Efficient Resource Management for Linear Logic Proof Search

Iliano Cervesato[1], Joshua S. Hodas[2], and Frank Pfenning[1]

[1] Department of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213-3891, USA
e-mail: {iliano|fp}@cs.cmu.edu
[2] Computer Science Department, Harvey Mudd College
Claremont, CA 91711, USA
e-mail: hodas@cs.hmc.edu

Abstract. The design of linear logic programming languages and theorem provers opens a number of new implementation challenges not present in more traditional logic languages such as Horn clauses (*Prolog*) and hereditary Harrop formulas (λ*Prolog*). Among these, the problem of efficiently managing the linear context when solving a goal is of crucial importance for the use of these systems in non-trivial applications. This paper studies this problem in the case of *Lolli* [6] (though its results have application to other systems). We first give a proof-theoretic presentation of the operational semantics of this language as a resolution calculus. We then present a series of resource management systems designed to eliminate the non-determinism in the distribution of linear formulas that undermines the efficiency of a direct implementation of this system.

1 Introduction

Linear logic [2] views logical assumptions as consumable resources. This allows elegant and concise formalizations of a number of problems which are difficult to represent in traditional logics. In particular, many problems centered around the notion of a state that evolves as a computation proceeds fall into this category. Consequently, several logic programming languages based on linear logic have been designed in the last five years [1, 3, 6, 10]. Others are the subject of extensive research. Each proposal is accompanied by interesting theoretical results that show its computational relevance, and by numerous examples that prove its practical significance. However, to our knowledge, usable implementations have thus far been released only for *Lolli* [6] and *Lygon* [3].

Linear logic programming languages offer the implementor new challenges not present in more traditional logic languages such as *Prolog* or λ*Prolog*. Among these, the efficient management of the linear formulas contained in the context is of crucial importance for the use of these languages in non-trivial applications.

This issue is particularly simple in *Prolog*: The only predicates that can modify the program are the extra-logicals `assert` and `retract`, which have global effect. In languages admitting implications in goals, λ*Prolog* [9] and *Elf* [13] for example, the use of scoped assumptions causes the program to grow and

This paper will appear in the proceedings of the *1996 International Workshop on Extensions of Logic Programming*, Leipzig, Germany, March 28–30 1996.

contract like a stack. The matter is more complicated in the case of linear logic due to the strict rules placed on the use and reuse of assumptions.

The problem is best exemplified by considering the rule for proving $G_1 \otimes G_2$:

$$\frac{\Delta_1 \longrightarrow G_1 \quad \Delta_2 \longrightarrow G_2}{\Delta_1, \Delta_2 \longrightarrow G_1 \otimes G_2} \otimes_R$$

When the interpreter needs to use this rule during the bottom-up search for a proof, the assumption set has not already been divided into Δ_1 and Δ_2. The naive choice is to generate all partitions of the assumption set until a pair Δ_1, Δ_2 with the desired properties is found. This non-deterministic behavior is clearly a problem since the number of partitions grows exponentially with the number of assumptions in the context. Considering the frequency with which $\otimes$ and other multiplicative connectives occur in practice, an interpreter for a linear logic programming language based on such a generate-and-test algorithm would be usable only for toy problems.

In this paper, we will provide a deterministic solution to this problem, as well as to less apparent issues in context management involving the additive connectives and constants. We do not treat other sources of non-determinism, which can be handled according to standard techniques in a logic programming framework, or that we might want to keep open in a theorem prover. We will focus our attention on the language *Lolli* [5, 6], that we used to test the techniques described below. However, our results have already been applied to a prototype implementation of a programming language based on Miller's specification logic *Forum* [10], and should apply equally well to implementations of other linear logic programming languages such as *Lygon* [3]. It is also possible to adapt these techniques to the development of theorem provers for linear logic.

We do not provide proofs of the soundness and completeness theorems that relate the systems presented here. We believe these results are simple enough to be reconstructed by the reader, who is referred to Hodas' dissertation [5] for proofs relevant to the first two systems.

2 Resolution for Linear Hereditary Harrop Formulas

The programming language *Lolli* [5, 6] is based on the fragment of linear logic freely generated by the operators $\top$, &, $\multimap$, $\supset$ and $\forall$. The connective $\supset$ is called *intuitionistic implication* and is defined as $A \supset B \equiv !A \multimap B$. Positive occurrences of $\mathbf{0}$, $\mathbf{1}$, $\oplus$, $\otimes$, $!$, $\exists$ and the syntactic equality among atomic formulas, $a \doteq a'$, are also allowed, as they do not invalidate any essential properties of the language. This extended fragment is called the language of *linear hereditary Harrop formulas* (*LHHF* for short).

The logic of *LHHF* is conveniently described by sequents of the form:

$$\Gamma; \Delta \Longrightarrow G$$

where Γ and Δ are multisets of implicitly labelled negative formulas (only $\top$, &, $\multimap$, $\supset$ and $\forall$ are allowed as their principal connective) called the *intuitionistic*

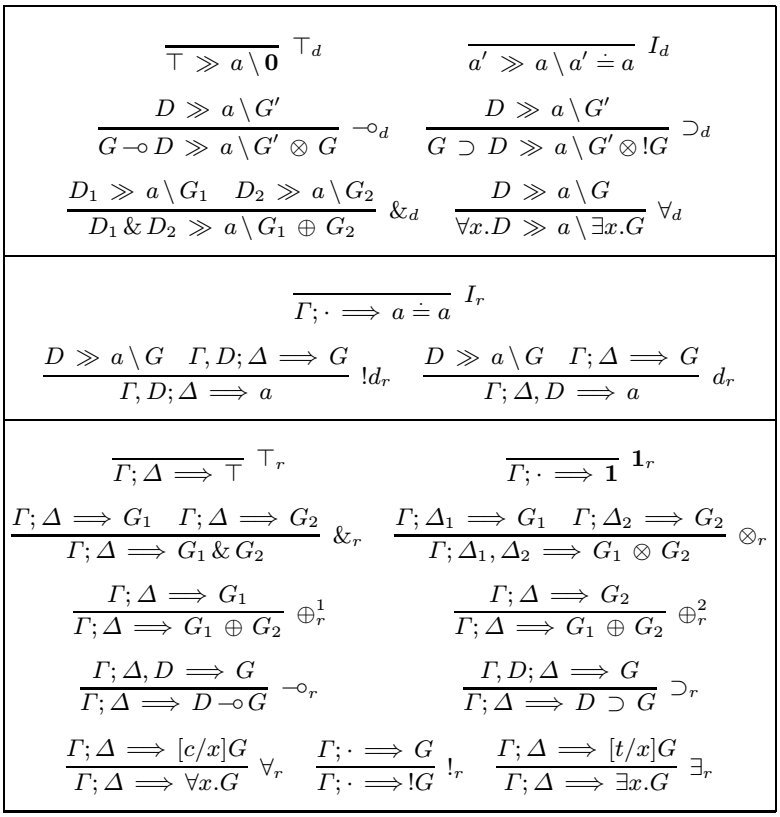

Fig. 1. $\mathcal{R}$: A Resolution Calculus for $LHHF$[3].

and the *linear* context respectively, and together constitute the *program*. G is a positive formula called the *goal*. The formulas in the intuitionistic context are implicitly preceded by the modal operator !, so that the expression above translates to the more traditional sequent $!\Gamma, \Delta \Longrightarrow G$. This manner of structuring the sequents and the use of $\supset$ retains desirable aspects of the semantics of ! (in particular formulas in the intuitionistic context can be used arbitrarily many times), while preventing unwanted behaviors.

Hodas and Miller discuss a proof system, $\mathcal{L}$, for $LHHF$ based on sequents of this form [6]. They also prove the soundness and completeness of $\mathcal{L}$ with respect to the usual rules for linear logic restricted to the language of $LHHF$. Most importantly, they proved that $LHHF$ possesses the necessary computa-

[3] In this and all subsequent proof systems, the right introduction rule for universal quantification is assumed to carry the usual proviso that the introduced constant does not appear free in the lower sequent. Similarly, the variable x does not appear free in a in rule $\forall_d$.

tional properties to be considered an *abstract logic programming language* [11]. In particular, every proof in $\mathcal{L}$ can be transformed into an equivalent proof that consults the program only when the goal formula is atomic (thus proofs are *goal-directed* [11]), and at that point selects and operates upon a single program formula in order to proceed with the derivation (thus proofs are *focused* [1]). Proofs with both properties are called *uniform*. Hodas and Miller capture this behavior in the system $\mathcal{L}'$ which eliminates the left-hand rules of the logic in favor of a single rule for *backchaining*.

In Fig. 1 we present a new *resolution system*, called $\mathcal{R}$, for *LHHF*. This system is different from but equivalent to the system $\mathcal{L}'$. It is easy to show that the judgment $\Gamma; \Delta \Longrightarrow G$ is provable in $\mathcal{R}$ if and only if the sequent $\Gamma; \Delta \longrightarrow G$ is provable in $\mathcal{L}'$.

The rules in the bottom section of Fig. 1 describe how to reduce non-atomic goal formulas. They stem from the right introduction rules of linear logic, and are essentially identical to the right rules for $\mathcal{L}'$ [6]. $\mathcal{R}$ differs from $\mathcal{L}'$ in the treatment of atomic goal formulas. In order to solve these goals, Hodas and Miller rely on the function $\| \cdot \|$, which converts a formula in the program to a (possibly infinite) set of clauses, each defining a single ground atom. Here, we embed the process of clause selection and elaboration into the proof system itself, giving it a more syntactic and operational flavor.

When the goal formula a is atomic (Fig. 1 center), a program formula D is selected from either the intuitionistic context (rule $!d_r$) or from the linear context (rule d_r). In either case, it is passed to the *formula decomposition judgment*

$$D \gg a \setminus G$$

(Fig. 1, top) together with the atomic formula a in order to extract a goal formula G which is equivalent to D in a sense to be explained below. At this point, the computation proceeds by solving G. Note that when d_r is used, D is removed from the context so that it may not be used again subsequently.

Program formulas can be seen as partial definitions of the atomic propositions or predicate symbols that can appear in a goal position. In Horn logic, each program clause $G \supset a$ participates in the definition of a single atom (a). When we admit free uses of conjunction, a single program formula can partially define several atomic formulas; for example $G \supset (a_1 \wedge a_2)$ takes part in the definition for two distinct atoms (a_1 and a_2). When selecting a program formula D from the context in order to solve an atomic goal a, we would like to transform D into an equivalent *clause* of the form $G \multimap a$, that can be immediately used to reduce the problem of proving a to that of solving G. This is essentially what is achieved by the formula decomposition judgment $D \gg a \setminus G$.

3 A Resource Consumption Calculus for LHHF

The resolution calculus presented in the last section does not commit to any strategy in order to split the linear context when processing multiplicative goals from the bottom up. The non-determinism involved in this open choice can be

$$\frac{}{\Gamma; \Delta^I \backslash \Delta^I \Longrightarrow a \doteq a}\ I_{rm_1} \qquad \frac{}{\Gamma; \Delta^I \backslash \Delta^I \Longrightarrow \mathbf{1}}\ \mathbf{1}_{rm_1}$$

$$\frac{}{\Gamma; \Delta, \Delta^O \backslash \Delta^O \Longrightarrow \top}\ \top_{rm_1}$$

$$\frac{D \gg a \backslash G \quad \Gamma, D; \Delta^I \backslash \Delta^O \Longrightarrow G}{\Gamma, D; \Delta^I \backslash \Delta^O \Longrightarrow a}\ !d_{rm_1} \qquad \frac{D \gg a \backslash G \quad \Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G}{\Gamma; \Delta^I, D \backslash \Delta^O \Longrightarrow a}\ d_{rm_1}$$

$$\frac{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1 \quad \Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_2}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1 \,\&\, G_2}\ \&_{rm_1} \qquad \frac{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1 \oplus G_2}\ \oplus^1_{rm_1}$$

$$\frac{\Gamma; \Delta^I \backslash \Delta' \Longrightarrow G_1 \quad \Gamma; \Delta' \backslash \Delta^O \Longrightarrow G_2}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1 \otimes G_2}\ \otimes_{rm_1} \qquad \frac{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_2}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1 \oplus G_2}\ \oplus^2_{rm_1}$$

$$\frac{\Gamma; \Delta, \Delta^O, D \backslash \Delta^O \Longrightarrow G}{\Gamma; \Delta, \Delta^O \backslash \Delta^O \Longrightarrow D \multimap G}\ \multimap_{rm_1} \qquad \frac{\Gamma, D; \Delta^I \backslash \Delta^O \Longrightarrow G}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow D \supset G}\ \supset_{rm_1}$$

$$\frac{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow [c/x]G}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow \forall x.G}\ \forall_{rm_1} \qquad \frac{\Gamma; \cdot \backslash \_ \Longrightarrow G}{\Gamma; \Delta^I \backslash \Delta^I \Longrightarrow !G}\ !_{rm_1} \qquad \frac{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow [t/x]G}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow \exists x.G}\ \exists_{rm_1}$$

Fig. 2. $\mathcal{RM}_1$: A Resource Management System for *LHHF*

computationally harmful unless we devise a sound and complete method to split the linear context deterministically. Let us restate the problem in terms of the proof system just described. The resolution rule for the connective $\otimes$ is as follows:

$$\frac{\Gamma; \Delta_1 \Longrightarrow G_1 \quad \Gamma; \Delta_2 \Longrightarrow G_2}{\Gamma; \underbrace{\Delta_1, \Delta_2}_{\Delta} \Longrightarrow G_1 \otimes G_2}\ \otimes_r$$

In order to solve the goal $G_1 \otimes G_2$, we need to split the original linear context Δ into Δ_1 and Δ_2 such that G_1 can be solved using the resources in Δ_1 and G_2 can be solved using the resources in Δ_2. Since intuitionistic formulas are reusable, Γ is copied to the two premisses. Assume that Δ contains n formulas. Then there are 2^n possible splits. In the worst case, finding a workable split (or determining that none exists) will require trying them all.

This problem was given a deterministic solution by Hodas and Miller in [6] in what they called the *I/O model* of execution for *Lolli*. We will instead use the name *resource management system* and refer to this deduction system as $\mathcal{RM}_1$.

The rule above, $\otimes_r$, attempts to split the context Δ at a stage when the resources needed to prove the two subgoals G_1 and G_2 are completely unknown. However, if the original goal is to succeed, all resources not used to prove G_1 will be used to solve G_2, and vice versa. The key idea behind the resource consumption model is, therefore, to upgrade the passive role of goal formulas as

resource users to the more active role of *resource consumers*. Under this view, we will give one of the subgoals, G_1 say, the whole linear context Δ; it will consume part of it and return the remaining portion Δ_2 to be used by G_2.[4]

This basic idea is formalized in Fig. 2 by means of judgments of the form:

$$\Gamma; \Delta^I \setminus \Delta^O \Longrightarrow G$$

where Δ^I is the linear part of the context that is given as *input* in order to solve G. In general, G will be just one of the subgoals produced during the derivation of a top-level goal A. The proof of G will consume part of Δ^I and return the portion it did not use as the *output context* Δ^O, that will need to be consumed by some other subgoal derived from A. Clearly the output context for the original overall goal A should be empty. Indeed, the soundness and completeness theorem for resource consumption states that $\Gamma; \Delta \Longrightarrow G$ is derivable if and only if $\Gamma; \Delta \setminus \cdot \Longrightarrow G$ is derivable, where "$\cdot$" represents the empty context. The proof of this statement, as well as for similar results concerning subsequent systems, follows by straightforward generalization and simple induction. An important invariant of $\mathcal{RM}_1$, as well as of the enhanced versions to be introduced, is that, when the judgment $\Gamma; \Delta^I \setminus \Delta^O \Longrightarrow G$ is derivable, the output context Δ^O is always a submultiset of the input context Δ^I.

In their original paper, Hodas and Miller write this judgment $I\{G\}O$, with G being the goal formula, and I and O being the input and the output contexts respectively [6]. The main difference with respect to our judgment is that in their presentation I and O are lists of items that can be either program formulas or the special constant `del`. This is very close to their original *Prolog* implementation of *LHHF*. Here Δ^I and Δ^O are instead multisets of formulas. This is consistent with the resolution judgment presented in Sect. 2, and permits easier proofs of soundness and completeness. We also make use of the clause decomposition judgment in place of the special predicate pickr which they appeal to.

When considering the judgment $\Gamma; \Delta^I \setminus \Delta^O \Longrightarrow G$, we adopt a computational point of view in which the schematic variables Γ, Δ^I and G are given as input to the rules, while Δ^O is returned as an output value from the resolution of the goal. This is consistent with a left-to-right subgoal selection strategy, that we adopt as well. Note, however, that the rules themselves do not commit to this operational interpretation. Rather, they are fully declarative.

We will not discuss this system in detail since it is isomorphic to the one presented by Hodas and Miller. We will simply point out a few features that will be relevant to the discussion of the refinements we present below.

The resolution rules for the equality test (rule I_r) and for the multiplicative unit (rule $\mathbf{1}_r$) require an empty linear context, i.e. solving these goals does not consume resources. In $\mathcal{RM}_1$ we model this behavior by returning as output the same context these rules received as input. In a similar fashion, the exponential ! expects its subgoal to be solvable in an empty linear context. Therefore, rule

[4] This change in perspective corresponds to the shift from data-oriented to object-oriented programming, and fits well with a common view held in the linear logic community of goals as active processes.

$!_{rm_1}$ passes the empty linear context to its premiss and returns the whole input context as output. In this rule and elsewhere, we write an output context that must be empty due to global invariants as "$\_$". We use "$\cdot$" for the empty context in other circumstances, e.g. when the emptiness of a context needs to be checked to match a rule, or when setting an input context to empty.

In the resolution system, $\top$ succeeds as a goal in any linear context. We model this behavior by allowing this formula to consume an arbitrary portion Δ of its input context.

The operational behavior of additive conjunction & requires that we solve both subgoals G_1 and G_2 in the same linear context. This is modelled in $\mathcal{RM}_1$ by giving the original input context to both G_1 and G_2, and expecting them to return the same output context, Δ^O, that will be the output context of the compound formula $G_1 \,\&\, G_2$ (rule $\&_{rm_1}$).

The rule for multiplicative implication ($\multimap_{rm_1}$) requires some attention. Let Δ^I be the original input context. In order to process this connective, we need to augment Δ^I with the antecedent D of the implication. Let Δ^O be the context returned after solving its consequent G. We can return Δ^O as the output of the proof of $D \multimap G$ only if we are sure that the newly added instance of D does not appear in Δ^O. This is because this D must be consumed during the proof of G. We enforce this constraint by writing Δ^I as Δ, Δ^O and passing Δ, Δ^O, D as the input context to the premiss of the rule. By expecting Δ^O as the output of the whole subproof, we assert that Δ, D represents the portion of the input context that is consumed while proving G. No such complications are needed for the rule dealing with intuitionistic implication since its assumption is added to the intuitionistic context.

4 Removing Non-Determinism from the Treatment of $\top$

While the resource management policy enforced by system $\mathcal{RM}_1$ removes the most serious cause of non-determinism present in the resolution system $\mathcal{R}$, it is not yet fully deterministic. This is due to the operational semantics of the logical constant $\top$. As presented in rule $\top_{rm_1}$, this goal is allowed to consume any portion Δ of its input context. If it contains n formulas, we are left with 2^n possible output contexts Δ^O that might be passed to the remaining computation.

Hodas and Miller initially underestimated the importance of this issue [6]. However the subsequent development of sample applications to accompany the first public release of *Lolli* showed this problem to be critical in practice. The solution we describe is adapted from Hodas' dissertation [5], and was incorporated into that implementation.

Roughly speaking, the idea is that once a $\top$ has been encountered, the remaining subgoals do not need to consume all of their input context since the unused formulas could be "pumped back" to the place in the proof tree where $\top$ was first seen. That is, $\top$ should not actively consume resources on its own; rather, it should give permission to later goals to ignore resources which otherwise would have to be consumed.

We obtain this behavior by adding an extra parameter to the resource management judgments of $\mathcal{RM}_1$. We now use sequents of the form:

$$\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow_v G$$

where v is a boolean-valued flag (the $\top$-*flag* as Hodas called it, or *slack indicator* as we will often refer to it) to be considered as another output argument of the resolution of the goal G. Whenever $v = 0$, the resolution of G uses exactly the resources in $\Delta^I - \Delta^O$. If instead this flag has the value 1, G uses $\Delta^I - \Delta^O$ for sure, but may also absorb part or all of the output context Δ^O. In this case, we say that Δ^O is the *slack* of that branch of the proof tree. When $v = 0$, the computation has no slack.

Due to the limited space available, we do not present the full system, which we call $\mathcal{RM}_2$. We will however describe some of its most critical rules.

The main changes with respect to $\mathcal{RM}_1$ concern the rules that close the proof trees, and the binary rules. Rules I_{rm_2} and $\mathbf{1}_{rm_2}$:

$$\frac{}{\Gamma; \Delta^I \backslash \Delta^I \Longrightarrow_0 a \doteq a}\ I_{rm_2} \qquad \frac{}{\Gamma; \Delta^I \backslash \Delta^I \Longrightarrow_0 \mathbf{1}}\ \mathbf{1}_{rm_2}$$

both pass their linear context as the output context for the remainder of the computation, since neither can consume any resources. These rules set the $\top$-flag to 0 since no occurrence of $\top$ is encountered during the proof of either $\mathbf{1}$ or the equality test. In contrast, when $\top$ is processed as a goal in rule $\top_{rm_2}$:

$$\frac{}{\Gamma; \Delta^I \backslash \Delta^I \Longrightarrow_1 \top}\ \top_{rm_2}$$

it passes its input context as output too, but raises the $\top$-flag, indicating that it can be considered to have consumed some of those resources if that proves necessary. The subsequent computation will use this information for context management.

Rule $\&_{rm_1}$ is split into four rules in $\mathcal{RM}_2$. Each rule handles one possible combination of $\top$-flags returned by the two premisses. If no $\top$ was encountered while solving either G_1 or G_2 (rule $\&^1_{rm_2}$), then the context is managed as in the previous system and the $\top$-flag for the proof of the compound goal $G_1 \,\&\, G_2$ is set to 0.

When exactly one of the two premisses sets the slack indicator, then the behavior of the rule is determined by the other premiss. Consider for example the case where the left premiss sets the $\top$-flag (the other case, rule $\&^3_{rm_2}$, is symmetrical). We have the following rule:

$$\frac{\Gamma; \Delta^I \backslash \Delta_1, \Delta^O \Longrightarrow_1 G_1 \quad \Gamma; \Delta^I \backslash \Delta^O \Longrightarrow_0 G_2}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow_0 G_1 \,\&\, G_2}\ \&^2_{rm_2}$$

Let Δ be the portion of the context used while proving G_2 (clearly, $\Delta^I = \Delta, \Delta^O$). Since both G_1 and G_2 must consume the same portion of the context, the proof of G_1 can use part of Δ but no formula from Δ^O. However, it does not need to consume explicitly all the formulas in Δ, because, unlike G_2, its slack indicator

is set. We can therefore write the output context of G_1 as Δ_1, Δ^O, where Δ_1 is the actual slack of this branch of the proof tree, and is some submultiset of Δ. The $\top$-flag for the proof of $G_1 \,\&\, G_2$ is set to 0: since both premisses must consume the same resources and G_2 cannot take up slack, the composed goal cannot have any slack. For the same reason, the output context of $G_1 \,\&\, G_2$ is Δ^O.

In the final case, if both premisses return their $\top$-flag set to 1, both subgoals allow arbitrary slack. Therefore, we set the $\top$-flag for the proof of the compound formula, since in this case any excess resources can be "pumped back" to both premisses. The output context for the compound goal is the intersection of the output contexts returned by each of the premisses: since both branches must end up having consumed the same resources, only what is not used in either branch can be forwarded. This yields the following rule:

$$\frac{\Gamma; \Delta^I \setminus \Delta_1^O \Longrightarrow_1 G_1 \quad \Gamma; \Delta^I \setminus \Delta_2^O \Longrightarrow_1 G_2}{\Gamma; \Delta^I \setminus \Delta_1^O \cap \Delta_2^O \Longrightarrow_1 G_1 \,\&\, G_2} \;\&^4_{rm_2}$$

Slack handling in the rule for multiplicative conjunction is quite simple since resources are allowed to flow freely from one premiss to the other. We set the $\top$-flag if either subgoal allows slack. The overall output context is the linear context returned after proving the right premiss.

Finally, the rule for ! resets the $\top$-flag regardless of whether $\top$ has been encountered while solving its subgoal or not. Since the output context must coincide with the input context in this rule, there is no place for any slack.

It is important to note that $\mathcal{RM}_1$ and $\mathcal{RM}_2$ improve the efficiency of the resolution system $\mathcal{R}$ in two different ways. The proofs obtainable in $\mathcal{RM}_1$ are in one to one correspondence with the derivations we could achieve with $\mathcal{R}$. $\mathcal{RM}_1$ improves the efficiency of proof-search by pruning from the search space branches corresponding to unsuccessful splits of the linear context. In contrast, the system $\mathcal{RM}_2$ actually collapses some proofs by identifying successful derivations that differ only on the distribution of unused assumptions among various occurrences of $\top$. For example, consider an attempt to solve the goal $a \multimap a \multimap a \multimap (\top \otimes \top)$ in the empty context. There are eight distinct proofs in $\mathcal{RM}_1$ corresponding to the different ways of dividing the consumption of the a's between the two occurrences of $\top$, but there is only one proof in $\mathcal{RM}_2$.

5 Improving the Treatment of Additive Conjunction

The system $\mathcal{RM}_2$ presented in the last section achieves determinism in context management in the sense that no arbitrary context splitting choices remain. Nevertheless, a close examination of the rules reveals that some serious efficiency and completeness problems still remains. In particular, the rules concerning & are unsatisfactory. The problem is already present in $\mathcal{RM}_1$, where we had the following rule:

$$\frac{\Gamma; \Delta^I \setminus \Delta^O \Longrightarrow G_1 \quad \Gamma; \Delta^I \setminus \Delta^O \Longrightarrow G_2}{\Gamma; \Delta^I \setminus \Delta^O \Longrightarrow G_1 \,\&\, G_2} \;\&_{rm_1}$$

Assuming a sequential execution for the two premisses, this rule requires that we first solve G_1 obtaining, say, an output context Δ_1^O. Then G_2 will be proved and return the output context Δ_2^O. At this point, and only at this point, we check that Δ_1^O and Δ_2^O are equal.

Even though this test can be done efficiently (for example by having a bit vector where each position records whether the corresponding resource has been used), we may end up rejecting many pairs of proofs before finding a pair that consumes the same set of resources. At best this is inefficient. At worst, when a proof of G_2 proceeds down a divergent path that it might avoid with better pruning, it leads to added incompleteness in the system. Further, in a language with a notion of side-effect (such as screen output), an avoidable failed proof may nevertheless produce a recordable effect.

Consider the following example, written using *Lolli*'s concrete syntax:

```
test :- (a & b), c.  % Comma is syntax for multiplicative conjunction
LINEAR c.                      % This goes in the linear context
a.
b :- c, write "Some Output".   % Fails, but prints
```

According to the current execution model, the goal '`?- test`' is solved by first proving `a` (without consuming any linear resources), then attempting to prove `b`. The clause for this goal is selected and its body attempted. The linear resource `c` is consumed, the message is printed, and `b` succeeds. At this point, the resources consumed while solving `a` and `b` are compared and the conjunction fails since the latter conjunct used `c` while the former did not. This causes the failure of the original query. Clearly, it would be preferable for the attempt to solve `b` to fail as soon as `c` is accessed, so that the message is never printed.[5]

In order to recognize more quickly those failures caused by the second goal incorrectly accessing resources unused by the first, we could modify the rule $\&_{rm_1}$ as follows:

$$\frac{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1 \quad \Gamma; \Delta^I - \Delta^O \backslash \cdot \Longrightarrow G_2}{\Gamma; \Delta^I \backslash \Delta^O \Longrightarrow G_1 \,\&\, G_2}\ \&'$$

In this rule, we give G_2 exactly the portion of the linear context that it can use and expect the empty context as an output. In this way, the resources not consumed by G_1 are inaccessible to G_2; this achieves our purposes.

This change will not, however, help the system to detect failures caused by the second conjunct failing to consume resources that the first conjunct does use. To see how this becomes an issue, consider another *Lolli* program:

```
test :- (a, c) & b.
LINEAR a.                          % This goes in the linear context
LINEAR c.                          % This goes in the linear context
b :- c & (write "Some Output", c). % Fails, but prints
```

[5] Even in a *Prolog*-based implementation, where the constraint on the output contexts is enforced by unification rather than an after-the-fact check, the same problem occurs if we replace the body of the rule for `b` with `c, write "Some Output", true`.

If we execute the query '`?- test`' the system will first solve the goal to the left of the additive conjunction by consuming `a` and then `c`. At this point it will attempt to prove `b`. Since the left conjunct has used all of the resources in the input context, `b` can and must use them all as well (so there is no new restriction added by the change to the rule for & we just described). The rule for `b` is selected, and its left conjunct is solved using just `c`. At this point, since the right conjunct can only use `c` but the overall proof of `b` was supposed to use both `a` and `c`, we know enough to fail. Unfortunately, the system will not recognize this situation and will print the message. The resource `c` will then be consumed and the proof of `b` will succeed, having consumed `c`. Only when checking that all the resources passed to `b` have been used, will the system finally recognize the failure, and cause the original query to fail.

In order to obtain the desired behavior, we modify the form of our judgment to include three input contexts on the left of the arrow:

$$\Gamma; \Xi; \Delta^I \setminus \Delta^O \Longrightarrow_v G$$

In this judgment (which also features the slack indicator of $\mathcal{RM}_2$) the input linear context is logically divided into two parts: the *strict* context Ξ that must be entirely consumed during the resolution of the goal G, and the *non-strict* context Δ^I whose contents might be consumed while solving G. Thus Ξ will be managed like the linear context in the system $\mathcal{R}$; only Δ^I may transmit unused resources to the output Δ^O, as in $\mathcal{RM}_2$. The rules defining the semantics of this judgment are represented in Fig. 3; they constitute the system $\mathcal{RM}_3$. We will now briefly describe their principal characteristics.

First, since we have split the linear context, we need to provide two separate rules for accessing a linear formula when the goal is atomic (rules $d^1_{rm_3}$ and $d^2_{rm_3}$). The rules for the equality judgment and for proving the goal $\mathbf{1}$ are straightforward (rules I_{rm_3} and $\mathbf{1}_{rm_3}$): since neither is allowed to consume any resources, the strict context (which contains resources that must be consumed) must be empty; the non-strict context is passed over unmodified as output. In contrast, the rule for $\top$ deletes whatever portion of the strict context it is provided with, and forwards as output its non-strict context, while setting the $\top$-flag to indicate that the output is now slack (rule $\top_{rm_3}$).

The rules for & are more complicated. In order to solve the goal $G_1 \,\&\, G_2$ with respect to the linear context $\Xi; \Delta^I$, we first solve G_1 in $\Xi; \Delta^I$, obtaining the output context Δ' (remember, Ξ must be entirely consumed). Two different courses of action are now possible, depending on the value of the slack indicator:

1. If this flag was not set (rule $\&^1_{rm_3}$), G_2 must consume everything that has been used by G_1, i.e. Ξ as well as $\Delta^I - \Delta'$. These two components are packaged together into the strict context of the judgment for G_2. Since this goal is not allowed to consume any other resources, it is given an empty non-strict context.
2. If the resolution of G_1 has encountered an occurrence of $\top$ and slack is admitted (rule $\&^2_{rm_3}$), G_2 must still consume every resource used by G_1 (i.e. $\Xi, \Delta^I - \Delta'$), but is also allowed to access the resources not used by this goal

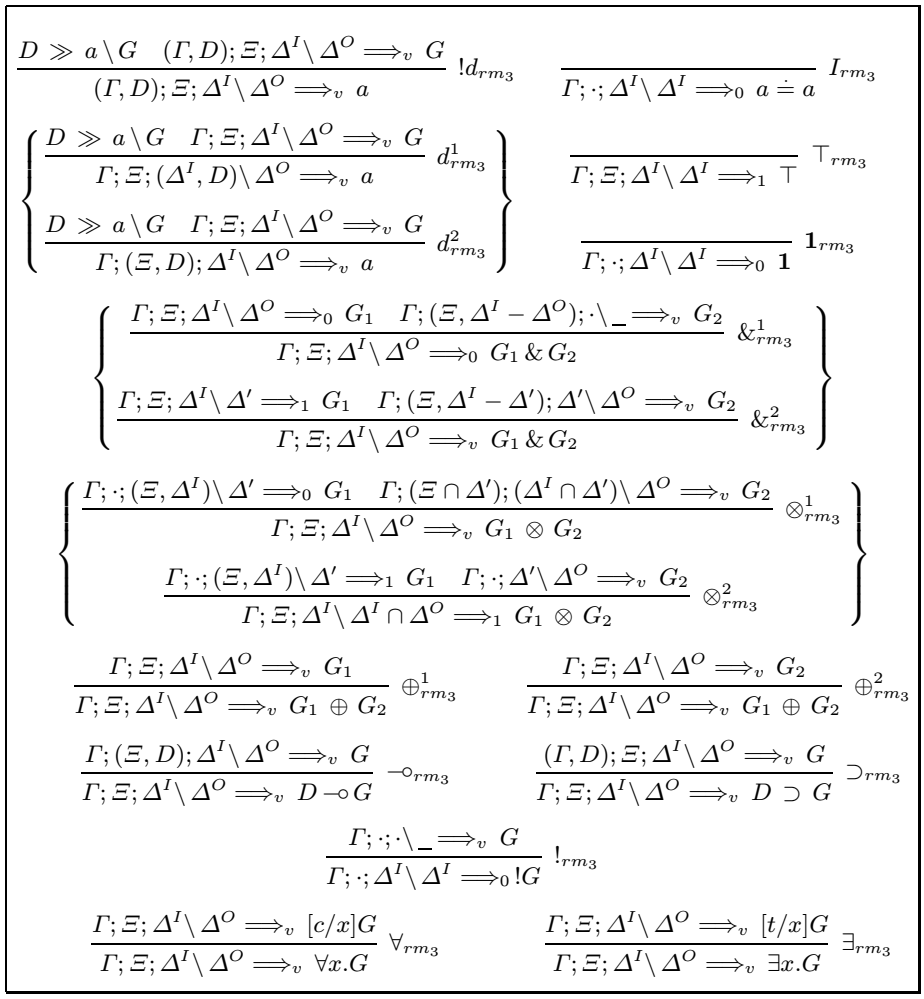

Fig. 3. $\mathcal{RM}_3$:An Improved Resource Management System for *LHHF*.

(Δ'), therefore, we supply this as the non-strict context for the proof of G_2. The output context and slack indicator for this second premiss then provide the corresponding values for the lower sequent.

When solving a goal of the form $G_1 \otimes G_2$, the strict context Ξ must be consumed by either G_1 or G_2. Since the first of these subgoals may use an arbitrary part of Ξ as well as some portion of the non-strict context Δ^I, we put both Ξ and Δ^I in the non-strict context of G_1 and leave the strict context empty. As with $\&$, how to solve G_2 depends on the value of the $\top$-flag.

1. If no slack is allowed (rule $\otimes^1_{rm_3}$), G_2 must consume whatever portion of the original strict context G_1 did not use, and may consume some formulas in Δ^I that were not already consumed by G_1. Therefore, we restore the remainder of Ξ and Δ^I to the strict and non-strict contexts of the judgment for G_2, respectively. To do this we take the intersection of these multisets with the output context Δ' of G_1.
2. If the slack indicator was set by the proof of G_1 (rule $\otimes^2_{rm_3}$), all the strict resources in the original Ξ can be "pumped back" to G_1 in the case that G_2 does not use them. Therefore we call this goal with an empty strict context and the output context of G_1, Δ', as its non-strict context. We must be careful, however, not to return strict resources from Ξ as part of the output context of $G_1 \otimes G_2$, since they are presumed to have been consumed by the slack consumer in G_1. Therefore we intersect the context returned by G_2 with the original non-strict input context Δ^I of the composed goal.

The rule dealing with linear implication takes advantage of the strict context to simplify the task of managing the new assumption (rule $\multimap_{rm_3}$). Since D must be used while proving G, it is simply put into the strict context of this subgoal. The rules for $\supset$, !, $\oplus$, and the quantifiers display no interesting new features.

The system $\mathcal{RM}_3$ provides a satisfactory solution to all the resource management problems we discussed in the previous sections. Unfortunately, it does so at a rather high price since most of its rules involve complex operations on the context (exhaustive tests on the status of one of the contexts, shuffling formulas from the strict to the non-strict context or vice versa, etc.). Furthermore, the order in which assumptions are made must be preserved so that the programmer can predict in which sequence clauses are tried when solving atomic goals.

Thus we store the intuitionistic, strict and non-strict assumptions in a common data structure, differentiating the role of each formula by means of a tag. Further, when a formula is consumed, it is generally more efficient to mark it as such (rather than actually delete it) in order to facilitate backtracking. In this type of implementation, each time we perform a test to check, for instance, if the strict context is empty, we have to visit all the formulas present in all contexts. Similar costs are incurred when we perform operations like taking the intersection of two contexts.

We have achieved a substantial improvement in performance by maintaining additional information about the program, in particular the number of formulas present in the strict and non-strict contexts. Then, checking the emptiness of the strict context reduces to an inexpensive arithmetic comparison, for example.

The rules for handling the tensor still perform a relatively expensive operation, since they must move the contents of the strict context into the non-strict context unless the former is initially empty. We can eliminate this overhead for nested occurrences of $\otimes$ by requiring this connective to be parsed as a left associative operator. In this way, the topmost occurrence of $\otimes$ will undergo the shuffling process. But, since all inner occurrences appear in the left conjunct (G_1 in rules $\otimes^i_{rm_3}$), they will be proved with an empty strict context, avoiding any additional shuffling.

The techniques presented in this section have been applied into an enhanced version of the *ML* interpreter for *Lolli*. The declarative nature of the rules makes these same ideas applicable to implementations based on other programming paradigms. In particular, the original *Prolog* prototype for *Lolli* [6] can be easily adapted to take advantage of these observations.

6 Conclusions and Related Work

The issue of efficient context management has proved to be crucial for the use of linear logic programming languages in non-trivial applications. In this paper, we have presented a general technique that not only eliminates sources of non-determinism deriving from naive context management, but also permits early recognition of certain failure situations. We have implemented these ideas in the interpreter for a forthcoming release of the language *Lolli*. Tests showed a general improvement in performance and in some examples arbitrary speed-ups. We also achieved convergence for some previously non-terminating programs. The determinism also simplifies the programmer's task: Despite the apparent complexity of $\mathcal{RM}_3$ it is relatively straightforward to predict the operational behavior of programs and avoid inefficient generate-and-test situations.

To our knowledge, the only other authors who have been concerned with the issue of efficiency in context management for linear logic programming languages are the designers of *Lygon*. In a recent publication [4], they build on the work of Hodas and Miller and independently develop a system with the characteristics of Hodas' efficient handling of $\top$. They do not, however, present a notion equivalent to our strict context, and make no mention of techniques akin to our linear formula counters to reduce the overhead at the implementation level.

Our analysis was motivated primarily the goal of building an efficient interpreter, but should also be applicable to the design of compilers which, of course, will ultimately be necessary for the execution of large programs. We expect that compilation techniques developed for *Prolog* [8] and λ*Prolog* [7, 12] may be combined with our methods.

The results described in the paper can be applied to other programming languages based on linear logic. Hodas and Polakow have extended the system $\mathcal{RM}_3$ to Miller's specification logic *Forum* [10] and have based a prototype implementation on it. These techniques should extend just as easily to implementations of *Lygon* [3, 4] and other linear languages. Finally, since *Forum* is complete for all of classical linear logic, they are also clearly applicable to the design of theorem provers.

Acknowledgments

We would like to thank Vladimir Alexiev, James Harland, Dale Miller and Roberto Virga, as well as the anonymous reviewers, for their valuable comments on early versions of this paper.

References

1. Jean-Marc Andreoli and Remo Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing* 9:3–4, 1991.
2. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–101, 1987.
3. James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming. In *Proceedings of the International Logic Programming Symposium, San Diego, California, October 1991*, V. Saraswat and K. Ueda, eds., pp. 304-318.
4. James Harland and Michael Winikoff. Deterministic resource management for the linear logic programming language Lygon. Technical Report TR 94/23, Melbourne University, Department of Computer Science, 1994.
5. Joshua S. Hodas, *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*, Ph.D. Dissertation from University of Pennsylvania, Department of Computer and Information Science, May 1994. Available electronically at `http://www.cs.hmc.edu/~hodas/papers/`.
6. Joshua S. Hodas and Dale Miller. Logic Programming in a Fragment of Linear Logic. *Journal of Information and Computation*, 110(2):327–365, 1994.
7. Keehang Kwon. *Towards a Verified Abstract Machine for a Logic Programming Language with a Notion of Scope.* PhD thesis, Department of Computer Science, Duke University, December 1994. Available as Technical Report CS-1994-36.
8. Timothy G. Lindholm and Richard A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. *Proceedings of the Fourth International Conference on Logic Programming*, J.L. Lassez, ed., pp 21–39, MIT Press 1987.
9. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schröder-Heister editor, *Proceedings of the International Workshop on Proof-Theoretical Extensions of Logic Programming*, pages 253–281, Tübingen, Germany, 1989, Springer-Verlag LNAI 475.
10. Dale Miller. A Multiple-Conclusion Meta-Logic. *Proceedings of the 1994 Symposium on Logics in Computer Science*, S. Abramsky, ed., pp. 272–281.
11. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming, *Annals of Pure and Applied Logic*, 51:125–157, 1991.
12. Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Technical Report CS-1994-35, Department of Computer Science, Duke University, October 1994. To appear in *Journal of Logic Programming.*
13. Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.