

A FORMALISATION OF WEAK NORMALISATION (WITH RESPECT TO PERMUTATIONS) OF SEQUENT CALCULUS PROOFS

A. A. ADAMS

Abstract

Dyckhoff and Pinto present a weakly normalising system of reductions on derivations in the cut-free intuitionistic sequent calculus, where the normal derivations are characterised as the fixed points of the composition of the Prawitz translations into natural deduction and back. This paper presents a formalisation of the system, including a proof of the weak normalisation property for the formalisation. More details can be found in earlier work by the author. The formalisation has been kept as close as possible to the original presentation to allow an evaluation of the state of proof assistance for such methods, and to ensure similarity of methods, and not merely similarity of results. The formalisation is restricted to the implicational fragment of intuitionistic logic.

1. *Introduction*

Research into the properties of logics (meta-theory) is a rich and diverse field. Despite the tools for implementing object logics in logical frameworks [28, 29] developed over the last twenty years, little has been done in the area of machine assistance for meta-theory. Investigations of the properties of λ -calculi [8, 27, 2] and typechecking Pure Type Systems [49] have been carried out, but the combination of these approaches into systems for performing general meta-theoretic proofs about sequent-style calculi with proof terms has not been investigated. Work on meta-theory has usually concentrated on a single theory, and on developing object logic-specific formalisations; for example, [7, 46]. The work presented here has coalesced into another specific formalisation. The lack of work on a general framework for such investigations pushes individual attempts at formal meta-theory into such single tracks. The approach described in [42], using ELF, lacks the automated production of induction schemes and relies for its correctness on the schema-checking that takes place outside the formal meta-system.

Meta-theoretic proofs are just the sort of complex proofs for which machine-assistance is most needed. Until recently, the capabilities of proof assistants and logical frameworks were unequal to the task of useful assistance for general meta-theory. While it may have been possible, the amount of work involved in encoding each logic was prohibitive in relation to the assistance gained. Performing formal proofs by hand would have been easier than attempting to formalise the systems within a machine-assisted framework.

This paper presents the main points of a formalisation of meta-theory for the implicational fragments of three sequent-style (or ‘logistic’ style [47]) systems: the sequent calculus (**LJ**),

Received 18 May 1999, revised 19 November 1999; published 7th February 2000.

2000 Mathematics Subject Classification 03F03

© 2000, A. A. Adams

a variant of the sequent calculus (**MJ**) and a sequent-style presentation of natural deduction (**NJ**). Using this formalisation, proofs of properties of the systems and of relationships between the systems have been performed. Most of this work has involved formalising and checking existing informal proofs [15, 17], but there have been some forays into areas left unexplored in the informal development. A more detailed description and discussion of this work can be found in [1].

In addition to demonstrating the capabilities of modern theorem-proving software in respect of formalising meta-theory, the formalisation described in this paper demonstrates a number of the core problems that must be addressed when considering tools for formal meta-theory. In particular, the problem of converting non-primitive recursive functions using named variables to primitive recursive form using de Bruijn indices is an interesting one.

Supporting this paper is the full development, compatible with *Coq* Version 6.1. Most of the examples of the formalisation are taken directly from this supporting code, the exception being functions defined by mutual recursion (see Section 3.2 for details). This material is to be found in [Appendix A](#).

1.1. Notation

When discussing the meta-theory of logical systems, ‘proof’ can easily become overloaded with too many meanings. To avoid this, the following nomenclature has been adopted in this paper: ‘proof’ is reserved for the proof of a meta-theoretic result; ‘derivation’ will be used to indicate a proof at the object level within the sequent calculi **LJ** and **MJ**; ‘deduction’ will be used to indicate a proof at the object level within the natural deduction calculus **NJ**. All three systems are term calculi, and the proof terms will be referred to as ‘derivation terms’ or ‘deduction terms’ as appropriate.

1.2. Overview

The paper is set out as follows. Section 2 includes an overview of the informal theory. Section 3 gives a quick introduction to the *Coq* proof assistant and its type theory. Next, Section 4 presents the interesting highlights of the formalisation. Section 5 gives some pointers to related work, and how this formalisation fits within the field; finally, Section 6 draws conclusions about the current and future directions of such work.

2. The informal theory

This section contains a brief overview of the theory being formalised. A more complete version of the original proofs can be found in [17]. Note that for brevity universal quantification has been elided from all the theorems stated. All variables and meta-variables in the stated theorems should be assumed to be universally quantified at the appropriate level unless otherwise quantified. In addition, a fairly obvious naming system for variables is used throughout, with the following type assignments: $m:\mathbf{M}$, $ms:\mathbf{Ms}$, $a:\mathbf{A}$, $n:\mathbf{N}$, $l:\mathbf{L}$ (also holding for subscripted versions of the variable names).

2.1. Overview

It has long been a piece of logic folklore that two cut-free sequent calculus derivations are really the same if and only if they correspond to the same normal natural deduction. Kleene [30] discusses permutability of inferences in sequent calculus without reference to the corresponding natural deductions. The relationships between individual sequent calculus derivations can be described using a set of permutations, such that two cut-free sequent

calculus derivations are inter-permutable if and only if they correspond to the same normal natural deduction. An obvious extension of this idea is to try to produce a set of reductions which replace the bi-directional permutations, and indeed to try and find a confluent set of reductions, which lead to a ‘normal’ form for sequent calculus derivations.

But what is ‘normal’ in this sense? In [17] ‘normal’ is defined syntactically (see Table 7) in such a way that the normal derivations are immutable under the composition of the Prawitz translations into natural deduction and back. The translation from natural deduction to sequent calculus, unlike the reverse translation [43, 18], has not been formally defined in the early literature. [43] describes the general form of this translation (here called ρ) but does not give a full syntactic definition. It is also described in [48]. In fact, the translation is naturally formed as the composition of the translations via an intermediate calculus, due to Herbelin in [26] and refined by Dyckhoff and Pinto as the *permutation-free sequent calculus* in [15]. Note that this new calculus is called **MJ** in [15] to avoid confusion between Herbelin’s name *LJT* and Dyckhoff’s different calculus *LJT* in [14]. There are therefore two distinct parts to this work. **MJ** must be shown to be isomorphic to natural deduction [15], and the reductions must be shown to be normalising with respect to the retraction of **LJ** into itself via **MJ**. Note that similar ideas to those of Dyckhoff and Pinto in [17] may also be found in work by Mints [36, 37].

We will next present an overview of the three calculi, as presented in [15], followed by the relationships between them, the permutation reduction and brief descriptions of the informal proofs from [15]. Figure 1 will show as an example the proof of the S combinator in standard intuitionistic logic and each of **NJ**, **MJ** and **LJ**. We also include an example of a permutation of a proof of the S combinator in **LJ** to a different proof in **LJ** (which of course are both equivalent to the same proofs in **NJ** and **MJ**).

2.2. Three sequent-style calculi

To present a coherent picture of the three systems, a single approach is taken for all three. The systems are defined using a sequent-style notation, although only **LJ** and **MJ** are sequent calculi in the sense of Gentzen’s original version [22], while **NJ** is a sequent-style calculus equivalent to natural deduction with assumption classes [32]. All three systems are cut-free. Normalisation for **NJ** and cut-elimination for LJ^{+cut} are well-known, and cut-elimination for MJ^{+cut} has been shown in [26]; see also [16]. **NJ** also differs from a standard presentation of the simply-typed λ -calculus in its splitting of terms into *normal* (**N**) and *applicative* (**A**) terms. *Normal* terms (**N**) have the form:

$$\lambda x_1 \dots x_n. ((\dots (x t_1) \dots t_{m-1}) t_m)$$

where the t_i are *normal*. The sets of proof terms of these systems are **A** and **N** for **NJ**, **M** and **Ms** for **MJ**, and **L** for **LJ**, defined as follows:

$$\begin{aligned} \mathbf{N} &::= \lambda \mathbf{V}.\mathbf{N} \mid \mathbf{a}(\mathbf{A}) & \mathbf{M} &::= (\mathbf{V} ; \mathbf{Ms}) \mid \lambda \mathbf{V}.\mathbf{M} \\ \mathbf{A} &::= \mathbf{ap}(\mathbf{A}, \mathbf{N}) \mid \mathbf{var}(\mathbf{V}) & \mathbf{Ms} &::= [] \mid \mathbf{M} :: \mathbf{Ms} \\ \mathbf{L} &::= \mathbf{vr}(\mathbf{V}) \mid \mathbf{app}(\mathbf{V}, \mathbf{L}, \mathbf{V}.\mathbf{L}) \mid \lambda \mathbf{V}.\mathbf{L} \end{aligned}$$

where **V** is the set of variables (x, y, \dots) and ‘.’ is a binding operator. The term $\mathbf{app}(x, l_1, y.l_2)$ is the term of **L** representing an occurrence of the *Implies Left* rule: the translation into natural deduction satisfies

$$|\mathbf{app}(x, l_1, y.l_2)| = |\mathbf{ap}(x, |l_1|/y)|l_2|,$$

where $[N_1/x]N_2$ is the usual substitution of N_1 for x in N_2 . Taking P, Q, R as meta-variables for formulae and Γ for contexts (a context is a function from a finite set of variables to formulae), the rules for the three systems are shown in Table 1.

For those unfamiliar with such presentations of logics, Figure 1 shows proofs of the S-Combinator in standard natural deduction, standard simply typed λ -calculus and in **LJ**, **MJ** and **NJ**. The commonly occurring context $\Gamma = x : A, y : A \supset B, z : A \supset (B \supset C)$ is used for brevity in the proof trees, and the abstraction steps creating this context have been omitted.

2.3. Relationships between the calculi

The first requirement is to establish the translations between the systems, and to prove the admissibility of these translations. That is, it must be established that a translated proof term proves the same formula, given the same context. The translation functions are shown in Tables 2 and 3. The theorems establishing the relationships between the calculi and showing admissibility of the translations are shown in Table 4.

2.4. Permutations in **LJ**

Once the basic definition of each system has been formalised, and the required relationships between the systems have been proved to hold (in particular the isomorphism between **MJ** and **NJ**), the next task is to formally define the system of reductions on untyped terms of **L**, shown as the relation ‘ \succ ’ in Table 5. The reflexive transitive closure of this relation (\succ^*) is defined in the usual way.

To show how permutations affect proofs in **LJ** we show in Figure 2 two proofs taken from [15], the first of which is also shown in Figure 1.

Next, we must prove that subject reduction holds for \succ (see Table 6): Theorem `L_Admiss_Perm1` shows subject reduction for one step of \succ and theorem `L_Admiss_Permn` for the closure \succ^* . Subject reduction is the property, of a reduction relation on terms in a type theory, that a term reduced (in this case by \succ) retains the type of the original term. When thinking in terms of logic rather than type theory, subject reduction means that a reduced proof (term) still proves the same formulae (type) as the original. Finally, weak normalisation of the set of reductions follows from three theorems, `Norm_Imperm_L`, `Norm_L_rho` and `Norm_Red` (also in Table 6), as per the specification of weak normalisation for *abstract reduction systems* in [31, Definition 2.0.3(2)]. The normal form to which terms are rewritten is defined informally in Table 7, the formal definition being very similar. `Norm_Imperm_L` asserts that a term l in **L** in normal form cannot be further reduced by \succ . `Norm_L_rho` states that any term in **L** which is the image via $\bar{\rho}$ of a term in **M** is normal (and therefore by `Norm_Imperm_L` is irreducible). Finally, `Norm_Red` asserts that there exists a (possibly empty) path of reductions \succ^* from any term l in **L** to its image via a translation into **M** and back to **L**. Since there is no simple way of directly stating weak normalisation, the main result that is formalised here consists of these three lemmas.

2.5. Informal proofs

Most of the theorems in Tables 4 and 6 are provable by straightforward structural induction — mutual inductions for theorems involving **MJ** or **NJ**. The proof of theorem `App_Red_M` (see Table 6) requires a stronger induction principle, however. The definition of $\bar{\rho}$ also requires some justification of its termination property. The height of terms in **M**

Natural Deduction

$$\frac{\frac{A \supset (B \supset C) \quad A}{(B \supset C)} \quad \frac{A \supset B \quad A}{B}}{C}$$

NJ

$$\frac{\frac{\frac{\Gamma \triangleright \text{var}(z) : A \supset (B \supset C)}{\Gamma \triangleright \text{ap}(\text{var}(z), \text{an}(\text{var}(x))) : (B \supset C)} \quad \frac{\frac{\Gamma \triangleright \text{var}(x) : A}{\Gamma \gg \text{an}(\text{var}(x)) : A} \quad \frac{\Gamma \triangleright \text{var}(y) : A \supset B \quad \frac{\Gamma \triangleright \text{var}(x) : A}{\Gamma \gg \text{an}(\text{var}(x)) : A}}{\Gamma \triangleright \text{ap}(\text{var}(y), \text{an}(\text{var}(x))) : B}}}{\Gamma \gg \text{an}(\text{ap}(\text{var}(y), \text{an}(\text{var}(x)))) : B}}}{\Gamma \triangleright \text{ap}(\text{ap}(\text{var}(z), \text{an}(\text{var}(x))), \text{an}(\text{ap}(\text{var}(y), \text{an}(\text{var}(x)))))) : C}}{\Gamma \gg \text{an}(\text{ap}(\text{ap}(\text{var}(z), \text{an}(\text{var}(x))), \text{an}(\text{ap}(\text{var}(y), \text{an}(\text{var}(x)))))) : C}}$$

 Simply Typed λ -Calculus

$$\frac{\frac{\Gamma \vdash z : A \supset (B \supset C) \quad \Gamma \vdash x : A}{\Gamma \vdash (zx) : (B \supset C)} \quad \frac{\Gamma \vdash y : A \supset B \quad \Gamma \vdash x : A}{\Gamma \vdash (yx) : (B \supset C)}}{\Gamma \vdash ((zx)(yx)) : C}$$

LJ

$$\frac{\frac{\frac{w : (B \supset C), v : B, \Gamma \quad u : C, w : (B \supset C), v : B, \Gamma}{\rightarrow \text{vr}(v) : B} \quad \frac{u : C, w : (B \supset C), v : B, \Gamma}{\rightarrow \text{vr}(u) : C}}{w : (B \supset C), v : B, \Gamma \rightarrow \text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u)) : C}}}{\frac{\Gamma \rightarrow \text{vr}(x) : A \quad v : B, \Gamma \rightarrow \text{app}(\text{vr}(z), \text{vr}(x), w.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u))) : C}}{\Gamma \rightarrow \text{app}(\text{vr}(y), \text{vr}(x), v.\text{app}(\text{vr}(z), \text{vr}(x), w.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u)))) : C}}$$

MJ

$$\frac{\frac{\frac{\frac{\Gamma \xrightarrow{A} [] : A}{\Gamma \Rightarrow (x; []) : A} \quad \frac{\Gamma \xrightarrow{B} [] : B}{\Gamma \xrightarrow{B} (x; []) : B}}{\Gamma \xrightarrow{A \supset B} (x; []) : B}}{\frac{\frac{\Gamma \xrightarrow{A} [] : A}{\Gamma \Rightarrow (x; []) : A} \quad \frac{\frac{\Gamma \xrightarrow{B} (y; (x; [])) : B}{\Gamma \xrightarrow{B \supset C} (y; (x; [])) : B} \quad \Gamma \xrightarrow{C} [] : C}}{\Gamma \xrightarrow{A \supset (B \supset C)} (x; []) : C}}{\Gamma \Rightarrow (z; (x; [])) : C}$$

Figure 1: Proofs of the S-combinator in various systems

Table 1: Proof rules for **NJ**, **MJ** and **LJ**

NJ	
$\frac{x : P, \Gamma \gg n : Q}{\Gamma \gg \lambda x.n : (P \supset Q)} \supset I$	
$\frac{\Gamma \triangleright a : P}{\Gamma \gg \text{an}(a) : P} \text{ AN}$	
$\frac{\Gamma \triangleright a : (P \supset Q) \quad \Gamma \gg n : P}{\Gamma \triangleright \text{ap}(a, n) : Q} \supset E$	
$x : P \in \Gamma \quad \overline{\Gamma \triangleright \text{var}(x) : P} \text{ A-Axiom}$	
MJ	
$x : P \in \Gamma \quad \frac{\Gamma \xrightarrow{p} ms : R}{\Gamma \Rightarrow (x ; ms) : R} \text{ Choose}$	
$\frac{x : P, \Gamma \Rightarrow m : Q}{\Gamma \Rightarrow \lambda x.m : (P \supset Q)} \text{ Abstract}$	
$\overline{\Gamma \xrightarrow{p} [] : P} \text{ Meet}$	
$\frac{\Gamma \Rightarrow m : P \quad \Gamma \xrightarrow{Q} ms : R}{\Gamma \xrightarrow{p \supset Q} m :: ms : R} \supset S$	
LJ	
$x : P \in \Gamma \quad \overline{\Gamma \rightarrow \text{vr}(x) : P} \text{ L-Axiom}$	
$z : (P \supset Q) \in \Gamma \quad \frac{\Gamma \rightarrow l_1 : P \quad x : Q, \Gamma \rightarrow l_2 : R}{\Gamma \rightarrow \text{app}(z, l_1, x.l_2) : R} \supset L$	
$\frac{x : P, \Gamma \rightarrow l : Q}{\Gamma \rightarrow \lambda x.l : P \supset Q} \supset R$	

Using permutation (app_app1), we have the following single-step term reduction:

$$\frac{\text{app}(\text{vr}(y), \text{vr}(x), v.\text{app}(\text{vr}(z), \text{vr}(x), w.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u))))}{>} \quad z \neq v$$

$$\text{app}(\text{vr}(z), \text{app}(\text{vr}(y), \text{vr}(x), v.\text{vr}(x)), w.\text{app}(\text{vr}(y), \text{vr}(x), v.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u))))$$

which gives us the following single step of reduction of proof trees:

$$\frac{\frac{w : (B \supset C), v : B, \Gamma \quad u : C, w : (B \supset C), v : B, \Gamma}{\rightarrow \text{vr}(v) : B \quad \rightarrow \text{vr}(u) : C}}{v : B, \Gamma \rightarrow x : A \quad w : (B \supset C), v : B, \Gamma \rightarrow \text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u)) : C}}{\frac{\Gamma \rightarrow \text{vr}(x) : A \quad v : B, \Gamma \rightarrow \text{app}(\text{vr}(z), \text{vr}(x), w.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u))) : C}{\Gamma \rightarrow \text{app}(\text{vr}(y), \text{vr}(x), v.\text{app}(\text{vr}(z), \text{vr}(x), w.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u)))) : C}}{>}$$

$$\frac{\frac{\Gamma_3 \rightarrow \text{vr}(v) : B \quad \Gamma_4 \rightarrow \text{vr}(u) : C}{\Gamma_3 \rightarrow \text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u)) : C}}{\Gamma_2 \rightarrow \text{vr}(x) : A \quad \Gamma_3 \rightarrow \text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u)) : C}}{\frac{\Gamma \rightarrow \text{vr}(x) : A \quad \Gamma_1 \rightarrow \text{vr}(x) : A}{\Gamma \rightarrow \text{app}(\text{vr}(y), \text{vr}(x), v.\text{vr}(x)) : A} \quad \Gamma_2 \rightarrow \text{app}(\text{vr}(y), \text{vr}(x), v.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u))) : C}}{\Gamma \rightarrow \text{app}(\text{vr}(z), \text{app}(\text{vr}(y), \text{vr}(x), v.\text{vr}(x)), w.\text{app}(\text{vr}(y), \text{vr}(x), v.\text{app}(\text{vr}(w), \text{vr}(v), u.\text{vr}(u)))) : C}$$

where

$$\begin{aligned} \Gamma &= x : A, y : A \supset B, z : A \supset (B \supset C) \\ \Gamma_1 &= v : B, \Gamma \\ \Gamma_2 &= w : B \supset C, \Gamma \\ \Gamma_3 &= v : B, w : B \supset C, \Gamma \\ \Gamma_4 &= u : C, v : B, w : B \supset C, \Gamma \end{aligned}$$

Figure 2: Permutation of proofs of the S-combinator in LJ

Table 2: Translation functions for proof terms (A)

$$\theta : \mathbf{M} \rightarrow \mathbf{N}$$

$$\begin{aligned} \theta(x ; ms) &=_{\text{def}} \theta'(\text{var}(x), ms) \\ \theta(\lambda x.m) &=_{\text{def}} \lambda x.(\theta(m)) \end{aligned}$$

$$\theta' : \mathbf{A} \times \mathbf{Ms} \rightarrow \mathbf{N}$$

$$\begin{aligned} \theta'(a, []) &=_{\text{def}} \text{an}(a) \\ \theta'(a, m :: ms) &=_{\text{def}} \theta'(\text{ap}(a, \theta(m)), ms) \end{aligned}$$

$$\psi : \mathbf{N} \rightarrow \mathbf{M}$$

$$\begin{aligned} \psi(\text{an}(a)) &=_{\text{def}} \psi'(a, []) \\ \psi(\lambda x.n) &=_{\text{def}} \lambda x.(\psi(n)) \end{aligned}$$

$$\psi' : \mathbf{A} \times \mathbf{Ms} \rightarrow \mathbf{M}$$

$$\begin{aligned} \psi'(\text{var}(x), ms) &=_{\text{def}} (x ; ms) \\ \psi'(\text{ap}(a, n), ms) &=_{\text{def}} \psi'(a, (\psi(n)) :: ms) \end{aligned}$$

$$\bar{\rho} : \mathbf{M} \rightarrow \mathbf{L}$$

$$\begin{aligned} \bar{\rho}(x ; []) &=_{\text{def}} \text{vr}(x) \\ \bar{\rho}(x ; m :: ms) &=_{\text{def}} \text{app}(x, \bar{\rho}(m), z.\bar{\rho}(z ; ms)) && z \text{ new} \\ \bar{\rho}(\lambda x.m) &=_{\text{def}} \lambda x.\bar{\rho}(m) \end{aligned}$$

$$\bar{\phi} : \mathbf{L} \rightarrow \mathbf{M}$$

$$\begin{aligned} \bar{\phi}(\text{vr}(x)) &=_{\text{def}} (x ; []) \\ \bar{\phi}(\text{app}(x, l_1, y.l_2)) &=_{\text{def}} \text{sub}(x, \bar{\phi}(l_1), y, \bar{\phi}(l_2)) \\ \bar{\phi}(\lambda x.l) &=_{\text{def}} \lambda x.\bar{\phi}(l) \end{aligned}$$

$$\text{sub} : \mathbf{V} \times \mathbf{M} \times \mathbf{V} \times \mathbf{M} \rightarrow \mathbf{M}$$

$$\begin{aligned} \text{sub}(x, m, y, (y ; ms)) &=_{\text{def}} (x ; m :: \text{subs}(x, m, y, ms)) \\ \text{sub}(x, m, y, (z ; ms)) &=_{\text{def}} (z ; \text{subs}(x, m, y, ms)) && z \neq y \\ \text{sub}(x, m, y, \lambda z.m') &=_{\text{def}} \lambda z.\text{sub}(x, m, y, m') \end{aligned}$$

$$\text{subs} : \mathbf{V} \times \mathbf{M} \times \mathbf{V} \times \mathbf{Ms} \rightarrow \mathbf{Ms}$$

$$\begin{aligned} \text{subs}(x, m, y, []) &=_{\text{def}} [] \\ \text{subs}(x, m, y, m' :: ms) &=_{\text{def}} \text{sub}(x, m, y, m') :: \text{subs}(x, m, y, ms) \end{aligned}$$

Table 3: Translation functions for proof terms (B)

$\rho : \mathbf{N} \rightarrow \mathbf{L}$ $\rho(n) =_{\text{def}} \bar{\rho}(\psi(n))$
$\phi : \mathbf{L} \rightarrow \mathbf{N}$ $\begin{aligned} \phi(\text{vr}(x)) &=_{\text{def}} \text{an}(\text{var}(x)) \\ \phi(\text{app}(x, l_1, y.l_2)) &=_{\text{def}} [\text{ap}(x, \phi(l_1))/y]\phi(l_2) \\ \phi(\lambda x.l) &=_{\text{def}} \lambda x.\phi(l) \end{aligned}$

and \mathbf{Ms} is defined thus:

$$\begin{aligned} \text{height}(\lambda x.m) &=_{\text{def}} 1 + \text{height}(m) \\ \text{height}([]) &=_{\text{def}} 0 \\ \text{height}(m :: \text{ms}) &=_{\text{def}} 1 + \max(\text{height}(m), \text{height}(\text{ms})). \end{aligned}$$

This function can be used as a measure to prove the termination of $\bar{\rho}$. It is also useful as a well-founded measure for the required strong induction principle.

3. The proof assistant Coq

The system chosen for this formalisation was *Coq* [4], a proof assistant for the *Calculus of Inductive Constructions (CIC)* [9, 39]. The syntax of *Coq* is quite readable, provided that the reader is aware of the conventions used to represent non-ASCII symbols in ASCII text and the basics of the type theory that underlies the system. The main points of the notation used in this paper are noted below.

3.1. Types, sorts, etc.

CIC has two basic sorts: `Prop` and `Set`. Each of these is actually the base of a hierarchy of universes (`Type` and `Typeset` respectively) as in Martin-Löf Type Theory [33]. The hierarchy can be ignored by the user, since the system automatically keeps track of universes above the base cases.

3.2. Definitions

Two basic definition mechanisms are used: `Inductive` (for defining objects and families of sorts `Prop` and `Set`) and `Recursive Definition` (for functions). Thus the definition of natural numbers (`nat`) in *Coq* is as follows. (The number 0 is a reserved token in *Coq*, so the letter `O` is used.)

`Inductive`

```

nat : Set :=
  O : nat |
  S : nat -> nat.
    
```

Table 4: Relationships between the calculi

$\psi_eq1 : \psi(\theta(m)) = m$	
$\psi_eq2 : \psi(\theta'(a, ms)) = \psi'(a, ms)$	
$\theta_eq1 : \theta(\psi(n)) = n$	
$\theta_eq2 : \theta(\psi'(a, ms)) = \theta'(a, ms)$	
$N_Admis_ \theta : \frac{\Gamma \Rightarrow m : R}{\Gamma \gg \theta(m) : R}$	
$N_Admis_ \theta' : \frac{\Gamma > a : P \quad \Gamma \xrightarrow{p} ms : R}{\Gamma \gg \theta'(a, ms) : R}$	
$M_Admis_ \psi : \frac{\Gamma \gg n : R}{\Gamma \Rightarrow \psi(n) : R}$	
$M_Admis_ \psi' : \frac{\Gamma > a : P \quad \Gamma \xrightarrow{p} ms : R}{\Gamma \Rightarrow \psi'(a, ms) : R}$	
$\bar{\phi}_eq1 : \bar{\phi}(\bar{\rho}(m)) = m$	$\rho_eq1 : \rho(\theta(m)) = \bar{\rho}(m)$
$\bar{\phi}_eq2 : \theta(\bar{\phi}(l)) = \phi(l)$	$\rho_eq2 : \phi(\rho(n)) = n$
$L_Admis_ \bar{\rho} : \frac{\Gamma \Rightarrow m : R}{\Gamma \rightarrow \bar{\rho}(m) : R}$	$M_Admis_ \bar{\phi} : \frac{\Gamma \rightarrow l : R}{\Gamma \Rightarrow \bar{\phi}(l) : R}$
$N_Admis_ \phi : \frac{\Gamma \rightarrow l : R}{\Gamma \gg \phi(l) : R}$	$L_Admis_ \rho : \frac{\Gamma \gg n : R}{\Gamma \rightarrow \rho(n) : R}$

Table 5: Permutations of proofs in LJ

(lm)	$\frac{l_1 > l_2}{\lambda x.l_1 > \lambda x.l_2}$	
(app1)	$\frac{l_1 > l_2}{\text{app}(x, l_1, y.l_3) > \text{app}(x, l_2, y.l_3)}$	
(app2)	$\frac{l_2 > l_3}{\text{app}(x, l_1, y.l_2) > \text{app}(x, l_1, y.l_3)}$	
(app_wkn)	$\text{app}(x, l_1, y.l_2) > l_2$	$y \notin l_2$
(app_app1)	$\frac{\text{app}(x, l_1, z.\text{app}(y, l_2, w.l_3)) > \text{app}(y, \text{app}(x, l_1, z.l_2), w.\text{app}(x, l_1, z.l_3))}{}$	$y \neq z$
(app_app2)	$\frac{\text{app}(x, l_1, y.\text{app}(y, l_2, z.l_3)) > \text{app}(x, l_1, y'.\text{app}(y', \text{app}(x, l_1, y.l_2), z.\text{app}(x, l_1, y.l_3)))}{}$	$y' \text{ new}$
(app_lm)	$\text{app}(x, l_1, y.\lambda z.l_2) > \lambda z.\text{app}(x, l_1, y.l_2)$	

Mutual Inductive definitions are allowed using a `Mutual . . . with . . .` construct. The addition function may be defined thus:

Recursive Definition

```
plus:nat->nat->nat :=
  0 j => j |
  (S i) j => (S (plus i j)).
```

Function definition using the `Recursive Definition` syntax is restricted in *Coq* V6.1, the version used for this formalisation, to (higher-order) primitive recursion for single functions. There have been two minor releases since then whose capabilities may have changed in this respect. A more complex definition mechanism allows definition of recursive functions using a fixpoint operator [23] which also allows mutual recursive definitions. All the definitions in this paper can be expressed using the natural syntactic generalisation of `Recursive Definition` to mutual recursion: definitions in this paper will be expressed in this way to enhance readability.

3.3. The minimality principle and inversion of predicates

Inductive definitions in *Coq* are interpreted under a *minimality principle*. That is, when an Inductive definition is made, the object being defined is taken to be the smallest object satisfying the rules as stated in the definition. Thus, if the less-than relation on natural numbers is defined as the propositional function:

Table 6: Subject reduction and weak normalisation

L_Admsis_Perm1 :	$\frac{l_1 \succ l_2 \quad \Gamma \rightarrow l_1 : R}{\Gamma \rightarrow l_2 : R}$
L_Admsis_Permn :	$\frac{l_1 \succ^* l_2 \quad \Gamma \rightarrow l_1 : R}{\Gamma \rightarrow l_2 : R}$
L_Permn_lm	$\frac{l_1 \succ^* l_2}{\lambda x.l_1 \succ^* \lambda x.l_2}$
L_Permn_app1	$\frac{l_1 \succ^* l_2}{\text{app}(x, l_1, y.l_3) \succ^* \text{app}(x, l_2, y.l_3)}$
L_Permn_app2	$\frac{l_2 \succ^* l_3}{\text{app}(x, l_1, y.l_2) \succ^* \text{app}(x, l_1, y.l_3)}$
Norm_Imperm_L :	$\text{Normal}(l) \Rightarrow \sim \exists l_0. l \succ l_0$
Norm_L_ρ̄ :	$\text{Normal}(\bar{\rho}(m))$
App_Red_M :	$\text{app}(x, \bar{\rho}(m_1), y.\bar{\rho}(m_2)) \succ^* \bar{\rho}(\text{sub}(x, m_1, y, m_2))$
Norm_Red :	$l \succ^* \bar{\rho}(\bar{\phi}(l))$

 Table 7: Definition of normality for terms in **L**

l is *normal* if it is
 a variable, or
 of the form $\lambda x.l'$ where l' is normal, or
 of the form $\text{app}(x, l_1, y.l_2)$
 where
 l_1 is normal;
 l_2 is fully normal with respect to the variable y .

l is *fully normal* with respect to x if it is
 equal to x , or
 of the form $\text{app}(x, l_1, y.l_2)$
 where
 l_1 is normal;
 l_2 is fully normal with respect to y ;
 $x \notin l_1, l_2$.

Inductive

```
lt : nat->nat->Prop :=
  lt_0 : (i:nat)(lt 0 (S i)) |
  lt_S : (i, j:nat)(lt i j)->(lt (S i) (S j)),
```

then all the true propositions that are members of this family are built up from a basic fact ($lt\ 0\ (S\ n)$) and a finite sequence of implications incrementing both arguments (lt_S).

Similarly, if we have a premiss ($lt\ i\ j$), then there are only two possibilities for this:

$$i=0 \wedge j=(S\ n) \quad \text{or} \quad i=(S\ m) \wedge j=(S\ n) \wedge (lt\ m\ n).$$

It would be possible to prove this as an *Inversion Lemma*, but this is not necessary, as there is a tactic to perform such a case analysis on a premiss of the current sequent [4, Ch.8].

3.4. Logical notation in ASCII

Lambda abstraction is represented (following AUTOMATH [11]) by square brackets; for example, $[x:A]x$ is the unnamed identity function on a set A.

Universal quantification is represented by parentheses; for example, symmetry of equality in a set A would be stated $(x, y:A)x=y \rightarrow y=x$.

The symbol \rightarrow is used both for function typing and to represent logical implication. Conjunction is represented as \wedge , and disjunction as \vee .

4. Formalisation issues

This section deals with the definitions of the parts of a sequent: the formulae, the context (represented as a list of formulae) and the derivation/deduction terms, followed by the definitions of the families of `Props` representing **LJ** and **MJ** derivations and **NJ** deductions.

4.1. Formulae, contexts, variables and proof terms

The central issue in formalising sequent-style calculi with derivation/deduction terms is the handling of variable bindings and references. There are two different forms of variable occurrence in proof terms: bound and free variables. Much of the benefit from using term calculi is derived from proving theorems about the terms which may then be used in proving theorems about derivations or deductions. For instance, the theorems ψ_eq_1 and ψ_eq_2 show a relationship between the derivation terms of **MJ** and the deduction terms of **NJ**. These theorems are required in order to prove the more interesting theorems $N_Admis_θ$ and $N_Admis_θ'$ regarding derivations in **MJ** and deductions in **NJ**.

This problem of variable binding and references is an old one in computer-aided reasoning. The problems of renaming, α -conversion and substitution have been dealt with in various ways. The most common way of dealing with bound variables in λ -calculi has been with nameless dummy variables, also called *de Bruijn indices* [10]. (In fact, *Coq* itself uses de Bruijn indices internally, together with a persistent naming mechanism for display and interaction.) Another, more recent, idea has been to use a higher-order abstract syntax to define equivalence classes of concrete terms to represent the abstract α -convertible terms required [12, 24]. A similar but simpler approach is outlined in [34] and more thoroughly explained in [35].

This paper presents a formalisation using de Bruijn indices for both the bound and free variables. Similar formalisations of λ -calculi appear in [27, 44]. See Section 5 for a deeper discussion of the alternative approaches.

So, we define a set of formulae (**F**) to contain an infinite set of atomic formulae plus the implicational formulae:

```
Inductive
  F:Set :=
    form : nat->F |
    Impl : F->F->F.
```

The set **Hyps** of contexts is defined as a list of **F**(ormulae) using syntactic constructions together with the polymorphic list library provided with *Cog*: this library provides the function `length:(A:Set)(list A)->nat`, giving the length of lists, and theorems for some of its properties. The syntax for **Hyps** is equivalent to the inductive definition:

```
Inductive
  Hyps:Set :=
    MT : Hyps |
    Add_Hyp : F->Hyps->Hyps.
```

`In_Hyps:nat->F->Hyps->Prop` is a predicate that tests whether a particular formula is present at a particular depth within a specified context, and is the formalisation of the concept $x:P \in \Gamma$. The set **V** of nameless variables is defined as an abbreviation for the natural numbers.

Decidability of equality for **V**, **F** and **Hyps** is proved, as is the decidability of some other relations, such as the occurrence or non-occurrence of a variable in a term. (See [1] for details of how and of why this is important to the formalisation.)

Thus, the proof terms of the three systems may be defined in the following way.

```
Inductive
  L:Set :=
    vr : V->L |
    app : V->L->L->L |
    lm : L->L.
```

```
Mutual Inductive
  M:Set :=
    sc : V->Ms->M |
    lambda : M->M
```

```
with
  Ms:Set :=
    nil : Ms |
    mcons : M->Ms->Ms.
```

```
Mutual Inductive
  N:Set :=
    lam : N->N |
    an : A->N
```

```
with
  A:Set :=
    ap : A->N->A |
    var : V->A.
```

From the definitions for **M** and **Ms** the following induction principle is semi-automatically generated. (Some simple cut-and-paste and an easy proof are currently required for induction principles derived from mutual inductive definitions.)

```
(P : M -> Prop)
(P0 : Ms -> Prop)
  ((v : V) (ms : Ms) (P0 ms) -> (P (sc v ms))) ->
  ((m : M) (P m) -> (P (lambda m))) ->
  (P0 mnil) ->
  ((m : M) (P m) -> (ms : Ms) (P0 ms) -> (P0 (mcons m ms))) ->
  (((m : M) (P m)) /\ ((ms : Ms) (P0 ms))).
```

In the more standard logical notation this is the induction scheme:

$$\frac{\begin{array}{l} \forall x : \mathbf{V}. \forall ms : \mathbf{Ms}. P_0(ms) \supset P(x ; ms) \\ \forall x : \mathbf{V}. \forall m : \mathbf{M}. P(m) \supset P(\lambda x. m) \\ P_0([\]) \\ \forall m : \mathbf{M}. P(m) \supset \forall ms : \mathbf{Ms}. P_0(ms) \supset P_0(m :: ms) \end{array}}{(\forall m : \mathbf{M}. P(m)) \wedge (\forall ms : \mathbf{Ms}. P_0(ms))}$$

where P and P_0 are predicates on **M** and **Ms** respectively.

4.2. Derivations and deductions

We may now define the predicates representing derivations and deductions within the three calculi. In type theory this is done by defining predicates which map a context, a term and a formula to a `Prop` (with an extra formula added for the stoup in **Ms** derivations for **MJ**). Each rule/axiom (as shown in Table 1) provides a constructor clause for these inductively defined predicates. In this way we formalise the requirement that a valid derivation or deduction must be built up only from the primitive rules of the object logic, while allowing reasoning about the systems at the meta-level. For brevity we show this only for **MJ** in Figure 3.

As may be seen from those definitions, the de Bruijn indexing flows seamlessly between terms in sequents which change the context and the binding operators at the head of the term (`Abstract` rule). This is also the case for **NJ** and **LJ** as defined in this formalisation. This is due to the particular nature of the logics in question, and does not necessarily generalise to other logics. In particular, a formalisation of linear logic would not work in this fashion, and a more complex variable-referencing mechanism would be required. See Section 6 for a further discussion of this problem. Other operations, such as substitutions (`sub` in Table 2) and weakening, require `lift` and `drop` operations as defined in [27] to ensure the correctness of the de Bruijn indexing.

Figure 4 shows the induction scheme semi-automatically produced for derivations in **MJ**. This complicated scheme is not strictly necessary, although its use does simplify proofs somewhat. The inversion tactic of `Coq` provides us with a relatively simple way to avoid using induction schemes based on the derivation/deduction predicate, but only because derivation/deduction terms so closely encode the derivation/deduction. The application of this scheme is equivalent to using the mutual induction principle for the terms **M** and **Ms**, and then performing an inversion on the derivation in the antecedent of the current goal, for example inverting:

```
(M_Deriv h (lambda m) (Impl P Q))
to give the premiss: (M_Deriv (Add_Hyp P h) m Q).
```

Mutual Inductive

```

M_Deriv : Hyps -> M -> F -> Prop :=
  Choose : (h:Hyps)(i:V)(P:F)(ms:Ms)(R:F)
    (In_Hyps i P h)->
    (Ms_Deriv h P ms R)->
    (M_Deriv h (sc i ms) R) |
  Abstract :
    (h:Hyps)(P:F)(m:M)(Q:F)
    (M_Deriv (Add_Hyp P h) m Q)->
    (M_Deriv h (lambda m) (Impl P Q))

```

with

```

Ms_Deriv : Hyps -> F -> Ms -> F -> Prop :=
  Meet : (h:Hyps)(P:F)
    (Ms_Deriv h P mnil P) |
  Implies_S :
    (h:Hyps)(m:M)(P:F)(Q:F)(ms:Ms)(R:F)
    (M_Deriv h m P)->
    (Ms_Deriv h Q ms R)->
    (Ms_Deriv h (Impl P Q) (mcons m ms) R).

```

Figure 3: Formal definition of derivations in **MJ**

While it is difficult to see how one might directly define an induction on derivation height, one may perform such inductions indirectly by using a strong induction on terms, followed by case analyses and inversions. Machine support for such methods may not seem necessary for the implicational fragment presented in this paper, but the complexity of the induction schemes rapidly increases when new connectives are added, and other aspects of the machine support (such as tracking of case analyses) become ever more useful.

So, while induction schemes of such size are not strictly necessary, consider the formalisation extended to full propositional logic. The induction scheme for the derivation/deduction terms will be even larger than that shown in Figure 4 for derivations, and the use of such schemes followed by inversions performed on the antecedents would be time-consuming and awkward. So where straightforward structural induction is sufficient, it is useful to be able to perform this directly on the derivations/deductions, rather than via induction on the derivation/deduction terms, followed by inversion of premisses.

4.3. Permutation

Table 5 shows the permutations in the usual informal syntax. The formalisation of these rules was complex. The exact variable namings and renamings that form an integral part of the reductions are subtle, and it is only when one looks at the typed case that one can fully decipher the meanings of the reductions and formalise them to capture the correct translations. Figure 5 shows the formalised version of two of the actual permutations; that is, rules that permute proof terms rather than the rules allowing permutation within subterms. (See (app_lm) in Table 5 for an example of a rule allowing permutation of subterms.)

The formalisation of (app_app2) as `l_perm1_app_app2` highlights the complexity of the process. Figure 6 shows the informal version of the typed reduction rule. Only the leaves and root of the relevant proof-tree fragments are shown, since they contain all the information necessary for the analysis.

```

(P:(h:Hyps)(m:M)(f:F)(M_Deriv h m f)->Prop)
(P0:(h:Hyps)(f:F)(ms:Ms)(f0:F)(Ms_Deriv h f ms f0)->Prop)
  ((h:Hyps)(i:V)(f2:F)(ms:Ms)(R:F)
    (i0:(In_Hyps i f2 h))
    (ms0:(Ms_Deriv h f2 ms R))
    (P0 h f2 ms R m)->
    (P h (sc i ms) R (Choose h i f2 ms R i0 ms0)))->
((h:Hyps)(f2:F)(m:M)(Q:F)
  (m0:(M_Deriv (Add_Hyp f2 h) m Q))
  (P (Add_Hyp f2 h) m Q m0)->
  (P h (lambda m) (Impl f2 Q) (Abstract h f2 m Q m0)))->
((h:Hyps)(f2:F)(P0 h f2 mnil f2 (Meet h f2)))->
((h:Hyps)(m:M)(f2, Q:F)(ms:Ms)(R:F)
  (m0:(M_Deriv h m f2))
  (P h m f2 m0)->
  (ms0:(Ms_Deriv h Q ms R))
  (P0 h Q ms R ml)->
  (P0 h (Impl f2 Q) (mcons m ms) R
    (Implies_S h m f2 Q ms R m0 ms0)))->
((h:Hyps)(m:M)(f:F)(m0:(M_Deriv h m f))(P h m f m0))\
((h:Hyps)(f:F)(m:Ms)(f0:F)(ms0:(Ms_Deriv h f m f0))
  (P0 h f m f0 ms0)).

```

Figure 4: Formal induction scheme for derivations in MJ

Each of the leaves of a tree corresponds to a particular occurrence of a named term (variable or term of \mathbf{L} : x , y , y' , l_1 , l_2 , l_3) in the root of that tree. So, for each of the three different occurrences of the terms l_1 and x in the root of the second tree there is a leaf with l_1 or x as the principal term. A comparison of the contexts of these leaves with the original leaf in the first tree shows the differences in the de Bruijn indices for the terms. Thus the first occurrences of x and l_1 are unchanged in the formalisation of the reduction, the second occurrences are both lifted once, and the third occurrences are lifted twice.

The most complex variations in the contexts occur for l_3 : originally the bindings for variables are $z : P_2, y : (P_1 \supset P_2), \Gamma$; that is, de Bruijn index occurrences in l_3 are dereferenced as $z \leftarrow 0; y \leftarrow 1$, and higher numbers reference entries in the context Γ . In the permuted derivation the bindings are $y : (P_1 \supset P_2), z : P_2, y' : (P_1 \supset P_2), \Gamma$. Since y' does not appear in l_3 , but must be accounted for in the referencing to variables in Γ , l_3 must be lifted by 2 (that is, $(S(S\ O))$). Also, the occurrences of y and z are switched, so the de Bruijn references must be exchanged — exchange is defined only for switching references to a binding depth and its successor. This may be done without loss of generality, since any general exchange can be expressed in terms of multiple applications of this pairwise exchange. Similar analyses give us the lifting, dropping and exchanging requirements for each permutation as shown in Figure 5. The admissibility of various structural rules has been proved in the formalisation for all three systems. While strengthening, weakening and exchange are all obviously admissible for all three systems, this has only been formally proved where required for other results.

```

Inductive
  L_Perm1 : L->L->Prop :=
    :
    :
  l_perm1_app_wkn :
    (x:V)(l1, l2:L)
    ~ (Occurs_In_L 0 l2)->
    (L_Perm1 (app x l1 l2) (drop_L 0 l2)) |
    :
    :
  l_perm1_app_app2 :
    (x:V)(l1, l2, l3:L)
    (L_Perm1 (app x l1 (app 0 l2 l3))
      (app x
        l1
        (app 0
          (app (lift_V 0 x)
            (lift_L 0 l1)
            (lift_L (S 0) l2))
          (app (lifts_V (S (S 0)) 0 x)
            (lifts_L (S (S 0)) 0 l1)
            (L_Exchange 0
              (lift_L (S (S 0)) l3))))))) |
    :
    :

```

Figure 5: Formalised permutations

4.4. Non-primitive recursive definitions

The definition of $\bar{\rho}$ in Table 2 is not primitive recursive when directly translated into the formal syntax. This is due to the recursive call of $\bar{\rho}(z; ms)$ for new z in the third definitional equation. The argument of this recursive call is not a sub-term of $(x; m :: ms)$, the original argument. When such definitions are translated into the formal syntax, new non-primitive recursive elements may also be added in the form of lifting and/or dropping operations on the terms passed into the recursive calls.

The direct translation of $\bar{\rho}$ (from Table 2) would be:

Recursive Definition

```

rhobar : M->L :=
  (semicolon x nil) => (vr x) |
  (semicolon x (mcons m ms)) =>
    (app x (rhobar m)
      (rhobar (semicolon 0
        (lift_Ms 0 ms)))) |
  (lambda m) => (lm (rhobar m)).

```

Since `app` is a binding operator for its third argument, `0` represents a new variable, and `ms` has been lifted to take this binding into account. The following definition is used, and

$$\begin{array}{c}
 (z : P_2), (y : (P_1 \supset P_2)), \Gamma \rightarrow l_3 : R \\
 (y : (P_1 \supset P_2)), \Gamma \rightarrow l_2 : P_1 \\
 \Gamma \rightarrow l_1 : P_0 \\
 (x : (P_0 \supset (P_1 \supset P_2))) \in \Gamma \\
 \vdots \\
 \Gamma \rightarrow \text{app}(x, l_1, y.\text{app}(y, l_2, z.l_3)) : R \\
 \\
 > \\
 (y : P_1 \supset P_2), (z : P_2), (y' : (P_1 \supset P_2)), \Gamma \rightarrow l_3 : R \\
 (z : P_2), (y' : (P_1 \supset P_2)), \Gamma \rightarrow l_1 : P_0 \\
 (x : (P_0 \supset (P_1 \supset P_2))) \in (z : P_2), (y' : (P_1 \supset P_2)), \Gamma \\
 (y : (P_1 \supset P_2)), \Gamma \rightarrow l_2 : P_1 \\
 (y' : (P_1 \supset P_2)), \Gamma \rightarrow l_1 : P_0 \\
 (x : (P_0 \supset (P_1 \supset P_2))) \in (y' : (P_1 \supset P_2)), \Gamma \\
 (y' : (P_1 \supset P_2)) \in (y' : (P_1 \supset P_2)), \Gamma \\
 \Gamma \rightarrow l_1 : P_0 \\
 (x : (P_0 \supset (P_1 \supset P_2))) \in \Gamma \\
 \vdots \\
 \Gamma \rightarrow \text{app}(x, l_1, y'.\text{app}(y', \text{app}(x, l_1, y.l_2), z.\text{app}(x, l_1, y.l_3))) : R
 \end{array}$$

Figure 6: Permutation of proof tree fragments by app_app2

proved equivalent to the above by proving that the three parts of the above definition hold; that is, we prove that $(x : V) ((\text{semicolon } x \text{ nil}) = (\text{vr } x))$, and so on.

Recursive Definition

```

rhubar : M->L :=
  (semicolon x nil) => (vr x) |
  (semicolon x (mcons m ms)) =>
    (app x (rhubar m) (rhubar' (S O) ms)) |
  (lambda m) => (lm (rhubar m))

```

with

```

rhubar' : nat->Ms->L :=
  i nil => (vr O) |
  i (mcons m ms) =>
    (app O
      (lifts_L i O (rhubar m))
      (rhubar' (S i) ms)).

```

The operation `lifts_L` is merely a multiple iteration of the usual lifting operation (as defined in [27]). In this case $(\text{rhubar } m)$ is lifted i times with respect to O . Effectively, the lifting operations are being pushed up the evaluation tree until they no longer interfere with the primitive recursive nature of the definition. A similar, but slightly more complicated, procedure is used to produce a primitive recursive version of `sub`, pushing the actual substitution outside the primitive recursive calls. Note that as mentioned in Section 3.2 the Recursive Definition format in Coq V6.1 does not allow mutual recursive definitions, so the actual formalisation uses the more primitive (and more difficult to read) Fixpoint format.

4.5. Formal proof of App_Red_M

As mentioned in Section 2.5, the proof of App_Red_M (amongst others) uses strong induction on the height of terms in \mathbf{M} and \mathbf{Ms} . The formalised strong induction principle is:

```
(P:M->Prop)
(P0:Ms->Prop)
((m:M)((m1:M)(lt (Height_M m1) (Height_M m))->
  (P m1))
 /\((ms1:Ms)(lt (Height_Ms ms1) (Height_M m))->
  (P0 ms1))->
  (P m))->
((ms:Ms)((ms1:Ms)(lt (Height_Ms ms1) (Height_Ms ms))->
  (P0 ms1))
 /\((m1:M)(lt (Height_M m1) (Height_Ms ms))->
  (P m1))->
  (P0 ms))->
  ((m:M)(P m)) /\ ((ms:Ms)(P0 ms)).
```

In this case, we are trying to prove the theorem:

$$\text{app}(x, \bar{\rho}(m_1), y, \bar{\rho}(m_2)) \succ^* \bar{\rho}(\text{sub}(x, m_1, y, m_2))$$

which may be proved by strong induction on m_2 , without requiring any equivalent property for an object of type \mathbf{Ms} . To allow the use of mutual induction properties generally, a goal of the appropriate type is required. Such a goal is produced by defining propositional functions with the correct type. Since, in the case of App_Red_M, the second conjunct is not needed for proof of the first, a trivially true propositional function is defined. The two propositional functions defined here are:

Definition

```
app_red_m : M->Prop :=
  [m:M] (x:V) (m1:M)
    (L_Permn (app x (rhopar m1) (rhopar m))
      (rhopar (MsubstVMV x m1 0 m))).
```

Definition

```
app_red_ms : Ms->Prop :=
  [ms:Ms] (x,y:V) (m1:M)
    (L_Permn (app x (rhopar m1) (rhopar (sc y ms)))
      (app x (rhopar m1) (rhopar (sc y ms)))).
```

The goal $((m:M)(\text{app_red_m } m)) \wedge ((ms:Ms)(\text{app_red_ms } ms))$ is therefore appropriate for the application of the strong mutual induction principle above, while unfolding the definitions of app_red_m and app_red_ms puts the induction hypotheses and goals into forms amenable to proof.

Similar procedures are needed for all proofs involving mutual inductions. Often, a proof about, say, only an object in \mathbf{M} is required, and the appropriate second proposition for objects in \mathbf{Ms} must be synthesised in order to allow the proof to proceed. Examples of this can be found in Table 4. The theorems ψ_{eq2} and θ_{eq2} are not of real interest in themselves, yet they provide vital induction hypotheses for the proofs of ψ_{eq1} and θ_{eq1} .

5. Related work

Formalisations of this kind are still rare, and in those that exist there is a wide variety of approaches to the subject. This formalisation has used nameless dummy variables and functions, as opposed to various schemes for named variables and the use of functional relations rather than functions.

Recent formalisations such as those of Nazareth and Nipkow [38], Altenkirch [2] and Barras [3] employ similar methods to the ones highlighted here. Although the methods of [2] are similar to those presented here, only a single logic, System F, was formalised. System F is a more powerful and complicated logic than **NJ** etc., so different issues are addressed: those of formalising a single powerful logic rather than the relationships between multiple (simpler) logics. The main differences in approach are those dictated by the differences between the LEGO system (pre-1993) and the *Coq* system (in 1997).

Coquand [8] presents another formalisation of a normalisation proof, again using a de Bruijn encoding, although the details of the proof are very dependent on the capabilities and style of *Alf*, the system used for the formalisation.

Pfenning [41] takes a very different approach to the problem of representing sequent calculus in a form suitable for formal meta-theoretic reasoning, a form of higher-order abstract syntax, similar to the methods of Despeyroux, Felty and Hirschowitz [12] using *Coq*. For this work, the definitions of proof terms use the variables of the framework or proof assistant itself (respectively Elf [40] and *Coq*) to encode the variables of the logic being studied. This approach has obviously met with some success, although the underlying ideas can be quite difficult to grasp, and must be approached separately for each proof assistant. In *Coq* particularly, there is a definite trade-off to be made in order to use a higher-order abstract syntax. In encoding the terms by using the underlying system, one first defines pseudo-terms and then defines a predicate narrowing the scope to those terms that one wishes to allow. Even these terms are not fully specified, but are equivalence classes of terms that are indistinguishable in judgements. One cannot define functions over the narrowed terms, but instead must use functional relations; the intensional equality of the system must be discarded and an extensional equality relation defined and used. A more significant problem is that one loses the automatic production of provably correct induction schemes. These added complexities would appear to separate the formalisation from the original informal theory quite heavily. So, while one gains confidence in the results, perhaps, one may not gain confidence in the informal proofs.

There has been an increasing amount of research into higher-order abstract syntax for various application areas in the last few years, some of the most notable work having been published since the development presented here was completed. In particular, recent work by Gabbay and Pitts [20] suggests the use of a non-standard set theory (the Fraenkel–Mostowski permutation model of set theory with atoms) which allows the definition of languages with binders. This approach gives α -conversion of terms involving bound variables ‘for free’, but does not give substitution ‘for free’. The central theoretical ideas are similar to those of [19]. Current development of this idea is being performed in *Coq*.

McKinna and Pollack [34] introduce yet another approach to such formalisations. McKinna refers to this method as *first-order abstract syntax* for terms with (restricted) *higher-order abstract syntax* for judgements. McKinna has claimed, in a personal communication with the author, that it should be possible to use such a syntax for a different formalisation in *Coq* of the theory presented here, using functional relations instead of *Coq* functions to define the translation and substitution functions, etc., while still allowing use of the *Coq*

intensional equality for theorems such as ψ_eq1 . This possibility remains unexplored at present. Certainly the work on algorithms for checking Pure Type Systems has a number of similarities with the work presented in this paper (see also [35]).

6. Conclusions and further work

Following the proof of strong normalisation for a system of reductions by Schwichtenberg in [45], weak normalisation was formally proved using the conditional variants for which strong normalisation holds. Very little work was required to re-do these proofs with the extra conditions, indicating the robustness of *Coq*'s proof-scripting mechanisms. Formalisation of Schwichtenberg's proof, however, would have required the development of yet another variant of the logics, still a formidable undertaking despite the experience gained in formalising the other three systems.

This paper has presented a formalisation of sequent-style calculi using de Bruijn indices; that is, nameless variables. It has shown that it is possible to follow standard informal proof techniques in a formal environment using this representation. The areas where the formal development differs from the informal development is precisely where the informal development makes use of assumptions about variable renaming. While other formalisations such as the Coquand–McKinna–Pollack abstract syntax may allow named variables, and consequently easier human-readable proofs, there is a consequent loss of equivalence between informal and formal developments where functions are used. Current implementations of higher-order abstract syntax are difficult to use, and still seldom allow for the automatic production of induction schemes within the restrictions of tools such as *Coq*. In the longer term, however, this approach could well bear productive fruit, although it is impossible to judge at this stage how soon tools might become available to allow simple higher-order abstract syntax and the automatic production of induction schemes.

The use of de Bruijn indices in the formalisation brings its own problems to the work, notably the problems with primitive recursive definition highlighted in Section 4.4, and an increase in the number of intermediate lemmas (in particular, proving relationships between functions and renumbering of nameless variables). There is also the matter of the gap between the informal and formal definitions. This gap may undermine the confidence in the informal proofs gained from a formalisation of the proofs. The wider gap between informal and formal definitions produced by the current methods of first- or higher-order abstract syntax would seem to indicate that de Bruijn nameless variables would be the best (though not an ideal) choice for a formalisation of [15], which extends the proofs to full propositional logic. An extension of the informal proofs to first-order logic would produce such a large informal proof that confidence in those informal proofs could be quite low. It was with this in mind that the original formalisation was undertaken. As we have seen, however, the effort involved in formalising such proofs is still considerable, although extension of an existing development to more connectives would hopefully prove less onerous than the initial task. Should such a task be undertaken, it would be likely that work on automation of routine aspects of the proofs would prove beneficial. For instance, the formal proofs show similarities to those for which *rippling*[6] is a successful approach. However, the existing implementation of *rippling* is only useful for equational reasoning, although the background for a more general form, *relational rippling* [5], does exist.

Similar problems exist in other logics, most notably [21] on permutations in linear logic. As mentioned above, the formalisation presented here would not be completely appropriate for work with linear logic. The problems come when dealing with a term calculus and its

interactions with the context-splitting rules:

$$\frac{\Gamma \vdash g_1 : G_1 \quad \Delta \vdash g_2 : G_2}{\Gamma, \Delta \vdash g : G}$$

where the context in the conclusion of the rule (Γ, Δ) is split into portions (Γ and Δ) for the premisses. References to the free variables of the term (that is, the variables referencing formulae in the context) may be radically altered. Some possible ways of avoiding this problem involve amending the context in some way, barring certain formulae from use in a branch of the proof tree, or encoding the variables as binary trees of integers rather than simply as integers. It is difficult to judge in advance whether such encodings would justify their use, as compared with moving to an abstract syntax with the concomitant other changes mentioned above.

Coq is only one of the systems available, and other systems of similar power and maturity (for example, HOL[25] and *NuPRL*[7]) may bring different strengths and weaknesses to bear on such problems. In particular, it is possible that the extensional equality mechanisms in HOL might bring an abstract syntax closer to the original informal development. One problem with this idea, however, is that meta-theory of logical systems is usually performed in an intuitionistic setting (whether the object logic under consideration is intuitionistic or classical), so the classical logic of HOL may not be appropriate.

Appendix A. The full *Coq* 6.1 development

This appendix contains a README file and a tarred gzipped file containing the full formal development described in this paper, and is to be found at

<http://www.lms.ac.uk/jcm/3/lms1999-009/appendix-a/>.

The formal development was performed with Version 6.1 of the *Coq* proof system, and may not work with more recent releases. Instructions for running the formal proofs with *Coq* 6.1 are included in the README file.

References

1. A. A. ADAMS, ‘Tools and Techniques for Machine-Assisted Meta-Theory’, PhD Thesis, School of Mathematical and Computational Sciences, University of St Andrews, UK, 1997. 2, 14
2. TH. ALTENKIRCH, ‘A formalisation of the strong normalisation proof for System F in LEGO’, *Typed lambda calculus and applications*, Lecture Notes in Computer Science 664 (ed. M. Bezem and J. F. Groote, Springer-Verlag, 1993) pp. 13–28. 1, 21, 21
3. B. BARRAS, ‘*Coq en Coq*’, Tech. Rep. 3026, INRIA (1996). 21
4. B. BARRAS, S. BOUTIN, C. CORNES, J. COURANT, J.C. FILLIATRE, E. GIMÉNEZ, H. HERBELIN, G. HUET, C. MUÑOZ, C. MURTHY, C. PARENT, C. PAULIN, A. SAÏBI and B. WERNER, ‘The *Coq* Proof Assistant Reference Manual (Version 6.1)’, Tech. Rep., INRIA, 1996. Available on-line with the *Coq* distribution from <ftp.inria.fr>. 9, 13
5. A. BUNDY and V. LOMBART, ‘Relational rippling: a general approach’, Proceedings, 14th International Joint Conference on Artificial Intelligence (ed. C. Mellish, IJCAI, 1995) 175–181. 22

6. A. BUNDY, A. STEVENS, F. VAN HARMELEN, A. IRELAND and A. SMAILL, ‘Rippling: a heuristic for guiding inductive proofs’, *Artificial Intelligence* (1993) 185–253. 22
7. R. L. CONSTABLE, S. F. ALLEN and OTHERS, *Implementing Mathematics with the NuPrI proof development system* (Prentice-Hall, 1986). 1, 23
8. C. COQUAND, *From semantics to rules: a machine assisted analysis*, Lecture Notes in Computer Science 832 (Springer-Verlag, 1993) 91–105. 1, 21
9. TH. COQUAND and G. HUET, *Constructions: a higher order proof system for mechanizing Mathematics*, Lecture Notes in Computer Science 203 (Springer-Verlag, 1985) 151–184. 9
10. N. G. DE BRUIJN, ‘ λ -calculus notation with nameless dummies, a tool for automatic formula manipulation’, *Indag. Math* 34 (1972) 381–392. 13
11. N. G. DE BRUIJN, *A survey of the project AUTOMATH* (Academic Press, 1980) 579–606. 13
12. J. DESPEYROUX, A. FELTY and A. HIRSCHOWITZ, *Higher-order abstract syntax in Coq* Lecture Notes in Computer Science 902 (Springer-Verlag, 1995) 124–138. 13, 21
13. P. DYBJER, B. NORDSTRÖM and J. SMITH (eds), *Types for proofs and programs*, Proceedings, International Workshop TYPES ’94, Lecture Notes in Computer Science (Springer-Verlag, 1994).
14. R. DYCKHOFF, ‘Contraction-free sequent calculi for intuitionistic logic’, *J. Symbolic Logic* 57 (1992) 795–807. 3
15. R. DYCKHOFF and L. PINTO, ‘A permutation-free sequent calculus for intuitionistic logic’, Tech. Rep. CS/96/9, University of St Andrews, 1996. 2, 3, 3, 3, 3, 3, 4, 22
16. R. DYCKHOFF and L. PINTO, ‘Cut-elimination and Herbelin’s sequent calculus for intuitionistic logic’, *Studia Logica* 60 (1998) 107–118. 3
17. R. DYCKHOFF and L. PINTO, ‘Permutability of proofs in intuitionistic sequent calculi’, *Theoret. Comput. Sci.* 212 (1999) 141–155. 2, 2, 3, 3
18. A. FELTY, *A logic program for transforming sequent proofs to natural deduction proofs*, Lecture Notes in Artificial Intelligence 475 (Springer-Verlag, 1989) 157–178. 3
19. M. FIORE, G. PLOTKIN and D. TURI, ‘Abstract syntax and variable binding’, Proceedings, 14th Annual Symposium on Logic in Computer Science (ed. G. Longo, IEEE Computer Society Press, Washington, 1999). 21
20. M. J. GABBAY and A. M. PITTS, ‘A new approach to abstract syntax involving binders’, Proceedings, 14th Annual Symposium on Logic in Computer Science (ed. G. Longo, IEEE Computer Society Press, Washington, 1999). 21
21. D. GALMICHE and G. PERRIER, ‘On proof normalisation in linear logic’, *Theoret. Comput. Sci.* 135 (1994) 67–110. 22
22. G. GENTZEN, ‘Investigations into logical deduction’ *The collected papers of Gerhard Gentzen* (Translated from 1934 original in German), Studies in Logic and the Foundations of Mathematics (ed. M. E. Szabo, North-Holland, 1969) 68–131. 3
23. E. GIMINEZ, ‘Codifying guarded definitions with recursive schemes’, *Types for proofs and programs*, Proceedings, International Workshop TYPES ’94, Lecture Notes in Computer Science (ed. P. Dybjer, B. Nordström and J. Smith, Springer-Verlag, 1994) 39–59. 11

24. A. D. GORDON and T. MELHAM, ‘Five axioms of alpha-conversion’, *Theorem proving in higher order logics*, Proceedings, 9th International Conference, Lecture Notes in Computer Science 1125 (ed. J. von Wright, J. Grundy and J. Harrison, Springer-Verlag, 1996) 173–190. 13
25. M. J. C. GORDON and T. F. MELHAM (eds), *Introduction to HOL* (Cambridge University Press, 1993). 23
26. H. HERBELIN, *A λ -calculus structure isomorphic to Gentzen-style sequent calculus structure*, Lecture Notes in Computer Science 933 (Springer-Verlag, 1994) 61–75. 3, 3
27. G. HUET, ‘Residual theory in λ -calculus: a complete Gallina development’, *J. Funct. Programming* 3 (1994) 371–394. 1, 13, 15, 19
28. G. HUET and G. PLOTKIN (eds), *Logical frameworks* (Cambridge University Press, 1991). 1
29. G. HUET and G. PLOTKIN (eds), *Logical environments* (Cambridge University Press, 1993). 1
30. S. C. KLEENE, ‘Permutability of inferences in Gentzen’s calculi LK and LJ’, *Mem. Amer. Math. Soc.* (1952) 1–26. 2
31. J. W. KLOP, *Term rewriting systems* (Oxford University Press, 1992) 1–116. 4
32. D. LEIVANT, ‘Assumption classes in natural deduction’, *Zeitschrift für Math. Logik* 25 (1979) 1–4. 3
33. P. MARTIN-LÖF, *Intuitionistic type theory* (Bibliopolis, 1984). 9
34. J. MCKINNA and R. POLLACK, ‘Pure type systems formalized’, *Typed lambda calculus and applications*, Lecture Notes in Computer Science 664 (ed. M. Bezem and J. F. Groote, Springer-Verlag, 1993) 289–305. 13, 21
35. J. H. MCKINNA and R. POLLACK, ‘Some type theory and lambda calculus formalised’, *J. Automat. Reason.*, Special Issue on Formal Proof, ed. F. Pfenning, 23 (1999) 373–409. 13, 22
36. G. MINTS, ‘Cut-elimination and normal forms of sequent derivations’, Tech. Rep. CSLI-94-193, Stanford University, 1994. 3
37. G. MINTS, *Normal forms for sequent derivations* (A. K. Peters, Wellesley, MA, 1996) 469–492. 3
38. D. NAZARETH and T. NIPKOW, ‘Formal verification of Algorithm W: the monomorphic case’, *Theorem proving in higher order logics*, Proceedings, 9th International Conference, Lecture Notes in Computer Science 1125 (ed. J. von Wright, J. Grundy and J. Harrison, Springer-Verlag, 1996) 331–345. 21
39. C. PAULIN-MOHRING, ‘Inductive definitions in the system *Coq*. Rules and properties’, *Typed lambda calculus and applications*, Lecture Notes in Computer Science 664 (ed. M. Bezem and J. F. Groote, Springer-Verlag, 1993) 328–345. 9
40. F. PFENNING, ‘Logic programming in the LF logical framework’, *Logical environments* (ed. G. Huet and G. Plotkin, Cambridge University Press, 1993) 149–181. 21
41. F. PFENNING, ‘A structural proof of cut elimination and its representation in a logical framework’, Tech. Rep. CMU-CS-94-218, Carnegie Mellon University (1994). 21
42. F. PFENNING and E. ROHWEDDER, *Implementing the meta-theory of deductive systems*, Lecture Notes in Artificial Intelligence 607 (Springer-Verlag, 1992) 537–551. 1

43. D. PRAWITZ, ‘Natural deduction’, Ph.D. thesis, Acta Universitatis Stockholmensis, 1965. 3, 3
44. A. SAÏBI, ‘Formalization of a λ -calculus with explicit substitutions in Coq’, *Types for proofs and programs*, Proceedings, International Workshop TYPES '94, Lecture Notes in Computer Science (ed. P. Dybjer, B. Nordström and J. Smith, Springer-Verlag, 1994) 183–202. 13
45. H. SCHWICHTENBERG, ‘Termination of permutative conversions in intuitionistic Gentzen calculi’, *Theoretical Comput. Sci.* 212 (1999) 247–260. 22
46. N. SHANKAR, *Metamathematics, machines, and Gödel’s proof*, Cambridge Tracts in Theoretical Computer Science (Cambridge University Press, 1994). 1
47. M. E. SZABO (ed.), *The collected papers of Gerhard Gentzen* (Translated from 1934 original in German), Studies in Logic and the Foundations of Mathematics (North-Holland, 1969). 1
48. A. S. TROELSTRA and H. SCHWICHTENBERG, *Basic proof theory* (Cambridge University Press, 1996). 3
49. L. S. VAN BENTHEM JUTTING, J. MCKINNA and POLLACK R., ‘Checking algorithms for pure type systems’, Lecture Notes in Computer Science 806 (Springer-Verlag, 1994) 19–61. 1

A. A. Adams aaa@dcs.st-and.ac.uk

Division of Computer Science
School of Mathematics and Computer Science
University of St Andrews
North Haugh
St Andrews KY16 9SS