 An Overview of PVS | 3 |
| 3 The Church–Rosser Theorem | 6 |
| 3.1 Introduction | 6 |
| 3.2 A Whirlwind Tour of the Church–Rosser Theorem | 7 |
| 3.2.1 A summary of the Encoding & Proof | 10 |
| 3.3 A Tour of the Encoding of the λ -calculus & the Church–Rosser Proof | 12 |
| 3.3.1 Syntax | 12 |
| 3.3.2 Alpha | 14 |
| 3.3.3 Quotient | 15 |
| 3.3.4 β and δ | 15 |
| 3.3.5 Closures | 16 |
| 3.3.6 Parallel | 16 |
| 3.3.7 Proof | 17 |
| 3.4 Church–Rosser Conclusions | 19 |
| 3.4.1 Previous Work | 19 |
| 3.4.2 Conclusions concerning the Work Reported Here | 20 |
| 3.4.3 PVS Statistics | 20 |
| 3.4.4 Acknowledgements | 21 |

| | | |
|----------|---|-----------|
| 4 | The CIU Theorem | 22 |
| 4.1 | Introduction | 22 |
| 4.1.1 | Historical Background | 23 |
| 4.2 | Syntax of Terms | 24 |
| 4.2.1 | Background | 24 |
| 4.2.2 | The Syntax of Expressions | 25 |
| 4.3 | Semantics of Terms | 34 |
| 4.3.1 | Operational Semantics | 34 |
| 4.3.2 | Uniform Semantics | 37 |
| 4.3.3 | Approximation and Equivalence | 39 |
| 4.4 | The Proof of CIU | 39 |
| 4.4.1 | The CIU Theorem | 39 |
| 4.4.2 | The CIU Proof | 40 |
| 4.5 | CIU Conclusions | 45 |
| 4.5.1 | PVS Statistics | 45 |
| 4.5.2 | Acknowledgments | 46 |
| 5 | Conclusions | 47 |

List of Figures

| | | |
|-----|--|----|
| 3.1 | Single Step β -value-reduction | 9 |
| 3.2 | Parallel β -value-reduction | 10 |
| 3.3 | Rank property for \mathbb{R}^* | 16 |

Chapter 1

Introduction

This thesis presents the first use of PVS as a tool for specification and verification of results in operational semantics. Such semi-automated methods for theorem proving are of importance because they require all details to be addressed and thus produce an extremely rigorous result. This is in contrast with standard mathematical practice which typically leaves much unsaid.

A description of PVS can be found in chapter 2. This includes an overview of PVS specifications, an analysis of the typing system including the generation of type correctness conditions (TCCs) and a sketch of PVS proofs including the interface between the user and the prover, and the internal representation of a proof. Of particular note is the use by PVS of judgements to help reduce the number of TCCs generated. This chapter also gives an account of the PVS abstract data type (ADT) facility for the definition of inductive structures. Each ADT definition also produces an entire theory containing axioms and definitions for the data type which are essential when dealing with such structures. Little prior use of this feature could be found.

The details of the Church–Rosser proof are given in chapter 3. In particular, the ways in which the proof differs from previous computer-based proofs are emphasised, and any resulting difficulties described. This chapter provides a background introduction to the λ -calculus and the Church–Rosser theorem itself including an outline of the proof. The actual proof is based on the Tait–Martin-Löf notion of parallel reduction. To give an overview of the original aspects of the proof:

- we use the PVS system, especially its built-in ADT facility, to verify a version of the Church–Rosser theorem;
- we use the named-variable approach and thus avoid the use of *tricky* representations of variables;

- we explicitly define α -equivalence, and thus must prove a number of properties of this relation;
- we treat the more complex call-by-value version of the λ -calculus as opposed to the call-by-name version used in previous proofs.

The main aim of this chapter however is as a preliminary evaluation of PVS as a tool for working with operational logics for realistic programming languages.

Chapter 4 covers the proof of a recent result in operational semantics, namely the CIU theorem for uniform λ -languages. Thus it represents the first use of PVS and specifically its abstract data type facility for the verification of a non-trivial theorem. This chapter has several purposes:

- it gives a presentation of the syntax of uniform λ -languages as well as the definitions of operational approximation and equivalence;
- it gives a detailed analysis of the proof of the CIU theorem itself;
- it provides an account of the actual mechanized proof;
- it highlights gaps found in existing theory and thus provides evidence for the possible benefit of the use of such rigorous techniques for the testing of current theory.

As with the Church–Rosser proof, great effort is taken to model the exact syntax and structure of the language without tailoring the specification to a representation which is easier for PVS to understand. As such, no *tricky* representations of variables are used, and as with our earlier work, α congruence is explicitly treated.

Chapter 2

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 type checking is undecidable. As a consequence the type checker 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 type checker. 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. Each judgement generates one TCC which must be proved in the same way as other TCCs. However, a TCC which can be immediately proved by a previous judgement or judgements is not generated. This also applies to TCCs which are generated whilst proving a lemma. While judgements allow a wide range of typing properties to be specified, they are in no way as diverse as ordinary lemmas. In addition to this, judgements are only automatically used to suppress the production of TCCs, so for all other applications they must be explicitly called. 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, orderings relations, and abstract data types, including a list definition.

The prelude also contains a number of judgements concerning such things as sets both finite and otherwise, functional classifications (eg. all bijective functions are surjective), and numbers. These judgements can be used together to prove more complicated type facts. For example, there are judgements stating that the set of positive rational numbers is a subset of the set of positive real numbers and the product of two positive rational numbers is a positive rational number. So the type checker can conclude that the product of two positive rational numbers is also a positive real number.

A proof in PVS is viewed and edited via a sequent-style representation, consisting of antecedent and consequent formulas. Logically, this is equivalent to stating that the conjunction of the antecedents implies the disjunction of the consequents. Certain operations can split the current proof into two or more subproofs, so the overall internal structure of a proof is a tree. A proof tree is considered true, if all of its leaves are trivially true. For the purpose of identification, each sequent formula is assigned a unique number automatically.

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. The highest level strategy, *grind*, is useful in a wide number of situations, and can frequently finish off a proof automatically. 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.

Most commands accept arguments and keywords to define or alter their effect. Arguments include such things as sequent numbers, lemma names and expressions. Keywords may be used to specify the type of action to take, or alter the action in some way. For example, when calling the *grind* strategy, a list of lemmas to use can be specified via the *rewrites* and *theories* keywords. As mentioned, each sequent is uniquely numbered, meaning that for some commands to work properly, a sequent number must be supplied. However, modifying the specification may cause these sequent numbers to change (for example, an alteration may add an antecedent formula to a proof, which will cause some or all of the current antecedent formulae to be numbered differently).

This can be disastrous for a proof because some of the commands may now be working on different sequent formulae. PVS allows sequent numbers to be specified in a more general fashion (eg $'-'$ means any antecedent formula), which helps alleviate the problem, but does not work in all cases.

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.

Chapter 3

The Church–Rosser Theorem

3.1 Introduction

In this chapter we present a formalized proof [8] of the Church–Rosser theorem for a version of the call-by-value lambda calculus [33] in the spirit of Landin’s ISWIM [18]. The proof is developed in the PVS [4] 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 contexts [20], developing Feferman-Landin logic [24] or proving the Curry-Howard isomorphism theorem for various versions of typed lambda calculi, and the corresponding logics [38].

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 [36, 16, 35, 32, 26, 30] (see section 3.4 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 [5], or by using the variables of the logical framework itself [11], rather than incorporating one into the encoded system.

Only the treatment by McKinna and Pollack [26] 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 [26] 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 `consing` 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.

3.2 A Whirlwind Tour of the Church–Rosser Theorem

Following in the footsteps of [34] 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) [3]. 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 [18] (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 possess 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:

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

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 ($FV(e)$), 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 is usual to define a notion of α -equivalence, \equiv^α , and either remark that λ -expressions are now only distinguished up to \equiv^α , or much less frequently form the quotient Λ / \equiv^α . The latter of course requires first establishing that \equiv^α 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 \equiv^α of λ -abstractions, $\lambda x_0. e_0 \equiv^\alpha \lambda x_1. e_1$, reduces to showing $e_0[x_0 := y] \equiv^\alpha e_1[x_1 := y]$ 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. $y \notin FV(e_0) \cup FV(e_1)$), or we can require that it is true for all such y . Even though these two 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, $\mapsto^{\beta v}$, is parametric in a δ function, $\delta : O(A^n) \mapsto V$, and is generated by the rules in figure 3.1. As defined $\mapsto^{\beta v}$ 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, $\diamond(R)$, iff

$$(\forall e_0, e_1, e_2)(e_0 R e_1 \wedge e_0 R e_2 \Rightarrow (\exists e_3)(e_1 R e_3 \wedge e_2 R e_3))$$

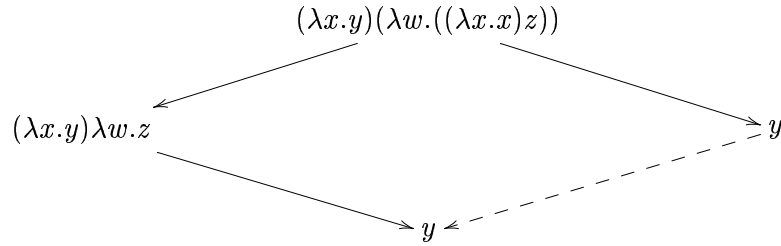
A relation is said to be *Church–Rosser* or *confluent* iff $\diamond(R^*)$. The Church–Rosser theorem states that $\diamond(\mapsto^{\beta v})$. Since $\mapsto^{\beta v}$ is neither reflexive nor transitive it is not the case that $\diamond(\mapsto^{\beta v})$. To see

$$\begin{array}{l}
(\beta_v) \quad (\lambda x.e)v \xrightarrow{\beta_v} e[x := v] \quad \text{for } v \text{ a value.} \\
(\delta) \quad o(e_1, \dots, e_n) \xrightarrow{\beta_v} v \quad \text{if } e_1, \dots, e_n \in \text{Dom}(\delta) \text{ and } \delta(o, e_1, \dots, e_n) = v. \\
(C_\lambda) \quad \frac{e_0 \xrightarrow{\beta_v} e_1}{\lambda x.e_0 \xrightarrow{\beta_v} \lambda x.e_1} \\
(C_{\text{left}}) \quad \frac{e_0 \xrightarrow{\beta_v} e_1}{e(e_0) \xrightarrow{\beta_v} e(e_1)} \quad (C_{\text{right}}) \quad \frac{e_0 \xrightarrow{\beta_v} e_1}{e_0(e) \xrightarrow{\beta_v} e_1(e)} \\
(C_{\delta i}) \quad \frac{e \xrightarrow{\beta_v} e'}{o(e_1, \dots, e_i, e, e_{i+2}, \dots, e_n) \xrightarrow{\beta_v} o(e_1, \dots, e_i, e', e_{i+2}, \dots, e_n)} \quad \text{for } 0 \leq i \leq n.
\end{array}$$

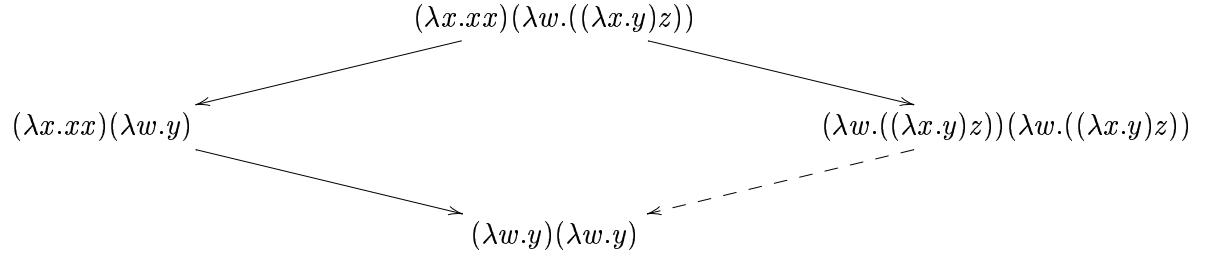
Figure 3.1: Single Step β -value-reduction

this consider the following two counterexamples. In each case the dotted arrow does not belong to $\xrightarrow{\beta_v}$.

Reflexivity:



Transitivity:



The Tait–Martin-Löf proof of this theorem involves defining a parallel reduction relation, \xrightarrow{p} , 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 3.2 for the complete definition. The proof then follows from establishing three facts:

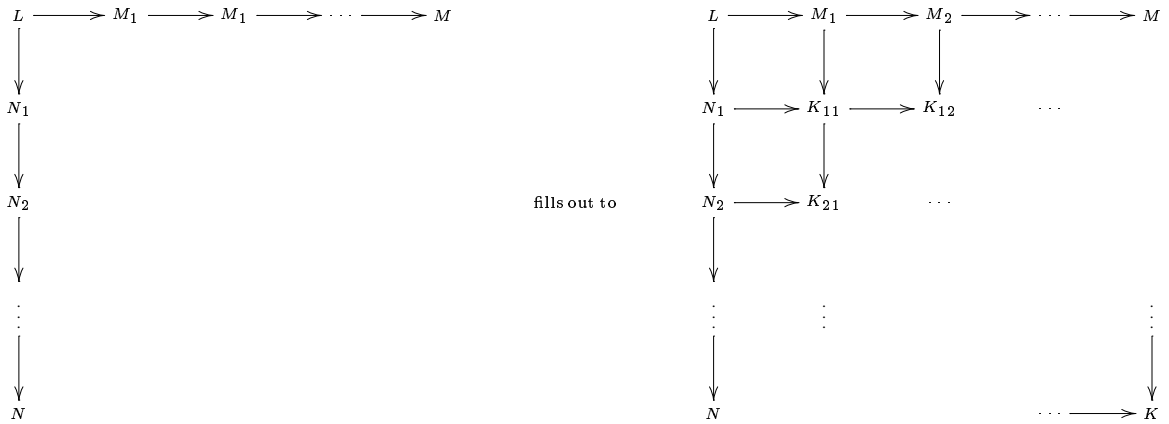
1. $\diamond(\xrightarrow{p})$ holds. This is the delicate part of the proof. Our version of the proof is a structural induction on the proof that $e_0 \xrightarrow{p} e_1$ and proceeds by case analysis on the last step in this proof. This is not the only method of proof, for example Takahashi [39] has recently published a proof that does not analyse the reductions, but rather relies on taking the *maximum*

$$\begin{array}{l}
(\text{P}_{\text{refl}}) \quad e \xrightarrow{p} e \\
(\text{P}_{\beta v}) \quad \frac{e \xrightarrow{p} e' \quad v \xrightarrow{p} v'}{(\lambda x.e)v \xrightarrow{p} e'[x := v]} \quad \text{for } v \text{ a value.} \\
(\delta) \quad o(e_1, \dots, e_n) \xrightarrow{p} v \quad \text{if } e_1, \dots, e_n \in \text{Dom}(\delta) \text{ and } \delta(o, e_1, \dots, e_n) = v. \\
(\text{P}_\lambda) \quad \frac{e_0 \xrightarrow{p} e_1}{\lambda x.e_0 \xrightarrow{p} \lambda x.e_1} \\
(\text{P}_{\text{app}}) \quad \frac{e_0 \xrightarrow{p} e_1 \quad e_2 \xrightarrow{p} e_3}{e_0(e_2) \xrightarrow{p} e_1(e_3)} \quad (\text{P}_\delta) \quad \frac{e_i \xrightarrow{p} e'_i \quad \text{for } 0 \leq i \leq n.}{o(e_1, \dots, e_n) \xrightarrow{p} o(e'_1, \dots, e'_n)}
\end{array}$$

Figure 3.2: Parallel β -value-reduction

parallel reduction step (called the *complete development*). We do not follow this version of the proof.

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



3. \xrightarrow{p}^* is the same relation as $\xrightarrow{\beta v}^*$. Pollack points out [34] 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.

3.2.1 A summary of the Encoding & Proof

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

- **Syntax:**

Define the syntax of the λ -expressions, and related notions of expression rank, free and bound variables, renaming, and substitution.

- **Alpha:**

Define \equiv^α and establish that it is an equivalence (congruence) relation, and that renaming, and substitution are functional modulo \equiv^α . 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 \equiv^α .

- **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:

$$e \mapsto^p e' \Rightarrow (e \in V \Rightarrow e' \in V)$$

$$(e_0 \mapsto^p e_1 \wedge v_0 \mapsto^p v_1) \Rightarrow e_0[x := v_0] \mapsto^p e_1[x := v_1]$$

- **Proof:**

The proof now consists of the three steps described above.

– $\diamond(\mapsto^p)$ holds

– $\diamond(\mapsto^p)$ implies that $\diamond(\mapsto^{p*})$

– \mapsto^{p*} is the same relation as $\mapsto^{\beta v*}$

3.3 A Tour of the Encoding of the λ -calculus & the Church–Rosser Proof

3.3.1 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 $(\forall y \in \text{Fin}(X)) \text{new}(y) \notin y$.

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

```
 $\Lambda[A: \text{TYPE}+, O: \text{TYPE}+, \# : [O \rightarrow \text{nat}]]: \text{DATATYPE}$ 
BEGIN
  IMPORTING  $X$ 
   $\text{Var}(x: X): \text{Var?}$ 
   $\lambda(x: X, e: \Lambda): \lambda?$ 
   $\text{app}(e: \Lambda, e': \Lambda): \text{app?}$ 
   $\text{K}(a: A): \text{K?}$ 
   $\delta(o: O, l: \text{list}[\Lambda]): \delta?$ 
END  $\Lambda$ 
```

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 chapter 2). 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.

$$rk(e) = 1 \quad e \in (X \cup A)$$

$$rk(\lambda x.e) = 1 + rk(e)$$

$$rk(e(e_0)) = 1 + rk(e) + rk(e_0)$$

$$rk(o(e_1, e_2, \dots, e_n)) = 1 + rk(e_1) + rk(e_2) + \dots + rk(e_n)$$

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

$$\text{FV}(x) = \{x\}$$

$$\text{FV}(a) = \{\}$$

$$\text{FV}(\lambda x.e) = \text{FV}(e) - \{x\}$$

$$\text{FV}(e(e_0)) = \text{FV}(e) \cup \text{FV}(e_0)$$

$$\text{FV}(o(e_1, e_2, \dots, e_n)) = \text{FV}(e_1) \cup \text{FV}(e_2) \cup \dots \cup \text{FV}(e_n)$$

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 $\text{FV}(e)$ is defined as having type *setof*(X), so taking the *new* of $\text{FV}(e)$ will lead to the generation of a TCC. Including a judgement, however stating that $\text{FV}(e) \in \text{Fin}(X)$ in our specification suppresses the production of such TCCs. The prelude contains judgements about finite sets unions, intersection and so forth (e.g. $(\forall X, Y \in \text{Fin}(T))(X \cup Y) \in \text{Fin}(T)$). Thus, even expressions of the form $\text{new}(\text{FV}(e) \cup \text{FV}(e_0))$ 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

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

It is in defining the renaming function that we first run into trouble with TCCs. As mentioned in chapter 2, 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:

$$e[y := z] : \text{Recursive } \Lambda =$$

Cases e of :

...

$$\lambda x.e_0 : \text{let } x_0 = \text{new}(\text{FV}(e) \cup \{y, z\}) \text{ in}$$

$$\lambda x_0.(e_0[x := x_0][y := z])$$

...

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 renaming function. In particular we express that the rank of its value is no greater than the rank of its argument:

$$e[y := z] : \text{Recursive} \{e_0 \in \Lambda \mid rk(e_0) \leq rk(e)\} = \dots$$

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

3.3.2 Alpha

We formally define $\overset{\alpha}{\equiv}$ using an inductive definition. As mentioned in section 3.2, 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 $e_0[x_0 := y] \overset{\alpha}{\equiv} e_1[x_1 := y]$. This gives us the greatest control over the new variable, and hence the greatest ease at proving theorems.

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

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 [20]:

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

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

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

3.3.3 Quotient

In defining the quotient space modulo α equivalence, $\stackrel{\alpha}{\equiv}$, 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 $\Lambda / \stackrel{\alpha}{\equiv}$ is defined as:

$$\text{FV}(E) = \{x \mid (\exists e)(q(e) = E \wedge x \in \text{FV}(e))\} \quad \text{where } E \text{ is in the quotient space } \Lambda / \stackrel{\alpha}{\equiv}.$$

However $\stackrel{\alpha}{\equiv}$ preserves FV, and therefore:

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

From now on e, e_i, \dots will range over the newly formed quotient space. Working with the quotient space allows us to forget about $\stackrel{\alpha}{\equiv}$, 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:

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

3.3.4 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 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.

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

Figure 3.3: Rank property for R^*

3.3.5 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.

$$e_0 R e_1 \Rightarrow \lambda x.e_0 R \lambda x.e_1 \wedge e(e_0) R e(e_1) \wedge e_0(e) R e_1(e) \wedge o(\dots, e_0, \dots) R o(\dots, e_1, \dots)$$

We use a different definition to [34] for the transitive reflexive closure of a relation.

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

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.

$$rk?(e, e, 0)$$

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

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 $\neg(e R^* e_0)$. This rank gives the important result shown in figure 3.3, which is an integral part of all inductive proofs on the transitive, reflexive closure of a relation.

3.3.6 Parallel

The definition of \mapsto^p is inductive with the only difficulty coming from the β -reduction and λ -abstraction cases. As in the case for $\stackrel{\alpha}{\equiv}$, 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 \mapsto^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 \mapsto^p , the

initial representation becomes irrelevant. To illustrate our handling of λ -abstractions we provide the formal definition for the β -reduction case:

$$(\forall x \notin \text{FV}(e))(\exists e_0, e_1, v_0, v_1)(e = (\lambda x.e_0)(v_0) \wedge e' = e_1[x := v_1] \wedge e_0 \mapsto^p e_1 \wedge v_0 \mapsto^p v_1) \Rightarrow e \mapsto^p e'$$

Before we give a detailed analysis of the proof of $\diamond(\mapsto^p)$ let us outline our motivations for forming the quotient space. There are many possible \mapsto^p relations over the original (non quotient) Λ . The difficulty in defining such a relation is how to incorporate $\stackrel{\alpha}{\equiv}$ into it. We consider the two approaches that we attempted. Our first approach involves replacing equality, $=$, the P_{refl} axiom of figure 3.2 of section 3.2 by $\stackrel{\alpha}{\equiv}$ (i.e: $e \stackrel{\alpha}{\equiv} e_0 \Rightarrow e \mapsto^p e_0$). 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 \mapsto^p without $\stackrel{\alpha}{\equiv}$, and then defining another relation, say $\mapsto^{p\alpha}$, by:

$$(\exists e_0, e_1)(e \stackrel{\alpha}{\equiv} e_0 \wedge e' \stackrel{\alpha}{\equiv} e_1 \wedge e_0 \mapsto^p e_1) \Rightarrow e \mapsto^{p\alpha} e'$$

Unfortunately this too leads to problems in proving \mapsto^p for applications, λ -abstractions and δ -reduction.

Ideally we would like to add $\stackrel{\alpha}{\equiv}$ 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):

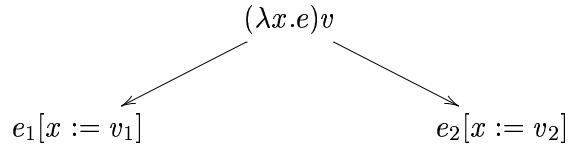
$$(\exists e_0, e_1, e_2, e_3)(e \stackrel{\alpha}{\equiv} e_0(e_1) \wedge e' \stackrel{\alpha}{\equiv} e_2(e_3) \wedge e_0 \mapsto^p e_2 \wedge e_1 \mapsto^p e_3) \Rightarrow e \mapsto^p e'$$

Defining \mapsto^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 $\stackrel{\alpha}{\equiv}$ 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 $\stackrel{\alpha}{\equiv}$.

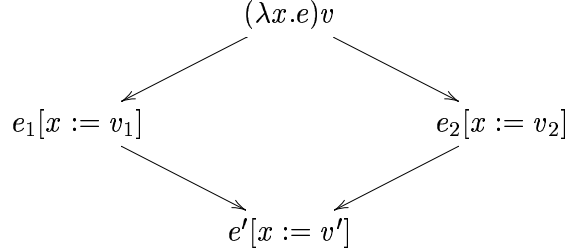
3.3.7 Proof

Before we prove $\diamond(\mapsto^p)$ we need to establish an important property of \mapsto^p that is required for the β -reduction case, namely that \mapsto^p is preserved by substitution (see section 3.3). To see where it is used, consider the following case in the $\diamond(\mapsto^p)$ proof. Suppose that $e \mapsto^p e_1$, $e \mapsto^p e_2$,

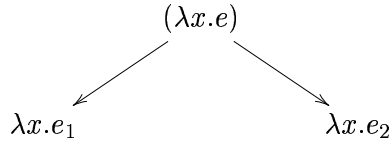
$v \xrightarrow{p} v_1$ and $v \xrightarrow{p} v_2$. Then



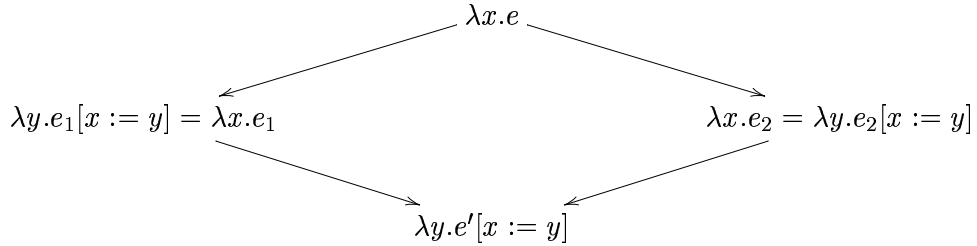
Now by the induction hypothesis we can find an e' and a v' so that $e_i \xrightarrow{p} e'$ and $v_i \xrightarrow{p} v'$ for $i \in \{1, 2\}$. Thus we can complete the diagram:



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



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



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 \xrightarrow{p} , atoms can only reduce to themselves. Suppose that $o(\bar{a}) \xrightarrow{p} v_0$ and $o(\bar{a}) \xrightarrow{p} o(\bar{b})$ where $a_i \xrightarrow{p} b_i$ for $1 \leq i \leq n$. However $a_i \in A$ implies that $a_i = b_i$. Thus $o(\bar{b}) \xrightarrow{p} v_0$.

All in all, proving the diamond lemma for \xrightarrow{p} 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 $\vdash^p \rightarrow^* = \vdash^{\beta v} \rightarrow^*$ is a relatively simple process in comparison. The proof consists of a number of separate lemmas which are given below:

$$\text{(PBs)} \quad e \vdash^p \rightarrow e_0 \Rightarrow e \vdash^{\beta v} \rightarrow^* e_0$$

$$\text{(PsBs)} \quad e \vdash^p \rightarrow^* e_0 \Rightarrow e \vdash^{\beta v} \rightarrow^* e_0$$

$$\text{(BPs)} \quad e \vdash^{\beta v} \rightarrow e_0 \Rightarrow e \vdash^p \rightarrow^* e_0$$

$$\text{(BsPs)} \quad e \vdash^{\beta v} \rightarrow^* e_0 \Rightarrow e \vdash^p \rightarrow^* e_0$$

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.

3.4 Church–Rosser Conclusions

We finish off this chapter with a discussion of previous related work, together with the conclusions drawn from the work presented here.

3.4.1 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 [36, 16, 35, 32, 26, 30].

The earliest treatment was by Shankar using the Boyer-Moore theorem prover [36], and later appears as a chapter in his PhD thesis [37]. 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 [16]. He uses de Bruijn indices in his formalization and establishes Church–Rosser as a corollary to his treatment of residuals.

Rasmussen [35] 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 [30] 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 commuting 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 [39].

Pfenning in [32] presents a development of the Tait–Martin-Löf proof in his Elf implementation of the Edinburgh LF [11]. 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.

3.4.2 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.

3.4.3 PVS Statistics

The actual proof of Church–Rosser in PVS took 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/> [8].

3.4.4 Acknowledgements

In the course of the work described in this chapter 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 results in this chapter appear in [10] and we thank the two anonymous CATS'01 referees for their insightful comments, that helped improve both the veracity and presentation of this chapter.

Chapter 4

The CIU Theorem

4.1 Introduction

In this chapter we report on the results of our second sophisticated and substantial use of PVS that establishes a recent result in operational semantics. This is of interest not only because it requires the substantial development of current higher order techniques in operational semantics, but also because it exposed several gaps in the published presentation of the result. Thus this work exemplifies the possible benefit of serious formalization in contrast to standard mathematical practice which typically leaves much unsaid. We also take great pains to formalize the actual theoretical treatment, rather than adapting it to the tastes of either the machine, or PVS. In this regard we were almost completely successful, only on two occasions was it necessary to deviate, slightly, from the exact formal treatment. We will mention them in the narrative. In this sense we made no use of the *tricky* representations that McKinna and Pollack discuss [26]. In our earlier work [10, 8] described in chapter 3 we carried out in PVS, for the first time, a *named variable proof* of the Church–Rosser result for Landin’s call-by-value Iswim, without eliminating α congruence by *tricky* encodings. Thus our work can be seen as an attempt to reconcile theory, with formal verification in practice. We should also point out that prior to this earlier work very little use of PVS’s inductive abstract data types had been made. Thus this work also represents the first use of these aspects of PVS to verify a non-trivial recent result, as opposed to a classic result that has been used somewhat as a benchmark [36, 16, 35, 32, 26, 30].

Thus this chapter has several, hopefully complementary, purposes. On the one hand it is a detailed presentation of the proof of the CIU theorem for uniform λ -languages, on the other hand it is a road map for the actual mechanized proof [9], and the issues raised in its development. We

also try and address some of the issues that are raised in presenting both a theoretical and the corresponding formal development. We will use the word *theoretical* to refer to the treatment of the subject matter as it normally appears in journal publications such as [41, 24], to contrast it with the word *formal* that refers to the analogous notion, as formalized in the corresponding PVS development. Also in this chapter we use the notation `file :<line number>` to refer to the particular line in the unpacked file, whose name is `file.pvs`, of [9]. So for example `CIU :210` is the actual statement of the main theorem presented in this chapter. We also extend this notation to include the name of the theorem, lemma, or definition when this is of interest. Thus `CIU : CIU :210` refers to the theorem `CIU` in the file `CIU.pvs` that lies on line 210 of the unpacked version of [9]. This system of presentation works well for the statements of the results contained in the development, but not for the formal proofs. An area of PVS that needs more attention.

4.1.1 Historical Background


Much work has been done to develop methods for reasoning about operational approximation and equivalence. Methods developed for reasoning about operational approximation and equivalence include: general schemes for establishing equivalence; context lemmas (alternative characterizations that reduce the number of contexts to be considered); and (bi)simulation relations (alternative characterizations or approximations based on co-inductively defined relations). An early example is Robin Milner's context lemma [27] which greatly simplifies the proof of operational equivalence in the case of the typed λ calculus by reducing the contexts to be considered to a simple chain of applications. Carolyn Talcott [40] studies general notions of equivalence for languages based on the call-by-value λ calculus, and develops several schemes for establishing properties of such relations. Doug Howe [13] develops a schema for proving congruence for a class of languages with a particular style of operational semantics. This schema succeeds in capturing many simple functional programming language features. Building on this work, Howe [14] uses an approach similar to the idea of uniform computation to define structured evaluation systems in which the form of the evaluation rules guarantees that (bi)simulation relations are congruences. The form of the rules is specified using meta variables with arities and higher-order substitutions. This syntax enrichment is very similar to the notions of place-holder and filling used here to specify uniform semantics. The idea of using such meta terms to specify classes of rules giving rise to reduction relations with special properties was used by Peter Aczel in [1] to prove a general Church-Rosser theorem and in Klop [17] to develop the theory of Combinatory Reduction Systems. Meta terms are also used in describing a unification procedure for higher-order patterns by Tobias Nipkow in [28]. Mason

and Talcott ([22, 23]) 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 [41]. It is this form of the lemma that we concentrate on in this chapter.

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, \dots , range over the set X . $\mathbf{P}_\omega(X)$ is the set of finite subsets of X . $\text{Fmap}[X_0, X_1]$ 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 $\text{app}(f, x)$ for the application $f(x)$ of the function f to the argument x . We write $\text{Dom}(f)$ for the domain of a function and $\text{Rng}(f)$ for its range. Thus if $f \in \text{Fmap}[X_0, X_1]$, then $\text{Dom}(f) \in \mathbf{P}_\omega(X_0)$. For any function f , $f\{x \mapsto x'\}$ is the function f' such that $\text{Dom}(f') = \text{Dom}(f) \cup \{x\}$, $f'(x) = x'$, and $f'(z) = f(z)$ for $z \neq x, z \in \text{Dom}(f)$. Also $f[X$ is the restriction of f to X : the function f' such that $\text{Dom}(f') = \text{Dom}(f) \cap X$ and $f'(x) = f(x)$ for $x \in \text{Dom}(f')$. $\mathbf{N} = \{0, 1, 2, \dots\}$ is the set of natural numbers and i, j, n, n_0, \dots 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:

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

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

4.2 Syntax of Terms

4.2.1 Background

In this section we present both the general framework as described in [41], and more recently in [24]. Concurrently we will describe how these notions are interpreted into PVS, as well as any interesting observations concerning their formalization.

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

The uniformity requirements are that each reduction rule hold not only for traditional expressions, but also for expressions containing parameters or meta variables. This parametric notion of computation is best presented using the idea of a context and the treatment here follows the general theory presented in [20].

4.2.2 The Syntax of Expressions

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

The *Theoretical* Treatment of Syntax

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

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

The basic syntax of a λ -language is then determined by specifying three sets:

(A): a countable set of atoms, \mathbf{A} , disjoint from \mathbf{X} and \mathbf{P} ;

(**O**): 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

(**V**): the set of value expressions, a subset of expressions, \mathbf{V} , that we will specify in more detail immediately after the definition of the syntax.

We assume that \mathbf{O} contains at least the binary operation `app` (lambda application). As we shall see later by taking $\mathbf{A} = \{ \}$ and $\mathbf{O} = \{\text{app}\}$ we obtain the expressions of the pure call-by-value λ calculus, Λ_v .

Definition 4.2.1 ($\mathbf{E}, \mathbf{L}, \mathbf{S}, \mathbf{F}$):

FLL – version – 03 :1–160

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

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

FLL – version – 03 :43-54

$$\mathbf{L} = \lambda \mathbf{X}. \mathbf{E}$$

FLL – version – 03 :49

$$\mathbf{S} = \text{Fmap}[\mathbf{X}, \mathbf{V}]$$

$$\mathbf{F} = \text{Fmap}[\mathbf{P}, \mathbf{E}]$$

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

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

The *Formal* Treatment of Syntax

In keeping with our attempt to be faithful to the theoretical treatment, we use finite maps in PVS to represent the finite maps σ and Σ . Also the entire development is parametric in a set of atoms and operations (which includes the binary `app`, together with the arities of those operations. This theory

is appropriately named `Landin`, and can be found in `FLL – version – 03 :11` Implementation of the syntax within PVS is a straightforward use of the *datatype with subtypes* facility. There are really only two technical issues that arise, and one design issue. We will look at the technical issues first, then discuss the design issue.

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

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

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

Auxilliary Syntactic Notions

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

Given the above implementation of the basic syntax, we must now develop the more basic auxilliary notions that are glossed over rather tersely in any theoretical treatment. We must define the rank of an expression, so as to be able to define by induction and recursion more complex notions. We must define derived notions such as the free variables of an expression, and the set of

parameters that occur in it. These sets will be finite, and this fact itself must be verified, usually as TCCs.

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

Rank

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

Definition 4.2.2 ($\text{rank}(e)$):

Rank :1-87

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

Derived Syntactic Notions

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 4.2.3 ($\text{FV}(e)$, $\text{FP}(e)$):

Vars :1-95

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

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

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

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

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

Definition 4.2.4 ($\text{Traps}(e)$): Traps :1-35

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

$$\text{Traps}(X^\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} \text{Traps}(\sigma(x)) \cup \text{Dom}(\sigma) \quad \text{Traps :20}$$

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.

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 [10], careful treatments of the λ -calculus will define, by structural recursion, the notion of a variable renaming $e^{\{x \mapsto y\}}$. One nice property possessed by variable renamings is that it preserves rank, unlike substitution, and to establish its totality we must build that fact into its type:

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

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

Definition 4.2.5 (**Renaming** $e^{\{x \mapsto y\}}$): Subst :26

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

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

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

Substitution

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

Definition 4.2.7 (Substitution e^σ):

Subst :110–182

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

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

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

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

Lemma 4.2.8 (Renaming): $e^{\{x \mapsto y\}} = e^{\{x \mapsto y\}}$

Subst : Subst_Rename :166

Filling

Definition 4.2.9 (Filling e^Σ):

Fill :1–73

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

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

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

α Equivalence

While the notion of being equivalent modulo the renaming of bound variables easily extends to this more general setting [20] 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 $\overset{\alpha}{\equiv}$ [20]:

Definition 4.2.10 ($\overset{\alpha}{\equiv}$):

WeakAlpha :1–108

$$\begin{array}{l}
 \text{(Base)} \quad \frac{}{v \overset{\alpha}{\equiv} v} \quad \text{provided } v \in \mathbf{A} \cup \mathbf{X} \\
 \text{(Op)} \quad \frac{e_0 \overset{\alpha}{\equiv} e'_0 \quad \dots \quad e_n \overset{\alpha}{\equiv} e'_n}{\vartheta(e_0, \dots, e_n) \overset{\alpha}{\equiv} \vartheta(e'_0, \dots, e'_n)} \\
 \text{(Subst)} \quad \frac{\text{Dom}(\Delta_0) = \text{Dom}(\Delta_1) \quad \Delta_0(\delta) \overset{\alpha}{\equiv} \Delta_1(\delta) \quad \forall \delta \in \text{Dom}(\Delta_i)}{\Delta_0 \overset{\alpha}{\equiv} \Delta_1} \\
 \Delta_0, \Delta_1 \in \mathbf{S} \text{ or } \Delta_0, \Delta_1 \in \mathbf{F} \\
 \text{(Param)} \quad \frac{\sigma_0 \overset{\alpha}{\equiv} \sigma_1}{X^{\sigma_0} \overset{\alpha}{\equiv} X^{\sigma_1}} \\
 \text{(Lambda)} \quad \frac{e_0^{\{x_0 \mapsto \nu\}} \overset{\alpha}{\equiv} e_1^{\{x_1 \mapsto \nu\}} \quad \text{for } \nu \text{ fresh.}}{\lambda x_0. e_0 \overset{\alpha}{\equiv} \lambda x_1. e_1}
 \end{array}$$

There is substantial leeway in the precise way one formalizes $\overset{\alpha}{\equiv}$. These issues are discussed at length in [26] so we shall not dwell on them here. The main point to make is that we attempted two formalizations, in keeping with the well tested rule of having weak introduction principles and strong elimination principles. The strong form was developed in Alpha :1-450 where we actually formalized the notion of being a proof in the above system, and defined $\overset{\alpha}{\equiv}$ by

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

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 [10]. The two forms were proved equivalent in WeakAlpha :85 . However it was the weaker form that proved the most useful in the subsequent development of the CUI theorem.

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

Lemma 4.2.11 ($\overset{\alpha}{\equiv}$): 1. $e_0 \overset{\alpha}{\equiv} e_1 \Rightarrow \text{rank}(e_0) = \text{rank}(e_1)$. WeakAlpha : WeakRank :76

2. $\overset{\alpha}{\equiv}$ is reflexive. WeakAlpha : WeakReflexive :88


- | | |
|--|--------------------------------|
| 3. $\stackrel{\alpha}{\equiv}$ is symmetric. | WeakAlpha : WeakSymmetric :94 |
| 4. $\stackrel{\alpha}{\equiv}$ is transitive. | WeakAlpha : WeakTransitive :97 |
| 5. $e_0 \stackrel{\alpha}{\equiv} e_1 \Rightarrow \text{FV}(e_0) = \text{FV}(e_1)$. | WeakAlpha : Alpha_FV :104 |

Substitution preserves α Equivalence

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

Lemma 4.2.12 (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 in [20] the proof of this is somewhat delicate, actually it is even more delicate than suggested there. Since the proof presented there does not stand up to the test of formalizing in PVS. In [20] the suggested proof requires establishing, directly, the following three properties by simultaneous induction. 

Lemma 4.2.13 (Alpha): AlphaSubst : AlphaSubst_Swap :132

1. $(e_0 \stackrel{\alpha}{\equiv} e_1 \wedge e'_0 \stackrel{\alpha}{\equiv} e'_1) \Rightarrow e_0^{\{x \mapsto e'_0\}} \stackrel{\alpha}{\equiv} e_1^{\{x \mapsto e'_0\}}$
2. $(\nu_0 \notin \text{FV}(e) \wedge \nu_0 \neq x \neq \nu_1) \Rightarrow (e_0^{\{\nu_0 \mapsto \nu_1\}})^{\{x \mapsto e\}} \stackrel{\alpha}{\equiv} (e_0^{\{x \mapsto e\}})^{\{\nu_0 \mapsto \nu_1\}}$
3. $(e_0^{\{x \mapsto \nu\}})^{\{\nu \mapsto y\}} \stackrel{\alpha}{\equiv} e_0^{\{x \mapsto y\}} \quad \text{for } \nu \notin \text{FV}(e_0)$

However the actual proof first must establish (by simultaneous induction) the corresponding three properties for renaming rather than substitution 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 4.2.13 is actually a property of renaming. It is now relatively simple to prove that substitution preserves α congruence.

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

Lemma 4.2.14 (SubstProps):

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

Filling preserves α Equivalence

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

Lemma 4.2.15 (AlphaFill):

AlphaFill : AlphaFill_Theorem :51

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

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

Lemma 4.2.16 (FillSwap):

AlphaFill : AlphaFill_Swap :47

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

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

Lemma 4.2.17 (FillProps):

AlphaFill :65–82

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

Definition 4.2.18 (Term, Closed):

Closed :1-27

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

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

Values

Definition 4.2.19 (Value Expressions (V)):

Values :1-122

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

(triv) \mathbf{V} is closed under $\stackrel{\alpha}{\equiv}$

Values :75



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

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

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

We let v range over \mathbf{V} .

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

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

Lemma 4.2.20 (inv): Values : ValuesSubst_Lemma :99

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

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

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

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

4.3 Semantics of Terms

4.3.1 Operational Semantics

In a λ -language 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 4.3.1 (State expressions (Z)): Computation :19

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

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



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

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

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

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

Definition 4.3.2 (Computation States (CS)): Computation :46

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

Definition 4.3.3 (Reduction (\longrightarrow , \longrightarrow^\gg) and Definedness (\downarrow)): Semantics :28

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

Definedness: Semantics :38

$$(\zeta : e) \downarrow \Leftrightarrow (e \in \mathbf{V}) \vee (\zeta : e \longrightarrow^\gg \zeta' : v)$$

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

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

Equal:

$$(\zeta_0 : e_0) = (\zeta_1 : e_1) \Leftrightarrow (\exists v \in \mathbf{V}, \zeta \in \mathbf{Z}) \left(\bigwedge_{j < 2} (\zeta_j : e_j) \longrightarrow^\gg \zeta : v \right)$$

Equivalent: Semantics :48

$$(\zeta_0 : e_0) \sim (\zeta_1 : e_1) \Leftrightarrow (\zeta_0 : e_0) \updownarrow (\zeta_1 : e_1) \wedge (\zeta_0 : e_0) \downarrow \Rightarrow (\zeta_0 : e_0) = (\zeta_1 : e_1)$$

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 [10]. They are treated in `re1 :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 a non-constructor operator applied to the appropriate number of value expressions. Redexes and value expressions must be disjoint, thus we must account for the fact that some expressions of the form $\vartheta(v_1, \dots, v_n)$ may be value expressions.

Definition 4.3.4 (Redexes (\mathbf{E}_r)):

Computation :65

The set of redexes, \mathbf{E}_r , is defined by:

$$\mathbf{E}_r = \mathbf{O}_n(\mathbf{V}^n) - \mathbf{V}$$

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

Lemma 4.3.5 (Redex Uniformity):

Computation :??

Redexes satisfy the following uniformity conditions:

(triv) \mathbf{E}_r is closed under $\stackrel{\alpha}{\equiv}$

Computation :??

(vsub) $r \in \mathbf{E}_r \Rightarrow r^\sigma \in \mathbf{E}_r$

Computation :88

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

Computation :??

We use the distinguished parameter \bullet to denote the *evaluation* parameter (or hole), and we define the notion of a reduction context, R , accordingly. Reduction contexts (also called evaluation contexts in the literature) identify the subexpression of an expression in which reduction to a value must occur next. They themselves represent the remainder of the computation, i.e the *continuation*. In our approach they correspond to the left-first, call-by-value reduction strategy of [33] and were first introduced by [6].

Definition 4.3.6 (Reduction Contexts (\mathbf{R})):

Computation :70

The set of reduction contexts, \mathbf{R} , is the subset of \mathbf{E} defined by

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

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

Observe that both the definition of redex, and the definition of reduction contexts depend on the particular choice of values, and thus vary from one λ -language to another. Also note that \mathbf{R} will satisfy a similar set of uniformity conditions as those satisfied by states (Definition 4.3.1):

Lemma 4.3.7 (Rcx Uniformity):

Computation :1-233

Reduction contexts satisfy the following uniformity conditions:

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

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

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

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

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

Lemma 4.3.8 (Decomposition):

Computation : Rcx_Decomposition_Theorem :179

For any λ -language, if $e \in \mathbf{E}$ then either $e \in \mathbf{V}$ or e can be written uniquely as either

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

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

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

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

4.3.2 Uniform Semantics

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

Definition 4.3.9 (Uniformity (U)):

Semantics :1–121



A λ -language is said to have *uniform semantics* if it satisfies the following:

(i) Functional modulo \equiv and implicit bindings:

Semantics :70

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

$$\left(\bigwedge_{j < 2} \zeta : e \longrightarrow \zeta_j : e_j\right) \Rightarrow \zeta_0^{\{\circ \mapsto e_0\}} \equiv \zeta_1^{\{\circ \mapsto e_1\}}$$

(ii) Uniform in value substitutions:

Semantics :75

$$\zeta : e \longrightarrow \zeta' : e' \Rightarrow (\zeta : e)^\sigma \longrightarrow (\zeta' : e')^\sigma \quad \text{provided } \text{Dom}(\sigma) \cap (\text{Traps}(\zeta) \cup \text{Traps}(\zeta')) = \emptyset \text{ and } \circ \notin \text{FP}(\sigma).$$

(iii) State evaluation:

Semantics :79

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

(iv) Well-founded:

Semantics :82

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

(v) Parametric:

Semantics :86

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

(vi) Dichotomy:

Semantics :90

Assuming $\circ \notin \text{Dom}(\Sigma)$, if $(\zeta : e)^\Sigma \longrightarrow \zeta' : e'$ then either

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

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

(vii) Closure:

Semantics :95 & 100

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

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

In the languages we consider (U) holds for the following reasons. (U.i) holds because the only non-determinism in a reduction step is the choice of names used in the state context. (U.ii) holds because reductions that do not depend on the values of free variables, are parametric in the values that those variables take. (U.iii) holds because reduction of $\circ : \zeta^{\{\circ \mapsto e\}}$ essentially recreates

the state context ζ . **(U.iv)** follows since if a state is defined, then any reduction makes progress. Clearly if the reduct state is defined, then the original state is defined. **(U.v)** and **(U.vi)** formalizes the uniformity requirement for reduction steps. **(U.vi)** states that either computation touches a parameter or is parametric. These are satisfied by reduction rules that treat the reduction context as an abstract entity, and that depend on the kind of construction of a redex argument, but not on any information about subparts. This is easily expressed using the parameters. Finally, **(U.vii)** holds because computation neither introduces new parameters, nor new free variables. Furthermore the state parameter does not propagate into the expression being evaluated.

4.3.3 Approximation and Equivalence

Now we define operational approximation and equivalence on terms and lay the ground work for studying properties of these relations. In what follows we fix a particular distinguished parameter, X , distinct from \circ and \bullet . We let C range over expressions with X as the only free parameter. Such expressions play the role of traditional λ -calculus contexts, and we extend our convention, stated in definition 4.3.6, of sometimes writing $C[e]$ instead of $C^{X \mapsto e}$. Note however that for example the traditional context $\lambda z.\text{app}(y, \lambda x.\text{app}([\], z))$ does not correspond to $\lambda z.\text{app}(y, \lambda x.\text{app}(X, z))$ but rather to one where the trappings have been made explicit at the occurrence of the X parameter: $\lambda z.\text{app}(y, \lambda x.\text{app}(X^{\{x \mapsto x, z \mapsto z\}}, z))$.

Definition 4.3.10 (Approximation $e_0 \sqsubseteq e_1$, Equivalence $e_0 \cong e_1$): Approx :1–76

For terms e_0, e_1 define

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

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

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

4.4 The Proof of CIU

4.4.1 The CIU Theorem

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

Theorem 4.4.1 (CIU): CIU : CIU :210



For a λ -language with uniform semantics:

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

under the assumption that $FV(e_0) = FV(e_1)$.

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

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

Lemma 4.4.2 (Closed):

CIU : Ciu_Closed :83



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

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

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

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

To see that the first three are all necessary consider:

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

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

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

Lemma 4.4.3 (Delay):

CIU : Ciu_Delay :100

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

4.4.2 The CIU Proof

The fact that one can present a syntactic reduction system for imperative λ -calculi was discovered independently in 1986-1987 in [23], and in [7]. As well as being conceptually elegant, it has also provided the necessary tools for several key results and proofs. In addition to eliminating messy

isomorphism considerations, to deal with arbitrary choice of names of newly allocated structures, it also was a key step leading to the formulation of the CIU theorem, first presented in [22]. There are now several proofs of this result in the literature. The theorem first appeared in [23] 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 [41]. A second, distinct, proof was presented in [12] that simply shows that the CIU relation is a congruence. This proof also appears in a more general setting in [21].

Proof: (CIU \Rightarrow)

CIU : CIUR :153

The (CIU \Rightarrow) direction is relatively simple, and requires producing a context C such that $C[e_i]$ evaluates to $\zeta : R[e_i^\sigma]$. 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 $x \in \text{Dom}(\sigma)$. Extend σ to $\hat{\sigma}$ by mapping each $x \in \text{Traps}(\zeta) - \text{Dom}(\sigma)$ to itself. Note that $e_i^{\hat{\sigma}} = e_i^\sigma$. Choose an new parameter X and consider the expression $C = \zeta^{\{\circ \mapsto R^{\{\bullet \mapsto X^{\hat{\sigma}}\}}\}}$.

Lemma 4.4.4:

CIU : CIUR_Helper2 :129

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

Now by this lemma $\circ : C[e_j]$ are closed, thus by definition 4.3.10 we have that

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

Then $\circ : C[e_j] \uparrow \zeta : R[e_j^\sigma]$ by lemma 4.4.4, and **(Unif.iii)**. Thus $\zeta : R[e_0^\sigma] \preceq \zeta : R[e_1^\sigma]$ as desired. \square

Proof: (CIU \Leftarrow)

CIU : CIUL :206

Assume that

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

We prove

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

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

The proof is relatively straightforward modulo the hard case. The hard case is when e_0 is a value, and $(\zeta : e)^{\{X \mapsto e_0\}}$ non-trivially reduces to a value. This case itself must be proved by

another induction. This induction is on the number of *non-nested occurrences* of X that occur to the left, or below, where the computation is currently taking place. An occurrence of X in e is said to be *non-nested* if it is not in the scope of a λ -expression, or in the range of a substitution annotating a parameter.

(CIU \Leftarrow) Base Case:

CIU : CIUL_Base :156

Suppose that $(\zeta : e)^{\{X \mapsto e_0\}}$ is a value state. Thus $e^{\{X \mapsto e_0\}}$ must be a value. Thus by **(dich)** of definition 4.2.19, either e is a value or else it is of the form X^σ for some σ , and e_0 itself must be a value. In the case where e is a value, we have that $e^{\{X \mapsto e_1\}}$ is also a value by **(fill)** of definition 4.2.19, and hence $(\zeta : e)^{\{X \mapsto e_1\}}$ is also value state (note that $\zeta^{\{X \mapsto e_1\}} \in \mathbf{Z}$ is implicit in the hypothesis). So suppose that e_0 is a value, and that e is of the form X^σ for some σ . Also let $\zeta_j = \zeta^{\{X \mapsto e_j\}}$ and $\sigma_j = \sigma^{\{X \mapsto e_j\}}$ for $0 \leq j < 2$. Then in this case

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

is a value state, and hence

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

must also be a value state by **(fill)** of definition 4.2.19. We also claim that $\zeta_1 : e_0^{\sigma_1}$ is closed by lemma 4.4.2. By assumption $\zeta_1 : e_1^{\sigma_1}$ is closed. Thus we may use the **(ciu)** hypothesis with $\zeta = \zeta_1$, $R = \bullet$, and $\sigma = \sigma_1$ to conclude that

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

Now simply observe that

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

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

(CIU \Leftarrow) Induction Case:

CIU : CIUL_IH :166

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

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

(CIU \Leftarrow) Induction Case (i):

CIU : CIUL :221

In (i) we have by the induction hypothesis that $(\zeta' : e')^{\{X \mapsto e_0\}} \preceq (\zeta' : e')^{\{X \mapsto e_1\}}$, and thus by

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

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

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

CIU : CIUL :221

Since $e_0 \notin \mathbf{V}$ we have by **(Decomposition)** (i.e. lemma 4.3.8) that either e_0 uniquely decomposes into $R_0[r_0]$, or else $R_0[X_0^{\sigma_0}]$. However the latter case is ruled out since e_0 is a term. So $e_0 = R_0[r_0]$. Now by lemma **(delay)**

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

Now since $R[R_0^\sigma]$ is a reduction context, and r_0^σ is still a redex by **(inv)** (i.e. lemma 4.2.20), we have that $(\zeta : R[e_0^\sigma])$ does not touch the parameter X and so reduces uniformly by **(Unif.vi)**. Thus $(\zeta : R[e_0^\sigma]) \longrightarrow (\zeta' : e')$ for some $(\zeta' : e')$. Thus by **(Unif.v)** $(\zeta : R[e_0^\sigma])^{\{X \mapsto e_0\}} \longrightarrow (\zeta' : e')^{\{X \mapsto e_0\}}$ and hence $(\zeta' : e')^{\{X \mapsto e_0\}} \downarrow$, so by the induction hypothesis $(\zeta' : e')^{\{X \mapsto e_1\}} \downarrow$. Now also by **(Unif.v)** $(\zeta : R[e_0^\sigma])^{\{X \mapsto e_1\}} \longrightarrow (\zeta' : e')^{\{X \mapsto e_1\}}$, consequently we may conclude that $(\zeta : R[e_0^\sigma])^{\{X \mapsto e_1\}} \downarrow$. Now put $\zeta_1 = \zeta^{\{X \mapsto e_1\}}$, $R_1 = R^{\{X \mapsto e_1\}}$, and $\sigma_1 = \sigma^{\{X \mapsto e_1\}}$. Then by lemma 4.4.2 we may instantiate **(ciu)** and conclude

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

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

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

CIU : CIUL_Value :182

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

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

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

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

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

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

Now the induction hypothesis can be used to show that

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

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

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

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

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

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

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

We do this by considering the expression:

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

obtained by filling all but the *right most non-nested occurrence* of X in $e(X, X, \dots, X \mid e_1, e_1, \dots, e_1)$.

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:

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

It then only remains to show that $e(X, X, \dots, X, e_1 \mid e_1, e_1, \dots, e_1)$ satisfies the induction hypothesis, and has one less occurrence of the parameter X . Clearly the number of non-nested

parameters decreases by one. To see that $e' = e(X, X, \dots, X, e_1 \mid e_1, e_1, \dots, e_1)$ has the desired decomposition, and reduces non-trivially we consider two cases, depending on whether or not e_1 is a value or not. When e_1 is a value the same decomposition remains valid, and as the computation is uniform, non-triviality follows. If e_1 is not a value, then since it is a term it must be of the form: $R_1[r_1]$. Then $C' = e(X, X, \dots, X, R_1 \mid e_1, e_1, \dots, e_1)$ and $c' = r_1$ suffices. \square

4.5 CIU Conclusions

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

The formalization of the *annotated holes* technique is also a highlight of the work. Providing the unusual technique with a unquestionable basis. However the most important aspect of the work reported here is the way it, to mix metaphors, dragged the groundbreaking paper [41] over the hot coals, or put it through a fine tooth comb. The final product [24] is certainly the better for it. To recap: the process of formalizing the CIU theorem revealed three major categories of problems with the published theoretical versions. Unstated closure conditions on the set of values, and set of states. Unstated but necessary uniformity requirements needed for the proof to be carried out. Finally some extra conditions on the terms or contexts considered in the actual CIU proof to ensure that replacing one expression preserves the *closedness* of the computation states involved.

4.5.1 PVS Statistics

The actual proof of CIU in PVS took approximately four months (July 00 through early November 00). The actual machine checked proof involves the proving of two hundred and sixty six (266) distinct facts, and takes PVS two thousand six hundred and sixty six (2662) seconds (44 minutes) of CPU time running on a Linux machine configured with 2 GBytes of main memory and 4×550 MHz Xeon PIII processors. The dump file containing all the PVS definitions, facts, and proofs is 8.362 MBytes and is available from <http://mcs.une.edu.au/~pvs/> [9]. It thus represents roughly four times as much work as was required to prove the Church–Rosser theorem in PVS, as described in the earlier chapter and reported in [10].

4.5.2 Acknowledgments

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

Chapter 5

Conclusions

The most important conclusion is that PVS can be used as a tool in modern operational techniques. The abstract data type mechanism of PVS is extremely useful in defining such structures as the syntax of expressions for languages, although some aspects of this feature could be improved to provide more automated proving. All in all, however, the use of ADTs for inductive definitions removes a lot of the more menial work from the specification process.

Several bugs were also discovered in PVS itself and these have been submitted to the developers of PVS at Menlo Park. The discovery of bugs in the implementation of PVS leads to improvements both in the integrity of the system, and also the prover. Since PVS version 2.1 was announced on April 24, 1997, roughly 500 bugs have been submitted. These include problems with all aspects of PVS such as the type checker, parser and prover.

To our knowledge the work on the Church–Rosser theorem is the first time that α -equivalence has been formalized in a mechanized proof. Thus certain details concerning the properties of α -equivalence have for the first time, been subject to the rigors of formal verification. Thus we can conclude that reasoning with \equiv^α and its properties in a theorem prover is a practical option. However, working in the quotient space modulo \equiv^α is still preferable due to the reduced complexity.

The process of formalizing the CIU theorem revealed three major categories of problems with the published theoretical versions. Unstated closure conditions on the set of values, and set of states. Unstated but necessary uniformity requirements needed for the proof to be carried out. Finally some extra conditions on the terms or contexts considered in the actual CIU proof to ensure that replacing one expression preserves the *closedness* of the computation states involved. Thus such methods of verification are indeed useful for testing existing theory for problems such as the omission of necessary conditions and assumptions.

The CPU time taken to prove the CIU theorem is much greater than the CPU time for the

Church–Rosser theorem (2662 seconds for CIU as opposed to 362 for Church–Rosser). This is in part due to the more complex type definitions for CIU including subtypes for expressions and values. The generation of TCC both in the specification and in the prover played a large part in the amount of time required for the CIU proof. Methods for combating this problem either with judgements or lemmas would definitely be a key issue of further work done in PVS.

So finally in summary, the main conclusions that can be drawn from the work done in this thesis are:

- PVS can be used as a tool in modern operational techniques;
- the work on the Church–Rosser theorem is the first known proof that formalizes α congruence;
- the proof of the CIU theorem is the first use of PVS to prove a recent result in operational semantics;
- formal verification can be used to uncover problems in existing theory.

Thus in the future, tools like PVS are likely to play an important role in language design, implementation and even program development itself.

Bibliography

- [1] P. Aczel. A generalized Church-Rosser Theorem, July 1978.
- [2] R. W. Butler, B. Dutertre, D. Jamsek, S. Owre, and D. Griffioen. PVS finite set library, 1997. available at http://pvs.csl.sri.com/pvs/libraries/finite_sets.dmp.
- [3] A. Church and J.B. Rosser. Some properties of conversion. *Transactions of the AMS*, 39:472–482, 1936.
- [4] 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.
- [5] 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*, 34(5):381–392, 1972.
- [6] 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.
- [7] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.
- [8] J. Ford. The Church–Rosser theorem in PVS, 2000. PVS dump file (2.4 Megabytes) available at <http://mcs.une.edu.au/~pvs/>.
- [9] J. Ford. The CIU theorem in PVS, 2000. PVS dump file (approximately 9 Megabytes) available at <http://mcs.une.edu.au/~pvs/>.
- [10] J. Ford and I. A. Mason. Operational Techniques in PVS – A Preliminary Evaluation. In *Proceedings of the Australasian Theory Symposium, CATS '01*, 2001.

- [11] R. Harper, H. Honsell, and G. Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.
- [12] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 1995.
- [13] D. J. Howe. Equality in the lazy lambda calculus. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.
- [14] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
- [15] G. Huet. Residual Theory in λ -Calculus: A Formal Development. Technical Report 2009, INRIA, 1993.
- [16] G. Huet. Residual Theory in λ -Calculus: A Formal Development. *Journal of Functional Programming*, 4(3):371–394, 1994. An earlier version appeared as [15].
- [17] J. Klop. *Combinatory Reduction Systems*. Number 127 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1980.
- [18] P. J. Landin. A correspondence between Algol60 and Church’s lambda notation. *Comm. ACM*, 8:89–101, 158–165, 1965.
- [19] I. A. Mason. Parametric Computation. In James Harland, editor, *Proceedings of the Australasian Theory Symposium, CATS ’97*, pages 103 – 112, 1997.
- [20] I. A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, 12:171–201, 1999. An abridged version appears as [19].
- [21] I. A. Mason, S. F. Smith, and C. L. Talcott. From Operational Semantics to Domain Theory. *Information and Computation*, 128(1):26–47, 1996.
- [22] 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.
- [23] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

- [24] 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.
- [25] 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.
- [26] J. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23, 1999. An abridged version appeared as [25].
- [27] R. Milner. Fully abstract models of typed λ -calculi. *Theoretical Computer Science*, 4:1–22, 1977.
- [28] T. Nipkow. Higher-order critical pairs. In *Sixth Annual Symposium on Logic in Computer Science*. IEEE, 1991.
- [29] 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.
- [30] T. Nipkow. More Church-Rosser Proofs (in Isabelle/HOL), 2000. to appear in *Journal of Automated Reasoning*. An abridged version appears as [29].
- [31] 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.
- [32] 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 [31].
- [33] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [34] 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.
- [35] O. Rasmussen. The Church–Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, 1995.

- [36] N. Shankar. A Mechanical Proof of the Church-Rosser Theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, 1988.
- [37] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge University Press, 1994.
- [38] M. Sorensen and P. Urzyczyn. Lectures on the Curry-Howard Isomorphism. Technical Report 14, DIKU Report, 1998.
- [39] M. Takahashi. Parallel reductions in the λ -calculus. *Information and Computation*, 118:120–127, 1995.
- [40] C. L. Talcott. *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation*. PhD thesis, Stanford University, 1985.
- [41] C. L. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1996.