Isolating Resource Consumption in
Linear Logic Proof Search
(extended abstract)

Pablo Lópeza, Ernesto Pimentela, Joshua S. Hodasb, Jeffrey Polakowb, and Lubomira Stoilovab

a Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Campus de Teatinos, 29071 Málaga, Spain
[lopez,ernesto]@lcc.uma.es
http://www.lcc.uma.es/~[lopez,ernesto]
b Department of Computer Science
Harvey Mudd College
Claremont, CA 91711, USA
[hodas,jpolakow,lstoilova]@cs.hmc.edu
http://www.cs.hmc.edu/~hodas

Abstract
This work presents an extension of the Tag-Frame resource management system previously developed by the authors. The extended proof system is able to isolate the consumption of a given goal/clause without incurring significant extra runtime costs. We believe this feature may have a number of applications, in particular for debugging linear logic programs and specifications in a proof-theoretic setting. This point is illustrated by means of a simple example.

1 Introduction
It is well known that linear logic languages and logical frameworks alike are powerful conceptual tools for the specification of a number of systems. The

1 Lópeza and Pimentel were partially supported by the project TIC2001-2705-C03-02 funded by the Spanish Ministry of Science and Technology. During the summer of 2001, Lópeza was also partially supported as a visiting researcher by a travel grant from Harvey Mudd College.

2 Stoilova was partially supported by a grant from the Harvey Mudd College Computer Science Clinic Program.

©2002 Published by Elsevier Science B. V.
clean, declarative notion of state provided by linearity opens the door to elegant and perspicuous encodings of sequential and concurrent systems. The expressive power of linear logic, however, poses difficult problems to implementors. Simply stated, efficient proof search in linear logic is hard to implement. The preceding years have witnessed a lot of work aimed at providing efficient proof search strategies. The permutability and invertibility of the sequent rules have been thoroughly studied and the results obtained have been effectively applied to formulate the strategies of focussing [12] and uniformity [12]. The important (and maybe more specific to linear logic) topic of resource management has also attracted a fair amount of interest, and quite a number of proof systems have been developed for different fragments of the logic [9][6][3][11][18].

Surprisingly, however, the topic of debugging linear logic programs and specifications has been largely ignored. Frequently, a linear logic program fails just because the pattern of consumption of resources is not the one that the programmer was expecting. For example, a few resources may remain to be consumed, preventing the proof from being completed. When faced to such a situation, a simple no answer is completely useless.

In this work, we propose an extension of the traditional input/output operational semantics of resource management systems. The extended system is able to isolate the consumption of a given goal/clause without incurring significant extra runtime costs. We argue that the isolation of consumption provides a first step to a proof-theoretic approach to debugging. Some other possible applications of this feature may include profiling, program analysis and optimization, and programming idioms.

This paper is structured as follows. In section 2 the problem of resource distribution in linear logic is posed and two basic approaches to solve it are briefly reviewed. Next section briefly describes the TF resource management system, previously developed by the authors [8]. Section 4 presents the TFD proof system, an extension of the TF resource management system that isolates the consumption of resources. The debugging capabilities of the extended system are then illustrated by means of a simple example. Finally, conclusions and further work are outlined.

2 Resource Management Systems in a Nutshell

The problem of resource management in linear logic is best exemplified by the bottom-up application of the \(-\circ_L\) rule:

\[
\frac{\Delta_1 \rightarrow A}{\Delta_1, A \circ B \rightarrow C} \quad \frac{\Delta_2, B \rightarrow C}{\Delta = \Delta_1 \uplus \Delta_2}
\]

when applying this rule bottom-up, the context \(\Delta\) must be split in two contexts \(\Delta_1\) and \(\Delta_2\). The number of splittings is exponential in the number of formulae in \(\Delta\), thus simply backtracking over all of them would introduce an exponential
choice point in the proof search. This is clearly intolerable for a practical implementation of a programming language or theorem prover. There are two approaches to overcome this problem: *ad hoc* operational semantics and distribution constraints.

The *ad hoc* operational semantics approach modifies the contexts and the rules of the proof system to impose a fixed proof strategy called the input/output model of resource consumption. The key idea of this strategy is that the context $\Delta$ is not split but passed as input to the left premiss. Then the left subproof consumes a portion ($\Delta_1$) of the input and returns a leftover ($\Delta_2$) as output. This output is finally forwarded as the input to the right premiss. In that way, $\Delta$ is lazily split into $\Delta_1$ and $\Delta_2$. While it is true that fixing the proof strategy may reduce the applicability of this approach, it also endows the proof search with a predictable operational semantics, an essential feature of logic programming languages. Quite a number of these systems have been developed [9,7,6,3,11,8], and most of them have been effectively and efficiently implemented. Furthermore, the implementation technology has matured enough as to yield a compiled version for a significant fragment of the Lolli language [10].

The distribution of resources via constraints [5,2] is a more general, algebraic approach where the specification of the problem and the method of solution are clearly distinguished. Boolean expressions, possibly containing boolean variables, are attached to every formula in the context. The intuition behind this idea is that a formula is really present in the context if and only if the attached boolean expression; i.e. the distribution constraint, evaluates to true. The rules of the proof system are then modified to generate and propagate distribution constraints, thus the proof search poses a constraint system –i.e. a distribution problem – that has somehow to be solved. The method of solution determines both the final distribution and the proof strategy. From a theoretical point of view, the generality of these systems turns them into frameworks where different distribution strategies can be analyzed, combined, and compared. From a practical point of view, the relationship between the generation/propagation of constraints and its solution has not been addressed [5], and therefore the implementation issues are not as evolved as in the *ad hoc* operational semantics approach. In fact, the only implementation the authors are aware of is the Iktara parallel theorem prover [4] developed in the context of the Concert project.

### 3 The Tag-Frame System

The Tag-Frame system [8] ($\mathcal{TF}$), shown in Fig.1, is a resource management system that follows the *ad hoc* operational semantics tradition and embodies most of best aspects of such systems previously developed. This includes the input/output flow of the seminal solution [9], the lazy treatment of $\top$ [7], the separation of contexts into lax and strict zones [3], and the optimized
The main contribution of the \textit{T\!F} system is a novel optimization of the additive conjunction \&. Hitherto, the best implementation technique known to deal with the additive conjunction has been the optimization proposed by Cervesato, Hodas, and Pfenning in [3]. Nevertheless, for this optimization to work properly, both the input and output of the left premiss have to be stored and processed to compute the input of the right premiss. Furthermore, in some cases the intersection of the output of both premises must be computed to produce the output of the overall rule. In practice, this imposes a severe load in both memory and time performance. The \textit{T\!F} system avoids these extra computations. A \textit{T\!F} sequent is of the general form:

$$\Delta_I \Delta_O \delta : \pi \xrightarrow{\sigma} \sigma' v G$$

where:

- $\Delta_I$ is the input context, a multiset of tagged formulas.
- $\Delta_O$ is the output context, a multiset of tagged formulas. Note that $\Delta_I$ and $\Delta_O$ have the same cardinality. Formulas in the \textit{T\!F} system are not removed from the context when used but are simply marked as consumed. This is essential for the optimization of \& to work. In addition, it may help in implementing backtracking efficiently.
- $\delta$ is a set of tags. Input formulas tagged with a member of $\delta$ are strict. In contrast to the \textit{Frame} system [11], strict formulas are not consecutive but may be scattered through the input context. The system thereby retains clause order, which is the desired behavior for a logic programming language.
- $\pi$ is a stack of frames (i.e. sets) of tags. Input formulas tagged with a member of $\pi$ are optional (or lax).
- $\sigma$ is a set of tags, usually referred to as the consumption markers. Each time an input formula is consumed, its tag is replaced by an arbitrary element of $\sigma$.
- $\sigma'$ is a set of tags. Output formulas tagged with a member of $\sigma'$ have been effectively consumed.
- $v$ is the usual output $\top$-flag indicating that a $\top$ was encountered in the subtree, and that unconsumed resources may be implicitly weakened.
- $G$ is the goal formula.

The inner workings of the \textit{T\!F} system are complex, and beyond the scope of this paper. The system is explained and its soundness and completeness are discussed in [8]. Nevertheless, the following formal result (whose statement is slightly simplified here) is essential to justify the extended system proposed in this work.

\textbf{Lemma 3.1 (Local Consumption)} \textit{For all $\Delta_I$, $\Delta_O$, $\delta$, $\pi$, $\sigma$, $\sigma'$, $v$, and $G$}
The key idea of the \( T\mathcal{F} \) system is that the left premiss of the \( T\mathcal{F} \&\mathcal{R} \) rule provides a trace of its consumption so that the right premiss has that trace to follow. This is accomplished by introducing a new consumption marker \( d \) that
isolates the consumption of the left premiss. This idea can actually be applied to any sequent rule so that the consumption of any goal can be isolated.

In particular, consider the \( \text{T F pick} \) rule:

\[
\frac{\Delta_L D^d \Delta_R \Delta_O \pi \sigma' \sigma \mid D \gg A}{\Delta_L D^t \Delta_R \Delta_O \pi \sigma' \sigma \mid A} \quad \text{T F pick}
\]

\( t \in \pi, d \in \sigma \)

A new pick rule can be defined in order to isolate the consumption of the goal introduced by a given clause \( D \) as follows:

\[
\frac{\Delta_L D^s \Delta_R \Delta_O \pi \sigma' \{d\} \mid D \gg A}{\Delta_L D^t \Delta_R \Delta_O \pi \sigma' \sigma \mid A} \quad \text{T F pick}
\]

\( t \in \pi, s \in \sigma, d \text{ new} \)

Note that a new consumption marker \( d \) has been introduced to mark and hence isolate the consumption of \( D \gg A \). Since \( d \) is new, \([\Delta_L D^t \Delta_R \{d\}] = \emptyset\) and then, by the local consumption lemma, the consumption of \( D \gg A \) is simply \([\Delta_O]_{\sigma'}\). In addition, the overall consumption is now \([\Delta_O]_{\sigma' \cup \sigma}\); i.e. the resources already consumed plus those consumed by \( D \gg A \).

Instead of replacing the original \( \text{T F pick} \) rule by this new one, a new proof system \( \text{T FD} \) (see Fig.2) is defined that keeps both versions of the pick rule. The \( \text{T FD} \) system is not only a resource management system, since tags are used not only for imposing consumption constraints but for debugging purposes. As far as we know, this is the first time the operational semantics of a resource management system is extended to cope with a notion other than consumption constraints. Of course, not every clause needs to be debugged. The new version of the pick rule will be applied only to those clauses so chosen by the user. This is similar to setting a spy-point in a debugger. Clauses being debugged are shown in the proofs with a tilde on them (\( \tilde{D} \)). The two pick rules of the \( \text{T FD} \) system are then:

\[
\frac{\Delta_L D^d \Delta_R \Delta_O \pi \sigma' \sigma \mid D \gg A}{\Delta_L D^t \Delta_R \Delta_O \pi \sigma' \sigma \mid A} \quad \text{T FD pick}
\]

\( t \in \pi, d \in \sigma \)

\[
\frac{\Delta_L \tilde{D}^s \Delta_R \Delta_O \pi \sigma' \{d\} \mid D \gg A}{\Delta_L \tilde{D}^t \Delta_R \Delta_O \pi \sigma' \sigma \mid A} \quad \text{T FD pick & mark}
\]

\( t \in \pi, s \in \sigma, d \text{ new} \)

Using the \( \text{T FD pick & mark} \) rule, the consumption of an application \( D \gg A \) of a clause being debugged \( \tilde{D} \) is \([\Delta_O]_{\sigma'}\).

It may be argued that the consumption of \( D \gg A \) can also be determined applying the local consumption lemma to the conclusion of the original \( \text{T F pick} \) rule; that is:

\[
[\Delta_O]_{\sigma'} - [\Delta_I]_{\sigma} = [\Delta_I]_{\hat{\pi}} \cup ([\Delta_I]_{\hat{\pi}} - [\Delta_O]_{\hat{\pi}})
\]
On the other hand, the TFD system does not incur such space overload.

A distinctive feature of linear logic programs is that they grow and shrink over the course of execution. In particular, clauses can be dynamically added to the program by the linear implication. However, the user can only set spy-points on the clauses present in the original logic program. The idea can further be extended to deal with dynamic clauses. To that end, a new system does not incur such space overload.

This, nevertheless, involves storing the input logic program $\Delta_I = \Delta_L D^I \Delta_R$. On the other hand, the TFD system does not incur such space overload.

A distinctive feature of linear logic programs is that they grow and shrink over the course of execution. In particular, clauses can be dynamically added to the program by the linear implication. However, the user can only set spy-points on the clauses present in the original logic program. The idea can further be extended to deal with dynamic clauses. To that end, a new
debugging linear implication is defined as follows:

\[
\tilde{D}^t \Delta I \triangleright \tilde{D} \Delta O \xrightarrow{\sigma : \pi \sigma' v} G
\]

\[
\Delta I \Delta O \xrightarrow{\delta : \pi \sigma' v} D \sim \circ G \quad (t \in \delta)
\]

Each time the \( TFD \sim \circ R \) rule is applied, the linear clause \( D \) is marked to be debugged and added to the logic program. It is worth to note that no constraint is imposed on the form of \( D \), so no limitation is imposed on higher-order programming support.

The main contribution of the \( TFD \) system is that it is able to isolate the consumption of a given goal/clause without interfering with the traditional operational semantics of resource management systems. We are currently studying practical applications of this feature. Though these are by no means restricted to that of debugging, this seems to be the most promising one. In particular, the \( TFD \) system can be combined with a stepwise proof construction process to provide a basic proof-theoretic debugging framework.

5 Debugging a simple Lolli program

We shall illustrate the debugging capabilities of the \( TFD \) system through a trivial example. Let us consider a simple Lolli program with two predicates \( p \) and \( q \) standing for two ping-pong players. The turn is represented by two linear facts \( ping \) and \( pong \). Mutual exclusion between both turns must be ensured. Player \( p \) waits for his turn \( pong \) to come, then hits the ball changing the turn to \( ping \), and goes on playing. Player \( q \) exhibits a behavior dual to that of \( p \). A predicate \( go \) starts the match, assigning the turn to one of the players and facing them. An inexperienced Lolli programmer may write the following code:

\[
\begin{align*}
p & : - pong, (ping - o p). \\
q & : - ping, (pong - o q). \\
go & : - pong - o (p,q).
\end{align*}
\]

and discover it to fail with a concise \( \text{No} \). This does not provide much clue as to what’s wrong with the program. In this simple case, a better clue would be to know how many times the ball was hit and by whom. In resources consumption terms, how many linear facts \( ping \) and \( pong \) where consumed and who consumed them. To discover the answer, we may set a spy-point on the first and only clause of predicate \( p \), which shall be referred to as \( C_p \). (Similarly, the only clause of predicate \( q \) shall be referred to as \( C_q \).) The system may then provide the user with a trace of the consumption behavior of clause \( C_p \). The (unprovable) partial derivation built should be similar to the one depicted below:
When clause $C_p$ is applied, the $T \mathcal{FD} \text{pick \& mark}$ rule introduces a new consumption marker $d$. Therefore, resources consumed when solving the goal $C_p \gg p$ shall be marked with this new tag. The sequent

\[
\begin{align*}
\vdots & : \overline{C}_p, C_q, pong^d, ping \vdash \{t_2\} : \text{nil} \quad \{d\} \quad p \\
\vdots & : \overline{C}_p, C_q, pong^d \vdash \{t_2\} : \text{nil} \quad \{d\} \quad \text{ping} \rightarrow\text{o} \quad p
\end{align*}
\]

is unprovable, since $C_p$ requires a $pong$ to be proved and there are neither $pong$ nor player $q$ to provide one. The simple inspection of the output context reveals that there was only one $pong$ consumed by $C_p$, so the match lasted one hit only.

## 6 Conclusions Further Work

In this work, we have extended the traditional operational semantics of resource management system to provide a trace of resource consumption, without incurring extra runtime costs. Though the applications of the ideas presented are not limited to those related to debugging, our discussion has revolved around this important (and unfortunately forgotten) topic. In the long term, our aim is twofold. On the one hand, we want to explore novel applications of the isolation of resource consumption in linear logic proof search. On the other hand, this work may be the first step in providing a rigorous, proof-theoretic approach to declarative debugging.

## References


