TOOLS AND TECHNIQUES FOR MACHINE-ASSISTED META-THEORY

Andrew A. Adams

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

1997

Tools and Techniques for Machine-Assisted Meta-Theory

A thesis submitted to the

UNIVERSITY OF ST ANDREWS

for the degree of

DOCTOR OF PHILOSOPHY

by

Andrew A. Adams

School of Mathematical and Computational Sciences

University of St Andrews

August 1997

I, Andrew A. Adams, hereby certify that this thesis, which is approximately 40,000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date 20/10/97 **signature of candidate**

I was admitted as a research student in October 1994 and as a candidate for the degree of Doctor of Philosophy in October 1995; the higher study for which this is a record was carried out in the University of St Andrews between 1994 and 1997.

date 20/10/97 **signature of candidate**

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 21/10/97 **signature of supervisor**

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any <u>bona fide</u> library or research worker.

date 20/10/97 **signature of candidate**

Abstract

Machine-assisted formal proofs are becoming commonplace in certain fields of mathematics and theoretical computer science. New formal systems and variations on old ones are constantly invented. The meta-theory of such systems, i.e. proofs *about* the system as opposed to proofs *within* the system, are mostly done informally with a pen and paper. Yet the meta-theory of deductive systems is an area which would obviously benefit from machine support for formal proof. Is the software currently available sufficiently powerful yet easy enough to use to make machine assistance for formal meta-theory a viable proposition?

This thesis presents work done by the author on formalising proof theory from [DP97a] in various formal systems: *SEQUEL* [Tar93, Tar97], *Isabelle* [Pau94] and *Coq* [BB+96]. *SEQUEL* and *Isabelle* were found to be difficult to use for this type of work. In particular, the lack of automated production of induction principles in *SEQUEL* and *Isabelle* undermined confidence in the resulting formal proofs. *Coq* was found to be suitable for the formalisation methodology first chosen: the use of nameless dummy variables (de Bruijn indices) as pioneered in [dB72]. A second approach (inspired by the work of McKinna and Pollack [vBJMR94, MP97]) formalising named variables was also the subject of some initial work, and a comparison of these two approaches is presented. The formalisation was restricted to the implicational fragment of propositional logic. The informal theory has been extended to cover full propositional logic by Dyckhoff and Pinto, and extension of the formalisation using de Bruijn indices would appear to present few difficulties. An overview of other work in this area, in terms of both the tools and formalisation methods, is also presented.

The theory formalised differs from other such work in that other formalisations have involved only one calculus. [DP97a] involves the relationships between three different calculi. There is consequently a much greater requirement for equality reasoning in the formalisation.

It is concluded that a formalisation of any significance is still difficult, particularly one involving multiple calculi. No tools currently exist that allow for the easy representation of even quite simple systems in a way that fits human intuitions while still allowing for automatic derivation of induction principles. New work on integrating higher order abstract syntax and induction may be the way forward, although such work is still in the early stages.

Contents

1 Introduction **1**

1.1 Logical Frameworks 1

1.2 Terminology 3

1.2.1 Sequent Calculus and Natural Deduction 3

1.2.2 Proofs, Derivations and Deductions 3

1.2.3 Unfolding 3

1.3 The Requirement for a Meta-Logic 4

1.4 Overview 4

2 Permutation of Derivations in Sequent Calculus **6**

2.1 Overview 6

2.2 Three Sequent-Style Calculi 7

2.3 Relationships Between the Calculi 8

2.4 Permutations in **LJ** 9

2.5 Weak Normalisation of Permutations 9

2.5.1 The Equivalence of **MJ** and **NJ** 10

2.5.2 Proof that Permutation Reduction is Weakly Normalising 10

3 Formalisation in *Isabelle* **17**

3.1 A Brief Overview of *Isabelle* 17

3.2 An *Isabelle* Object Logic as a Meta-Logic 18

3.2.1 Syntax 18

3.2.2 Logical Rules in the Meta-Logic and the *Isabelle* Meta-Logic 19

3.3 *Isabelle* as a Tool 21

3.4 *Isabelle*'s ASCII notation 23

4 Formalisation in *SEQUEL* **24**

4.1 Introduction to *SEQUEL* 24

4.2 Meta-Theory in a *SEQUEL* Framework 25

4.3 Generalisation of the Method 27

4.4 Using a Logical Framework for Meta-Theory 29

5 A Brief Introduction to Formalisation in *Coq* **30**

5.1 A Quick Overview of *Coq* 30

5.1.1 The Basis of the Type Theory 30

5.1.2 Logical Notation in ASCII 30

5.1.3 Definitions 31

5.1.4 The Minimality Principle and Inversion of Predicates 32

5.1.5 Performing Proofs in *Coq* 33

5.2 Formalisation of Proof Terms in *Coq* 35

6 An Initial Formalisation in *Coq* **37**

6.1 De Bruijn Indices 37

6.2 Formulae, Contexts and Variables 39

6.3 Derivations and Deductions 41

6.3.1 Summary 43

6.4 Conclusions 44

7 A Formalisation in *Coq* Using de Bruijn Indices **45**

7.1 Initial Definitions . . . 45

7.2 Decidability of Relations . . . 46

7.2.1 `Setifb` . . . 47

7.2.2 Lifting . . . 48

7.2.3 The Usefulness of Boolean Functions . . . 49

7.2.4 The Usefulness of Propositional Functions . . . 50

7.3 Translation Functions . . . 51

7.4 Derivations and Deductions . . . 53

7.4.1 Structural Rules . . . 54

7.5 Permutation . . . 55

7.6 Proof Techniques . . . 58

7.6.1 Induction Principles . . . 58

7.6.1.1 Inductions on Simple Inductive Sets . . . 58

7.6.1.2 Induction for More Complex Sets . . . 59

7.6.1.3 Direct Induction over Families . . . 59

7.6.1.4 Induction with Inversion . . . 60

7.6.2 Strong Induction Principles . . . 60

7.7 Summary and Conclusions . . . 62

8 A Formalisation in *Coq* Using Named Variables **63**

8.1 Background of the Coquand-McKinna-Pollack Approach . . . 63

8.2 **NJ** Formalised with Named Abstract Syntax . . . 64

8.2.1 First Order Abstract Syntax for Terms . . . 64

8.2.2 (Restricted) Higher Order Abstract Syntax for Judgements . . . 67

8.2.2.1 The *CMP* Approach for General Judgements and Predicates 69

8.2.3 Complexity of the *CMP* Approach . . . 69

8.3 Scope of the Formalisation 69

9 Related Work: Tools and Techniques **71**

9.1 Introduction 71

9.2 Formalisations Using de Bruijn Indices 71

9.2.1 Strong Normalization of System F in *LEGO* 71

9.2.2 Verification of Algorithm $\mathcal{W}$: The Monomorphic Case 72

9.2.3 Church-Rosser Proofs in *Isabelle/HOL* 73

9.2.4 *Coq* in *Coq* 73

9.3 A Formal Theory of Pure Type Systems 74

9.4 Five Axioms of α-Conversion 74

9.5 *HOL*, *ALF*, *Coq* and *LEGO* 74

9.6 Higher Order Abstract Syntax 75

9.7 Higher Order Abstract Syntax with Induction 76

9.7.1 Restricted Higher Order Abstract Syntax with Induction in *Coq* . . . 77

9.7.2 *HOAS* with Primitive Recursion 77

9.7.3 First Order Logic with Definitions and Natural Number Induction . . 78

10 Conclusions and Further Work **79**

10.1 Frameworks vs. Proof Assistants 79

10.2 Expansion of the Formalisation of the Permutation Theorem 79

10.2.1 New Tactics for *Coq* 80

10.2.2 Rippling 81

10.2.3 The Permutability Theorem for First Order Logic 81

10.2.4 Strong Normalisation of Permutation Reduction 82

10.3 Other Logics, Other Problems 83

10.4 De Bruijn Indices, the *CMP* Method and *HOAS*: Conclusions 83

10.4.1 De Bruijn Indices 83

10.4.2 The *CMP* Method 85

10.4.3 *HOAS* 86

Bibliography **88**

A Primary Definitions and Lemmas in *Coq* **94**

A.1 De Bruijn Index Formalisation 94

A.2 *CMP* Method Formalisation 108

B Full Development in *Coq* using de Bruijn Indices **116**

Chapter 1

Introduction

The Study Of Formal Deductive Systems (*logics*) has a long history, reaching back through the history of mathematics. With the advent of powerful digital computers in the latter half of the twentieth century, we have seen an explosive increase of interest in formal logics, in large part as a tool to understand the operation of those very computers. Increasingly over the last two or three decades, investigations using these formal logics have been carried out in software environments specifically designed for such work. The process of development is fairly clear. A researcher invents a new system which is then implemented in a suitable language or environment and theorems are formally validated within the deductive system, either through interaction, or automatically by using pre-programmed methods. The processes modelled by these logics are complex, and recursive structures common.

1.1 Logical Frameworks

Techniques have been developed over the last two decades to make these investigations easier, in particular to ease the job of defining the new logics in a formal environment. To this end logical frameworks [HP91, HP93] implemented as *ALF*[AGNvS94], *Elf* [Pfe91], *Isabelle* [Pau88] and *SEQUEL* [Tar93] have been developed. These frameworks provide different but internally coherent approaches to the implementation of formal logics, freeing the designer to work on theoretical issues and use of the system rather than tedious details of program correctness and issues of representation in a general purpose language. The resulting implementations are very useful in proving object-level theorems of the logic and for exploring the deductive system. However, the implementation of a logic in a logical framework does not give one access to machine support for meta-level judgements about the logic, as opposed to deductions within the logic. For such theoretical work a pen and paper is still the primary tool for most researchers. Some work has been done with machine-assisted formal meta-theory, but it remains a very small part of the larger field.

The literature on logical frameworks and on special purpose implementations of common logics (e.g. the various *Isabelle* object logics and the various implementations of type theories: *NuPrl* [CA+86], *ALF* [AGNvS94] and *Coq* [BB+96]) contains many varied arguments about the necessity for machine support when performing formal proofs. The issue of *confidence* underlies most of these: *confidence* that the theorem really is a consequence of the axioms and rules of the logic, particularly confidence that one has not missed vital cases in an induction or case-splitting step, and that any definitions are acceptable within the logic. These arguments are no less valid for the study of the logics themselves as for working within these logics. In fact, they may carry more weight. If the modelling power of a logical system depends on, for example, the confluence of its type inference algorithm, then we require assurance that the said algorithm really has that property. Such proofs tend to be long and complex, requiring inductions and case analyses involving a large number of variations on a theme. The phrases "similarly" and "obviously" are very common in such work. It is unusual, though not unknown, that the "similar" proof method in these cases does apply. Consider the following, however: two constructions may appear almost identical, and therefore proofs about the properties may require the same steps. If an error has been made in some related definitions then what is true for one may not hold for the "similar" case. *Proof* is an interactive process, which leads to a deeper understanding of the underlying theory, as well as a mechanical verification of facts. Errors in the formulation, or subtle differences leading to divergent proof requirements, may be missed in the standard informal approach.

Until recently, the machine environments available were not at all suitable to the demands of formal meta-theory. Either the environment simply did not have sufficient logical power to allow the required proofs to be performed or, more commonly, the amount of work required to encode the logics and perform meta-theoretic proofs was prohibitive. Formal meta-theory is an expanding field, however, so we wish to examine some of the environments currently available to see how easy such work now is, how easy it may become, and what direction development of environments should take to encourage this important step forward.

1.2 Terminology

1.2.1 Sequent Calculus and Natural Deduction

We are interested in two kinds of calculi: *sequent calculi* and *natural deduction calculi* [Gen33, Pra65]. A good introduction to the two kinds of calculi can be found in [TS96, §1.3]. In order to study both kinds in a common framework, we will present natural deduction calculi in a *sequent-style* (called the *logistical style* in [Gen34]). [TS96, §2.1.4] presents a sequent-style version of natural deduction. The differences between these kinds of calculi can be seen if we examine the rule for logical conjunction (*and*), written as $\wedge$. For sequent-style natural deduction we might have the following three rules for conjunction:[1]

$$\frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2} \wedge\mathrm{I} \quad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_1} \wedge\mathrm{E}_1 \quad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_2} \wedge\mathrm{E}_2,$$

while for sequent calculus we might have the two rules:

$$\frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2} \wedge\mathrm{R} \quad \frac{\Gamma, F_1, F_2 \vdash F_3}{\Gamma, F_1 \wedge F_2 \vdash F_3} \wedge\mathrm{L}.$$

The two rules $\wedge$I and $\wedge$R are identical, but there are striking differences between the rules $\wedge\mathrm{E}_{1/2}$ and the rule $\wedge$L. The primary difference between a natural deduction calculus and a sequent calculus is that the sequent calculus includes rules which change formulae occurring in the *context* (the sets Γ of formulae).

1.2.2 Proofs, Derivations and Deductions

Since the word *proof* can become overused when discussing meta-theory, we will adopt the following convention: *proof* refers to the proof of a meta-theoretic result; when discussing object-level *proofs*, the words *derivation* or *deduction* will be used, depending on the type of logic being investigated. *Derivations* are proofs within sequent calculi. *Deductions* are proofs within *natural deduction* calculi (even when those calculi are presented in a sequent-style).

1.2.3 Unfolding

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, with formal parameters replaced by actual parameters. So,

[1] Where the F_i are meta-variables for formulae, and Γ is a meta-variable for sets of formulae.

if we have the function *plus* for natural numbers defined by the equations:

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

then unfolding the first application of *plus* in

$$plus(S(S(0)), plus(S(i), j))$$

gives

$$S(plus(S(0), plus(S(i), j)))$$

1.3 The Requirement for a Meta-Logic

Implementations of logics such as first order intuitionistic logic, classical linear logic etc., are coded within the machine environment in a way that allows the user to perform complex derivations/deductions within the logic thus defined. The aim of such work is to prove complex object-level statements. Investigations into the properties of these logics require different tools. To perform such investigations, induction is invariably required at the level of reasoning about derivations/deductions. We wish to be able to define the notion of a derivation/deduction within the system. Even if the logic we are reasoning about has no need for a term assignment system representing the derivations/deductions (as it might not if provability is the only issue of interest), we may want a term assigned to derivations/deductions to aid reasoning at the meta-level. With first order theories, we are interested in the witnessing term when proving formulae, but at the meta-level, we only wish to know that appropriate terms exist, and explicit encodings in a logical framework may complicate the meta-theory without providing any more confidence in the resulting proofs.

1.4 Overview

In this thesis, we will examine three environments: *Isabelle* [Pau88], *SEQUEL* [Tar93] and *Coq* [BB$^+$96]. The first two are found to be unsuited to the work we wish to do. *Coq* is found to be adequate although not ideal. Some work was also done in *ALF* [AGNvS94], but this was never a fully released system and has now been superseded by a new system *HALF*.[2] The methodology of *ALF* (that of directly editing proof terms for Martin-Löf's monomorphic type theory [NPS90]) did not lend itself to work with multiple calculi, particularly with

[2]The implementation of *HALF* is an ongoing project that has no official documentation yet, and is not available outside Chalmers. Some work done in *HALF* has been published, most notably [CN96].

the need for equality reasoning about translated proof terms. The meta-theory we will be exploring in this formal setting is taken from [DP97a] with background material in [DP96]. The informal meta-theory developed there is closely linked with work by Herbelin in [Her94]. The informal development from [DP97a] is shown in §2. Following this, we will briefly examine attempts at formalising these examples using *Isabelle* in §3 and *SEQUEL* in §4. §5 contains a brief overview of the proof assistant *Coq*, and discusses some of the choices made for the formalisations presented in §§6–8. We examine other approaches in §9, briefly looking at other formalisations of meta-theory with particular attention to the approaches. In §10 we draw conclusions about the work presented in the thesis and give some indicators of further possibilities in this area. We briefly examine the extension of the formalisation to cover the example theorems in the universally quantified implicative fragment of first order logic. Extension to the full propositional cases would appear to involve little challenge but would require a fair amount of time to perform the proofs. We also draw conclusions about the relative merits of de Bruijn indices and the named variable syntax used in §8. We compare the tools used for the various formalisations in §§3–8, and indicate the requirements for tools which would better support further work in formal meta-theory. Finally, in §A we highlight some of the important definitions of the *Coq* formalisations and then in §B we give the full development of the formalisation using de Bruijn indices.

Chapter 2

Permutation of Derivations in Sequent Calculus

This chapter contains a brief overview of the theory being formalised. A more complete version can be found in [DP97a].

2.1 Overview

It has long been a piece of logic folklore that two intuitionistic sequent calculus derivations are really the same if, and only if, they correspond to the same natural deduction. To paraphrase [GLT89, p.39]:

> The translation from sequent calculus into natural deduction is not 1–1: different proofs of the same sequent, differing only in the order of application of the rules, have the same translation.
>
> In some sense, we should think of the natural deductions as the true "proof" objects. The sequent calculus is only a system which enables us to work on these objects: $A \vdash B$ tells us that we have a deduction of B under the hypotheses A.

[Kle52] discusses permutability of inferences in sequent calculus without reference to the corresponding natural deductions, and some of his permutations do not maintain equality of the image. Similar ideas may also be found in [Min96]. The relationships between individual sequent calculus derivations can be described using a set of permutations, such that two sequent calculus derivations are inter-permutable if and only if they correspond to the same 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.

But what is 'normal' in this sense? In [DP97a] 'normal' is defined syntactically 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 [Pra65, Fel89], has not been explicitly defined in the early literature. Prawitz [Pra65] does, however, describe the steps of this translation (here called ρ), which is also described in [TS96]. Prawitz' translation is from normal deductions in natural deduction into the sequent calculus. Gentzen [Gen34] described a translation of non-normal natural deductions in the sequent calculus with cut. In fact, the translation is naturally formed as the composition of the translations via an intermediate calculus, the *permutation-free sequent calculus* due to Herbelin in [Her94] and refined by Dyckhoff and Pinto in [DP96]. There are therefore two distinct parts to this work. The new calculus[1] **MJ** must be shown to be isomorphic to natural deduction [DP96] and the reductions must be shown to be normalising with respect to the retraction of **LJ** onto itself via **MJ**.

The permutation reductions in [DP97a] have been shown to be strongly normalising, with some simple extra constraints on their application, in [Sch]. The informal proof of strong normalisation of this system appears as a corollary of a result for another calculus which allows further fine-grained reasoning about the relationship between a derivation in **MJ** and its equivalent derivation in **LJ**. The work in [Sch] has appeared too recently for a formalisation to be performed and the results included here.

2.2 Three Sequent-Style Calculi

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 $\mathbf{NJ}^{+cut}$ and $\mathbf{LJ}^{+cut}$ is well-known, and cut-elimination for $\mathbf{MJ}^{+cut}$ has been shown in [Her94] (see also [DP98]). **NJ** also differs from a standard presentation of the simply-typed λ-calculus in its splitting of terms into *normal* (N) and *applicative* (A)

[1] Called **MJ** in [DP96] to avoid confusion between Herbelin's name LJT in [Her94] and Dyckhoff's different calculus LJT in [Dyc92].

terms. *Normal* terms (**N**) have the form:

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

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

$$\begin{array}{rcl} \mathbf{N} & ::= & \lambda\mathbf{V}.\mathbf{N} \mid an(\mathbf{A}) \\ \mathbf{A} & ::= & ap(\mathbf{A},\mathbf{N}) \mid var(\mathbf{V}) \end{array} \qquad \begin{array}{rcl} \mathbf{M} & ::= & (\mathbf{V}\,;\mathbf{Ms}) \mid \lambda\mathbf{V}.\mathbf{M} \\ \mathbf{Ms} & ::= & [\,] \mid \mathbf{M} :: \mathbf{Ms} \end{array}$$

$$\mathbf{L} \ ::= \ vr(\mathbf{V}) \mid app(\mathbf{V},\mathbf{L},\mathbf{V}.\mathbf{L}) \mid \lambda\mathbf{V}.\mathbf{L}$$

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

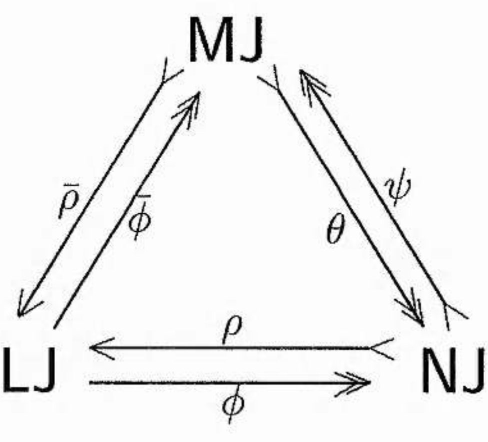
$$|app(x, l_1, y.l_2)| = [ap(x, |l_1|)/y]|l_2|.$$

Taking P, Q, R as meta-variables for formulae and Γ for contexts[2], the rules for the three systems are in table 2.1 on page 11. The judgement forms for each calculus are summarised here:

Calculus (term)	Judgement Form	Calculus (term)	Judgement Form
NJ(N)	$\Gamma \rhd\!\rhd n : P$	**NJ(A)**	$\Gamma \rhd a : P$
MJ(M)	$\Gamma \Rightarrow m : P$	**MJ(Ms)**	$\Gamma \xrightarrow[Q]{} ms : P$
LJ(L)	$\Gamma \rightarrow l : P$		

2.3 Relationships Between the Calculi

Following our definition of the three calculi, we define functions which translate derivation/deduction terms between calculi, and show how the translations interact. These functions (derived from [Gen33, Pra65, DP98]) are shown in table 2.2 on page 12, and various theorems regarding their interaction are shown in table 2.3. These theorems include those showing that translated derivation/deduction terms still derive/deduce the same formula in the same context (theorems N_Admis_$\theta(')$, M_Admis_$\psi(')$, L_Admis_ρ, L_Admis_$\bar{\rho}$, N_Admis_ϕ and M_Admis_$\bar{\phi}$). The names of the theorems (e.g. $\psi\theta$) shown in table 2.3 are derived from the names used in the formalisation described in §7, with names of Greek letters (e.g. `rho`) replaced by the correct symbol (ρ). The diagram below shows how the translation functions relate derivations/deductions in the calculi:



2.4 Permutations in LJ

Now that we have introduced each of the calculi, and the translations between them, we may define a relation permuting derivations in **LJ**. This is the relation shown as $\succ$ in table 2.4. $\succ^*$ is defined as the reflexive transitive closure of $\succ$ in the usual way. Once we have defined the $\succ^*$ relation for untyped terms, we must show the admissibility of sub-term reduction for the new relation (see table 2.6 on page 16, theorems L_Permn_lm, L_Permn_app1 and L_Permn_app2): i.e. that reducibility of a term implies the reducibility of any superterm. The *Weak Normalisation Property* of $\succ^*$ follows from the three theorems Norm_Imperm_L, Norm_L_$\bar{\rho}$ and Norm_Red (see table 2.6), as per the specification of weak normalisation for *abstract reduction systems* in [Klo92, Definition 2.0.3(2)]. The normal form to which terms are rewritten is defined informally in table 2.5.

[DP97a] contained a conjecture that by adding certain side-conditions to the system of reductions the system would be strongly normalising. In [Sch], Schwichtenberg proposed that only the restriction that l_3 must be *fully normal wrt w* for *app_app1* or *app_app2* to be applied, was needed. He then proved strong normalisation for the resulting system as a corollary of a theorem involving another intermediate calculus.

2.5 Weak Normalisation of Permutations

The aim of this work was originally to define an equivalence class of derivations in **LJ** each of which mapped to the same derivation in **MJ** (and, by the bijection between **MJ** and

[2] Contexts are defined to be functions from a finite set of variables to a set of formulae.

NJ, to the same deduction in **NJ**). As the informal exploration continued the equivalence relation was replaced by an oriented reduction relation, and the goal developed into a search for a strongly normalising reduction relation. As a partial step towards this goal, a weakly normalising reduction relation was developed: $\succ$, as shown above. As mentioned in §2.1, some minor modifications of the weakly normalising reduction relation leads to a strongly normalising relation, the proof of which is a corollary of a similar proof in [Sch]. [Sch], however, introduces yet another calculus which further identifies the steps in translation of derivations in **LJ** to derivations in **MJ** (and so to the equivalent deductions in **NJ**). We will ignore the work in [Sch] here, since the formalisation we wish to examine later only covers the weakly normalising permutation reduction relation and **MJ**.

2.5.1 The Equivalence of MJ and NJ

[DP96] (an expanded version of [DP97a]) includes proofs of the equivalence of the full propositional versions of **MJ** and **NJ**. These proofs are performed simply using the obvious mutual induction schemes inferred from the definitions of **M**, **Ms**, **N** and **A**.

2.5.2 Proof that Permutation Reduction is Weakly Normalising

[DP96] also includes a proof of the theorem that the permutation reduction relation defined in table 2.4 is weakly normalising. The major work involved in this is the proof of the lemma called App_Red_M in table 2.6:

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

where $\bar{\rho}$ is the translation function from **M** to **L**:

$\bar{\rho} : \mathbf{M} \to \mathbf{L}$		
$\bar{\rho}(x\,;[\,]) =_{def} vr(x)$		
$\bar{\rho}(x\,; m :: ms) =_{def} app(x, \bar{\rho}(m), z.\bar{\rho}(z\,; ms))$		z new
$\bar{\rho}(\lambda x.m) =_{def} \lambda x.\bar{\rho}(m)$		

Since this is a non-standard recursion ($z\,; ms$ is not a sub-term of $x\,; m :: ms$ in the second definitional equation) a standard inductive argument will not provide us with an appropriate induction hypothesis for conjectures involving $\bar{\rho}$. A *measure* induction principle is therefore defined for performing induction on terms in **M** and **Ms**, which may be used to prove conjectures involving $\bar{\rho}$ such as App_Red_M above. A similar process is used in the formalisation described in §7.6.2.

Table 2.1: Proof Rules for **NJ**, **MJ**, **LJ**.

NJ

$$\frac{\Gamma, x : P \rhd\!\rhd\ n : Q}{\Gamma \rhd\!\rhd\ \lambda x.n : (P \supset Q)} \supset \mathrm{I}$$

$$\frac{\Gamma \rhd a : P}{\Gamma \rhd\!\rhd\ an(a) : P} \text{ AN-Axiom}$$

$$\frac{\Gamma \rhd a : (P \supset Q) \quad \Gamma \rhd\!\rhd\ a : P}{\Gamma \rhd ap(a, n) : Q} \supset \mathrm{E}$$

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

MJ

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

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

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

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

LJ

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

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

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

Table 2.2: Translation functions for proof terms.

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

Table 2.3: Relationships between the calculi

$$\psi\theta: \quad \psi(\theta(m)) = m$$

$$\psi\theta'\psi': \quad \psi(\theta'(a, ms)) = \psi'(a, ms)$$

$$\theta\psi: \quad \theta(\psi(n)) = n$$

$$\theta\psi'\theta': \quad \theta(\psi'(a, ms)) = \theta'(a, ms)$$

$$\textbf{N\_Admis\_}\theta: \quad \frac{\Gamma \Rightarrow m{:}R}{\Gamma \rhd\!\rhd \theta(m){:}R}$$

$$\textbf{N\_Admis\_}\theta': \quad \frac{\Gamma \rhd a{:}P \quad \Gamma \xrightarrow[P]{} ms{:}R}{\Gamma \rhd\!\rhd \theta'(a, ms){:}R}$$

$$\textbf{M\_Admis\_}\psi: \quad \frac{\Gamma \rhd\!\rhd n{:}R}{\Gamma \Rightarrow \psi(n){:}R}$$

$$\textbf{M\_Admis\_}\psi': \quad \frac{\Gamma \rhd a{:}P \quad \Gamma \xrightarrow[P]{} ms{:}R}{\Gamma \Rightarrow \psi'(a, ms){:}R}$$

$$\bar{\phi}\bar{\rho}: \quad \bar{\phi}(\bar{\rho}(m)) = m \qquad \rho\theta\bar{\rho}: \quad \rho(\theta(m)) = \bar{\rho}(m)$$

$$\theta\bar{\phi}\phi: \quad \theta(\bar{\phi}(l)) = \phi(l) \qquad \phi\rho: \quad \phi(\rho(n)) = n$$

$$\textbf{L\_Admis\_}\bar{\rho}: \quad \frac{\Gamma \Rightarrow m{:}R}{\Gamma \rightarrow \bar{\rho}(m){:}R} \qquad \textbf{M\_Admis\_}\bar{\phi}: \quad \frac{\Gamma \rightarrow l{:}R}{\Gamma \Rightarrow \bar{\phi}(l){:}R}$$

$$\textbf{N\_Admis\_}\phi: \quad \frac{\Gamma \rightarrow l{:}R}{\Gamma \rhd\!\rhd \phi(n){:}R} \qquad \textbf{L\_Admis\_}\rho: \quad \frac{\Gamma \rhd\!\rhd n{:}R}{\Gamma \rightarrow \rho(n)R}$$

Table 2.4: Permutations of Derivations in **LJ**

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

Table 2.5: Normal Forms of terms in **L** wrt $\succ$

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 var-normal with respect to the variable y.

l is var-normal wrt x if it is

equal to $vr(x)$, or

of the form $app(x, l_1, y.l_2)$

where

l_1 is normal;

l_2 is var-normal wrt y;

$x \notin l_1, l_2$.

Table 2.6: Subject Reduction and Weak Normalisation

L_Admis_Perm1 :	$\dfrac{l_1 \succ l_2 \quad \Gamma \to l_1 : R}{\Gamma \to l_2 : R}$
L_Admis_Permn :	$\dfrac{l_1 \succ^* l_2 \quad \Gamma \to l_1 : R}{\Gamma \to l_2 : R}$
L_Permn_lm	$\dfrac{l_1 \succ^* l_2}{\lambda x.l_1 \succ^* \lambda x.l_2}$
L_Permn_app1	$\dfrac{l_1 \succ^* l_2}{app(x, l_1, y.l_3) \succ^* app(x, l_2, y.l_3)}$
L_Permn_app2	$\dfrac{l_2 \succ^* l_3}{app(x, l_1, y.l_2) \succ^* app(x, l_1, y.l_3)}$
Norm_Imperm_L :	Normal(l) $\Rightarrow \sim l \succ l_0$
Norm_L_$\bar{\rho}$:	Normal($\bar{\rho}(m)$)
App_Red_M :	$app(x, \bar{\rho}(m_1), y.\bar{\rho}(m_2)) \succ^* \bar{\rho}(sub(x, m_1, y, m_2))$
Norm_Red :	$l \succ^* \bar{\rho}(\bar{\phi}(l))$

Chapter 3

Formalisation in *Isabelle*

3.1 A Brief Overview of *Isabelle*

As with most logic software, *Isabelle* uses an ASCII notation for the non-ASCII symbols of logic. §3.4 gives a basic introduction to this, but throughout this chapter standard logical notation will be used for ease of reading.

Isabelle is a highly modular system with many incompatible object logics developed around a single core: the *Pure* system.

The *Pure* system allows for the definition of sorts, subsorts and types. Types may inhabit the global sort, the primitive sort *logic*, or any of the defined sorts or subsorts. Polymorphism is implemented by means of the sorts. Types are simply declared and constructors for the types defined as functions (there is no distinction between general function definitions and constructor functions for types).

Isabelle's meta-logic is implemented in a natural deduction style [Pra65] using the same symbol ($\Longrightarrow$) as the connective between the premises and conclusion, and the connective between assumptions and premises. So the rule of implication introduction which is usually represented in natural deduction as:

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \supset B} \supset \mathrm{I}$$

would be represented in *Isabelle* as

$$(A \Longrightarrow B) \Longrightarrow (A \supset B)$$

This overloading of $\Longrightarrow$ is a confusing aspect of the language of *Isabelle* for new users, but is not a serious problem. The *Isabelle* meta-logic is not designed to be used directly as a proof system. *Isabelle* was designed to allow users to implement the logic in wish they wish to prove theorems. For our work in formal meta-theory, we must therefore define our own meta-logic in which proofs about logical systems (such as **NJ**, **MJ LJ**) may be performed. The semantics of our meta-logical connectives will be defined by relating their meaning to the *Isabelle* connectives. Various packages supplied with the basic system (such as the equational reasoning package) in fact require rules of a specific form relating the new *Isabelle* object logic connective to an *Isabelle* meta-level connective.

3.2 An *Isabelle* Object Logic as a Meta-Logic

Since the *Isabelle* meta-logic is designed for the implementation of object logics, and not for direct use as a proof system, a three-level hierarchy must be used. At the bottom there is the *Isabelle* meta-logic. Above that is the meta-logic used for reasoning about the systems **NJ** and **MJ**. The meta-logic we implement as an *Isabelle* object logic is intuitionistic first-order logic with built-in size induction schemes, simple arithmetic (the natural numbers, addition and a "less than" relation), and with first order terms. At the top are the systems **NJ** and **MJ** themselves. The different levels are used as shown in the table below

Logic	Use
Object Logics **NJ** and **MJ**	Proof of Theorems
Meta-Logic	Proving Properties of Proof Systems
Isabelle Meta-Logic	Tactics and Forward Proof

The aim of this work is to provide machine support for the meta-logic. It is not an aim to make the object logics particularly usable within this system, although they must of course be correctly defined.

3.2.1 Syntax

We define the *sort* of *terms*, which includes the deduction terms for **NJ**, derivation terms for **MJ**, formulae, object-level variables and hypothesis lists. Quantification for the meta-logic is allowed over terms of specific types.

The usual symbol $\Longrightarrow$ is used for the *Isabelle* meta-logic implication, and any free variables are implicitly universally quantified within the *Isabelle* meta-logic. The equality relation within the *Isabelle* meta-logic is represented as $\equiv$. This *Isabelle* meta-logic equality is defined with respect to syntactic equality of terms, but it is usual to extend it to include *Isabelle* object logic (our meta-logic) equality.

The following symbols are used for the various connectives required for the meta-logic:

$=$	Equality of proof terms
$\longrightarrow$	Implication
$\wedge$	Conjunction
$\forall$	Universal quantification of terms over meta-logical predicates
$\rhd$	Derivability in **A**
$\rhd\!\rhd$	Derivability in **N**
$\Rightarrow$	Derivability in **M**
$\xrightarrow[F]{}$	Derivability in **Ms**[1]

Since only the implicational fragments of **MJ** and **NJ** are dealt with, the only object-level connective required is implication ($\supset$). To illustrate the use of some of the above connectives, we show the rule of our meta-logic which performs case analysis of a term in **M**.

$$((\forall x.\forall ms.(m = (x\,;ms)) \longrightarrow P(x\,;ms)) \wedge (\forall x.\forall m'.(m = \lambda x.m') \longrightarrow P(\lambda x.m'))) \Longrightarrow P(m)$$

where P is some predicate abstracted (at the *Isabelle* meta-level) over objects of type **M**.

3.2.2 Logical Rules in the Meta-Logic and the *Isabelle* Meta-Logic

Isabelle supports both forwards and backwards chaining as methods of proof. Backwards chaining involves the usual method of applying a rule to the current goal and having a set of sub-goals returned. When supplied with a conjecture G to prove, *Isabelle* automatically applies the identity implication rule ($G \Longrightarrow G$) to it, setting the basic goal to G and initialising the sub-goaler to a single sub-goal of G also. Forward chaining allows a user to combine rules and axioms to produce a new rule, which may or may not depend on sub-goals. In this way, the user may build up a complete proof tree applicable to the current goal. Proofs are seldom performed this way, although completely deterministic tactics may be built and named in this manner, avoiding the need to program them in ML.[2]

[1] Where F is a Formula or a variable ranging over formulae.

[2] Non-deterministic tactics still require programming in ML, however.

The rules for the *Isabelle* meta-logic are not used for proving theorems in general, but are for writing tactics in ML, and for writing ML tacticals to generate tactics. The useful rules for proving theorems are those programmed into each object logic, so we need to implement such rules as part of our meta-logic.

There are certain ML functionals written to help define sets of rules when implementing object logics. These require the prior provision of object logic (our meta-logic) versions of common connectives as arguments. The ML functionals then produce rule sets derived from these. One of the most commonly used sets is the equality reasoning, which takes a set of equality rules defined using an object level equality, and rules specifying that the object level implication and equality are derivable from the *Isabelle* meta-logic implication and equality, and returns a tactic which will use the provided equalities as a rewriting system and rewrite to a fixpoint in both the current goals and their local assumptions. There is no attempt to prevent looping of these rules, and it is up to the programmer of the object logic to ensure that the equality rules are appropriately ordered to avoid this.

In the implementation of the example, the rules linking the meta-logic and the *Isabelle* meta-logic connectives are:

$$(a = b) \Longrightarrow (a \equiv b)$$

$$(P \Longrightarrow Q) \Longrightarrow (P \longrightarrow Q).$$

The first of these defines our meta-logic equality relation as an equality relation for the system. The definition of our equality relation must include (but is not restricted to) rules showing symmetry, reflexivity and transitivity for the relation. We may then use an ML functional to provide a simplification tactic performing rewriting using our meta-logic equality. This simplification includes unfolding of functions such as θ which have been defined using the *Isabelle* meta-logic equality ($\equiv$).

The second of these is the definition of our meta-logical implication connective ($\longrightarrow$). We are stating that we may derive $P \longrightarrow Q$ if we can derive Q by assuming P.

To prove properties of the proof terms, such as theorems $\psi\theta$ and $\psi\theta'\psi'$, we require an induction principle. Again, we must define an induction principle manually within the *Isabelle* meta-logic for each class of objects upon which we wish to perform induction. This is where we find the greatest barrier to using *Isabelle* for this work. Given the complex, one might almost say unreadable, nature of the *Isabelle* source text, definition of an induction principle for complex, mutually defined, inductive objects becomes a non-trivial task. Mistakes are not easy to spot, nor is one ever completely sure that one's implementation is absolutely correct.

For example in order to perform induction proofs for simultaneous proof of $\psi\theta$ and $\psi\theta'\psi'$, the following rules had to be encoded into *Isabelle*:

- A definition of the size function for objects of types **M**, **Ms**, **A**and **N**, including objects formed from the translation functions ϕ and θ.
- A principle of induction over the size of an object.
- A number of rules about natural numbers including an ordering function.
- Case analyses of objects of type **M** and **Ms**.

Many of these are quite complex rules, and the prospect of having to implement them individually for each type of proof object etc. in each new logic for which meta-theory is required would be a waste of time, as it would lend little extra confidence in the results for much extra work.

There is yet more work involved in defining rules to allow the proof of theorems such as **N_Admis_**θ. Either a new induction principle for proof on the structure or size of derivations is needed or two versions of each rule in the object logic are needed — an introduction and elimination version for assumptions and goals involving derivations assumed to be correct.[3]

Therefore manual implementation of these principles appears to be a dead end in *Isabelle.* So we come to the requirement for writing a new top-level which uses *Isabelle* as a proof engine and accepts definition of inductive objects and functions, returning appropriate induction principles, from which we may derive appropriate structural induction schemas. Use of one of the existing *Isabelle* object logics would also be possible. The *HOL* object logic (re-implementing the *HOL* theorem prover [GM93]) includes facilities for automatic derivation of induction principles, but is based on classical higher order logic. Since most proof theory (even that studying classical logics) is done constructively then using a system such as *Isabelle*/HOL to formalise such work would seem inappropriate.

3.3 *Isabelle* as a Tool

Isabelle has a medium-sized community active in using object logics and in programming new object logics. There is a smaller community working on improving *Isabelle* and on programming more general functions and functionals in ML for use with the system (for

[3] A similar problem was encountered when attempting an implementation in the sequent notation of *SEQUEL*(see §4).

an overview see [Pau95a]). There is constant development of the system, for instance four releases of upgrades to the system were made in 1995, these improving the major overhaul of the system released late in 1994. Further upgrades to the *Isabelle*-94 system have been released regularly since 1995.

Very few of the commonly-used large systems are completely stable: A few major and a number of minor upgrades of *Isabelle* have been released in the last two years. Work in the area of machine supported logic is therefore always requiring maintenance. How much maintenance is needed for each upgrade depends upon both the nature of the upgrade and the nature of the work undertaken. The *Isabelle* development team usually produce a program which can transform the majority of proof scripts into new versions, although some interaction may be necessary to complete this properly. The scripting capabilities of *Isabelle* are adequate to alleviate this problem in the main. Tactics and tacticals may often need major overhauls to keep up with the latest version, and this is another reason why writing large amounts of code on top of a specific version of *Isabelle* does not appear to be a very attractive method of producing generally useful machine-assistance for meta-theoretic work, given the regularity of the upgrade releases.

The documentation of the system is very varied, even within the areas of meta-programmer, programmer and user documentation. Some parts of each type of user's area of interest are very well-documented, while some are barely touched and others require one to look at the original code to see how the system operates. While there is a good introduction to using the *Isabelle* system for performing proofs in existing object logics in [Kal94], there is no similar paper introducing the basics of writing object logics, and one must wade through the large [Pau94] which includes many internal technicalities mixed in with the necessary information to start writing an *Isabelle* object logic.

From a user's point of view, *Isabelle* is neither very easy nor very difficult to use. The interface could be much improved, but that could be said of most freely available academic-written software, since the interface is the least interesting part of the work for those writing these complex systems. The proof paradigm is a little odd for someone more used to automated systems using a sequent-style calculus, and there are certain obvious top-level controls not present where they might be expected. These problems are being addressed slowly by the growing community of *Isabelle* programmers and meta-programmers, and support for users is currently very good amongst those on the electronic mailing list devoted to it. Whether these situations will continue as and when the user community grows is difficult to judge. Given the difficulties involved in programming *Isabelle* for use as a general tool for machine-assisted meta-theory,[4] it would appear to be a poor candidate for further development. This conclusion also appears in [BC93].

3.4 *Isabelle*'s ASCII notation

To give a flavour of the *Isabelle* ASCII notations, here are some of the connectives and predicates mentioned in §3.2.1 with their ASCII notation. The *Isabelle* meta-logic symbols are provided by the system, whereas the symbols for **MJ** and **NJ** and the meta-logic are defined using the complex *Isabelle* mixfix system. The system is moderately good at representing what is wanted, although the documentation is somewhat obscure, and the type system leaves the parsing difficult to manage.

Symbol	ASCII
Isabelle Meta-Logic	
$\forall$	!!
$\Longrightarrow$	==>
Meta-Logic	
$\longrightarrow$	-- >
$\wedge$	^
$\forall x.P(x)$	ALL x. P(x)
Object Logic	
$\Gamma, x{:}P, \Delta \xrightarrow[P]{} \mathrm{Ms}{:}R$	\$H, x:P, \$G – (P) – – > Ms:R

[4] Especially the problems with implementation of induction schemes.

Chapter 4

Formalisation in *SEQUEL*

4.1 Introduction to *SEQUEL*

SEQUEL [Tar93, Tar97] is a logical framework in the LCF [GMW79] style. It has an ASCII syntax for representing single-conclusion sequents in the style of a typed lambda calculus. Rewriting rules may be defined on the terms or types of the sequents. A logic specified by these sequents is compiled into Common Lisp (with a type checker added).

The propositions of *SEQUEL*'s notation are expressions of the form $w * t$, so the rule for a non-term propositional calculus rule $\wedge R$ might be written:

```
:name And-R
<A> |- P? * thm
<A> |- Q? * thm
thus
<A> |- (P? & Q?) * thm
```

If we are encoding a term calculus, however, the natural method of representation would be:

```
:name And-R
<A> |- t1? * P?
<A> |- t2? * Q?
thus
<A> |- (pair t1? t2?) * (P? & Q?)
```

These ASCII representations, although necessary for programming *SEQUEL*, are more difficult to read than the more usual forms, so for the remainder of this chapter such rules will be written

$$\frac{\Gamma \vdash t_1 : P \quad \Gamma \vdash t_2 : Q}{\Gamma \vdash pair(t_1, t_2) : (P \wedge Q)} \wedge\text{R}.$$

4.2 Meta-Theory in a *SEQUEL* Framework

In order to work on the metatheoretic level within a *SEQUEL* framework, we define the propositions to be of the form

`(der X G D)` or `(der MS F G D)`

where

$$\begin{aligned} X &::= A \mid N \mid M \\ G &::= nil \mid (concons\ D\ G) \mid \gamma \\ D &::= t : F \end{aligned}$$

and where F are formulae, t terms of **A**, **N**, **M**, **Ms**, and γ the object logic contexts. `(der X G D)` represents a deduction in **NJ** (if `X` is in **A** or **N**) or a derivation in **MJ** (if `X` is in **M**), and `(der MS F G D)` represents derivations in **MJ** where `F` is the "stoup" formula which appears under the sequent arrow, e.g. $P \supset Q$ in the conclusion of the rule:

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

We also define the standard intuitionistic predicate logic connectives, equality between terms or formulae, unfolding of functional expressions, and conditions pertaining to binding of a variable to a formula in a context (`G`). Again these ASCII representations, although necessary for *SEQUEL*, will not be given here. Similar representations are used for the predicate logic connectives between terms.

We need to implement two proof methods as part of the definition of the meta-logic — proof by induction on the size of, and case analysis of, proof terms. Case analysis is a simple matter to encode, but induction is more difficult. We define a general method of proof by induction, dependent on the definition of a polymorphic function *size*:

$$\frac{\Gamma \vdash \forall x : \tau.(\forall y : \tau.((size(y) < size(x)) \supset \Delta[y/z]) \supset \Delta[x/z])}{\Gamma \vdash \forall z : \tau.\Delta}.$$

This is in fact a single principle which does not cover mutual definitions. It is possible to make use of this method for mutual recursive types using the following general approach.

Say we have two predicates $P_0 : A \to Prop$, $P_1 : B \to Prop$, where A and B are mutually recursively defined sets and P_0 and P_1 are mutually recursively defined predicates. If we wish to prove:

$$\forall a : A.P_0(a)$$

then we prove:

$$\forall b : B.(\forall a : A.size(a) < size(b) \supset P_0(a)) \supset P_1(b)$$

using induction on the size of b, and then proceed to prove the required theorem by induction on the size of a.

To illustrate the techniques used in this development, we take the example theorems **N_Admis_θ** and **N_Admis_θ'**.
After translation,**N_Admis_θ** appears as the conjecture:

$$\vdash \forall m : \mathbf{M}.(\gamma \Rightarrow m : R)) \supset ((\gamma \rhd\!\rhd \theta(m) : R)$$

and after applying size induction we are left with the conjectures:

$$m : \mathbf{M},\ x : \mathbf{V},\ ms : \mathbf{Ms},\ (m = (x\,;ms)),\ (\gamma \Rightarrow m : R),$$
$$\text{Ind-Hyp} \vdash (\gamma \rhd\!\rhd \theta(m) : R)$$

$$m : \mathbf{M},\ x : \mathbf{V},\ m_1 : \mathbf{M},\ (m = (\lambda x.m_1)),\ (\gamma \Rightarrow m : R),$$
$$\text{Ind-Hyp} \vdash (\gamma \rhd\!\rhd \theta(m) : R)$$

where Ind-Hyp is the assumption

$$\forall w_1 : \mathbf{M}.(\forall w_2 : \mathbf{F}.(\forall w_3 : \mathbf{Context}.((size(w_1) < size(m)) \supset (w_3 \Rightarrow w_1 : w_2)))).$$

In the hypotheses of the first case $(m = (x\,;ms))$, we are assuming $(\gamma \Rightarrow (x\,;ms) : R)$.[1] This sequent can only be formed in a valid derivation (in **MJ**) as the conclusion of the rule

$$\frac{\Gamma \Rightarrow \mathrm{M} : P \quad \Gamma \xrightarrow[Q]{} \mathrm{Ms} : R}{\Gamma \xrightarrow[P \supset Q]{} \mathrm{M} :: \mathrm{Ms} : R} \supset \mathrm{S},$$

so the γ context in our example must include (for some formula P) the assumption $x : P$, and we may also assume $(\gamma,\ x : P \xrightarrow[P]{} ms : R)$.[2]

$$m : \mathbf{M},\ x : \mathbf{V},\ P : \mathbf{F},\ ms : \mathbf{Ms},\ (m = (x\,;ms)),\ (x : P \in \gamma),$$
$$(\gamma \xrightarrow[P]{} ms : R),\ \text{Ind-Hyp} \vdash (\gamma \rhd\!\rhd \theta(x\,;ms) : R)$$

where P is a new formula. (We may want to delay our choice of P, since it can be any formula, in which case we would use a place-holder variable and check that any instantiation was a formula. In this case, we need a new formula here.)

[1] By substituting $(x\,;ms)$ for m in $(\gamma \Rightarrow m : R)$.

[2] We are effectively inverting an assumption. See §5.1.4 for more details on inversion of assumptions in *Coq*.

Looking to the goal, we can unfold $\theta(x\,;ms)$ to $\theta'(var(x), ms)$. Instantiating the restricted form of **N_Admis_θ'** to[3]:

$$(((\gamma \rhd var(x) : P) \wedge (\gamma \xrightarrow[P]{} ms : R)) \supset (\gamma \rhd\!\rhd \theta'(var(x), ms) : R))$$

and adding it as an assumption we get:

$$\begin{gathered} m : \mathbf{M},\ x : \mathbf{V},\ P : \mathbf{F},\ ms : \mathbf{Ms},\ (m = (x\,;ms)), \\ (x : P \in \gamma),\ (\gamma \xrightarrow[P]{} ms : R), \\ (((\gamma \rhd var(x) : P) \wedge (\gamma \xrightarrow[P]{} ms : R)) \supset (\gamma \rhd\!\rhd \theta'(var(x), ms) : R)) \\ \text{Ind-Hyp} \vdash (\gamma \rhd\!\rhd \theta'(var(x), ms) : R) \end{gathered}$$

We use the implication-left rule to proceed to the following goals:

$$\begin{gathered} m : \mathbf{M},\ x : \mathbf{V},\ P : \mathbf{F},\ ms : \mathbf{Ms},\ (m = (x\,;ms)), \\ (x : P \in \gamma),\ (\gamma \xrightarrow[P]{} ms : R), \\ \text{Ind-Hyp} \vdash (\gamma \rhd var(x) : P), \end{gathered}$$

$$\begin{gathered} m : \mathbf{M},\ x : \mathbf{V},\ P : \mathbf{F},\ ms : \mathbf{Ms},\ (m = (x\,;ms)), \\ (x : P \in \gamma),\ (\gamma \xrightarrow[P]{} ms : R), \\ \text{Ind-Hyp} \vdash (\gamma \xrightarrow[P]{} ms : R) \end{gathered}$$

The second of these follows immediately. Looking at the first, we see that the goal

$$(\gamma \rhd var(x) : P)$$

is of a form that might be discharged via the A-Axiom rule:

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

provided we can show that $x : P \in \gamma$. This is one of the hypotheses, so we have proved the main conjecture for the case of $m = (x\,;ms)$.

4.3 Generalisation of the Method

The interesting points of this proof were the uses of the rules of **NJ** and **MJ** in the hypotheses and goal. The uses we made, informally, of these rules can now be formalised below and, through analogy, appropriate *SEQUEL* axioms can now be coded for all the rules of **NJ** and **MJ**.

[3] Together with an extra premise which can be proved from the inductive hypothesis Ind-Hyp.

$$\frac{\Gamma \ \vdash \ x : p? \in \gamma \quad \Gamma \ \vdash \ \gamma \xrightarrow[p?]{} ms? : r?}{\Gamma \ \vdash \gamma \Rightarrow (x;\ ms?) : r?} \text{ Select-G}$$

$$\frac{\Gamma \ \vdash \ ((x : p?) \mid \gamma) \Rightarrow m? : q?}{\Gamma \ \vdash \gamma \Rightarrow (\lambda\ x.m?) : (p? \supset q?)} \text{ Abstract-G}$$

$$\frac{x : p? \in \ \gamma,\ \gamma \xrightarrow[p?]{} ms? : r?,\ \Gamma \ \vdash \ \Delta}{\gamma \Rightarrow (x;\ ms?) : r?,\ \Gamma \ \vdash \Delta} \text{ Select-H}$$

$$\frac{(p? = (q? \supset r?)),\ ((x : q?) \mid \gamma) \Rightarrow m? : r?,\ \Gamma \ \vdash \ \Delta}{\gamma \Rightarrow (\lambda\ x.m?) : p?,\ \Gamma \ \vdash \Delta} \text{ Abstract-H}$$

$$\frac{}{\Gamma \ \vdash \gamma \xrightarrow[p?]{} nil : p?} \text{ Meet-G}$$

$$\frac{\Gamma \ \vdash \ \gamma \Rightarrow m? : p? \quad \Gamma \ \vdash \ \gamma \xrightarrow[q?]{} ms? : r?}{\Gamma \ \vdash \gamma \xrightarrow[(p? \supset q?)]{} (m? :: ms?) : r?} \supset \text{S-G}$$

$$\frac{(p? = q?),\ \Gamma \ \vdash \ \Delta}{\gamma \xrightarrow[p?]{} nil : q?,\ \Gamma \ \vdash \Delta} \text{ Meet-H}$$

$$\frac{(p? = (q? \supset r?)),\ \gamma \Rightarrow m? : q?,\ \gamma \xrightarrow[r?]{} ms? : s?,\ \Gamma \ \vdash \ \Delta}{\gamma \xrightarrow[p?]{} (m? :: ms?) : s?,\ \Gamma \ \vdash \Delta} \supset \text{S-H}$$

$$\frac{\Gamma \ \vdash \ \gamma \rhd a? : p?}{\Gamma \ \vdash \gamma \rhd\!\!\rhd (\text{An } a?) : p?} \text{ AN-Axiom-G}$$

$$\frac{\Gamma \ \vdash \ ((x : p?) \mid \gamma) \rhd\!\!\rhd n? : q?}{\Gamma \ \vdash \gamma \rhd\!\!\rhd (\lambda\ x.n?) : (p? \supset q?)} \supset \text{I-G}$$

$$\frac{\gamma \rhd a? : p?,\ \Gamma \ \vdash \ \Delta}{\gamma \rhd\!\!\rhd (\text{An } a?) : p?,\ \Gamma \ \vdash \Delta} \text{ AN-Axiom-H}$$

$$\frac{(p? = (q? \supset r?)),\ ((x : q?) \mid \gamma) \rhd\!\!\rhd n? : r?,\ \Gamma \ \vdash \ \Delta}{\gamma \rhd\!\!\rhd (\lambda\ x.n?) : p?,\ \Gamma \ \vdash \Delta} \supset \text{I-H}$$

$$\frac{\Gamma \ \vdash \ x : p? \in \gamma}{\Gamma \ \vdash \gamma \rhd (\text{Var } x) : p?} \text{ A-Axiom-G}$$

$$\frac{\Gamma \ \vdash \ \gamma \rhd a? : (p? \supset q?) \quad \Gamma \ \vdash \ \gamma \rhd\!\!\rhd n? : p?}{\Gamma \ \vdash \gamma \rhd (\text{Ap } a?\ n?) : q?} \supset \text{E-G}$$

$$\frac{x : p? \in \ \gamma,\ \Gamma \ \vdash \ \Delta}{\gamma \rhd (\text{Var } x) : p?,\ \Gamma \ \vdash \Delta} \text{ A-Axiom-H}$$

$$\frac{\gamma \rhd a? : (p? \supset q?),\ \gamma \rhd\!\!\rhd n? : p?,\ \Gamma \ \vdash \ \Delta}{\gamma \rhd (\text{Ap } a?\ n?) : q?,\ \Gamma \ \vdash \Delta} \supset \text{E-H}$$

So for the eight rules of **MJ** and **NJ**, we produce sixteen rules for our meta-logic. In general, if we have a rule in the object system of the form:

$$\frac{a' : A',\ \Delta \vdash b' : B' \quad \Delta \vdash c : C}{a : A,\ \Delta \vdash b : B}\ \text{Rule},$$

then we need two rules in the meta-system of the form:

$$\frac{\begin{array}{cc}\Gamma \vdash ((a' : A' :: (\gamma \backslash a : A)) \Rightarrow b' : B') & \Gamma \vdash ((\gamma \backslash a : A) \Rightarrow c : C) \\ \Gamma \vdash (\gamma \Rightarrow b' : B') & \Gamma \vdash (a : A \in \gamma)\end{array}}{\Gamma \vdash (\gamma \Rightarrow b : B)}\ \text{Rule-G},$$

$$\frac{(D = B),\ (a : A \in \gamma),\ ((a' : A' :: (\gamma \backslash a : A)) \Rightarrow b' : B'),\ ((\gamma \backslash a : A) \Rightarrow c : C) \vdash \Delta}{(\gamma \Rightarrow a : D),\ \Gamma \vdash \Delta}\ \text{Rule-H}.$$

Together with these rules, a specification of how the γ contexts are handled is required, but that is a simple mechanical process.

Conversion of the single object-level rule to the more complex meta-level rules might be automated, although there are some problems with this, most notably with the formalisation of side-conditions on rules. *SEQUEL* includes a fast, easy to use method of specifying side-conditions as guards on the application of rules, which might be very difficult to translate from object- to meta-level. Using extra sequents — while a slower, more cumbersome method — might provide the answer to these problems.

We shall see in the later sections on formalisation in *Coq*, that this process has already been automated in a very general fashion in proof assistants such as *LEGO* and *Coq*. *Rule-G*'s definition is part of the standard definition-time analysis of a recursive propositional function, while *Rule-H* is an *Inversion Lemma* on the propositional function (see §5.1.4 for details of *Inversion* in *Coq*).

4.4 Using a Logical Framework for Meta-Theory

Given its basic design, it was always obvious that *SEQUEL could* be used for defining frameworks for meta-theoretic proofs. As with Pure *Isabelle* however, it is clear that a great deal of work would be involved in developing a system for performing formal meta-theory in any logical framework. A more constrained system with a recursive definition mechanism and, particularly, the automatic production of induction principles, would appear to be required. A number of such systems are available, and in the next few chapters we examine various formalisations in the proof assistant *Coq*, which fulfills these requirements.

Chapter 5

A Brief Introduction to Formalisation in *Coq*

5.1 A Quick Overview of *Coq*

Coq [BB+96] is 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 thesis are noted below.

5.1.1 The Basis of the Type Theory

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 since the system automatically keeps track of universes above the base cases.

5.1.2 Logical Notation in ASCII

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

Universal quantification is represented by round brackets; e.g. symmetry of equality in a set A would be stated `(x,y:A)x=y->y=x`.

`->` is used both for function typing and to represent logical implication. Conjunction is represented as /\ and disjunction as \/.

5.1.3 Definitions

Three basic definition mechanisms are used: `Inductive` for defining objects and families of sorts `Prop` and `Set`; `Recursive Definition` and `Fixpoint` for functions. Thus the definition[1] of natural numbers (`nat`) in *Coq* is:

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

Mutual `Inductive` definitions are allowed using a `Mutual...with...` construct so, for example, the mutual definition of *even* and *odd* predicates on natural numbers would be:

```
Mutual Inductive
    even: nat->Prop :=
        even_O : (even O) |
        even_s_odd : (n:nat)(odd n)->(even (S n))
with
    odd : nat->Prop :=
        odd_s_even : (n:nat)(even n)->(odd (S n)).
```

The addition function may be defined thus:

```
Recursive Definition
    plus:nat->nat->nat :=
        O j => j |
        (S i) j => (S (plus i j)).
```

Function definition using the `Recursive Definition` syntax is restricted to (higher order) primitive recursion. `Fixpoint` [Gim94] is, as the name suggests, a recursive fixpoint operator which allows definition of (mutual) recursive functions using case analysis via the `Case` and `Cases` operators. The addition function could therefore also be defined in the following two ways:

[1] The number 0 is a reserved token in *Coq*, so the letter O is used.

```
Fixpoint
    plus [i:nat]:nat->nat :=
        [j:nat]<nat>Case i of
            j
            [i':nat](S (plus i' j))
        end.
```

The construct `Case i of` deconstructs the term `i` into its inductive definitional clauses (here `O` and `(S i')` for some `i':nat`), and any new variables are named. The first clause has no new variables because `i` has been decomposed to a ground clause of `O`. A recent innovation (and a more readable syntax) uses the new construct `Cases` [BB+96, §11], which extends `Case` deconstruction to dependent types using a syntax more like the functional programming language ML:

```
Fixpoint
    plus [i:nat]:nat->nat :=
        [j:nat]Cases i of
            O => j |
            (S i') => (S (plus i' j))
        end.
```

`Recursive Definition` is useful since it is integrated into a simplifier tactic (called by the command `Simpl`). To allow unfolding of `Fixpoint` definitions, each line of the definition must be proved as a named lemma and `Rewrite` with the name as argument applied. The `Cases` construct is a recent innovation in *Coq*, and is thus not always used in the work presented in this thesis. `Recursive Definition` has, technically, been superseded by `Fixpoint` in *Coq*, but is still part of the system for backwards compatibility, and because the simplifier tactic has not yet been updated.

5.1.4 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 minimal object satisfying the rules as stated in the definition: i.e. all objects which are a member of the type (family) must have been constructed by the clauses defining the type (family). 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_O : (i:nat)(lt O (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 (`lt_O`): `(n:nat)(lt O (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=O /\ j=(S n)` or `i=(S m) /\ j=(S n) /\ (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 (sub-)conjecture [BB+96, Ch.8].

5.1.5 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 term which inhabits that type. Unlike *ALF*, in which the user directly constructs the term, construction of the term in *Coq* is done by the program, 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.

We may envisage a completed proof (in *CIC*) as a tree of sequents such as:

$$\cfrac{\cfrac{\cfrac{}{\Gamma \vdash t_2 : (O : \mathtt{nat})}\ O}{\Gamma \vdash t_1 : ((S\ O) : \mathtt{nat})}\ S}{\Gamma \vdash t_0 : ((S\ O) =_{\mathtt{nat}} (S\ O))}\ refl\_eq$$

where the t_i are terms of *CIC*, and Γ is the current environment (which includes definitions and local assumptions). Unless we request *Coq* to print out the t_i, we shall never see them. Mostly the user is not concerned with these terms unless they are programming tactics. In order to prove the fact that $1 = 1$ in *Coq* (the statement above), we present this as a type. Since 1 is a ground term, we require no quantifiers (as shown). When we present *Coq* with such a term as a named or un-named conjecture (via the `Lemma` or `Goal` commands), a partial proof tree is initiated. This partial proof tree contains the initial sequent:

$$\Gamma \vdash t? : ((S\ O) =_{\mathtt{nat}} (S\ O))$$

where $t?$ is a placeholder for a term. As we progress through the proof, this placeholder will gradually be refined into a proper term of *CIC*. Giving the command `Apply refl_eq`, which tells *Coq* to apply the lemma stating reflexivity of equality, the term $t?$ would be replaced by the term witnessing `refl_eq`, with a place-holding term for the proof that the two arguments to equality (which must be syntactically equal) are of the correct type. The rest of this, very simple, proof is performed completely automatically by the type-checking engine of *Coq*, according to the definition of the natural numbers (`nat`).

We next illustrate the display of current sub-goals. *Coq* presents sequents such as

$$t_1 : T_1, \ldots, t_n : T_n \vdash t_0 : T_0$$

as

```
t1 : T1
   .
   .
   .
tn : Tn
============================
   t0 : T0
```

Say 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 sub-goal


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

Initially, there is only a single (sub)goal to be proved. Where we have more than one sub-goal remaining to be proved (i.e. more than one branch of the proof tree which is not closed by an axiom) we may have *Coq* show us either all the remaining sub-goals or only one at a time.

We wish to move the universally quantified variable `i` into the current context with a name new to the context (since the current context is empty, the name will remain as `i`). We do this by matching the conclusion of the universal quantifier introduction rule:[2]

$$\frac{\Gamma, y : T \vdash [y/x]G}{\Gamma \vdash \forall x : T.G}\ \forall\text{-I}$$

[2]With a side-condition that y is not free in Γ, T or G.

to the sequent above, so that Γ matches the empty sequent, x and y match `i`, T matches `nat` and G matches `(lt i (S i))`. *Coq* then prints

```
1 sub-goal

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

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

```
2 sub-goals

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

sub-goal 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. Note that this is simply an interface matter; we cannot assume that the hypotheses of the second sub-goal are identical to the fully printed first sub-goal. We may have *Coq* show us the full sequent for sub-goal 2:

```
sub-goal 2 is:

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

5.2 Formalisation of Proof Terms in *Coq*

The central issue in formalising sequent-style calculi with proof terms is the handling of variable bindings and references. There are two different forms of variable occurrence in proof terms: bound and free variables. In a sequent, we would expect all variables to be bound, i.e. there should be no references to objects outside the sequent, but when dealing simply with proof terms (as we do for the theorem $\psi\theta$ in table 2.3), we may have variables which reference formulae in an unspecified context rather than occurrences of binding constructors such as λ and *app*. Specifying a context would clutter the proof unnecessarily, provided that the theorem being proved is true for all possible contexts.

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 for formal treatments of λ-calculi in recent years has been nameless dummy variables, also called *de Bruijn indices* [dB72].[3] 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 [DFH95, GM96]. A similar but simpler approach is used in [MP93, MP97]

In the following three chapters we will look at three methods of formalising our example theory in *Coq*. The first method (§6) uses de Bruijn indices for the bound variables in a term and an encoding derived from the (object-level) context for free variables. There are some problems with this approach so §7 shows a formalisation using de Bruijn indices for both bound and free variables. Finally, in §8 we shall look at a method for using named variables developed by McKinna and Pollack (with suggestions by Coquand) used in [MP93, MP97]. A deeper discussion of the various approaches is contained in §9.

[3] In fact, *Coq* itself uses de Bruijn indices internally together with a persistent naming mechanism for display and interaction.

Chapter 6

An Initial Formalisation in *Coq*

This chapter presents a formalisation of the example theory using de Bruijn indices for bound variables in terms and an encoding of the current context for free variables. It was initially thought that this would avoid certain problems regarding context manipulation for operations such as weakening. It turned out that the problems did not exist, and that this encoding produced problems of its own. The next chapter will present a formalisation built by amending this one, which uses de Bruijn indices for both bound and free variables.

6.1 De Bruijn Indices

First we need to explain standard de Bruijn indices, before we enter into the variant used here. This standard de Bruijn approach is used in the next chapter.

We will use the well-known simply-typed λ-calculus [Bar84, Appendix A] for this exposition, since it is slightly simpler than the calculi **NJ**, **MJ** and **LJ**. In the following description of the simply-typed λ-calculus meta-variables P and Q range over Formulae (F), $\mathbf{V}$ is a set of variables and the Γ are contexts as before.

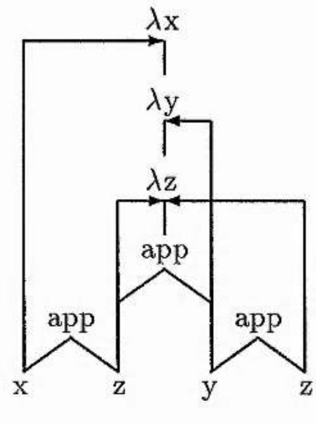
$$t ::= \mathbf{V} \mid \lambda\mathbf{V}.t \mid (t\ t) \qquad F ::= o \mid F \supset F \qquad \Gamma ::= [] \mid \Gamma, \mathbf{V} : F$$

$$\frac{\Gamma, x : P \vdash t : Q}{\Gamma \vdash \lambda x.t : (P \supset Q)} \supset \mathrm{I} \quad \frac{\Gamma \vdash t_1 : (P \supset Q) \quad \Gamma \vdash t_2 : P}{\Gamma \vdash (t_1\ t_2) : Q} \supset \mathrm{E} \quad \frac{}{\Gamma, x : P \rhd x : P}\ \mathrm{Axiom}$$

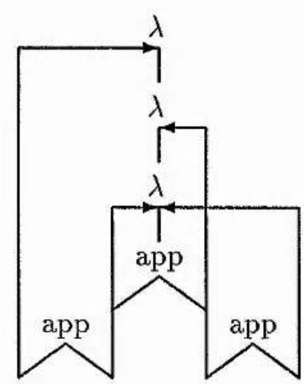
We will use the last stage of the proof tree in the derivation of the S combinator as an example later:

$$\frac{x : (P \supset (Q \supset R)) \vdash \lambda y.\lambda z.((x\ z)\ (y\ z)) : (P \supset Q) \supset (P \supset R)}{\vdash \lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z)) : (P \supset (Q \supset R)) \supset (P \supset Q) \supset (P \supset R)} \supset \mathrm{I}$$

If we take the term for the S combinator and view it as a tree structure, we have:



Now, the names of the bound variables do not matter in this instance, since with the graphical references, all that matters is that a particular leaf (variable occurrence) refers to a particular node (binding constructor). So, we might view the term S as:



This picture, while valid and useful for human interaction, would be difficult to formalise directly (higher order abstract syntax is a method of doing this with pointers). What we may do, therefore, is use the natural numbers to reference binding occurrences, since all we are interested in when making a reference to a bound variable is which λ is being referenced. There are two ways to do this: either the number refers to the number of binding operators[1] between the reference and the operator it references, or the number refers to the number of binding operators between the root of the syntax tree and the occurrence of the operator being referenced. The first of these is the more common method of representation, but both may be useful depending on the application. Using the leaf-to-binder counting, the partial

[1] In simply typed λ-calculus there is only the one binding operator (λ). In other systems, there may be more than one binder [NPS90].

deduction of the S combinator becomes:

$$\frac{[(P \supset (Q \supset R))] \vdash \lambda.\lambda.((2\ 0)\ (1\ 0)) : (P \supset Q) \supset (P \supset R)}{[] \vdash \lambda.\lambda.\lambda.((2\ 0)\ (1\ 0)) : (P \supset (Q \supset R)) \supset (P \supset Q) \supset (P \supset R)} \supset \mathrm{I}$$

where indices which count beyond the local binders reference formulae in the context, which is represented as a list. For the simply typed λ-calculus, the indexing flows seamlessly in rules such as $\supset$ I. This is not the case for all sequent-style calculi. Any logic involving splitting of the context, such as linear logics in particular, will require renaming of indexing in such rules. This is one of the weaknesses of de Bruijn indices as a general methodology.

For both methods, insertion or deletion of an abstraction in the term (e.g. η-expansion and β-reduction respectively) require changes to the indices. These changes involve *lifting* and *dropping*. As an example take the β-reduction below:

$$\lambda z.\lambda y.((\lambda x.\lambda w.(x\ (w\ y)))\ z)$$

reduces to:

$$\lambda z.\lambda y.\lambda w.(z\ (w\ y)).$$

Using leaf-to-binder de Bruijn indices this process becomes:

$$\begin{aligned} & \lambda.\lambda.((\lambda.\lambda.(1\ (0\ 2)))\ 1) \\ = \quad & \lambda.\lambda.\lambda.(2\ (0\ 1)) \end{aligned}$$

While performing these calculations, we must ensure that the referencing depths are kept updated, which is why the z which is originally a '1' becomes a '2' and the y which is originally a '2' becomes a '1', but w is represented by a '0' which stays constant. For a deeper examination of the role of lifting and dropping in using de Bruijn indices see §7.2.2 or [Hue94]. Lifting and dropping also come into play when defining the structural rules such as weakening (also called *thinning* from a literal translation of the term used in [Gen34]), where dropping is the process that must be carried out on a term when deleting an unused formula from the context.

6.2 Formulae, Contexts and Variables

We begin by defining an infinite set of formulae $\mathbf{F}$: which are either atomic $(f_0, f_1, \ldots)$ or implicative:

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

In propositional logics, such as the implicative fragments we are studying, the exact form of the atomic formulae does not matter. For the meta-theoretic proofs we are interested in, we will be working with universally quantified formulae in the theorems. The *S-combinator*, for example, is usually represented as

$$(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$$

which is parametric in A, B and C. In our syntax above the S-combinator would be

`(Impl (Impl (Impl A (Impl B C)) (Impl A B)) (Impl A C))`

Following this, the set of hypothesis lists (or *contexts*) for sequents can be defined as the set **Hyps**:

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

Since the word *context* is also used to refer to hypotheses in the current sequent in *Coq*, object-logic contexts will be referred to as hypothesis lists. The set **V** of nameless variables is defined as follows:

```
Inductive
    V:Set :=
        vfree : Hyps->V |
        vbnd : nat->V.
```

The `vbnd` constructor is used to denote bound variables within a derivation/deduction term and so uses natural numbers to refer to occurrences of binding operators, in the usual de Bruijn technique (see [dB72] for details). The `vfree` constructor is used to denote free variables within a derivation/deduction term, i.e. variables which reference a formula in the hypothesis list. The referencing mechanism consists of using the list before the addition of a new formula to reference that new formula. This use of a hypothesis list to represent free variables is more complex than use of the length of the hypothesis list or some other natural number encoding. It helps to specify the hypothesis list in which the derivation/deduction term has been created, and allows a distinction between free variables which were created with respect to different hypothesis lists of the same length. For example, during a proof involving structural rules, the hypothesis list will change in ways other than being extended by new formulae.

Equality is proved decidable for all these sets, together with decidability of some other relations, such as occurrence or non-occurrence of a free variable in a term (see §7.2 for more details in a different but related formalisation).

Thus, the derivation/deduction terms of the three systems are defined in the following way:

```
Mutual Inductive
    N:Set :=
        lam : N->N |
        an : A->N
with
    A:Set :=
        ap : A->N->A |
        var : V->A.
```

```
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.
```

This formalisation of **M** and **Ms** gives the following induction principle:[2]

```
(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))).
```

This is equivalent to the induction scheme:

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

6.3 Derivations and Deductions

All the components of a sequent have now been defined, and so the the propositional functions representing derivations/deductions may now be defined. Given the size of such definitions only derivations within **MJ** are shown here. `L_Deriv`, `N_Deduc` and `A_Deduc` are similarly defined.

[2]This is semi-automatically produced. 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 of the *Coq* system.

```
Mutual Inductive
    M_Deriv : Hyps -> M -> F -> Prop :=
        Choose : (h:Hyps)(i:Hyps)(P:F)(ms:Ms)(R:F)
                  (In_Hyps i P h)->
                   (Ms_Deriv h P ms R)->
                    (M_Deriv h (sc (vfree i) ms) R) |
        Abstract : (h:Hyps)(P:F)(m:M)(Q:F)
                    ~(Occurs_Free_In_M h m)->
                     (M_Deriv (Add_Hyp P h)
                              (bnd_to_free_M 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 6.1: Formal Definition of Derivations in **MJ**

Figure 6.1 shows the *Coq* definition for derivations in **MJ** and figure 6.2 (on page 43) shows the induction scheme semi-automatically produced for induction. The complexity of these induction principles shows why machine support is desirable for such work, and why a system such as *Coq*, with the ability to derive such principles (semi-)automatically, and the capability to prove such principles sound, is required.

The main point to be noted about `M_Deriv` is the *newness* or *freshness* condition:

`~(Occurs_Free_In_M h m)`

which occurs in the `Abstract` rule. 'h' is the free variable used to reference the formula (`P`) which is added to the hypothesis list in the premise. The non-occurrence of `h` as a free variable in the derivation term `m` is required to ensure that derivation terms do not contain variables outside the hypothesis list of the sequent. The same side-condition is required for similar reasons in [MP93, p.297, rule *LDA*] (see also §8.2.2).

```
(P:(h:Hyps)(m:M)(f:F)(M_Deriv h m f)->Prop)
 (P0:(h:Hyps)(f:F)(m:Ms)(f0:F)(Ms_Deriv h f m f0)->Prop)
  ((h,i:Hyps)(P1:F)(ms:Ms)(R:F)
    (i0:(In_Hyps i P1 h))
     (m:(Ms_Deriv h P1 ms R))
      (P0 h P1 ms R m)->
       (P h (sc (vfree i) ms) R (Choose h i P1 ms R i0 m)))->
   ((h:Hyps)(P1:F)(m:M)(Q:F)
     (n:~(Occurs_Free_In_M h m))
      (m0:(M_Deriv (Add_Hyp P1 h) (bnd_to_free_M h m) Q))
       (P (Add_Hyp P1 h) (bnd_to_free_M h m) Q m0)->
        (P h (lambda m) (Impl P1 Q) (Abstract h P1 m Q n m0)))->
    ((h:Hyps)(P1:F)(P0 h P1 mnil P1 (Meet h P1)))->
     ((h:Hyps)(m:M)(P1,Q:F)(ms:Ms)(R:F)(m0:(M_Deriv h m P1))
       (P h m P1 m0)->
        (m1:(Ms_Deriv h Q ms R))
         (P0 h Q ms R m1)->
          (P0 h (Impl P1 Q) (mcons m ms) R
                (Implies_S h m P1 Q ms R m0 m1)))->
      ((h:Hyps)(m:M)(f:F)(m0:(M_Deriv h m f))(P h m f m0))/\
      ((h:Hyps)(f:F)(ms:Ms)(f0:F)
         (m0:(Ms_Deriv h f ms f0))(P0 h f ms f0 m0)).
```

Figure 6.2: Induction scheme for derivations in **MJ**

6.3.1 Summary

The formal derivation term

`(lambda (sc (vbnd O) (mcons (sc (vfree MT) mnil) mnil)))`

in the context of a hypothesis list

`(Add_Hyp (f O) (Add_Hyp (f 1) MT))`

represents the informal term of **MJ**

$$\lambda x.(x\,;((y\,;[])::[]))$$

in the context of a hypothesis list

$$[x{:}\,f_0, y{:}\,f_1]$$

6.4 Conclusions

This hybrid approach of combining named free variables and nameless bound variables appeared at first to be a way of avoiding problems with some of the structural rules. On deeper examination, it became apparent that there were no real problems. This hybrid approach requires functions for both lifting/dropping and for the substitution of free variables for bound variables as for the McKinna and Pollack approach (see figure 8.1 on page 66 in §8). Since we must prove theorems about the interaction between each new function and each of these support functions, we are creating more work than necessary by using this approach. We describe a full formalisation, using only de Bruijn indices, of the example theory from §2 in the next chapter §7 and then some initial work using named variables in §8. This hybrid approach may have some uses, however, which we will examine in §10.

Chapter 7

A Formalisation in *Coq* Using de Bruijn Indices

This chapter presents a formalisation using de Bruijn indices for both the bound and free variables. Similar formalisations of typed λ-calculi appear in [Bar96, NN96].

7.1 Initial Definitions

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 propositional functions representing **MJ** derivations.

The set of formulae, **F**, is defined as before:

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

The set of contexts **Hyps** is defined using syntactic constructions to be an abbreviation for a list of **F**(ormulae), using the polymorphic list library provided with *Coq*. The length of a list, function `length` of type `(A:Set)(list A)->nat`, and some of its properties are made available with this library without the need to re-prove them for a new implementation. The syntax for **Hyps** is equivalent to the inductive definition:

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

`Len_Hyps` is defined as `(length Hyps)`.

The set **V** of nameless variables is defined as an abbreviation for the natural numbers. Note that the lack of differentiation between free and bound variables makes this much simpler than before.

Thus, the derivation terms of the three systems are defined in the following way:

```
Mutual Inductive
    N:Set :=
        lam : N->N |
        an : A->N
with
    A:Set :=
        ap : A->N->A |
        var : V->A.
```

```
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.
```

Note that these definitions (and therefore also any induction schemes derived) are identical to those in the previous chapter. The structure of these terms does not change despite the difference in the definition of the set **V**. The differences will manifest themselves in the definitions of functions involving variables, for instance substitution, and in the definitions of the propositional functions representing derivations in the calculus.

7.2 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 §5.1.4:

```
Inductive
    lt : nat->nat->Prop :=
        lt_O : (i:nat)(lt O (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 :=
        O O => false |
        O (S j) => true |
        (S i) O => false |
        (S i) (S j) => (ltb i j).
```

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

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

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

The decidability of `lt`,

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

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 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.

7.2.1 Setifb

We wish to be able to define functions over the sets of derivation/deduction 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 B:Set.
Recursive Definition
    Setifb : bool->B->B->B :=
        true x y => x
        false x y => y.
```

When we discharge the `Hypothesis B`, `Setifb` is defined as the polymorphic *if* over general sets, with type `(B:Set)bool->B->B->B`.

7.2.2 Lifting

Lifting is a necessary operation for using de Bruijn indices correctly. An implementation for standard untyped λ-calculus terms can be seen in [Hue94]. Here we will use the standard substitution function in **N** and **A** to illustrate `Lift_N` and `Lift_A`. Informally, we can mutually define substitution of an **A** for a variable in an **N** or an **A**:[1]

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

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

$$\lambda x.\lambda y.[var(x)/y]\lambda z.an(\lambda u.an(ap(ap(var(u),an(var(y))),an(var(z)))))$$

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

Unfolding the application of substitution once, we get:

$$\lambda x.\lambda y.\lambda z.[var(x)/y]an(\lambda u.an(ap(ap(var(u),an(var(y))),an(var(z)))))$$

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

As can be seen, no changes of name were required to move the substitution 'through' the lambda abstraction,[2] but for the de Bruijn indices, each variable in $[var(x)/y]$ 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

[1] We are assuming that the variable names are chosen so as to avoid problems with capturing free variables in a_0.

[2] This is due to the careful selection of distinct names for all the variables.

the following sequence of terms:

$$\lambda x.\lambda y.\lambda z.an([var(x)/y]\lambda u.an(ap(ap(var(u), an(var(y))), an(var(z)))))$$
$$\lambda.\lambda.\lambda.an([var(2)/1]\lambda.an(ap(ap(var(0), an(var(2))), an(var(1)))))$$
$$\lambda x.\lambda y.\lambda z.an(\lambda u.[var(x)/y]an(ap(ap(var(u), an(var(y))), an(var(z)))))$$
$$\lambda.\lambda.\lambda.an(\lambda.[var(3)/2]an(ap(ap(var(0), an(var(2))), an(var(1)))))$$
$$\vdots$$
$$\lambda x.\lambda y.\lambda z.an(\lambda u.an(ap(ap(var(u), an(var(x))), an(var(z)))))$$
$$\lambda.\lambda.\lambda.an(\lambda.an(ap(ap(var(0), an(var(3))), an(var(1)))))$$

The important point to notice here is that the de Bruijn reference variables in the substitution term $[var(x)/y]$ increase by one every time we unfold the application of substitution through an abstraction operator. 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:

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

7.2.3 The Usefulness of Boolean Functions

We shall now show 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 formal definition of lift for variables and 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 lifting operations for derivation/deduction terms generally, without repeating the parts of those proofs dealing with variable occurrences.

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 derivation/deduction terms:

$$\forall i : \mathtt{nat},\ t : \mathbf{T}.\ \downarrow_i\uparrow_i t = t,$$
$$\forall i : \mathtt{nat},\ t : \mathbf{T}. i \notin t \supset\uparrow_i\downarrow_i t = t,$$

where **T** is one of {**M**, **Ms**, **N**, **A**, **L**, **V**}. These theorems have only been proved in the formalisation where necessary: for **V**, **M** and **Ms**: see pages 154 and 155 in §B.

7.2.4 The Usefulness of Propositional Functions

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 §5.1.4 . 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 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 could not do this without proving the equivalence of `nateqb` and $=_{nat}$.

7.3 Translation Functions

Having defined the derivation/deduction terms and variable adjustment functions, we can now proceed to the functions translating derivation/deduction terms between the three systems, as shown in table 2.2. 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 2.2 to use de Bruijn indices, we get the following:

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

The second recursive call in the right hand side of the second definitional equation is not primitive recursive: $(0;\uparrow_0 ms)$ is not a sub-expression of $(x\,;m::ms)$. We may avoid part of the problem by using a mutual definition such as:

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

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 may also remove the first argument (a **V**), since only 0 is ever passed as that argument. The delayed *lifts* are performed where necessary by $\uparrow^n_m$ which is equivalent to $\uparrow_m$ repeated n times:

$$\begin{array}{rcl}
\bar{\rho}(x\,;[\,]) & =_{def} & vr(x) \\
\bar{\rho}(x\,;m::ms) & =_{def} & app(x,\bar{\rho}(m),\bar{\rho}'(ms,1)) \\
\bar{\rho}(\lambda x.m) & =_{def} & \lambda x.\bar{\rho}(m) \\
\bar{\rho}'([\,],n) & =_{def} & vr(0) \\
\bar{\rho}'(m::ms,n) & =_{def} & app(0,\uparrow^n_0\bar{\rho}(m),\bar{\rho}'(ms,n+1))
\end{array}$$

We now reach the following formal *Coq* definitions:[3]

```
Fixpoint
    rhobar [m:M] : L :=
        Cases m of
            (sc x mnil) => (vr x) |
            (sc (mcons m' ms)) => (app x (rhobar m') (rhobar' ms (S O))) |
            (lambda m') => (lm (rhobar m'))
        end
with
    rhobar' [ms:Ms] : nat->L :=
        [n:nat]Cases ms of
            mnil => (vr O) |
            (mcons m ms') =>
                (app O (lifts_L n O (rhobar m)) (rhobar' ms' (S n)))
        end.
```

where `lifts_L` is the formal version of $\uparrow^n_m$. This is the form of the definition in the formalisation. It is easier Since these definitions are primitive recursive, they are accepted by *Coq* without problem. We must now show that this formal `rhobar` is equivalent to the original version above. This requires us to prove the three lemmas:

```
RhoBar1 : (x:V)(rhobar (sc x mnil))=(vr x)
RhoBar2 :
        (ms:Ms)(x:V)(m:M)
         (rhobar (sc x (mcons m ms)))=
          (app x (rhobar m) (rhobar (sc O (lift_Ms O ms))))
RhoBar3 : (m:M)(rhobar (lambda m))=(lm (rhobar m))
```

[3] The definition is given using the `Cases` operator for ease of comparison with the informal definition. The actual formalisation was done using the `Case` operator and can be seen on page 186 in §B.

which are the formal *Coq* versions of the first set of definitional equations using de Bruijn indices shown above. As we shall see in §7.6, proof of RhoBar2 requires stronger induction methods than the standard ones.

Many lemmas have been proved regarding the interactions between the translation functions and the appropriate versions 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. All the lemmas proved may be found in §B near pages 154 and 155.

7.4 Derivations and Deductions

All the components of sequents have now been defined, as have a number of strategic reasoning aids. Propositional functions representing derivations/deductions may now be defined. Again, we will only show the definition for derivations within **MJ**.

```
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 §10 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.

7.4.1 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 2.3, 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 a generic sequent-style calculus. Exchange is not necessary for the proofs of theorems in table 2.3, but is essential for some of the proofs about permutation of derivations of **LJ**, shown in table 2.6.

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

$$\begin{array}{r} x \text{ not free in } t \\ x : P \in \Gamma \end{array} \qquad \frac{\Gamma \vdash t : R}{\Gamma \setminus x : P \vdash t : R}\ \textbf{Strengthening}$$

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

This is, of course, 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 derivation/deduction 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 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.

7.5 Permutation

Table 2.4 on page 14 shows the permutations in the usual informal syntax. Formalising these rules was more complex than might be thought. 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 7.1 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 7.2 shows the informal version of the typed reduction rule. Only the leaves and root of the 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 (a variable or a term of **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 Γ, l_3 must be lifted by 2 (`(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 7.1. 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 proved where it has been 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_app1 :
                (x,z:V)(l1,l2,l3:L)
                 ((Occurs_In_L 0 l2)\/(Occurs_In_L (S 0) l3))->
                  (Norm'_L l3)->
                   (L_Perm1 (app x l1 (app (S z) l2 l3))
                            (app z
                                 (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)
                 ((Occurs_In_L 0 l2)\/(Occurs_In_L (S 0) l3))->
                  (Norm'_L l3)->
                   (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 7.1: Formalised Permutations (see page 218 in §B)

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

$$\succ$$

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

Side-conditions: y'new and $(y\in l_2 \text{ or } y\in l_3)$

Figure 7.2: Proof Tree Fragment for Permutation App_App2

One final point to note about the formal permutations is highlighted in the side-conditions and the left hand side of `l_perm1_app_app1`:

```
l_perm1_app_app1 :
        (x,z:V)(l1,l2,l3:L)
         ((Occurs_In_L 0 l2)\/(Occurs_In_L (S 0) l3))->
          (Norm'_L l3)->
           (L_Perm1 (app x l1 (app (S z) l2 l3)) ...)
```

which formalises:[4]

$$\begin{array}{lcr}
 & app(x,l_1,y.app(z,l_2,w.l_3)) & y\neq z \\
\textit{(app_app1)} & \succ & (y\in l_2\vee y\in l_3) \\
 & \ldots &
\end{array}$$

The interesting point is that the inequality side-condition $(y\neq z)$ does not appear explicitly in the formalisation. The use of `(S z)` (instead of just `z`) forces this variable to differ from the bound variable `0` which is the translation of the binder "$y.$" in the informal version. We could use `z`, and include an explicit side-condition, but the version above allows slightly cleaner and shorter proofs, and is an obvious use of de Bruijn indexing.

[4] The extra side-condition of l_3 being fully normal with respect to y (`(Norm'_L l3)`) is an addition due to Schwichtenberg: see §7.7 for explanation.

7.6 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.

7.6.1 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 §5.1.3, *Coq* derives the induction principle:

```
(P:nat->Prop)
 (P O)->
  ((n:nat)(P n)->(P (S n)))->
   (n:nat)(P n).
```

7.6.1.1 Inductions on Simple Inductive Sets

Suppose we wish to prove the conjecture about natural numbers from §5.1.5:

`(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 §7.6.2), 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).

7.6.1.2 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 induction proofs involving such families.

7.6.1.3 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:

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

and apply our induction principle derived from the above family. This method is used in the formalisation when proving theorem `L_Admis_Weaken` (the admissibility of weakening in **LJ**). We define the function `l_admis_weaken` (see page 194 in §B):

```
Definition l_admis_weaken :
        (h:Hyps)(l:L)(P:F)(L_Deriv h l P)->Prop :=
         [h:Hyps][l:L][P:F][D:(L_Deriv h l P)]
          (j:nat)(Q:F)
           (lt j (S (Len_Hyps h)))->
            (L_Deriv (Weaken_Hyps j Q h) (lift_L j l) P).
```

and then proceed to prove:

```
Lemma L_admis_weaken :
        (h:Hyps)(l:L)(P:F)(D:(L_Deriv h l P))
        (l_admis_weaken h l P D).
```

by applying the induction principle derived from the definition of `L_Deriv`. The actual theorem `L_Admis_Weaken` follows simply from `L_admis_weaken` by unfolding the definition of `l_admis_weaken`.

7.6.1.4 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 §5.1.4) 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 derivation/deduction terms are designed to represent the derivations/deductions, 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 derivation/deduction term then inverting the judgement hypotheses'.

7.6.2 Strong Induction Principles

As mentioned in §7.3, proofs of theorems involving $\bar{\rho}$ require a different induction principle from the automatically generated 'standard' principle inferred from the definition of **M** and **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:

$$\forall P\colon (\mathrm{N} \to \mathit{Prop}).(\forall j\colon \mathrm{N}.(\forall i\colon \mathrm{N}.i < j \supset P(i)) \supset P(j)) \supset \forall n\colon \mathrm{N}.P(n).$$

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.[5]

The definition of $\bar{\rho}$ in [DP97a] 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 **M** and **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 derivation/deduction 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 derivation/deduction term then inversion of the judgement hypotheses' method described in §7.6.1.2 above.

[5] An extension should appear in the next full release of the *Coq* system.

7.7 Summary and Conclusions

In this chapter we have reviewed a formalisation of the theory from §2 in *Coq* using de Bruijn's nameless dummy variables. The formalisation completes the proof of weak normalisation for permutation reduction in the implicational fragment of propositional logic. Proofs of the same conjectures for full propositional logic are unlikely to require more complex methods, although such proofs would be long and tedious. Some automation of the procedures would therefore be useful. The *Coq* tactic `Auto`, when given appropriate `Hints` as to which lemmas to apply, produces some automation, particularly for simple linear arithmetic problems arising from de Bruijn index manipulation. However, there is a definite boundary, beyond which the Auto tactic is not designed to work, which is in the search for appropriate bindings in lemmas with variables which appear in the premises but which do not appear in the conclusion. `Auto` will not find such bindings, even if exact matches to the premises are found in the current context. Other than writing tactics designed to automate the few linear arithmetic problems not solved by `Auto` (such as those requiring complex transitive arguments), automation of the proof procedures needed for the work presented here would appear very difficult. The method of interactive proof exhibits a strong similarity to the automated methods of *rippling* [BS+93] and *relational rippling* [BL95]. §10 examines this relationship in some more detail.

Initial work on the permutability theorems Norm_Imperm_L and Norm_Red was performed using a formalisation of the original version of the permutations shown in table 2.6. Following the proof of strong normalisation for the system of reductions by Schwichtenberg in [Sch], weak normalisation was 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.

While the approach was successful, there are obvious problems remaining with the de Bruijn indices approach. The lifting and dropping of variable referencing, and the lack of names in itself, divorces the formalisation of the theory from the usual informal approach. Given that one of the aims of such formalisation is to increase our confidence in those informal results, the gap between the formal and informal syntaxes of the object systems is unfortunate. In the next chapter we examine a methodology proposed by McKinna and Pollack (with some suggestions by Coquand), laid out in some detail in [MP97], and its application to the example problem in *Coq*.

Chapter 8

A Formalisation in *Coq* Using Named Variables

8.1 Background of the Coquand-McKinna-Pollack Approach

McKinna and Pollack have been involved in formalising a substantial theory regarding *Pure Type Systems* (*PTS*) for a number of years. They have published papers showing the results [MP93, vBJMR94, Pol94], and recently submitted [MP97], which contains a more abstract view of their approach. Their work represents a very large development of a single abstract system (one which includes the Calculus of Constructions [CH85], a fragment of *CIC*, as a specific example). Their work is done in *LEGO* [LP92, Pol94], a proof assistant which can be instantiated to use a number of type theories, including *The Extended Calculus of Constructions* [Luo94], which is very similar to *CIC* and it is this instantiation that McKinna and Pollack use.

The Coquand-McKinna-Pollack (*CMP*) method represents a rejection of de Bruijn indices as counter-intuitive. When we are performing informal proofs about typed λ-calculi, we do not think of the λ terms as de Bruijn terms, we think of them as terms with named variables which have α-conversion built in. We recognise the equivalence of, for example, $\lambda x.x$ and $\lambda y.y$ with little effort. Definitions are all made involving named variables, and lifting and dropping are nowhere in our minds. Since the only approach allowing named variables known when their work started (see §9.6 on higher order abstract syntax) did not allow proofs by induction, McKinna and Pollack, with some suggestions by Coquand, developed their method for using named variables in a way independent of the particular calculus.

At the core of their approach is the distinction between *variables* and *parameters*: bound and free variables. The idea of distinguishing between these two sets is contained in [Gen34, Pra65] amongst others. Using this distinction, the *CMP* approach is described by McKinna [McK96] as "first order abstract syntax for terms with (restricted) higher order abstract syntax for judgements". The novel part of their approach involves the use of two different, but provably equivalent, formal judgements for each informal judgement in which we are interested. The equivalence of the two judgements allows us to derive stronger induction principles for the formal judgement we wish to use in proofs.

8.2 NJ Formalised with Named Abstract Syntax

8.2.1 First Order Abstract Syntax for Terms

Consider the informal definition of **NJ**:

$$\begin{aligned} \mathbf{N} &::= \lambda\mathbf{V}.\mathbf{N} \mid an(\mathbf{A}) \\ \mathbf{A} &::= ap(\mathbf{A},\mathbf{N}) \mid var(\mathbf{V}) \end{aligned}$$

NJ

$$\frac{\Gamma, x : P \rhd\!\rhd n : Q}{\Gamma \rhd\!\rhd \lambda x.n : (P \supset Q)} \supset \mathrm{I}$$

$$\frac{\Gamma \rhd a : P}{\Gamma \rhd\!\rhd an(a) : P} \text{ AN-Axiom}$$

$$\frac{\Gamma \rhd a : (P \supset Q) \quad \Gamma \rhd\!\rhd a : P}{\Gamma \rhd ap(a, n) : Q} \supset \mathrm{E}$$

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

and the role of the free and bound variables. As an argument to *var* we must be able to distinguish between variables which reference a λ binder (bound variables) and those which reference a formula in the local context (free variables). The properties we wish our variables to have are:

- Decidable equality.
- Availability of *new* variables when compared to a finite set of existing variables.

For the purpose of formalising **NJ**, **MJ** and **LJ**, we require only a single set of names, `Vars` with the following assumed properties:

```
Var : Set
New_Var : (list Var)->Var
New_New_Var : (l:(list Var))~(In Var (New_Var l) l).
```

i.e. that Var is a *CIC* set, and that there is an operator (`New_Var`) which, when given a list of `Vars` will return a new `Var` which is not in the given list (`New_New_Var`). We assume that there is a boolean equality function for `Var`, which is equivalent to propositional equality, as shown for the natural numbers in §7.2. These assumptions allow us to show decidability of propositional equality for `Var`. We also include the definition of `Setifb` as shown in §7.2.1. We then define a set `V` which distinguishes between bound and free variables thus:

```
Inductive V : Set :=
        BV : Var->V |
        FV : Var->V.
```

These two sets, `Var` and `V`, are used in the definition of formal deduction terms for **NJ**:

```
Mutual Inductive
    N:Set :=
        lam : Var->N->N |
        an : A->N
with
    A:Set :=
        ap : A->N->A |
        var : V->A.
```

This definition does not account for α-convertible terms in the same way that de Bruijn indices do. For example we wish to identify the two terms

`(lam x (an (var (BV x))))`

and

`(lam y (an (var (BV y))))`

(i.e. $\lambda x.x$ and $\lambda y.y$) as equal. We must define an equality predicate which captures this notion. We shall show the formal definition of such a predicate in the next section 8.2.2, but first we require a support function which substitutes a free variable (constructed with `FV`) for a bound variable (constructed with `BV`) in a term. Figure 8.1 shows the formal definition of such functions for sets **V**, **N** and **A**. As is often the case with `Fixpoint` definitions, we define a secondary function using `Fixpoint` and then a non-recursive primary version with the arguments in an order appropriate for human reading. (`BTF` stands for Bound To Free.)

```
Recursive Definition
    VBTF : Var->Var->V->V :=
        x y (BV z) => (Setifb V (Vareqb x z) (FV y) (BV z)) |
        x y (FV z) => (FV z).

Fixpoint
    NBTF1 [n:N]: Var->Var->N :=
        [b,f:Var]Cases n of
            (lam x n') =>
                (Setifb N (Vareqb x b)
                (lam x n')
                (lam x (NBTF1 n' b f))) |
            (an a) => (an (ABTF1 a b f))
        end with
    ABTF1 [a:A]: Var->Var->A :=
        [b,f:Var]Cases a of
            (ap a' n) => (ap (ABTF1 a' b f) (NBTF1 n b f)) |
            (var x) => (var (VBTF b f x))
        end.

Recursive Definition
    NBTF : Var->Var->N->N :=
        b f n => (NBTF1 n b f).

Recursive Definition
    ABTF : Var->Var->A->A :=
        b f a => (ABTF1 a b f).
```

Figure 8.1: Replacing Bound Variables with Free Variables

8.2.2 (Restricted) Higher Order Abstract Syntax for Judgements

We wish to define an equality predicate which we will use instead of the syntactic equality of *Coq* where necessary. There are a number of ways of formalising the predicate, but the *CMP* approach requires two forms: `Neq` and `Neq'`, as shown in §A.2 on pages 111 and 112 respectively. These definitions are almost identical. The difference is in the treatment of the `lam` constructor (as might be expected).

```
Mutual Inductive
    Neq : N->N->Prop :=
        lameq :
                (x,y,f:Var)(n1,n2:N)
                 ~(Free_In_N f n1)->
                  ~(Free_In_N f n2)->
                   (Neq (NBTF x f n1) (NBTF y f n2))->
                    (Neq (lam x n1) (lam y n2)) |
                .
                .
                .

Mutual Inductive
    Neq' : N->N->Prop :=
        lameq' :
                (x,y:Var)(n1,n2:N)
                 ((f:Var)~(Free_In_N f n1)->
                  ~(Free_In_N f n2)->
                   (Neq' (NBTF x f n1) (NBTF y f n2)))->
                    (Neq' (lam x n1) (lam y n2)) |
                .
                .
                .
```

The method of showing α-conversion is fairly straightforward: every time a binding constructor (`lam` being the only one for **N** and **A**) is met while recursing through the terms, the variables being bound are replaced in both terms by a single common free variable which did not previously occur in the terms. When we reach variable occurrences (with the `Var` constructor) we expect them to be the same free variable (i.e. the same `Var` with constructor `FV`). This only works with terms which have no hanging bound variable occurrences (bound variables which appear as `(Var (BV x))` for which no binder `lam x` can be found further up the parse tree of the term). The two variants of this method require (for `Neq`) that the property holds for all (new) free variables when we recurse down through `lam`, and (for `Neq'`) that there exists at least one (new) free variable for which the property holds.

When we come to use the α-conversion equality relation, such as proving that `Neq` is transitive, we would like to have the induction hypotheses from the scheme generated by `Neq'`. When we wish to recurse through a `lam` occurrence, however, we would like to apply `lameq`. The heart of the *CMP* approach is that for each judgement we wish to formalise (including those formalising derivations/deductions) we define variants such as those shown above. A particular method (detailed in [MP97]) allows one to prove the equivalence of any two such specific judgements (though each proof must be performed separately, as there does not appear to be a general higher order statement of the property that can be usefully proved and then applied). Once the bi-implication showing equivalence of the two judgement forms has been proved, a fairly simple proof can be done for the required induction scheme (see also page 112 in §A.2:

```
Lemma N_A_eq_ind' :
        (P:(n,n0:N)(Neq n n0)->Prop)
        (P0:(a,a0:A)(Aeq a a0)->Prop)
         ((x,y:Var)(n1,n2:N)
           (n:(f:Var)~(Free_In_N f n1)->~(Free_In_N f n2)->
                  (Neq (NBTF x f n1) (NBTF y f n2)))
            ((f:Var)
              (n0:~(Free_In_N f n1))
               (n3:~(Free_In_N f n2))
                (P (NBTF x f n1) (NBTF y f n2) (n f n0 n3)))->
             (P (lam x n1) (lam y n2) (lameq x y n1 n2 n)))->
          ((a1,a2:A)(a:(Aeq a1 a2))
              (P0 a1 a2 a)->(P (an a1) (an a2) (aneq a1 a2 a)))->
            ((a1:A)(n1:N)(a2:A)(n2:N)
                   (a:(Aeq a1 a2))
                    (P0 a1 a2 a)->
                     (n:(Neq n1 n2))
                        (P n1 n2 n)->
                         (P0 (ap a1 n1) (ap a2 n2)
                               (apeq a1 n1 a2 n2 a n)))->
              ((x:Var)(P0 (var (FV x)) (var (FV x)) (vareq x)))->
              ((n,n0:N)(n1:(Neq n n0))(P n n0 n1))/\
               ((a,a0:A)(a1:(Aeq a a0))(P0 a a0 a1)).
```

8.2.2.1 The *CMP* Approach for General Judgements and Predicates

In performing formal meta-theoretic proofs, we deal with formalisations of judgements and of predicates. Both of these are formalised as predicates in *Coq* (and *LEGO*). The *CMP* approach is that we use the same procedure for all the predicates in *Coq*. The method shown above for formalising equality of deduction terms is equally applicable to the formalisation of derivations in **NJ**.

The method above, of defining a universal variant (following the form of `Neq`, see 67) and an existential variant (following the form of `Neq'`, see 67) of the abstract predicate or judgement we are formalising, allows us to ignore bound variables almost entirely, by replacing them with (new) free variables when we pass beneath binders. Other methods of formalisation involve inductively defining predicates which use a local context to account for bound variable names. The experience of McKinna and Pollack [vBJMR94, MP93, MP97] is that the induction schemes derived from such definitions are often unsuitable for proving the conjectures being made. The induction schemes derived as described briefly above are more suitable to the formal development, and the homogeneity of the approach leads to induction hypotheses being of the appropriate (i.e. usable) form even when dealing with more than one predicate in a proof.

8.2.3 Complexity of the *CMP* Approach

The *CMP* approach requires a large amount of initial work in performing formalisations. Some can be carried across between developments, but not a great deal. As well as the `BTF` functions shown above, functions dealing with renaming free variables to other free variables (in single and parallel cases) are required in order to prove the necessary equivalences between universal and existential variants of complex typing judgements. Length (aka *height*) induction is also required for these proofs. Once the initial development has been carried out, there is still an overhead in extending a formalisation in that lemmas showing the relationship between new functions and each of the variable handling functions are required.

8.3 Scope of the Formalisation

The formalisation of the theory from §2 using this method in *Coq* was limited by the time available. The formalisation covers only the systems **MJ** and **NJ**, and theorems required to prove the bijection between them (including $\psi\theta(')$, M_Admis_$\psi(')$ and N_Admis_$\theta(')$). The primary definitions and lemmas are shown in §A.2.

Chapter 9

Related Work: Tools and Techniques

9.1 Introduction

This chapter presents an overview of the various approaches and tools used in the area of formal meta-theory. §9.2 starts us off with nameless dummy variables, also known as *de Bruijn indices*, as used in §§6 and 7, reviewing some of the many formalisations which have used that approach. We then describe the work of McKinna and Pollack, using the approach described in §8, followed by a discussion of the main ideas of *higher order abstract syntax* in §9.6. Finally we examine the attempts to combine higher order abstract syntax with induction and recursion in §9.7.

9.2 Formalisations Using de Bruijn Indices

9.2.1 Strong Normalization of System F in *LEGO*

[Alt93] presents a formalization of strong normalization for System F using the *LEGO* proof assistant [LP92]. The terms of System F are defined by Altenkirch in the standard de Bruijn manner. The types of System F are also defined using de Bruijn indices, but here a *LEGO* dependent type is used which also encodes the number of free variables in a term (see [Alt93] for an explanation as to why this is useful for types but unnecessary for terms).

Altenkirch's conclusions about the viability of *Computer Aided Formal Reasoning* is very up-beat:

> However, the fact that formalizing the proof after understanding it was not too much of an additional effort seems to justify the belief that *Computer Aided Formal Reasoning* may serve as a useful tool in mathematical research in future.

However, he does admit that:

> However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening.

The ease with which Altenkirch formalised this complex result reflects the usability of the system (*LEGO*), and the method (de Bruijn indices), for this particular kind of theory, and also Altenkirch's proficiency with the system, method and theory. As with many works of formal meta-theory, Altenkirch's proofs are simplified by the fact that he was working with only a single calculus. His approach is close to the work done by Coquand in *ALF* [Coq93], which also uses a semantic argument to prove strong normalization (this time of simply typed λ-calculus) where the terms are encoded using de Bruijn indices.

9.2.2 Verification of Algorithm $\mathcal{W}$: The Monomorphic Case

Algorithm $\mathcal{W}$ is the original type inference algorithm presented by Milner in [Mil78], which forms the basis of the ML type system, and, by extension, the type systems of many of the strongly typed functional languages currently available. Nazareth and Nipkow in [NN96] claim the first formal proof of soundness and completeness of algorithm $\mathcal{W}$ with respect to the typing rules. They deal only with the monomorphic case (not including the `let` construct), but state that they are unaware of any other formalisations involving algorithm $\mathcal{W}$. [NN96] presents a proof in *Isabelle/HOL* (a re-implementation of the *HOL* proof assistant using *Isabelle* as a framework). The formalisation uses standard de Bruijn indexing techniques for representing the terms for which algorithm $\mathcal{W}$ computes the types. This formalisation has two effects: firstly, the informal proofs of soundness and completeness of algorithm $\mathcal{W}$, which follow similar lines, gain credibility; secondly, the importance of the *new variable* problem as a non-trivial aspect of the proof is raised, together with a weakening of one of the conditions on a subsidiary part of the algorithm.

Despite their success with the proof in the monomorphic case, Nazareth and Nipkow believe that extension to "an object language with a `let`-construct and polymorphic types" is "likely to be a substantial piece of work".

9.2.3 Church-Rosser Proofs in *Isabelle/HOL*

There have been many formalisations of the *Church-Rosser* theorem for untyped λ-calculus with β-reduction, e.g. [Hue94, Sha94]. In [Nip96], Nipkow claims the first formalisation of Church-Rosser for β-η-reduction. Again Nipkow uses the standard de Bruijn indexing technique in *Isabelle/HOL* in order to formalise various aspects of λ-calculus. The work concentrates on abstract notions of the various properties of binary relations, using these to show the appropriate properties of the various calculi (λ-calculus with β-, η- and β-η-reduction). There is also a high level of automation present. Nipkow's conclusions are:

> It should be obvious from the above comparisons that the field [formal meta-theory] as a whole is making progress: formalizations have become more natural and shorter, and the degree of automation is increasing. We are also beginning to reuse other people's work (as in the case of Rasmussen's proofs). Yet each system still has painful shortcomings, for example arithmetic in the case of *Isabelle*. More work on the integration of decision procedures is urgently needed.

9.2.4 *Coq* in *Coq*

[Bar96] presents a formalisation of the *Calculus of Constructions* (*CoC*) [CH85], a fragment of *CIC*. The formalisation, extensively studied in [Bar96], covers strong normalisation and decidability of type inference for *CoC*. A *Caml Light* program is extracted which performs type inference or type checking for *CoC*. As a test of the program, the term derived from a formal proof of Newman's Lemma in *Coq* is re-verified by the program, with reasonable performance. The eventual aim of such work is to formally extract a kernel (type inference engine and type checker) for *CIC*, which may form the basis of a new version of *Coq*, a bootstrapping method similar to that used for *ACL2*, the latest of the Boyer-Moore family of provers [BM79, BM88].

Since *Coq* uses de Bruijn indices internally, it is unsurprising that Barras also uses them to produce a kernel for a fragment of its underlying calculus. An approach such as the *CMP* method, using an abstract type of variables, would not allow for the direct extraction of a program. However, by specifying a set of variables which have the appropriate properties a new kernel using names might be extracted.

9.3 A Formal Theory of Pure Type Systems

The methodology of the *CMP* method is described in §8. Here, we review the work done by McKinna and Pollack using that method. McKinna and Pollack began by formalising informal proofs by van Benthem Jutting and others (presented in [vBJ93] and elsewhere previously), and have since extended the formalisation to cover new ground, including a formal development of the theory of untyped λ-calculus with β-reduction. Their work is done using *LEGO* in its instantiation of the Extended Calculus of Constructions [Luo94, LP92]. This calculus is similar to *CIC*, the underlying calculus of *Coq*, although the top-level syntaxes of the two systems are rather different. Several versions of the basic *PTS* rules are presented and various equivalencies are proved. This does not require new machinery, since the term and type languages are not extended, only the rules for deriving judgements in the *PTS*. The complete development is an impressive body of formal proof, although as with all such developments the only way to understand what is being done is to run portions of the proof scripts line by line through *LEGO*. Even expert users of systems such as *LEGO*, *Isabelle* and *Coq* cannot run proofs in their heads from the statement of a conjecture and the proof commands in a script.

9.4 Five Axioms of α-Conversion

Gordon and Melham in [GM96] present a set of axioms for *HOL* which encode α-conversion for object languages with binding. The approach shows abstract similarities to the *CMP* method, differing mostly owing to the very different styles of the underlying systems *HOL* and *LEGO*. Similarities with the work on restricted higher order abstract syntax (see §§9.6 and 9.7) in [DFH95] are also evident. The primary distinction of their method is the encoding of an initial set of untyped lambda terms, which may then be differentiated by predicates to form sets of terms for different languages. The initial presentation in [GM96] includes only the definition of standard untyped λ-calculus terms, but the extension to other systems of syntax (such as the terms of **LJ** as presented in §2) would seem simple.

9.5 *HOL*, *ALF*, *Coq* and *LEGO*

In the previous sections we have briefly reviewed formalisations of proofs of properties of typed and untyped λ-calculi in various systems: *HOL*, *ALF*, *Coq* and *LEGO*. Since the main work presented in this thesis has been performed in *Coq*, it has been presented in more detail than the other systems. Nevertheless, it seems appropriate to set out some of the strengths of each system.

ALF seems one of the weakest systems available. It was never, however, a properly released and supported system, and has now been superseded by the still-experimental *HALF*. No documentation is available for *HALF*, although work done with it has been published in [CN96]. *HALF*, like *ALF*, is based on Martin-Löf's monomorphic type theory. One of the aims of the new system is to improve interaction and automation, areas where *ALF* was quite weak. Until the developers are satisfied enough with *HALF* to produce a full release, it is probably inadvisable to undertake large formalisations using *HALF*.

HOL, in its two incarnations as a stand-alone system [GM93] and an *Isabelle* object logic [Pau95b], implements a version of classical higher-order logic. Both versions are well implemented, and fairly mature, systems. They are somewhat divergent in their higher-level capabilities, particularly in the complex tactics available, though not in the underlying calculus.

Coq and *LEGO* are based on similar underlying calculi, and their capabilities are therefore also similar. The group working on *Coq* in the last few years has been larger, and the system developed more, although this leads to the corresponding problem of keeping up-to-date with new system releases. *LEGO* has developed less, and the core system has remained stable, allowing more time to be spent on new proofs and less on maintaining old ones. *Coq* is probably more accessible to the first-time user, however, with its extensible grammar syntax and more developed interface.

9.6 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 α-conversion and β-reduction 'for free' from the framework notions of conversion and reduction. However, the method of defining a set of terms which uses the framework variables as its variables is inadmissible in current frameworks with inductive definitions, such as *Coq* [PPM89]. The problem is in the definition of binding operators, such as λ, as might be expected. If we are defining a type *term* in a framework which allows *HOAS*, then the type of the λ 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 simply typed 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 λ 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. Since induction is such a central tool for meta-theory of the systems we might wish to investigate, *HOAS* would not currently appear to be a reasonable candidate for such work.

9.7 Higher Order Abstract Syntax with Induction

There have been several recent investigations into how a system of *HOAS* might be implemented within a framework allowing induction on the same terms. We will look at three approaches: a restricted version of *HOAS* developed in *Coq* in [DFH95], work on implementing primitive recursion within *HOAS* as a first step towards induction from [DPS96] and lastly a new framework proposal including *HOAS* and natural number induction in [MM97].

9.7.1 Restricted Higher Order Abstract Syntax with Induction in *Coq*

The main presentation of this work is [DFH95]. Owing to the restrictions presented in the previous sections from [PPM89], *HOAS* is not usable in *Coq*. What is possible is to assume an abstract set of variables $\mathbf{V}$, and then define our λ abstractor as having type:

$$(\mathbf{V} \to \mathit{term}) \to \mathit{term}.$$

As with the *CMP* approach, we must define our own equality predicate on terms. While we gain α-conversion from the framework (*Coq*) we do not gain β-reduction for free. There are also *exotic* terms included in the definitions of such a set: i.e. terms which satisfy the definition but which are not within the intended scope. The solution to this problem is two-fold. All definitions are made with respect to a notion of equivalence classes of terms, together with a validity requirement which excludes the exotic terms. This definition allows standard inductive arguments to be applied, although we may no longer define functions on our terms, and instead must use functional relations, which moves us further from the informal theories we may wish to formalise. In general, this restricted form of *HOAS* is too complex and too far from the informal theories to be a good solution.

9.7.2 *HOAS* with Primitive Recursion

[DPS96] is a large report detailing

> ...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.

The system presented in that report uses a modal λ-calculus to encode a system of *primitive recursive functionals*, in a manner inspired by linear logic. As of publication of the report, only a simply typed version of their theory had been developed and no implementation work had been done. This represents a significant step forward, and is the basis for ongoing work. It is unknown how long development will take and swift availability of a combined system is unlikely.

9.7.3 First Order Logic with Definitions and Natural Number Induction

[MM97] contains an overview of a proposal (laid out in full in [McD97]) for a system which, again, might allow *HOAS* to be combined with a system allowing induction. Here, the approach is somewhat different from that of Pfenning *et al.* McDowell and Miller start with a calculus of partial inductive definitions and add the natural numbers to produce $\mathbf{FO}\lambda^{\Delta\mathbf{N}}$. By implementing the natural numbers as part of the framework, together with the elimination principle allowing induction over the naturals, some forms of induction for other types may be derived via measure functions.

Chapter 10

Conclusions and Further Work

10.1 Frameworks vs. Proof Assistants

Initial work, as shown in §3 and §4, was carried out in logical frameworks. While it was possible to perform appropriate formalisations in these systems, it was necessary to encode induction principles as rules of the system. Addition of induction principles to a logic in order to improve its power is a traditional and valid method. However, the complexity of the inductions we required undermined our confidence that the principles we were adding to the system were correct. Since there are systems, (such as *Coq* and *LEGO*) which allow proof of such principles as part of their higher order logic, it would seem obvious that such systems are more suited to the formalisation of meta-theory. *Isabelle* and *SEQUEL* would be useful frameworks in which to encode a new system specifically designed for general meta-theoretic investigations. However, the theoretical basis of such systems (requiring as it does both induction and some form of higher order abstract syntax) is still an area of active research. An attempt to produce such a system would almost certainly take longer than was available for this project and it is doubtful that any progress would have been made with the motivating problem of formalising the permutation theorem.

10.2 Expansion of the Formalisation of the Permutation Theorem

As stated in §7.7, the informal proofs of the theorems in §2 have been extended to full propositional logic [DP97b]. Extension of either of the formalisations in §7 or §8 to full propositional logic would probably not require methods any more complex than those already used. The only substantive change to the theory of the implicational fragment in §2 is that the terms of type **Ms** are no longer simply lists of terms of type **M**. Thus, certain proofs which follow by simple list induction will require full proof by mutual structural or size induction. Since the list induction proofs are merely employed because they are available and shorter, rather than because of any doubt as to the viability of the full method, this should cause no problems.

10.2.1 New Tactics for *Coq*

Since extension to full propositional logic would involve some long and tedious proofs, it would seem sensible to consider programming subject-specific tactics for such a purpose. Identifying tactics which would be of general enough application to justify the work required to write them (writing tactics in *Coq* is a fairly time-consuming process) is difficult. Some simple syntactic abbreviations are obvious, and some have been programmed into the formalisation already. For instance, a common operation is to use the decidability of equality on variables (for both the *CMP* method and de Bruijn indices): if we have two variables `x` and `y` in our environment, we wish to perform a case split on `x=y\/~(x=y)`. When reasoning about a substitution, for instance, such case splits are often necessary. To perform this case analysis without any special-purpose tactics, the following commands suffice:

```
Cut x=y\/~x=y.
Intros c; Case c; Clear c; Intro.
```

provided `c` is not the name of a hypothesis in the current context. This process leaves us with three sequents to prove where we had one before. If we have added the decidability of equality on variables to the *Coq* `Hints` list, we may have the *cut goal* `x=y\/~x=y` automatically proved by `Auto` using the command:

```
Cut x=y\/~x=y; Auto; Intros c; Case c; Clear c; Intro.
```

We can then use the extensible grammar capabilities of *Coq* to define `Vcomp x y` to be equivalent to the above sequence, and the pretty printer to ensure that the same text is returned as part of a proof script. If there is already a hypothesis with name `c`, however, we will be reduced to using the full command with a different name. Using the Caml level of programming tactics, we could extend the `Vcomp` command to use a new name for the intermediate hypothesis `c`.

This is all very simple, and there are a number of cases like it, both in terms of extensions to the command grammar of *Coq* and with simple tactics. More complex tactics which would be useful are more difficult to identify. Certainly one tedious area highlighted by the formalisations was the use of the `Fixpoint` recursive function definition method. The existing *simplifier*, which reduces terms to a normal form without unfolding recursive functions further than necessary (see [BB+96]), only takes account of functions defined using the `Recursive Definition` construct. Since *Recursive Definition* does not allow mutual recursive functions, of which there are quite a number in the permutability theory, we must use `Fixpoint` and interactively perform rewriting. An extension to the simplifier tactic to use definitions made via `Fixpoint` would greatly simplify the proofs in the formalisations shown.

To go further than this, there is a recognisable pattern in many of the proofs in this formalisation. That pattern, to someone well-versed in the technique, is obviously rippling [BS+93, BL95].

10.2.2 Rippling

Rippling is the most successful method in the proof planning approach pioneered by Bundy *et al.* [BvHHS91]. Currently, rippling is primarily concerned with equality and functional expressions, but an extension to general relations has been studied, although not integrated into the main proof planning tool, *Clam*.

While performing the proofs of the theorems leading up to weak normalisation of the permutation reduction relation, we come across many proofs where the obvious method corresponds extremely well to rippling. The interactive search process that preceded a proof being found seemed to correspond well to the search mechanism of proof planning (with rippling as the primary method). Without an implementation of proof planning that interfaces to *Coq*, or a formalisation in a system for which proof planning is available, this is difficult to check without a long and involved by-hand proof planning analysis of the formalisation.

Providing an interface for *Clam* to *Coq* and integrating the relational rippling (necessary for the proofs involving derivations/deductions) technique into *Clam* would provide a powerful tool for simplifying the proof process involved in this formalisation. Particularly when faced with the tedious details of multiple connectives and the many similar sub-proofs entailed, such a combination would be an invaluable tool.

10.2.3 The Permutability Theorem for First Order Logic

As well as extending the existing weak normalisation result for permutability of inferences in **LJ** to full propositional logic, following the informal proofs, there is also the case of extension to first order logic. This has not been done in the informal work to date. One of the main motivations of the formalisation was to explore the possibilities of a formal proof for the first order case. While extension to full first order logic is the eventual aim, the universal-implicative fragment would be a useful test case.

In order to represent first order theories in a manner suitable for meta-theoretic reasoning, we must consider the proof process and its resulting proofs. To re-iterate a statement from §1.3: "Implementations [in a logical framework] of logics such as first order intuitionistic logic, classical linear logic etc., are coded within the machine environment in a way that allows the user to perform complex derivations/deductions within the logic thus defined. The aim of such work is to prove complex object-level statements (or enumerate their proofs)." This is particularly the case when we examine first order logic. A useful implementation of first order logic has "objects" about which theorems are proved. The precise structure of these "objects" is not our concern when dealing with the meta-theory of first order logic. We require a definition of them made with broad brush strokes, enabling a particular implementation the freedom to specify the objects of interest without too many restrictions.

So, we wish to encode unsorted first order logic in a manner which allows us to reason about its properties without needing to know too much about the objects over which our quantifications range. We therefore specify a set of expressions in an abstract manner, allowing us to reason about them without specifying too closely what their meaning is. We have an infinite set of constants, each of which has a natural number associated with it which is its arity. Terms (e.g. witnessing terms proving existential statements) can be built up from these constants in functional expressions and used in our meta-theoretic reasoning, without any actual semantics attached to these terms save their arity.

10.2.4 Strong Normalisation of Permutation Reduction

As stated in §7.7, [Sch] includes a proof of strong normalisation for a weakened version of the permutation reduction relation shown in §2 (for which weak normalisation was shown in the formalisation studied in §7). The proof of *SN* for permutation reduction is a corollary of a result involving yet another calculus. Extension of the formalisation (either using de Bruijn indices or the *CMP* method) to cover Schwichtenberg's proof would be interesting, as would explorations into a direct proof of *SN* for the weakened permutation reduction relation using only **LJ** and **MJ**.

10.3 Other Logics, Other Problems

There is a large body of informal meta-theory waiting to be formalised. The scope for such formalisations is limited only by the willingness of people to expend the time and effort to learn the techniques and become familiar with the tools.

One obvious candidate for formalisation is the permutation of inferences in Linear Logic [Gir87, GP94]. Linear logic, with its plethora of connectives, provides a rigorous challenge to the logician working informally. With so many interconnections to consider, the possibilities of an omission are very high, demanding meticulous care in approaching such work. The more detail that is spelled out in the informal proofs, the closer such work is to the formal approach demonstrated in this thesis. There do not appear to have been many attempts at formalising complex arguments about linear logic, although there may be some in progress now. The amount of work required to lay the groundwork for such an undertaking both deters, and delays the exposition of, such work. In particular, the standard de Bruijn approach does not work well if applied in a naive manner to the meta-theory of linear logic. See §10.4.1 for an exposition of the problem and some suggestions for a solution.

10.4 De Bruijn Indices, the *CMP* Method and *HOAS*: Conclusions

10.4.1 De Bruijn Indices

> I don't like de Bruijn indices myself.
>
> — N.G. de Bruijn

The above quote appears at the start of [DFH95]. De Bruijn indices are not what we really want, which is a formal environment in which to do proofs in a way that allows our creativity free reign while ensuring correctness of our work. De Bruijn indices are a relatively easy way to ensure some correctness. They are easy to implement and understand. If we make an error in our initial formalisation of terms with de Bruijn indices it will be easily spotted and corrected. However, the question of whether our encoding of functions and relations (such as $\bar{\rho}$ or `M_Deriv`) using de Bruijn indices is correct is more difficult. The more complex our definitions become, and the further away our framework leads us[1] from our original, informal intuitions, the less the confidence gained from the formalisation transfers back to our original

[1] For example compare the original, informal, definition of $\bar{\rho}$ and the numerous transformed versions until we get the primitive recursive formal version.

work. In some cases this is not a problem. For instance, Barras' work on formalising *CoC* in *Coq* makes good use of de Bruijn indices: a program derived from a named syntax might be very much less efficient. The formalisation shown in §7 is sufficiently close to the informal version to be useful, but the differences still remain and are the cause of some dissatisfaction with the results.

The really positive aspect of de Bruijn indices is the fact that they are useful now. Within certain limits they are easy to use and while there is some expansion of the proof requirements to handle the arithmetic, much of that can already be automated (in *Coq* at least). The overheads of using de Bruijn indices are mostly linear. Every time a new function is introduced, the relationship with the de Bruijn indexing functions *lift* and *drop* must be derived, but little else is required. In particular, there is little start-up cost that has not already been done in a number of formalisations, particularly the one shown here. The final point in favour of de Bruijn indices is that α-convertible terms are equal terms within the framework used (here *Coq*). Any framework such as *Coq* or *LEGO* which includes reasonable support for equality reasoning and rewriting will be easier to use when dealing with de Bruijn indices rather than a user-defined α-convertibility relation for equality.

As has been mentioned a number of times, however, not all logics are easy to encode using de Bruijn indices. Any logic which includes structural changes to the context as part of a rule will violate the smooth transition from binder-reference to context reference. Take for instance the right-rule for tensor ($\otimes$), or any of a number of other multiplicative rules, in intuitionistic linear logic (*ILL*) [Gir87]:

$$\frac{\Gamma_1 \vdash t_a : A \quad \Gamma_2 \vdash t_b : B}{\Gamma_1, \Gamma_2 \vdash tsr(t_a, t_b) : A \otimes B} \otimes\text{-R}$$

The problems with a de Bruijn index formalisation are caused by the splitting of the context between the conclusion and the premises. Unlike those of **NJ**, **MJ** and **LJ**, the rules of ILL contain more complex changes to the context than simple growth by addition of new formulae. t_a and t_b in the premises are not equal to t_a and t_b in the conclusion in terms of variable referencing. The hybrid approach described in §6, which uses de Bruijn indices for bound variables but a different encoding for free variables, might well prove an adequate solution, without the overheads involved in using the *CMP* method. Another possible solution, retaining use of de Bruijn indices, would be to amend the contexts in some way to block the use of the same formula in both branches of the proof tree. More exploration of these methods would be needed to show if they retained enough simplicity to justify not moving to the *CMP* method or another form of named variable syntax.

10.4.2 The *CMP* Method

The approach of McKinna and Pollack is obviously successful, as shown by the impressive body of work they have accumulated in their "hobby" time about *PTS* and λ-calculus. When working with a large body of proofs involving a single term structure, the initial overheads of α-conversion, variable replacement etc. pale in comparison to the overall proof effort. The overhead involved in showing the relationship of each new definition to the variable replacement functions is approximately equivalent to the overhead involved in using de Bruijn indices, where the relationship with *lift* and *drop* must be shown for new functions. New inductive relations also require the equivalence of the existentially and universally quantified variants as described in §8. So, in total, the *CMP* method involves more work than using de Bruijn indices. Why, then, would it be worth using? Well, once the initial formalisation has been done, further work takes approximately equivalent effort to de Bruijn indices, but the use of named variables keeps the formalisation closer to the informal definitions. In particular, function definitions remain closer to the informal definition. Consider the informal, de Bruijn index and *CMP* formalisations of *sub* from table 2.2:

$sub : \mathbf{V} \times \mathbf{M} \times \mathbf{V} \times \mathbf{M} \to \mathbf{M}$		
$sub(x, m, y, (y\,;ms)) =_{def}$	$(x\,; m :: subs(x, m, y, ms))$	
$sub(x, m, y, (z\,;ms)) =_{def}$	$(z\,; subs(x, m, y, ms))$	$z \neq y$
$sub(x, m, y, \lambda z.m') =_{def}$	$\lambda z.sub(x, m, y, m')$	$z \neq y$

Coq formal de Bruijn index lemma representing lines 1 and 2:

```
Lemma MSVMV1 :
        (x:V)(m:M)(y,z:V)(ms:Ms)
          (MsubstVMV x m y (sc z ms)) =
           (Setifb M (nateqb y z)
                   (sc x (mcons m (MssubstVMV x m z ms)))
                   (sc (drop_V y z) (MssubstVMV x m y ms))).
```

Coq formal *CMP* approach lemma representing lines 1 and 2:

```
Lemma MSVMV1 :
        (x:V)(m:M)(y,z:Var)(ms:Ms)
          (MsubstVMV x m y (sc (BV z) ms)) =
           (Setifb M (Vareqb y z)
                   (sc x (mcons m (MssubstVMV x m z ms)))
                   (sc (BV z) (MssubstVMV x m y ms))).
```

Coq formal de Bruijn index lemma representing line 3:

```
Lemma MSVMV2 : (x:V)(m:M)(m':M)(y:V)
               (MsubstVMV x m y (lambda m')) =
                (lambda (MsubstVMV (lift_V O x) (lift_M O m) (S y) m')).
```

Coq formal *CMP* approach lemma representing line 3:

```
Lemma MSVMV2 : (x:V)(m:M)(m':M)(y,z:Var)
               (MsubstVMV x m y (lambda z m')) =
                (Setifb M (Vareqb y z)
                        (lambda z m')
                        (lambda z (MsubstVMV x m y m'))).
```

The exact `Fixpoint` definitions, of course, do not matter, as it is these equality lemmas in which we are interested. The lack of *lift* and *drop* in the *CMP* version makes it easier to compare the formal and informal versions. (The formalisations of *subs* exhibit few differences and are both similar to the informal definition.)

When choosing between de Bruijn indices and the *CMP* method for a formalisation, the judgement will always be tricky. The more different term structures involved, the more initial overhead the *CMP* method will contain, and the more work will have to be done using the α-conversion predicate instead of direct syntactic equality. The formalisation described in §7 did not contain all of the support functions and proofs that must be done for the method to be applied properly. There is such a plethora of functions and theorems to be proved when developing a formalisation using the *CMP* method that few researchers performing formalisations will be willing to proceed. To enhance the usability of this method tactics to automate the proof of the many lemmas required, and even to derive their form would be needed.

10.4.3 *HOAS*

Higher order abstract syntax appears to be an elegant solution to the problem of variable handling. Since most frameworks already have a method for handling variables, it seems an obvious requirement that we should not have to solve the same problem at both levels. However, the incompatibility between frameworks allowing higher order abstract syntax and the well-known restrictions on methods for defining inductive structures with strong elimination principles, currently rules out this approach. As shown in this thesis, induction plays too large a role to be left to an informal correctness argument: such a method removes too much of the gain from machine support to leave the formalisation effort worthwhile.

The work by Miller and McDowell [MM97], and Pfenning *et al.* [DPS96], though still in the early stages, holds out promise for a more satisfactory solution in the long term. In the short term, however, we appear to be left with de Bruijn indices and manually-defined named syntaxes such as the *CMP* approach, or a hybrid of both. For those developing such tools, the following capabilities seem to be required:

- named variables,
- inductive definitions,
- recursive definitions,
- automatic derivation of elimination/induction principles,
- the capability of proving new induction principles sound,
- list, set and multiset handling of contexts

Bibliography

[AGM92] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors. *Handbook of Logic in Computer Science Vol 2: Computational Structures.* Oxford, 1992.

[Alt93] Th. Altenkirch. A formalisation of the strong normalisation proof for System F in LEGO. In [BG93], 13–28.

[AGNvS94] Th. Altenkirch, V. Gaspes, B. Nordström, and B. von Sydow. A User's Guide to *ALF*, 1994. Available from ftp.cs.chalmers.se.

[BN94] H. Barendregt and T. Nipkow, editors. *Types for proofs and programs: international workshop TYPES '93: selected papers.* Springer-Verlag LNCS 806, 1994.

[Bar84] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North Holland, 1984.

[Bar96] B. Barras. *Coq* en *Coq*. Technical Report 3026, INRIA, 1996.

[BB+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.

[BC93] D. A. Basin and R. L. Constable. Metalogical Frameworks. In [HP93], 1–29.

[BC96] S. Berardi and M. Coppo, editors. *Types for proofs and programs: international workshop TYPES '95: selected papers.* Springer-Verlag LNCS 1158, 1996.

[BG93] M. Bezem and J. F. Groote, editors. *Typed Lambda Calculus and Applications.* Springer-Verlag LNCS 664, 1993.

[BGM93] E. Börger, Y. Gurevich, and K. Meinke, editors. *Computer Science Logic '93.* Springer-Verlag LNCS 832, 1993.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[Buc85] B. Buchberger, editor. *EUROCAL '85 Vol. 1.* Springer-Verlag LNCS 203, 1985.

[BL95] A. Bundy and V. Lombart. Relational rippling: a general approach. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 175–181. IJCAI, 1995.

[BS+93] A. Bundy, A. Stevens, et al. Rippling: a heuristic for guiding inductive proofs. *Art. Int.*, 185–253, 1993.

[BvHHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with Proof Plans for Induction. *J. Automated Reasoning*, 303–324, 1991.

[CN96] J. Cederquist and S. Negri. A Constructive Proof of the Heine-Borel Covering Theorem for Formal Reals. In [BC96].

[CA+86] R. L. Constable, S. F. Allen, et al. *Implementing Mathematics with the NuPrl Proof Development System.* Prentice-Hall, 1986.

[Coq93] C. Coquand. From Semantics to rules: A machine assisted analysis. In [BGM93].

[CH85] Th. Coquand and G. Huet. Constructions: A Higher Order Proof System for Mechanizing Mathematics. In [Buc85], 151–184.

[dB72] N. G. de Bruijn. λ-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. In [SH80], 579–606.

[DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-Order Abstract Syntax in *Coq.* In [PDC95], 124–138.

[DPS96] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive Recursion for Higher Order Abstract Syntax. Research Report CMU-CS-96-172, School of Computer Science, Carnegie Mellon University, 1996.

[DNS94] P. Dybjer, B. Nordström, and J. Smith, editors. *Types for proofs and programs: International Workshop TYPES '94: proceedings.* Springer-Verlag LNCS 996, 1994.

[Dyc92] R. Dyckhoff. Contraction-Free Sequent Calculi for Intuitionistic Logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.

[DP96] R. Dyckhoff and L. Pinto. A Permutation-Free Sequent Calculus for Intuitionistic Logic. Research Report CS/96/9, School of Mathematical and Computational Sciences, University of St Andrews, 1996.

[DP97a] R. Dyckhoff and L. Pinto. Permutability of proofs in intuitionistic sequent calculi, 1997. Submitted for publication, extended version available as [DP97b].

[DP97b] R. Dyckhoff and L. Pinto. Permutability of proofs in intuitionistic sequent calculi. Research Report CS/97/7, School of Mathematical and Computational Sciences, University of St Andrews, 1997.

[DP98] R. Dyckhoff and L. Pinto. Cut-Elimination and a Permutation-Free Sequent Calculus for Intuitionistic Logic. *Studia Logica (to appear)*, 1998.

[Fel89] A. Felty. A Logic Program for Transforming Sequent Proofs to Natural Deduction Proofs. In [SH89], 157–178.

[GP94] D. Galmiche and G. Perrier. On Proof Normalisation in Linear Logic. *Theoretical Computer Science*, 135(1):67–110, 1994.

[Gen33] G. Gentzen. On the Relation Between Intuitionistic and Classical Arithmetic. In [Sza69], 53–67.

[Gen34] G. Gentzen. Investigations into Logical Deduction. In [Sza69], 68–131. Translated from 1934 original in German.

[Gim94] E. Giminez. Codifying guarded definitions with recursive schemes. In [DNS94], 39–59.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GLT89] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. CUP, 1989.

[GM96] A. D. Gordon and T. Melham. Five Axioms of Alpha-Conversion. In [vWGH96], 173–190.

[GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL*. CUP, 1993.

[Her94] H. Herbelin. A λ-calculus Structure Isomorphic to Gentzen-style Sequent Calculus Structure. In [PT94], 61–75.

[Hue94] G. Huet. Residual Theory in λ-calculus: A Complete Gallina Development. *J. Functional Programming*, 3(4):371–394, 1994.

[HP91] G. Huet and G. Plotkin, editors. *Logical Frameworks.* CUP, 1991.

[HP93] G. Huet and G. Plotkin, editors. *Logical Environments.* CUP, 1993.

[Kal94] S. Kalvala. A Gentle Introduction to *Isabelle*, 1994. Available with system documentation.

[Kle52] S. C. Kleene. Permutability of inferences in Gentzen's calculi LK and LJ. *Mem. Amer. Math. Soc.*, 1–26, 1952.

[Klo92] J. W. Klop. Term Rewriting Systems. In [AGM92].

[Lei79] D. Leivant. Assumption Classes in Natural Deduction. *Zeitschrift für math. Logik*, 25:1–4, 1979.

[Luo94] Z. Luo. *Computation and Reasoning.* Clarendon Press, 1994.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, UK, 1992.

[MMMS90] M. Main, A. Melton, M. Mislove, and D. Schmidt, editors. *International Conference on the Mathematical Foundations of Programming Semantics.* Springer-Verlag LNCS 442, 1990.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[McD97] R. C. McDowell. *Proving Meta-Theorems in a Logical Framework.* PhD, Computer and Information Science Department, University of Pennsylvania, 1997. In preparation.

[MM97] R. 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.

[MP97] J. H. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalised. Submitted, 1997.

[MS96] M. A. McRobbie and J. K. Slaney, editors. *Automated deduction, Cade-13: 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30-August 3, 1996: proceedings.* Springer-Verlag LNAI 1104, 1996.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences,* 17:348–375, 1978.

[Min94] G. Mints. Cut-elimination and normal forms of sequent derivations. Technical Report CSLI-94-193, Stanford University, 1994.

[Min96] G. Mints. Normal forms of sequent derivations. In [Odi96], 469–492. Also part of [Min94].

[NN96] D. Nazareth and T. Nipkow. Formal Verification of Algorithm W: The Monomorphic Case. In [vWGH96], 331–345.

[Nip96] T. Nipkow. More Church-Rosser Proofs (in Isabelle/HOL). In [MS96].

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf type theory: an introduction.* Oxford University PressP, 1990.

[Odi96] P. Odifreddi, editor. *Kreiseliana.* A. K. Peters, Wellesley (Massachusetts), 1996.

[PT94] L. Pacholski and J. Tiuryn, editors. *Computer Science Logic '94.* Springer-Verlag LNCS 933, 1994.

[PM93] C. Paulin-Mohring. Inductive definitions in the system *Coq*: Rules and properties. In [BG93].

[Pau88] L. C. Paulson. The Foundation of a Generic Theorem Prover. *J. Automated Reasoning,* 5:363–396, 1988.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[Pau95a] L. C. Paulson, editor. *First* Isabelle *Users Workshop*, 1995. Contact Paulson, L. C. (lcp@cl.cam.ac.uk) for copies.

[Pau95b] L. C. Paulson. *Introduction to* Isabelle. Computer Laboratory, Cambridge University, 1995.

[Pfe91] F. Pfenning. Logic programming in the LF logical framework. In [HP91], 149–181.

[PPM89] F. Pfenning and C. Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In [MMMS90], 209–228.

[PDC95] G. Plotkin and M. Dezani-Ciancaglini, editors. *Typed Lambda Calculus and Applications*. Springer-Verlag LNCS 902, 1995.

[Pol94] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD, Department of Computer Science, University of Edinburgh, 1994.

[Pra65] D. Prawitz. *Natural Deduction.* Almquist & Wiksell, 1965.

[SH89] P. Schroeder-Heister, editor. *Extensions of Logic Programming.* Springer-Verlag LNAI 475, 1989.

[Sch] H. Schwichtenberg. Termination of permutative conversions in intuitionistic Gentzen calculi. Submitted for publication, Jan 97.

[SH80] J. P. Seldin and J. R. Hindley, editors. *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism.* Academic Press, 1980.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge Tracts in Theoretical Computer Science. Cambridge, 1994.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1969.

[Tar93] M. Tarver. A Language for Implementing Arbitrary Logics. In *Proceedings of the 13th International Joint Conference on Art. Int.*, 839–844, 1993.

[Tar97] M. Tarver. *Functional Programming and Automated Deduction in* **SEQUEL**. Wiley, 1997. (Forthcoming).

[TS96] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* CUP, 1996.

[vBJ93] L. S. van Benthem Jutting. Typing in Pure Type Systems. *Information and Computation*, 105(1):30–41, 1993.

[vBJMR94] L. S. van Benthem Jutting, J. McKinna, and Pollack R. Checking Algorithms for Pure Type Systems. In [BN94], 19–61.

[vWGH96] J. von Wright, J. Grundy, and J. Harrison, editors. *Theorem Proving in Higher Order Logics: 9th International Conference.* Springer-Verlag LNCS 1125, 1996.

Appendix A

Primary Definitions and Lemmas in *Coq*

A.1 De Bruijn Index Formalisation

The following are some of the main definitions and lemmas from the de Bruijn index formalisation examined in §7.

```
Section boolean_extension.

Hypothesis genset:Set.
Recursive Definition
    Setifb : bool->genset->genset->genset :=
        true x y => x |
        false x y => y.

End boolean_extension.

Recursive Definition
    nateqb : nat->nat->bool :=
        O O => true |
        (S i) O => false |
        O (S j) => false |
        (S i) (S j) => (nateqb i j).
```

```
Lemma nateqb_is_eq1 : (i,j:nat)i=j->(nateqb i j)=true.

Lemma nateqb_is_eq2 : (i,j:nat)(nateqb i j)=true->i=j.

Lemma nateqb_is_eq3 : (i,j:nat)(~i=j)->(nateqb i j)=false.

Lemma nateqb_is_eq4 : (i,j:nat)((nateqb i j)=false)->~i=j.

Recursive Definition
    max_nat : nat->nat->nat :=
        i j => (Setifb nat (ltb i j) j i).

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

Inductive
    In_Hyps : nat->F->Hyps->Prop :=
        inhyps_base : (P:F)(h:Hyps)
                        (In_Hyps O P (Add_Hyp P h)) |
        inhyps_rec : (n:nat)(P,Q:F)(h:Hyps)
                        (In_Hyps n P h)->
                         (In_Hyps (S n) P (Add_Hyp Q h)).

Definition V :Set := nat.

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 :=
        mnil : 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.

Fixpoint
   theta [m:M]:N :=
        <N>Case m of
            [x:V][ms:Ms](theta1' ms (var x))
            [m:M](lam (theta m))
        end with
    theta1' [ms:Ms]:A->N :=
        [a:A]<N>Case ms of
            (an a)
            [m:M][ms:Ms](theta1' ms (ap a (theta m)))
        end.

Recursive Definition
    theta' : A -> Ms -> N :=
        a ms => (theta1' ms a).

Fixpoint
    psi [n:N]:M :=
        <M>Case n of
            [n:N](lambda (psi n))
```

```
        [a:A](psi' a mnil)
    end with
  psi' [a:A]:Ms->M :=
    [ms:Ms]<M>Case a of
        [a':A][n:N](psi' a' (mcons (psi n) ms))
        [x:V](sc x ms)
    end.

Lemma thetapsi:
    (n:N)((theta(psi n)) = n).

Lemma thetapsi'theta':
    (a:A)(ms:Ms)((theta (psi' a ms)) = (theta' a ms)).

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)).

Lemma Lift_Lift_V_Bridge : (x:V)(i,j:nat)
    (lt i j)->
     (lift_V i (lift_V j x))=
      (lift_V (S j) (lift_V i x)).

Recursive Definition
  drop_V : nat->V->V :=
    j i => (Setifb V (ltb i j) i (pred i)).

Inductive
  Occurs_In_V : nat->V->Prop :=
```

```
    Occurs_in_v : (i,j:nat)i=j->
                    (Occurs_In_V i j).


Inductive
   Occurs_In_L : nat->L->Prop :=
      Occurs_in_vr :
             (i:nat)(x:V)
              (Occurs_In_V i x)->
               (Occurs_In_L i (vr x)) |
      Occurs_in_app1 :
             (i:nat)(x:V)(l1,l2:L)
              (Occurs_In_V i x)->
               (Occurs_In_L i (app x l1 l2)) |
      Occurs_in_app2 :
             (i:nat)(x:V)(l1,l2:L)
              (Occurs_In_L i l1)->
               (Occurs_In_L i (app x l1 l2)) |
      Occurs_in_app3 :
             (i:nat)(x:V)(l1,l2:L)
              (Occurs_In_L (S i) l2)->
               (Occurs_In_L i (app x l1 l2)) |
      Occurs_in_lm :
             (i:nat)(l:L)
              (Occurs_In_L (S i) l)->
               (Occurs_In_L i (lm l)).


Fixpoint
   MsubstVMV1 [m:M] : V->M->V->M :=
      [x:V][m':M][i:V]<M>Case m of
          [z:V][ms:Ms]
              (Setifb M (nateqb i z)
                      (sc x (mcons m' (MssubstVMV1 ms x m' z)))
                      (sc (drop_V i z) (MssubstVMV1 ms x m' i)))
          [m'':M]
              (lambda (MsubstVMV1 m'' (lift_V O x) (lift_M O m') (S i)))
      end with
```

```
    MssubstVMV1 [ms:Ms] : V->M->V->Ms :=
        [x:V][m':M][i:V]<Ms>Case ms of
            mnil
            [m'':M][ms':Ms](mcons (MsubstVMV1 m'' x m' i)
                                  (MssubstVMV1 ms' x m' i))
        end.

Recursive Definition
    MsubstVMV : V->M->V->M->M :=
        x m i m' => (MsubstVMV1 m' x m i).

Recursive Definition
    MssubstVMV : V->M->V->Ms->Ms :=
        x m i ms => (MssubstVMV1 ms x m i).

Recursive Definition
    phi : L -> N :=
        (vr x) => (an (var x)) |
        (app x l1 l2) =>
                (NsubstAV (ap (var x) (phi l1)) O (phi l2)) |
        (lm l) => (lam (phi l)).

Recursive Definition
    phibar : L->M :=
        (vr x) => (sc x mnil) |
        (app x l1 l2) =>
                (MsubstVMV x (phibar l1) O (phibar l2)) |
        (lm l) => (lambda (phibar l)).

Recursive Definition
    lifts_L : nat->nat->L->L :=
        i j (vr x) => (vr (lifts_V i j x)) |
        i j (app x l l0) =>
                (app (lifts_V i j x)
                (lifts_L i j l)
                (lifts_L i (S j) l0)) |
```

```
        i j (lm l) => (lm (lifts_L i (S j) l)).


Fixpoint
    rhobar [m:M] : L :=
        <L>Case m of
            [x:V][ms:Ms]
                <L>Case ms of
                    (vr x)
                    [m:M][ms:Ms](app x (rhobar m) (rhobar' ms (S O)))
                end
            [m:M](lm (rhobar m))
        end
with
    rhobar' [ms:Ms] : nat->L :=
        [i:nat]<L>Case ms of
            (vr O)
            [m:M][ms:Ms](app O (lifts_L i O (rhobar m)) (rhobar' ms (S i)))
        end.


Recursive Definition
    rhobari : nat->Ms->L :=
        i ms => (rhobar' ms i).


Lemma phibarrhobar :
        (m:M)(phibar (rhobar m))=m.


Lemma phirho : (n:N)(phi (rho n))=n.



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)).

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).

Mutual Inductive
    N_Deduc : Hyps -> N -> F -> Prop :=
        Implies_I :
                (h:Hyps)(P:F)(n:N)(Q:F)
```

```
                (N_Deduc (Add_Hyp P h) n Q)->
                 (N_Deduc h (lam n) (Impl P Q)) |
        AN_Axiom :
                (h:Hyps)(a:A)(P:F)
                 (A_Deduc h a P)->
                  (N_Deduc h (an a) P)
with
    A_Deduc : Hyps -> A -> F -> Prop :=
        Implies_E :
                (h:Hyps)(a:A)(P:F)(Q:F)(n:N)
                 (A_Deduc h a (Impl P Q))->
                  (N_Deduc h n P)->
                   (A_Deduc h (ap a n) Q) |
        A_Axiom :
                (h:Hyps)(i:V)(P:F)
                 (In_Hyps i P h)->
                  (A_Deduc h (var i) P).

Lemma M_Admis_Psi :
        (h:Hyps)(n:N)(R:F)
         (N_Deduc h n R)->
          (M_Deriv h (psi n) R).

Lemma M_Admis_Psi' :
        (h:Hyps)(a:A)(ms:Ms)(R:F)(P:F)
         (A_Deduc h a P)->
          (Ms_Deriv h P ms R)->
           (M_Deriv h (psi' a ms) R).

Lemma N_Admis_Theta :
        (h:Hyps)(m:M)(R:F)
         (M_Deriv h m R)->
          (N_Deduc h (theta m) R).

Lemma N_Admis_Theta' :
        (h:Hyps)(P:F)(ms:Ms)(R:F)
```

```
        (Ms_Deriv h P ms R)->
        ((a:A)((A_Deduc h a P)->
         (N_Deduc h (theta' a ms) R))).

Recursive Definition
    Weaken_Hyps : nat->F->Hyps->Hyps :=
        O P h => (Add_Hyp P h) |
        (S n) P MT => MT |
        (S n) P (Add_Hyp Q h) => (Add_Hyp Q (Weaken_Hyps n P h)).

Lemma N_Admis_Weaken :
        (h:Hyps)(n:N)(P:F)(j:nat)(Q:F)
         (N_Deduc h n P)->
           (lt j (S (Len_Hyps h)))->
            (N_Deduc (Weaken_Hyps j Q h) (lift_N j n) P).

Lemma A_Admis_Weaken :
        (h:Hyps)(a:A)(P:F)(j:nat)(Q:F)
         (A_Deduc h a P)->
           (lt j (S (Len_Hyps h)))->
            (A_Deduc (Weaken_Hyps j Q h) (lift_A j a) P).

Lemma L_Admis_Weaken :
        (h:Hyps)(l:L)(P,Q:F)(j:nat)
         (L_Deriv h l P)->
          (lt j (S (Len_Hyps h)))->
           (L_Deriv (Weaken_Hyps j Q h) (lift_L j l) P).

Recursive Definition
    Hyps_Exchange : nat->Hyps->Hyps :=
        i MT => MT |
        i (Add_Hyp P MT) => (Add_Hyp P MT) |
        O (Add_Hyp P (Add_Hyp Q h)) =>
                (Add_Hyp Q (Add_Hyp P h)) |
        (S i) (Add_Hyp P (Add_Hyp Q h)) =>
                (Add_Hyp P (Hyps_Exchange i (Add_Hyp Q h))).
```

```
Recursive Definition
    V_Exchange : nat->V->V :=
        i j => (Setifb V (nateqb i j)
                         (S i)
                         (Setifb V (nateqb (S i) j) i j)).

Recursive Definition
    L_Exchange : nat->L->L :=
        i (vr x) => (vr (V_Exchange i x)) |
        i (app x l1 l2) =>
                (app (V_Exchange i x)
                     (L_Exchange i l1)
                     (L_Exchange (S i) l2)) |
        i (lm l) => (lm (L_Exchange (S i) l)).

Lemma L_Admis_Exch :
        (h:Hyps)(l:L)(R:F)(j:nat)(P,Q:F)
         (L_Deriv h l R)->
          (In_Hyps j P h)->
           (In_Hyps (S j) Q h)->
            (L_Deriv (Hyps_Exchange j h)
                     (L_Exchange j l)
                     R).

Lemma RhoBar1 : (x:V)
        (rhobar (sc x mnil))=(vr x).

Lemma RhoBar2 : (ms:Ms)(x:V)(m:M)
        (rhobar (sc x (mcons m ms)))=
         (app x (rhobar m) (rhobar (sc O (lift_Ms O ms)))).

Lemma RhoBar3 : (m:M)
        (rhobar (lambda m))=(lm (rhobar m)).

Lemma L_Admis_RhoBar : (h:Hyps)(m:M)(P:F)
```

```
        (M_Deriv h m P)->
         (L_Deriv h (rhobar m) P).

Lemma L_Admis_Rho : (h:Hyps)(n:N)(P:F)
        (N_Deduc h n P)->
         (L_Deriv h (rho n) P).

Mutual Inductive
    Norm_L : L->Prop :=
        norm_vr : (x:V)(Norm_L (vr x)) |
        norm_app :
                (x:V)(l1,l2:L)
                 (Norm_L l1)->
                  (Norm'_L l2)->
                   (Norm_L (app x l1 l2)) |
        norm_lm :
                (l:L)
                 (Norm_L l)->
                  (Norm_L (lm l))
with
    Norm'_L : L->Prop :=
        norm'_vr : (Norm'_L (vr O)) |
        norm'_app :
                (l1,l2:L)
                 (Norm_L l1)->
                  (Norm'_L l2)->
                   ~(Occurs_In_L O l1)->
                    ~(Occurs_In_L (S O) l2)->
                     (Norm'_L (app O l1 l2)).

Lemma Norm_L_RhoBar : (m:M)
        (Norm_L (rhobar m)).

Lemma Norm'_L_RhoBar : (ms:Ms)
        (Norm'_L (rhobar (sc O (lift_Ms O ms)))).
```

```
Inductive
    L_Perm1 : L->L->Prop :=
        l_perm1_lm :
                (l1,l2:L)
                   (L_Perm1 l1 l2)->
                    (L_Perm1 (lm l1) (lm l2)) |
         l_perm1_app1 :
                (i:V)(l11,l12,l2:L)
                   (L_Perm1 l11 l12)->
                     (L_Perm1 (app i l11 l2) (app i l12 l2)) |
         l_perm1_app2 :
                (i:V)(l1,l21,l22:L)
                    (L_Perm1 l21 l22)->
                      (L_Perm1 (app i l1 l21) (app i l1 l22)) |
         l_perm1_app_wkn :
                (x:V)(l1,l2:L)
                 ~(Occurs_In_L O l2)->
                  (L_Perm1 (app x l1 l2) (drop_L O l2)) |
        l_perm1_app_app1 :
                (x,y:V)(l1,l2,l3:L)
                 ((Occurs_In_L O l2)\/(Occurs_In_L (S O) l3))->
                  (Norm'_L l3)->
                   (L_Perm1 (app x l1 (app (S y) l2 l3))
                            (app y
                                 (app x l1 l2)
                                 (app (lift_V O x)
                                      (lift_L O l1)
                                      (L_Exchange O l3)))) |
         l_perm1_app_app2 :
                (x:V)(l1,l2,l3:L)
                 ((Occurs_In_L O l2)\/(Occurs_In_L (S O) l3))->
                  (Norm'_L l3)->
                   (L_Perm1 (app x l1 (app O l2 l3))
                            (app x
                                 l1
                                 (app O
```

```
                                        (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)))).


Inductive
    L_Permn : L->L->Prop :=
        l_permn_base :
            (l0,l1:L)
               l0=l1->
                (L_Permn l0 l1) |
        l_permn_rec :
            (l0,l1,l2:L)
             (L_Perm1 l0 l1)->
              (L_Permn l1 l2)->
               (L_Permn l0 l2).

Lemma L_Admis_Perm1 :
        (l,l0:L)(h:Hyps)(P:F)
         (L_Perm1 l l0)->
          (L_Deriv h l P)->
           (L_Deriv h l0 P).

Lemma L_Permnn :
        (l,l0,l1:L)
         (L_Permn l l0)->
          (L_Permn l0 l1)->
           (L_Permn l l1).
```

```
Lemma L_Admis_Permn :
        (h:Hyps)(l0,l1:L)(P:F)
         (L_Permn l0 l1)->
          (L_Deriv h l0 P)->
           (L_Deriv h l1 P).

Lemma App_Red_M :
        (x:V)(m1,m:M)
         (L_Permn (app x (rhobar m1) (rhobar m))
                  (rhobar (MsubstVMV x m1 O m))).

Lemma Norm_Red :
        (l:L)(L_Permn l (rhobar (phibar l))).
```

A.2 *CMP* Method Formalisation

The following are some of the main definitions and lemmas from the *CMP* method formalisation examined in §8.

```
Parameter Var:Set.

Parameter Vareqb : Var->Var->bool.

Parameter Vareqb_is_eq1 :
        (x,y:Var)
         x=y->
          (Vareqb x y)=true.

Parameter Vareqb_is_eq2 :
        (x,y:Var)
         (Vareqb x y)=true->
          x=y.

Lemma Vareqb_is_eq3 :
        (x,y:Var)
```

```
        ~x=y->
         (Vareqb x y)=false.

Lemma Vareqb_is_eq4 :
        (x,y:Var)
         (Vareqb x y)=false->
          ~x=y.
Parameter New_Var : (list Var)->Var.

Parameter New_New_Var :
        (l:(list Var))
         ~(In Var (New_Var l) l).

Inductive V : Set :=
        BV : Var->V |
        FV : Var->V.

Recursive Definition
    VBTF : Var->Var->V->V :=
        x y (BV z) => (Setifb V (Vareqb x z) (FV y) (BV z)) |
        x y (FV z) => (FV z).

Recursive Definition
    VFTF : Var->Var->V->V :=
        f1 f2 (BV b) => (BV b) |
        f1 f2 (FV f3) => (FV (Setifb Var (Vareqb f1 f3) f2 f3)).

Mutual Inductive
    N:Set :=
        lam : Var->N->N |
        an : A->N
with
    A:Set :=
        ap : A->N->A |
        var : V->A.
```

```
Fixpoint
    NBTF1 [n:N]: Var->Var->N :=
        [b,f:Var]Cases n of
            (lam x n') =>
                (Setifb N (Vareqb x b)
                        (lam x n')
                        (lam x (NBTF1 n' b f))) |
            (an a) => (an (ABTF1 a b f))
        end with
    ABTF1 [a:A]: Var->Var->A :=
        [b,f:Var]Cases a of
            (ap a' n) => (ap (ABTF1 a' b f) (NBTF1 n b f)) |
            (var x) => (var (VBTF b f x))
        end.

Mutual Inductive
    Nclosed : N->Prop :=
        lamclosed :
                (x,y:Var)(n:N)
                 (Nclosed (NBTF x y n))->
                  (Nclosed (lam x n)) |
        anclosed :
                (a:A)
                 (Aclosed a)->
                  (Nclosed (an a))
with
    Aclosed : A->Prop :=
        apclosed :
                (a:A)(n:N)
                 (Aclosed a)->
                  (Nclosed n)->
                   (Aclosed (ap a n)) |
        varclosed :
                (x:Var)
                  (Aclosed (var (FV x))).
```

```
Mutual Inductive
    Nclosed' : N->Prop :=
        lamclosed' :
                (x:Var)(n:N)
                 ((y:Var)(Nclosed' (NBTF x y n)))->
                  (Nclosed' (lam x n)) |
        anclosed' :
                (a:A)
                 (Aclosed' a)->
                  (Nclosed' (an a))
with
    Aclosed' : A->Prop :=
        apclosed' :
                (a:A)(n:N)
                 (Aclosed' a)->
                  (Nclosed' n)->
                   (Aclosed' (ap a n)) |
        varclosed' :
                (x:Var)
                  (Aclosed' (var (FV x))).


Mutual Inductive
    Neq : N->N->Prop :=
        lameq :
                (x,y,f:Var)(n1,n2:N)
                 ~(Free_In_N f n1)->
                  ~(Free_In_N f n2)->
                   (Neq (NBTF x f n1) (NBTF y f n2))->
                    (Neq (lam x n1) (lam y n2)) |
        aneq :
                (a1,a2:A)
                 (Aeq a1 a2)->
                  (Neq (an a1) (an a2))
with
    Aeq : A->A->Prop :=
        apeq :
```

```
              (a1:A)(n1:N)(a2:A)(n2:N)
               (Aeq a1 a2)->
                (Neq n1 n2)->
                 (Aeq (ap a1 n1) (ap a2 n2)) |
        vareq :
              (x:Var)
               (Aeq (var (FV x)) (var (FV x))).


Mutual Inductive
    Neq' : N->N->Prop :=
        lameq' :
              (x,y:Var)(n1,n2:N)
               ((f:Var)~(Free_In_N f n1)->
                ~(Free_In_N f n2)->
                 (Neq' (NBTF x f n1) (NBTF y f n2)))->
                  (Neq' (lam x n1) (lam y n2)) |
        aneq' :
              (a1,a2:A)
               (Aeq' a1 a2)->
                (Neq' (an a1) (an a2))
with
    Aeq' : A->A->Prop :=
        apeq' :
              (a1:A)(n1:N)(a2:A)(n2:N)
               (Aeq' a1 a2)->
                (Neq' n1 n2)->
                 (Aeq' (ap a1 n1) (ap a2 n2)) |
        vareq' :
              (x:Var)
               (Aeq' (var (FV x)) (var (FV x))).


Lemma N_A_eq_ind' :
        (P:(n,n0:N)(Neq n n0)->Prop)
        (P0:(a,a0:A)(Aeq a a0)->Prop)
         ((x,y:Var)(n1,n2:N)
           (n:(f:Var)~(Free_In_N f n1)->~(Free_In_N f n2)->
```

```
                (Neq (NBTF x f n1) (NBTF y f n2)))
         ((f:Var)
           (n0:~(Free_In_N f n1))
            (n3:~(Free_In_N f n2))
             (P (NBTF x f n1) (NBTF y f n2) (n f n0 n3)))->
          (P (lam x n1) (lam y n2) (lameq x y n1 n2 n)))->
      ((a1,a2:A)(a:(Aeq a1 a2))
           (P0 a1 a2 a)->(P (an a1) (an a2) (aneq a1 a2 a)))->
         ((a1:A)(n1:N)(a2:A)(n2:N)
                (a:(Aeq a1 a2))
                 (P0 a1 a2 a)->
                  (n:(Neq n1 n2))
                     (P n1 n2 n)->
                      (P0 (ap a1 n1) (ap a2 n2)
                          (apeq a1 n1 a2 n2 a n)))->
           ((x:Var)(P0 (var (FV x)) (var (FV x)) (vareq x)))->
           ((n,n0:N)(n1:(Neq n n0))(P n n0 n1))/\
            ((a,a0:A)(a1:(Aeq a a0))(P0 a a0 a1)).
Mutual Inductive
    N_Deduc : Hyps -> N -> F -> Prop :=
        Implies_I :
                (H:Hyps)(P:F)(b,f:Var)(n:N)(Q:F)
                 ~(Free_In_N f n)->
                  ~(Free_In_Hyps f H)->
                   (N_Deduc (Add_Hyp f P H)
                            (NBTF b f n) Q)->
                    (N_Deduc H (lam b n) (Impl P Q)) |
        AN_Axiom :
                (H:Hyps)(a:A)(P:F)
                 (A_Deduc H a P)->
                  (N_Deduc H (an a) P)
with
    A_Deduc : Hyps -> A -> F -> Prop :=
        Implies_E :
                (H:Hyps)(a:A)(P:F)(Q:F)(n:N)
                 (A_Deduc H a (Impl P Q))->
```

```
                (N_Deduc H n P)->
                 (A_Deduc H (ap a n) Q) |
        A_Axiom :
               (H:Hyps)(i:Var)(P:F)
                (In_Hyps i P H)->
                 (A_Deduc H (var (FV i)) P).

Lemma Neq_Deduc :
        (H:Hyps)(n1,n2:N)(P:F)
         (N_Deduc H n1 P)->
          (Neq n1 n2)->
           (N_Deduc H n2 P).

Lemma Aeq_Deduc :
        (H:Hyps)(a1,a2:A)(P:F)
         (A_Deduc H a1 P)->
          (Aeq a1 a2)->
           (A_Deduc H a2 P).

Lemma Meq_Deriv :
        (H:Hyps)(m1,m2:M)(P:F)
         (M_Deriv H m1 P)->
          (Meq m1 m2)->
           (M_Deriv H m2 P).

Lemma Mseq_Deriv :
        (H:Hyps)(ms1,ms2:Ms)(P,Q:F)
         (Ms_Deriv H P ms1 Q)->
          (Mseq ms1 ms2)->
           (Ms_Deriv H P ms2 Q).

Lemma N_Admis_Theta :
        (h:Hyps)(m:M)(R:F)
         (M_Deriv h m R)->
          (N_Deduc h (theta m) R).
```

```
Lemma N_Admis_Theta' :
        (h:Hyps)(P:F)(ms:Ms)(R:F)
         (Ms_Deriv h P ms R)->
         ((a:A)((A_Deduc h a P)->
          (N_Deduc h (theta' a ms) R))).

Lemma M_Admis_Psi :
        (h:Hyps)(n:N)(R:F)
         (N_Deduc h n R)->
          (M_Deriv h (psi n) R).

Lemma M_Admis_Psi' :
        (h:Hyps)(a:A)(ms:Ms)(R:F)(P:F)
         (A_Deduc h a P)->
          (Ms_Deriv h P ms R)->
           (M_Deriv h (psi' a ms) R).
```

Appendix B

Full Development in *Coq* using de Bruijn Indices

This appendix includes all the definitions and the statements of the lemmas proved in the development of the meta-theory from §2 using de Bruijn indices (approximately 4000 lines of *Coq* code). Not included are the many lines of proof script (an extra 6500 lines approximately).

```
Declare ML Module "autocontra".

Grammar tactic simple_tactic :=
        [ "Auto_Contra" identarg($id)] ->
        [(TRY (auto_contra $id))].

Grammar tactic simple_tactic :=
        [ "Auto_Contra"] ->
        [(TRY (auto_contran))].

Require Bool.

Lemma bool_dec1
     : (b:bool)~(Is_true b)->(Is_true (negb b)).

Lemma bool_dec2
     : (b:bool)~b=true->(Is_true (negb b)).

Lemma bool_dec3
     : (b:bool)(Is_true b)->(b=true).

Lemma bool_dec4
     : (b:bool)~b=false->b=true.

Lemma bool_dec5
     : (b:bool)~b=true->b=false.

Lemma bool_dec6
     : (b:bool)b=false->~b=true.

Lemma bool_dec7
     : (b:bool)b=true->~b=false.

Section boolean_extension.

Hypothesis genset:Set.
```



```
Recursive Definition
    Setifb : bool->genset->genset->genset :=
        true x y => x |
        false x y => y.

Lemma orbor : (b1,b2:bool)
        (orb b1 b2)=true->b1=true\/b2=true.

End boolean_extension.

Lemma ororb : (b1,b2:bool)
        (b1=true\/b2=true)->(orb b1 b2)=true.

Lemma orbor1 :  (b1,b2:bool)
        (orb b1 b2)=false->b1=false/\b2=false.

Lemma ororb1 : (b1,b2:bool)
        (b1=false/\b2=false)->(orb b1 b2)=false.

Lemma sym_andb : (b1,b2:bool)
        (andb b1 b2)=(andb b2 b1).

Lemma andbf : (b:bool)
        (andb b false)=false.

Inductive
    lt : nat->nat->Prop :=
        lt_O : (i:nat)(lt O (S i)) |
        lt_S : (i,j:nat)(lt i j)->(lt (S i) (S j)).

Lemma S1 : (i,j:nat)(i=j)->~i=(S j).

Lemma S2 : (i,j:nat)(i=j)->~i=(S (S j)).

Lemma Splus : (i,j:nat)(plus i (S j))=(S (plus i j)).
```

```
Lemma Not_Splus : (i,j:nat)~(S (plus i j))=j.

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

Lemma nateqb_sym : (i,j:nat)(nateqb i j)=(nateqb j i).

Lemma nateqb_is_eq1 : (i,j:nat)i=j->(nateqb i j)=true.

Lemma nateqb_is_eq2 : (i,j:nat)(nateqb i j)=true->i=j.

Lemma nateqb_is_eq3 : (i,j:nat)(~i=j)->(nateqb i j)=false.

Lemma nateqb_is_eq4 : (i,j:nat)((nateqb i j)=false)->~i=j.

Definition nat_compare1 : nat->nat->Prop :=
[i,j:nat]((nateqb i j)=true)\/(nateqb i j)=false.

Lemma nateqb_dec : (i,j:nat)(nat_compare1 i j).

Definition nat_compare : nat->nat->Prop :=
[i,j:nat]i=j\/~i=j.

Lemma nateq_dec : (i,j:nat)(nat_compare i j).

Lemma nateq_dec1 : (i,j:nat)(P:Prop)
                i=j->
                 ~i=j->
                  P.

Recursive Definition
    ltb : nat->nat->bool :=
        0 0 => false |
```



```
        0 (S j) => true |
        (S i) 0 => false |
        (S i) (S j) => (ltb i j).

Lemma ltb_is_lt1 : (i,j:nat)
                (lt i j)->(ltb i j)=true.

Lemma ltb_is_lt2 : (i,j:nat)
                (ltb i j)=true->(lt i j).

Lemma ltb_is_lt3 : (i,j:nat)
                ~(lt i j)->(ltb i j)=false.

Lemma ltb_is_lt4 : (i,j:nat)
                (ltb i j)=false->~(lt i j).

Lemma lt_not_eq1 : (i,j:nat)
                (lt i j)->~i=j.

Definition nat_compare2 : nat->nat->Prop :=
[i,j:nat](lt i j)\/i=j\/(lt j i).

Lemma natlt_dec : (i,j:nat)(nat_compare2 i j).

Lemma ltS : (i,j:nat)
                (lt i j)->(lt i (S j)).

Lemma ltSplus1 : (i,j:nat)
                (lt i (S (plus i j))).

Lemma ltSplus2 : (i,j:nat)
                (lt i (S (plus j i))).

Lemma ltSplus3 : (i,j,k:nat)
                (lt i j)->
                 (lt i (S (plus k j))).
```

```
Lemma ltplus1 : (i,j,k:nat)
               (lt i k)->
                (lt O j)->
                 (lt i (plus j k)).

Lemma plus_bridge : (i,j:nat)
               (plus i (S j))=(S (plus i j)).

Lemma Slt : (i,j:nat)
               (lt (S i) j)->
                (lt i j).

Lemma notltbii : (i,j:nat)
               i=j->
                (ltb i j)=false.

Lemma ltplus2 : (j,h,i:nat)
               (lt i h)->
                (lt i (plus j h)).

Lemma lt_trans : (k,i,j:nat)
               (lt i j)->
                (lt j k)->
                 (lt i k).

Lemma lt_trans1 : (i,k,j:nat)
               (lt i j)->
                (lt j k)->
                 (lt (S i) k).

Lemma ltnotlt : (i,j:nat)
        (lt i j)->~(lt j i).

Lemma ltnot0 : (i,j:nat)
        (lt i j)->~j=O.
```



```
Lemma not_eq_notO_not_pred_eq : (i,j:nat)
        ~i=O->
         ~j=O->
          ~i=j->
           ~(pred i)=(pred j).

Lemma ltiSi : (i,j:nat)
        i=j->
         (lt i (S j)).

Lemma notltii : (i,j:nat)
        i=j->
         ~(lt i j).

Lemma ltS_ltpred : (i,j:nat)
         (lt (S i) j)->
          (lt i (pred j)).

Lemma ltpred_ltS : (i,j:nat)
        (lt i (pred j))->
         (lt (S i) j).

Lemma lt_S_le : (j,i:nat)
        (lt i j)->
         (S i)=j\/(lt (S i) j).

Lemma lt_S_le2 : (i,j:nat)
        (lt i (S j))->
         (lt i j)\/i=j.

Lemma lt_trans2 : (i,j,k:nat)
        (lt i (S j))->
         (lt j k)->
          (lt i k).
```

```
Lemma lt_trans3 : (i,j,k:nat)
        (lt i j)->
         (lt j (S k))->
          (lt i k).

Lemma not_lt_eq_lt : (i,j:nat)
        ~(lt i j)->
         i=j\/(lt j i).

Lemma ltS_neq_lt : (j,i:nat)
        (lt i (S j))->
         ~i=j->
          (lt i j).

Lemma lt_not_ltpred : (i,j:nat)
        (lt i j)->
         ~(lt (pred j) i).

Lemma lt_not_ltS : (i,j:nat)
        (lt i j)->
         ~(lt j (S i)).

Lemma plus_eq2 : (i,j:nat)
        (plus (S i) j)=
         (S (plus i j)).

Lemma plus_right_id : (i:nat)
        (plus i O)=i.

Lemma sym_plus : (i,j:nat)
        (plus i j)=(plus j i).

Lemma ltS_not_lt : (i,j:nat)
        (lt i (S j))->
         ~(lt j i).
```



```
Lemma ltplus3 : (i,j:nat)
        (lt O j)->
         (lt i (plus j i)).

Lemma ltplus4 : (i,j,k:nat)
        (lt (plus i j) k)->
         (lt i k).

Lemma ltplus6: (i,j,k,n:nat)
        (lt k i)->
         (lt n j)->
          (lt (plus k n) (plus i j)).

Lemma ltplus5: (k,j,i,n:nat)
        (lt (plus i j) (plus k n))->
         (lt i k)\/(lt j k)\/(lt i n)\/(lt j n).

Recursive Definition
    max_nat : nat->nat->nat :=
        i j => (Setifb nat (ltb i j) j i).

Lemma max_nat0 : (i:nat)
        (max_nat i O)=i.

Lemma sym_max_nat : (i,j:nat)
        (max_nat i j)=(max_nat j i).

Lemma lt_max_nat : (i,j,k:nat)
        (lt i (S (max_nat j k)))->
         ((lt i (S j)) /\ (lt k (S j)))\/
          ((lt i (S k)) /\ (lt j (S k))).

Lemma eq_max_nat : (i,j,k:nat)
        i=(S (max_nat j k))->
         (i=(S j)/\ (lt k (S j)))\/
          (i=(S k)/\ (lt j (S k))).
```

```
Lemma lt_max_nat1 : (i,j,k:nat)
        i=j->
         (lt i (S (max_nat j k))).

Lemma lt_max_nat2 :
        (i,j,k:nat)
         (lt i j)->
          (lt i (max_nat j k)).

Lemma S_max_nat_bridge0 :
        (i,j:nat)
         (S (max_nat i j))=(max_nat (S i) (S j)).

Grammar tactic simple_tactic :=
        [ "NComp2" command:command($i) command:command($j)] ->
        [let $1 = <<(nat_compare2 $i $j)>> in
         <:tactic:< Cut $1 ;
                [Destruct 1; [Intro | Destruct 1; Intro] | Auto]>>].

Grammar tactic simple_tactic :=
        [ "NComp" command:command($i) command:command($j)] ->
        [let $1 = <<(nat_compare $i $j)>> in
         <:tactic:< Cut $1 ;
                 [Destruct 1; Intro |
                  Auto]>>].

Grammar tactic simple_tactic :=
        [ "Induction_clear" identarg($i)] ->
        [ <:tactic:<Induction $i; Clear $i>>].

Grammar tactic simple_tactic :=
        [ "Injection_clear" identarg($i)] ->
        [ <:tactic:<Injection $i; Clear $i; Intros>>].

Inductive F:Set :=
```



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

Recursive Definition Feqb : F->F->bool :=
(form x) (form y) => (nateqb x y) |
(form x) (Impl P' Q') => false |
(Impl P Q) (form y) => false |
(Impl P Q) (Impl P' Q') => (andb (Feqb P P') (Feqb Q Q')).

Lemma Feqb_sym : (P,Q:F)(Feqb P Q)=(Feqb Q P).

Lemma Feqb_is_eq1 : (P,Q:F)(P=Q)->(Feqb P Q)=true.

Lemma Feqb_is_eq2 :
        (P,Q:F)(Feqb P Q)=true->P=Q.

Lemma Feqb_is_eq3 : (P,Q:F)(~P=Q)->(Feqb P Q)=false.

Lemma Feqb_is_eq4 : (P,Q:F)(Feqb P Q)=false->~P=Q.

Definition F_compare1 : F->F->Prop :=
[P,Q:F]((Feqb P Q)=true)\/(Feqb P Q)=false.

Lemma Feqb_dec : (P,Q:F)(F_compare1 P Q).

Definition F_compare : F->F->Prop :=
                [P,Q:F]P=Q\/~P=Q.

Lemma Feq_dec : (P,Q:F)(F_compare P Q).

Lemma Feq_dec1 : (i,j:F)(P:Prop)
                i=j->
                 ~i=j->
                  P.

Lemma Feq_dec2 : (i,j:F)(P:Prop)
```

```
        i=j->
         ~i=j->
          ~P.

Grammar command command3 :=
        ["Hyps"] -> [$0=<<(list F)>>].

Grammar command command3 :=
        ["MT"] -> [$0=<<(nil F)>>].

Grammar command command3 :=
        ["Add_Hyp" command:command($f) command:command($h)] ->
        [$0 = <<(cons F $f $h)>>].

Grammar command command3 :=
        ["Len_Hyps" command:command($h)] ->
        [$0 = <<(length F $h)>>].

Lemma Add_Hyp1 :
        (i:Hyps)(P:F)~(Add_Hyp P i)=i.

Recursive Definition
    Hypseqb : Hyps->Hyps->bool :=
        MT MT => true |
        MT (Add_Hyp P' h') => false |
        (Add_Hyp P h) MT => false |
        (Add_Hyp P h) (Add_Hyp P' h') => (andb (Feqb P P') (Hypseqb h h')).

Lemma Hypseqb_sym :
        (i,j:Hyps)(Hypseqb i j)=(Hypseqb j i).

Lemma Hypseqb_is_eq1 :
        (i,j:Hyps)i=j->(Hypseqb i j)=true.

Lemma Hypseqb_is_eq2 :
        (i,j:Hyps)(Hypseqb i j)=true->i=j.
```



```
Lemma Hypseqb_is_eq3 :
        (i,j:Hyps)(~i=j)->(Hypseqb i j)=false.

Lemma Hypseqb_is_eq4 :
        (i,j:Hyps)(Hypseqb i j)=false->~i=j.

Definition Hyps_compare1 : Hyps->Hyps->Prop :=
[i,j:Hyps]((Hypseqb i j)=true)\/(Hypseqb i j)=false.

Lemma Hypseqb_dec :
        (i,j:Hyps)(Hyps_compare1 i j).

Definition Hyps_compare :Hyps->Hyps->Prop :=
        [i,j:Hyps]i=j\/~i=j.

Lemma Hypseq_dec :
        (i,j:Hyps)(Hyps_compare i j).

Inductive
    In_Hyps : nat->F->Hyps->Prop :=
        inhyps_base : (P:F)(h:Hyps)
                        (In_Hyps O P (Add_Hyp P h)) |
        inhyps_rec : (n:nat)(P,Q:F)(h:Hyps)
                        (In_Hyps n P h)->
                         (In_Hyps (S n) P (Add_Hyp Q h)).

Definition V :Set := nat.

Lemma In_lt :
        (h:Hyps)(x:V)(P:F)
         (In_Hyps x P h)->
          (lt x (Len_Hyps h)).

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

Scheme M_Ms_ind1 := Induction for M Sort Prop
 with Ms_M_ind1 := Induction for Ms Sort Prop.

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

Mutual Inductive
  N:Set :=
    lam : N->N |
    an : A->N
with
  A:Set :=
    ap : A->N->A |
    var : V->A.

Scheme N_A_ind1 := Induction for N Sort Prop
 with A_N_ind1 := Induction for A Sort Prop.
```



```
Lemma N_A_ind
   : (P:N->Prop)
     (P0:A->Prop)
      ((n:N)(P n)->(P (lam n)))
       ->((a:A)(P0 a)->(P (an a)))
         ->((a:A)(P0 a)->(n:N)(P n)->(P0 (ap a n)))
           ->((v:V)(P0 (var v)))
             ->(((n:N)(P n)) /\ ((a:A)(P0 a))).

Fixpoint
  theta [m:M]:N :=
     <N>Case m of
       [x:V][ms:Ms]
          (theta1' ms (var x))
       [m:M]
          (lam (theta m))
end with
   theta1' [ms:Ms]:A->N :=
     [a:A]<N>Case ms of
          (an a)
       [m:M][ms:Ms]
          (theta1' ms (ap a (theta m)))
end.

Recursive Definition theta' : A -> Ms -> N :=
a ms => (theta1' ms a).

Lemma th1 : (x:V)(ms:Ms)((theta (sc x ms)) = (theta' (var x) ms)).

Lemma th2 : (m:M)((theta (lambda m)) = (lam (theta m))).

Lemma th3 : (a:A)((theta' a mnil) = (an a)).

Lemma th4 : (m:M)(ms:Ms)(a:A)((theta' a (mcons m ms))
      = (theta' (ap a (theta m)) ms)).
```

```
Fixpoint
    psi [n:N]:M :=
        <M>Case n of
            [n:N]
                (lambda (psi n))
            [a:A]
                (psi' a mnil)
end with
    psi' [a:A]:Ms->M :=
        [ms:Ms]<M>Case a of
            [a':A][n:N]
                (psi' a' (mcons (psi n) ms))
            [x:V]
                (sc x ms)
end.

Lemma ps1 : (n:N)((psi (lam n)) = (lambda (psi n))).

Lemma ps2 : (a:A)((psi (an a)) = (psi' a mnil)).

Lemma ps3 : (a:A)(n:N)(ms:Ms)((psi' (ap a n) ms)
        = (psi' a (mcons (psi n) ms))).

Lemma ps4 : (x:V)(ms:Ms)((psi' (var x) ms) = (sc x ms)).

Definition thpsids :N->Prop :=
[n:N]((theta (psi n)) = n).

Definition thps'th's :A->Prop :=
[a:A](ms:Ms)((theta (psi' a ms)) = (theta' a ms)).

Lemma thpsid : ((n:N)(thpsids n))/\((a:A)(thps'th's a)).

Lemma thetapsi:
        (n:N)((theta(psi n)) = n).
```



```
Lemma thetapsi'theta':
        (a:A)(ms:Ms)((theta (psi' a ms)) = (theta' a ms)).

Definition psthids :M->Prop :=
[m:M]((psi (theta m)) = m).

Definition psth'ps's :Ms->Prop :=
[ms:Ms](a:A)((psi (theta' a ms)) = (psi' a ms)).

Lemma psthid : ((m:M)(psthids m))/\((ms:Ms)(psth'ps's ms)).

Lemma psitheta:
        (m:M)((psi(theta m))=m).

Lemma psitheta'psi':
        (ms:Ms)(a:A)((psi (theta' a ms)) = (psi' a ms)).

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)).

Fixpoint
    lift_M1 [m:M] : nat->M :=
        [i:nat]<M>Case m of
            [x:V][ms:Ms]
                (sc (lift_V i x) (lift_Ms1 ms i))
            [m':M]
                (lambda (lift_M1 m' (S i)))
end with
```

```
    lift_Ms1 [ms:Ms] : nat->Ms :=
        [i:nat]<Ms>Case ms of
                mnil
            [m:M][ms':Ms]
                (mcons (lift_M1 m i) (lift_Ms1 ms' i))
end.

Recursive Definition
    lift_M : nat->M->M :=
        i m => (lift_M1 m i).

Recursive Definition
    lift_Ms : nat->Ms->Ms :=
        i ms => (lift_Ms1 ms i).

Fixpoint
    lift_N1 [n:N] : nat->N :=
        [i:nat]<N>Case n of
            [n':N]
                (lam (lift_N1 n' (S i)))
            [a:A]
                (an (lift_A1 a i))
end with
    lift_A1 [a:A] : nat->A :=
        [i:nat]<A>Case a of
            [a':A][n:N]
                (ap (lift_A1 a' i) (lift_N1 n i))
            [x:V]
                (var (lift_V i x))
end.

Recursive Definition lift_N : nat->N->N :=
i n => (lift_N1 n i).

Recursive Definition lift_A : nat->A->A :=
i a => (lift_A1 a i).
```



```
Lemma LIFTM1 : (i:nat)(x:V)(ms:Ms)
        ((lift_M i (sc x ms)) =
         (sc (lift_V i x) (lift_Ms i ms))).

Lemma LIFTM2 : (i:nat)(m:M)
        ((lift_M i (lambda m)) = (lambda (lift_M (S i) m))).

Lemma LIFTM3 : (i:nat)(lift_Ms i mnil)=mnil.

Lemma LIFTM4 : (i:nat)(m:M)(ms:Ms)((lift_Ms i (mcons m ms)) =
                                (mcons (lift_M i m) (lift_Ms i ms))).

Lemma LIFTN1 : (i:nat)(n:N)((lift_N i (lam n)) = (lam (lift_N (S i) n))).

Lemma LIFTN2 : (i:nat)(a:A)((lift_N i (an a)) = (an (lift_A i a))).

Lemma LIFTN3 : (i:nat)(a:A)(n:N)
        ((lift_A i (ap a n)) = (ap (lift_A i a) (lift_N i n))).

Lemma LIFTN4 : (i:nat)(x:V)((lift_A i (var x)) = (var (lift_V i x))).

Lemma Lift_Lift_V_Bridge : (x:V)(i,j:nat)
        (lt i j)->
         (lift_V i (lift_V j x))=
          (lift_V (S j) (lift_V i x)).

Lemma Lift_Lift_L_Bridge : (l:L)(i,j:nat)
        (lt i j)->
         (lift_L i (lift_L j l))=
          (lift_L (S j) (lift_L i l)).

Definition lift_lift_n_bridge : N->Prop :=
        [n:N](i,j:nat)
        (lt i j)->
         (lift_N i (lift_N j n))=
```

```
        (lift_N (S j) (lift_N i n)).

Definition lift_lift_a_bridge : A->Prop :=
      [a:A](i,j:nat)
      (lt i j)->
       (lift_A i (lift_A j a))=
        (lift_A (S j) (lift_A i a)).

Lemma lift_lift_n_Bridge :
      ((n:N)(lift_lift_n_bridge n))/\
       ((a:A)(lift_lift_a_bridge a)).

Lemma Lift_Lift_N_Bridge :
      (n:N)(i,j:nat)
      (lt i j)->
       (lift_N i (lift_N j n))=
        (lift_N (S j) (lift_N i n)).

Lemma Lift_Lift_A_Bridge :
      (a:A)(i,j:nat)
      (lt i j)->
       (lift_A i (lift_A j a))=
        (lift_A (S j) (lift_A i a)).

Lemma Lift_Lift_V_Bridge0 : (x:V)(i,j:nat)
      (lt j i)->
       (lift_V i (lift_V j x))=
        (lift_V j (lift_V (pred i) x)).

Lemma Lift_Lift_L_Bridge0 : (l:L)(i,j:nat)
      (lt j i)->
       (lift_L i (lift_L j l))=
        (lift_L j (lift_L (pred i) l)).

Definition lift_lift_n_bridge0 : N->Prop :=
      [n:N](i,j:nat)
```



```
      (lt j i)->
       (lift_N i (lift_N j n))=
        (lift_N j (lift_N (pred i) n)).

Definition lift_lift_a_bridge0 : A->Prop :=
      [a:A](i,j:nat)
      (lt j i)->
       (lift_A i (lift_A j a))=
        (lift_A j (lift_A (pred i) a)).

Lemma lift_lift_n_Bridge0 :
      ((n:N)(lift_lift_n_bridge0 n))/\
       ((a:A)(lift_lift_a_bridge0 a)).

Lemma Lift_Lift_N_Bridge0 :
      (n:N)(i,j:nat)
      (lt j i)->
       (lift_N i (lift_N j n))=
        (lift_N j (lift_N (pred i) n)).

Lemma Lift_Lift_A_Bridge0 :
      (a:A)(i,j:nat)
      (lt j i)->
       (lift_A i (lift_A j a))=
        (lift_A j (lift_A (pred i) a)).

Lemma Lift_Lift_V_Bridge1 : (x:V)(i,j:nat)
      i=j->
       (lift_V i (lift_V j x))=(lift_V (S j) (lift_V i x)).

Definition lift_lift_n_bridge1 : N->Prop :=
      [n:N](i,j:nat)
       i=j->
        (lift_N i (lift_N j n))=(lift_N (S j) (lift_N i n)).

Definition lift_lift_a_bridge1 : A->Prop :=
```

```
        [a:A](i,j:nat)
         i=j->
          (lift_A i (lift_A j a))=(lift_A (S j) (lift_A i a)).

Lemma lift_lift_n_Bridge1 :
        ((n:N)(lift_lift_n_bridge1 n))/\((a:A)(lift_lift_a_bridge1 a)).

Lemma Lift_Lift_N_Bridge1 :
        (n:N)(i,j:nat)
         i=j->
          (lift_N i (lift_N j n))=(lift_N (S j) (lift_N i n)).

Lemma  Lift_Lift_A_Bridge1 :
        (a:A)(i,j:nat)
         i=j->
          (lift_A i (lift_A j a))=(lift_A (S j) (lift_A i a)).

Lemma Lift_Lift_L_Bridge1 :(l:L)(i,j:nat)
         i=j->
          (lift_L i (lift_L j l))=(lift_L (S j) (lift_L i l)).

Recursive Definition drop_V : nat->V->V :=
j i => (Setifb V (ltb i j) i (pred i)).

Recursive Definition
    drop_L : nat->L->L :=
        i (vr x) => (vr (drop_V i x)) |
        i (app x l1 l2) =>
                (app (drop_V i x) (drop_L i l1) (drop_L (S i) l2)) |
        i (lm l) => (lm (drop_L (S i) l)).

Fixpoint
    drop_M1 [m:M] : nat->M :=
        [i:nat]<M>Case m of
            [x:V][ms:Ms]
                (sc (drop_V i x) (drop_Ms1 ms i))
```



```
            [m':M]
                (lambda (drop_M1 m' (S i)))
end with
    drop_Ms1 [ms:Ms] : nat->Ms :=
        [i:nat]<Ms>Case ms of
                mnil
            [m:M][ms':Ms]
                (mcons (drop_M1 m i) (drop_Ms1 ms' i))
end.

Recursive Definition
    drop_M : nat->M->M :=
        i m => (drop_M1 m i).

Recursive Definition
    drop_Ms : nat->Ms->Ms :=
        i ms => (drop_Ms1 ms i).

Fixpoint
    drop_N1 [n:N] : nat->N :=
        [i:nat]<N>Case n of
            [n':N]
                (lam (drop_N1 n' (S i)))
            [a:A]
                (an (drop_A1 a i))
end with
    drop_A1 [a:A] : nat->A :=
        [i:nat]<A>Case a of
            [a':A][n:N]
                (ap (drop_A1 a' i) (drop_N1 n i))
            [x:V]
                (var (drop_V i x))
end.

Recursive Definition
    drop_N : nat->N->N :=
```

```
        i n => (drop_N1 n i).

Recursive Definition
    drop_A : nat->A->A :=
        i a => (drop_A1 a i).

Lemma DROPM1 : (i:nat)(x:V)(ms:Ms)((drop_M i (sc x ms)) =
                            (sc (drop_V i x) (drop_Ms i ms))).

Lemma DROPM2 : (i:nat)(m:M)
        ((drop_M i (lambda m)) = (lambda (drop_M (S i) m))).

Lemma DROPM3 : (i:nat)(drop_Ms i mnil)=mnil.

Lemma DROPM4 : (i:nat)(m:M)(ms:Ms)((drop_Ms i (mcons m ms)) =
                            (mcons (drop_M i m) (drop_Ms i ms))).

Lemma DROPN1 : (i:nat)(n:N)((drop_N i (lam n)) = (lam (drop_N (S i) n))).

Lemma DROPN2 : (i:nat)(a:A)((drop_N i (an a)) = (an (drop_A i a))).

Lemma DROPN3 : (i:nat)(a:A)(n:N)
        ((drop_A i (ap a n)) = (ap (drop_A i a) (drop_N i n))).

Lemma DROPN4 : (i:nat)(x:V)((drop_A i (var x)) = (var (drop_V i x))).

Inductive
    Occurs_In_V : nat->V->Prop :=
        Occurs_in_v : (i,j:nat)i=j->
                        (Occurs_In_V i j).

Inductive
    Occurs_In_L : nat->L->Prop :=
        Occurs_in_vr :
                (i:nat)(x:V)
                 (Occurs_In_V i x)->
```



```
                  (Occurs_In_L i (vr x)) |
        Occurs_in_app1 :
                (i:nat)(x:V)(l1,l2:L)
                 (Occurs_In_V i x)->
                  (Occurs_In_L i (app x l1 l2)) |
        Occurs_in_app2 :
                (i:nat)(x:V)(l1,l2:L)
                 (Occurs_In_L i l1)->
                  (Occurs_In_L i (app x l1 l2)) |
        Occurs_in_app3 :
                (i:nat)(x:V)(l1,l2:L)
                 (Occurs_In_L (S i) l2)->
                  (Occurs_In_L i (app x l1 l2)) |
        Occurs_in_lm :
                (i:nat)(l:L)
                 (Occurs_In_L (S i) l)->
                  (Occurs_In_L i (lm l)).

Mutual Inductive
    Occurs_In_M : nat->M->Prop :=
        Occurs_in_sc1 :
                (i:nat)(x:V)(ms:Ms)
                 (Occurs_In_V i x)->
                  (Occurs_In_M i (sc x ms)) |
        Occurs_in_sc2 :
                (i:nat)(x:V)(ms:Ms)
                 (Occurs_In_Ms i ms)->
                  (Occurs_In_M i (sc x ms)) |
        Occurs_in_lambda :
                (i:nat)(m:M)
                 (Occurs_In_M (S i) m)->
                  (Occurs_In_M i (lambda m))
with
    Occurs_In_Ms : nat->Ms->Prop :=
        Occurs_in_mcons1 :
                (i:nat)(m:M)(ms:Ms)
```

```
            (Occurs_In_M i m)->
             (Occurs_In_Ms i (mcons m ms)) |
      Occurs_in_mcons2 :
            (i:nat)(m:M)(ms:Ms)
             (Occurs_In_Ms i ms)->
              (Occurs_In_Ms i (mcons m ms)).

Mutual Inductive
    Occurs_In_N : nat->N->Prop :=
      Occurs_in_lam :
            (i:nat)(n:N)
             (Occurs_In_N (S i) n)->
              (Occurs_In_N i (lam n)) |
      Occurs_in_an :
            (i:nat)(a:A)
             (Occurs_In_A i a)->
              (Occurs_In_N i (an a))
with
    Occurs_In_A : nat->A->Prop :=
      Occurs_in_ap1 :
            (i:nat)(a:A)(n:N)
             (Occurs_In_A i a)->
              (Occurs_In_A i (ap a n)) |
      Occurs_in_ap2 :
            (i:nat)(a:A)(n:N)
             (Occurs_In_N i n)->
              (Occurs_In_A i (ap a n)) |
      Occurs_in_var :
            (i:nat)(x:V)
             (Occurs_In_V i x)->
              (Occurs_In_A i (var x)).

Recursive Definition Occurs_In_V1 : V->V->bool :=
i j => (nateqb i j).

Lemma OIV1_is_OIV1 : (i,x:V)
```



```
                (Occurs_In_V i x)->
                 (Occurs_In_V1 i x)=true.

Lemma OIV1_is_OIV2 : (i:V)(x:V)
                (Occurs_In_V1 i x)=true->
                 (Occurs_In_V i x).

Lemma OIV1_is_OIV3 : (i:V)(x:V)
                ~(Occurs_In_V i x)->
                 (Occurs_In_V1 i x)=false.

Lemma OIV1_is_OIV4 : (i:V)(x:V)
                (Occurs_In_V1 i x)=false->
                 ~(Occurs_In_V i x).

Definition OIV_compare : V->V->Prop :=
[i:V][x:V](Occurs_In_V i x)\/~(Occurs_In_V i x).

Lemma OIV_dec :
        (i:V)(x:V)
         (OIV_compare i x).

Recursive Definition
    Occurs_In_L1 : V->L->bool :=
        i (vr x) => (Occurs_In_V1 i x) |
        i (app  x l1 l2) =>
                (orb (Occurs_In_V1 i x)
                     (orb (Occurs_In_L1 i l1)
                          (Occurs_In_L1 (S i) l2))) |
        i (lm l) => (Occurs_In_L1 (S i) l).

Lemma OIL1_is_OIL1 :
        (l:L)(i:V)
         (Occurs_In_L i l)->
          (Occurs_In_L1 i l)=true.
```

```
Lemma OIL1_is_OIL2 : (l:L)(i:V)
                (Occurs_In_L1 i l)=true->
                 (Occurs_In_L i l).

Lemma OIL1_is_OIL3 :
        (l:L)(i:V)
         ~(Occurs_In_L i l)->
          (Occurs_In_L1 i l)=false.

Lemma OIL1_is_OIL4 :
        (l:L)(i:V)
         (Occurs_In_L1 i l)=false->
          ~(Occurs_In_L i l).

Definition OIL_compare : V->L->Prop :=
[i:V][l:L](Occurs_In_L i l)\/~(Occurs_In_L i l).

Lemma OIL_dec :
        (l:L)(i:V)
         (OIL_compare i l).

Fixpoint
    Occurs_In_M2 [m:M] : V->bool :=
        [i:V]<bool>Case m of
            [x:V][ms:Ms]
                (orb (Occurs_In_V1 i x) (Occurs_In_Ms2 ms i))
            [m':M]
                (Occurs_In_M2 m' (S i))
end with
    Occurs_In_Ms2 [ms:Ms] : V->bool :=
        [i:V]<bool>Case ms of
                false
            [m:M][ms':Ms]
                (orb (Occurs_In_M2 m i) (Occurs_In_Ms2 ms' i))
end.
```



```
Recursive Definition
    Occurs_In_M1 : V->M->bool :=
        x m => (Occurs_In_M2 m x).

Recursive Definition
    Occurs_In_Ms1 : V->Ms->bool :=
        x ms => (Occurs_In_Ms2 ms x).

Lemma OIM1 : (i,x:V)(ms:Ms)
        (Occurs_In_M1 i (sc x ms))
        =(orb (Occurs_In_V1 i x)
              (Occurs_In_Ms1 i ms)).

Lemma OIM2 : (i:V)(m:M)
        (Occurs_In_M1 i (lambda m))=
         (Occurs_In_M1 (S i) m).

Lemma OIM3 : (i:V)
        (Occurs_In_Ms1 i mnil)=false.

Lemma OIM4 : (i:V)(m:M)(ms:Ms)
        (Occurs_In_Ms1 i (mcons m ms))=
         (orb (Occurs_In_M1 i m)
              (Occurs_In_Ms1 i ms)).

Definition oim1_is_oim1 : M->Prop :=
[m:M](i:V)(Occurs_In_M i m)->
              (Occurs_In_M1 i m)=true.

Definition oims1_is_oims1 : Ms->Prop :=
[ms:Ms](i:V)(Occurs_In_Ms i ms)->
              (Occurs_In_Ms1 i ms)=true.

Lemma oiM1_is_oiM1 :
        ((m:M)(oim1_is_oim1 m))/\
         ((ms:Ms)(oims1_is_oims1 ms)).
```

```
Lemma OIM1_is_OIM1 : (i:V)(m:M)
(Occurs_In_M i m)->
              (Occurs_In_M1 i m)=true.

Lemma OIMs1_is_OIMs1 : (i:V)(ms:Ms)
(Occurs_In_Ms i ms)->
                (Occurs_In_Ms1 i ms)=true.

Definition oim1_is_oim2 : M->Prop :=
[m:M](i:V)(Occurs_In_M1 i m)=true->
              (Occurs_In_M i m).

Definition oims1_is_oims2 : Ms->Prop :=
[ms:Ms](i:V)(Occurs_In_Ms1 i ms)=true->
                (Occurs_In_Ms i ms).

Lemma oiM1_is_oiM2 :
        ((m:M)(oim1_is_oim2 m))/\
         ((ms:Ms)(oims1_is_oims2 ms)).

Lemma OIM1_is_OIM2 : (i:V)(m:M)
        (Occurs_In_M1 i m)=true->
         (Occurs_In_M i m).

Lemma OIMs1_is_OIMs2 :
        (i:V)(ms:Ms)
         (Occurs_In_Ms1 i ms)=true->
          (Occurs_In_Ms i ms).

Lemma OIM1_is_OIM3 :
        (i:V)(m:M)
         ~(Occurs_In_M i m)->
          (Occurs_In_M1 i m)=false.

Lemma OIMs1_is_OIMs3 :
```



```
        (i:V)(ms:Ms)
         ~(Occurs_In_Ms i ms)->
          (Occurs_In_Ms1 i ms)=false.

Lemma OIM1_is_OIM4 :
        (i:V)(m:M)
         (Occurs_In_M1 i m)=false->
          ~(Occurs_In_M i m).

Lemma OIMs1_is_OIMs4 :
        (i:V)(ms:Ms)
         (Occurs_In_Ms1 i ms)=false->
          ~(Occurs_In_Ms i ms).

Definition OIM_compare : V->M->Prop :=
[i:V][m:M](Occurs_In_M i m)\/~(Occurs_In_M i m).

Lemma OIM_dec :
        (i:V)(m:M)
         (OIM_compare i m).

Definition OIMs_compare : V->Ms->Prop :=
[i:V][ms:Ms](Occurs_In_Ms i ms)\/~(Occurs_In_Ms i ms).

Lemma OIMs_dec :
        (i:V)(ms:Ms)
         (OIMs_compare i ms).

Fixpoint
    Occurs_In_N2 [n:N] : V->bool :=
        [i:V]<bool>Case n of
            [n':N]
                (Occurs_In_N2 n' (S i))
            [a:A]
                (Occurs_In_A2 a i)
end with
```

```
    Occurs_In_A2 [a:A] : V->bool :=
        [i:V]<bool>Case a of
            [a':A][n:N]
                (orb (Occurs_In_A2 a' i) (Occurs_In_N2 n i))
            [x:V]
                (Occurs_In_V1 i x)
end.

Definition Occurs_In_N1 : V->N->bool :=
[i:V][n:N](Occurs_In_N2 n i).

Definition Occurs_In_A1 : V->A->bool :=
[i:V][a:A](Occurs_In_A2 a i).

Lemma OIN1 : (i:V)(n:N)
        (Occurs_In_N1 i (lam n))=
         (Occurs_In_N1 (S i) n).

Lemma OIN2 : (i:V)(a:A)
        (Occurs_In_N1 i (an a))=
         (Occurs_In_A1 i a).

Lemma OIN3 : (i:V)(a:A)(n:N)
        (Occurs_In_A1 i (ap a n))=
         (orb (Occurs_In_A1 i a) (Occurs_In_N1 i n)).

Lemma OIN4 : (i,x:V)
        (Occurs_In_A1 i (var x))=
         (Occurs_In_V1 i x).

Definition oin1_is_oin1 : N->Prop :=
        [n:N](i:V)
         (Occurs_In_N i n)->
          (Occurs_In_N1 i n)=true.

Definition oia1_is_oia1 : A->Prop :=
```



```
        [a:A](i:V)
         (Occurs_In_A i a)->
          (Occurs_In_A1 i a)=true.

Lemma oiN1_is_oiN1 :
        ((n:N)(oin1_is_oin1 n))/\
         ((a:A)(oia1_is_oia1 a)).

Lemma OIN1_is_OIN1 :
        (n:N)(i:V)
         (Occurs_In_N i n)->
          (Occurs_In_N1 i n)=true.

Lemma OIA1_is_OIA1 :
        (a:A)(i:V)
         (Occurs_In_A i a)->
          (Occurs_In_A1 i a)=true.

Definition oin1_is_oin2 : N->Prop :=
[n:N](i:V)(Occurs_In_N1 i n)=true->
            (Occurs_In_N i n).

Definition oia1_is_oia2 : A->Prop :=
[a:A](i:V)(Occurs_In_A1 i a)=true->
            (Occurs_In_A i a).

Lemma oiN1_is_oiN2 :
        ((n:N)(oin1_is_oin2 n))/\
         ((a:A)(oia1_is_oia2 a)).

Lemma OIN1_is_OIN2 :
        (n:N)(i:V)
         (Occurs_In_N1 i n)=true->
          (Occurs_In_N i n).

Lemma OIA1_is_OIA2 :
```

```
        (a:A)(i:V)
         (Occurs_In_A1 i a)=true->
          (Occurs_In_A i a).

Lemma OIN1_is_OIN3 :
        (i:V)(n:N)
         ~(Occurs_In_N i n)->
          (Occurs_In_N1 i n)=false.

Lemma OIA1_is_OIA3 :
        (i:V)(a:A)
         ~(Occurs_In_A i a)->
          (Occurs_In_A1 i a)=false.

Lemma OIN1_is_OIN4 :
        (i:V)(n:N)
         (Occurs_In_N1 i n)=false->
          ~(Occurs_In_N i n).

Lemma OIA1_is_OIA4 :
        (i:V)(a:A)
         (Occurs_In_A1 i a)=false->
          ~(Occurs_In_A i a).

Definition OIN_compare : V->N->Prop :=
[i:V][n:N](Occurs_In_N i n)\/~(Occurs_In_N i n).

Lemma OIN_dec :
        (i:V)(n:N)
         (Occurs_In_N i n)\/~(Occurs_In_N i n).

Definition OIA_compare : V->A->Prop :=
[i:V][a:A](Occurs_In_A i a)\/~(Occurs_In_A i a).

Lemma OIA_dec :
        (i:V)(a:A)
```



```
         (Occurs_In_A i a)\/~(Occurs_In_A i a).

Lemma NOI_Lift_V_Bridge0 :
        (x,i:V)
         ~(Occurs_In_V i x)->
          (lift_V i x)=
           (lift_V (S i) x).

Definition NOI_lift_n_bridge0 : N->Prop :=
        [n:N](i:V)
         ~(Occurs_In_N i n)->
          (lift_N i n)=
           (lift_N (S i) n).

Definition NOI_lift_a_bridge0 : A->Prop :=
        [a:A](i:V)
         ~(Occurs_In_A i a)->
          (lift_A i a)=
           (lift_A (S i) a).

Lemma NOI_lift_n_Bridge0 :
        ((n:N)(NOI_lift_n_bridge0 n))/\
         ((a:A)(NOI_lift_a_bridge0 a)).

Lemma NOI_Lift_N_Bridge0 :
        (n:N)(i:V)
         ~(Occurs_In_N i n)->
          (lift_N i n)=
           (lift_N (S i) n).

Lemma NOI_Lift_A_Bridge0 :
        (a:A)(i:V)
         ~(Occurs_In_A i a)->
          (lift_A i a)=
           (lift_A (S i) a).
```

```
Lemma NOI_Lift_L_Bridge0 :
        (l:L)(i:V)
         ~(Occurs_In_L i l)->
          (lift_L i l)=
           (lift_L (S i) l).

Lemma NOI_Drop_V_Bridge0 :
        (x:V)(i,j:V)
         (lt i j)->
          ~(Occurs_In_V (S j) x)->
           ~(Occurs_In_V j (drop_V i x)).

Definition noi_drop_m_bridge0 : M->Prop :=
        [m:M](i,j:V)
         (lt i j)
          ->~(Occurs_In_M (S j) m)
             ->~(Occurs_In_M j (drop_M i m)).

Definition noi_drop_ms_bridge0 : Ms->Prop :=
        [ms:Ms](i,j:V)
         (lt i j)
          ->~(Occurs_In_Ms (S j) ms)
             ->~(Occurs_In_Ms j (drop_Ms i ms)).

Lemma noi_drop_m_Bridge0 :
        ((m:M)(noi_drop_m_bridge0 m))/\
         ((ms:Ms)(noi_drop_ms_bridge0 ms)).

Lemma NOI_Drop_M_Bridge0 :
        (m:M)(i,j:V)
         (lt i j)
          ->~(Occurs_In_M (S j) m)
             ->~(Occurs_In_M j (drop_M i m)).

Lemma NOI_Drop_Ms_Bridge0 :
        (ms:Ms)(i,j:V)
```



```
         (lt i j)
          ->~(Occurs_In_Ms (S j) ms)
             ->~(Occurs_In_Ms j (drop_Ms i ms)).

Lemma NOI_Lift_V : (x:V)(i:nat)
        ~(Occurs_In_V i (lift_V i x)).

Lemma NOI_Lift_V2 :(x:V)(i,j:nat)
        ~(Occurs_In_V (S i) x)->
         ~(Occurs_In_V i x)->
          ~(Occurs_In_V (S i) (lift_V j x)).

Definition noi_lift_m : M->Prop :=
        [m:M](i:nat)
         ~(Occurs_In_M i (lift_M i m)).

Definition noi_lift_ms : Ms->Prop :=
        [ms:Ms](i:nat)
         ~(Occurs_In_Ms i (lift_Ms i ms)).

Lemma NOI_lift_m :
        ((m:M)(noi_lift_m m))/\
         ((ms:Ms)(noi_lift_ms ms)).

Lemma NOI_Lift_M :
        (m:M)(i:nat)
         ~(Occurs_In_M i (lift_M i m)).

Lemma NOI_Lift_Ms :
        (ms:Ms)(i:nat)
         ~(Occurs_In_Ms i (lift_Ms i ms)).

Definition noi_lift_m1 : M->Prop :=
        [m:M](i:nat)(j:nat)
         ~(Occurs_In_M (S i) m)->
          ~(Occurs_In_M i m)->
```

```
        ~(Occurs_In_M (S i) (lift_M j m)).

Definition noi_lift_ms1 : Ms->Prop :=
        [ms:Ms](i:nat)(j:nat)
         ~(Occurs_In_Ms (S i) ms)->
          ~(Occurs_In_Ms i ms)->
           ~(Occurs_In_Ms (S i) (lift_Ms j ms)).

Lemma NOI_lift_m1 :
        ((m:M)(noi_lift_m1 m))/\
         ((ms:Ms)(noi_lift_ms1 ms)).

Lemma NOI_Lift_M1 :
        (m:M)(i,j:nat)
         ~(Occurs_In_M (S i) m)->
          ~(Occurs_In_M i m)->
           ~(Occurs_In_M (S i) (lift_M j m)).

Lemma NOI_Lift_Ms1 :
        (ms:Ms)(i,j:nat)
         ~(Occurs_In_Ms (S i) ms)->
          ~(Occurs_In_Ms i ms)->
           ~(Occurs_In_Ms (S i) (lift_Ms j ms)).

Lemma NOI_Lift_L : (l:L)(i:nat)
        ~(Occurs_In_L i (lift_L i l)).

Lemma NOI_Lift_V3 : (x:V)(i,j:nat)
        (lt j i)->
         ~(Occurs_In_V i x)->
          ~(Occurs_In_V (S i) (lift_V j x)).

Lemma NOI_Lift_V4 : (x:V)(i,j:nat)
        j=i->
         ~(Occurs_In_V i x)->
          ~(Occurs_In_V (S i) (lift_V j x)).
```



```
Fixpoint MsubstVMV1 [m:M] : V->M->V->M :=
        [x:V][m':M][i:V]<M>Case m of
        [z:V][ms:Ms]
         (Setifb M (nateqb i z) (sc x (mcons m' (MssubstVMV1 ms x m' z)))
                                (sc (drop_V i z) (MssubstVMV1 ms x m' i)))
        [m'':M]
         (lambda (MsubstVMV1 m'' (lift_V O x) (lift_M O m') (S i)))
end with
MssubstVMV1 [ms:Ms] : V->M->V->Ms :=
        [x:V][m':M][i:V]<Ms>Case ms of
        mnil
        [m'':M][ms':Ms]
         (mcons (MsubstVMV1 m'' x m' i) (MssubstVMV1 ms' x m' i))
end.

Recursive Definition
    MsubstVMV : V->M->V->M->M :=
        x m i m' => (MsubstVMV1 m' x m i).

Recursive Definition
    MssubstVMV : V->M->V->Ms->Ms :=
        x m i ms => (MssubstVMV1 ms x m i).


Lemma Drop_Lift_V :
        (x:V)(i:nat)
         (drop_V i (lift_V i x))=x.

Definition drop_lift_m : M->Prop :=
        [m:M](i:nat)(drop_M i (lift_M i m))=m.

Definition drop_lift_ms : Ms->Prop :=
        [ms:Ms](i:nat)(drop_Ms i (lift_Ms i ms))=ms.

Lemma drop_lift_M :
        ((m:M)(drop_lift_m m))/\((ms:Ms)(drop_lift_ms ms)).
```

```
Lemma Drop_Lift_M : (i:nat)(m:M)(drop_M i (lift_M i m))=m.

Lemma Drop_Lift_Ms : (i:nat)(ms:Ms)(drop_Ms i (lift_Ms i ms))=ms.

Lemma MSVMV1 :
        (x:V)(m:M)(y,z:V)(ms:Ms)
          (MsubstVMV x m y (sc z ms)) =
           (Setifb M
                   (nateqb y z)
                   (sc x (mcons m (MssubstVMV x m z ms)))
                   (sc (drop_V y z) (MssubstVMV x m y ms))).

Lemma MSVMV2 : (x:V)(m:M)(m':M)(y:V)
               (MsubstVMV x m y (lambda m')) =
                (lambda (MsubstVMV (lift_V O x) (lift_M O m) (S y) m')).

Lemma MSVMV3 : (x:V)(m:M)(y:V)((MssubstVMV x m y mnil) = mnil).

Lemma MSVMV4 : (x:V)(m:M)(y:V)(m':M)(ms:Ms)
               ((MssubstVMV x m y (mcons m' ms)) =
                (mcons (MsubstVMV x m y m')
                       (MssubstVMV x m y ms))).


Lemma Lift_Drop_V :
        (x:V)(i:nat)
         ~(Occurs_In_V i x)->
          (lift_V i (drop_V i x))=x.

Definition lift_drop_m : M->Prop :=
        [m:M](i:nat)
         ~(Occurs_In_M i m)->
          (lift_M i (drop_M i m))=m.

Definition lift_drop_ms : Ms->Prop :=
        [ms:Ms](i:nat)
```



```
         ~(Occurs_In_Ms i ms)->
          (lift_Ms i (drop_Ms i ms))=ms.

Lemma lift_drop_M :
        ((m:M)(lift_drop_m m))/\((ms:Ms)(lift_drop_ms ms)).

Lemma Lift_Drop_M :
        (i:nat)(m:M)
         ~(Occurs_In_M i m)->
          (lift_M i (drop_M i m))=m.

Lemma Lift_Drop_Ms :
        (i:nat)(ms:Ms)
         ~(Occurs_In_Ms i ms)->
          (lift_Ms i (drop_Ms i ms))=ms.

Recursive Definition
    VsubstAV : A->V->V->A :=
        a i j => (Setifb A (nateqb i j) a (var (drop_V i j))).

Fixpoint
    NsubstAV1 [n:N]: A->nat->N :=
        [a:A][i:nat]<N>Case n of
            [n':N]
                (lam (NsubstAV1 n' (lift_A O a) (S i)))
            [a':A]
                (an (AsubstAV1 a' a i))
end with
    AsubstAV1 [a:A] : A->nat->A :=
        [a':A][i:nat]<A>Case a of
            [a'':A][n:N]
                (ap (AsubstAV1 a'' a' i) (NsubstAV1 n a' i))
            [x:V]
                (VsubstAV a' i x)
end.
```

```
Recursive Definition
    NsubstAV : A->V->N->N :=
        a i n => (NsubstAV1 n a i).

Recursive Definition
    AsubstAV : A->V->A->A :=
        a i a' => (AsubstAV1 a' a i).

Lemma NSAV1 : (a:A)(i:nat)(n:N)
                ((NsubstAV a i (lam n)) =
                 (lam (NsubstAV (lift_A O a) (S i) n))).

Lemma NSAV2 : (a:A)(i:nat)(a':A)
                ((NsubstAV a i (an a')) =
                 (an (AsubstAV a i a'))).

Lemma NSAV3 : (a:A)(i:nat)(a':A)(n:N)
                ((AsubstAV a i (ap a' n)) =
                 (ap (AsubstAV a i a') (NsubstAV a i n))).

Lemma NSAV4 : (a:A)(i:nat)(x:V)
                ((AsubstAV a i (var x)) =
                 (VsubstAV a i x)).

Recursive Definition
    phi : L -> N :=
        (vr x) => (an (var x)) |
        (app x l1 l2) =>
                (NsubstAV (ap (var x) (phi l1)) O (phi l2)) |
        (lm l) => (lam (phi l)).

Recursive Definition
    phibar : L->M :=
        (vr x) => (sc x mnil) |
        (app x l1 l2) =>
                (MsubstVMV x (phibar l1) O (phibar l2)) |
```



```
        (lm l) => (lambda (phibar l)).

Definition lift_psi_bridge : N->Prop :=
[n:N](i:nat)(lift_M i (psi n)) = (psi (lift_N i n)).

Definition lift_psi'_bridge : A->Prop :=
        [a:A](ms:Ms)(i:nat)(lift_M i (psi' a ms))=
         (psi' (lift_A i a) (lift_Ms i ms)).

Lemma Lift_psi_Bridge :
            ((n:N)(lift_psi_bridge n))/\
            ((a:A)(lift_psi'_bridge a)).

Lemma Lift_Psi_Bridge : (n:N)(i:nat)
        (lift_M i (psi n)) = (psi (lift_N i n)).

Lemma Lift_Psi'_Bridge : (a:A)(ms:Ms)(i:nat)
        (lift_M i (psi' a ms))=(psi' (lift_A i a) (lift_Ms i ms)).

Lemma Lift_Theta_Bridge :
        (m:M)(i:nat)(lift_N i (theta m)) = (theta (lift_M i m)).

Lemma Lift_Theta'_Bridge : (a:A)(ms:Ms)(i:nat)
(lift_N i (theta' a ms))=(theta' (lift_A i a) (lift_Ms i ms)).

Lemma Lift_Lift_M_Bridge :
        (m:M)(i,j:nat)
         (lt i j)->
          (lift_M i (lift_M j m))=
           (lift_M (S j) (lift_M i m)).

Lemma Lift_Lift_Ms_Bridge :
        (ms:Ms)(i,j:nat)
         (lt i j)->
          (lift_Ms i (lift_Ms j ms))=
           (lift_Ms (S j) (lift_Ms i ms)).
```

```
Lemma Lift_Lift_M_Bridge0 :
        (m:M)(i,j:nat)
         (lt j i)->
          (lift_M i (lift_M j m))=
           (lift_M j (lift_M (pred i) m)).

Lemma Lift_Lift_Ms_Bridge0 :
        (ms:Ms)(i,j:nat)
         (lt j i)->
          (lift_Ms i (lift_Ms j ms))=
           (lift_Ms j (lift_Ms (pred i) ms)).

Lemma Lift_Lift_M_Bridge1 :
        (m:M)(i,j:nat)
         i=j->
          (lift_M i (lift_M j m))=
           (lift_M (S j) (lift_M i m)).

Lemma Lift_Lift_Ms_Bridge1 :
        (ms:Ms)(i,j:nat)
         i=j->
          (lift_Ms i (lift_Ms j ms))=
           (lift_Ms (S j) (lift_Ms i ms)).

Definition msub_psi_bridge : N->Prop :=
        [n2:N](n1:N)(x,y:V)
         (MsubstVMV x (psi n1) y (psi n2))=
          (psi (NsubstAV (ap (var x) n1) y n2)).

Definition msub_psi'_bridge : A->Prop :=
        [a:A](x:V)(n:N)(y:V)(ms:Ms)
         (MsubstVMV x (psi n) y (psi' a ms))=
          (psi' (AsubstAV (ap (var x) n) y a) (MssubstVMV x (psi n) y ms)).

Lemma Msub_psi_bridge :
```



```
        ((n:N)(msub_psi_bridge n))/\((a:A)(msub_psi'_bridge a)).

Lemma Msub_Psi_Bridge :
        (n2:N)(n1:N)(x,y:V)
         (MsubstVMV x (psi n1) y (psi n2))=
          (psi (NsubstAV (ap (var x) n1) y n2)).

Lemma Msub_Psi'_Bridge :
        (a:A)(x:V)(n:N)(y:V)(ms:Ms)
         (MsubstVMV x (psi n) y (psi' a ms))=
          (psi' (AsubstAV (ap (var x) n) y a) (MssubstVMV x (psi n) y ms)).

Lemma Nsub_Theta_Bridge :
        (x:V)(m1:M)(y:V)(m2:M)
         (NsubstAV (ap (var x) (theta m1)) y (theta m2))
          =(theta (MsubstVMV x m1 y m2)).

Definition theta_drop_m_bridge : M->Prop :=
        [m:M](i:nat)(theta (drop_M i m))=(drop_N i (theta m)).

Definition theta'_drop_ms_bridge : Ms->Prop :=
        [ms:Ms](a:A)(i:nat)
        (theta' (drop_A i a) (drop_Ms i ms))=(drop_N i (theta' a ms)).

Lemma theta_drop_M_bridge :
        ((m:M)(theta_drop_m_bridge m))/\
         ((ms:Ms)(theta'_drop_ms_bridge ms)).

Lemma Theta_Drop_M_Bridge :
        (m:M)(i:nat)(theta (drop_M i m))=(drop_N i (theta m)).

Lemma Theta'_Drop_Ms_Bridge :
        (ms:Ms)(a:A)(i:nat)
         (theta' (drop_A i a) (drop_Ms i ms))=(drop_N i (theta' a ms)).

Lemma Psi_Drop_N_Bridge :
```

```
        (n:N)(i:nat)
         (psi (drop_N i n))=(drop_M i (psi n)).

Lemma Psi'_Drop_A_Bridge :
        (a:A)(ms:Ms)(i:nat)
         (psi' (drop_A i a) (drop_Ms i ms))=(drop_M i (psi' a ms)).

Lemma thetaphibarphi : (l:L)(theta (phibar l))=(phi l).

Lemma psiphiphibar : (l:L)(psi (phi l))=(phibar l).

Lemma OI_Lift_V1_1:
        (x:V)(i,j:nat)
         (lt i j)->
          (Occurs_In_V1 i (lift_V j x))=(Occurs_In_V1 i x).

Lemma OI_Lift_V1_2:
        (x:V)(i,j:nat)
         i=j->
          (Occurs_In_V1 i (lift_V j x))=false.

Lemma OI_Lift_V1_3 :
        (x:V)(i,j:nat)
         (lt j i)->
          (Occurs_In_V1 (S i) (lift_V j x))=
           (Occurs_In_V1 i x).

Lemma OI_Lift_V1_4 : (x:V)(i,j:nat)
        (lt j i)->
         (Occurs_In_V1 i (lift_V j x))=
          (Occurs_In_V1 (pred i) x).

Definition oi_lift_m1_1: M->Prop :=
        [m:M](i,j:nat)
         (lt i j)->(Occurs_In_M1 i (lift_M j m))=(Occurs_In_M1 i m).
```



```
Definition oi_lift_ms1_1: Ms->Prop :=
        [ms:Ms](i,j:nat)
         (lt i j)->
          (Occurs_In_Ms1 i (lift_Ms j ms))=(Occurs_In_Ms1 i ms).

Lemma oi_lift_M1_1 :
        ((m:M)(oi_lift_m1_1 m))/\
         ((ms:Ms)(oi_lift_ms1_1 ms)).

Lemma OI_Lift_M1_1:
        (m:M)(i,j:nat)
         (lt i j)->
          (Occurs_In_M1 i (lift_M j m))=(Occurs_In_M1 i m).

Lemma OI_Lift_Ms1_1:
        (ms:Ms)(i,j:nat)
         (lt i j)->
          (Occurs_In_Ms1 i (lift_Ms j ms))=(Occurs_In_Ms1 i ms).

Definition oi_lift_m1_2: M->Prop :=
        [m:M](i,j:nat)
         i=j->(Occurs_In_M1 i (lift_M j m))=false.

Definition oi_lift_ms1_2: Ms->Prop :=
        [ms:Ms](i,j:nat)
         i=j->(Occurs_In_Ms1 i (lift_Ms j ms))=false.

Lemma oi_lift_M1_2 :
        ((m:M)(oi_lift_m1_2 m))/\
         ((ms:Ms)(oi_lift_ms1_2 ms)).

Lemma OI_Lift_M1_2:
        (m:M)(i,j:nat)
         i=j->
          (Occurs_In_M1 i (lift_M j m))=false.
```

```
Lemma OI_Lift_Ms1_2:
        (ms:Ms)(i,j:nat)
         i=j->
          (Occurs_In_Ms1 i (lift_Ms j ms))=false.

Definition oi_lift_m1_3: M->Prop :=
        [m:M](i,j:nat)
         (lt j i)->
          (Occurs_In_M1 (S i) (lift_M j m))=(Occurs_In_M1 i m).

Definition oi_lift_ms1_3: Ms->Prop :=
        [ms:Ms](i,j:nat)
         (lt j i)->
          (Occurs_In_Ms1 (S i) (lift_Ms j ms))=(Occurs_In_Ms1 i ms).

Lemma oi_lift_M1_3 :
        ((m:M)(oi_lift_m1_3 m))/\
         ((ms:Ms)(oi_lift_ms1_3 ms)).

Lemma OI_Lift_M1_3:
        (m:M)(i,j:nat)
         (lt j i)->
          (Occurs_In_M1 (S i) (lift_M j m))=(Occurs_In_M1 i m).

Lemma OI_Lift_Ms1_3:
        (ms:Ms)(i,j:nat)
         (lt j i)->
          (Occurs_In_Ms1 (S i) (lift_Ms j ms))=(Occurs_In_Ms1 i ms).

Definition oi_lift_m1_4: M->Prop :=
        [m:M](i,j:nat)
         (lt j i)->
          (Occurs_In_M1 i (lift_M j m))=(Occurs_In_M1 (pred i) m).

Definition oi_lift_ms1_4: Ms->Prop :=
        [ms:Ms](i,j:nat)
```



```
         (lt j i)->
          (Occurs_In_Ms1 i (lift_Ms j ms))=
           (Occurs_In_Ms1 (pred i) ms).

Lemma oi_lift_M1_4 :
        ((m:M)(oi_lift_m1_4 m))/\
         ((ms:Ms)(oi_lift_ms1_4 ms)).

Lemma OI_Lift_M1_4 :
        (m:M)(i,j:nat)
         (lt j i)->
          (Occurs_In_M1 i (lift_M j m))=(Occurs_In_M1 (pred i) m).

Lemma OI_Lift_Ms1_4 :
        (ms:Ms)(i,j:nat)
         (lt j i)->
          (Occurs_In_Ms1 i (lift_Ms j ms))=
           (Occurs_In_Ms1 (pred i) ms).

Definition oi_lift_n1_1: N->Prop :=
        [n:N](i,j:nat)
         (lt i j)->(Occurs_In_N1 i (lift_N j n))=(Occurs_In_N1 i n).

Definition oi_lift_a1_1: A->Prop :=
        [a:A](i,j:nat)
         (lt i j)->(Occurs_In_A1 i (lift_A j a))=(Occurs_In_A1 i a).

Lemma oi_lift_N1_1 :
        ((n:N)(oi_lift_n1_1 n))/\
         ((a:A)(oi_lift_a1_1 a)).

Lemma OI_Lift_N1_1:
        (n:N)(i,j:nat)
         (lt i j)->
          (Occurs_In_N1 i (lift_N j n))=(Occurs_In_N1 i n).
```

```
Lemma OI_Lift_A1_1:
        (a:A)(i,j:nat)
         (lt i j)->
          (Occurs_In_A1 i (lift_A j a))=(Occurs_In_A1 i a).

Definition oi_lift_n1_2 : N->Prop :=
        [n:N](i,j:nat)
         i=j->(Occurs_In_N1 i (lift_N j n))=false.

Definition oi_lift_a1_2 : A->Prop :=
        [a:A](i,j:nat)
         i=j->(Occurs_In_A1 i (lift_A j a))=false.

Lemma oi_lift_N1_2 :
        ((n:N)(oi_lift_n1_2 n))/\
         ((a:A)(oi_lift_a1_2 a)).

Lemma OI_Lift_N1_2:
        (n:N)(i,j:nat)
         i=j->
          (Occurs_In_N1 i (lift_N j n))=false.

Lemma OI_Lift_A1_2:
        (a:A)(i,j:nat)
         i=j->
          (Occurs_In_A1 i (lift_A j a))=false.

Definition oi_lift_n1_3: N->Prop :=
        [n:N](i,j:nat)
         (lt j i)->
          (Occurs_In_N1 (S i) (lift_N j n))=(Occurs_In_N1 i n).

Definition oi_lift_a1_3: A->Prop :=
        [a:A](i,j:nat)
         (lt j i)->
          (Occurs_In_A1 (S i) (lift_A j a))=(Occurs_In_A1 i a).
```



```
Lemma oi_lift_N1_3 :
        ((n:N)(oi_lift_n1_3 n))/\
         ((a:A)(oi_lift_a1_3 a)).

Lemma OI_Lift_N1_3 :
        (n:N)(i,j:nat)
         (lt j i)->
          (Occurs_In_N1 (S i) (lift_N j n))=(Occurs_In_N1 i n).

Lemma OI_Lift_A1_3 :
        (a:A)(i,j:nat)
         (lt j i)->
          (Occurs_In_A1 (S i) (lift_A j a))=(Occurs_In_A1 i a).

Definition oi_lift_n1_4: N->Prop :=
        [n:N](i,j:nat)
         (lt j i)->
          (Occurs_In_N1 i (lift_N j n))=(Occurs_In_N1 (pred i) n).

Definition oi_lift_a1_4: A->Prop :=
        [a:A](i,j:nat)
         (lt j i)->
          (Occurs_In_A1 i (lift_A j a))=(Occurs_In_A1 (pred i) a).

Lemma oi_lift_N1_4 :
        ((n:N)(oi_lift_n1_4 n))/\
         ((a:A)(oi_lift_a1_4 a)).

Lemma OI_Lift_N1_4 :
        (n:N)(i,j:nat)
         (lt j i)->
          (Occurs_In_N1 i (lift_N j n))=(Occurs_In_N1 (pred i) n).

Lemma OI_Lift_A1_4 :
        (a:A)(i,j:nat)
```

```
        (lt j i)->
         (Occurs_In_A1 i (lift_A j a))=(Occurs_In_A1 (pred i) a).


Lemma OI_Lift_L1_4 :
        (l:L)(i,j:nat)
         (lt j i)->
          (Occurs_In_L1 i (lift_L j l))=
           (Occurs_In_L1 (pred i) l).

Definition noi_msub_b_bridge : M->Prop :=
        [m:M](x:V)(m1:M)(i:nat)
         ~(Occurs_In_M i m)->
          (MsubstVMV x m1 i m)=(drop_M i m).

Definition noi_mssub_b_bridge : Ms->Prop :=
        [ms:Ms](x:V)(m1:M)(i:nat)
         ~(Occurs_In_Ms i ms)->
          (MssubstVMV x m1 i ms)=(drop_Ms i ms).

Lemma noi_msub_b_Bridge :
        ((m:M)(noi_msub_b_bridge m))/\
         ((ms:Ms)(noi_mssub_b_bridge ms)).

Lemma NOI_Msub_Bridge :
        (m:M)(x:V)(m1:M)(i:nat)
         ~(Occurs_In_M i m)->
          (MsubstVMV x m1 i m)=(drop_M i m).

Lemma NOI_Mssub_Bridge :
        (ms:Ms)(x:V)(m1:M)(i:nat)
         ~(Occurs_In_Ms i ms)->
          (MssubstVMV x m1 i ms)=(drop_Ms i ms).

Lemma Lift_Drop_V_Bridge1 : (x:V)(i,j:nat)
        (lt j i)->
         ~(Occurs_In_V j x)->
```



```
          (lift_V i (drop_V j x))=
           (drop_V j (lift_V (S i) x)).

Definition lift_drop_n_bridge1 : N->Prop :=
        [n:N](i,j:nat)
         (lt j i)->
          ~(Occurs_In_N j n)->
           (lift_N i (drop_N j n))=
            (drop_N j (lift_N (S i) n)).

Definition lift_drop_a_bridge1 : A->Prop :=
        [a:A](i,j:nat)
         (lt j i)->
          ~(Occurs_In_A j a)->
           (lift_A i (drop_A j a))=
            (drop_A j (lift_A (S i) a)).

Lemma lift_drop_n_Bridge1 :
        ((n:N)(lift_drop_n_bridge1 n))/\((a:A)(lift_drop_a_bridge1 a)).

Lemma Lift_Drop_N_Bridge1 :
        (n:N)(i,j:nat)
         (lt j i)->
          ~(Occurs_In_N j n)->
           (lift_N i (drop_N j n))=
            (drop_N j (lift_N (S i) n)).

Lemma Lift_Drop_A_Bridge1 :
        (a:A)(i,j:nat)
         (lt j i)->
          ~(Occurs_In_A j a)->
           (lift_A i (drop_A j a))=
            (drop_A j (lift_A (S i) a)).

Lemma Drop_Lift_V_Bridge1 :
        (x:V)(i,j:nat)
```

```
       ~(Occurs_In_V i x)->
        (lt j (S i))->
         (drop_V (S i) (lift_V j x))=
          (lift_V j (drop_V i x)).

Definition drop_lift_n_bridge1 : N->Prop :=
       [n:N](i,j:nat)
        ~(Occurs_In_N i n)->
         (lt j (S i))->
          (drop_N (S i) (lift_N j n))=
           (lift_N j (drop_N i n)).

Definition drop_lift_a_bridge1 : A->Prop :=
       [a:A](i,j:nat)
        ~(Occurs_In_A i a)->
         (lt j (S i))->
          (drop_A (S i) (lift_A j a))=
           (lift_A j (drop_A i a)).

Lemma drop_lift_n_Bridge1 :
       ((n:N)(drop_lift_n_bridge1 n))/\
        ((a:A)(drop_lift_a_bridge1 a)).

Lemma Drop_Lift_N_Bridge1 :
       (n:N)(i,j:nat)
        ~(Occurs_In_N i n)->
         (lt j (S i))->
          (drop_N (S i) (lift_N j n))=
           (lift_N j (drop_N i n)).

Lemma Drop_Lift_A_Bridge1 :
       (a:A)(i,j:nat)
        ~(Occurs_In_A i a)->
         (lt j (S i))->
          (drop_A (S i) (lift_A j a))=
           (lift_A j (drop_A i a)).
```



```
Lemma Lift_Vsub_Bridge0 :
       (x:V)(i,j:nat)(a:A)
        (lt i j)->
         (lift_A j (VsubstAV a i x))=
          (VsubstAV (lift_A j a) i (lift_V (S j) x)).

Definition lift_nsub_b_bridge0 : N->Prop :=
       [n:N](i,j:nat)(a:A)
        (lt i j)->
         (lift_N j (NsubstAV a i n))=
          (NsubstAV (lift_A j a) i (lift_N (S j) n)).

Definition lift_asub_b_bridge0 : A->Prop :=
       [a:A](i,j:nat)(a1:A)
        (lt i j)->
         (lift_A j (AsubstAV a1 i a))=
          (AsubstAV (lift_A j a1) i (lift_A (S j) a)).

Lemma lift_nsub_b_Bridge0 :
       ((n:N)(lift_nsub_b_bridge0 n))/\ ((a:A)(lift_asub_b_bridge0 a)).

Lemma Lift_Nsub_Bridge0 :
       (n:N)(i,j:nat)(a:A)
        (lt i j)->
         (lift_N j (NsubstAV a i n))=
          (NsubstAV (lift_A j a) i (lift_N (S j) n)).

Lemma Lift_Asub_Bridge0 :
       (a:A)(i,j:nat)(a1:A)
        (lt i j)->
         (lift_A j (AsubstAV a1 i a))=
          (AsubstAV (lift_A j a1) i (lift_A (S j) a)).

Lemma Lift_Msub_Bridge0 :
       (x:V)(m,m0:M)(i,j:nat)
```

```
    (lt i j)->
     (lift_M j (MsubstVMV x m i m0))=
      (MsubstVMV (lift_V j x) (lift_M j m) i (lift_M (S j) m0)).

Lemma Lift_Mssub_Bridge0 :
    (x:V)(m:M)(ms:Ms)(i,j:nat)
     (lt i j)->
      (lift_Ms j (MssubstVMV x m i ms))=
       (MssubstVMV (lift_V j x) (lift_M j m) i (lift_Ms (S j) ms)).

Definition lift_msub_bridge2 : M->Prop :=
    [m:M](x,y:V)(m1:M)
     (MsubstVMV x m1 (S y) (lift_M y (lift_M (S y) m)))=
      (MsubstVMV x m1 y (lift_M (S (S y)) (lift_M (S y) m))).

Definition lift_mssub_bridge2 : Ms->Prop :=
    [ms:Ms](x,y:V)(m1:M)
     (MssubstVMV x m1 (S y) (lift_Ms y (lift_Ms (S y) ms)))=
      (MssubstVMV x m1 y (lift_Ms (S (S y)) (lift_Ms (S y) ms))).

Lemma Lift_msub_bridge2 :
    ((m:M)(lift_msub_bridge2 m))/\
     ((ms:Ms)(lift_mssub_bridge2 ms)).

Lemma Lift_Msub_Bridge2 :
    (x:V)(m1:M)(y:V)(m:M)
     (MsubstVMV x m1 (S y) (lift_M y (lift_M (S y) m)))=
      (MsubstVMV x m1 y (lift_M (S (S y)) (lift_M (S y) m))).

Lemma Lift_Mssub_Bridge2 :
    (x:V)(m1:M)(y:V)(ms:Ms)
     (MssubstVMV x m1 (S y) (lift_Ms y (lift_Ms (S y) ms)))=
      (MssubstVMV x m1 y (lift_Ms (S (S y)) (lift_Ms (S y) ms))).

Lemma Lift_Vsub_Bridge1 :
    (x:V)(i,j:nat)(a:A)
```



```
     i=j->
      (lift_A j (VsubstAV a i x))=
       (VsubstAV (lift_A j a) i (lift_V (S j) x)).

Lemma Lift_Vsub_Bridge2 :
    (x:V)(i,j:nat)(a:A)
     (lt j i)->
      (lift_A j (VsubstAV a i x))=
       (VsubstAV (lift_A j a) (S i) (lift_V j x)).

Definition lift_nsub_b_bridge1 : N->Prop :=
    [n:N](i,j:nat)(a:A)
     i=j->
      (lift_N j (NsubstAV a i n))=
       (NsubstAV (lift_A j a) i (lift_N (S j) n)).

Definition lift_asub_b_bridge1 : A->Prop :=
    [a:A](i,j:nat)(a1:A)
     i=j->
      (lift_A j (AsubstAV a1 i a))=
       (AsubstAV (lift_A j a1) i (lift_A (S j) a)).

Lemma lift_nsub_b_Bridge1 :
    ((n:N)(lift_nsub_b_bridge1 n))/\ ((a:A)(lift_asub_b_bridge1 a)).

Lemma Lift_Nsub_Bridge1 :
    (n:N)(i,j:nat)(a:A)
     i=j->
      (lift_N j (NsubstAV a i n))=
       (NsubstAV (lift_A j a) i (lift_N (S j) n)).

Lemma Lift_Asub_Bridge1 :
    (a:A)(i,j:nat)(a1:A)
     i=j->
      (lift_A j (AsubstAV a1 i a))=
       (AsubstAV (lift_A j a1) i (lift_A (S j) a)).
```

```
Definition lift_nsub_b_bridge2 : N->Prop :=
        [n:N](i,j:nat)(a:A)
         (lt j i)->
          (lift_N j (NsubstAV a i n))=
           (NsubstAV (lift_A j a) (S i) (lift_N j n)).

Definition lift_asub_b_bridge2 : A->Prop :=
        [a:A](i,j:nat)(a1:A)
         (lt j i)->
          (lift_A j (AsubstAV a1 i a))=
           (AsubstAV (lift_A j a1) (S i) (lift_A j a)).

Lemma lift_nsub_b_Bridge2 :
        ((n:N)(lift_nsub_b_bridge2 n))/\ ((a:A)(lift_asub_b_bridge2 a)).

Lemma Lift_Nsub_Bridge2 :
        (n:N)(i,j:nat)(a:A)
         (lt j i)->
          (lift_N j (NsubstAV a i n))=
           (NsubstAV (lift_A j a) (S i) (lift_N j n)).

Lemma Lift_Asub_Bridge2 :
        (a:A)(i,j:nat)(a1:A)
         (lt j i)->
          (lift_A j (AsubstAV a1 i a))=
           (AsubstAV (lift_A j a1) (S i) (lift_A j a)).

Lemma Lift_Vsub_Bridge3 :
        (x:V)(i,j:nat)(a:A)
         i=j->
          (lift_A j (VsubstAV a i x))=
           (VsubstAV (lift_A j a) (S i) (lift_V j x)).

Definition lift_nsub_b_bridge3 : N->Prop :=
        [n:N](i,j:nat)(a:A)
```



```
         i=j->
          (lift_N j (NsubstAV a i n))=
           (NsubstAV (lift_A j a) (S i) (lift_N j n)).

Definition lift_asub_b_bridge3 : A->Prop :=
        [a:A](i,j:nat)(a1:A)
         i=j->
          (lift_A j (AsubstAV a1 i a))=
           (AsubstAV (lift_A j a1) (S i) (lift_A j a)).

Lemma lift_nsub_b_Bridge3 :
        ((n:N)(lift_nsub_b_bridge3 n))/\ ((a:A)(lift_asub_b_bridge3 a)).

Lemma Lift_Nsub_Bridge3 :
        (n:N)(i,j:nat)(a:A)
         i=j->
          (lift_N j (NsubstAV a i n))=
           (NsubstAV (lift_A j a) (S i) (lift_N j n)).

Lemma Lift_Asub_Bridge3 :
        (a:A)(i,j:nat)(a1:A)
         i=j->
          (lift_A j (AsubstAV a1 i a))=
           (AsubstAV (lift_A j a1) (S i) (lift_A j a)).

Lemma Lift_Msub_Bridge1 :
        (x:V)(m,m0:M)(i,j:nat)
         i=j->
          (lift_M j (MsubstVMV x m i m0))=
           (MsubstVMV (lift_V j x) (lift_M j m) i (lift_M (S j) m0)).

Lemma Lift_Mssub_Bridge1 :
        (x:V)(m:M)(ms:Ms)(i,j:nat)
         i=j->
          (lift_Ms j (MssubstVMV x m i ms))=
           (MssubstVMV (lift_V j x) (lift_M j m) i (lift_Ms (S j) ms)).
```

```
Lemma Lift_Phi_Bridge :
        (l:L)(i:nat)
         (lift_N i (phi l))=
          (phi (lift_L i l)).

Lemma Lift_PhiBar_Bridge :
        (l:L)(i:nat)
         (lift_M i (phibar l))=
          (phibar (lift_L i l)).

Lemma Drop_Lift_L :
        (l:L)(x:V)
         (drop_L x (lift_L x l))=l.

Definition oi_theta : M->Prop :=
        [m:M](x:V)
         (Occurs_In_M x m)->
          (Occurs_In_N x (theta m)).

Definition oi_theta' : Ms->Prop :=
        [ms:Ms](a:A)(x:V)
         ((Occurs_In_Ms x ms)\/(Occurs_In_A x a))->
          (Occurs_In_N x (theta' a ms)).

Lemma OI_theta :
        ((m:M)(oi_theta m))/\
         ((ms:Ms)(oi_theta' ms)).

Lemma OI_Theta :
        (m:M)(x:V)
         (Occurs_In_M x m)->
          (Occurs_In_N x (theta m)).

Lemma OI_Theta' :
        (ms:Ms)(a:A)(x:V)
```



```
         ((Occurs_In_Ms x ms)\/(Occurs_In_A x a))->
          (Occurs_In_N x (theta' a ms)).

Definition oi_psi : N->Prop :=
        [n:N](x:V)
         (Occurs_In_N x n)->
          (Occurs_In_M x (psi n)).

Definition oi_psi' : A->Prop :=
        [a:A](ms:Ms)(x:V)
         ((Occurs_In_A x a)\/(Occurs_In_Ms x ms))->
          (Occurs_In_M x (psi' a ms)).

Lemma OI_psi :
        ((n:N)(oi_psi n))/\
         ((a:A)(oi_psi' a)).

Lemma OI_Psi :
        (n:N)(x:V)
         (Occurs_In_N x n)->
          (Occurs_In_M x (psi n)).

Lemma OI_Psi' :
        (a:A)(ms:Ms)(x:V)
         ((Occurs_In_A x a)\/(Occurs_In_Ms x ms))->
          (Occurs_In_M x (psi' a ms)).

Definition noi_theta : M->Prop :=
        [m:M](x:V)
         ~(Occurs_In_M x m)->
          ~(Occurs_In_N x (theta m)).

Definition noi_theta' : Ms->Prop :=
        [ms:Ms](x:V)(a:A)
         ~(Occurs_In_Ms x ms)->
          ~(Occurs_In_A x a)->
```

```
    ~(Occurs_In_N x (theta' a ms)).

Lemma noi_Theta :
        ((m:M)(noi_theta m))/\
         ((ms:Ms)(noi_theta' ms)).

Lemma NOI_Theta :
        (x:V)(m:M)
        ~(Occurs_In_M x m)->
          ~(Occurs_In_N x (theta m)).

Lemma NOI_Theta' :
        (x:V)(a:A)(ms:Ms)
         ~(Occurs_In_A x a)->
          ~(Occurs_In_Ms x ms)->
           ~(Occurs_In_N x (theta' a ms)).

Recursive Definition
    Height_L : L->nat :=
        (vr x) => O |
        (app x l1 l2) => (S (max_nat (Height_L l1) (Height_L l2))) |
        (lm l) => (S (Height_L l)).

Definition
    lt_Height_L : L->L->Prop :=
        [l1,l2:L](lt (Height_L l1) (Height_L l2)).

Lemma WF_Height_L :
        (well_founded L lt_Height_L).

Lemma L_Height_ind1 :
        (P:L->Prop)
         ((l0:L)((l1:L)(lt_Height_L l1 l0)->(P l1))->(P l0))->
          (l:L)(P l).

Lemma L_Height_ind :
```



```
        (P:L->Prop)
         ((l0:L)((l1:L)(lt (Height_L l1) (Height_L l0))->(P l1))->(P l0))->
          (l:L)(P l).

Lemma Height_Lift_L :
        (l:L)(i:nat)
         (Height_L (lift_L i l))=
          (Height_L l).

Fixpoint
    Height_M [m:M] : nat :=
        <nat>Case m of
        [x:V][ms:Ms](S (Height_Ms ms))
        [m:M](S (Height_M m))
        end with
    Height_Ms [ms:Ms] : nat :=
        <nat>Case ms of
        O
        [m:M][ms:Ms](S (max_nat (Height_M m) (Height_Ms ms)))
        end.

Lemma HTM1 : (x:V)(ms:Ms)
        (Height_M (sc x ms))=(S (Height_Ms ms)).

Lemma HTM2 : (m:M)
        (Height_M (lambda m))=(S (Height_M m)).

Lemma HTM3 :
        (Height_Ms mnil)=O.

Lemma HTM4 : (m:M)(ms:Ms)
        (Height_Ms (mcons m ms))=(S (max_nat (Height_M m) (Height_Ms ms))).

Lemma Height_Ms_Zero_Nil1 : (ms:Ms)
        ~ms=mnil->
         (lt O (Height_Ms ms)).
```

```
Lemma Height_Ms_Zero_Nil : (ms:Ms)
        (Height_Ms ms)=O->
         ms=mnil.

Lemma Height_M_not_eq_not_eq :
        (m:M)(m0:M)
         ~(Height_M m)=(Height_M m0)->
          ~m=m0.

Lemma Height_Ms_not_eq_not_eq :
        (ms:Ms)(ms0:Ms)
         ~(Height_Ms ms)=(Height_Ms ms0)->
          ~ms=ms0.

Definition height_lift_m : M->Prop :=
        [m:M](i:nat)
         (Height_M (lift_M i m))=
          (Height_M m).

Definition height_lift_ms : Ms->Prop :=
        [ms:Ms](i:nat)
         (Height_Ms (lift_Ms i ms))=
          (Height_Ms ms).

Lemma height_lift_M :
        ((m:M)(height_lift_m m))/\
         ((ms:Ms)(height_lift_ms ms)).

Lemma Height_Lift_M :
        (m:M)(i:nat)
         (Height_M (lift_M i m))=
          (Height_M m).

Lemma Height_Lift_Ms :
        (ms:Ms)(i:nat)
```



```
         (Height_Ms (lift_Ms i ms))=
          (Height_Ms ms).

Section HeightMind.

Variable P:M->Prop.
Variable P0:Ms->Prop.

Definition QSM : M->Prop :=
        [m:M]
         ((m1:M)
           ((lt (Height_M m1) (Height_M m)) \/
             (Height_M m1)=(Height_M m))->
            (P m1))/\
         ((ms1:Ms)
           ((lt (Height_Ms ms1) (Height_M m)) \/
             (Height_Ms ms1)=(Height_M m))->
            (P0 ms1)).

Definition QSMs : Ms->Prop :=
        [ms:Ms]
         ((m1:M)
           ((lt (Height_M m1) (Height_Ms ms)) \/
             (Height_M m1)=(Height_Ms ms))->
            (P m1))/\
         ((ms1:Ms)
           ((lt (Height_Ms ms1) (Height_Ms ms)) \/
             (Height_Ms ms1)=(Height_Ms ms))->
            (P0 ms1)).

Lemma M_Ms_szind1 :
        (((m:M)(QSM m))/\((ms:Ms)(QSMs ms)))->
         ((m:M)(P m))/\((ms:Ms)(P0 ms)).

Lemma M_Ms_Height_ind :
   ((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)).

Recursive Definition
    lifts_V : nat->nat->V->V :=
        O j x => x |
        (S i) j x => (lift_V j (lifts_V i j x)).

Recursive Definition
    lifts_L : nat->nat->L->L :=
        i j (vr x) => (vr (lifts_V i j x)) |
        i j (app x l l0) =>
                (app (lifts_V i j x)
                (lifts_L i j l)
                (lifts_L i (S j) l0)) |
        i j (lm l) => (lm (lifts_L i (S j) l)).

Fixpoint
    lifts_M1 [m:M] : nat->nat->M :=
        [i,j:nat]
        <M>Case m of
            [x:V][ms:Ms]
                (sc (lifts_V i j x) (lifts_Ms1 ms i j))
            [m:M]
                (lambda (lifts_M1 m i (S j)))
        end
with
    lifts_Ms1 [ms:Ms] : nat->nat->Ms :=
        [i,j:nat]
        <Ms>Case ms of
            mnil
            [m:M][ms:Ms]
```



```
                (mcons (lifts_M1 m i j) (lifts_Ms1 ms i j))
        end.

Recursive Definition lifts_M : nat->nat->M->M :=
i j m => (lifts_M1 m i j).

Recursive Definition lifts_Ms : nat->nat->Ms->Ms :=
i j ms => (lifts_Ms1 ms i j).

Lemma LIFTSM1 : (i,j:nat)(x:V)(ms:Ms)
        (lifts_M i j (sc x ms))=
         (sc (lifts_V i j x) (lifts_Ms i j ms)).

Lemma LIFTSM2 : (i,j:nat)(m:M)
        (lifts_M i j (lambda m))=
         (lambda (lifts_M i (S j) m)).

Lemma LIFTSM3 : (i,j:nat)
        (lifts_Ms i j mnil)=mnil.

Lemma LIFTSM4 : (i,j:nat)(m:M)(ms:Ms)
        (lifts_Ms i j (mcons m ms))=
         (mcons (lifts_M i j m) (lifts_Ms i j ms)).

Lemma Lifts_L0 : (l:L)(j:nat)
        (lifts_L O j l)=l.

Definition lifts_m0 : M->Prop :=
        [m:M](j:nat)
         (lifts_M O j m)=m.

Definition lifts_ms0 : Ms->Prop :=
        [ms:Ms](j:nat)
         (lifts_Ms O j ms)=ms.

Lemma Lifts_m0 :
```

```
        ((m:M)(lifts_m0 m))/\
         ((ms:Ms)(lifts_ms0 ms)).

Lemma Lifts_M0 :
        (m:M)(j:nat)
         (lifts_M O j m)=m.

Lemma Lifts_Ms0 :
        (ms:Ms)(j:nat)
         (lifts_Ms O j ms)=ms.

Lemma Lifts_L_rec1 : (l:L)(i,j:nat)
        (lifts_L (S i) j l)=(lift_L j (lifts_L i j l)).

Definition lifts_m_rec1 : M->Prop :=
        [m:M](i,j:nat)
         (lifts_M (S i) j m)=(lift_M j (lifts_M i j m)).

Definition lifts_ms_rec1 : Ms->Prop :=
        [ms:Ms](i,j:nat)
         (lifts_Ms (S i) j ms)=(lift_Ms j (lifts_Ms i j ms)).

Lemma Lifts_m_rec1 :
        ((m:M)(lifts_m_rec1 m))/\
         ((ms:Ms)(lifts_ms_rec1 ms)).

Lemma Lifts_M_rec1 :
        (m:M)(i,j:nat)
         (lifts_M (S i) j m)=(lift_M j (lifts_M i j m)).

Lemma Lifts_Ms_rec1 :
        (ms:Ms)(i,j:nat)
         (lifts_Ms (S i) j ms)=(lift_Ms j (lifts_Ms i j ms)).

Lemma Lifts_V_rec2 :
        (x:V)(i,j:nat)
```



```
         (lifts_V (S i) j x)=
          (lifts_V i j (lift_V j x)).

Lemma Lifts_L_rec2 :
        (l:L)(i,j:nat)
         (lifts_L (S i) j l)=
          (lifts_L i j (lift_L j l)).

Definition lifts_m_rec2 : M->Prop :=
        [m:M](i,j:nat)
         (lifts_M (S i) j m)=(lifts_M i j (lift_M j m)).

Definition lifts_ms_rec2 : Ms->Prop :=
        [ms:Ms](i,j:nat)
         (lifts_Ms (S i) j ms)=(lifts_Ms i j (lift_Ms j ms)).

Lemma Lifts_m_rec2 :
        ((m:M)(lifts_m_rec2 m))/\
         ((ms:Ms)(lifts_ms_rec2 ms)).

Lemma Lifts_M_rec2 :
        (m:M)(i,j:nat)
         (lifts_M (S i) j m)=(lifts_M i j (lift_M j m)).

Lemma Lifts_Ms_rec2 :
        (ms:Ms)(i,j:nat)
         (lifts_Ms (S i) j ms)=(lifts_Ms i j (lift_Ms j ms)).

Lemma Lifts_Msub_Bridge0 :
        (k:nat)(x:V)(m,m0:M)(i,j:nat)
         (lt i j)->
          (lifts_M k j (MsubstVMV x m0 i m))=
           (MsubstVMV (lifts_V k j x)
                      (lifts_M k j m0) i
                      (lifts_M k (S j) m)).
```

```
Lemma Lifts_Mssub_Bridge0 :
        (k:nat)(x:V)(ms:Ms)(m0:M)(i,j:nat)
         (lt i j)->
          (lifts_Ms k j (MssubstVMV x m0 i ms))=
           (MssubstVMV (lifts_V k j x)
                       (lifts_M k j m0)
                       i
                       (lifts_Ms k (S j) ms)).

Lemma Lifts_Msub_Bridge1 :
        (k:nat)(x:V)(m,m0:M)(i,j:nat)
         i=j->
          (lifts_M k j (MsubstVMV x m0 i m))=
           (MsubstVMV (lifts_V k j x)
                      (lifts_M k j m0) i
                      (lifts_M k (S j) m)).

Lemma Lifts_Mssub_Bridge1 :
        (k:nat)(x:V)(ms:Ms)(m0:M)(i,j:nat)
         i=j->
          (lifts_Ms k j (MssubstVMV x m0 i ms))=
           (MssubstVMV (lifts_V k j x)
                       (lifts_M k j m0)
                       i
                       (lifts_Ms k (S j) ms)).

Lemma Lifts_PhiBar_Bridge : (l:L)(i,j:nat)
        (lifts_M i j (phibar l))=
         (phibar (lifts_L i j l)).

Lemma Lifts_Lift_V_Bridge3 :
        (x:V)(i,j,k:nat)
         ~(lt k j)->
          (lifts_V i j (lift_V k x))=
           (lift_V (plus i k) (lifts_V i j x)).
```



```
Lemma Lifts_Lift_L_Bridge3 :
        (l:L)(i,j,k:nat)
         ~(lt k j)->
          (lifts_L i j (lift_L k l))=
           (lift_L (plus i k) (lifts_L i j l)).

Lemma Lifts_Lifts_L_Bridge0 :
        (i,j,k,n:nat)(l:L)
         k=n->
          (lifts_L i k (lifts_L j n l))=
           (lifts_L (plus i j) n l).


Fixpoint
    rhobar [m:M] : L :=
        <L>Case m of
            [x:V][ms:Ms]
                <L>Case ms of
                        (vr x)
                    [m:M][ms:Ms]
                        (app x (rhobar m) (rhobar' ms (S O)))
                end
            [m:M]
                (lm (rhobar m))
        end
with
    rhobar' [ms:Ms] : nat->L :=
        [i:nat]
        <L>Case ms of
                (vr O)
            [m:M][ms:Ms]
                (app O (lifts_L i O (rhobar m)) (rhobar' ms (S i)))
        end.

Recursive Definition rhobar1 : nat->Ms->L :=
i ms => (rhobar' ms i).
```

```
Recursive Definition rho : N->L :=
n => (rhobar (psi n)).

Lemma rhothetarhobar : (m:M)
        (rho (theta m))=(rhobar m).

Lemma Rhobar1 : (x:V)
        (rhobar (sc x mnil))=(vr x).

Lemma Rhobar2 : (x:V)(m:M)
        (rhobar (sc x (mcons m mnil)))=
         (app x (rhobar m) (vr O)).

Lemma Rhobar3 : (x:V)(m:M)(ms:Ms)
        (rhobar (sc x (mcons m ms)))=
         (app x (rhobar m) (rhobar1 (S O) ms)).

Lemma Rhobar4 : (m:M)
        (rhobar (lambda m))=(lm (rhobar m)).

Lemma Rhobar5 : (i:nat)
        (rhobar1 i mnil)=(vr O).

Lemma Rhobar6 : (m:M)(ms:Ms)(i:nat)
        (rhobar1 i (mcons m ms))=
         (app O
              (lifts_L i O (rhobar m))
              (rhobar1 (S i) ms)).

Definition phibarrhobar1 : M->Prop :=
        [m:M]
         (phibar (rhobar m))=m.

Definition phibarrhobar2 : Ms->Prop :=
        [ms:Ms]
         (m:M)(i:nat)
```



```
          ((m1:M)
            (lt (Height_M m1) (Height_Ms (mcons m ms)))->
             (phibar (rhobar m1))=m1)->
           (phibar (rhobar1 i (mcons m ms)))=
            (sc O (lifts_Ms i O (mcons m ms))).

Lemma Phibarrhobar :
        ((m:M)(phibarrhobar1 m))/\
         ((ms:Ms)(phibarrhobar2 ms)).

Lemma phibarrhobar :
        (m:M)(phibar (rhobar m))=m.

Lemma phibarrhobar_ms :
        (i:nat)(m:M)(ms:Ms)
         (phibar (rhobar1 i (mcons m ms)))=
            (sc O (lifts_Ms i O (mcons m ms))).

Lemma phirho : (n:N)(phi (rho n))=n.

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)).
```

```
Scheme L_Deriv_ind1 := Induction for L_Deriv Sort Prop.

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).

Scheme M_Ms_Deriv_ind1 := Induction for M_Deriv Sort Prop
  with Ms_M_Deriv_ind1 := Induction for Ms_Deriv Sort Prop.

Lemma M_Ms_Deriv_ind :
        (P:(h:Hyps)(m:M)(f:F)(M_Deriv h m f)->Prop)
        (P0:(h:Hyps)(f:F)(m:Ms)(f0:F)(Ms_Deriv h f m f0)->Prop)
         ((h:Hyps)(i:V)(P1:F)(ms:Ms)(R:F)
          (i0:(In_Hyps i P1 h))
           (m:(Ms_Deriv h P1 ms R))
            (P0 h P1 ms R m)->
             (P h (sc i ms) R (Choose h i P1 ms R i0 m)))->
```



```
         ((h:Hyps)(P1:F)(m:M)(Q:F)
          (m0:(M_Deriv (Add_Hyp P1 h) m Q))
           (P (Add_Hyp P1 h) m Q m0)->
            (P h (lambda m) (Impl P1 Q) (Abstract h P1 m Q m0)))->
         ((h:Hyps)(P1:F)(P0 h P1 mnil P1 (Meet h P1)))->
         ((h:Hyps)(m:M)(P1,Q:F)(ms:Ms)(R:F)
          (m0:(M_Deriv h m P1))
           (P h m P1 m0)->
            (m1:(Ms_Deriv h Q ms R))
             (P0 h Q ms R m1)->
              (P0 h (Impl P1 Q) (mcons m ms) R
                (Implies_S h m P1 Q ms R m0 m1)))->
          ((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)
               (m0:(Ms_Deriv h f m f0))(P0 h f m f0 m0)).

Mutual Inductive
    N_Deduc : Hyps -> N -> F -> Prop :=
        Implies_I :
                (h:Hyps)(P:F)(n:N)(Q:F)
                 (N_Deduc (Add_Hyp P h) n Q)->
                  (N_Deduc h (lam n) (Impl P Q)) |
        AN_Axiom :
                (h:Hyps)(a:A)(P:F)
                 (A_Deduc h a P)->
                  (N_Deduc h (an a) P)
with
    A_Deduc : Hyps -> A -> F -> Prop :=
        Implies_E :
                (h:Hyps)(a:A)(P:F)(Q:F)(n:N)
                 (A_Deduc h a (Impl P Q))->
                  (N_Deduc h n P)->
                   (A_Deduc h (ap a n) Q) |
        A_Axiom :
                (h:Hyps)(i:V)(P:F)
                 (In_Hyps i P h)->
```

```
              (A_Deduc h (var i) P).

Scheme N_A_Deduc_ind1 := Induction for N_Deduc Sort Prop
  with A_N_Deduc_ind1 := Induction for A_Deduc Sort Prop.

Lemma N_A_Deduc_ind
     : (P:(h:Hyps)(n:N)(f:F)(N_Deduc h n f)->Prop)
        (P0:(h:Hyps)(a:A)(f:F)(A_Deduc h a f)->Prop)
         ((h:Hyps)
          (P1:F)
           (n:N)
            (Q:F)
             (n0:(N_Deduc (Add_Hyp P1 h) n Q))
              (P (Add_Hyp P1 h) n Q n0)
               ->(P h (lam n) (Impl P1 Q) (Implies_I h P1 n Q n0)))
        ->((h:Hyps)
            (a:A)
             (P1:F)
              (a0:(A_Deduc h a P1))
               (P0 h a P1 a0)->(P h (an a) P1 (AN_Axiom h a P1 a0)))
           ->((h:Hyps)
               (a:A)
                (P1,Q:F)
                 (n:N)
                  (a0:(A_Deduc h a (Impl P1 Q)))
                   (P0 h a (Impl P1 Q) a0)
                    ->(n0:(N_Deduc h n P1))
                       (P h n P1 n0)
                        ->(P0 h (ap a n) Q
                            (Implies_E h a P1 Q n a0 n0)))
              ->((h:Hyps)
                  (i:V)
                   (P1:F)
                    (i0:(In_Hyps i P1 h))
                     (P0 h (var i) P1 (A_Axiom h i P1 i0)))
                 ->((h:Hyps)(n:N)(f:F)(n0:(N_Deduc h n f))(P h n f n0))/\
```



```
                  ((h:Hyps)(a:A)(f:F)(a0:(A_Deduc h a f))(P0 h a f a0)).

Definition m_admis_psi :
        (h:Hyps)(n:N)(R:F)(N_Deduc h n R)-> Prop :=
         [h:Hyps][n:N][R:F][prf:(N_Deduc h n R)](M_Deriv h (psi n) R).

Definition m_admis_psi' :
        (h:Hyps)(a:A)(P:F)(A_Deduc h a P)->Prop :=
         [h:Hyps][a:A][P:F][prf:(A_Deduc h a P)]
          (R:F)(ms:Ms)((Ms_Deriv h P ms R) -> (M_Deriv h (psi' a ms) R)).

Lemma M_admis_psi :
        ((h:Hyps)(n:N)(R:F)(prf:(N_Deduc h n R))(m_admis_psi h n R prf))/\
         ((h:Hyps)(a:A)(R:F)(prf:(A_Deduc h a R))(m_admis_psi' h a R prf)).

Lemma M_Admis_Psi :
        (h:Hyps)(n:N)(R:F)
         (N_Deduc h n R)->
          (M_Deriv h (psi n) R).

Lemma M_Admis_Psi' :
        (h:Hyps)(a:A)(ms:Ms)(R:F)(P:F)
         (A_Deduc h a P)->
          (Ms_Deriv h P ms R)->
           (M_Deriv h (psi' a ms) R).

Definition n_admis_theta : (h:Hyps)(m:M)(P:F)(M_Deriv h m P)->Prop :=
[h:Hyps][m:M][R:F][prf:(M_Deriv h m R)](N_Deduc h (theta m) R).

Definition n_admis_theta' :
        (h:Hyps)(P:F)(ms:Ms)(R:F)(Ms_Deriv h P ms R)->Prop :=
         [h:Hyps][P:F][ms:Ms][R:F][prf:(Ms_Deriv h P ms R)]
          (a:A)((A_Deduc h a P) -> (N_Deduc h (theta' a ms) R)).

Lemma N_admis_theta :
        ((h:Hyps)(m:M)(P:F)
```

```
    (prf:(M_Deriv h m P))(n_admis_theta h m P prf)) /\
     ((h:Hyps)(P:F)(ms:Ms)(R:F)(prf:(Ms_Deriv h P ms R))
           (n_admis_theta' h P ms R prf)).

Lemma N_Admis_Theta :
    (h:Hyps)(m:M)(R:F)
     (M_Deriv h m R)->
      (N_Deduc h (theta m) R).

Lemma N_Admis_Theta' :
    (h:Hyps)(P:F)(ms:Ms)(R:F)
     (Ms_Deriv h P ms R)->
     ((a:A)((A_Deduc h a P)->
      (N_Deduc h (theta' a ms) R))).

Recursive Definition
  Weaken_Hyps : nat->F->Hyps->Hyps :=
    O P h => (Add_Hyp P h) |
    (S n) P MT => MT |
    (S n) P (Add_Hyp Q h) => (Add_Hyp Q (Weaken_Hyps n P h)).

Lemma In_Weaken_Hyps :
    (i,j:nat)(h:Hyps)(P,Q:F)
     (lt j (S (Len_Hyps h)))->
      (In_Hyps i P h)->
       (In_Hyps (lift_V j i) P (Weaken_Hyps j Q h)).

Definition n_admis_weaken :
    (h:Hyps)(n:N)(P:F)(N_Deduc h n P)->Prop :=
     [h:Hyps][n:N][P:F][D:(N_Deduc h n P)]
      (j:nat)(Q:F)
       (lt j (S (Len_Hyps h)))->
        (N_Deduc (Weaken_Hyps j Q h) (lift_N j n) P).

Definition a_admis_weaken :
    (h:Hyps)(a:A)(P:F)(A_Deduc h a P)->Prop :=
```



```
     [h:Hyps][a:A][P:F][D:(A_Deduc h a P)]
      (j:nat)(Q:F)
       (lt j (S (Len_Hyps h)))->
        (A_Deduc (Weaken_Hyps j Q h) (lift_A j a) P).

Lemma N_admis_weaken :
    ((h:Hyps)(n:N)(R:F)(n0:(N_Deduc h n R))(n_admis_weaken h n R n0))/\
     ((h:Hyps)(a:A)(R:F)(a0:(A_Deduc h a R))(a_admis_weaken h a R a0)).

Lemma N_Admis_Weaken :
    (h:Hyps)(n:N)(P:F)(j:nat)(Q:F)
     (N_Deduc h n P)->
      (lt j (S (Len_Hyps h)))->
       (N_Deduc (Weaken_Hyps j Q h) (lift_N j n) P).

Lemma A_Admis_Weaken :
    (h:Hyps)(a:A)(P:F)(j:nat)(Q:F)
     (A_Deduc h a P)->
      (lt j (S (Len_Hyps h)))->
       (A_Deduc (Weaken_Hyps j Q h) (lift_A j a) P).

Definition l_admis_weaken :
    (h:Hyps)(l:L)(P:F)(L_Deriv h l P)->Prop :=
     [h:Hyps][l:L][P:F][D:(L_Deriv h l P)]
      (j:nat)(Q:F)
       (lt j (S (Len_Hyps h)))->
        (L_Deriv (Weaken_Hyps j Q h) (lift_L j l) P).

Lemma L_admis_weaken :
    (h:Hyps)(l:L)(P:F)(D:(L_Deriv h l P))
     (l_admis_weaken h l P D).

Lemma L_Admis_Weaken :
    (h:Hyps)(l:L)(P,Q:F)(j:nat)
     (L_Deriv h l P)->
      (lt j (S (Len_Hyps h)))->
```

```
        (L_Deriv (Weaken_Hyps j Q h) (lift_L j 1) P).

Recursive Definition
    Strengthen_Hyps : nat->Hyps->Hyps :=
        n MT => MT |
        O (Add_Hyp Q h) => h |
        (S i) (Add_Hyp Q h) => (Add_Hyp Q (Strengthen_Hyps i h)).

Lemma Drop_S_Bridge_nat :
        (i,j:nat)
         ~i=j->
          (drop_V (S i) (S j))=
           (S (drop_V i j)).

Lemma In_Strength :
        (h:Hyps)(i,j:nat)(P:F)
         (In_Hyps i P h)->
          ~i=j->
           (In_Hyps (drop_V j i) P (Strengthen_Hyps j h)).

Definition
    n_admis_strengthen : (h:Hyps)(n:N)(Q:F)(N_Deduc h n Q)->Prop :=
        [h:Hyps][n:N][Q:F][D:(N_Deduc h n Q)]
         (i:nat)
          ~(Occurs_In_N i n)->
           (N_Deduc (Strengthen_Hyps i h) (drop_N i n) Q).

Definition
    a_admis_strengthen : (h:Hyps)(a:A)(Q:F)(A_Deduc h a Q)->Prop :=
        [h:Hyps][a:A][Q:F][D:(A_Deduc h a Q)]
         (i:nat)
          ~(Occurs_In_A i a)->
           (A_Deduc (Strengthen_Hyps i h) (drop_A i a) Q).

Lemma N_admis_strengthen :
        ((h:Hyps)(n:N)(Q:F)(n0:(N_Deduc h n Q))
```



```
        (n_admis_strengthen h n Q n0))/\
        ((h:Hyps)(a:A)(Q:F)(a0:(A_Deduc h a Q))
         (a_admis_strengthen h a Q a0)).

Lemma N_Admis_Strengthen :
        (h:Hyps)(n:N)(Q:F)(i:nat)
         (N_Deduc h n Q)->
          ~(Occurs_In_N i n)->
           (N_Deduc (Strengthen_Hyps i h) (drop_N i n) Q).

Lemma A_Admis_Strengthen :
        (h:Hyps)(a:A)(Q:F)(i:nat)
         (A_Deduc h a Q)->
          ~(Occurs_In_A i a)->
           (A_Deduc (Strengthen_Hyps i h) (drop_A i a) Q).

Definition l_admis_strengthen : (h:Hyps)(l:L)(Q:F)(L_Deriv h l Q)->Prop :=
        [h:Hyps][l:L][Q:F][l0:(L_Deriv h l Q)]
         (i:nat)
          ~(Occurs_In_L i l)->
           (L_Deriv (Strengthen_Hyps i h) (drop_L i l) Q).

Lemma L_admis_strengthen : (h:Hyps)(l:L)(Q:F)(l0:(L_Deriv h l Q))
                        (l_admis_strengthen h l Q l0).

Lemma L_Admis_Strengthen :
        (h:Hyps)(l:L)(Q:F)(i:nat)
         (L_Deriv h l Q)->
          ~(Occurs_In_L i l)->
           (L_Deriv (Strengthen_Hyps i h) (drop_L i l) Q).

Recursive Definition
    Hyps_Exchange : nat->Hyps->Hyps :=
        i MT => MT |
        i (Add_Hyp P MT) => (Add_Hyp P MT) |
        O (Add_Hyp P (Add_Hyp Q h)) =>
```

```
            (Add_Hyp Q (Add_Hyp P h)) |
      (S i) (Add_Hyp P (Add_Hyp Q h)) =>
            (Add_Hyp P (Hyps_Exchange i (Add_Hyp Q h))).

Recursive Definition
   V_Exchange : nat->V->V :=
      i j => (Setifb V
                  (nateqb i j)
                  (S i)
                  (Setifb V
                        (nateqb (S i) j)
                        i
                        j)).

Recursive Definition
   L_Exchange : nat->L->L :=
      i (vr x) => (vr (V_Exchange i x)) |
      i (app x l1 l2) =>
            (app (V_Exchange i x)
                 (L_Exchange i l1)
                 (L_Exchange (S i) l2)) |
      i (lm l) => (lm (L_Exchange (S i) l)).

Fixpoint
   M_Exchange1 [m:M] : nat->M :=
      [i:nat]<M>Case m of
         [x:V][ms:Ms]
            (sc (V_Exchange i x) (Ms_Exchange1 ms i))
         [m':M]
            (lambda (M_Exchange1 m' (S i)))
end with
   Ms_Exchange1 [ms:Ms] : nat->Ms :=
      [i:nat]<Ms>Case ms of
            mnil
         [m:M][ms':Ms]
            (mcons (M_Exchange1 m i) (Ms_Exchange1 ms' i))
```



```
end.

Recursive Definition
   M_Exchange : nat->M->M :=
      i m => (M_Exchange1 m i).

Recursive Definition
   Ms_Exchange : nat->Ms->Ms :=
      i ms => (Ms_Exchange1 ms i).

Fixpoint
   N_Exchange1 [n:N] : nat->N :=
      [i:nat]<N>Case n of
         [n':N]
            (lam (N_Exchange1 n' (S i)))
         [a:A]
            (an (A_Exchange1 a i))
end with
   A_Exchange1 [a:A] : nat->A :=
      [i:nat]<A>Case a of
         [a':A][n:N]
            (ap (A_Exchange1 a' i) (N_Exchange1 n i))
         [x:V]
            (var (V_Exchange i x))
end.

Recursive Definition N_Exchange : nat->N->N :=
i n => (N_Exchange1 n i).

Recursive Definition A_Exchange : nat->A->A :=
i a => (A_Exchange1 a i).

Lemma MExch1 : (i:nat)(x:V)(ms:Ms)
      ((M_Exchange i (sc x ms)) =
       (sc (V_Exchange i x) (Ms_Exchange i ms))).
```

```
Lemma MExch2 : (i:nat)(m:M)
        ((M_Exchange i (lambda m)) = (lambda (M_Exchange (S i) m))).

Lemma MExch3 : (i:nat)(Ms_Exchange i mnil)=mnil.

Lemma MExch4 : (i:nat)(m:M)
        (ms:Ms)((Ms_Exchange i (mcons m ms)) =
         (mcons (M_Exchange i m) (Ms_Exchange i ms))).

Lemma NExch1 : (i:nat)(n:N)
        ((N_Exchange i (lam n)) = (lam (N_Exchange (S i) n))).

Lemma NExch2 : (i:nat)(a:A)
        ((N_Exchange i (an a)) = (an (A_Exchange i a))).

Lemma NExch3 : (i:nat)(a:A)(n:N)
        ((A_Exchange i (ap a n)) =
         (ap (A_Exchange i a) (N_Exchange i n))).

Lemma NExch4 : (i:nat)(x:V)
        ((A_Exchange i (var x)) = (var (V_Exchange i x))).

Lemma Hyps_ref_eq :
        (i,j:V)(P,Q:F)(h:Hyps)
         i=j->
          (In_Hyps i P h)->
           (In_Hyps j Q h)->
            P=Q.

Lemma V_Exch_S_Bridge :
        (i,j:nat)
         (V_Exchange (S i) (S j))=
          (S (V_Exchange i j)).

Lemma V_Exch_id :
        (i:nat)
```



```
        (V_Exchange i i)=(S i).

Lemma In_Exch1 :
        (h:Hyps)(i,j:nat)(P:F)
         (lt i j)->
          (In_Hyps i P h)->
           (In_Hyps i P (Hyps_Exchange j h)).

Lemma In_Exch2 :
        (h:Hyps)(i,j:nat)(P:F)
         (lt (S j) i)->
          (In_Hyps i P h)->
           (In_Hyps i P (Hyps_Exchange j h)).

Lemma In_Exch :
        (i,j:nat)(h:Hyps)(P,Q,R:F)
         (In_Hyps i P h)->
          (In_Hyps j Q h)->
           (In_Hyps (S j) R h)->
            (In_Hyps (V_Exchange j i) P (Hyps_Exchange j h)).

Definition l_admis_exch1 :
        (h:Hyps)(l:L)(R:F)(L_Deriv h l R)->Prop :=
         [h:Hyps][l:L][R:F][lO:(L_Deriv h l R)](j:nat)(P,Q:F)
          (In_Hyps j P h)->
           (In_Hyps (S j) Q h)->
            (L_Deriv (Hyps_Exchange j h)
                     (L_Exchange j l)
                     R).

Lemma L_admis_exch :
        (h:Hyps)(l:L)(R:F)(D:(L_Deriv h l R))
         (l_admis_exch1 h l R D).

Lemma L_Admis_Exch :
        (h:Hyps)(l:L)(R:F)(j:nat)(P,Q:F)
```

```
        (L_Deriv h l R)->
         (In_Hyps j P h)->
          (In_Hyps (S j) Q h)->
           (L_Deriv (Hyps_Exchange j h)
                    (L_Exchange j l)
                    R).

Lemma Hyps_Exchange_Top :
        (P,Q:F)(h:Hyps)
         (Hyps_Exchange O (Add_Hyp P (Add_Hyp Q h)))=
          (Add_Hyp Q (Add_Hyp P h)).

Lemma L_Admis_Exch_Top :
        (P,Q,R:F)(h:Hyps)(l:L)
         (L_Deriv (Add_Hyp P (Add_Hyp Q h)) l R)->
          (L_Deriv (Add_Hyp Q (Add_Hyp P h)) (L_Exchange O l) R).

Definition n_admis_exch :
        (h:Hyps)(n:N)(R:F)(N_Deduc h n R)->Prop :=
         [h:Hyps][n:N][R:F][n0:(N_Deduc h n R)](j:nat)(P,Q:F)
          (In_Hyps j P h)->
           (In_Hyps (S j) Q h)->
            (N_Deduc (Hyps_Exchange j h)
                     (N_Exchange j n)
                     R).

Definition a_admis_exch :
        (h:Hyps)(a:A)(R:F)(A_Deduc h a R)->Prop :=
         [h:Hyps][a:A][R:F][a0:(A_Deduc h a R)](j:nat)(P,Q:F)
          (In_Hyps j P h)->
           (In_Hyps (S j) Q h)->
            (A_Deduc (Hyps_Exchange j h)
                     (A_Exchange j a)
                     R).

Lemma N_admis_exch :
```



```
        ((h:Hyps)(n:N)(R:F)(n0:(N_Deduc h n R))(n_admis_exch h n R n0))/\
         ((h:Hyps)(a:A)(R:F)(a0:(A_Deduc h a R))(a_admis_exch h a R a0)).

Lemma N_Admis_Exch : (h:Hyps)(n:N)(R:F)(j:nat)(P,Q:F)
        (N_Deduc h n R)->
         (In_Hyps j P h)->
          (In_Hyps (S j) Q h)->
           (N_Deduc (Hyps_Exchange j h)
                    (N_Exchange j n)
                    R).

Lemma A_Admis_Exch : (h:Hyps)(a:A)(R:F)(j:nat)(P,Q:F)
        (A_Deduc h a R)->
         (In_Hyps j P h)->
          (In_Hyps (S j) Q h)->
           (A_Deduc (Hyps_Exchange j h)
                    (A_Exchange j a)
                    R).

Lemma V_Exchange_inv :
        (x:V)(i,j:nat)
         i=j->
          (V_Exchange i (V_Exchange j x))=x.

Lemma L_Exchange_inv :
        (l:L)(i,j:nat)
         i=j->
          (L_Exchange i (L_Exchange j l))=l.

Lemma M_Admis_Weaken :
        (h:Hyps)(m:M)(P:F)(j:nat)(Q:F)
         (M_Deriv h m P)->
          (lt j (S (Len_Hyps h)))->
           (M_Deriv (Weaken_Hyps j Q h) (lift_M j m) P).

Lemma Ms_Admis_Weaken :
```

```
        (ms:Ms)(h:Hyps)(R:F)(P:F)(j:nat)(Q:F)
         (Ms_Deriv h R ms P)->
          (lt j (S (Len_Hyps h)))->
           (Ms_Deriv (Weaken_Hyps j Q h) R (lift_Ms j ms) P).

Lemma M_Admis_Weaken_Top :
        (h:Hyps)(m:M)(P:F)
         (M_Deriv h m P)->
          (Q:F)(M_Deriv (Add_Hyp Q h) (lift_M 0 m) P).

Lemma Ms_Admis_Weaken_Top :
        (ms:Ms)(h:Hyps)(R,P:F)
         (Ms_Deriv h R ms P)->
          (Q:F)(Ms_Deriv (Add_Hyp Q h) R (lift_Ms 0 ms) P).

Lemma N_Admis_Weaken_Top :
        (h:Hyps)(n:N)(P:F)
         (N_Deduc h n P)->
          (Q:F)(N_Deduc (Add_Hyp Q h) (lift_N 0 n) P).

Lemma A_Admis_Weaken_Top :
        (h:Hyps)(a:A)(P:F)
         (A_Deduc h a P)->
          (Q:F)(A_Deduc (Add_Hyp Q h) (lift_A 0 a) P).

Lemma L_Admis_Weaken_Top :
        (h:Hyps)(l:L)(P:F)
         (L_Deriv h l P)->
          (Q:F)(L_Deriv (Add_Hyp Q h) (lift_L 0 l) P).

Definition lift_rhobar_bridge : M->Prop :=
        [m:M](i:nat)
         (lift_L i (rhobar m))=
          (rhobar (lift_M i m)).

Definition lift_rhobar1_bridge : Ms->Prop :=
```



```
        [ms:Ms](i,j:nat)(m:M)
         (lt 0 i)->
          ((m1:M)(k:nat)
            (lt (Height_M m1) (Height_Ms (mcons m ms)))->
             (lift_L k (rhobar m1))=
              (rhobar (lift_M k m1)))->
           (rhobar1 i (lift_Ms j (mcons m ms)))=
            (lift_L (plus i j) (rhobar1 i (mcons m ms))).

Lemma lift_rhobar_Bridge :
        ((m:M)(lift_rhobar_bridge m))/\
         ((ms:Ms)(lift_rhobar1_bridge ms)).

Lemma Lift_RhoBar_Bridge :
        (m:M)(i:nat)
         (lift_L i (rhobar m))=
          (rhobar (lift_M i m)).

Lemma Lift_RhoBar1_Bridge :
        (ms:Ms)(i,j:nat)(m:M)
         (lt 0 i)->
          (rhobar1 i (lift_Ms j (mcons m ms)))=
            (lift_L (plus i j) (rhobar1 i (mcons m ms))).

Lemma Lifts_RhoBar_Bridge :
        (i,j:nat)(m:M)
         (lifts_L i j (rhobar m))=
          (rhobar (lifts_M i j m)).

Lemma Lift_RhoBar1_Bridge1 :
        (ms:Ms)(i,j:nat)
         (lt 0 i)->
          (rhobar1 i (lift_Ms j ms))=
            (lift_L (plus i j) (rhobar1 i ms)).

Lemma RhoBar1_Lifts_Ms1 :
```

```
        (ms:Ms)(i,j:nat)
         (rhobar1 j (lifts_Ms (S i) O ms))=
          (rhobar1 (S j) (lifts_Ms i O ms)).

Lemma RhoBar1_Lifts_Ms :
        (i,j:nat)(ms:Ms)
         (rhobar1 j (lifts_Ms i O ms))=
          (rhobar1 (plus j i) ms).

Definition rhobar21 : M->Prop :=
        [m:M](x:V)(ms:Ms)
         ((ms1:Ms)(i:nat)
           (lt (Height_Ms ms1) (Height_Ms (mcons m ms)))->
            (rhobar1 (S i) ms1)=
             (rhobar (sc O (lifts_Ms (S i) O ms1))))->
          (rhobar (sc x (mcons m ms)))=
           (app x (rhobar m) (rhobar (sc O (lift_Ms O ms)))).

Definition rhobar22 : Ms->Prop :=
        [ms:Ms](i:nat)
         (rhobar1 (S i) ms)=
          (rhobar (sc O (lifts_Ms (S i) O ms))).

Lemma Rhobar21 :
        ((m:M)(rhobar21 m))/\
         ((ms:Ms)(rhobar22 ms)).

Lemma RhoBar1 : (x:V)
        (rhobar (sc x mnil))=(vr x).

Lemma RhoBar2 : (ms:Ms)(x:V)(m:M)
        (rhobar (sc x (mcons m ms)))=
         (app x (rhobar m) (rhobar (sc O (lift_Ms O ms)))).

Lemma RhoBar3 : (m:M)
        (rhobar (lambda m))=(lm (rhobar m)).
```



```
Definition l_admis_rhobar_m : M->Prop :=
        [m:M](h:Hyps)(P:F)
         (M_Deriv h m P)->
          (L_Deriv h (rhobar m) P).

Definition l_admis_rhobar_ms : Ms->Prop :=
        [ms:Ms](h:Hyps)(P,Q:F)
         ((m1:M)(h1:Hyps)(P1:F)
           (lt (Height_M m1) (Height_Ms ms))->
            (M_Deriv h1 m1 P1)->
             (L_Deriv h1 (rhobar m1) P1))->
          (Ms_Deriv (Add_Hyp Q h) Q ms P)->
           (L_Deriv (Add_Hyp Q h) (rhobar (sc O ms)) P).

Lemma L_Admis_RhoBar1 :
        ((m:M)(l_admis_rhobar_m m))/\
         ((ms:Ms)(l_admis_rhobar_ms ms)).

Lemma L_Admis_RhoBar : (h:Hyps)(m:M)(P:F)
        (M_Deriv h m P)->
         (L_Deriv h (rhobar m) P).

Lemma L_Admis_Rho : (h:Hyps)(n:N)(P:F)
        (N_Deduc h n P)->
         (L_Deriv h (rho n) P).

Definition
    n_admis_sub :
        (h:Hyps)(n:N)(P:F)(N_Deduc h n P)->Prop :=
         [h:Hyps][n:N][P:F][D:(N_Deduc h n P)]
          (g:nat)(Q:F)(a0:A)
           (In_Hyps g Q h)->
            (A_Deduc (Strengthen_Hyps g h) a0 Q)->
             (N_Deduc (Strengthen_Hyps g h)
                      (NsubstAV a0 g n)
```

```
                P).

Definition
    a_admis_sub :
        (h:Hyps)(a:A)(P:F)(A_Deduc h a P)->Prop :=
         [h:Hyps][a:A][P:F][D:(A_Deduc h a P)]
         (g:nat)(Q:F)(a0:A)
          (In_Hyps g Q h)->
            (A_Deduc (Strengthen_Hyps g h) a0 Q)->
             (A_Deduc (Strengthen_Hyps g h)
                      (AsubstAV a0 g a)
                      P).

Lemma N_admis_sub :
        ((h:Hyps)(n:N)(P:F)(D:(N_Deduc h n P))(n_admis_sub h n P D))/\
         ((h:Hyps)(a:A)(P:F)(D:(A_Deduc h a P))(a_admis_sub h a P D)).

Lemma N_Admis_Sub :
        (h:Hyps)(n:N)(P:F)(Q:F)(a0:A)(g:nat)
         (N_Deduc h n P)->
          (In_Hyps g Q h)->
            (A_Deduc (Strengthen_Hyps g h) a0 Q)->
             (N_Deduc (Strengthen_Hyps g h)
                      (NsubstAV a0 g n)
                      P).

Lemma A_Admis_Sub :
        (h:Hyps)(a:A)(P:F)(Q:F)(a0:A)(g:nat)
         (A_Deduc h a P)->

          (In_Hyps g Q h)->
            (A_Deduc (Strengthen_Hyps g h) a0 Q)->
             (A_Deduc (Strengthen_Hyps g h)
                      (AsubstAV a0 g a)
                      P).
```



```
Lemma SW_Id :
        (h:Hyps)(i:nat)(P:F)
         (Strengthen_Hyps i (Weaken_Hyps i P h))=
          h.

Definition n_admis_phi : (h:Hyps)(l:L)(P:F)(L_Deriv h l P)->Prop :=
[h:Hyps][l:L][P:F][l0:(L_Deriv h l P)](N_Deduc h (phi l) P).

Lemma N_Admis_Phi_1 : (h:Hyps)(l:L)(P:F)(l0:(L_Deriv h l P))
                (n_admis_phi h l P l0).

Lemma N_Admis_Phi : (h:Hyps)(l:L)(P:F)
        (L_Deriv h l P)->
         (N_Deduc h (phi l) P).

Lemma M_Admis_PhiBar : (h:Hyps)(l:L)(P:F)
        (L_Deriv h l P)->
         (M_Deriv h (phibar l) P).

Lemma Exchange_Lift_V :
        (v,x,y:V)
         (x=y)->
          (V_Exchange x (lift_V y v))=
           (lift_V (S y) v).

Definition exchange_lift_m : M->Prop :=
        [m:M](x,y:V)
         x=y->
          (M_Exchange x (lift_M y m))=
           (lift_M (S y) m).

Definition exchange_lift_ms : Ms->Prop :=
        [ms:Ms](x,y:V)
         x=y->
          (Ms_Exchange x (lift_Ms y ms))=
           (lift_Ms (S y) ms).
```

```
Lemma Exchange_lift_m :
        ((m:M)(exchange_lift_m m))/\
         ((ms:Ms)(exchange_lift_ms ms)).

Lemma Exchange_Lift_M :
        (x,y:V)(m:M)
         x=y->
          (M_Exchange x (lift_M y m))=
           (lift_M (S y) m).

Lemma Exchange_Lift_Ms :
        (x,y:V)(ms:Ms)
         x=y->
          (Ms_Exchange x (lift_Ms y ms))=
           (lift_Ms (S y) ms).

Lemma Lift_Exchange_V1 :
        (v,x,y:V)
         (lt x (S y))->
          (lift_V x (V_Exchange y v))=
           (V_Exchange (S y) (lift_V x v)).

Definition lift_exchange_m1 : M->Prop :=
        [m:M](x,y:nat)
         (lt x (S y))->
          (lift_M x (M_Exchange y m))=
           (M_Exchange (S y) (lift_M x m)).

Definition lift_exchange_ms1 : Ms->Prop :=
        [ms:Ms](x,y:nat)
         (lt x (S y))->
          (lift_Ms x (Ms_Exchange y ms))=
           (Ms_Exchange (S y) (lift_Ms x ms)).

Lemma lift_exchange_M1 :
```



```
        ((m:M)(lift_exchange_m1 m))/\
         ((ms:Ms)(lift_exchange_ms1 ms)).

Lemma Lift_Exchange_M1 :
        (m:M)(x,y:nat)
         (lt x (S y))->
          (lift_M x (M_Exchange y m))=
           (M_Exchange (S y) (lift_M x m)).

Lemma Lift_Exchange_Ms1 :
        (ms:Ms)(x,y:nat)
         (lt x (S y))->
          (lift_Ms x (Ms_Exchange y ms))=
           (Ms_Exchange (S y) (lift_Ms x ms)).

Definition exchange_rhobar_bridge : M->Prop :=
        [m:M](x:V)
         (L_Exchange x (rhobar m))=
          (rhobar (M_Exchange x m)).

Definition exchange_rhobar1_bridge : Ms->Prop :=
        [ms:Ms](x,y:V)
         (L_Exchange x (rhobar (sc y ms)))=
          (rhobar (M_Exchange x (sc y ms))).

Lemma Exchange_rhobar_bridge :
        ((m:M)(exchange_rhobar_bridge m))/\
         ((ms:Ms)(exchange_rhobar1_bridge ms)).

Lemma Exchange_RhoBar_Bridge :
        (x:nat)(m:M)
         (L_Exchange x (rhobar m))=
          (rhobar (M_Exchange x m)).

Definition height_m_exchange : M->Prop :=
        [m:M](x:nat)
```

```
        (Height_M (M_Exchange x m))=
         (Height_M m).

Definition height_ms_exchange : Ms->Prop :=
        [ms:Ms](x:nat)
         (Height_Ms (Ms_Exchange x ms))=
          (Height_Ms ms).

Lemma Height_m_exchange :
        ((m:M)(height_m_exchange m))/\
         ((ms:Ms)(height_ms_exchange ms)).

Lemma Height_M_Exchange :
        (m:M)(x:nat)
         (Height_M (M_Exchange x m))=
          (Height_M m).

Lemma Height_Ms_Exchange :
        (ms:Ms)(x:nat)
         (Height_Ms (Ms_Exchange x ms))=
          (Height_Ms ms).

Definition msub_exch_bridge1 : M->Prop :=
        [m:M](x,y:V)(m1:M)
         (MsubstVMV x m1 y (M_Exchange y m))=
          (MsubstVMV x m1 (S y) m).

Definition mssub_exch_bridge1 : Ms->Prop :=
        [ms:Ms](x,y:V)(m1:M)
         (MssubstVMV x m1 y (Ms_Exchange y ms))=
          (MssubstVMV x m1 (S y) ms).

Lemma Msub_exch_bridge1 :
        ((m:M)(msub_exch_bridge1 m))/\
         ((ms:Ms)(mssub_exch_bridge1 ms)).
```



```
Lemma Msub_Exch_Bridge1 :
        (m:M)(x,y:V)(m1:M)
        (MsubstVMV x m1 y (M_Exchange y m))=
         (MsubstVMV x m1 (S y) m).

Lemma Mssub_Exch_Bridge1 :
        (ms:Ms)(x,y:V)(m1:M)
         (MssubstVMV x m1 y (Ms_Exchange y ms))=
          (MssubstVMV x m1 (S y) ms).

Mutual Inductive
    Norm_L : L->Prop :=
        norm_vr : (x:V)(Norm_L (vr x)) |
        norm_app :
                (x:V)(l1,l2:L)
                 (Norm_L l1)->
                  (Norm'_L l2)->
                   (Norm_L (app x l1 l2)) |
        norm_lm :
                (l:L)
                 (Norm_L l)->
                  (Norm_L (lm l))
with
    Norm'_L : L->Prop :=
        norm'_vr : (Norm'_L (vr O)) |
        norm'_app :
                (l1,l2:L)
                 (Norm_L l1)->
                  (Norm'_L l2)->
                   ~(Occurs_In_L O l1)->
                    ~(Occurs_In_L (S O) l2)->
                     (Norm'_L (app O l1 l2)).

Scheme Norm_Norm'_L_ind1 := Induction for Norm_L Sort Prop
  with Norm'_Norm_L_ind1 := Induction for Norm'_L Sort Prop.
```

```
Lemma Norm_Norm'_L_ind :
        (P:(1:L)(Norm_L 1)->Prop)
        (P0:(1:L)(Norm'_L 1)->Prop)
         ((x:V)(P (vr x) (norm_vr x)))
          ->((x:V)
              (l1,l2:L)
               (n:(Norm_L l1))
                (P l1 n)
                 ->(n0:(Norm'_L l2))
                    (P0 l2 n0)
                     ->(P (app x l1 l2) (norm_app x l1 l2 n n0)))
          ->((1:L)(n:(Norm_L 1))(P 1 n)->(P (lm 1) (norm_lm 1 n)))
             ->(P0 (vr O) norm'_vr)
                ->((l1,l2:L)
                    (n:(Norm_L l1))
                     (P l1 n)
                      ->(n0:(Norm'_L l2))
                         (P0 l2 n0)
                          ->(n1:~(Occurs_In_L O l1))
                             (n2:~(Occurs_In_L (S O) l2))
                              (P0 (app O l1 l2)
                                (norm'_app l1 l2 n n0 n1 n2)))
                  ->((1:L)(n:(Norm_L 1))(P 1 n))/\
                    ((1:L)(n:(Norm'_L 1))(P0 1 n)).

Fixpoint
    Norm_Lb [1:L] : bool :=
        <bool>Case 1 of
            [x:V]
                true
            [x:V][l0,l1:L]
                (andb (Norm_Lb l0)
                      (Norm'_Lb l1))
            [1:L]
                (Norm_Lb 1)
        end with
```



```
    Norm'_Lb [1:L] : bool :=
        <bool>Case 1 of
            [x:V]
                (nateqb x O)
            [x:V][l0,l1:L]
                (andb (nateqb x O)
                (andb (negb (Occurs_In_L1 O l0))
                (andb (negb (Occurs_In_L1 (S O) l1))
                (andb (Norm_Lb l0)
                      (Norm'_Lb l1)))))
            [1:L]
                false
        end.

Lemma NMLB1 :
        (x:V)
         (Norm_Lb (vr x))=true.

Lemma NMLB2 :
        (x:V)(l0,l1:L)
         (Norm_Lb (app x l0 l1))=
         (andb (Norm_Lb l0)
               (Norm'_Lb l1)).

Lemma NMLB3 :
        (1:L)
         (Norm_Lb (lm 1))=
          (Norm_Lb 1).

Lemma NMLB4 :
        (x:V)
         (Norm'_Lb (vr x))=
          (nateqb x O).

Lemma NMLB5 :
        (x:V)(l0,l1:L)
```

```
        (Norm'_Lb (app x 10 11))=
        (andb (nateqb x 0)
        (andb (negb (Occurs_In_L1 0 10))
        (andb (negb (Occurs_In_L1 (S 0) 11))
        (andb (Norm_Lb 10)
              (Norm'_Lb 11))))).

Lemma NMLB6 :
        (1:L)
         (Norm'_Lb (lm 1))=
          false.

Lemma nmlb1_is_nmlb1 :
        (1:L)
         ((Norm_L 1)->
          (Norm_Lb 1)=true)/\
         ((Norm'_L 1)->
          (Norm'_Lb 1)=true).

Lemma NMLB1_is_NMLB1 :
        (1:L)
         ((Norm_L 1)->
          (Norm_Lb 1)=true).

Lemma NM'LB1_is_NM'LB1 :
        (1:L)
         ((Norm'_L 1)->
          (Norm'_Lb 1)=true).

Lemma nmlb1_is_nmlb2 :
        (1:L)
         ((Norm_Lb 1)=true->
          (Norm_L 1))/\
         ((Norm'_Lb 1)=true->
          (Norm'_L 1)).
```



```
Lemma NMLB1_is_NMLB2 :
        (1:L)
         ((Norm_Lb 1)=true->
          (Norm_L 1)).

Lemma NM'LB1_is_NM'LB2 :
        (1:L)
         ((Norm'_Lb 1)=true->
          (Norm'_L 1)).

Lemma NMLB1_is_NMLB3 :
        (1:L)
         (~(Norm_L 1)->
          (Norm_Lb 1)=false).

Lemma NM'LB1_is_NM'LB3 :
        (1:L)
         (~(Norm'_L 1)->
          (Norm'_Lb 1)=false).

Lemma NMLB1_is_NMLB4 :
        (1:L)
         ((Norm_Lb 1)=false->
          ~(Norm_L 1)).

Lemma NM'LB1_is_NM'LB4 :
        (1:L)
         ((Norm'_Lb 1)=false->
          ~(Norm'_L 1)).

Definition NML_compare : L->Prop :=
        [1:L]
         ((Norm_L 1)\/~(Norm_L 1)).

Lemma NML_dec : (1:L)(NML_compare 1).
```

```
Definition NM'L_compare : L->Prop :=
        [l:L]
         ((Norm'_L l)\/~(Norm'_L l)).

Lemma NM'L_dec : (l:L)(NM'L_compare l).

Definition noi_rhobar1 : M->Prop :=
        [m:M](i:nat)
         ~(Occurs_In_L (S i)
          (rhobar (lift_M i (lift_M i m)))).

Definition noi_rhobar2 : Ms->Prop :=
        [ms:Ms](i:nat)
         ~(Occurs_In_L (S i)
          (rhobar (sc 0 (lift_Ms i (lift_Ms i ms))))).

Lemma noi_rhobar :
        ((m:M)(noi_rhobar1 m))/\
         ((ms:Ms)(noi_rhobar2 ms)).

Lemma NOI_RhoBar1 :
        (m:M)(i:nat)
         ~(Occurs_In_L (S i)
          (rhobar (lift_M i (lift_M i m)))).

Lemma NOI_RhoBar2 :
        (ms:Ms)(i:nat)
         ~(Occurs_In_L (S i)
          (rhobar (sc 0 (lift_Ms i (lift_Ms i ms))))).

Definition norm_l_rhobar_m : M->Prop :=
        [m:M](Norm_L (rhobar m)).

Definition norm_l_rhobar_ms : Ms->Prop :=
        [ms:Ms]
          (Norm'_L (rhobar (sc 0 (lift_Ms 0 ms)))).
```



```
Lemma norm_l_rhobar :
        ((m:M)(norm_l_rhobar_m m))/\
         ((ms:Ms)(norm_l_rhobar_ms ms)).

Lemma Norm_L_RhoBar : (m:M)
        (Norm_L (rhobar m)).

Lemma Norm'_L_RhoBar : (ms:Ms)
        (Norm'_L (rhobar (sc 0 (lift_Ms 0 ms)))).

Inductive
    L_Perm1 : L->L->Prop :=
        l_perm1_lm :
                (l1,l2:L)
                  (L_Perm1 l1 l2)->
                   (L_Perm1 (lm l1) (lm l2)) |
         l_perm1_app1 :
                (i:V)(l11,l12,l2:L)
                  (L_Perm1 l11 l12)->
                    (L_Perm1 (app i l11 l2) (app i l12 l2)) |
         l_perm1_app2 :
                (i:V)(l1,l21,l22:L)
                   (L_Perm1 l21 l22)->
                     (L_Perm1 (app i l1 l21) (app i l1 l22)) |
         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,z:V)(l1,l2,l3:L)
                 ((Occurs_In_L 0 l2)\/(Occurs_In_L (S 0) l3))->
                  (Norm'_L l3)->
                   (L_Perm1 (app x l1 (app (S z) l2 l3))
                            (app z
                                 (app x l1 l2)
```

```
                    (app (lift_V 0 x)
                         (lift_L 0 11)
                         (L_Exchange 0 13)))) |
    l_perm1_app_app2 :
        (x:V)(11,12,13:L)
         ((Occurs_In_L 0 12)\/(Occurs_In_L (S 0) 13))->
          (Norm'_L 13)->
           (L_Perm1 (app x 11 (app 0 12 13))
                  (app x
                    11
                    (app 0
                         (app (lift_V 0 x)
                              (lift_L 0 11)
                              (lift_L (S 0) 12))
                         (app (lifts_V (S (S 0)) 0 x)
                              (lifts_L (S (S 0)) 0 11)
                              (L_Exchange 0
                                (lift_L (S (S 0)) 13)))))) |
    l_perm1_app_lm : (x:V)(11,12:L)
            (L_Perm1 (app x 11 (lm 12))
                  (lm (app (lift_V 0 x)
                           (lift_L 0 11)
                           (L_Exchange 0 12)))).

Scheme L_Perm1_ind1 := Induction for L_Perm1 Sort Prop.

Lemma perm_not_norm_11 :
        (10,11,12:L)(x:V)
         (Occurs_In_L 0 11)->
          ~(Norm_L (app x 10 (app 0 11 12))).

Lemma perm_not_norm_12 :
        (10,11,12:L)(x:V)
         (Occurs_In_L (S 0) 12)->
          ~(Norm_L (app x 10 (app 0 11 12))).
```



```
Lemma Norm'_Norm_L :
        (1:L)
         (Norm'_L 1)->
          (Norm_L 1).

Lemma Perm_not_norm_1 :
        (10,11:L)
         (L_Perm1 10 11)->
          ~(Norm_L 10).

Lemma Perm_Not_Norm_L :
        (10,11:L)
         (L_Perm1 10 11)->
          ~(Norm_L 10).

Lemma Norm_Imperm_L :
        (10,11:L)
         (Norm_L 10)->
          ~(L_Perm1 10 11).

Inductive
    L_Permn : L->L->Prop :=
        l_permn_base :
            (10,11:L)
               10=11->
                (L_Permn 10 11) |
        l_permn_rec :
            (10,11,12:L)
             (L_Perm1 10 11)->
              (L_Permn 11 12)->
               (L_Permn 10 12).

Scheme L_Permn_ind1 := Induction for L_Permn Sort Prop.

Definition l_admis_perm1 :
        (1,10:L)(L_Perm1 1 10)->Prop :=
```

```
        [1,10:L][d:(L_Perm1 1 10)]
          (h:Hyps)(P:F)(L_Deriv h 1 P)->
           (L_Deriv h 10 P).

Lemma L_admis_perm1 :
        (1,10:L)(D:(L_Perm1 1 10))
         (l_admis_perm1 1 10 D).

Lemma L_Admis_Perm1 :
        (1,10:L)(h:Hyps)(P:F)
         (L_Perm1 1 10)->
          (L_Deriv h 1 P)->
           (L_Deriv h 10 P).

Lemma L_Perm1n :
        (1,10:L)
         (L_Perm1 1 10)->
          (L_Permn 1 10).

Definition l_permnn : (1,10:L)(L_Permn 1 10)->Prop :=
        [1,10:L][d:(L_Permn 1 10)]
         (11:L)
          (L_Permn 10 11)->
           (L_Permn 1 11).

Lemma L_permnn :
        (1,10:L)(d:(L_Permn 1 10))
         (l_permnn 1 10 d).

Lemma L_Permnn :
        (1,10,11:L)
         (L_Permn 1 10)->
          (L_Permn 10 11)->
           (L_Permn 1 11).

Definition l_permn_app1 :
```



```
        (1,10:L)(L_Permn 1 10)->Prop :=
         [1,10:L][D:(L_Permn 1 10)]
          (x:V)(11:L)
           (L_Permn (app x 1 11) (app x 10 11)).

Lemma L_permn_app1 :
        (1,10:L)(D:(L_Permn 1 10))
         (l_permn_app1 1 10 D).

Lemma L_Permn_app1 :
        (1,10,11:L)(x:V)
         (L_Permn 1 10)->
          (L_Permn (app x 1 11) (app x 10 11)).

Definition l_permn_app2 :
        (10,11:L)(L_Permn 10 11)->Prop :=
         [10,11:L][D:(L_Permn 10 11)]
          (x:V)(1:L)
           (L_Permn (app x 1 10) (app x 1 11)).

Lemma L_permn_app2 :
        (10,11:L)(D:(L_Permn 10 11))
         (l_permn_app2 10 11 D).

Lemma L_Permn_app2 :
        (1,10,11:L)(x:V)
         (L_Permn 10 11)->
          (L_Permn (app x 1 10) (app x 1 11)).

Lemma L_Permn_app :
        (x:V)(10,11,12,13:L)
         (L_Permn 10 11)->
           (L_Permn 12 13)->
            (L_Permn (app x 10 12) (app x 11 13)).

Definition l_permn_lm :
```

```
        (l0,l1:L)(L_Permn l0 l1)->Prop :=
         [l0,l1:L][d:(L_Permn l0 l1)]
            (L_Permn (lm l0) (lm l1)).

Lemma L_permn_lm :
        (l0,l1:L)(d:(L_Permn l0 l1))
         (l_permn_lm l0 l1 d).

Lemma L_Permn_lm :
        (l,l0:L)
         (L_Permn l l0)->
          (L_Permn (lm l) (lm l0)).

Definition l_admis_permn :
        (l,l0:L)(L_Permn l l0)->Prop :=
         [l,l0:L][d:(L_Permn l l0)]
          (h:Hyps)(P:F)
           (L_Deriv h l P)->
            (L_Deriv h l0 P).

Lemma L_admis_permn :
        (l,l0:L)(d:(L_Permn l l0))
         (l_admis_permn l l0 d).

Lemma L_Admis_Permn :
        (h:Hyps)(l0,l1:L)(P:F)
         (L_Permn l0 l1)->
          (L_Deriv h l0 P)->
           (L_Deriv h l1 P).

Definition oi_rhobar_m : M->Prop :=
        [m:M](x:V)
         (Occurs_In_M x m)->
          (Occurs_In_L x (rhobar m)).

Definition oi_rhobar_ms : Ms->Prop :=
```



```
        [ms:Ms](x,y:V)
         (Occurs_In_Ms x ms)->
          (Occurs_In_L (S x) (rhobar (sc y (lift_Ms O ms)))).

Lemma oi_rhobar_M :
        ((m:M)(oi_rhobar_m m))/\
         ((ms:Ms)(oi_rhobar_ms ms)).

Lemma OI_RhoBar_M :
        (m:M)(x:V)
         (Occurs_In_M x m)->
          (Occurs_In_L x (rhobar m)).

Lemma OI_RhoBar_Ms :
        (ms:Ms)(x,y:V)
         (Occurs_In_Ms x ms)->
          (Occurs_In_L (S x) (rhobar (sc y (lift_Ms O ms)))).

Lemma NOI_RhoBar_M :
        (m:M)(x:V)
         ~(Occurs_In_L x (rhobar m))->
          ~(Occurs_In_M x m).

Lemma NOI_RhoBar_Ms :
        (ms:Ms)(x:V)
         ~(Occurs_In_L (S x) (rhobar (sc O (lift_Ms O ms))))->
          ~(Occurs_In_Ms x ms).

Definition oi_rhobar_m1 : M->Prop :=
        [m:M](x:V)
         (Occurs_In_L x (rhobar m))->
          (Occurs_In_M x m).

Definition oi_rhobar_ms1 : Ms->Prop :=
        [ms:Ms](x:V)
         (Occurs_In_L (S x) (rhobar (sc O (lift_Ms O ms))))->
```

```
        (Occurs_In_Ms x ms).

Lemma oi_rhobar_M1 :
      ((m:M)(oi_rhobar_m1 m))/\
       ((ms:Ms)(oi_rhobar_ms1 ms)).

Lemma OI_RhoBar_M1 :
      (x:V)(m:M)
       (Occurs_In_L x (rhobar m))->
        (Occurs_In_M x m).

Lemma OI_RhoBar_Ms1 :
      (x:V)(ms:Ms)
       (Occurs_In_L (S x) (rhobar (sc O (lift_Ms O ms))))->
        (Occurs_In_Ms x ms).

Lemma NOI_RhoBar_M1 :
      (x:V)(m:M)
       ~(Occurs_In_M x m)->
        ~(Occurs_In_L x (rhobar m)).

Lemma NOI_RhoBar_Ms1 :
      (x:V)(ms:Ms)
       ~(Occurs_In_Ms x ms)->
        ~(Occurs_In_L (S x) (rhobar (sc O (lift_Ms O ms)))).

Lemma Drop_RhoBar_Bridge :
      (x:V)(m:M)
       ~(Occurs_In_M x m)->
        (drop_L x (rhobar m))=
         (rhobar (drop_M x m)).

Lemma Drop_Lift_M_Bridge1 :
      (m:M)(i,j:nat)
       ~(Occurs_In_M i m)->
        (lt j (S i))->
```



```
         (drop_M (S i) (lift_M j m))=(lift_M j (drop_M i m)).

Lemma Drop_Lift_Ms_Bridge1 :
      (ms:Ms)(i,j:nat)
       ~(Occurs_In_Ms i ms)->
        (lt j (S i))->
         (drop_Ms (S i) (lift_Ms j ms))=(lift_Ms j (drop_Ms i ms)).

Definition app_red_m : M->Prop :=
      [m:M](x:V)(m1:M)
        (L_Permn (app x (rhobar m1) (rhobar m))
                 (rhobar (MsubstVMV x m1 O m))).

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

Lemma app_red :
      ((m:M)(app_red_m m))/\
       ((ms:Ms)(app_red_ms ms)).

Lemma App_Red_M :
      (x:V)(m1,m:M)
       (L_Permn (app x (rhobar m1) (rhobar m))
                (rhobar (MsubstVMV x m1 O m))).

Lemma Norm_Red :
      (l:L)(L_Permn l (rhobar (phibar l))).
```