Working with Linear Logic in Coq

James Power and Caroline Webster

Department of Computer Science,
National University of Ireland Maynooth,
Maynooth, Co. Kildare, Ireland.

Abstract. In this paper we describe the encoding of linear logic in the Coq system, a proof assistant for higher-order logic. This process involved encoding a suitable consequence relation, the relevant operators, and some auxiliary theorems and tactics. The encoding allows us to state and prove theorems in linear logic, and we demonstrate its use through two examples: a simple blocks world scenario, and the Towers of Hanoi problem.

1 Introduction

In this paper we describe an encoding of intuitionistic linear logic using the Coq proof assistant. This encoding allows for the type-checking of specifications in linear logic, and the construction and validation of proofs in that logic.

Previous work in mechanising or implementing linear logic has either encoded it as a distinct object logic [8], or as a logic programming language such as Lygon [6] or Lolli [7]. One of the specific features of our encoding is that the constructs of the Coq environment may be used in association with the linear logic proofs, including in particular the extensive set of inductive datatypes and their properties and proofs that form the Coq library.

In what follows we give a brief overview of linear logic and the Coq proof assistant. We then present our encoding of linear logic, and give examples of its use for two well-known problems: the blocks world scenario, and the Towers of Hanoi problem[1].

1.1 Linear Logic

Linear Logic is a sub-structural logic in the sense of [4] in that it rejects the use of two of the structural rules found in classical logic, specifically weakening and contraction. These rules are presented formally in Figure 1: basically, weakening allows us to have unused hypothesis, while contraction allows us to disregard the number of times a hypothesis is used.

Thus, when we write a deduction in linear logic of the form $\Gamma \vdash A$ we are stating that *all* of the assumptions in Γ are used exactly *once* in the deduction

[1] Computational descriptions of these problems can be found in many programming texts, e.g. [12]

Let Γ be any list of predicates, and let A and C be any predicates; then:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathit{Weakening} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathit{Contraction}$$

Fig. 1. *Weakening and Contraction rules for classical logic.* These classical rules allow us to replicate and delete premises during a proof; their absence is fundamental to linear logic.

---

of A. For this reason linear logic is often described as "resource-sensitive", and we may think of $\Gamma \vdash A$ as representing a process that consumes the resources Γ in the production of A.

This restriction in our use of assumptions gives us a richer power of specification, and is particularly suitable for state-based systems. If we represent the state of a system by some linear predicate, then a linear deduction of the form $\Gamma \vdash A$ represents a transition from the state represented by Γ to one represented by A. For example, if we use $day(X)$ as a predicate to represent the current day, the deduction

$$day(sunday) \vdash day(monday)$$

would seem nonsensical or even contradictory in classical logic, but in linear logic it would represent the transition between these days (thus "removing $day(sunday)$ from the current state, and "replacing" it with $day(monday)$). Indeed, the well-known Curry-Howard isomorphism between constructive logic and functional programming languages can be extended to linear logic in order to deal with the problems of state in functional programming [1, 11].

Clearly this has implications for the operators that can be used in linear logic. The conjunction of two propositions A and B in classical logic, written $A \wedge B$ allows us to deduce either or both of A and B. For example, in classical logic we can prove the following two theorems:

$$\begin{array}{c} A \wedge B, C \\ \vdots \\ A \wedge C \end{array} \qquad\qquad \begin{array}{c} A \wedge B, B \rightarrow D \\ \vdots \\ A \wedge D \end{array}$$

In the first case we have used $A \wedge B$ solely for the production of A; in the second we have used it to produce both A and B.

Linear logic allows us to distinguish between these two uses of conjunction: we write $A \mathbin{\&} B$ if we wish to only use one of either A or B, and we write $A \otimes B$ if we wish to use both.

Similarly classical disjunction, written $A \vee B$, can be established by proving one of A or B, or indeed both. In linear logic we write $A \oplus B$ for the disjunction where just one of A or B has been proven, and use $A \parr B$ for the case where both have been established.

Linear implication now internalises linear deduction, and thus $A \multimap B$ denotes a process whose use will consume and A and produce a B.

In what follows we will concentrate on intuitionistic linear logic (ILL)[2]; the sequent rules for the relevant connectives are presented in Figure 15.

1.2 The Coq Proof Assistant

Our system was developed using the Coq proof assistant[3], which is based on the Calculus of Inductive Constructions [2]. As well as the normal benefits of a proof assistant such as uniformity of notation and verification of type-correctness, Coq also provides three enhancements to ordinary constructive logic:

- Coq implements a higher-order constructive logic, facilitating in particular, the descriptions of object logics within the framework
- Coq has two type hierarchies: `Set` of constructive types, and `Prop` for classical logic. This allows specifications to be developed in ordinary classical logic, and then verified in the same framework against programs written within `Set`.
- Coq supports inductive (and co-inductive) definitions, giving a natural logical extension of the definition-by-cases style of programming found in functional languages

In order to codify ILL using this system we need to be able to introduce a type of linear predicates, provide a syntax for the linear connectives and, most fundamentally, define their operation by encoding their left and right rules of the sequent calculus presentation of these connectives.

Some systems designed specifically for encoding logics such as Isabelle [9] distinguish between the system's own meta logic, and the object logic that this can be used to define. In contrast to this, Coq provides a single homogeneous system with a single built-in concept of deduction, and so our definition of ILL will have to exist as an ordinary datatype within this system. In fact, we base our encoding around the linear consequence operator, encoding it as an ordinary two-place relation between linear predicates.

1.3 Mixing the Logics

Linear logic is perhaps most usefully applied to state-based problems when used along with classical or intuitionistic logic. In particular, while state-specific assertions can be phrased in ILL, it is often useful to be able to express global invariants in classical logic, since it should be possible to use these as often as possible.

Classical intuitionistic predicates may be incorporated into ILL following [5] by marking them with modal operator. Thus we write $!A$ to denote "arbitrarily many A's", and the rules for this operator assert that we are free to use A zero, one or many times as we need.

[2] Since we are concentrating on ILL, the sequent rules have the pleasant property of all containing a single predicate on the right-hand-side, which considerably simplifies the symbolic manipulation of goals

It is one of the particular features of our approach that we have an alternative to the use of such an operator. By encoding ILL as a simple consequence relation within the Coq system, we may freely mix linear assertions pertaining to the state with intuitionistic or classical assertions that are state-invariant. Since the objects that are used in the linear predicates range over Coq datatypes, these may also be constrained by ordinary (intuitionistic or classical) predicates, providing unrestricted use.

The significant benefit here is that all of the existing theories developed for Coq do not need to be changed for use with linear theorems, but can be integrated directly into the proofs. This is particularly useful when dealing with datatypes such as the natural numbers or lists, where re-encoding "linear" versions of the results would provide a significant overhead.

2 Encoding the Sequent Rules

In this section we present some of the Coq code used in setting up our ILL proof system. The code presented here is fragmentary and for illustrative purposes; the full source is available from the authors' web page [10]. As indicated above, setting up the proof system involves two main steps: Defining a type of ILL predicates and their associated connectives, and then defining a consequence operator and the associated sequent rules.

2.1 The Linear Connectives

First we must define a set of linear predicates which we call `ILinProp`; we define this by cases as the smallest set closed under the linear connectives:

```
Inductive ILinProp : Set :=
| Implies : (ILinProp) -> (ILinProp) -> ILinProp
| One : ILinProp
| Plus : (ILinProp) -> (ILinProp) -> ILinProp
| Times : (ILinProp) -> (ILinProp) -> ILinProp
| Top : ILinProp
| With : (ILinProp) -> (ILinProp) -> ILinProp
| Zero : ILinProp
.
```

Since `ILinProp` is now defined as an ordinary Coq type, we can use arbitrary Coq variables as linear propositions and predicates; for example we can declare propositions `A` and `B`, as well as a predicate `N` over the natural numbers as follows:

```
Variable A,B:ILinProp.        (* A and B are linear propositions *)

Variable N:nat -> ILinProp.   (* N is a linear predicate *)
```

As might be expected Coq provides syntax-definition and pretty-printing facilities that allow us to use infix and prefix notation for the operators. For

Operator	Symbol	Syntax in Coq	Unit	Unit's Syntax
Times	$\otimes$	`**`	**1**	`One`
With	&	`&&`	$\top$	`Top`
Plus	$\oplus$	`++`	**0**	`Zero`
Implies	$\multimap$	`-o`		

Fig. 2. *Coq syntax for ILL operators and units.* For reference, this table defines the syntax defined and used in our Coq code to represent the various symbols used in ILL.

example we can augment the grammar rules and give pretty-printing rules to represent the $\otimes$ operator by `**` used infix by defining[3]:

```
Grammar command  command6 :=
  Times [ command5(c1) "**" command6(c2) ] -> [<<(Times c1 c2)>>].
Syntax constr level 6:
  PTimes [<<(Times c1 c2)>>] -> [ c1:L "**" c2:E ].
```

For reference the syntax used in this paper for the ILL operators is given in Figure 2.

2.2 The Linear Consequence Operator

The linear consequence operator is defined inductively as a relation between any list of predicates (the hypotheses) and some other predicate (the conclusion). In our implementation this is a two-place function, giving us a result in Coq's truth domain, `Prop`:

```
Coq < Check LinCons.
LinCons
     : (list ILinProp)->ILinProp->Prop
```

As for the connectives, we can define an infix operator "`|-`" to denote this relation, and give it a suitably low precedence.

The sequent rules can now all be coded individually, using Coq's implication to represent deduction. As an example, the $\multimap_L$ rule of Figure 15 is encoded as:

```
Inductive LinCons : (list ILinProp) -> ILinProp -> Prop :=
(* ... intervening rules omitted ... *)
| ImpliesLeft :
  (A,B,C : ILinProp)(D1,D2 : (list ILinProp))
  ((D1 |- A) -> (D2 ^ `B |- C) -> (D1 ^ D2 ^ `(A -o B)  |- C))
```

[3] This also gives the operator a precedence level of 6, and the grammar rules imposes right associativity

```
(* Here A,B:ILinProp, G,D:(list ILinProp) and P:Prop *)
Axiom example1 : Empty |- A.
Axiom example2 : G |- A.
Axiom example3 : (G |- A) -> (D |- B).
Axiom example4: P -> (G |- A).
```

Fig. 3. *Examples of some sequent-based rules.* These examples demonstrate the general format of ILL rules in Coq. `Axiom` is a keyword in Coq introducing an axiom into the system; the actual ILL elements represented are respectively: an axiom, a deduction rule, a sequent rule, and a deduction rule preconditioned by a classical assumption.

In this rule the variables `D1,D2,A,B` and `C` are universally quantified, list concatenation is denoted by the infix "^" symbol and, as a shorthand, the singleton list containing `B` is written as `‘B`.

In general, an axiom in linear logic is represented as an axiom in Coq, with an empty predicate-list (represented here by the constant `Empty`) to the left of the consequence relation; for example, to state that some predicate A is valid, we can assert something like `example1` of Figure 3.

The standard conditional sequent is represented as a relation between a list of predicates and another predicate, as in `example2`.

A deduction, such as one of the sequent rules or a theorem may be represented by one linear sequent that is conditional on another; thus a rule of the form

$$\frac{G \vdash A}{D \vdash B}\ example3$$

can be represented by using Coq's implication "`->`" for the deduction, as in `example3` of Figure 3

Finally, the combination of linear and ordinary classical assumptions can be represented as in `example4`; here the intuitionistic predicate `P` may share variables with the other predicates, and thus can assert state-invariant properties of any of the data they might refer to.

2.3 Some Simple Tactics

Coq's proof system naturally provides a number of built-in tactics which can be applied in a goal-directed proof; these can be combined using sequencing or selection, represented by the ";" and `Orelse` operators respectively. There are also a number of ways of adding extra functionality to this proof system. One of the most straightforward of these is the definition of *tactic macros*, which allow for parameterised definitions of tactic combinations.

For example, most of the left sequent rules of Figure 15 specify an arbitrary left context, represented by the predicate list Γ. However, in many cases there is just a single assumption to the left of the "`|-`", and these rules are thus inapplicable. To circumvent this, we can define a simple tactic macro `LeftApply`

```
Tactic Definition LeftApply [$myTac] :=
  [<:tactic:<
    ((Apply $myTac) Orelse
    (Apply AddNilFront; Apply $myTac; Rewrite <- app_nil_front))
  >>].

Tactic Definition LinSplit  :=
  [<:tactic:<
    LeftApply TimesLeft; Apply TimesRight; Try (Apply Identity)
  >>].
```

Fig. 4. *Two simple tactic macros in Coq.* Here we define two tactic macros `LeftApply` and `LinSplit`. In the first case the macro is parameterised by an arbitrary tactic `myTac`, and we use `AddNilFront` and `app_nil_front` to handle the addition and deletion of an empty list to the premises. In the second we use Coq's tactic `Try` which will test for applicability before applying the tactic.

in Figure 4 that will try to apply a rule directly and, if this fails, will append an empty list to the premises and try the rule again.

Another example of the use of tactic macros is the definition of `LinSplit` in Figure 4. This tactic macro is defined to deal with deductions of the form $(A \otimes B) \vdash (C \otimes D)$, splitting it into two sub-proofs, $A \vdash C$ and $B \vdash D$. This is a relatively common situation, since the state of a system in ILL is often represented by a series of linear predicates joined by the $\otimes$ operator.

We note that in each case it is up to the user to apply the appropriate tactic macro; neither is premised by a syntactic precondition that would test its viability in the context of the current premises and goal. Thus these macros are relatively unsophisticated features that serve mainly to eliminate some of the repetition from a user-directed proof.

3 A Small Example: The Blocks World

The blocks world scenario is one of the classic simple examples of problem solving, commonly used in the field of Artificial Intelligence. Here we use this example not for its planning aspects, but as an example of a state-based system where the goals and transitions can be represented in ILL.

The blocks world assumes the existence of some finite set of blocks, as well as a robot arm which can be used to move them. The actions of the arm typically involve stacking and unstacking the blocks so that, in the simplest case, we may regard this as a problem in just two dimensions.

To represent this in ILL we must first decide on a set of predicates to represent the state of the system. These are fairly straightforward, and easily represented in Coq, as shown in Figure 5. We have:

```
Variable on    : Block -> Block -> ILinProp.
Variable table : Block -> ILinProp.
Variable clear : Block -> ILinProp.
Variable holds : Block -> ILinProp.
Variable empty : ILinProp.
```

Fig. 5. *Coq code for the predicates representing the state in the blocks world.* Here we declare the five basic predicates that will represent the state, and give their types. The first three predicates are used to describe the relationships of the blocks to each other, the final two are used to describe the status of the robot arm

- A relation `on` between blocks, where `(on x y)` means that block `x` is on block `y`
- Unary predicates over blocks: `(table x)` indicates that block `x` is on the table (i.e. there is no block underneath it), and `clear x` indicates that there is no block on top of `x`
- Two predicates relating to the status of the robot arm: `(holds x)` is true if the arm is holding block `x`, whereas `empty` is true if the arm holds no block

The second step is to define the set of allowable actions. Each such action is defined as a linear deduction, where we specify the pre- and post-states of the action using the above predicates. The two basic actions relate to the robot arm picking up and putting down a single block, and are given in Figure 6

The action `get` says that if the arm was empty and block `x` had nothing on top of it, then after performing the action the arm will hold `x`, and either `x` had been on the table, or it had been on some other block `y`, in which case we add the information that `y` is now clear to the state.

The action `put` states that starting from a state where the arm is holding a block `x` we can move to a state where the arm is empty, there is nothing on top of `x`, and we have either placed it on the table, or on top of some other block which was previously clear.

We note in both cases the partitioning of the post-state into those parts which always apply, and those which involve a choice (represented here by the $\&$ operator). In particular, when the choice is not universally applicable, we can use linear implication to suitably pre-condition it. The unit $\mathbf{1}$, as the identity for $\otimes$, is used with linear implication to denote a process which consumes resources but does not produce anything.

We may establish special cases of each as *lemmas* in Coq: that is, they can be derived from the axioms `get` and `put`. For example, it is useful to have a version of `get` referring solely to the case where block `x` was on some other block `y`. As an example of goal-directed proof in Coq we present the proof script for this lemma in Figure 8, and the corresponding deduction in Figure 9. As can be seen from a comparison of these figures there is a close correspondence, with

```
(* The basic actions *)
Axiom get :
  (x,y:Block)
  (`(empty ** (clear x))
|- (holds x) ** (((table x) -o One) && ((on x y) -o (clear y)))).

Axiom put :
  (x,y:Block)
  (`(holds x)
|- empty ** (clear x) **  ((table x) && ((clear y) -o (on x y)))).
```

Fig. 6. *Coq code for the basic actions allowed in the blocks world.* The actions `get` and `put` refer to the robot arm picking up and putting down a block.

```
Lemma gettb :
  (x:Block)
  (`(empty ** (clear x) ** (table x))  |- (holds x)).
Lemma puton :
  (x,y:Block)
  (`((holds x) ** (clear y)) |- empty ** (clear x) ** (on x y)).
Lemma puttb :
  (x:Block)
  (`(holds x)  |- empty ** (clear x) ** (table x)).
```

Fig. 7. *Specific Cases of* `put` *and* `get`. To faciliate the construction our proofs we define some specific cases of the uses of `put` and `get`. Each of these can be proved within the system, in a manner analogous to Figure 8

some extra manipulation being required in the Coq proof to rearrange to goal into a format suitable for the application of some rules.

The other special cases we can establish are shown in Figure 7.

As an example of the use of these actions, and of the structure of a state-based proof in our encoding of ILL, consider the two blocks-world scenarios shown in Figure 10. Clearly the second is attainable from the first by putting block `b` on the table, and then putting `a` on top of it.

In fact, we may state this as a theorem in Coq:

```
Theorem SwapAB :
  (`(empty ** (clear b) ** (on b a) ** (table a) ** (table c) ** (clear c))
|-
  (on a b) ** Top).
```

Here we use $\top$ in our goal as a sink for any unused predicates, since we know that $\Gamma \vdash \top$ for any Γ.

```
Lemma geton :
  (x,y:Block)
  (`(empty ** (clear x) ** (on x y))  |- (holds x) ** (clear y)).
Proof.
  Intros.
  LeftApply TimesAssocR. LeftApply TimesComm. LeftApply TimesLeft.
  LinCut (holds x)**((table x)-o One)&&((on x y)-o(clear y)).
  Apply get.
  LeftApply TimesLeft.
  LeftApply ExchList.
  Apply TimesRight.
  Apply Identity.  (* to (holds x) *)
  LeftApply WithLeft2.
  Apply AddNilFront. LeftApply ExchList. LeftApply ImpliesLeft.
  Apply Identity.  (* to (on x y) *)
  Simpl; Apply Identity.  (* to (clear y) *)
Qed.
```

Fig. 8. *The statement and proof of lemma* `geton` *in Coq.* This lemma establishes a special case of the use of predicate `get` described earlier, where the block being picked up had been on some other block.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\mathit{holds}\ x) \vdash (\mathit{holds}\ x)}\ \mathit{Id.} \qquad \dfrac{\dfrac{\dfrac{}{(\mathit{on}\ x\ y) \vdash (\mathit{on}\ x\ y)}\ \mathit{Id.} \quad \dfrac{}{(\mathit{clear}\ y) \vdash (\mathit{clear}\ y)}\ \mathit{Id.}}{(\mathit{on}\ x\ y), ((\mathit{on}\ x\ y) \multimap (\mathit{clear}\ y)) \vdash (\mathit{clear}\ y)}\ \multimap_L}{(\mathit{on}\ x\ y), \left(\begin{array}{l}((\mathit{table}\ x) \multimap \mathbf{1})\& \\ ((\mathit{on}\ x\ y) \multimap (\mathit{clear}\ y))\end{array}\right) \vdash (\mathit{clear}\ y)}\ \&_{L_2}}{(\mathit{holds}\ x), (\mathit{on}\ x\ y), \left(\begin{array}{l}((\mathit{table}\ x) \multimap \mathbf{1})\& \\ ((\mathit{on}\ x\ y) \multimap (\mathit{clear}\ y))\end{array}\right) \vdash (\mathit{holds}\ x) \otimes (\mathit{clear}\ y)}\ \otimes_R}{(\mathit{on}\ x\ y), (\mathit{holds}\ x), \left(\begin{array}{l}((\mathit{table}\ x) \multimap \mathbf{1})\& \\ ((\mathit{on}\ x\ y) \multimap (\mathit{clear}\ y))\end{array}\right) \vdash (\mathit{holds}\ x) \otimes (\mathit{clear}\ y)}\ \mathit{Exch}}{(\mathit{on}\ x\ y), (\mathit{holds}\ x) \otimes \left(\begin{array}{l}((\mathit{table}\ x) \multimap \mathbf{1})\& \\ ((\mathit{on}\ x\ y) \multimap (\mathit{clear}\ y))\end{array}\right) \vdash (\mathit{holds}\ x) \otimes (\mathit{clear}\ y)}\ \otimes_L}{\mathit{empty} \otimes (\mathit{clear}\ x), (\mathit{on}\ x\ y) \vdash (\mathit{holds}\ x) \otimes (\mathit{clear}\ y)}\ \mathit{Cut}(\mathit{get}\ x\ y)}{\mathit{empty} \otimes (\mathit{clear}\ x) \otimes (\mathit{on}\ x\ y) \vdash (\mathit{holds}\ x) \otimes (\mathit{clear}\ y)}\ \otimes_L
$$

Fig. 9. *A proof in linear logic of the* `geton` *lemma.* This proof is presented for comparison with the corresponding Coq proof script, shown in Figure 8. The crucial $\mathit{Cut}(\mathit{get}\ x\ y)$ proof step here corresponds to the `LinCut` on line 3 of the Coq script above.

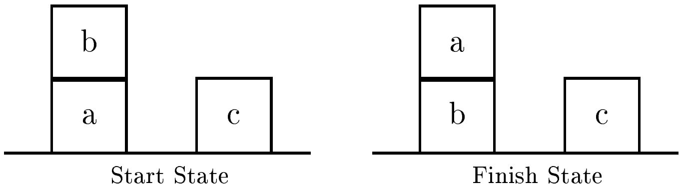

Fig. 10. *Two Blocks World Scenarios.* Our goal is to prove that there is a sequence of actions that will reverse the ordering of blocks `a` and `b`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{empty} \otimes (\mathit{clear}\ b) \otimes (\mathit{on}\ b\ a) \otimes (\mathit{table}\ a) \otimes (\mathit{table}\ c) \otimes (\mathit{clear}\ c)}{(\mathit{empty}) \otimes (\mathit{clear}\ b) \otimes (\mathit{on}\ b\ a) \otimes (\mathit{table}\ a) \otimes \top}\,(\top_R)}{(\mathit{holds}\ b) \otimes (\mathit{clear}\ a) \otimes (\mathit{table}\ a) \otimes \top}\,(\mathit{geton}\ b\ a)}{\mathit{empty} \otimes (\mathit{table}\ b) \otimes (\mathit{clear}\ b) \otimes (\mathit{clear}\ a) \otimes (\mathit{table}\ a) \otimes \top}\,(\mathit{puttb}\ b)}{(\mathit{table}\ b) \otimes (\mathit{clear}\ b) \otimes (\mathit{holds}\ a) \otimes \top}\,(\mathit{gettb}\ a)}{(\mathit{table}\ b) \otimes \mathit{empty} \otimes (\mathit{on}\ a\ b) \otimes (\mathit{clear}\ a) \otimes \top}\,(\mathit{puton}\ a\ b)}{(\mathit{on}\ a\ b) \otimes \top}\,(\top_R)$$

Fig. 11. *An outline of the proof.* Here we give an outline of the proof that we can move between the scenarios of Figure 10. The proof is basically a list of the states that we go through as the four actions are applied.

The proof proceeds as outlined in Figure 11. To code this in Coq we must also follow these steps but, there will also be some extra manipulation to be done. This relates specifically to managing the predicates that describe the current states, in particular:

- Separating them into two groups: those that changed with the current action and those that did not. Our tactic macro `LinSplit` of Figure 4 can then be used to eliminate the predicates that did not change for this action
- Re-arranging the remaining predicates in the same order as the preconditions of the relevant action

Such work is usually implicit in handwritten proofs, but in a proof assistant they add an unpleasant overhead to the task of theorem proving. The approach taken in our proof was to wrap each of the four basic actions in a lemma which performed the necessary state-manipulations in addition to applying the action.

The structure of the proof itself then mirrored that of Figure 11 quite closely. Since Coq works in a goal directed fashion, a direct proof would involve applying the actions in reverse order. However, in practice it was far simpler to proceed in a step-by-step fashion *forwards* using the *Cut* rule to establish a new state for each step.

This would seem to be a general feature of this kind of proof using our encoding of ILL. Consider the general case of a goal of the form $S_0 \vdash S_n$, where our proof should involve constructing the intermediate states $S_1, \ldots, S_{n-1}$. If there are more than a few steps it will be likely that the final state S_n will be quite different from the initial state S_0, and hence a goal-directed proof must concentrate almost exclusively on manipulating S_n. When we use the *Cut* rule however, we generate two subgoals, thus:

$$\frac{S_0 \vdash S_1 \quad S_1 \vdash S_n}{S_0 \vdash S_n}\ Cut(S_1)$$

In our proof of $S_0 \vdash S_1$ we will be able to manipulate both S_0 and S_1 to good effect, since they will only differ by a single atomic action. Our proof of $S_1 \vdash S_n$ can then proceed by using the *Cut* rule with S_2. As well as making the individual steps much simpler, this will also make the structure of the proof much clearer, and advantage both when constructing and when reading the proof.

4 An Example with Induction: The Towers of Hanoi

The Towers of Hanoi is one of the classic examples of the use of recursion in programming. The problem space consists of a set of discs of various sizes which can fit onto any one of three pegs. Initially all the discs are stacked in increasing order of size on the first peg. The goal is to move the discs to the third peg, but only moving one disc at a time, and never placing a larger disc on a smaller one.

This problem is easily formulated as a recursive function, since the task of moving n discs from a source to a destination peg, using some middle peg as intermediate storage, can be broken down into three subtasks: moving $n-1$ discs to the middle peg, moving the remaining disc to the destination peg, and moving the $n-1$ discs from the middle to the destination peg.

We can encode this problem in our implementation of ILL in a manner analogous to the blocks world example, where in this instance the state is represented by a list of the discs on each peg. However, we can go further than the typical computational solution in that we can prove the correctness of our algorithm - expressed as a sequence of proof steps - as we develop it. In particular, we can show that the invariant of never having a larger disc on top of a smaller one is maintained.

This proof is also of interest since it demonstrates the interaction of our encoding of ILL with Coq's built-in predicates and datatypes. While the discs may move from peg to peg, the information regarding their size, and thus constraining which disc may be placed on which, is state-invariant, and so may be captured with a classical predicate. Also, the proof will proceed by induction over the number of discs to be moved, and we will utilise Coq's built in `list` datatype, and its associated principles of induction to achieve this.

To formulate the problem in Coq we assume the existence of sets `Pole` and `Disc`, as well as a relationship `onPole` telling us what discs are on a pole, and a `smaller` ordering on discs, which we can generalise to a `canTxfrTo` relation between one disc and a list of discs. Next we must assume that it is possible to move a single disc, and prove that this scales up to n discs. This assumption

```
Axiom Txfr :
  (p1,p2:Pole) (f:Disc)(fs,ts:(list Disc))
  (canTxfrTo f ts) ->
  (`((onPole p1 (cons f fs))**(onPole p2 ts))
|-
  ((onPole p1 fs) ** (onPole p2 (cons f ts)))).
```

Fig. 12. *An axiom stating that we can move one disc.* Here we are transferring the disc `f` from pole `p1` to pole `p2`. The state is represented by the linear predicates, with the classical predicate `canTxfrTo` maintaining the global invariant that larger discs cannot be placed on smaller ones.

```
Lemma Move :
  (dTop,dBot:(list Disc)) (p1,p2,p3:Pole) (d2,d3:(list Disc))
  (ordered dTop) ->
  (canMoveTo dTop dBot) -> (canMoveTo dTop d2) -> (canMoveTo dTop d3) ->
  (`((onPole p1 (dTop^dBot))**(onPole p2 d2)**(onPole p3 d3))
|-
  (onPole p1 dBot) ** (onPole p2 d2) ** (onPole p3 (dTop^d3))).
```

Fig. 13. *A lemma that we can move any number of discs.* Here `dTop` represents the discs being moved from pole `p1` to pole `p3`, using `p2` for intermediate storage. None of the discs in lists `dBot`, `d2` or `d3` are moved in this operation.

is represented as an axiom `Txfr` of Figure 12, indicating that the appropriate change of state is valid.

To show that this can scale up to n discs, we must prove lemma `Move` of Figure 13 that allows an arbitrary list of discs to be moved from one peg to another, provided that the global invariant prohibiting us from putting a larger disc on a smaller one is maintained. Since we wish to use this repeatedly, we must allow for the possibility that the middle and destination pegs already contain some discs, and that we do not wish to move all the discs from the source peg.

This rather innocent requirement imposes a considerable amount of extra work on constructing the proof as opposed to writing the equivalent program since, when allowing the middle and destination pegs to be non-empty, we must also ensure that this does not prohibit us from moving discs from the source peg to them. Thus, in our statement of the lemma in Figure 13, we also must include several classical preconditions covering the relationship between the discs. These conditions are represented by the predicate `canMoveTo` which is a generalisation of the `smaller` relationship to one between lists of discs.

Base Case: `dTop` is $Empty$, the empty list

$$\frac{(onPole\ p1\ (Empty^{\wedge}dBot)) \otimes (onPole\ p2\ d2) \otimes (onPole\ p3\ d3)}{(onPole\ p1\ dBot) \otimes (onPole\ p2\ d2) \otimes (onPole\ p3\ (Empty^{\wedge}d3))}\ Id.$$

Inductive Case: `dTop` is $(ds^{\wedge}d)$

$$\begin{array}{c}
(onPole\ p1\ ((ds^{\wedge}d)^{\wedge}dBot)) \otimes (onPole\ p2\ d2) \otimes (onPole\ p3\ d3) \\
\vdots\ (Move_{n-1}\ p1\ p2) \\
(onPole\ p1\ (d^{\wedge}dBot)) \otimes (onPole\ p2\ (ds^{\wedge}d2)) \otimes (onPole\ p3\ d3) \\
\vdots\ (Txfr\ p1\ p3) \\
(onPole\ p1\ dBot) \otimes (onPole\ p2\ (ds^{\wedge}d2)) \otimes (onPole\ p3\ (d^{\wedge}d3)) \\
\vdots\ (Move_{n-1}\ p2\ p3) \\
(onPole\ p1\ dBot) \otimes (onPole\ p2\ d2) \otimes (onPole\ p3\ ((ds^{\wedge}d)^{\wedge}d3))
\end{array}$$

Fig. 14. *An outline of the proof for the Towers of Hanoi.* In the inductive case we represent the list of discs to be moved by the topmost $n-1$ discs `ds`, and the next disc `d`. The remaining discs in `dBot`, `d2` and `d3` are not moved. We have simplified the proof considerably by omitting the conditions relating to the classical invariants.

The proof proceeds by induction over the size of the list of discs `dTop` that is to be moved[4], and the general structure follows the state-based "cut-and-prove" strategy used in the blocks world example. A rough outline of the proof is given in Figure 14.

It should be noted that each of the two inductive uses of `Move` also generates a requirement to satisfy the four classical invariants in the proof, and that this considerably increases the size of the proof above its purely "computational" counterpart.

Once this lemma has been proved, the original problem is now just a special case of the `Move` lemma:

```
Theorem Hanoi :
  (p1,p2,p3:Pole) (ds:(list Disc))
  (ordered ds) ->
  (`((onPole p1 ds) ** (empty p2) ** (empty p3))
|-
  (empty p1) ** (empty p2) ** (onPole p3 ds)).
```

5 Conclusions and Further Work

In this paper we have given an overview of our encoding of ILL in the Coq proof assistant, based around the representation of the linear consequence operator as a

[4] In fact, it proceeds by reverse induction, since we wish to split a list of length n into the first $n-1$ discs and the remaining bottom disc.

two-place predicate over linear-logic terms. We have given two simple examples of the use of the system: a scenario from the blocks world to demonstrate a state-based proof, and the towers of Hanoi problem to demonstrate the use of an inductive proof.

The encoding itself was relatively straightforward, and much of the initial work with the system was concerned with building up a library of auxiliary lemmas relating to properties such as the associativity and commutativity of operators.

We believe the main utility of this work is to demonstrate the feasibility of using Coq as a proof system for linear logic, with the particular advantage of being able to integrate the existing Coq datatypes, and classical assumptions into the system. Even for the small examples presented here this approach showed benefits, particularly in terms of the proofs relating to lists used in the Towers of Hanoi example.

One aspect not examined here was the possibility of automating the proofs. Our work to date indicates that this would have to have at least two components: a context-handling system to deal with the order and associativity between the predicates in the hypothesis list, and a general proof strategy, such as found in a linear-logic based programming language. Coq provides mechanisms to allow the construction of automated theorem proving, and we believe that the work done to date provides a promising basis for this next step.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[3] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual. rapport technique 177, INRIA, July 1995.

[4] Kosta Dosen and Peter Schroder-Heister, editors. *Substructural Logics*. Studies in Logic and Computation. Oxford University Press, 1993.

[5] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[6] James Harland and David Pym. A note on the implementation and applications of linear logic programming languages. In G. Gupta, editor, *Proceedings of the Seventeenth Annual Computer Science Conference, Christchurch*, pages 647–658, January 1994.

[7] Joshua S. Hodas. Lolli: An extension of λprolog with linear context management. In D. Miller, editor, *Workshop on the λProlog Programming Language*, pages 159–168, Philadelphia, Pennsylvania, August 1992.

[8] Sara Kalvala and Valeria de Paiva. Mechanizing linear logic in Isabelle. In L.C. Paulson, editor, *Proceedings of the Isabelle Users Workshop*, Cambridge, England, September 1995.

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

[10] James Power. WWW home page. http://www.cs.may.ie/~jpower, May 1999.

[11] Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North-Holland.

[12] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, 1984.

A Sequent Rules for Intuitionistic Linear Logic

$$\frac{}{A \vdash A}\ \mathit{Identity} \qquad \frac{\Gamma \vdash B \quad \Delta, B \vdash C}{\Gamma, \Delta \vdash C}\ \mathit{Cut}(B) \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ \mathit{Exchange}$$

$$\frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\ \multimap_L \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \multimap_R$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ \otimes_L \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ \otimes_R$$

$$\frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C}\ \&_{L_1} \ \frac{\Gamma, B \vdash C}{\Gamma, A \,\&\, B \vdash C}\ \&_{L_2} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&_R$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\ \oplus_L \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \oplus_{R_1} \ \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \oplus_{R_2}$$

$$\frac{\Gamma \vdash C}{\Gamma, \mathbf{1} \vdash C}\ \mathbf{1}_L \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}_R$$

$$\frac{}{\Gamma, \mathbf{0} \vdash C}\ \mathbf{0}_L \qquad \frac{}{\Gamma \vdash \top}\ \top_R$$

Fig. 15. *Sequent Rules for Intuitionistic Linear Logic.* For intuitionistic linear logic we restrict our attention to the three structural riules, linear implication and the operators $\otimes$, $\&$ and $\oplus$ and their units.