A Stack-Based Lazy Splitting System for the Forum Specification Language

Pablo López Ernesto Pimentel

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Campus de Teatinos. 29071 Málaga
Phone: +34 5 213 13 96 Fax: +34 213 1397
{lopez,ernesto}@lcc.uma.es

Abstract. Linear Logic provides a logical framework for expressing fundamental computational concepts in a pure declarative style. As a consequence, it has been used as a sound foundation for the design of very expressive programming and specification languages. Unfortunately, linearity is as convenient for specifying as difficult to implement. In particular, the successful implementation of linear logic languages and provers involving context splitting strongly depends on the efficiency of the method computing a suitable split. A number of solutions have been proposed, referred to as *lazy splitting* or *resource management* systems. In this paper we present a stack-based lazy splitting system for Forum, although it can be applied to any linear logic language or prover as well. The main contribution is that we manage contexts as stacks, providing a simple and natural nested scope preservation. The implementation is simpler and much more efficient than other proposals, as current lazy splitting systems traverse and update contexts, which takes a time proportional to their cardinalities. On the other hand, pushing onto and popping from a stack are nearly constant-time, inexpensive operations. In addition, our system can also be viewed as an extension of the way hereditary Harrop languages – such as λProlog and Elf – manage the logic program, allowing the application of previous work invested in the implementation of these.
Keywords: Linear Logic, Logic Programming, Forum, Implementation, Lazy Splitting, Lazy Sequent Calculus

1 Introduction

Linear Logic, introduced by Girard in [7], is an extension of Intuitionistic and Classical Logics devised to overcome the expressiveness shortcomings of these. It is a *logic of resources.* While Intuitionistic Logic deals with the construction of proofs – it is a constructive logic – and Classical Logic deals with the truth, Linear Logic deals with finite and infinite resource management. As a result of this change of perspective, Linear Logic extends the expressiveness of both Intuitionistic and Classical Logics, allowing a simple and concise representation of situations difficult to express within these.

Linear Logic is a *logic for states*, where sequents encode a dynamically evolving global state and inference rules behave as atomic transition rules. Within a sequent, formulas stand for both entity states and compound transition methods. Linear Logic is also a *logic for concurrency*, where connectives are viewed as basic interaction patterns – such as synchronization, mutual exclusion and resource sharing – and formulas specify arbitrarily complex interacting processes. This capability for expressing fundamental computational concepts such as states and concurrency justifies the interest that Linear Logic has awaked in computer science in general, and in the logic programming community in particular. Since its inception, an effective linearizing wave has emerged, reaching and deeply soaking formalisms such as λ-calculus and type systems. From a theoretical point of view, Linear Logic provides a convenient logical framework for embedding a variety of sequential and concurrent computational models, according to the *proofs as computations* paradigm [19]. From a practical point of view, Linear Logic constitutes a sound basis for the design of expressive specification and programming languages integrating features such as garbage collection, object-orientation and concurrency in a purely logic declarative style, as well as for the logical specification of concurrent object-oriented systems. On the other hand, these languages and systems are usually hard to specify within Intuitionistic or Classical Logics, requiring the use of undesirable, impure, extra-logical features.

Unfortunately, and because of its high degree of non-determinism, linearity is as convenient for specifying as difficult to implement. Traditional language implementation techniques are insufficient to undertake the *efficient* implementation of linear logic languages. The linear nature of formulas – the key feature on which the expressiveness of Linear Logic holds – implies they must be used *exactly once* in a proof, arising major problems concerning their management. Some of the linear logic languages proposed, such as LO [1] and ACL [11], are based on rather restricted fragments of Linear Logic. Although these restrictions considerably simplify their implementations, they also imply a severe expressiveness loss. Other languages, such as Lolli [9], Lygon [22] and Forum [16, 18], are based on larger, more expressive and harder to implement fragments of Linear Logic. In particular, when implementing these richer languages we are faced with the problem of splitting the contexts between the subproofs introduced by a bottom-up application of the multiplicative rules $\otimes R$, $\invamp L$ and $\multimap L$. A number of solutions have been proposed for the context splitting problem, referred to as *lazy splitting* or *resource management* systems.

This paper presents a stack-based lazy splitting system for the Forum specification language. Our system can be viewed as a refinement of the resource management system proposed by Cervesato et al. for the Lolli programming language [3]. The main contribution is that we manage contexts as stacks instead of as multisets. Managing contexts as stacks has a number of advantages. In the first place, it is a very natural and simple solution to nested scope preservation. The implementation is also simpler and more efficient than other proposals. The former is particularly important, since lazy splitting is just an implementation technique whose final goal is precisely that of efficiency. In addition, our system can be seen as a "higher order" extension of the way logic languages based on hereditary Harrop formulas – such as λProlog and Elf – manage the logic program, and hence, it is envisaged that much of the research invested in the implementation of these could be reused for linear logic languages using our lazy splitting system. Finally, it is simple to preserve the order of clauses within the stack approach, an important topic when considering the implementation of a logic programming language. As the original Lolli system, our stack-based lazy splitting system is also general enough to be applied to the efficient implementation of any linear logic language or theorem prover involving context splitting, either single or multiple-conclusion.

The rest of the paper is organized as follows. In the following section, we briefly introduce the Forum specification language. Section 3 presents a stack-based lazy splitting system partially determining the resource assignment. Following two sections improve the proposed system by adding two important optimizations. Section 6 enunciates the soundness and completeness theorems. Finally, comparison to related work and conclusions are outlined.

2 The Forum Specification Language

Forum is a linear logic specification language proposed by Miller [16, 18] for integrating both abstractions and concurrency into a declarative setting. Since its inception, and due to its striking expressiveness, Forum has been used for specifying logics and proof systems [18, 20], operational semantics of programming languages [18, 5, 4] and hardware devices [5], and object-oriented programming features [6]. On the other hand, there has been a much minor interest in providing an efficient implementation of Forum, except for preliminary research efforts [10, 20, 13].

2.1 The Logic of Forum

Forum[1] is defined as the logic freely generated from Andreoli's *asynchronous* connectives ($\top$, $\bot$, $\&$, $\parr$, $?$ and $\forall$) and from the linear and intuitionistic implications ($\multimap$, $\Rightarrow$). Forum is not a fragment of Linear Logic but an alternative *asynchronous* presentation of all of Higher Order Linear Logic, since its primitive connectives constitute a complete – but not minimal – set, enabling an asynchronous encoding of the rest of linear connectives [16, 18].

The contexts of Forum sequents are divided into classical (Ψ) and linear ($\Delta, \mathcal{A}, \Gamma$) zones, separated from each other by semicolons. The formulas contained in the classical zone Ψ are implicitly replicated and discarded along the proof search as needed. As shown in Fig. 1, Forum sequents have the forms

$$\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; \Gamma \quad \text{and} \quad \Sigma : \Psi ; \Delta \xrightarrow{F} \mathcal{A}$$

[1] To reduce the gap between the original Forum proof system $\mathcal{F}$ and our lazy proof system $\mathcal{LF}$, we present here an equivalent proof system $\mathcal{F}'$ dispensing with "?" as in [16] and rearranging the Forum sequents.

$$\frac{}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; \top, \Gamma}\ \top\text{-}R \qquad \frac{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; G_1, \Gamma \quad \Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; G_2, \Gamma}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; G_1 \mathbin{\&} G_2, \Gamma}\ \&\text{-}R$$

$$\frac{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; \Gamma}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; \bot, \Gamma}\ \bot\text{-}R \qquad \frac{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; G_1, G_2, \Gamma}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; G_1 \parr G_2, \Gamma}\ \parr\text{-}R$$

$$\frac{\Sigma : \Psi ; D, \Delta \longrightarrow \mathcal{A} ; G, \Gamma}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; D \multimap G, \Gamma}\ \multimap\text{-}R \qquad \frac{\Sigma : D, \Psi ; \Delta \longrightarrow \mathcal{A} ; G, \Gamma}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; D \Rightarrow G, \Gamma}\ \Rightarrow\text{-}R$$

$$\frac{y : \tau, \Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; G[y/x], \Gamma}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; \forall_\tau x.G, \Gamma}\ \forall\text{-}R \qquad \frac{\Sigma : \Psi ; \Delta \longrightarrow A, \mathcal{A} ; \Gamma}{\Sigma : \Psi ; \Delta \longrightarrow \mathcal{A} ; A, \Gamma}\ \text{atomic -}R$$

$$\frac{\Sigma : D, \Psi ; \Delta \xrightarrow{D} \mathcal{A}}{\Sigma : D, \Psi ; \Delta \longrightarrow \mathcal{A} ; \emptyset}\ \text{decide } \Psi \qquad \frac{\Sigma : \Psi ; \Delta \xrightarrow{D} \mathcal{A}}{\Sigma : \Psi ; D, \Delta \longrightarrow \mathcal{A} ; \emptyset}\ \text{decide } \Delta$$

$$\frac{}{\Sigma : \Psi ; \emptyset \xrightarrow{A} A}\ \text{initial } \mathcal{A} \qquad \frac{}{\Sigma : \Psi ; \emptyset \xrightarrow{\bot} \emptyset}\ \bot\text{-}L \qquad \frac{\Sigma : \Psi ; \Delta \xrightarrow{D_i} \mathcal{A}}{\Sigma : \Psi ; \Delta \xrightarrow{D_1 \& D_2} \mathcal{A}}\ \&_i\text{-}L\ \ (i = 1, 2)$$

$$\frac{\Sigma : \Psi ; \Delta_1 \xrightarrow{D_1} \mathcal{A}_1 \quad \Sigma : \Psi ; \Delta_2 \xrightarrow{D_2} \mathcal{A}_2}{\Sigma : \Psi ; \Delta_1, \Delta_2 \xrightarrow{D_1 \parr D_2} \mathcal{A}_1, \mathcal{A}_2}\ \parr\text{-}L \qquad \frac{t : \tau \in \Sigma \quad \Sigma : \Psi ; \Delta \xrightarrow{D[t/x]} \mathcal{A}}{\Sigma : \Psi ; \Delta \xrightarrow{\forall_\tau x.D} \mathcal{A}}\ \forall\text{-}L$$

$$\frac{\Sigma : \Psi ; \Delta_1 \xrightarrow{D} \mathcal{A}_1 \quad \Sigma : \Psi ; \Delta_2 \longrightarrow \mathcal{A}_2 ; G}{\Sigma : \Psi ; \Delta_1, \Delta_2 \xrightarrow{G \multimap D} \mathcal{A}_1, \mathcal{A}_2}\ \multimap\text{-}L \qquad \frac{\Sigma : \Psi ; \Delta \xrightarrow{D} \mathcal{A} \quad \Sigma : \Psi ; \emptyset \longrightarrow \emptyset ; G}{\Sigma : \Psi ; \Delta \xrightarrow{G \Rightarrow D} \mathcal{A}}\ \Rightarrow\text{-}L$$

The rule ∀-R has the proviso that y is not declared in the signature Σ.

Fig. 1. Forum Proof System $\mathcal{F}'$

where Σ is a signature containing the definition of the non-logical constants involved, Ψ is a set of Σ-formulas, Δ and Γ are multisets of Σ-formulas, $\mathcal{A}$ is a multiset of atomic Σ-formulas, and F is a Σ-formula. The Forum sequents above are respectively translated into the linear logic sequents

$$!\Psi, \Delta \longrightarrow \mathcal{A}, \Gamma \quad \text{and} \quad !\Psi, \Delta, F \longrightarrow \mathcal{A}$$

where $!\Psi$ denotes the set obtained by applying ! to every formula in Ψ.

2.2 Proof Search in Forum

The main advantage of Forum is that it is more deterministic than Linear Logic while retaining all of its expressiveness. In particular, Forum proofs are *goal-directed*; that is, the shape of the proof tree depends on the goals to solve. Goal-directedness is formalized in Forum through multiple-conclusion uniform proofs – a natural generalization of single-conclusion uniform proofs [15] – defined by Miller in [18] as follows

Definition 1. *A cut-free sequent proof Ξ is* uniform *if for every subproof Ξ' of Ξ and for every non-atomic formula occurrence B in the right-hand side of the end-sequent of Ξ', there is a proof Ξ'' that is equal to Ξ' up to a permutation of inference rules and is such that the last inference rule in Ξ'' introduces the top-level logical connective of B.*

Forum is an *abstract logic programming language* [18] as it is complete with respect to uniform proofs. In addition, Forum proofs are also focused; that is, once a formula has been selected no further formula selection is considered but inference rules are applied to the selected formula and its subformulas until they are fully decomposed. Both uniformity and focusing – on the left only[2] – are firmly embedded into the $\mathcal{F}'$ proof system in such a way that the bottom-up proof search in $\mathcal{F}'$ is always the search for uniform, left-focused proofs.

2.3 Non-Determinism in Forum

The rather disciplined use of inference rules imposed in the $\mathcal{F}'$ proof system determines *when* its inference rules must be used, but not *how*. In particular, we can identify the following non-deterministic selections when searching for a proof in Forum:

- **program formula selection:** the *decide* rules select a program formula. In addition, the $\&_i L$ rule selects a program subformula.
- **term selection:** the $\forall L$ rule selects a simply typed λterm t substituting a given bounded variable x
- **split selection:** the $\multimap L$ and $⅋ L$ rules select a split of the linear contexts Δ and $\mathcal{A}$

In most logic programming languages the suitable program formula and term are selected by means of unification and backtracking. Forum allows empty-head program formulas which can always be applied (i.e. they always "unify"), making exhaustive backtracking a vain effort. In a sense, the application of these empty-head program formulas is not goal-directed, as the current goals do not provide any tip on their selection. Furthermore, Forum allows arbitrary higher order quantification over simply typed λterms, and hence, there is no unification algorithm providing all substitutions. We shall not address neither program formula nor term selection in this paper – previous work on λProlog can be applied to the latter – but shall present an efficient solution for the context splitting problem in the following sections.

3 Lazy Splitting: Putting Linear Logic to Work

The successful implementation of linear logic languages and theorem provers including splitting multiplicatives strongly depends on the efficiency of the method computing a suitable context split. In order to understand the importance of this, consider the $⅋ L$ rule of the $\mathcal{F}'$ proof system

$$\frac{\Sigma : \Psi; \Delta_1 \xrightarrow{D_1} \mathcal{A}_1 \quad \Sigma : \Psi; \Delta_2 \xrightarrow{D_2} \mathcal{A}_2}{\Sigma : \Psi; \underbrace{\Delta_1, \Delta_2}_{\Delta} \xrightarrow{D_1 ⅋ D_2} \underbrace{\mathcal{A}_1, \mathcal{A}_2}_{\mathcal{A}}} ⅋\text{-}L$$

[2] In [18] $\mathcal{A}$ and Γ are defined as lists and it is suggested decomposing goals in Γ from head to tail – yielding focusing on the right – but this is not always computationally advantageous (see 4.2)

When applying this rule bottom-up the contexts Δ and $\mathcal{A}$ must be split into Δ_1, Δ_2 and $\mathcal{A}_1$, $\mathcal{A}_2$ respectively. The problem here is that we do not know how these splits should be done. As the number of splits is exponential with respect to the cardinalities of Δ and $\mathcal{A}$, it is quite clear that a trivial implementation backtracking through all these splits is terribly inefficient. On the other hand, some languages and provers avoid this problem either by not including the splitting connectives or by using them under certain strict conditions assuring that context splitting is not required. The penalty for this simplification is a considerable loss of expressiveness, as these connectives play an important role in the computational interpretations of Linear Logic. This suggests that a better implementation technique must be developed instead, making it possible the practical exploitation of the expressive power of Linear Logic. A commonly adopted solution is to split the contexts lazily [22, 3, 13]. In this section we present a stack-based lazy splitting system partially removing the unnecessary non-determinism from the split selection.

3.1 The Essence of Lazy Splitting: Satiability of Proofs

The key leading to lazy splitting is that, in general, linear logic proofs are satiable; i.e. they consume what they need, but no more. We can take advantage of this satiability by letting proofs receiving an excess of formulas and returning this excess instead of failing. Hence, we do not split contexts but pass the whole of them to the proofs as input, which consume the formulas they need until satiated and return the unused ones as output. That is, as we cannot guess the required split *a priori* (i.e as input), we let the very proofs determine it *a posteriori* (i.e. as output), assigning formulas to proofs on demand. Once again, an optimal solution is achieved within computer science through the judicious postposition of decisions, "leaving till tomorrow what cannot be done today". Note that as described lazy splitting is neither implicitly sequential nor parallel. Both implementations are possible, being the main difference the use given to the output returned by the proofs. In this paper we are concerned with a sequential implementation of lazy splitting only.

3.2 A Lazy Sequent Calculus: Input and Output Contexts

According to the description above, lazy splitting requires a sort of bidirectional sequent calculus such that lazy proofs receive an excess of formulas as input and return it as output. Therefore, lazy sequents must include input and output contexts as follows

$$\Sigma : \underbrace{\Psi ; \Delta \longrightarrow \mathcal{A} ; \Gamma}_{input} / \underbrace{\Delta' ; \mathcal{A}'}_{output} \qquad \Sigma : \underbrace{\Psi ; \Delta \xrightarrow{F} \mathcal{A}}_{input} / \underbrace{\Delta' ; \mathcal{A}'}_{output}$$

Input contexts are the same as before, except for possibly containing an excess of formulas. Output contexts return this excess. Lazy sequent rules must assure a correct flow of input and output. Obviously, input flows from the root to the leaves of the proof tree as rules are applied, while output flows from the leaves to the root along a finished proof branch. Figure 2 shows some lazy sequent rules. Lazy axiom rules appearing at the leaves of the proof tree – such as *initial* $\mathcal{A}$ – transfer unused input to output. Lazy unary rules – such as *decide* Δ – are trivial as they simply pass input and output on the suitable direction. Lazy splitting rules do the real work. The lazy $\invamp L$ rule in Fig. 2 shows a sequential implementation of lazy splitting, where the input contexts Δ and $\mathcal{A}$ are not blindly split but directly passed to the left subproof as input, which consumes the formulas it requires and gently returns the unused ones – Δ' and $\mathcal{A}'$ – as output to be passed to the right subproof as input. Then, the right subproof consumes the formulas it needs and returns unused input as output – Δ'' and $\mathcal{A}''$ – to be further returned as the overall output of the whole proof. In a parallel implementation of lazy splitting, input contexts are simultaneously passed to both proofs, while output contexts are no further reassigned as input but used to check splitting constraints. Note that lazy splitting is concerned with linear input only, as classical input Ψ is not split but replicated. It should be clear that the lazy sequent at the root of a lazy proof cannot return any formula; i.e. its output contexts must be empty.

$$\frac{}{\Sigma : \Psi; \Delta \xrightarrow{A} A, \mathcal{A} / \Delta; \mathcal{A}}\text{initial } \mathcal{A} \qquad \frac{\Sigma : \Psi; \Delta \xrightarrow{F} \mathcal{A} / \Delta'; \mathcal{A}'}{\Sigma : \Psi; \Delta, F \longrightarrow \mathcal{A}; \emptyset / \Delta'; \mathcal{A}'}\text{decide } \Delta$$

$$\frac{\Sigma : \Psi; \Delta \xrightarrow{D_1} \mathcal{A} / \Delta'; \mathcal{A}' \qquad \Sigma : \Psi; \Delta' \xrightarrow{D_2} \mathcal{A}' / \Delta''; \mathcal{A}''}{\Sigma : \Psi; \Delta \xrightarrow{D_1 \invamp D_2} \mathcal{A} / \Delta''; \mathcal{A}''} \invamp\text{-}L$$

Fig. 2. Some lazy sequent rules

3.3 Eager Splitting versus Lazy Splitting: a Matter of Efficiency

Consider proving the Forum sequent $\Sigma : \Psi; a \invamp b \invamp c \longrightarrow a, b, c; \emptyset$. The proof involves splitting the contexts twice, as there are two occurrences of $\invamp$ on the left. All in all, there are 27 possible splits but only one of them is valid. Eager splitting backtracks through these splits until finding out the correct one. On the other hand, as shown in Fig. 3, lazy splitting completely removes this non-determinism finding out the correct split without backtracking. The goals a, b and c flow top-down along the proof tree from left to right. Every lazy proof consumes the formulas it needs and returns the unused ones. Even for a naive proof like this, the efficiency improvement of lazy splitting is striking.

3.4 Preserving Scope: Returnable and Non-Returnable Input

It is quite clear that a lazy proof can return an unused formula as output if and only if the afore-mentioned formula was previously passed to it as input. That

$$
\dfrac{
  \dfrac{
    \dfrac{\dfrac{}{\Sigma : \Psi; \emptyset \xrightarrow{a} a, b, c/\emptyset; b, c}\;\text{initial } \mathcal{A} \qquad \dfrac{}{\Sigma : \Psi; \emptyset \xrightarrow{b} b, c/\emptyset; c}\;\text{initial } \mathcal{A}}{\Sigma : \Psi; \emptyset \xrightarrow{a \parr b} a, b, c/\emptyset; c}\;\parr L
    \qquad
    \dfrac{}{\Sigma : \Psi; \emptyset \xrightarrow{c} c/\emptyset; \emptyset}\;\text{initial } \mathcal{A}
  }{\Sigma : \Psi; \emptyset \xrightarrow{a \parr b \parr c} a, b, c/\emptyset; \emptyset}\;\parr L
}{\Sigma : \Psi; a \parr b \parr c \longrightarrow a, b, c; \emptyset/\emptyset; \emptyset}\;\text{decide } \Delta
$$

Fig. 3. A lazy sequent proof

is, a proof cannot return what it has not previously borrowed. Although this fact seems fairly trivial, it must be explicitly controlled by the lazy rules. To illustrate the underlying problem, consider the naive lazy rules below

$$
\dfrac{\Sigma : \Psi; D, \Delta \longrightarrow \mathcal{A}; G, \Gamma/\Delta'; \mathcal{A}'}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}; D \multimap G, \Gamma/\Delta'; \mathcal{A}'} \multimap\text{-}R \qquad \dfrac{\Sigma : \Psi; \Delta \longrightarrow A, \mathcal{A}; \Gamma/\Delta'; \mathcal{A}'}{\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}; A, \Gamma/\Delta'; \mathcal{A}'}\text{atomic-}R
$$

These apparently innocent rules are unsound. Consider, for example, the unsound lazy proof in Fig. 4. The problem with this proof is that b is *out of scope*, a situation we show typesetting it in **boldface**. The left subproof cannot return any b, as there is no b to split. This flaw is due to the growing of the linear input: as right rules are applied, linear input grows more and more. The important point here is that formulas so added to the linear input cannot be returned by a proof. To avoid returning formulas out of scope, we distinguish two types of linear inputs: the *non-returnable input*, containing formulas which must be consumed; and the *returnable input*, containing formulas which could be consumed. These are referred to as strict and non-strict input in [3], respectively. Therefore, lazy sequents must include both non-returnable – Δ, $\mathcal{A}$ and Γ – and returnable – Π and $\mathcal{B}$ – input contexts

$$
\Sigma : \underbrace{\Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \Gamma}_{input} / \underbrace{\Pi'; \mathcal{B}'}_{output} \qquad \Sigma : \underbrace{\Psi; \Delta; \Pi \xrightarrow{F} \mathcal{A}; \mathcal{B}}_{input} / \underbrace{\Pi';\ \mathcal{B}'}_{output}
$$

Now, lazy rules must be updated to assure that output contexts – Π' and $\mathcal{B}'$ – return the unused returnable linear input only; i.e. $\Pi' \subseteq \Pi$ and $\mathcal{B}' \subseteq \mathcal{B}$ (see Proposition 1). Lazy right rules add the generated subformulas to the non-returnable input to force their consumption

$$
\dfrac{\Sigma : \Psi; D, \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; G, \Gamma/\Pi'; \mathcal{B}'}{\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; D \multimap G, \Gamma/\Pi'; \mathcal{B}'} \multimap\text{-}R
$$

$$
\dfrac{\Sigma : \Psi; \Delta; \Pi \longrightarrow A, \mathcal{A}; \mathcal{B}; \Gamma/\Pi'; \mathcal{B}'}{\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; A, \Gamma/\Pi'; \mathcal{B}'}\text{atomic -}R
$$

Lazy axiom rules assure that non-returnable input has been fully consumed, while lazy unary rules simply propagate contexts

$$
\dfrac{}{\Sigma : \Psi; \emptyset; \Pi \xrightarrow{A} A; \mathcal{B}/\Pi; \mathcal{B}}\text{initial } \mathcal{A} \qquad \dfrac{\Sigma : \Psi; \Delta; \Pi \xrightarrow{F} \mathcal{A}; \mathcal{B}/\Pi'; \mathcal{B}'}{\Sigma : \Psi; \Delta, F; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \emptyset/\Pi'; \mathcal{B}'}\text{decide } \Delta
$$

Again, lazy splitting rules are the most complex ones, requiring our attention. We discuss them in detail in the following section.

$$\dfrac{\dfrac{\dfrac{}{\Sigma : \Psi; a \xrightarrow{c} c/a; \emptyset}\;\text{initial }\mathcal{A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma : \Psi; b \xrightarrow{a} a/b; \emptyset}\;\text{initial }\mathcal{A}}{\Sigma : \Psi; a, b \longrightarrow a; \emptyset/b; \emptyset}\;\text{decide }\Delta}{\Sigma : \Psi; a, b \longrightarrow \emptyset; a/b; \emptyset}\;\text{atomic -}R}{\Sigma : \Psi; a \longrightarrow \emptyset; b \multimap a/\mathbf{b}; \emptyset}\;\multimap\text{-}R}{\Sigma : \Psi; a \xrightarrow{(b\multimap a)\multimap c} c/\mathbf{b}; \emptyset}\;\multimap\text{-}L \qquad \dfrac{\dfrac{\dfrac{}{\Sigma : \Psi; \xrightarrow{\mathbf{b}} b/\emptyset; \emptyset}\;\text{initial }\mathcal{A}}{\Sigma : \Psi; \mathbf{b} \longrightarrow b; \emptyset/\emptyset; \emptyset}\;\text{decide }\Delta}{\Sigma : \Psi; \mathbf{b} \longrightarrow \emptyset; b/\emptyset; \emptyset}\;\text{atomic -}R}{\Sigma : \Psi; a \xrightarrow{b\multimap(b\multimap a)\multimap c} c/\emptyset; \emptyset}\;\multimap\text{-}L$$

Fig. 4. An unsound lazy proof returning b out of scope

3.5 Preserving Scope: a Stack Is Enough for Linear Logic

Consider the lazy splitting rule sketch below

$$\dfrac{\Sigma : \Psi; ?; \Pi \xrightarrow{D_1} ?; \mathcal{B}/\Pi'; \mathcal{B}' \qquad \Sigma : \Psi; ?; \Pi' \xrightarrow{D_2} ?; \mathcal{B}'/\Pi''; \mathcal{B}''}{\Sigma : \Psi; \Delta; \Pi \xrightarrow{D_1 \parr D_2} \mathcal{A}; \mathcal{B}/\Pi''; \mathcal{B}''}\;\parr\text{-}L$$

Returnable input – Π and $\mathcal{B}$ – is managed as before. The problem now is how to deal with non-returnable input – Δ and $\mathcal{A}$ – ; represented by '?' above. On the one hand, non-returnable input must be split between both subproofs. To be lazily split, non-returnable input must be returnable with respect to the left subproof, so we transfer non-returnable input to the returnable input obtaining

$$\Sigma : \Psi; \emptyset; \Delta, \Pi \xrightarrow{D_1} \emptyset; \mathcal{A}, \mathcal{B}/\Delta', \Pi'; \mathcal{A}', \mathcal{B}'$$

On the other hand, the original non-returnable input – Δ and $\mathcal{A}$ – must also be fully consumed by both subproofs. To be fully consumed, the right subproof must consume any remaining portion of the non-returnable input returned by the left subproof as output. Hence, that portion – Δ' and $\mathcal{A}'$ – must be restored from the output of the left subproof and passed to the right subproof as non-returnable input. These previous comments lead to the rule sketch

$$\dfrac{\Sigma : \Psi; \emptyset; \Delta, \Pi \xrightarrow{D_1} \emptyset; \mathcal{A}, \mathcal{B}/\Delta', \Pi'; \mathcal{A}', \mathcal{B}' \qquad \Sigma : \Psi; \Delta'; \Pi' \xrightarrow{D_2} \mathcal{A}'; \mathcal{B}'/\Pi''; \mathcal{B}''}{\Sigma : \Psi; \Delta; \Pi \xrightarrow{D_1 \parr D_2} \mathcal{A}; \mathcal{B}/\Pi''; \mathcal{B}''}\;\parr\text{-}L$$

Finally, it is worth to note that to restore the remaining returnable input – Δ' and $\mathcal{A}'$ –, we need to extract the *most recently added* unused input from the output. This suggests imposing a LIFO discipline, managing returnable input contexts and output contexts as stacks. Our rule sketch is then refined into the definitive lazy rule

$$\frac{\Sigma : \Psi; \emptyset; \Delta :: \Pi \xrightarrow{D_1} \emptyset; \mathcal{A} :: \mathcal{B} / \Delta' :: \Pi'; \mathcal{A} :: \mathcal{B}' \quad \Sigma : \Psi; \Delta'; \Pi' \xrightarrow{D_2} \mathcal{A}'; \mathcal{B}' / \Pi''; \mathcal{B}''}{\Sigma : \Psi; \Delta; \Pi \xrightarrow{D_1 \mathbin{\&\!\!\!\!\!\!\!\rotatebox[origin=c]{180}{\&}} D_2} \mathcal{A}; \mathcal{B} / \Pi''; \mathcal{B}''} \parr\text{-}L$$

where '::' denotes a stack constructor, being its first and second arguments the top and the rest of the stack, respectively.

The idea of lazy splitting is not new [8, 22, 3]. The main contribution of this work is relying on a humble stack for this. Managing contexts as stacks has a number of advantages it deserve to point up. Using a stack is a simple and natural solution to preserve nested scopes within computer science. Consider, for example, the implementation of scope of identifiers in a block-structured programming language. Each time a new block is entered, a stack-frame is allocated to store its local definitions. When the block is exited, its stack-frame is deallocated and its local definitions disappear. In a sense, the left subproof of a splitting connective is similar to a block: pushing non-returnable input and setting it empty allocates a stack-frame for local definitions which cannot be accessed from out of the block. This stack-frame is deallocated when the left subproof is completed. Lazy splitting raises a problem of preserving nested scopes. Therefore, a stack provides a simple, natural and elegant solution.

Furthermore, stacks have also been used to preserve nested scopes within logic programming. Logic languages based on hereditary Harrop formulas – such as λProlog and Elf – manage the logic program as a stack of formulas, pushing onto the program clauses added by intuitionistic implication in goals. This stack-based scoping capability of intuitionistic implication in goals has been used to provide a logical foundation of modular programming [14]. Thus our lazy splitting system reveals a surprising fact: in Linear Logic, just as in hereditary Harrop formulas, a stack is enough for resource management. The stack-based lazy splitting system can then be viewed as a "higher-order" extension of this stack-based management, as both the logic program and the goals are managed as stacks: Ψ, Δ, $\mathcal{A}$ and Γ are stacks of formulas, while Π, $\mathcal{B}$, Π', and $\mathcal{B}'$ are stacks of stacks – i.e. "second-order" stacks – of formulas. Moreover, as suggested in [3], at the implementation level it is possible to use a single data structure to store the contexts. In particular, it is possible to interleave Ψ, Δ and Π into a single program stack and $\mathcal{A}$ and $\mathcal{B}$ into a single atomic goal stack. Thus we envisage that maybe using a single stack makes it possible to reuse results obtained from the research effort invested in the efficient implementation of hereditary Harrop languages. Existing implementations of continuation-passing style interpreters and abstract machines for these could be easily extended for linear logic languages.

In addition, storing the logic program as a single stack is ideal for preserving clause order as well. Although this is usually irrelevant for theorem provers, it lends a logic language a predictable behavior. Following with our block-structured programming language analogy above, visibility assures that local definitions hide external ones, either global or local to an enclosing block.

Both, visibility in block-structured languages and clause order in logic languages are easily implemented traversing the stack top-down.

Finally, the implementation of our stack-based lazy splitting system is also simpler and much more efficient than other proposals. Current lazy splitting systems involve traversing and updating contexts to preserve scope of formulas [22, 3, 20], which takes a time proportional to the cardinalities of the contexts. The efficiency of these operations is critical for the final performance of the lazy splitting system. Of course, pushing onto and popping from a stack are nearly constant-time and inexpensive operations, as they do not depend on the cardinalities of the contexts.

4 Optimizing $\&R$

The lazy splitting system presented removes much – but not all – of the non-determinism from the selection of a suitable split. There are, however, important optimizations proposed by Cervesato et al. [3] related to the treatment of the additive conjunction & and the additive truth $\top$, leading to a fully deterministic and much more efficient lazy splitting system. In this section, we apply the afore-mentioned optimization for the additive conjunction to our stack-based lazy splitting system, and propose a further optimization which may be useful for the implementation of linear logic theorem provers.

4.1 Learning from Experience: Looking-Ahead Needs

The $\&R$ rule constitutes a special case not discussed so far: both subproofs must consume exactly the same linear formulas. The non-returnable input causes no problem, as in any case it must be fully consumed by both subproofs. On the other hand, each subproof consumes a portion of the returnable input and returns the remaining portion as output. To ensure that both subproofs consume exactly the same portion of the returnable input, their output contexts must be synchronized; i.e. they must be equal. Thus we obtain the $\&R$ rule below

$$\frac{\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; G_1, \Gamma/\Pi'; \mathcal{B}' \quad \Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; G_2, \Gamma/\Pi'; \mathcal{B}'}{\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; G_1 \,\&\, G_2, \Gamma/\Pi'; \mathcal{B}'} \,\&\,\mathrm{R}$$

As pointed out in [3], this rule can be optimized. The left subproof determines the portions of returnable the input – Π and $\mathcal{B}$ – consumed. Since we know this consumption, we can take advantage of this knowledge and pass the consumed portions as non-returnable input to the right subproof, enforcing their consumption and removing any non-determinism. The optimized $\&R$ rule is then

$$\frac{\begin{array}{c}\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; G_1, \Gamma/\Pi'; \mathcal{B}' \\ \Sigma : \Psi; \Delta, \Pi - \Pi'; \mathtt{nil} \longrightarrow \mathcal{A}, \mathcal{B} - \mathcal{B}'; \mathtt{nil}; G_2, \Gamma/\Pi'; \mathcal{B}'\end{array}}{\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; G_1 \,\&\, G_2, \Gamma/\Pi'; \mathcal{B}'} \,\&\,\mathrm{R}$$

where `nil` represents the empty stack and "$-$" denotes multiset stack difference, recursively defined as:

$$(P_1 : \Pi_1) \;-\; (P_2 : \Pi_2) =_{def} (P_1 \backslash P_2) \uplus (\Pi_1 - \Pi_2)$$
$$\mathtt{nil} - \mathtt{nil} =_{def} \emptyset$$

where "\\" denotes multiset difference. Note that the above multiset stack difference is defined for stacks with the same depth only. In this setting, the multiset difference $P_1 \backslash P_2$ is always applied to multisets such that $P_2 \subseteq P_1$ (see Proposition 1). This fact could also be used to improve the efficiency of an implementation of the multiset stack difference. Of course, the optimization just described can only be applied to a sequential implementation of lazy splitting.

4.2 Avoiding Replication of Work by Delaying $\&R$

The treatment of the additive conjunction can be further optimized. As shown in the lazy $\&R$ rule above, goals to decompose – Γ – are replicated and passed to both subproofs, thus an identical goal decomposition task is carried out by them. Note that even in a parallel implementation of lazy splitting executed on a multiprocessor machine, work is not distributed but replicated. As Forum connectives permute over each other on the right-hand side, right rules can be applied in any order. To avoid the replication of work, the treatment of the additive conjunction & can be delayed until the other right rules have been applied. Unfortunately, goal decomposition has a side-effect on the logic program, as linear and intuitionistic implications in goals add clauses to it. Therefore, this optimization does not preserve clause order. Maybe, it could be possible to design an implementation delaying & and restoring clause order by merging program stacks, but its complexity seems to undermine any benefit. On the other hand, the optimization just described is trivial to implement in a linear logic theorem prover not preserving clause order.

5 Lazy Consumption: $\top$ Is Insatiable

The contents of the output are determined transferring unused returnable input at the leaves of the proof tree. The *initial* $\mathcal{A}$ and $\bot L$ rules transfer all the returnable input untouched

$$\frac{}{\Sigma : \Psi; \emptyset; \Pi \xrightarrow{A} A; \mathcal{B}/\Pi; \mathcal{B}}\,\text{initial } \mathcal{A} \qquad \frac{}{\Sigma : \Psi; \emptyset; \Pi \xrightarrow{\bot} \emptyset; \mathcal{B}/\Pi; \mathcal{B}}\,\bot\text{-}L$$

but the $\top R$ rule consumes all of the non-returnable input as well as an unknown portion of the returnable input

$$\frac{}{\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \top, \Gamma/\Pi'; \mathcal{B}'}\,\top\text{R}$$

Determining the final consumption of $\top$, however, is not trivial. Recall the key idea leading to lazy splitting: linear logic proofs are satiable; that is, they consume what they need, but no more. But $\top$ wants it all! In a sense, while the others connectives are satiable, $\top$ is insatiable. In the $\top R$ rule above, the returnable input must be split into two portions: one consumed by $\top$ and the other one returned as output. Thus we are faced with a context splitting problem again. Nevertheless, this splitting requires a quite different technique to be done lazily, as a portion of the formulas affected by it could have been consumed. In other words, this is not exactly lazy splitting but lazy consumption. Cervesato et al. [3] have proposed a very elegant solution to this problem where the real consumption of $\top$ is delayed. We shall present here how to incorporate their solution to our stack-based lazy splitting system.

The main idea is attaching a boolean *discard* indicator – referred to as *slack* indicator in [3] – to the output, denoting if it has been returned by a $\top$ or not. The lazy sequents have then the final form

$$\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \Gamma/\Pi; \mathcal{B}; d \quad \text{and} \quad \Sigma : \Psi; \Delta; \Pi \xrightarrow{D} \mathcal{A}; \mathcal{B}/\Pi'; \mathcal{B}'; d$$

being d the referred *discard* indicator. A $\top R$ rule consuming a portion of the returnable input could remove a formula which should be consumed in another proof; and hence such a rule must be combined with backtracking. To avoid this, as shown in Fig. 5, the lazy $\top R$ rule does not consume any portion of the returnable input but transfers it to the output and sets the discard indicator to 1. By returning all of the returnable input, the real consumption of $\top$ is delayed and all of the returnable input is made available to be consumed elsewhere. On the other hand, the consumption of the output returned by a $\top$ is no longer mandatory, as in any case it could have be consumed by that $\top$. Thus, if a portion of the returned output is not consumed elsewhere, it is assigned to any $\top$ it has visited and silently discarded. Thus the final consumption of $\top$ is lazily determined when the unused output is discarded. Of course, the other lazy axiom rules set the indicator to 0, as their output must be consumed elsewhere. Lazy unary rules are trivial, as they simple propagate both the contexts and the discard indicator. The lazy binary rules $\& R$, $⅋ L$ and $\multimap L$ are split into two rules, depending on the value of the discard indicator returned by the left subproof.

The key of the $\& R$ rule is that both subproof must consume the same input. Thus in a sequential implementation, we force the right subproof to consume the same as the previously completed left subproof. If the discard indicator of the left subproof is set to 0 (rule $\&_0 R$ in Fig. 5), then the consumption of this subproof has been fully determined and we pass it as non-returnable input to the right subproof. If the discard indicator of the left subproof is set to 1 (rule $\&_1 R$ in Fig. 5), then it could consume an arbitrary portion of its output; i.e. there is a lazy consumption. Thus we pass to the right subproof the known consumption of the left subproof as non-returnable input and the output of the left subproof as returnable input. The overall output of the proof is the output of the right subproof, as the left subproof lazily consumes any portion of its output in order

$$\frac{}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};\top,\Gamma/\Pi;\mathcal{B};1}\ \top\text{-}R \qquad \frac{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};\Gamma/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};\bot,\Gamma/\Pi';\mathcal{B}';d}\ \bot\text{-}R$$

$$\frac{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G_1,\Gamma/\Pi';\mathcal{B}';0 \quad \Sigma:\Psi;\Delta,\Pi-\Pi';\mathtt{nil} \longrightarrow \mathcal{A},\mathcal{B}-\mathcal{B}';\mathtt{nil};G_2,\Gamma/\mathtt{nil};\mathtt{nil};d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G_1 \,\&\, G_2,\Gamma/\Pi';\mathcal{B}';0}\ \&_0\text{-}R$$

$$\frac{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G_1,\Gamma/\Pi';\mathcal{B}';1 \quad \Sigma:\Psi;\Delta,\Pi-\Pi';\Pi' \longrightarrow \mathcal{A},\mathcal{B}-\mathcal{B}';\mathcal{B}';G_2,\Gamma/\Pi'';\mathcal{B}'';d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G_1 \,\&\, G_2,\Gamma/\Pi'';\mathcal{B}'';d}\ \&_1\text{-}R$$

$$\frac{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G_1,G_2,\Gamma/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G_1 \parr G_2,\Gamma/\Pi';\mathcal{B}';d}\ \parr\text{-}R$$

$$\frac{\Sigma:\Psi;\Delta,D;\Pi \longrightarrow \mathcal{A};\mathcal{B};G,\Gamma/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};D \multimap G,\Gamma/\Pi';\mathcal{B}';d}\ \multimap\text{-}R \qquad \frac{\Sigma:\Psi,D;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G,\Gamma/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};D \Rightarrow G,\Gamma/\Pi';\mathcal{B}';d}\ \Rightarrow\text{-}R$$

$$\frac{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};G[y/x],\Gamma/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};\forall x.G,\Gamma/\Pi';\mathcal{B}';d}\ \forall\text{-}R \qquad \frac{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A},A;\mathcal{B};\Gamma/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};A,\Gamma/\Pi';\mathcal{B}';d}\ \text{atomic -}R$$

$$\frac{\Sigma:\Psi;\Delta;\Pi \xrightarrow{D} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta,D;\Pi \longrightarrow \mathcal{A};\mathcal{B};\emptyset/\Pi';\mathcal{B}';d}\ \text{decide } \Delta$$

$$\frac{\Sigma:\Psi;\Delta;\Delta_1:\cdots:\Delta_i:\Pi \xrightarrow{D} \mathcal{A};\mathcal{B}/\Delta_1'::\cdots::\Delta_i':\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Delta_1::\cdots::\Delta_i,D:\Pi \longrightarrow \mathcal{A};\mathcal{B};\emptyset/\Delta_1'::\cdots::\Delta_i':\Pi';\mathcal{B}';d}\ \text{decide } \Pi$$

$$\frac{\Sigma:\Psi,D;\Delta;\Pi \xrightarrow{D} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d}{\Sigma:\Psi,D;\Delta;\Pi \longrightarrow \mathcal{A};\mathcal{B};\emptyset/\Pi';\mathcal{B}';d}\ \text{decide } \Psi$$

$$\frac{}{\Sigma:\Psi;\emptyset;\Pi \xrightarrow{A} A;\mathcal{B}/\Pi;\mathcal{B};0}\ \text{initial } \mathcal{A}$$

$$\frac{}{\Sigma:\Psi;\emptyset;\Pi \xrightarrow{A} \emptyset;\mathcal{A}_1:\cdots:\mathcal{A}_i,A:\mathcal{B}/\Pi;\mathcal{A}_1:\cdots:\mathcal{A}_i:\mathcal{B};0}\ \text{initial } \mathcal{B}$$

$$\frac{}{\Sigma:\Psi;\emptyset;\Pi \xrightarrow{\bot} \emptyset;\mathcal{B}/\Pi;\mathcal{B};0}\ \bot\text{-}L \qquad \frac{\Sigma:\Psi;\Delta;\Pi \xrightarrow{D_i} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \xrightarrow{D_1 \& D_2} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d}\ \&_i\text{-}L\ (i=1,2)$$

$$\frac{\Sigma:\Psi;\Delta;\Pi \xrightarrow{D[t/x]} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d}{\Sigma:\Psi;\Delta;\Pi \xrightarrow{\forall x.D} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d}\ \forall\text{-}L$$

$$\frac{\Sigma:\Psi;\Delta;\Pi \xrightarrow{D} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d_1 \quad \Sigma:\Psi;\emptyset;\mathtt{nil} \longrightarrow \emptyset;\mathtt{nil};G/\mathtt{nil};\mathtt{nil};d_2}{\Sigma:\Psi;\Delta;\Pi \xrightarrow{G \Rightarrow D} \mathcal{A};\mathcal{B}/\Pi';\mathcal{B}';d_1}\ \Rightarrow\text{-}L$$

$$\frac{\Sigma:\Psi;\emptyset;\Delta::\Pi \xrightarrow{D_1} \emptyset;\mathcal{A}::\mathcal{B}/\Delta'::\Pi';\mathcal{A}'::\mathcal{B}';0 \quad \Sigma:\Psi;\Delta';\Pi' \xrightarrow{D_2} \mathcal{A}';\mathcal{B}'/\Pi'';\mathcal{B}'';d}{\Sigma:\Psi;\Delta;\Pi \xrightarrow{D_1 \parr D_2} \mathcal{A};\mathcal{B}/\Pi'';\mathcal{B}'';d}\ \parr_0\text{-}L$$

$$\frac{\Sigma:\Psi;\emptyset;\Delta::\Pi \xrightarrow{D_1} \emptyset;\mathcal{A}::\mathcal{B}/\Delta'::\Pi';\mathcal{A}'::\mathcal{B}';1 \quad \Sigma:\Psi;\emptyset;\Delta'::\Pi' \xrightarrow{D_2} \emptyset;\mathcal{A}'::\mathcal{B}'/\Delta''::\Pi'';\mathcal{A}''::\mathcal{B}'';d}{\Sigma:\Psi;\Delta;\Pi \xrightarrow{D_1 \parr D_2} \mathcal{A};\mathcal{B}/\Pi'';\mathcal{B}'';1}\ \parr_1\text{-}L$$

$$\frac{\Sigma:\Psi;\emptyset;\Delta::\Pi \xrightarrow{D} \emptyset;\mathcal{A}::\mathcal{B}/\Delta'::\Pi';\mathcal{A}'::\mathcal{B}';0 \quad \Sigma:\Psi;\Delta';\Pi' \longrightarrow \mathcal{A}';\mathcal{B}';G/\Pi'';\mathcal{B}'';d}{\Sigma:\Psi;\Delta;\Pi \xrightarrow{G \multimap D} \mathcal{A};\mathcal{B}/\Pi'';\mathcal{B}'';d}\ \multimap_0\text{-}L$$

$$\frac{\Sigma:\Psi;\emptyset;\Delta::\Pi \xrightarrow{D} \emptyset;\mathcal{A}::\mathcal{B}/\Delta'::\Pi';\mathcal{A}'::\mathcal{B}';1 \quad \Sigma:\Psi;\emptyset;\Delta'::\Pi' \longrightarrow \emptyset;\mathcal{A}'::\mathcal{B}';G/\Delta''::\Pi'';\mathcal{A}''::\mathcal{B}'';d}{\Sigma:\Psi;\Delta;\Pi \xrightarrow{G \multimap D} \mathcal{A};\mathcal{B}/\Pi'';\mathcal{B}'';1}\ \multimap_1\text{-}L$$

Fig. 5. Forum Lazy Proof System $\mathcal{LF}$

to consume the same as the right one. The overall discard indicator of the proof is set to 1 if and only if both subproofs set it to 1, so the discarded overall output is lazily consumed by both subproofs.

The $\parr L$ rule must assure that the non-returnable input is both lazily split into and fully consumed by its subproofs. If the discard indicator of the left subproof is set to 0 (rule $\parr_0 R$ in Fig. 5), then the unused non-returnable input must be popped from the stack and passed as non-returnable input to the right subproof to force its consumption. If the discard indicator of the left subproof is set to 1 (rule $\parr_1 R$ in Fig. 5), then there is no need to force the consumption of the unused non-returnable input – as it could be lazily consumed by the left subproof – and therefore, it is not popped from the stack but passed as returnable input. However, since non-returnable input must be fully consumed by both subproofs, the unused remaining portion is popped from the output stack of the right subproof and discarded, becoming the lazy consumption of the left subproof. The overall discard indicator of the $\parr_i L$ rules is set to 1 if one of the subproofs set its indicator to 1, so the discarded output is lazily consumed by that subproof. The lazy left rules for $\multimap$ are isomorphic.

6 Soundness and Completeness

The $\mathcal{LF}$ lazy proof system in Fig. 5 is logically equivalent to the $\mathcal{F}'$ proof system in Fig. 1. The correctness and completeness theorems are enunciated below. The inclusion proposition states an important property of the lazy proof system; namely, scope of formulas is preserved. Formal proofs – by mutual structural induction – and auxiliary lemmas are omitted due to lack of space.

Definition 2. *We define an inclusion relation $\sqsubseteq$ on stacks of multisets as follows:*

1. $\mathtt{nil} \sqsubseteq \mathtt{nil}$
2. $A_1 : \Pi_1 \sqsubseteq A_2 : \Pi_2 \Leftrightarrow (A_1 \subseteq A_2) \wedge (\Pi_1 \sqsubseteq \Pi_2)$

To prove the Theorems 1 and 2, we need a number of auxiliary results that we do not include here. Some of them describe interesting properties of the proposed lazy proof system. We only present the following proposition, which states an essential feature involving scope preservation.

Proposition 1 (Inclusion). *The output is a substack of the returnable input*

1. If $\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \Gamma / \Pi'; \mathcal{B}'; d$ then $(\Pi' \sqsubseteq \Pi) \wedge (\mathcal{B}' \sqsubseteq \mathcal{B})$
2. If $\Sigma : \Psi; \Delta; \Pi \xrightarrow{D} \mathcal{A}; \mathcal{B} / \Pi'; \mathcal{B}'; d$ then $(\Pi' \sqsubseteq \Pi) \wedge (\mathcal{B}' \sqsubseteq \mathcal{B})$

In the following results, we refer to the $\mathcal{LF}$ or $\mathcal{F}'$ sequent calculus depending on the aspect of the involved sequent.

Theorem 1 (Soundness). *$\mathcal{LF}$ is sound with respect to $\mathcal{F}'$*

1. If $\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \Gamma / \Pi'; \mathcal{B}'; 0$ then $\Sigma : \Psi; \Delta, \Pi - \Pi' \longrightarrow \mathcal{A}, \mathcal{B} - \mathcal{B}'; \Gamma$

2. If $\Sigma : \Psi; \Delta; \Pi \xrightarrow{D} \mathcal{A}; \mathcal{B}/\Pi'; \mathcal{B}'; 0$ then $\Sigma : \Psi; \Delta, \Pi - \Pi' \xrightarrow{F} \mathcal{A}, \mathcal{B} - \mathcal{B}'$
3. If $\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \Gamma/\Pi'; \mathcal{B}'; 1$ then $\Sigma : \Psi; \Delta, \Pi - \Pi', \Delta' \longrightarrow \mathcal{A}, \mathcal{B} - \mathcal{B}', \mathcal{A}'; \Gamma$
4. If $\Sigma : \Psi; \Delta; \Pi \xrightarrow{D} \mathcal{A}; \mathcal{B}/\Pi'; \mathcal{B}'; 1$ then $\Sigma : \Psi; \Delta, \Pi - \Pi', \Delta' \xrightarrow{F} \mathcal{A}, \mathcal{B} - \mathcal{B}', \mathcal{A}'$

For all $\Delta', \mathcal{A}'$

This result establishes a mapping between lazy and eager proofs. Basically, a lazy proof is mapped into an eager proof receiving the portion of the returnable input which is actually consumed ($\Pi - \Pi'$ and $\mathcal{B} - \mathcal{B}'$). In addition, if the lazy proof returns a discard indicator set to 1, then the eager proof can be obtained even if more resources (Δ' and $\mathcal{A}'$) were considered.

Theorem 2 (Completeness). *$\mathcal{LF}$ is complete with respect to $\mathcal{F}'$*

1. If $\Sigma : \Psi; \Delta \longrightarrow \mathcal{A}; \Gamma$ then $\Sigma : \Psi; \Delta; \Pi \longrightarrow \mathcal{A}; \mathcal{B}; \Gamma/\Pi; \mathcal{B}; d$
2. If $\Sigma : \Psi; \Delta \xrightarrow{F} \mathcal{A}$ then $\Sigma : \Psi; \Delta; \Pi \xrightarrow{D} \mathcal{A}; \mathcal{B}/\Pi; \mathcal{B}; d$

For all $\Pi, \mathcal{B}$ *and* $d \in \{0, 1\}$

The completeness theorem states that every eager proof is mapped into a lazy proof, where all of the additional returnable input (Π and $\mathcal{B}$) is simply returned.

7 Related Work

The problem of resource management in linear logic proofs – and particularly the context splitting problem – is an active research field, and has received a certain amount of attention from the Linear Logic community. This is not surprising, since the successful implementation of practical linear logic languages and theorem provers strongly depends on the efficient management of linear resources.

The first attempt to reduce this non-determinism was the *input-output model of resource consumption* proposed by Hodas and Miller in [9]. This solution relied on list operations and partially solved this problem, as it managed $\top$ non-deterministically. Subsequently, Hodas provided a deterministic lazy treatment of $\top$ in his Ph.D. thesis [8].

Winikoff and Harland [21] developed a *lazy splitting system* for Lygon based on the Lolli input-output model [9]. In short, their system is based on tags: a formula is tagged to denote that its consumption is optional. There is a second tag denoting that the formula has visited a $\top$ occurrence. As the proof is constructed the system suitably tags and untags formulas, dealing with formulas individually instead of whole contexts which leads to a much less efficient implementation.

Cervesato et al. [3] proposed a *resource management system* for Lolli. Their system improves the treatment of the additive conjunction & and of the additive constant $\top$. The system relies on the use of costly multiset operations as intersections and unions. Also, for their system to be sound and complete, the formulas must be individually labelled to distinguish different occurrences from each other. These labels are not required in our lazy splitting system.

Urban [20] presented a resource management system for Forum based on [3]. This system is described using a quite visual notation – the box calculus – and still relies on costly multiset operations. The main novelty of his system is that it does not force consumption on the input but on the output by distinguishing *strict* and *slack* output. Unfortunately, checking for consumption constraints on the output is less efficient than checking for them on the input.

López and Pimentel [12] proposed a lazy splitting system for Forum based on [21]. It included a λProlog prototype implementation of Forum. The way this implementation managed lists representing the logic program and goals suggested us using a stack to preserve scope of resources. In particular, we realized that the number of tags attached to a given formula corresponded to the depth in the stack. Thus we began to cherish the idea of a stack-based lazy proof system and initiated a search for a suitable notation for it. Finally, the resource management system proposed by Cervesato et al. [3] – to which our work owes much – provided both a final insight into the resource management problem and a suitable notation for our stack-based approach.

8 Conclusions and Further Work

Lazy splitting strongly improves the performance of the implementation of any linear logic language or prover involving context splitting. We have implemented UMA Forum, a prototype interpreter of a subset of Forum supporting both eager and lazy splitting execution modes. This implementation has been used to compare the performance of both strategies. The results obtained show that using a lazy splitting strategy is essential, as problems taking several hours when solved in eager mode are solved within milliseconds in lazy mode. We are interested in studying the way to incorporate our stack-based system to other implementation techniques, such as continuation-passing style interpreters and abstract machines. Given the close relationship between Linear Logic and concurrency, we intend to use it as a sound foundation for a logic programming language providing a clean declarative notion of concurrent object. Of course, a truly concurrent implementation of such a language requires the parallelization of our stack-based lazy splitting system.

9 Acknowledgments

The authors are very grateful to Iliano Cervesato and Christian Urban for their helpful comments on previous versions of this work. Pablo López is supported by the Spanish Ministry of Education and Culture through a Ph.D. Research Scholarship. This work has been partially supported by the CICYT project TIC95-0433-C03-02.

References

1. J.M. Andreoli and R. Pareschi. "Linear Objects: Logical Processes with Built-in Inheritance". *New Generation Computing*, 9(3-4):445–473, 1991.
2. J.M. Andreoli. "Logic Programming with Focusing Proofs in Linear Logic". *Journal of Logic and Computation*, 2(3):445–473, 1992.
3. I. Cervesato, J. Hodas and F. Pfenning. "Efficient Resource Management for Linear Logic Proof Search". *Proceedings of the 1996 International Workshop on Extensions of Logic Programming*, 1996.
4. M. M. T. Chakravarty. *On the Massively Parallel Execution of Declarative Programs.* PhD Thesis, Technische Universität Berlin, Fachbereich Informatik, 1996.
5. J. Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD Thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
6. G. Delzanno. *Logic & Object-Oriented Programming in Linear Logic.* PhD Thesis, Department of Computer Science, University of Pisa-Genova-Udine, 1997.
7. J-Y. Girard. "Linear Logic". *Theoretical Computer Science*, 50, pp. 1–102. 1987.
8. J.S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation.* PhD Thesis. University of Pennsylvania, Department of Computer and Information Science, 1994.
9. J.S. Hodas and D. Miller. "Logic Programming in a Fragment of Intuitionistic Linear Logic". *Information and Computation*, 110(2), pp. 327–365, 1994.
10. J.S. Hodas and J. Polakow. "Forum as a Logic Programming Language: Preliminary Report". *Linear Logic 1996: extended abstracts and preliminary results.* Also in *Electronic Notes in Theoretical Computer Science, 3.* Elsevier-North Holland, 1996.
11. N. Kobayashi and A. Yonezawa. "ACL – a Concurrent Linear Logic Programming Paradigm". In *International Logic Programming Symposium*, pp. 279–294, 1993.
12. P. López . *Implementación de un Lenguaje de Programación basado en Lógica Lineal.* Master's Thesis supervised by E. Pimentel. Computer Science Department. University of Málaga. 1996.
13. P. López and E. Pimentel. "A Lazy Splitting System for Forum". In M. Falaschi, M. Navarro and A. Policriti, editors, *Proceedings of the Joint Conference on Declarative Programming – APPIA-GULP-PRODE'97*, pp. 247–258, Grado, Italy, 1997.
14. D. Miller. "Abstractions in Logic Programming". In P. Odifreddi *Logic in Computer Science*, Academic Press, pp. 329–359, 1990.
15. D. Miller, G. Nadathur, F. Pfenning and A. Scedrov. "Uniform proofs as a foundation for logic programming". *Annals of Pure and Applied Logic*, 51:125-157, 1991.
16. D. Miller. "A Multiple-Conclusion Meta-Logic". S. Abramsky, editor, *Proceedings of the 1994 Symposium on Logic in Computer Science*, pp. 272–281, 1994.
17. D. Miller. "A Survey of Linear Logic Programming". *Computational Logic: The Newsletter of the European Network in Computational Logic*, vol. 2, 2, pp. 63–67, 1995.
18. D. Miller. "Forum: A Multiple-Conclusion Specification Logic". *Theoretical Computer Science*, 165(1), pp.201–232, 1996.
19. A. Scedrov. "A Brief Guide to Linear Logic". *Bulletin of the European Association for Theoretical Computer Science*, 41:154–165, 1990.
20. C. Urban. *Forum and its Implementation.* MPhil Thesis. Department of Computer Science, University of St. Andrews, 1996.
21. M. Winikoff and J. Harland. "Deterministic Resource Management for the Linear Logic Programming Language Lygon". Technical Report 94/23. University of Melbourne. Department of Computer Science, 1994.
22. M. Winikoff and J. Harland. "Implementing the Linear Logic Programming Language Lygon". In J. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pp. 66–80, 1995.