

# Meta-Theory of Sequent-Style Calculi in *Coq*

A. A. Adams \*

May 22, 1997

## Abstract

We describe a formalisation of proof theory about sequent-style calculi, based on informal work in [DP96]. The formalisation uses de Bruijn nameless dummy variables (also called de Bruijn indices) [dB72], and is performed within the proof assistant *Coq* [BB<sup>+</sup>96]. We also present a description of some of the other possible approaches to formal meta-theory, particularly an abstract named syntax and higher order abstract syntax.

## 1 Introduction

Formal proof has developed into a significant area of mathematics and logic. Until recently, however, such proofs have concentrated on proofs within logical systems, and meta-theoretic work has continued to be done informally. Recent developments in proof assistants and automated theorem provers have opened up the possibilities for machine-supported meta-theory. This paper presents a formalisation of a large theory comprising of over 200 definitions and more than 500 individual theorems about three different deductive systems.<sup>1</sup> The central difficulty in formal meta-theoretic proofs is the treatment of variable names in terms and contexts. There are three main approaches: de Bruijn indices [dB72], named abstract syntax [vBJMR94] and higher order abstract syntax [Pfe91]. The work presented here uses de Bruijn indices.

A coherent formalisation of the three systems as typed lambda calculi is presented in §3, together with statements of the main theorems of interest in both informal and formal syntax. §4 highlights some of the particular proof methods used in these proofs. Some basic choices about the method of formalisation are discussed in §5. §6 discusses the possibilities for extensions of the formalisation presented here. Finally, §7 draws some conclusions about the state of machine-support for meta-theory. We begin in §2 with an overview of *Coq* [BB<sup>+</sup>96], the system used in this formalisation.

### 1.1 Terminology

When discussing meta-theoretical results, the word ‘proof’ can become extremely overused. In order to avoid this, the following nomenclature will be used: ‘proof’ will refer to the proof of a meta-theorem; ‘derivation’ will be used to refer to the proofs in the two sequent calculi **MJ** and **LJ**; ‘deduction’ will be used to refer to proofs in the sequent-style natural deduction calculus **NJ**.

---

\*This work has been supported by the EC (BRA 7232, working group “GENTZEN”) and the UK’s EPSRC.

<sup>1</sup>Only the implicational fragments have been studied. Extensions to full propositional logic are unlikely to present further challenges to the approach shown here.

Some readers may need a definition of ‘*unfolding*’ as used in later sections. *Unfolding* is a process which takes a function application such as  $f(a, b)$  and replaces it with the body of the definition of  $f$ . So, if we have the function *plus* for natural numbers defined by the equations:

$$\begin{aligned} \text{plus}(0, n) &=_{\text{def}} n \\ \text{plus}(S(m), n) &=_{\text{def}} S(\text{plus}(m, n)) \end{aligned}$$

then unfolding the first application of *plus* in

$$\text{plus}(S(S(0)), \text{plus}(S(i), j))$$

gives

$$S(\text{plus}(S(0), \text{plus}(S(i), j)))$$

## 2 The Proof Assistant *Coq*

The system chosen for this formalisation was *Coq* [BB<sup>+</sup>96], a proof assistant for the *Calculus of Inductive Constructions (CIC)* [CH85, PM93]. The syntax of *Coq* is quite readable, providing 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.

### 2.1 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 [ML84]. The hierarchy can be ignored by the user of the system, which automatically keeps track of universes above the base cases.

### 2.2 Logical Notation in ASCII

Lambda abstraction is represented (following **AUTOMATH** [dB80]) by square brackets; e.g.  $[x:A]x$  is the anonymous identity function on a set **A**.

Universal quantification is represented by round brackets; e.g. reflexivity of equality in a set **A** would be stated  $(x, y:A)x=y \rightarrow y=x$ .

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

### 2.3 Definitions

Two basic definition mechanisms are used: **Inductive** (for defining objects and families of sort **Prop** and **Set**) and **Recursive Definition** (for functions). Thus the definition<sup>2</sup> of unary natural numbers (**nat**) in *Coq* is:

```
Inductive
  nat:Set :=
    0 : nat |
    S : nat->nat.
```

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

<sup>2</sup>The number 0 is a reserved token in *Coq*, so the letter 0 is used.

### 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 to (higher order) primitive recursion. A more complex definition mechanism allows definition of recursive functions using a fixpoint operator [Gim94] and also allows mutual definitions. All the definitions in this paper can be expressed using the natural generalisation of **Recursive Definition** to mutual recursion, and definitions will be expressed in this way to enhance readability.

## 2.4 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 (i.e. family of propositions):

### 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 true propositions which are members of this family are built up from a basic fact  $(n:nat)(lt\ 0\ (S\ n))$  and a finite sequence of implications incrementing both arguments ( $lt\_S$ ).

Similarly, if we have a hypothesis that  $(lt\ i\ j)$ , 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 no longer necessary, as there is a tactic to perform such a case analysis on a hypothesis of the current sequent [BB<sup>+</sup>96, Ch.8].

## 2.5 Performing Proofs in *Coq*

Later we shall be using the *Coq* representation of sequents to show proofs in progress. To prove a theorem in *Coq* we present the system with a type, for which we aim to construct a witnessing term that inhabits that type. Unlike *ALF*, in which the user directly constructs the term, construction of the term in *Coq* is done by the programme behind the scenes. We give the program commands which further the search for such a term. We shall work through part of a proof to demonstrate the proof display syntax. Proofs in *Coq* are performed in a root-to-leaf method, where the root of a proof tree is a theorem and the leaves are axioms. Suppose we are trying to prove the following simple theorem about natural numbers:

$$\forall i : \mathbf{N}. i < S(i).$$

In *Coq* syntax this is formalised as the type:

```
(i:nat)(lt i (S i))
```

Having entered this into *Coq* as a conjecture to be proved (under the name `ltiSi`) we are presented with the following display:

1 subgoal

```
=====
(i:nat)(lt i (S i))
```

Initially, there is only a single (sub)goal to be proved. We may have *Coq* show us all the remaining subgoals or only one at a time. If we then perform an introduction step for the universal quantifier we see:

1 subgoal

```
i : nat
=====
(lt i (S i))
```

Elimination on the type `nat` (i.e. induction) gives us:

2 subgoals

```
i : nat
=====
(lt 0 (S 0))
```

subgoal 2 is:

```
(n:nat)(lt n (S n))->(lt (S n) (S (S n)))
```

Here, *Coq* is showing us all the remaining sub-goals but only the first is displayed in full; only the conclusions (consequents) of the other goals are shown. We may have *Coq* show us the full sequent for subgoal 2:

subgoal 2 is:

```
i : nat
=====
(n:nat)(lt n (S n))->(lt (S n) (S (S n)))
```

### 3 LJ, MJ, NJ and their Formalisation

#### 3.1 Informal Definition of the Systems

To present a coherent picture of the three systems, a single approach is taken for each. 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 [Gen34], while **NJ** is a sequent-style calculus equivalent to natural deduction with assumption classes [Lei79]. All three systems are cut-free. Cut-elimination for **NJ**<sup>+cut</sup> and **LJ**<sup>+cut</sup> is well-known, and cut-elimination for **MJ**<sup>+cut</sup> has been shown in [Her94] (see also [DP97]). **NJ** also differs from a standard presentation of the simply-typed  $\lambda$ -calculus in its splitting of terms into *normal* (**N**) and *applicative* (**A**) terms.

The sets of deduction/derivation terms of these systems are **A** and **N** for **NJ**, **M** and **Ms** for **MJ**, and **L** for **LJ**, defined as follows:

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

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

$$|app(x, l_1, y.l_2)| = [ap(x, |l_1|)/y]|l_2|.$$

Taking  $P, Q, R$  as meta-variables for formulae and  $\Gamma$  for contexts<sup>3</sup>, the rules for the three systems are in table 3.1.

Having defined the systems, we now examine the relationships between them. Table 2 contains the informal function definitions for translating deduction/derivation terms between systems, and table 3.1 the identities and other theorems showing the full relationships between the calculi.

In particular, having shown that the systems  $\mathbf{MJ}$  and  $\mathbf{NJ}$  are in one-one correspondence, we may safely ignore  $\mathbf{NJ}$  for the most part and concentrate our efforts on  $\mathbf{LJ}$  and  $\mathbf{MJ}$ . While  $\mathbf{MJ}$  has turned out to be an interesting calculus in its own right, the initial aim of this theory was to investigate permutations in  $\mathbf{LJ}$ : which derivations in  $\mathbf{LJ}$  are truly different, and which differ only in the order of application of non-interacting rules. For this work, we define a system of permutations on derivations in  $\mathbf{LJ}$  which are shown to be immutable under the retraction via  $\mathbf{MJ}$  (immediately giving us the same result with respect to  $\mathbf{NJ}$ ). So, in table 3.1, we define a single-step permutation reduction on the untyped derivation terms of  $\mathbf{LJ}$ . The normal form to which terms reduce is defined in table 3.1. Finally, in table 3.1 we have the theorems such as subject-reduction and the theorems showing weak normalisation as per the specification of weak normalisation for *abstract reduction systems* in [Klo92, Definition 2.0.3(2)].

## 3.2 Formal Definition of the Systems

In this section we examine the formal definition of the systems presented informally above, including the auxiliary definitions required for us to reason about those formalised systems.

### 3.2.1 Variables and Proof Terms

The primary problem of representing systems such as these in a formal environment such as *Coq* is that of the representation of the variables  $\mathbf{V}$ . The specific approach taken here is that of de Bruijn indices [dB72]. In this approach, variables are represented by natural numbers, and there are no problems with  $\alpha$ -conversion or variable capture during substitution. There are problems with this approach: most obvious is that the gulf between our intuitive understanding of the underlying calculi and the formal representation is quite large. Much care needs to be taken to ensure that the formal representation does capture the informal theory correctly. See §5 for a discussion of de Bruijn indices and their merits relative to other approaches.

---

<sup>3</sup>Contexts are defined to be functions from a finite set of variables to a set of formulae.

---

## NJ

---

$$\frac{\Gamma, x : P \triangleright\triangleright n : Q}{\Gamma \triangleright\triangleright \lambda x.n : (P \supset Q)} \supset \text{I}$$

$$\frac{\Gamma \triangleright a : P}{\Gamma \triangleright\triangleright an(a) : P} \text{N-Axiom}$$

$$\frac{\Gamma \triangleright a : (P \supset Q) \quad \Gamma \triangleright\triangleright a : P}{\Gamma \triangleright ap(a, n) : Q} \supset \text{E}$$

$$\frac{}{\Gamma, x : P \triangleright var(x) : P} \text{A-Axiom}$$

---

## MJ

---

$$\frac{\Gamma, x : P \xrightarrow{P} ms : R}{\Gamma, x : P \Rightarrow (x ; ms) : R} \text{Choose}$$

$$\frac{\Gamma, x : P \Rightarrow m : Q}{\Gamma \Rightarrow \lambda x.m : (P \supset Q)} \text{Abstract}$$

$$\frac{}{\Gamma \xrightarrow{P} [] : P} \text{Meet}$$

$$\frac{\Gamma \Rightarrow m : P \quad \Gamma \xrightarrow{Q} ms : R}{\Gamma \xrightarrow{P \supset Q} m :: ms : R} \supset \text{S}$$

---

## LJ

---

$$\frac{}{\Gamma, x : P \rightarrow vr(x) : P} \text{L-Axiom}$$

$$\frac{\Gamma, z : P \supset Q \rightarrow l_1 : P \quad \Gamma, x : Q, z : P \supset Q \rightarrow l_2 : R}{\Gamma, z : P \supset Q \rightarrow app(z, l_1, x.l_2) : R} \supset \text{L}$$

$$\frac{\Gamma, x : P \rightarrow l : Q}{\Gamma \rightarrow \lambda x.l : P \supset Q} \supset \text{R}$$

Table 1: Proof Rules for **NJ**, **MJ**, **LJ**.

$\theta : \mathbf{M} \rightarrow \mathbf{N}$
$\theta(x ; ms) =_{def} \theta'(var(x), ms)$ $\theta(\lambda x . m) =_{def} \lambda x . (\theta(m))$
$\theta' : \mathbf{A} \times \mathbf{Ms} \rightarrow \mathbf{N}$
$\theta'(a, []) =_{def} an(a)$ $\theta'(a, m :: ms) =_{def} \theta'(ap(a, \theta(m)), ms)$
$\psi : \mathbf{N} \rightarrow \mathbf{M}$
$\psi(an(a)) =_{def} \psi'(a, [])$ $\psi(\lambda x . n) =_{def} \lambda x . (\psi(n))$
$\psi' : \mathbf{A} \times \mathbf{Ms} \rightarrow \mathbf{M}$
$\psi'(var(x), ms) =_{def} (x ; ms)$ $\psi'(ap(a, n), ms) =_{def} \psi'(a, (\psi(n)) :: ms)$
$\bar{\rho} : \mathbf{M} \rightarrow \mathbf{L}$
$\bar{\rho}(x ; []) =_{def} vr(x)$ $\bar{\rho}(x ; m :: ms) =_{def} app(x, \bar{\rho}(m), z . \bar{\rho}(z ; ms))$ <span style="float: right;"><math>z \text{ new}</math></span> $\bar{\rho}(\lambda x . m) =_{def} \lambda x . \bar{\rho}(m)$
$\bar{\phi} : \mathbf{L} \rightarrow \mathbf{M}$
$\bar{\phi}(vr(x)) =_{def} (x ; [])$ $\bar{\phi}(app(x, l_1, y . l_2)) =_{def} sub(x, \bar{\phi}(l_1), y, \bar{\phi}(l_2))$ $\bar{\phi}(\lambda x . l) =_{def} \lambda x . \bar{\phi}(l)$
$sub : \mathbf{V} \times \mathbf{M} \times \mathbf{V} \times \mathbf{M} \rightarrow \mathbf{M}$
$sub(x, m, y, (y ; ms)) =_{def} (x ; m :: subs(x, m, y, ms))$ $sub(x, m, y, (z ; ms)) =_{def} (z ; subs(x, m, y, ms))$ <span style="float: right;"><math>z \neq y</math></span> $sub(x, m, y, \lambda z . m') =_{def} \lambda z . sub(x, m, y, m')$
$subs : \mathbf{V} \times \mathbf{M} \times \mathbf{V} \times \mathbf{Ms} \rightarrow \mathbf{Ms}$
$subs(x, m, y, []) =_{def} []$ $subs(x, m, y, m' :: ms) =_{def} sub(x, m, y, m') :: subs(x, m, y, ms)$
$\rho : \mathbf{N} \rightarrow \mathbf{L}$
$\rho(n) =_{def} \bar{\rho}(\psi(n))$
$\phi : \mathbf{L} \rightarrow \mathbf{N}$
$\phi(vr(x)) =_{def} an(var(x))$ $\phi(app(x, l_1, y . l_2)) =_{def} [ap(x, \phi(l_1))/y]\phi(l_2)$ $\phi(\lambda x . l) =_{def} \lambda x . \phi(l)$

Table 2: Translation functions for proof terms.

---

$\psi\theta : \psi(\theta(m)) = m$	
$\psi\theta'\psi' : \psi(\theta'(a, ms)) = \psi'(a, ms)$	
$\theta\psi : \theta(\psi(n)) = n$	
$\theta\psi'\theta' : \theta(\psi'(a, ms)) = \theta'(a, ms)$	
<b>N_Admiss<sub><math>\theta</math></sub></b> : $\frac{\Gamma \Rightarrow m : R}{\Gamma \triangleright\triangleright \theta(m) : R}$	
<b>N_Admiss<sub><math>\theta'</math></sub></b> : $\frac{\Gamma \triangleright a : P \quad \Gamma \xrightarrow{P} ms : R}{\Gamma \triangleright\triangleright \theta'(a, ms) : R}$	
<b>M_Admiss<sub><math>\psi</math></sub></b> : $\frac{\Gamma \triangleright\triangleright n : R}{\Gamma \Rightarrow \psi(n) : R}$	
<b>M_Admiss<sub><math>\psi'</math></sub></b> : $\frac{\Gamma \triangleright a : P \quad \Gamma \xrightarrow{P} ms : R}{\Gamma \Rightarrow \psi'(a, ms) : R}$	
$\bar{\phi}\bar{\rho} : \bar{\phi}(\bar{\rho}(m)) = m$	$\rho\theta\bar{\rho} : \rho(\theta(m)) = \bar{\rho}(m)$
$\theta\bar{\phi}\phi : \theta\bar{\phi}(l) = \phi(l)$	$\phi\rho : \phi(\rho(n)) = n$
<b>L_Admiss<sub><math>\bar{\rho}</math></sub></b> : $\frac{\Gamma \Rightarrow m : R}{\Gamma \rightarrow \bar{\rho}(m) : R}$	<b>M_Admiss<sub><math>\bar{\phi}</math></sub></b> : $\frac{\Gamma \rightarrow l : R}{\Gamma \Rightarrow \bar{\phi}(l) : R}$
<b>N_Admiss<sub><math>\phi</math></sub></b> : $\frac{\Gamma \rightarrow l : R}{\Gamma \triangleright\triangleright \phi(n) : R}$	<b>L_Admiss<sub><math>\rho</math></sub></b> : $\frac{\Gamma \triangleright\triangleright n : R}{\Gamma \rightarrow \rho(n) : R}$

---

Table 3: Relationships between the calculi

---

$(lm)$	$\frac{l_1 \succ l_2}{\lambda x.l_1 \succ \lambda x.l_2}$	
$(app1)$	$\frac{l_1 \succ l_2}{app(x, l_1, y.l_3) \succ app(x, l_2, y.l_3)}$	
$(app2)$	$\frac{l_2 \succ l_3}{app(x, l_1, y.l_2) \succ app(x, l_1, y.l_3)}$	
$(app\_wkn)$	$app(x, l_1, y.l_2) \succ l_2$	$y \notin l_2$
$(app\_app1)$	$\frac{app(x, l_1, z.app(y, l_2, w.l_3))}{app(y, app(x, l_1, z.l_2), w.app(x, l_1, z.l_3))}$	$y \neq z$
$(app\_app2)$	$\frac{app(x, l_1, y.app(y, l_2, z.l_3))}{app(x, l_1, y'.app(y', app(x, l_1, y.l_2), z.app(x, l_1, y.l_3)))}$	$y' \text{ new}$
$(app\_lm)$	$app(x, l_1, y.\lambda z.l_2) \succ \lambda z.app(x, l_1, y.l_2)$	

---

Table 4: Permutations of Derivations in **LJ**

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

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

Table 5: Normal Forms of terms in **L** wrt  $\succ$

---

L_Adms_Perm1 :	$\frac{l_1 \succ l_2 \quad \Gamma \rightarrow l_1 : R}{\Gamma \rightarrow l_2 : R}$
L_Adms_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_appl	$\frac{\frac{l_1 \succ^* l_2}{app(x, l_1, y.l_3)} \succ^* app(x, l_2, y.l_3)}{app(x, l_1, y.l_2) \succ^* app(x, l_1, y.l_3)}$
Norm_Imperm_L :	Normal( $l$ ) $\Leftrightarrow \sim l \succ l_0$
Norm_L_ρ :	Normal( $\bar{\rho}(m)$ )
App_Red_M :	$app(x, \bar{\rho}(m_1), y.\bar{\rho}(m_2)) \succ^* \bar{\rho}(sub(x, m_1, y, m_2))$
Red_Norm :	$l \succ^* \bar{\rho}(\bar{\phi}(l))$

---

Table 6: Subject Reduction and Weak Normalisation

The set  $\mathbf{V}$  is defined simply as a pseudonym for the natural numbers. We may thus use  $\mathbf{V}$  and  $\mathbf{nat}$  interchangeably, depending on the context of use. Then we have the definition of the deduction terms for deductions in  $\mathbf{NJ}$ :

**Mutual Inductive**

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

**with**

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

So, we have three notations for the deduction/derivation terms of the systems: informal notation with named variables, informal notation with de Bruijn indices using decimal arabic numerals, and the *Coq* formal notation using de Bruijn indices and unary natural numbers. We illustrate each notation with the same term (corresponding to the  $\mathbf{S}$ -combinator)

Informal named syntax

$\lambda x.\lambda y.\lambda z.ap(ap(var(x), an(var(z))), ap(var(y), an(var(z))))$

Informal de Bruijn syntax

$\lambda.\lambda.\lambda.ap(ap(var(2), an(var(0))), ap(var(1), an(var(0))))$

Formal *Coq* de Bruijn syntax

`(lam (lam (lam (ap (ap (var 2), (an (var 0))), (ap (var 1), (an (var 0))))))`

We also have the following definitions for the derivation terms for derivations in **LJ** and **MJ**:

**Mutual Inductive**

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

with

```
Ms:Set :=
  mnil : Ms |
  mcons : M->Ms->Ms.
```

**Inductive**

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

From these definitions for **M** and **Ms**, the following induction principle is semi-automatically generated:<sup>4</sup>

```
(P:M->Prop)
(P0:Ms->Prop)
((v:V)(ms:Ms)(P0 m)->(P (sc v ms)))->
((m:M)(P m)->(P (lambda m)))->
(P0 mnil)->
((m:M)(P m)->(ms:Ms)(P0 m0)->(P0 (mcons m ms)))->
(((m:M)(P m)) /\ ((ms:Ms)(P0 ms))).
```

This is equivalent to the induction scheme:

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

Similar schemes are (semi-)automatically produced for **N** and **A** and for **L**.

Next we turn our attention to the other parts of the sequents we wish to represent. To define our calculi we need formulae and contexts. At this point, we are only interested in the implicational fragment of the propositional calculi, so formulae are defined:

**Inductive**

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

Abstractly, a context is a function mapping a set of distinct variables onto a set of formulae. Using de Bruijn indices, this is formalised by defining contexts as a list of Formulae. As will be seen in the formal definitions of derivations and deductions, the naming of de Bruijn indexing flows seamlessly from bound variables to free variables using this definition. We use the polymorphic **List** library provided with *Coq* to implement the contexts, and define some syntactic sugar to aid interaction. The syntactic sugar makes using contexts (called **Hyps** and actually an abbreviation for the type `(list F)`) the same as if they were defined thus:

<sup>4</sup>Some simple cut-and-paste and an easy proof is currently required for induction principles derived from mutual inductive definitions. A macro for automating this should be included in the next full release.

```

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

```

This allows us access to the standard theorems about lists (such as associativity of append) while retaining abstract clarity for the user. For example, `length: (list A) -> nat` is the polymorphic function computing the length of lists of type `A`. We therefore define `Len.Hyps` as an abbreviation for `(length Hyps)`, the (now) monomorphic function for computing the length of lists of `F`s.

### 3.2.2 Decidability of Relations

In order to perform meta-theoretic reasoning about derivations encoded using de Bruijn indices, we require the decidability of certain propositional functions over the natural numbers. In order to prove these, we approach the problem in an indirect way. We will look at the “less than” function over natural numbers as an example. First, we define “less than” (`lt`) as in §2.4:

```

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 we define a boolean function `ltb` which we will prove is equivalent:

```

Recursive Definition
  ltb : nat->nat->bool :=
    0 0 => false |
    0 (S j) => true |
    (S i) 0 => false |
    (S i) (S j) => (ltb i j).

```

Then we prove the four theorems (i.e. each direction of the bi-implications):

$$\forall i, j : \text{nat}. (\text{lt } i \ j) \Leftrightarrow (\text{ltb } i \ j) = \text{true}$$

$$\forall i, j : \text{nat}. \sim (\text{lt } i \ j) \Leftrightarrow (\text{ltb } i \ j) = \text{false}.$$

The decidability of `lt`,

$$\forall i, j : \text{nat}. (\text{lt } i \ j) \vee \sim (\text{lt } i \ j),$$

follows immediately from these theorems.

As mentioned above, this is an indirect approach to proving a theorem which is amenable to a more direct proof by a straightforward induction. There is method in this apparent madness, though. Each of the four theorems above is useful individually. So, using them to prove the decidability of `lt` is simply a bonus.

To show why we require both the propositional and boolean functions for `lt`, we must first look at a polymorphic *if* function.

### 3.2.3 Setifb

We wish to be able to define functions over the sets of deduction/derivation terms and over contexts. These functions should be easy to reason with and about. To this end,

we define a general notion of *If*, not contained in the basic library of *Coq*. In the standard libraries, **IF** is defined with type **Prop**->**Prop**->**Prop**->**Prop**. There is also **ifb** of type **bool**->**bool**->**bool**->**bool** where **bool** is the standard set **{true,false}**. What we require is a complete function using a boolean value as a test and with general inputs and output. Thus, we define **Setifb**:

**Hypothesis A:Set.**

**Recursive Definition**

```

Setifb : bool->A->A->A :=
  true x y => x
  false x y => y.

```

When we discharge the Hypothesis **A**, **Setifb** is defined as the polymorphic *if* over general sets.

### 3.2.4 Lifting and Dropping

*Lifting* is a necessary operation for using de Bruijn indices correctly. An implementation for standard untyped  $\lambda$ -calculus terms can be seen in [Hue94]. Here we will use the standard substitution function in **N** and **A** to illustrate **Lift<sub>N</sub>** and **Lift<sub>A</sub>**. Informally, we can mutually define substitution of an **A** for a variable in an **N** or an **A**:

$$\begin{aligned}
[a_0/x]\lambda y.n &= \lambda y.[a_0/x]n && x \neq y \\
[a_0/x]an(a) &= an([a_0/x]a) \\
[a_0/x]ap(a, n) &= ap([a_0/x]a, [a_0/x]n) \\
[a_0/x]var(y) &= var(y) && x \neq y \\
[a_0/x]var(x) &= a_0
\end{aligned}$$

Let us take as an example the following term including a substitution in both named and nameless variable formats:

```

λx.λy.[var(x)/y]λz.an(λu.an(ap(ap(var(u), an(var(y))), an(var(z))))))
      λ.λ.[var(1)/0]λ.an(λ.an(ap(ap(var(0), an(var(2))), an(var(1))))))

```

Unfolding the application of substitution once, we get:

```

λx.λy.λz.[var(x)/y]an(λu.an(ap(ap(var(u), an(var(y))), an(var(z))))))
      λ.λ.λ.[var(2)/1]an(λ.an(ap(ap(var(0), an(var(2))), an(var(1))))))

```

As can be seen, no changes of name were required to move the substitution ‘through’ the lambda abstraction,<sup>5</sup> but for the de Bruijn indices, each variable has been increased by one to take account of the extra levels of abstraction between the variable occurrence and its ‘parent’ abstraction. Continuing the process through to the end we have the following sequence of terms:

```

λx.λy.λz.an([var(x)/y]λu.an(ap(ap(var(u), an(var(y))), an(var(z))))))
      λ.λ.λ.an([var(2)/1]λ.an(ap(ap(var(0), an(var(2))), an(var(1))))))

λx.λy.λz.[var(x)/y]an(ap(ap(var(u), an(var(y))), an(var(z))))
      λ.λ.λ.[var(3)/2]an(ap(ap(var(0), an(var(2))), an(var(1))))

      ⋮

λx.λy.λz.an(λu.an(ap(ap(var(u), an(var(x))), an(var(z))))))
      λ.λ.λ.an(λ.an(ap(ap(var(0), an(var(3))), an(var(1))))))

```

---

<sup>5</sup>This is due to the careful selection of distinct names for all the variables.

The important point to notice here is that the de Bruijn reference variables increase by one every time we unfold the application of substitution through an abstraction operator.<sup>6</sup> In the above example, the only instances of variables within the term being substituted in  $(var(0))$  are free (within the scope of the term itself. If this term contains variables bound within the term, for instance  $ap(var(x), \lambda w. an(var(w))) (= ap(var(0), \lambda. an(var(0))))$ , then we require more care. Each time we unfold past an abstraction operator we need to increment the free variables within the term but leave the bound variables unchanged. This operation is called *lifting* and is defined thus:

$$\begin{aligned} \uparrow_i \lambda. n &=_{def} \lambda. \uparrow_{(i+1)} n \\ \uparrow_i an(a) &=_{def} an(\uparrow_i a) \\ \uparrow_i ap(a, n) &=_{def} ap(\uparrow_i a, \uparrow_i n) \\ \uparrow_i var(x) &=_{def} \text{if } x < i \text{ var}(x) \text{ else } var(x + 1) \end{aligned}$$

We now see the necessity for **Setifb**, and for the boolean versions of functions such as **ltb** and **nateqb** (boolean equality for **nat**). While it is possible to define functions performing branching on propositional functions (such as the definition of **lift\_rec** in [Hue94]) the use of boolean functions (proved equivalent to the propositional versions) provides greater clarity, in particular when we wish to consider the various cases involved in comparing two generically appearing numbers. Below, we show the lifting operation for derivation terms of **LJ**:

**Recursive Definition**

```
lift_V : nat->V->V :=
  i j => (Setifb V (ltb j i) j (S j)).
```

**Recursive Definition**

```
lift_L : nat->L->L :=
  i (vr x) => (vr (lift_V i x)) |
  i (app x l1 l2) =>
    (app (lift_V i x) (lift_L i l1) (lift_L (S i) l2)) |
  i (lm l) => (lm (lift_L (S i) l)).
```

The separation of **lift\_V** from the individual **lifting** operations for **L**, **A**, **N**, **M** and **Ms** allows us to prove general theorems about the behaviour of lift with regards to other functions operating on variables (such as *drop* and *exchange* below) and use these to show similar theorems about the deduction/derivation term lifting operations generally.

We also require the inverse function of lift, called *drop*, which lowers the value of the de Bruijn indices in a term. This is needed when an abstraction is deleted from a term. (In particular, we will see that lifting and dropping are precisely the functions needed for certain sequent structural operations such as weakening.) Dropping ( $\downarrow_i$ ) is defined in a very similar way to lifting, and the following theorems about lifting and dropping hold for all the sets of deduction/derivation terms:

$$\begin{aligned} \forall i : \text{nat}, t : \mathbf{T}. \downarrow_i \uparrow_i t &= t, \\ \forall i : \text{nat}, t : \mathbf{T}. i \notin t \supset \uparrow_i \downarrow_i t &= t, \end{aligned}$$

where **T** is one of **{M, Ms, N, A, L}**.

So, we have explained why we need the boolean version of equality and other **lt**, but why do we also need the propositional versions? The usefulness of the propositional version of these functions lies in the *Inversion* tactic described in §2.4. Were we to restrict ourselves to the boolean functions, we would have to prove inversion theorems for each function. Defining propositional and boolean functions and showing their equivalence allows us to use the standard inversion tactics for hypotheses and to use those hypotheses to

---

<sup>6</sup>Here, the only abstraction operator is  $\lambda$ , which abstracts on its single argument. The constructor function *app* for terms in **L** is also an abstraction operator on its third argument.

rewrite subterms of the goal involving the boolean version in `Setifb` constructs. Finally, in the case of `nat` equality, we wish to be able to use equality hypotheses as rewriting rules thus:

```

x, y: nat
H: x=y
=====
(P x y)

```

where `P` is some propositional term, can be simplified by using `H` as a rewriting rule to

```

x: nat
=====
(P x x)

```

If we had the hypothesis `H: (nateqb x y)` we would not be able to do this without having proved the equivalence of `nateqb` and `=nat`.

### 3.2.5 Translation Functions

Having defined the deduction/derivation terms and variable adjustment functions, we can now proceed to the functions translating deduction/derivation terms between the three systems, as shown in table 3.2.5. The definitions of the functions translating terms between **NJ** and **MJ** are fairly straightforward, since they are simple primitive recursive definitions, which do not change the level of abstraction of a variable occurrence with respect to its binding.

Of more interest are the translations involving **LJ**. In particular, the definition of  $\bar{\rho}$  requires considerable changes in order to be accepted by *Coq*'s function definition mechanism. If we transform the definition seen in table 3.2.5 to use de Bruijn indices, we get the following:

$$\begin{aligned}
\bar{\rho}(x ; []) &=_{def} vr(x) \\
\bar{\rho}(x ; m :: ms) &=_{def} app(x, \bar{\rho}(m), \bar{\rho}(0; \uparrow_0 ms)) \\
\lambda.m &=_{def} \lambda.\bar{\rho}(m)
\end{aligned}$$

The second recursive call in the right hand side of the second definitional equation is not primitive recursive.  $(0 ; ms)$  is not a sub-expression of  $(x ; m :: ms)$ . We may avoid part of the problem by using a mutual definition such as:

$$\begin{aligned}
\bar{\rho}(x ; m :: ms) &=_{def} \bar{\rho}'(x, m :: ms) \\
\lambda x.m &=_{def} \lambda x.\bar{\rho}(m) \\
\bar{\rho}'(x, []) &=_{def} vr(x) \\
\bar{\rho}'(x, m :: ms) &=_{def} app(x, \bar{\rho}(m), \bar{\rho}'(0, \uparrow_0 ms))
\end{aligned}$$

which is primitive recursive in all but one respect, that of the lifting operation required on  $ms$  in the fourth equation, necessary to retain variable reference consistency. We therefore add an extra argument to the definition of  $\bar{\rho}'$ , which tracks the number of lifting operations we have yet to do. We now reach the following formal *Coq* definitions:

Recursive Definition

```

rhopar : M->L :=
  (sc x mnil) => (vr x) |
  (sc x (mcons m ms)) =>
    (app x (rhopar m) (rhopar' (S 0) ms)) |
  (lambda m) => (lm (rhopar m))
with
rhopar' : nat->Ms->L :=
  i mnil => (vr 0) |
  i (mcons m ms) =>
    (app 0
      (lifts_L i 0 (rhopar m))
      (rhopar' (S i) ms)).

```

where `(lifts_L i j l)` simply repeats the lifting operation of `lift_L` on `l`, with respect to `j`, `i` times. Since these definitions are primitive recursive, they are accepted by *Coq* without problem. It remains to show that this definition of `rhopar` is equivalent to the formal version of the one above. This requires us to prove the three equalities:

**RhoBar1** :  $(x:V)(\text{rhopar } (\text{sc } x \text{ mnil}))=(\text{vr } x)$

**RhoBar2** :

$$(\text{ms}:\text{Ms})(x:V)(m:\text{M})
(\text{rhopar } (\text{sc } x \text{ (mcons } m \text{ ms)}))=
(\text{app } x \text{ (rhopar } m) \text{ (rhopar } (\text{sc } 0 \text{ (lift\_Ms } 0 \text{ ms)})))$$

**RhoBar3** :  $(m:\text{M})(\text{rhopar } (\text{lambda } m))=(\text{lm } (\text{rhopar } m))$

which are the formal *Coq* versions of the definitional equations above.

As we shall see in §4, proof of **RhoBar2** requires stronger induction methods than the standard ones.

A great many lemmas have been proved regarding the interactions between the translation functions and the appropriate version of lift and drop: mostly commutation lemmas. In some cases many variations of the basic lemma are required to take into account comparisons between variables, but this is all tedious detail, and the reader is directed to the proof scripts available with this paper for the details of these.

### 3.2.6 Derivations and Deductions

We now turn our attention to the formal definition of derivations and deduction within **LJ**, **MJ** and **NJ**. We have, above, already defined all the requisite parts of a sequent and now simply put these together in the appropriate way, following the rules shown in table 3.1. These definitions are quite long, so only the definition for the system **MJ** will be shown.

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).

```

The particular point that should be noted is the way in which the de Bruijn indexing works in the **Abstract** rule:

```

(h:Hyps)(P:F)(m:M)(Q:F)
(M_Deriv (Add_Hyp P h) m Q)->
(M_Deriv h (lambda m) (Impl P Q))

```

Variables in  $m$  which reference the initial `lambda` binder in the conclusion of the rule reference the free variable  $P$  in the premise of the rule. This same system also works for the formal definitions of **NJ** and **LJ**. We can take no credit for this, since it is a general property of the particular systems we are working with. Other sequent-style calculi do not necessarily have this property. For instance any linear calculus with context-splitting rules would not share this useful property. See §6 for some discussion on how we might cope with such problems. The fact that all three systems share this property makes our work much easier.

### 3.2.7 Structural Rules

It may be noted that our presentation of the systems does not include any structural rules. Some structural rules are necessary in the proofs of theorems in table 3.1, specifically those involving **LJ**. Again, any proof involving  $\bar{\rho}$  requires a strong induction principle.

The three structural rules we require, at different points, are *Weakening*, *Strengthening* and *Exchange*, as shown below for general sequent-style calculi. Exchange is not necessary for the proofs of theorems in table 3.1, but is essential for proofs about permutation of derivations of **LJ**.

$$x \text{ not free in } t \quad \frac{\Gamma \vdash t : R}{\Gamma, x : P \vdash t : R} \text{ Weakening}$$

$$x \text{ not free in } t \quad \frac{\Gamma \vdash t : R}{\Gamma \setminus x : P \vdash t : R} \text{ Strengthening}$$

$$\frac{\Gamma @ (x : P :: y : Q :: \Delta) \vdash t : R}{\Gamma @ (y : Q :: x : P :: \Delta) \vdash t : R} \text{ Exchange}$$

This, of course, is a representation using named variables. Considering these rules for use with a formal implementation using de Bruijn indices, we see that we need to alter the deduction/derivation term to take account of the change in the context. Careful consideration of *Weakening* and *Strengthening* reveals that lifting and dropping exhibit precisely the functionality that is needed, since all that is happening is that a non-occurring variable

```

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_app1 :
    (x, y:V) (l1, l2, l3:L)
      (L_Perm1 (app x l1 (app (S y) l2 l3))
        (app y
          (app x l1 l2)
          (app (lift_V 0 x)
            (lift_L 0 l1)
            (L_Exchange 0 l3)))))) |
  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))))))) |
  l_perm1_app_lm :
    (x:V) (l1, l2:L)
      (L_Perm1 (app x l1 (lm l2))
        (lm (app (lift_V 0 x)
          (lift_L 0 l1)
          (L_Exchange 0 l2))))

```

Figure 1: Formalised Permutations

is being added to or deleted from the context. Therefore, all we need to do is increase or decrease all the variables in the term which refer to a point beyond the change. The required function for exchange is simply to replace all references to a particular abstraction level with its successor and vice-versa.

### 3.2.8 Formalisation of Permutations in LJ

The permutation reduction defined informally in table 3.1 is more difficult to formalise than it may at first appear. The exact variable namings and renamings that form an integral part of the reductions are subtle, and it is only when looked at in the typed case that one can fully decipher the meanings of the reductions and formalise them to capture the correct translations. Figure 1 shows the formalised versions of the interesting permutations (i.e. the actual permutations, rather than the sub-term permutation rules).

The formalisation of `l_perm1_app_app2` highlights the complexity of the process. Figure 2 shows the informal version of the typed reduction rule. Only the leaves and root of the

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

Figure 2: Proof Tree Fragment for Permutation App\_App2

relevant derivation tree fragments are shown since they contain all the information necessary for the analysis.

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, 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  $\Gamma, y, z.l_3$ . In the permuted derivation the bindings are  $\Gamma, y', z, y.l_3$ . Since  $y'$  does not appear in  $l_3$ , but must be accounted for in the referencing to other variables in  $\Gamma$ ,  $l_3$  must be **lifted** by 2 ( $(\mathbf{S}(\mathbf{S} \ 0))$ ). Also, the occurrences of  $y$  and  $z$  are switched, so the de Bruijn references must be **Exchanged**. Similar analyses give us the lifting, dropping and exchanging requirements for each permutation as shown in figure 1.

## 4 Proof Techniques

In this section we discuss some of the facets of using the formalisation described above to actually perform proofs in *Coq*. Some of this focuses on general issues, some on specific problems with de Bruijn indices, and some on aspects of the *Coq* environment.

### 4.1 Induction Principles

Induction in *Coq*, as with most proof assistants based on type theory, is derived from the standard elimination principle for an inductive definition. So, for instance, from the definition of `nat` given in §2.3, *Coq* derives the induction principle:

```

(P: nat->Prop)
(P 0)->
((n:nat)(P n)->(P (S n)))->
(n:nat)(P n).

```

#### 4.1.1 Inductions on Simple Inductive Sets

Suppose we wish to prove the conjecture about natural numbers from §2.5:

```
(i:nat)(lt i (S i))
```

This requires induction over the natural numbers. If we wish to use the standard induction principle for natural numbers given above, there are various ways to invoke this, all being operationally equivalent, but each being more or less appropriate under different local proof conditions. The *Coq Induction* tactic will attempt to apply the induction scheme given above by using second-order pattern-matching to find a binding for **P** (here it binds to `[i:nat](lt i (S i))`). Sometimes the algorithm cannot find the appropriate set of bindings, at which point we may supply them using the command `Apply ... with ...`. Alternatively, we may define a predicate with the appropriate type (i.e. `nat->Prop`) which has the appropriate functional definition, at which point the algorithm should be able to correctly identify the bindings. When performing proofs involving mutually inductively defined sets (e.g. **M** and **Ms**) we have used this method of defining a predicate.

If we wish to use a non-standard induction principle (such as strong mathematical induction as shown in §4.2), we may not use the *Induction* tactic, which automatically uses the standard principle, but we may apply the principle to the conjecture (either directly or via a defined predicate to supply the bindings).

#### 4.1.2 Induction for More Complex Sets

When we have families of propositions such as `L_Deriv`:

**Inductive**

```

L_Deriv : Hyps -> L -> F -> Prop :=
  L_Axiom :
    (h:Hyps)(i:V)(P:F)
    (In_Hyps i P h)->
    (L_Deriv h (vr i) P) |
  Implies_L :
    (h:Hyps)(i:V)(P:F)(Q:F)(l1:L)(l2:L)(R:F)
    (In_Hyps i (Impl P Q) h)->
    (L_Deriv h l1 P)->
    (L_Deriv (Add_Hyp Q h) l2 R)->
    (L_Deriv h (app i l1 l2) R) |
  Implies_R :
    (h:Hyps)(P:F)(l:L)(Q:F)
    (L_Deriv (Add_Hyp P h) l Q)->
    (L_Deriv h (lm l) (Impl P Q)).

```

there are two ways in which we may approach proofs involving such a family.

### 4.1.3 Direct Induction over Families

Firstly, we may use induction directly on the family, for which we must supply bindings, since the algorithm cannot solve the second-order matching problem in these cases. So, we might define a predicate with type: with type

```
(l:Hyps)(l0:L)(f:F)(L_Deriv l l0 f)->Prop
```

and apply our induction principle derived from the above family.

### 4.1.4 Induction with Inversion

Some families are defined so that one of the arguments (here the argument of type  $L$ ) is composed in a tight correspondence with the formation of the family. In this case, we might also perform induction on this term and then use inversion (see §2.4) on the hypotheses involving the family to gain the correct induction hypotheses. When defining judgements for a deductive system with a term calculus, this should always be possible, since the deduction/derivation terms are designed to represent the proofs, and should therefore have an appropriate correspondence.

In general, we would use induction directly on the family. We shall see in the next section that when using strong induction methods, we will wish to use this second method of ‘inducting on the proof term then inverting the judgement hypotheses’.

## 4.2 Strong Induction Principles

As mentioned in §3.2.5, proofs of theorems involving  $\bar{\rho}$  require a different induction principle from the automatically generated ‘standard’ principle inferred from the definition of  $\mathbf{M}$  and  $\mathbf{Ms}$ . This standard principle is, basically, an immediate sub-term induction. That is, we assume that all the immediate sub-terms of some term have a property and then prove that the term itself has this property. For mutually defined sets, we have a slight variation on this theme in that we have two properties (usually mutually defined via a recursion similar to the original mutual set recursive definition). Performing the obvious eliminations we obtain induction hypotheses assuming the property appropriate to the type of each subterm. A stronger induction principle may be needed, such as with natural numbers needing strong mathematical induction:

$$\forall P: (\mathbf{N} \rightarrow Prop). (\forall j: \mathbf{N}. (\forall i: \mathbf{N}. i < j \supset P(i) \supset P(j)) \supset \forall n: \mathbf{N}. NP(n)).$$

*Coq* includes a library to ease production and proof of this principle (the *well-founded* library). Unfortunately, at present this does not cover mutually defined sets. It is therefore necessary to prove strong induction principles for mutually defined sets directly.<sup>7</sup>

The definition of  $\bar{\rho}$  in [DP96] requires some justification of its admissibility as a total function, since the recursion is non-standard. This justification takes the form of a measure function on  $\mathbf{M}$  and  $\mathbf{Ms}$  which equates to the *height* of a derivation: i.e. the length of the longest branch of the derivation tree.

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

This definition is easily translated into the formal *Coq* syntax. We prove various theorems about the height of terms, such as the fact that lifting or dropping of a deduction/derivation

<sup>7</sup>An extension should appear in the next full release of the *Coq* system.

term do not alter its height. We also prove the following induction principle, allowing us to perform induction on the height of a derivation in **MJ**:

```
(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))
```

where `Height_M` and `Height_Ms` are the formal functions calculating the *height* of a derivation term (and therefore a derivation) in **MJ**. This induction method is used by applying it first, and then performing *non-inductive elimination* (i.e. case-analysis) on the `m` and `ms`.

So, we have an induction principle which we may use to prove theorems involving  $\bar{\rho}$  about the derivation terms. If we wish to apply this strong induction principle to theorems about derivations involving  $\bar{\rho}$ , then we need to use the ‘induction on proof term then inversion of the judgement hypotheses’ method described in §4.1.2 above.

## 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 with functional relations.

Recent formalisations such as [NN96] and [Bar96] employ methods similar to the ones highlighted here. The methods of [Alt93] seem similar to those presented in this paper, although the syntax of the proof assistant as it was then, obscures the issues somewhat. Since System F, the logic being reasoned about in [Alt93], is more powerful than the logics being represented in this paper the formalisation is necessarily more complex. At base, however, the approaches seem close, differing mainly in ways dictated by the underlying proof assistants (LEGO and *Coq*).

[Coq93] presents another formalisation of a normalisation argument. The methods are closely based on the paradigm of *ALF*, and translation to the very different paradigm of a system like *Coq* is non-trivial. Again, a variant of the de Bruijn encoding was used.

[MP93] introduces 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 [McK96]. (A simpler name is the Coquand-McKinna-Pollack approach. It was developed by McKinna and Pollack, with some suggestions by Coquand, for their work on the meta-theory of Pure Type Systems: e.g. [vBJMR94].) Their approach involves defining a named-variable syntax for terms. An equality predicate for each set of terms used must be defined,<sup>8</sup> and substitutivity with respect to this equality needs to be included in many of the definitions. It might also necessitate the use of functional relations instead of *Coq* functions for  $\theta$ ,  $\psi$  etc. There are some advantages to this approach, for instance in the case of  $\bar{\rho}$ : a functional relation definition would provide a suitable elimination principle for proving theorems involving  $\bar{\rho}$ , rather than the necessity of defining the height induction.

<sup>8</sup>As with the higher order abstract syntax.

*Higher order abstract syntax* (from here on referred to as *HOAS*) is one of the central techniques of the *LF* approach, embodied particularly in the *Elf* framework [Pfe91]. The usage of this method is subtle, and works within logical frameworks such as *Elf*. Essentially we define the language that we wish to reason about using the variables of the framework to represent the local variables of the language. Thus, we obtain  $\alpha$ -conversion and  $\beta$ -reduction ‘for free’ from the framework notions of conversion and reduction. However, the definition of the set of terms requires a recursive occurrence in an unsound position in type theories which allow induction, such as *Coq* [PPM89]. The problem part of the appropriate definition is for the binding operators. If we are defining a type *term* in a framework which allows *HOAS*, then the type of the  $\lambda$  abstractor is:

$$(term \rightarrow term) \rightarrow term.$$

The part we are interested in is the antecedent of the type:

$$(term \rightarrow term).$$

In [PPM89], there is a restriction on recursive occurrences of the type being defined, which states that the type itself may not occur in a negative position in the antecedent. [PPM89, Definition 2, page 213], which we paraphrase here for the non-dependent case, defines negative occurrences:

$x$  occurs *negatively* in  $R$  if  
 $R = R_1 \rightarrow R_2$  and  
 $x$  occurs *positively* in  $R_1$  or  
 $x$  occurs *negatively* in  $R_2$

where

$x$  occurs *positively* in  $R$  if  
 $R = x$  or  
 $R = R_1 \rightarrow R_2$  and  
 $x$  occurs *negatively* in  $R_1$  or  
 $x$  occurs *positively* in  $R_2$ .

Thus, in:

$$(\underline{term} \rightarrow term) \rightarrow term.$$

the underlined occurrence of *term* is a negative occurrence in the antecedent of the type of the  $\lambda$  constructor and thus disallows the inductive definition of *term*. At present, although *HOAS* is a very powerful methodology, it cannot be implemented in a system in which induction is a core method.

[DFH95] presents a restricted form of *HOAS* which can be defined within *Coq*, but the resulting method is still unsatisfactory. [DPS96] presents a method of allowing primitive recursive definitions while using higher order abstract syntax in

... an important first step towards allowing the methodology of LF to be employed effectively in systems based on induction principles such as ALF, Coq or Nuprl, leading to a synthesis of currently incompatible paradigms.

Finally, there is some recent work devoted to developing a new framework designed for formal meta-theory. [MM97] presents a system based on Intuitionistic First Order Logic with definitions and natural numbers as part of the primitive system. Inductive arguments appeal to natural number induction. The paper presents on-going investigations.

[Ada97] looks at the related work in more detail.

## 6 Further Work

The informal theory has been developed [DP96] to include full propositional logic, and it is intended to extend the formalisation also. Extension of the formalised proofs to the first order case was the main motivation behind this work, since it is at this stage that machine support becomes invaluable for work of this kind. Strong normalisation for a different concept of permutations has been shown informally by Schwichtenberg in [Sch].

Similar problems exist in other logics, most notably [GP94] on permutations in linear logic. As mentioned above, the formalisation presented here would not be completely ap-posite for work with linear logic. The problems come when dealing with a term calculus and its interactions with the context-splitting rules. When a context is split (in root-to-leaf proof), the levels of referencing to the free variables of the term (referencing formulae in the context) are radically altered. Some possibilities for avoiding this problem are to amend the context in some way, barring certain formulae from use in a branch of the proof tree, or to encode variables as binary trees of integers rather than simply 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.

## 7 Conclusions

This paper has presented a formalisation of sequent-style calculi using de Bruijn 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 allow named variables, and consequently easier human-readable proofs, there may be a consequent loss of equivalence between informal and formal developments where functions are used. Higher order abstract syntaxes move even further away from the informal development and would appear to be more difficult to use. One primary concern in using de Bruijn indices has been to avoid a shift away from the intensional equality of the system to a new equality relation. Each of the alternative approaches requires this.

The use of de Bruijn indices in the formalisation brings its own problems, notably the problems with primitive recursive definition highlighted in §3.2.5, and the increase in the number of bridging (commutation) lemmas required. There is also the matter of the gap between the informal and formal definitions. Since one aim of formalising proofs is to gain confidence in one's informal theory, this may be undermined by a lack of correspondence between the formal and informal definitions.

At present it would appear that those wishing to formalise informal proofs have no easy route to that goal. De Bruijn indices are still the simplest and easiest to understand of the possible approaches, and most of the recent works in formal meta-theory have used this approach. Until a coherent, well-developed tool is available that supports named variables with an abstract syntax while still providing standard equality reasoning and straightforward inductive abilities, de Bruijn indices will probably continue to be the main platform for formal meta-theory.

*Coq* is only one of the systems available, and other systems of similar power and maturity (e.g. *HOL*, *NuPRL*) may bring different strengths and weaknesses to bear on such problems. In particular, it is possible that the extensional equality mechanisms in *HOL* [GM93] might bring an abstract syntax closer to the original informal development.

## References

- [Ada97] A. A. Adams. Approaches to Formal Meta-Theory. Submitted April 1997, 1997.
- [Alt93] Th. Altenkirch. A formalisation of the strong normalisation proof for System F in LEGO. In [BG93], 13–28.
- [Bar96] B. Barras. *Coq en Coq*. Technical report, INRIA, 1996. In French.
- [BB<sup>+</sup>96] B. Barras, S. Boutin, et al. The *Coq* Proof Assistant Reference Manual (Version 6.1). Technical report, INRIA, 1996. Available on-line with *Coq* distribution from ftp.inria.fr.
- [BG93] M. Bezem and J. F. Groote, editors. *Typed Lambda Calculus and Applications*. Springer-Verlag LNCS 664, 1993.
- [Coq93] C. Coquand. From Semantics to rules: A machine assisted analysis. Springer-Verlag LNCS 832, 1993.
- [CH85] Th. Coquand and G. Huet. Constructions: A Higher Order Proof System for Mechanizing Mathematics. 151–184. Springer-Verlag LNCS 203, 1985.
- [dB72] N. G. de Bruijn.  $\lambda$ -Calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation. *Indag. Math*, 34:381–392, 1972.
- [dB80] N. G. de Bruijn. A Survey of the Project AUTOMATH. 579–606. Academic Press, 1980.
- [DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-Order Abstract Syntax in *Coq*. 124–138. Springer-Verlag LNCS 902, 1995.
- [DPS96] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive Recursion for Higher Order Abstract Syntax. Technical Report CMU-CS-96-172, School of Computer Science, Carnegie Mellon University, 1996.
- [DP96] R. Dyckhoff and L. Pinto. Permutability of proofs in intuitionistic sequent calculi, 1996. Submitted for publication.
- [DP97] R. Dyckhoff and L. Pinto. Cut-Elimination and Herbelin’s Sequent Calculus for Intuitionistic Logic. *Studia Logica (to appear)*, 1997.
- [GP94] D. Galmiche and G. Perrier. On Proof Normalisation in Linear Logic. *Theoretical Computer Science*, 135(1):67–110, 1994.
- [Gen34] G. Gentzen. Investigations into Logical Deduction. In *The Collected Papers of Gerhard Gentzen*, Studies in Logic and the Foundations of Mathematics, 68–131. North-Holland, 1934. Translated from 1934 original in German.
- [Gim94] E. Gimenez. Codifying guarded definitions with recursive schemes. 39–59. Springer-Verlag LNCS 996, 1994.
- [GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL*. CUP, 1993.
- [Her94] H. Herbelin. A  $\lambda$ -calculus Structure Isomorphic to Gentzen-style Sequent Calculus Structure. 61–75. Springer-Verlag LNCS 933, 1994.
- [Hue94] G. Huet. Residual Theory in  $\lambda$ -calculus: A Complete Gallina Development. *J. Functional Programming*, 3(4):371–394, 1994.
- [Klo92] J. W. Klop. Term Rewriting Systems. Oxford, 1992.
- [Lei79] D. Leivant. Assumption Classes in Natural Deduction. *Zeitschrift für math. Logik*, 25:1–4, 1979.

- [ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
- [MM97] C. McDowell and D. Miller. A Logic for Reasoning with Higher-Order Abstract Syntax. (Extended Abstract). Submitted for publication, 1997.
- [McK96] J. McKinna. Private Communication, 1996.
- [MP93] J. McKinna and R. Pollack. Pure type systems formalized. In [BG93], 289–305.
- [NN96] D. Nazareth and T. Nipkow. Formal Verification of Algorithm W: The Monomorphic Case. Springer-Verlag LNCS 1125, 1996.
- [PM93] C. Paulin-Mohring. Inductive definitions in the system *Cog*. Rules and properties. In [BG93].
- [Pfe91] F. Pfenning. Logic programming in the LF logical framework. 149–181. CUP, 1991.
- [PPM89] P. Pfenning and C. Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. 209–228. Springer-Verlag LNCS 442, 1989.
- [Sch] S. Schwichtenberg. Termination of permutative conversions in intuitionistic Gentzen calculi. Submitted for publication, Jan 97.
- [vBJMR94] L. S. van Benthem Jutting, J. McKinna, and Pollack R. Checking Algorithms for Pure Type Systems. 19–61. Springer-Verlag LNCS 806, 1994.