Mechanizing Linear Logic in Isabelle

Sara Kalvala and Valeria de Paiva
Computer Laboratory, University of Cambridge
{sk,vcvp}@cl.cam.ac.uk

August 1995

Abstract

We present an implementation of propositional Linear Logic in the Isabelle proof system. Previous implementations of Linear Logic have often been geared to studies of efficiency of proof search; ours provides an environment for users to describe problems and to develop proofs interactively. Isabelle provides many facilities for developing a useful specification and verification environment from the basic formulation of logical systems. We briefly introduce the logic and Isabelle, and discuss some of the issues in automatic theorem proving in Linear Logic. We then describe the system we have built for executing proofs in Isabelle, and illustrate its use.

1 Introduction

This paper describes a prover for propositional Linear Logic [5]. There already exist other theorem provers for Linear Logic; developers of these provers have mostly concentrated on automatic theorem-proving and often have been driven by a concern with timing efficiency of proof search. These systems have been built directly in programming languages that allow the coding of fast procedures. Our concern, on the other hand, has been to supply the community involved with Linear Logic with a user-friendly proof system for interactively searching for proofs, and which provides many facilities for examining proofs and building applications using the logic.

Our implementation is a straightforward application of the Isabelle methodology [13]. Isabelle is a *generic* theorem prover: it provides an infra-structure onto which different logics can be added, and theorem-proving environments can be quickly developed and used, without compromising the rigour and security of proofs performed using it. We believe that the resulting system will be helpful to anyone who is trying to prove a theorem in Linear Logic by hand. The concern has been less with automation, and more with providing users with an environment where the logic can be studied without the need for coding procedures or understanding the low-level implementation details of the prover. But saying that, we have achieved a respectable amount of automation in proof search, even without trying very hard.

We believe this implementation illustrates the usefulness of the "logical framework" approach to building theorem provers: with very little effort involved, we have developed a proof system that is easy to understand and extend. This paper can therefore be seen as a case study in the use of Isabelle.

The paper is organised as follows. We first recall the main features of propositional Linear Logic and discuss some of the problems in mechanizing it. Then we recall the main features of Isabelle. In the next section we present our implementation and discuss its properties as well as our experience using it. Finally we discuss some related work, draw some conclusions and point at some future work.

2 Linear Logic

Traditionally, formulae in the context are considered to be permanent: once a formula is placed in the context (either as an assumption or as a result of previous proof steps) it can be used as many times as desired in building a proof. The basic idea of Linear Logic is that formulae are thought of as resources that can be produced as well as consumed in the construction of a proof. In their simplest form, each time a formula is used, it ceases to exist for further use. This property is achieved (in a sequent calculus formulation of the logic) by the removal of the usual rules of contraction and weakening. One consequence of this removal is that it leaves undefined the context propagation in rules for logical connectives; thus we have two versions of all logical connectives: the so-called multiplicatives and the additives.

The conventional use of formulae is restored with the use of *exponential* operators, modalities that allow a formula to be re-used as many times as needed. This separation between consumable and non-consumable values, and the associated power to pinpoint *where* in a proof a formula is used, justifies the relevance of a linear view of logic. Exponentials allow us to recover the full expressivity of intuitionistic logic.

In this paper, we will focus on the Intuitionistic fragment of Linear Logic (ILL). We recall the sequent-calculus presentation of ILL in Figure 1, mostly to establish notation. The logic consists of the multiplicative operators (the tensor ($\otimes$) and the linear implication ($-\circ$)), additive operators (conjunction (&) and disjunction ($\oplus$)), and the exponential (!).

Other formulations of ILL are also available, for example the natural-deduction style of rules [2]. The natural deduction in sequent form formulation of ILL also uses sequents, but the left-hand side of the sequents are only used to store dependencies. Thus, there are no more 'left' and 'right' rules, as combinators only appear on the right-hand side. Instead, rules are of the 'introduction' and 'elimination' form, in typical natural deduction style.

There are several features of ILL that make it an interesting object of study. The mapping from formulae in Intuitionistic Logic (IL) to ones in Linear Logic (which will by construction be in the intuitionistic fragment of LL) is known as the Girard translation [4], the core part of which is the decomposition of the usual implication $A \to B$ into the linear form $!A \multimap B$. Another feature of ILL is that the correspondence between proofs and λ-calculus terms has been well studied in this fragment.

Classical Linear Logic (CLL, or simply LL) is characterized by an involutive negation, which shares many of the desirable properties of the negation in Classical Logic. However, as explained by Girard, the linearity of the logic makes it semantically constructive [4]. The study of CLL is therefore interesting in its own right, and mechanizations of CLL could be useful proof tools. In fact, several researchers have presented us with theorem provers for this version, as described in Section 3.

Linear Logic has captured the attention of many researchers. Much of the research in LL has been concerned with the study of the logic itself, and the fact that it provides a medium to study subtle characteristics of proof theory. The 'essence' of a particular proof can be captured more precisely when the proof itself has not been obtained by just adding more and more information to the context. Proofs in LL isolate what is ultimately needed. In particular, if there is a proof in IL, there is a corresponding proof in ILL, which is more informative than the former in many ways.

Among other applications, LL provides a basis for modeling functional programming and logic programming, and can be used as a platform for reasoning about concurrency. Many people have claimed the adequacy of Linear Logic for studying state-oriented programming and non-monotonicity [1]. It is to be expected that some of these applications will be more easily achieved by the existence of a robust and easy-to-use mechanization of the logic.

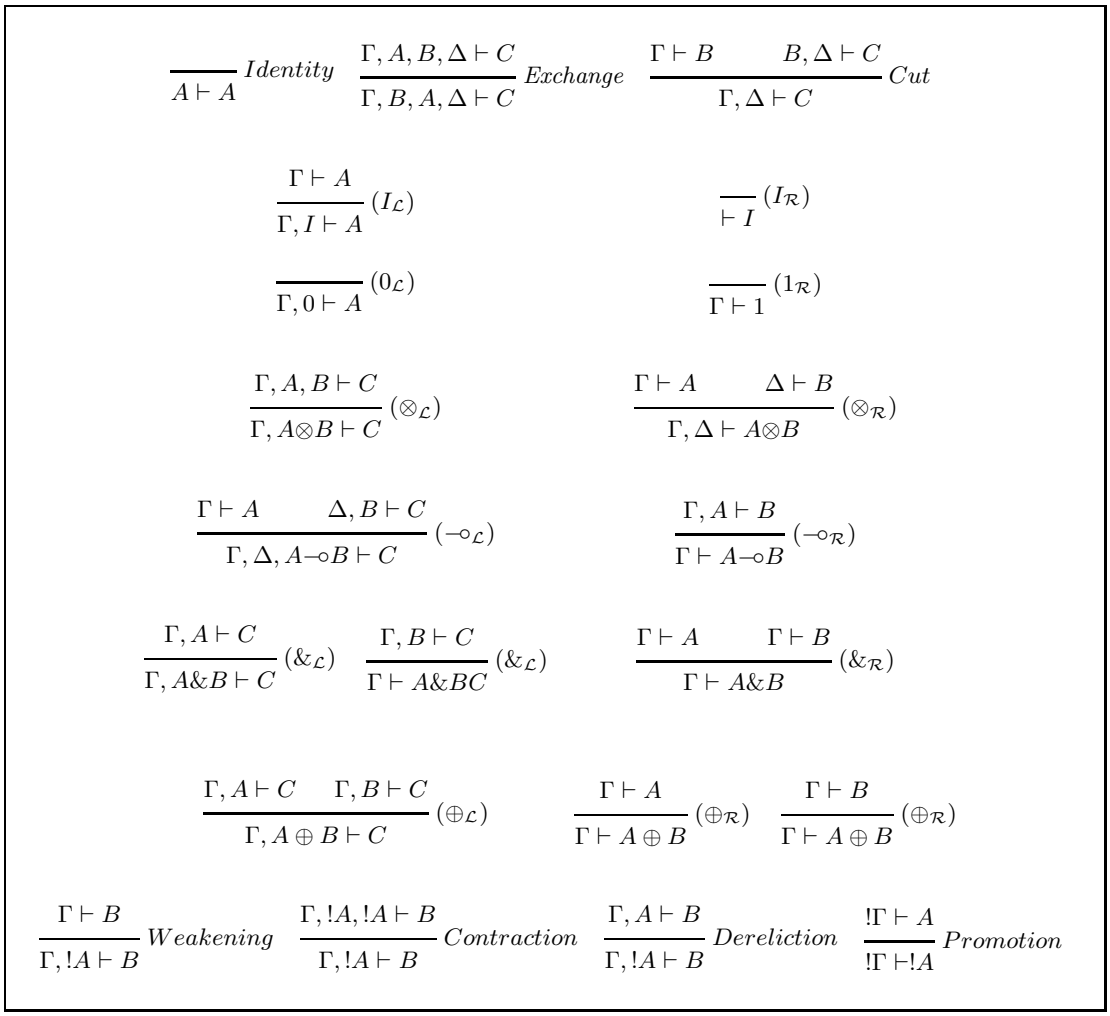

Figure 1: Intuitionistic Linear Logic

3 Previous efforts at mechanizing linear proofs

There have been some attempts to automate the process of proof construction in Linear Logic. In this section we describe some of the results.

The connection between proof theory and logic programming is described by Hodas and Miller [7], where they show the applications of a logic programming version of a fragment of LL, and address the problem of splitting the context in the goal-directed use of LL. A further extension of this work [12] describes a meta-logic in which the usual operators of λ-Prolog coexist with linear operators as well as primitives for describing concurrency. Linear logic is part of the *meta*-logic; it is also often useful to have LL at the *object* level.

A well-known mechanization of Linear Logic is the one developed by Tammet [14]. He considers Classical LL, which allows him to use one-sided sequents. The main advantages of using one-sided sequents is that one needs a smaller set of rules, and many equivalences can be exploited. His prover is purpose-built, being implemented in Scheme (a variant of Lisp). Proofs are found in both bottom-up (what we call backward) or top-down (forward) ways. He makes use of the programming language and the concrete implementation of logical terms, to obtain a very efficient search algorithm.

Lincoln and Shankar have recently described another prover for classical first-order LL [11]. They have also exploited modifications to the form of sequents and, by expanding the data structures to carry lists of free variables and substitutions, have been able to deal with the quantified logic, through a Prolog implementation. They have also performed an extended timing analysis of the resulting implementation. Galmiche and Perrier have studied automation of proof search by using the permutativity in the order of application of rules in a sequent calculus proof [3]. These systems adapt the rules of LL to make the proof search more tractable. Particularly, they exploit features of *classical* LL to make these changes, therefore making it difficult to isolate the intuitionistic part of the logic.

We have decided to step back a little, and question the emphasis these systems have taken with fast and automatic proof search. Mechanized proof systems are often equated with automated provers: one simply formulates the background information and a supposition in the language accepted by the program, and the program returns, in some finite amount of time, an answer of 'yes' or 'no' as to the provability of the supposition. There is however another alternative outlook in the theorem-proving community, emphasizing the development of *proof checkers*. With this approach, the creative aspects of the proof process are in the control of a person rather than a program, by allowing user interaction if and whenever necessary. The proof system just executes proof steps supplied by the user while maintaining logical consistency.

The accepted fact that users have to do more things means that there must be better support for their activity—often in the form of a *tactic*-based interface. User control does not mean that the user must guide the checker through each and every step: automation of large parts of proof effort is achievable. However, often a proof depends on the human guide to be continued, and in a proof-checking paradigm this human assistance can be given in a straightforward manner. We believe a tool with this philosophy will be useful to the community interested in Linear Logic, and our system is particularly geared towards the proof checking paradigm. We hope the rest of this paper illustrates this aspect of our proof system.

4 Isabelle

Isabelle is a general purpose proof system, developed by Paulson at the University of Cambridge [13]. It is a representative of the so-called LCF family of provers, after a prover developed in the late 70's [6] and now virtually extinct, but which has spawned several descendants such as HOL and Nuprl, as well as the ML programming language.

In general, these proof systems, as well as other ones in widespread use (such as Boyer and Moore's system, PVS, etc.), are characterized by embodying one particular logic. Other sets of theorems and inferences representing other logics can be embedded, but they are directly interpreted in the underlying logic, and details of the particular logic can intrude subtly in the reasoning process.

Isabelle, on the other hand, is designed to allows users to supply their own object-level logic to work with, in a completely separate level from the workings of the 'meta-logic' underlying the representation and manipulation of terms in the object logic. Systems with this philosophy are known collectively as *logical frameworks*. While the other logical frameworks have been tailored to be suitable for reasoning *about* the object logic, Isabelle is very appropriate for reasoning *within* the object logic, not only because of the particular meta-logic used, but also because of the tools which make up the environment provided by Isabelle.

Isabelle uses the strong type system of the underlying programming language (Standard ML) to restrict the mechanisms for creation of *theorems*. Inference rules are theorems with (one or more) premises and a conclusion. Typically, theorems in Isabelle can be *axioms* or proved theorems. Axioms are stated directly, and accepted without proof. Other theorems must be derived from combinations of existing theorems, and discovering the way of combining them is the proof construction process.

Proofs are often attempted in a backward fashion: a goal is supplied by the user, and then successively rules are applied to the current goals to reduce them to simpler and simpler forms, until they correspond to existing theorems. In other words, suppose the aim is to prove goal `"C"`. If there exists a rule `"A ⇒ C"`, then it is enough to prove the goal `"A"`, as the theorem `"C"` is derived by combining the theorem `"A"` with the rule `"A ⇒ C"`. Conversely, the same theorem can be obtained by forward proof: obtaining theorem `"A"` and resolving it with the antecedent of the rule `"A ⇒ C"`. However, in practice the more complex the eventual proof is, the easier (relatively) it is to find it in a backward fashion rather than forward.

To allow theorems to be combined, Isabelle makes use of higher-order matching–a polymorphic version of Huet's procedure [8]. Terms themselves are represented as hereditary Harrop formulae. In practice, this level of detail is hidden from the user: axioms of the logic and goals for theorem-proving can be entered simply as strings, and any theorem is always pretty-printed. Resolution is handled by tactics, that leave as many of the variables free as possible, and that allow backtracking over instantiations by maintaining a (lazy) list of all unifications.

As a system for easy construction of specialized provers, Isabelle has several facilities for adding parsing and pretty-printing for object logics. New constants can be given mixfix notation, while more subtle (but optional) syntax features can be added in the Standard ML. Pattern matching is available, and in fact we use it extensively in our work to allow easy manipulation of sequents.

Rules need not be applied one by one by the user: automated procedures can take a set of rules and apply them successively and in different combinations until the goal is solved. Several rules could be applicable at any one point, or with several different instantiations for free variables. Proof search procedures keep all the information necessary to backtrack over reachable proof states. Several search algorithms are available in the system; in most cases all a user needs to do to invoke them is to decide which set of rules should be used to generate new proof states. Also, proof search tactics can be made to leave the proof state in a simpler condition, from which then the user can guide the proof until the goal is solved.

5 Our prover for ILL

In our endeavor to produce a proof systems for Linear Logic, we were guided by several principles:

- We wanted to use Isabelle: it seems sensible to use an "off-the-shelf" package. There is already some work on λ-Prolog, which provides similar use of unification, so developing a prover in Isabelle would provide a basis for comparison between the different systems.
- We wanted to achieve as much as possible within Isabelle's top level of interaction, and reduce the amount of specialized coding. This seems to be the way most users would like to use provers: not by learning how to program in (in this case) Standard ML but by proving theorems or changing primitive rules of the logic and examining the use of these in proving other theorems.
- Our interest is not in complete automation: we believe that, in practice, any user would be happy to interact with the system in short periods and get a proof completed reasonably quickly rather than have the system chug away for an hour. As such, we were interested in 'graceful failure' (which means that even if automated proof search fails the user can still learn something from the search) and the possibility of 'macro-stepping' through a proof, where the system speeds the proof search at tedious parts and the user guides the system in important parts of the proof.

Based on the ideas above, ILL has been coded into Isabelle. The support for sequents is based on the existing formalization of a variant of Gentzen's sequent-calculus (LK), due to Philippe de Groote. The declaration of linear operators and the addition of the rules shown in Figure 1 is straightforward. One trivial change to the rules is that, to allow the use of pattern matching, a

sequent such as $A \multimap B, \Gamma \vdash C$ must in fact be represented as $\Gamma, A \multimap B, \Delta \vdash C$, as in the first form it is implicit that the *first* formula in the sequent is the only one to which this rule must apply, which is not really intended.

A technical problem arises with the rule for (*Promotion*), where the notation $!\Gamma$ means that the operator ! is applied to *each* element of Γ. This is solved by representing the logical rule of (*Promotion*) as an inductive collection of three Isabelle rules, which together check that *each* item of the context is exponentiated. The three Isabelle rules can be written as follows:

$$\frac{\Gamma \vdash A}{\Gamma \parallel \quad \vdash A}\, Promotion1 \qquad \frac{!A, \Gamma \parallel \Delta \vdash A}{\Gamma \parallel !A, \Delta \vdash A}\, Promotion2 \qquad \frac{\parallel \Delta \vdash A}{\Delta \vdash !A}\, Promotion3$$

In the above, the $\parallel$ symbol represents a transient way of transfering items in the context from one part of the context to the other, where only items checked to be exponentiated are held. With this modification in place, we have obtained a complete mechanization of ILL.

In terms of proof search automation, we tried to use as much as possible Isabelle's in-built proof search functions. For sequent calculii, procedures exist for two kinds of search: a 'fast' depth-first search, and a kind of search where a heuristic—typically the size of the subgoal—is used to guide the search. It becomes possible to code a sophisticated search algorithm but, in our quest to reduce the amount of coding involved, we decided to use undifferentiated search.

However, with this first, straightforward formulation of the proof system for ILL, a problem arises when the proof depends on rules that can cause exhaustive search to non-terminate. Three such rules occur in Linear Logic:

- Exchange: The problem with exchange is the same one that many theorem provers face with commutativity laws.
- Contraction: Remembering that the proof process described is backward, a using the contraction rule corresponds to expanding the subgoal, and this operation can be applied again and again:

$$\Gamma, !A \vdash B \quad \Leftarrow \quad \Gamma, !A, !A \vdash B \quad \Leftarrow \quad \Gamma, !A, !A, !A \vdash B \quad \Leftarrow \quad \ldots$$

- Cut: Not only is there a problem with the fact that the cut rule always applies and hence can be applied *ad infinitum* as the Contraction rule, it is also not useful without information on *what* to cut. While Girard has shown that the cut rule can be eliminated from proofs in ILL, in practice it is very useful to have some form of cut for proof search.

The need to use the permutation rule can be eliminated by changing the representation of a sequent from a list-like structure into a multiset-like one. As an illustration, the $(\multimap_{\mathcal{L}})$ rule from Figure 1 is replaced by the following one:

$$\frac{\Gamma' \vdash A \qquad \Delta', B \vdash C \qquad \Gamma \uplus \Delta = \Gamma' \uplus \Delta'}{\Gamma, A \multimap B, \Delta \vdash C}\, (\multimap_{\mathcal{L}})$$

where we mean that the context Γ, Δ will be split into some contexts Γ', Δ', such that the multi-set union $\Gamma' \uplus \Delta'$ is equal to $\Gamma \uplus \Delta$. The important part is to keep this context division *lazy*, that is, to have the particular Γ and Δ left unspecified until the proof tree is fully built and the leaves of the tree solved. Then the values are filled in and the proof completed. The approach is similar to the one proposed by Hodas and Miller [7].

The main effect of the infinite application of contractions and cuts is that, if these rules must be used unrestrictedly, depth-first search becomes disallowed, and even other search algorithms become prohibitively expensive. While some proofs can be completed with a search algorithm

based on the size of the subgoal, other proof searches result in the use of a huge amount of space. Not only that, proof search in LL is undecidable, so there is no way to know that a search will complete successfully or not.

While many researchers have tried to solve the problem by fine-tuning the search algorithm, we decided on a different approach. The main tool for automation we have used is *proof by lemmas*, whereby some of the costs and difficulties of proof search are reduced by the use of additional lemmas, sometimes replacing primitive rules. For example, a lemma we have found to be useful in proof search is:

$$\frac{\Gamma, \Delta \vdash A}{\Gamma, !(A \multimap B), \Delta \vdash B} \multimap_L\_lemma$$

which corresponds to the LL proof:

$$\dfrac{\dfrac{\Gamma, \Delta \vdash A \qquad B \vdash B}{\Gamma, A \multimap B, \Delta \vdash B}\,(\multimap_{\mathcal{L}})}{\Gamma, !(A \multimap B), \Delta \vdash B}\,Dereliction$$

This small proof appears very frequently in many proofs, and performing the steps separately increases the branching factor considerably, mainly due to the division of context that occurs. So, we have added this derived rule into the 'pack' of rules to use in proof search. Likewise, other lemmas have been proved and used for proof search, rather than using all the original definitions[1]; these lemmas are presented in Figure 2.

$$\frac{}{\Gamma, !B \multimap 0, \Delta, !B, \Pi \vdash A}\,raa_1 \qquad \frac{}{\Gamma, !B, \Delta, !B \multimap 0, \Pi \vdash A}\,raa_2 \qquad \frac{\Gamma, !((!A \multimap 0) \& (!B \multimap 0)), \Delta \vdash C}{\Gamma, !(!(A \oplus B) \multimap 0), \Delta \vdash C} \otimes\_\&$$

$$\frac{\Gamma, B, \Delta, \Pi \vdash C}{\Gamma, A, \Delta, A \multimap B, \Pi \vdash C}\,mp_1 \qquad \frac{\Gamma, B, \Delta, \Pi \vdash C}{\Gamma, A \multimap B, \Delta, A, \Pi \vdash C}\,mp_2 \qquad \frac{\Gamma, !A, !B, \Delta \vdash C}{\Gamma, !(A \& B), \Delta \vdash C}\,\&\_lemma$$

$$\frac{\Gamma, \Delta \vdash A}{\Gamma, !(A \multimap B), \Delta \vdash B} \multimap_L\_lemma \qquad \frac{A, !A, \Gamma \vdash B}{\Gamma \vdash !A \multimap B} \multimap_R\_lemma \qquad \frac{\Gamma, !A \multimap 0, \Delta \vdash B}{\Gamma, !A \multimap (!A \multimap 0), \Delta \vdash B}\,a\_not\_a$$

Figure 2: Additional lemmas for proof search

It was said above that these rules are all simply presented to the automated search procedures. However, there is one more feature in how these rules are added. Rules can be declared as safe or unsafe; safe rules are ones that do not cause loss of information, and therefore over which there will be no need to backtrack.

We have explained above the principles behind the modification made to the logic to fit the needs of automated theorem-proving. Apart from the lazy division of the context, they are all quite straightforward, and can be believed in with a quick perusal. Even the lazy context division is not difficult to understand, as it can be seen in the axiomatization itself and one does not need to examine the implementation details in a different programming language. We believe that one of the advantages of using a system such as Isabelle is to allow users to know *exactly* what is going on. The concrete syntax of the rules, illustrated in Appendix A, bears witness to the ease of supplying a set of rules to Isabelle.

[1] We make no claim that this set of derived rules is complete; however, as the *logic* itself is known to be complete, the user can interactively use one of the rules not in the pack and finish the proof.

There were several slight variations of the above rules that we could have used. One possible representation (used in other mechanizations of LL) is to maintain the context division into two parts (exponentiated formulae and others) throughout, and not only for promotion as we have done. This modification of the rule could speed several operations on contexts, but we felt that ease of interaction with the users would not be maintained if they had to do some of the processing (separating exponentiated and non-exponentiated terms) themselves. Another possible modification would be to eliminate weakening and instead change the identity rule to accept any number of exponentiated items in the context. However, this made the pattern-matching of contexts in the context division untenable, and so we decided not to incorporate it. It should be noted that it was easy to try these changes, as the statement of the proof rules is completely transparent.

6 Further experiments

Using the final set of rules described in the previous section, we have been able to prove many theorems in ILL. Specifically, we have been able to prove automatically some lemmas proposed by Troelstra [15] and the conjectures obtained by translating lemmas in intuitionistic propositional logic obtained from Kleene [10] following the translation schema proposed by Girard [4]. While we have been able to prove a large number of theorems automatically, the important feature of this implementation is not automation, but its usefulness in exploring applications of the logic. We believe that, as Linear Logic gets better known, more applications will be proposed and actually attempted.

The use of Linear Logic as a base on which to embed other formalisms has been suggested quite often. As an example, it was easy to add the Girard translation scheme to the Isabelle system: it could be quickly supplied within only a few lines of uncomplicated statements (including infix and precedence information). This meant that the goal statements for proof could be entered in easy to read syntax. For example, the statement of lemma 60f from Kleene is readable in IL form:

```
val k60f = prove_il "|- - ((- A) | B) = - (- (A & (- B)))";
```

but substantially more unwieldy when supplied directly in ILL:

```
val k60f = "
 |- ! (! (! (! A -o 0) ++ ! B) -o 0) -o ! (! (A && ! B -o 0) -o 0) -o 0 &&
 ! (! (! (A && ! B -o 0) -o 0) -o 0) -o ! (! (! A -o 0) ++ ! B) -o 0" : thm
```

This is an example of the fact that with the tools available in Isabelle it is easy to avoid doing translations by hand, also avoiding doing too much programming to get encodings of applications. This is because Isabelle is particularly geared towards embeddings, and the same facilities used to enter the definition of a logic are available to build other syntax schemes and axioms on top of a logic.

The implementation described can be used to model problem domains that deal with resource creation and consumption, such as in Petri nets. A small example of such an application is the model of a laundry machine [1], shown in Appendix B. While this is a trivial example, the code shows how the facilities can be used for describing problems directly in ILL, and how proofs can be developed very easily.

Another idea in using a generic framework such as Isabelle is that we can experiment with variations of the logic—ideal for studying such a relatively new system as Linear Logic. In this vein, we have also experimented with a second implementation of ILL, this time with rules in the natural deduction style. We are in the process of using this implementation to run the suite of Kleene theorems, by experimenting with different packages of rules and lemmas for proof automation. Likewise, we have an encoding of classical LL, and again we are examining the construction of a set of rules (both axiomatic and derived) to automate proof search in it. The translation from a table illustrating the logic to the input to Isabelle is very direct.

7 Conclusions

We hope this paper has provided a feel for our implementation of Intuitionistic Linear Logic in Isabelle. The main point to remember is that the prover for ILL we have developed is appropriate for interactive development of proofs and descriptions of application domains and embeddings of other formalisms on top of Linear Logic. It also is meant to be taken as evidence of the ease of building theorem provers using Isabelle.

It seems natural to provide some comparisons of our proof system with previous implementations of Linear Logic. Firstly, we must reiterate that we have not built any specialized tool for proof search; proofs are found *within* the logic by directly applying rules and searching for proofs of the resulting goals. The other approach—of searching for proofs *outside* the logic, by means of a program in the implementing language—automatically gives the scope for much more efficiency. However, while we have not done a detailed analysis of our timing profile, the timings we have obtained are not hopelessly slow, considering that efficiency was not our main goal.

An important feature of the kind of automation we have achieved with Isabelle is that it allows the judicious mixture of automated macro steps and user control in critical steps in the proof (such as an interesting cut) which makes for the discovery of interesting and relevant proofs. For us, proof search is not a function that either returns a full proof for a formula or fails: one can use the package of lemmas and primitive rules, as well as other lemmas proved by the user, to explore the space of subgoals, and this is an important aspect of theorem-proving applications. Due to Isabelle's characteristics as an interactive prover, we can carry on an automatic proof for a while, then give it some user information and finish it off automatically again, a flexibility that we hope will be of use.

The fact that we have not implemented a completely new specialized prover but have instead instantiated a generic proof framework, carries with it all sorts of advantages. Proof tools could be used off-the-shelf with very little work. For example, Girard's translation of IL into ILL was very easily encoded in a dozen simple lines, including the mixfix notation and grammar precedences. Even the proof search algorithm we used was already given. On the other hand, further tools can be coded and integrated quite easily. We in fact plan to make user of other researchers' work in proof search and integrate their algorithms within the framework provided in our work. This framework is the application of the LCF approach: however the proof is found, a term is only marked as being a theorem if it follows directly from the axioms of the logic. In contrast to programs written in untyped languages, a prover built using Isabelle provides the assurance of a strong type system and a powerful meta-language: a lot of power is given to the user, without in any way compromising the security of the theorems.

While we have several implementations of Linear Logic, the one described here (ILL in sequent-calculus) is the one we have developed furthest. The main reason is that most available provers cover classical LL, and make use of particular representation schemes that do not allow an easy separation of the intuitionistic fragment. However, often one does want to conduct proofs restricted to the intuitionistic fragment. Some interest among the community studying Linear Logic focuses on finding appropriate proof terms and exploring the cut-elimination property; the implementation described here will be very useful for studying these issues, and we have already started to add terms to ILL.

As further work, we also plan to improve proof support for the other formulations (natural deduction ILL and double-sided sequent CLL). By maintaining several parallel versions of Linear Logic, all with the same mode of interaction, it becomes possible for a user to learn more about a particular variant of the logic. We have chosen the double-sided representation, as we believe some users of theorem-provers for LL may find the two-sided version more intuitive, even at the cost of efficiency. We also would like to experiment with more variations such as predicate Linear Logic and Full Iintuitionistic Linear Logic [9].

We believe the prover described in this paper will be useful for modeling applications of Linear Logic. Prospective users are encouraged to experiment with it. The Isabelle system itself has extensive documentation [13] and a large group of users.

Acknowledgements

This works builds on early work by Marcus Moore. We have profited from discussions with Gavin Bierman and Larry Paulson.

References

[1] V. Alexiev. Applications of Linear Logic to computation: An overview. Technical Report TR93-18, University of Alberta, Edmonton, 1993.

[2] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90, 1993.

[3] D. Galmiche and G. Perrier. Foundations of proof search strategies design in linear logic. In *Symposium on Logical Foundations of Computer Science*, volume 813 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[4] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987.

[5] J.-Y. Girard and Y. Lafont. Linear logic and lazy computation. In *Proceedings of TAPSOFT'87, vol. 2*, volume 250 of *Lecture Notes in Computer Science*, pages 52–66, 1987.

[6] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Springer-Verlag, 1979.

[7] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1994. To Appear.

[8] G. P. Huet. A unification algorithm for typed λ calculus. *Theoretical Computer Science*, 1, February 1975.

[9] M. Hyland and V. de Paiva. Full intuitionistic linear logic. *Annals of Pure and Applied Logic*, 64, 1993.

[10] S. C. Kleene. *Introduction to metamathematics*, volume 1 of *Bibliotheca mathematica*. North-Holland Publishing Co., 1952.

[11] P. Lincoln and N. Shankar. Proof search in first-order Linear Logic and other cut-free sequent calculii. In *Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1994.

[12] D. Miller. A multiple-conclusion meta-logic. In *Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1994.

[13] L. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[14] T. Tammet. Proof strategies in Linear Logic. Technical Report 70, Department of Computer Science, Chalmers University of Technology, 1993.

[15] A. Troelstra. *Lectures on linear logic*. Number 29 in CSLI lecture notes. Stanford, CA: Center for the Study of Language and Information, 1992.

A Axiomatization of LL used

The following is the text of an Isabelle 'theory file'; it shows the formalization of ILL in Isabelle syntax. (The representation of sequents is not shown here, for this consult the section on LK in the Isabelle manual.

```
ILinear = sequents +

consts
"><"    ::"[o, o] => o"             (infixr 35)
"-o"    ::"[o, o] => o"             (infixr 45)
"o-o"   ::"[o, o] => o"             (infixr 45)
FShriek :: "o => o"                 ("! _" [100] 1000)
"&&"    ::"[o, o] => o"             (infixr 35)
"++"    ::"[o, o] => o"             (infixr 35)
zero    ::"o"                       ("0")
top     ::"o"                       ("1")
eye     ::"o"                       ("I")
aneg    ::"o=>o"                    ("~_")


(* syntax for context manipulation *)
        Context      :: "two_seqi"
       "@Context"    :: "two_seqe"   ("((_)/ :=: (_))" [6,6] 5)

(* syntax for promotion rule *)

        PromAux :: "three_seqi"
      "@PromAux":: "three_seqe"      ("{_||_||_}")

rules

identity "P |- P"

zerol           "$G, 0, $H |- A"

liff_def        "P o-o Q == (P -o Q) >< (Q -o P)"

aneg_def        "~A == A -o 0"

derelict        "$F, A, $G |- C ==> $F, !A, $G |- C"

contract        "$F, !A, !A, $G |- C ==> $F, !A, $G |- C"

weaken          "$F, $G |- C ==> $G, !A, $F |- C"


promote2        "{ || $H || B} ==> $H |- !B"
promote1        "{!A, $G || $H || B} ==> {$G || $H, !A || B}"
promote0        "$G |- A ==> {$G || || A}"

tensl           "$H, A, B, $G |- C ==> $H, A >< B, $G |- C"
```

```
impr            "A, $F |- B ==> $F |- A -o B"

conjr           "[| $F |- A ; $F |- B |]  ==> $F |- (A && B)"

conjll          "$G, A, $H |- C ==> $G, A && B, $H |- C"

conjlr          "$G, B, $H |- C ==> $G, A && B, $H |- C"

disjrl          "$G |- A ==> $G |- A ++ B"

disjrr          "$G |- B ==> $G |- A ++ B"

disjl           "[| $G, A, $H |- C; $G, B, $H |- C |] ==> $G, A ++ B, $H |- C"


  (* RULES THAT DIVIDE CONTEXT *)

tensr           "[| $F, $J :=: $G; $F |- A ; $J |- B |]
                                          ==> $G |- A >< B"

impl            "[| $G, $F :=: $J, $H; B, $F |- C ; $G |- A |]
                                       ==> $J, A -o B, $H |- C"


cut " [| $J1, $H1, $J2, $H3, $J3, $H2, $J4, $H4 :=: $F ;
         $H1, $H2, $H3, $H4 |- A ;
         $J1, $J2, A, $J3, $J4 |- B                 |]  ==> $F |- B"


  (* CONTEXT MATCHING RULES *)

context1     "$G :=: $G"
context2     "$F, $G :=: $H, !A, $G ==> $F, A, $G :=: $H, !A, $G"
context3     "$F, $G :=: $H, $J ==> $F, A, $G :=: $H, A, $J"
context4a    "$F :=: $H, $G ==> $F :=: $H, !A, $G"
context4b    "$F, $H :=: $G ==> $F, !A, $H :=: $G"
context5     "$F, $G :=: $H ==> $G, $F :=: $H"

end

ML

(* Setting the parser and pretty-printer to deal with sequents and
constants which use them. *)


val parse_translation = [("@Trueprop",sequents.single_tr),
 ("@Context",sequents.two_seq_tr "Context"),
 ("@PromAux", sequents.three_seq_tr "PromAux")];

val print_translation = [("Trueprop",sequents.two_seq_tr' "@Trueprop"),
 ("Context",sequents.two_seq_tr'"@Context"),
 ("PromAux", sequents.three_seq_tr'"@PromAux")];
```

B A small example

B.1 Theory file defining the example

```
washing = ILinear +

consts

dollar,quarter,loaded,dirty,wet,clean   :: "o"

rules

change   "dollar |- (quarter >< quarter >< quarter >< quarter)"

load1    "quarter , quarter , quarter , quarter , quarter |- loaded"

load2    "dollar , quarter |- loaded"

wash     "loaded , dirty |- wet"

dry      "wet, quarter , quarter , quarter |- clean"

end
```

B.2 Proving a property of the example

```
(* first changing definitions into rules using the "cut" rule: *)

val changeI = [context1] RL ([change] RLN (2,[cut]));
val load1I =  [context1] RL ([load1]  RLN (2,[cut]));
val washI =   [context1] RL ([wash]   RLN (2,[cut]));
val dryI =    [context1] RL ([dry]    RLN (2,[cut]));


(* now to a proof: *)

goal thy "dollar , dollar , dirty |- clean";

by (best_tac (safe_cs add_safes (changeI @ load1I @ washI @ dryI)) 1);

(* goal proved *)
```