Implementing the Linear Logic Programming Language Lygon

Michael Winikoff[1]
winikoff@cs.mu.oz.au
http://www.cs.mu.oz.au/~winikoff

James Harland[2]
jah@cs.rmit.edu.au
http://www.cs.rmit.edu.au/~jah

Abstract

There has been considerable work aimed at enhancing the expressiveness of logic programming languages. To this end logics other than classical first order logic have been considered, including intuitionistic, relevant, temporal, modal and linear logic. Girard's linear logic has formed the basis of a number of logic programming languages. These languages are successful in enhancing the expressiveness of (pure) Prolog and have been shown to provide natural solutions to problems involving concurrency, natural language processing, database processing and various resource oriented problems. One of the richer linear logic programming languages is Lygon. In this paper we investigate the implementation of Lygon. Two significant problems that arise are the division of resources between sub-branches of the proof and the selection of the formula to be decomposed. We present solutions to both of these problems.

Keywords: Linear Logic, Logic Programming, Lygon, Implementation.

1 Introduction

Logic programming may be viewed as the interpretation of logical formulae as a programming language. This is done by giving an operational interpretation to the process of proof search. Traditional logic programming languages such as Prolog are based on the Horn clause subset of classical logic. Pure Prolog however, is lacking in expressiveness in a number of areas. A desire for richer and more expressive logic programming languages has led researchers in two directions.

The first direction is the use of richer subsets than Horn clauses. The second direction involves the use of logics other than classical. Some logics that have been considered as basis for logic programming languages include modal and temporal [14], relevant [4] and linear logics.

This paper is concerned with the language *Lygon* [7, 15, 22]. Lygon is a logic programming language which is based on *linear logic*. By design the class of formulae used is as large as possible so as to make the language as expressive as possible. Due to its basis in linear logic Lygon's applications include resource management, state handling and concurrency. Lygon includes as a subset a number of languages including Horn clause classical logic languages (such as pure Prolog), hereditary Harrop formulae languages (such as first order λ-Prolog) and some other proposals for languages based on linear logic (ACL [11], Lolli [10], LinLog [2], $\mathcal{LC}$ [18] and Forum [13]).

[1]Department of Computer Science, The University of Melbourne, Parkville, Melbourne, 3052, Australia.

[2]Department of Computer Sciences, Royal Melbourne Institute of Technology, GPO Box 2476V, Melbourne, 3001, Australia.

Implementing Lygon is nontrivial. In addition to the usual issues for logic programming languages there are a number of new ones. Two particular ones stand out.

1. In searching for a proof of a goal involving the connective $\otimes$ we need to split the context between the two subproofs. This is done in the reduction of the $\otimes$ rule. Since logic programming languages search for proofs in a bottom up fashion this splitting is nondeterministic and inefficient if done naïvely.

2. Consider searching for a proof of a multiple conclusioned goal. Each time an inference rule is to be applied we must first select the formula that is to be reduced. This formula is the *active formula*. If done naïvely this selection process implies that any proof involving multiple conclusions has significantly more non-determinism than necessary.

This paper discusses these two issues and presents solutions to them. The solutions presented have been incorporated in the current implementation of Lygon.

The paper is organised as follows. In Section 2 we briefly discuss relevant background. Section 3 discusses at some length the deterministic allocation of resources among sub-branches and presents our solution. In Section 4 we discuss the selection of the active formula and show how the results of Andreoli [2] and Galmiche and Perrier [5] can be used to derive a solution. Section 5 presents results, both theoretical and practical. We compare our work to other relevant work - primarily Lolli related - in Section 6 and conclude and discuss further work in Section 7.

2 Background

Due to space limitations we are unable to include an introduction to linear logic. The reader requiring an introduction will be well served by one of [1, 6, 12, 16].

2.1 The Sequent Calculus

The sequent calculus is a formalisation of the mathematical notion of proof. The inference rule

$$\frac{\Gamma_0 \quad \ldots \quad \Gamma_n}{\Gamma}\, A$$

states that from $\Gamma_0 \ldots \Gamma_n$ we can *infer* Γ. The name of the inference rule is A.

A *proof* is a tree of inference rules where the sequent to be proved (the Γ) is at the root of the tree and the leaves of the tree are instances of an axiom of some sort. These trees are depicted with the root below the leaves[3].

The structure of a sequent varies between logics and between different presentations of logics. Generally it is a *sequence* (hence the name) of formulae written: $\vdash F_1, F_2 \ldots F_3$.

The formalisation of a logic typically contains a rule for each connective. In addition there are usually *structural* rules which can be applied to any formula. One inference rule that is part of nearly every logic is the *interchange* rule which swaps the order of two formulae in the sequent. Often this rule is elided and we think of the sequents as *multisets*.

Inference rules formalising linear logic are as follows, where Γ is a multiset of formulae which are not of the form $?F$. a and b are arbitrary formulae. p and q are atoms and $?\Delta$ represents a multiset of formulae of the form $?F$.

[3] Unlike standard computer science practice

$$\frac{}{\vdash ?\Delta, p, p^\perp} \; Ax \qquad \frac{}{\vdash ?\Delta, \mathbf{1}} \; \mathbf{1}$$

$$\frac{}{\vdash ?\Delta, \top, \Gamma} \; \top \qquad \frac{\vdash ?\Delta, \Gamma}{\vdash ?\Delta, \bot, \Gamma} \; \bot$$

$$\frac{\vdash ?\Delta, a, \Gamma_1 \quad \vdash ?\Delta, b, \Gamma_2}{\vdash ?\Delta, a \otimes b, \Gamma_1, \Gamma_2} \; \otimes \qquad \frac{\vdash ?\Delta, a, \Gamma \quad \vdash ?\Delta, b, \Gamma}{\vdash ?\Delta, a \,\&\, b, \Gamma} \; \&$$

$$\frac{\vdash ?\Delta, a_i, \Gamma}{\vdash ?\Delta, a_1 \oplus a_2, \Gamma} \; \oplus \; i \in \{1, 2\} \qquad \frac{\vdash ?\Delta, a, b, \Gamma}{\vdash ?\Delta, a \parr b, \Gamma} \; \parr$$

$$\frac{\vdash ?\Delta, a}{\vdash ?\Delta, !a} \; ! \qquad \frac{\vdash a, ?a, ?\Delta, \Gamma}{\vdash ?a, ?\Delta, \Gamma} \; ?$$

$$\frac{\vdash ?\Delta, a[t/x], \Gamma}{\vdash ?\Delta, \exists x a, \Gamma} \; \exists \qquad \frac{\vdash ?\Delta, a[y/x], \Gamma}{\vdash ?\Delta, \forall x a, \Gamma} \; \forall$$

Where y is not free in $\Gamma, ?\Delta$.

$$\frac{\vdash ?\Delta, a\theta, \Gamma}{\vdash ?\Delta, p, \Gamma} \; Res$$

Where the program contains the clause $q \leftarrow a$ and $q\theta = p$ for some substitution θ.

Our presentation is nonstandard in that we encode contraction (copying) and weakening (deletion) into other rules. Note that a formula of the form $?F$ is one that can be freely copied and deleted. The deletion is done by the $\mathbf{1}$ and Ax rules, the copying is done by the $\otimes$ and $?D$ rules.

Logic programming is seen as a form of proof *search*. The search is done bottom-up - the process begins with a goal sequent and extends the tree upwards until all branches have been closed with axioms of some sort.

2.2 Lygon

This paper considers a subset of the full Lygon language as derived in [15]. Program clauses ($\mathcal{D}$) must be of the form

$$\forall x_1 \ldots x_n (A \leftarrow \mathcal{G})$$

Goals ($\mathcal{G}$) are summarised by the following syntax (where A is an atom):

$$\mathcal{G} \;::=\; \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid !\mathcal{G} \mid ?\mathcal{G} \mid \exists x \mathcal{G} \mid \forall x \mathcal{G} \mid \mathbf{1} \mid \bot \mid \top \mid A \mid A^\perp$$

The semantics of the language is dictated by the sequent calculus inference rules with the exception that the $\exists$ rule is implemented using unification and logical variables.

As an example, the toggle program in Lygon looks like[4]

$$\text{toggle} \leftarrow (\text{on} \otimes \text{off}^\perp) \oplus (\text{off} \otimes \text{on}^\perp)$$

[4] Actually "real" Lygon is rendered in ASCII and would be `toggle <- (on * neg off) @ (off * neg on).`

and the proof of the goal $\vdash$ toggle, on, off$^\perp$ is

$$\dfrac{\dfrac{\dfrac{\overline{\vdash \text{off}, \text{off}^\perp}\ Ax \quad \overline{\vdash \text{on}, \text{on}^\perp}\ Ax}{\vdash \text{off} \otimes \text{on}^\perp, \text{on}, \text{off}^\perp}\ \otimes}{\vdash (\text{on} \otimes \text{off}^\perp) \oplus (\text{off} \otimes \text{on}^\perp), \text{on}, \text{off}^\perp}\ \oplus}{\vdash \text{toggle}, \text{on}, \text{off}^\perp}\ Res$$

The Lygon home page [19] contains an introductory tutorial to Lygon, a collection of programs, links to papers on Lygon and the implementation. For more information on Lygon see [7, 22].

3 Determinising the $\otimes$ rule

The standard formulation of the inference rules for linear logic have a significant amount of nondeterminism. Some of this nondeterminism is unavoidable and does not create efficiency problems; for example the $\oplus$ rule. Consider however the $\otimes$ rule:

$$\frac{\vdash ?\Delta, a, \Gamma_1 \quad \vdash ?\Delta, b, \Gamma_2}{\vdash ?\Delta, a \otimes b, \Gamma_1, \Gamma_2}\ \otimes$$

When this rule is applied bottom up we need to divide the linear formulae in the sequent between the two sub-branches. This division can be done in a number of ways which is exponential in the number of formulae. Hence a naïve implementation which backtracks through all possibilities is not practical.

It is possible to modify the $\otimes$ rule so the division of formulae between sub-branches is done lazily and efficiently. The key idea is that all formulae are passed to the first sub-branch. Unused formulae are returned and then passed to the second sub-branch. We refer to this mechanism as *lazy splitting* (Hodas and Miller [9] refer to this mechanism as the input output model of resource consumption). Consider as an example the proof of $b \parr (\mathbf{1} \otimes b^\perp)$. We begin by using the $\parr$ rule to break off the b yielding $b, \mathbf{1} \otimes b^\perp$. We then process the left side of the $\otimes$ rule – we pass it the rest of the context (i.e. the b). We are now trying to prove $b, \mathbf{1}$. This succeeds with the b as unused residue. The b is then passed to the right branch of the $\otimes$ rule: $b, b^\perp$ which is an instance of the axiom rule.

The details of this solution however are not without a certain amount of subtlety – if care is not taken soundness can be compromised.

We begin our exposition by considering a fragment of Lygon excluding $\top$. Lazy splitting for this fragment is relatively simple; the real subtlety arises when $\top$ is reintroduced.

Consider the formula $(b \parr \mathbf{1}) \otimes b^\perp$. Clearly it is not provable. Consider now a naïve formulation of lazy splitting. Instead of a sequent of the form $\vdash \Gamma$ we use the notation $\Gamma \Rightarrow \Delta$ with the reading that the Δ is the excess formulae (also known as the *residue*) being returned unused. A successful proof cannot have excess resources and hence we require that its root be of the form $\Gamma \Rightarrow$

The standard sequent rules are modified as follows. The axiom rules are modified to return unused formulae:

$$\frac{}{p, p^\perp, \Gamma \Rightarrow \Gamma}\ Ax \qquad \frac{}{\mathbf{1}, \Gamma \Rightarrow \Gamma}\ \mathbf{1}$$

Note that lazy splitting does not affect formulae of the form $?F$ since these are not split by our version of the $\otimes$ rule. We elide these formulae from the rules below to clarify the presentation. The unary logical rules are modified to pass on returned formulae

$$\frac{a, b, \Gamma \Rightarrow \Delta}{a \parr b, \Gamma \Rightarrow \Delta}\ \parr$$

Finally, the $\otimes$ rule passes the excess from the left sub-branch into the right sub-branch.

$$\frac{a, \Gamma \Rightarrow \Delta \quad b, \Delta \Rightarrow \Lambda}{a \otimes b, \Gamma \Rightarrow \Lambda}\ \otimes$$

Using these rules we find however that $(b \parr \mathbf{1}) \otimes b^{\perp}$ is derivable!

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1}, b \Rightarrow b}\ \mathbf{1}}{\mathbf{1} \parr b \Rightarrow b}\ \parr \quad \dfrac{}{b, b^{\perp} \Rightarrow}\ Ax}{(\mathbf{1} \parr b) \otimes b^{\perp} \Rightarrow}\ \otimes$$

This problem arises since the axiom rules return any and all formulae. For lazy splitting to be valid we must respect a notion of scope; a formula may only be returned if it was present in the conclusion of the $\otimes$ rule. In the unsound derivation above b should not be returned by the $\mathbf{1}$ rule since it was introduced above the $\otimes$ rule. Another characterisation is that the $\otimes$ rule can't split what it doesn't have.

To prevent this problem we must keep track of which formulae are returnable. We do this by tagging a *returnable* formula with a superscript $\top$. Untagged formulae must be consumed in the current branch of the proof. Tagged formulae may be returned from the current branch. The marking may be thought of as a "proof of purchase" without which we cannot return the resource.

Note that we have to allow nestable tags to handle nested $\otimes$. The $\otimes$ rule adds and removes a single tag.

The revised axiom rules can only pass on unused formulae if they are tagged.

$$\frac{}{p, p^{\perp}, \Gamma^{\top} \Rightarrow \Gamma^{\top}}\ Ax \qquad \frac{}{\mathbf{1}, \Gamma^{\top} \Rightarrow \Gamma^{\top}}\ \mathbf{1}$$

The revised $\otimes$ rule

$$\frac{a, \Gamma^{\top} \Rightarrow \Lambda^{\top} \quad b, \Lambda \Rightarrow \Delta}{a \otimes b, \Gamma \Rightarrow \Delta}\ \otimes$$

marks all existing formulae as returnable and then passes them to the first sub-branch. Note that the formulae returned from the first sub-branch must have their tags removed before being passed to the second sub-branch. This ensures that formulae which are untagged in the conclusion of the $\otimes$ rule cannot be returned unused from the right premise and hence from the conclusion. As an example consider the formula $(b \parr (\mathbf{1} \otimes \mathbf{1})) \otimes b^{\perp}$ which is clearly unprovable. If we neglect to have our lazy $\otimes$ rule strip away the tags before passing formulae to the right premise we lose soundness since the above formula has a derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b^{\top}, \mathbf{1} \Rightarrow b^{\top}}\ \mathbf{1} \quad \dfrac{}{b^{\top}, \mathbf{1} \Rightarrow b^{\top}}\ \mathbf{1}}{b, \mathbf{1} \otimes \mathbf{1} \Rightarrow b^{\top}}\ \otimes}{b \parr (\mathbf{1} \otimes \mathbf{1}) \Rightarrow b^{\top}}\ \parr \quad \begin{array}{c}\vdots \\ b^{\top}, b^{\perp} \Rightarrow\end{array}}{(b \parr (\mathbf{1} \otimes \mathbf{1})) \otimes b^{\perp} \Rightarrow}\ \otimes$$

We now introduce an additional rule. The Use rule claims a formulae for use in the current sub-branch by stripping off the tags that allow it to be returned unused.

We define $a^{'}$ to remove all tags on a.

$$\frac{a', \Gamma \Rightarrow \Delta}{a, \Gamma \Rightarrow \Delta}\ Use$$

We have seen the rules for $\otimes$, $\wp$ and Ax. The rules for $\oplus$, ? , $\bot$, $\forall$, $\exists$ are similar to $\wp$. The remaining rules are ! and $\&$.

When we apply ! we must ensure that there are no other (linear) formulae. Thus we force all excess formulae to be returned. In a way ! is similar to the Ax and **1** rules.

$$\frac{a \Rightarrow}{!a, \Gamma^{\top} \Rightarrow \Gamma^{\top}}\ !$$

The other rule which is affected by the lazy splitting mechanism is the $\&$ rule.

$$\frac{a, \Gamma \Rightarrow \Delta_1 \quad b, \Gamma \Rightarrow \Delta_2}{a \,\&\, b, \Gamma \Rightarrow \Delta}\ \&$$

This rule has the constraint that $\Delta_1 = \Delta_2 = \Delta$. This enforces the requirement in the non-lazy $\&$ rule that the two sub-branches have the same context. Note that we prefer to have an explicit constraint since this constraint will need to be subsequently modified when $\top$ is reintroduced.

Looking at a complete proof and seeing sequents of the form $a^{\bot\top}, b^{\top}, a \Rightarrow b^{\top}$ one is often left with the feeling that there is still some magic at work. This is not so; when a proof is being constructed bottom up, the right hand side of the arrow ($\Rightarrow$) is left unbound on the way up and determined at the leaves.

The rules presented so far form a sound and complete collection of inference rules for the fragment of the logic excluding $\top$. These rules manage resources deterministically.

The lazy splitting version of $\top$ involves a significant amount of subtlety and has implications for the $\&$ rule which becomes rather complex.

Consider the non-lazy inference

$$\dfrac{\dfrac{}{\vdash \top, \Gamma_1}\ \top \qquad \begin{matrix}\vdots\\ \vdash G, \Gamma_2\end{matrix}}{\begin{matrix}\vdash \top \otimes G, \Gamma_1, \Gamma_2\\ \vdots\end{matrix}}\ \otimes$$

The rule for $\top$ simply consumes all formulae. Consider now the lazy splitting version of the above inference

$$\dfrac{\dfrac{}{\top, {\Gamma_1}^{\top}, {\Gamma_2}^{\top} \Rightarrow {\Gamma_2}^{\top}}\ \top \qquad \begin{matrix}\vdots\\ G, \Gamma_2 \Rightarrow\end{matrix}}{\begin{matrix}\top \otimes G, \Gamma_1, \Gamma_2 \Rightarrow\\ \vdots\end{matrix}}\ \otimes$$

The application of the $\top$ rule must somehow know which formulae are not to be consumed since they will be required later in the proof.

The lazy splitting version of the $\top$ rule has to divide formulae between $\top$ and the rest of the proof. This can be done in a number of ways which is exponential in the number of formulae.

Although this sounds similar to the problem in the original $\otimes$ rule there are two differences. Firstly, it is not possible to nest $\top$s so a single non-nestable tag will suffice. Secondly and more significantly, the direction is opposite – the first sub-branch of the $\otimes$ rule returns formulae which must be consumed by the second sub-branch. The $\top$ rule on the other hand will consume all formulae by tagging them appropriately then passing them on. The formulae passed on *have been consumed* by $\top$ but – in case they should not have been – can be "unconsumed".

We shall use a prefix ¿ ("maybe") to tag formulae which have been consumed by $\top$ and which can be "unconsumed" if necessary. Note that only returnable formulae are thus treated. Formulae which are not returnable are summarily consumed by the $\top$ rule.

$$\dfrac{}{\top, \Delta, \Gamma^{\top} \Rightarrow (\text{¿}\Gamma)^{\top}}\ \top$$

One might be tempted to define ¿ in terms of existing connectives, *viz.* $\text{¿}F \equiv F \oplus \bot$. There is however a subtle but important difference between the two. $\text{¿}F$ represents a formula which may have been consumed by $\top$, thus F is either consumed — in which case it must not be used anywhere — or unconsumed — in which case it must be consistently accounted for.

$F \oplus \bot$ on the other hand allows for some parts of the proof to choose F and other parts to choose $\bot$. Consider for example $((\top \otimes (b \,\&\, \mathbf{1})) \parr b^{\perp})$ which is not provable in the standard system; however if we use $b \oplus \bot$ in place of $\text{¿}b$ it has a derivation:

$$\dfrac{\dfrac{\dfrac{}{\top, {b^{\perp}}^{\top} \Rightarrow (b^{\perp} \oplus \bot)^{\top}}\ \top \qquad \dfrac{\dfrac{\dfrac{}{b, b^{\perp} \Rightarrow}\ Ax}{b, b^{\perp} \oplus \bot \Rightarrow}\ \oplus \qquad \dfrac{\dfrac{\dfrac{}{\mathbf{1} \Rightarrow}\ \mathbf{1}}{\mathbf{1}, \bot \Rightarrow}\ \bot}{\mathbf{1}, b^{\perp} \oplus \bot \Rightarrow}\ \oplus}{b \,\&\, \mathbf{1}, b^{\perp} \oplus \bot \Rightarrow}\ \&}{\top \otimes (b \,\&\, \mathbf{1}), b^{\perp} \Rightarrow}\ \otimes}{(\top \otimes (b \,\&\, \mathbf{1})) \parr b^{\perp} \Rightarrow}\ \parr$$

In addition to pointing out the difference between $\text{¿}F$ and $F \oplus \bot$ this suggests that the $\top$ rule cannot be simply added to the system derived so far since maintaining the consistent usage of formulae of the form $\text{¿}F$ requires a measure of global information. Thus, rather than having the axiom rules delete consumed formulae which have turned out to be unneeded (Ax')

$$\dfrac{}{p, p^{\perp}, \text{¿}\Delta, \Gamma^{\top} \Rightarrow \Gamma^{\top}}\ Ax'$$

we must return the consumed formulae which were unneeded, so that we can check that different branches agree on which consumed formulae have had to be unconsumed (Ax).

$$\dfrac{}{p, p^{\perp}, \text{¿}\Delta, \Gamma^{\top} \Rightarrow \Gamma^{\top}, \text{¿}\Delta}\ Ax$$

If we don't return the unused formulae then we lose soundness, and we can derive unprovable formulae such as $a, b, \top \otimes (a^{\perp} \,\&\, b^{\perp})$

$$\dfrac{\dfrac{}{a^{\top}, b^{\top}, \top \Rightarrow (\text{¿}a)^{\top}, (\text{¿}b)^{\top}}\ \top \qquad \dfrac{\dfrac{\dfrac{}{a, \text{¿}b, a^{\perp} \Rightarrow}\ Ax'}{\text{¿}a, \text{¿}b, a^{\perp} \Rightarrow}\ Use \qquad \dfrac{\dfrac{}{\text{¿}a, b, b^{\perp} \Rightarrow}\ Ax'}{\text{¿}a, \text{¿}b, b^{\perp} \Rightarrow}\ Use}{\text{¿}a, \text{¿}b, a^{\perp} \,\&\, b^{\perp} \Rightarrow}\ \&}{a, b, \top \otimes (a^{\perp} \,\&\, b^{\perp}) \Rightarrow}\ \otimes$$

We thus use Ax rather than Ax'. In order to do this we must modify the unary rules to pass on the returned formulae of the form $¿F$. We also need to have a look at the two binary rules — $\otimes$ and $\&$.

The $\otimes$ rule does the usual passing from the left sub-branch to the right. The only interesting point is that it is now possible for the left sub-branch to return formulae which do not have a $-^{\top}$ tag. These formulae (which are of the form $¿F$) must be returned by the conclusion without being passed to the right sub-branch. If this is not done soundness is compromised. Consider as an example the unprovable formulae $((\top \otimes \mathbf{1}) \parr a) \otimes a^{\perp}$. If we use a lazy $\otimes$ rule which passes formulae of the form $¿F$ into the right premise then there is a derivation of this formula:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\top, a^{\top} \Rightarrow (¿a)^{\top}}\,\top \qquad \dfrac{}{\mathbf{1}, ¿a \Rightarrow ¿a}\,\mathbf{1}}{\top \otimes \mathbf{1}, a \Rightarrow ¿a}\,\otimes}{(\top \otimes \mathbf{1}) \parr a \Rightarrow ¿a}\,\parr \qquad \dfrac{\dfrac{}{a, a^{\perp} \Rightarrow}\,Ax}{¿a, a^{\perp} \Rightarrow}\,Use}{((\top \otimes \mathbf{1}) \parr a) \otimes a^{\perp} \Rightarrow}\,\otimes$$

We now turn to the $\&$ rule. Consider formulae of the form $¿F$. Unlike formulae of the form $F^{\top}$ these cannot be returned from the current branch for use elsewhere. Formulae of the form $¿F$ can only be propagated downwards. The only rule which makes use of them (other than $\top$ is the $\&$ rule. The $\&$ rule uses these formula to enforce consistent consumption and undeletion between its two premises.

We now investigate the details of this mechanism. We begin by noting that the residue in the two premises of a $\&$ rule need not necessarily match. In order to be able to detect when a mismatch is invalid we define the notion of a $\top$-*like* (sub-)proof. The rules of the system are modified to track whether (sub-)proofs are $\top$-like. This information is used by the $\&$ rule when enforcing consistent consumption of formulae between its premises.

Despite the tagged input being the same in both premises it is possible for the residue to be different:

$$\dfrac{\dfrac{\dfrac{}{a^{\perp}, a \Rightarrow}\,Ax}{a^{\perp}, a^{\top} \Rightarrow}\,Use \qquad \dfrac{}{\top, a^{\top} \Rightarrow (¿a)^{\top}}\,\top}{a^{\perp} \,\&\, \top, a^{\top} \Rightarrow ??}\,\&$$

This sub-proof occurs in the proof of $((a^{\perp} \,\&\, \top) \otimes \mathbf{1}) \parr a$ (which for a change is actually provable) and hence had better succeed.

The intuition is that the $¿a$ represents a formula which has been consumed but which perhaps should not have been. That the left premise of the $\&$ rule does not contain $¿a$ indicates that the a must have been consumed in the left premise. That the right premise of the $\&$ does contain $¿a$ indicates that the right sub-proof is prepared to undelete the a which it has consumed. In order to be consistent with the left sub-proof this must not be allowed to happen. Thus the proof is valid if we prevent the a from being unconsumed elsewhere in the proof.

We define a (sub-)proof to be $\top$-*like* if adding formulae to its sequents yields a valid proof. Although we would normally require that the residue of the premises of a $\&$ rule must agree, if a premise is $\top$-like we can allow it to have extra residue. This extra residue represents consumed formulae. By not passing them on to the conclusion of the $\&$ rule we prevent them from being unconsumed, thus preserving soundness. Note that a $\&$ inference may have to be failed if its premises have incompatible residues and are not $\top$-like.

In order to be able to apply this check we need to track whether sub-proofs are $\top$-like. We add a tag to each sequent to indicate whether it is $\top$-like. $/true$ if the proof is $\top$-like and $/false$ if it isn't. The $\top$ rule is given $/true$, the Ax and $\mathbf{1}$ rules are $/false$. The unary rules pass on the tag. A $\otimes$ rule's conclusion is $\top$-like if at least one of its premises is. So, if the tags on the two premises are $/x$ and $/y$ the tag on the conclusion is $/x \vee y$. The conclusion of a $\&$ inference is $\top$-like if *both* premises are. So, if the tags on the two premises are $/x$ and $/y$ the tag on the conclusion is $/x \wedge y$.

The previous proof is valid since the right premise would be labeled with $/true$.

The returned formulae of the $\&$ rule's conclusion are calculated by taking the *intersection* of the returned formulae of the two premises. The proof fails if either premise has excess residue (formulae which are not in the intersection) and is tagged with $/false$. In the previous proof "??" would be empty (the intersection of an empty residue and $¿a^\top$).

One minor wrinkle is that while the intersection must agree on the formulae it may not agree on their tags. Specifically, the formulae in one premise may have a ¿ tag in addition to some $-^\top$ tags whereas those in the other premise may not have the ¿ tag; what value should we give to X in the proof:

$$\dfrac{\dfrac{}{a^\top, \top \Rightarrow (¿a)^\top}\,\top \qquad \dfrac{}{a^\top, \mathbf{1} \Rightarrow a^\top}\,\mathbf{1}}{a^\top, \top \,\&\, \mathbf{1} \Rightarrow X}\,\&$$

The solution is to pass on the ¿ tag if both premises have it and to strip it off if only one of the premises has it.

The rules derived form a solution to the problem of splitting the context between the two premises of a $\otimes$ inference. The rules work in a multiple-conclusioned setting which is a general case of the single-conclusion one. The rules are summarised in appendix A.

An expanded form of this section can be found in [20].

4 Selecting the Active Formula

Lygon is based on multiple conclusion linear logic. Each inference step selects a formula and applies the appropriate inference rule bottom up.

When designing a language based on a multiple conclusion logic one can at design time impose the constraint that selecting the active formula be done using "don't care" nondeterminism. Doing this yields a more limited class of formulae but simplifies the implementation. This path was taken in the design of $\mathcal{LC}$ [18] and LO [3]. In the design of Lygon the opposite choice was made. As a result Lygon has a significantly larger class of formulae but has to contend with the problem that selecting the active formulae may have to be done using "don't know" nondeterminism.

Although in general selecting the active formulae in Lygon may require backtracking, in certain situations we can safely commit to the selection. An investigation of when Lygon can safely use "don't care" nondeterminism to select the active formula involves an investigation of the permutability properties of linear logic.

Investigations of these properties have been done by Andreoli [2] and Galmiche and Perrier [5]. Note that there is a fair amount of overlap in the results of the two papers although their motivation is different.

A (partial) solution to the problem in Lygon of selecting the active formula can be obtained by simply applying the observations of these papers. This section briefly summarises the results of the two papers. For more details we refer the reader to the papers themselves.

In [2] Andreoli identifies two classes of connectives – *synchronous* and *asynchronous*. The latter can be permuted down in a proof and hence can be selected using “don’t care” nondeterminism. More formally, if there is a proof of a sequent containing a formula whose topmost connective is asynchronous then there is a proof of the sequent where that formula is the active formula.

As an example, consider the proof of $(a(1)^\perp \parr \perp), \exists x a(x)$. Since $\parr$ is asynchronous there exists a proof where the first inference applied is $\parr$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a(1)^\perp, a(1)}\,Ax}{\vdash a(1)^\perp, \perp, a(1)}\,\perp}{\vdash a(1)^\perp \parr \perp, a(1)}\,\parr}{\vdash a(1)^\perp \parr \perp, \exists x a(x)}\,\exists \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a(1)^\perp, a(1)}\,Ax}{\vdash a(1)^\perp, \exists x a(x)}\,\exists}{\vdash a(1)^\perp, \perp, \exists x a(x)}\,\perp}{\vdash a(1)^\perp \parr \perp, \exists x a(x)}\,\parr$$

The synchronous connectives are $\mathbf{1}$, $\mathbf{0}$, $\otimes$, $\oplus$, ! and $\exists$. The asynchronous connectives are $\top$, $\perp$, $\parr$, $\&$, ? and $\forall$.

A further observation made in [2] is that if we have selected a synchronous formula and its subformulae have a synchronous connective topmost then we can automatically select the subformulae. This process of continuing to select synchronous subformulae is known as *focusing*.

In [5] Galmiche and Perrier systematically analyse the permutability properties of linear logic. In addition to the properties noted they also observe that if the sequent is of the form $!F, ?\Delta$ then we can commit to the $!F$ as the active formula.

These observations (which are incorporated in the current version of Lygon) yield a significant reduction in the nondeterminism associated with selecting the active formula.

5 Summary of Results

5.1 Formal Results

For the next two theorems $\aleph$ is a multiset of formulae of the form ¿F.

Theorem 1 *(Soundness) If $\Gamma \Rightarrow \aleph$ is provable in the lazy-splitting sequent calculus then there exists a proof of $\vdash \Gamma$ in the standard sequent calculus.*
Proof: *Omitted due to lack of space (see [20, Theorem 4.14])*

Theorem 2 *(Completeness) If $\vdash \Gamma$ is provable in the standard sequent calculus then for some $\aleph$ there exists a proof of $\Gamma \Rightarrow \aleph$ in the lazy-splitting sequent calculus.*
Proof: *Omitted due to lack of space (see [20, Theorem 5.12])*

Theorem 3 *Given a proof in the standard sequent calculus there is a proof in the lazy-splitting sequent calculus which has the same structure upto the insertion of Use rules immediately before a formula is reduced.*
Proof: *An algorithm which maps proofs in the standard sequent calculus to proofs in the lazy-splitting sequent calculus is presented in [20, Algorithm 2]. It is easy to verify that the lazy-splitting proof produced conserves the structure of the proof.*

	A	B	C	D
Start	0.166	0.175	0.168	0.166
1	1.518	1.166	0.551	0.496
2	4.218	2.228	0.410	0.301
3	39.45	4.536	2.268	0.440
4	3.796	2.960	0.781	0.676
5	5.830	5.368	0.576	0.658

Figure 1: Benchmark Results

Theorem 4 *If there is a proof in the lazy-splitting sequent calculus then a proof search incorporating the observations of [2, 5] will find it.*
Proof: *The theorems in [2, 5] state that if there is a proof then there is a proof which is "normal" – that is it satisfies the optimisations summarised in section 4. According the theorem 3 there exists a corresponding lazy-splitting proof where the Use rule is only applied to a formula immediately before its reduction.*

5.2 The Implementation

The Lygon implementation (version 0.4) is written in BinProlog[5]. It consists of 505 lines of code[6] for the non-debugging version and 913 lines of code for the version including the debugger.

In order to determine whether the optimisations presented do represent improvements despite a certain amount of administrative overhead we have run a few short benchmarks.

The standard Lygon interpreter was modified to remove the optimisations resulting in four versions:

	LygonA	LygonB	LygonC	LygonD
Lazy Splitting	no	no	yes	yes
Active Formula Selection	no	yes	no	yes

Briefly, the first benchmark finds all paths between two given nodes in a cyclic graph and the second is an artificial test of splitting. The third benchmark demonstrates the use of the $\top$ connective in simulating "affine" predicates which can be used at *most* once. The fourth benchmark finds all solutions for an instance of a modified binpacking problem [7] and the final benchmark simulates a Petri net.

Each of the benchmarks was run on all four versions of Lygon. The timings are given in figure 1. All times are the average of ten runs on an idle DEC Alpha. Times are given in seconds and were measured by BinProlog. The source code for the benchmarks is available at

`http://www.cs.mu.oz.au/~winikoff/papers/bench/index.html`

In figure 1 the row labeled "Start" is the (CPU) time taken to start the appropriate Lygon interpreter.

Looking at the results we can see that the lazy splitting mechanism indeed represents a significant improvement. The difference in running time between LygonB

[5] Version 2.20
[6] including comments and whitespace

and LygonD ranges between a factor of three and a factor of fifteen - more than an order of magnitude!

The optimisations discussed in section 4 in general give a modest increase in performance. However, there are exceptions - the affine benchmark shows a significant speedup and the Petri net benchmark shows a slowdown. This variance is due to two factors.

Firstly, the implementation represents the goal as a (Prolog) list of formulae. Additionally, the selection of the active formula is done entirely at run time. It is possible to do at least some of the selection process at compile time. For example once the formula $a \otimes ((b \mathbin{\&} (d \invamp e)) \oplus f)$ is selected as the active formula it can be completely processed without any further selection. This is due to an application of focusing and the immediate processing of asynchronous connectives. The slowdown is simply the result of the overhead.

Secondly, the benefit gained depends to a very large extent on the goal. Some goals - for example one which does not make use of $\invamp$ - gain no benefit from intelligent selection of the active formula whereas others can gain a significant benefit.

An example of the latter are formulae of the form

$$(\mathbf{1} \invamp \bot \invamp \ldots \invamp \bot) \otimes \mathrm{print(x)} \otimes \mathrm{fail}$$

where there are n occurrences of $\bot$. The intelligent selection of the active formulae will find exactly one solution and do so rapidly. The versions of Lygon using the naïve selection mechanism (LygonA and LygonC) will find a rather large number of solutions[7].

6 Comparison with Other Work

The notion of lazy splitting was first introduced in [9], and a lazy method of finding goal-directed proofs (known in [9] as the input-output model of resource consumption) is given. An extension of this system presented in Hodas' thesis [8] handles the $\top$ rule lazily.

Lolli is based on a single conclusion logic whereas Lygon is based on a multiple conclusion logic. Thus (full) Lygon is a generalisation of Lolli. It is actually possible to encode Lygon into Lolli by introducing a new constant and using resolution to select the active formula. The goal $a \invamp b$ is encoded as $((a \multimap n) \otimes (b \multimap n)) \multimap n$ where n is the new constant. This translation is reminiscent of the double negation translation of classical into intuitionistic logic.

We feel, however, that a direct approach is more desirable for a number of reasons.

Firstly our presentation allowed us to use the permutability properties explored in [5, 2] to reduce the nondeterminism associated with selecting the active formula. In the Lolli encoding selecting the next formula to be reduced is done by the resolution rule. Reducing the nondeterminism under the Lolli encoding would require modifying the resolution rule to examine the body of each clause and decide accordingly whether the clause could be committed to. While this could be done it would seem to involve significantly more work than is required for the non-encoded presentation. Note furthermore that our solution handles the Lolli language as a simple special case in which all sequents just happen to have a single goal formula.

Secondly, the application of logical equivalences contains some subtleties in the context of proof theoretic arguments. The relationship between goal directed (ie.

[7] $(2^{n-1})\ (n-1)!$

operational) equivalence and logical equivalence is nontrivial. The logical equivalence of two formulae only implies operational equivalence if the two formulae are within the appropriate subset. For example, the two formulae $(a \oplus b) \multimap (a \oplus b)$ and $(a \multimap (a \oplus b)) \,\&\, (b \multimap (a \oplus b))$ are logically equivalent however the first is not a valid goal formula and does not have a goal directed proof even though the second does. For more on this sort of issue and on the general problem of designing logic programming languages see [21].

LinLog incorporates most of the optimisations noted in section 4. These optimisations are also applicable to Forum.

7 Conclusions and Further Work

We have shown how to eliminate the nondeterminism associated with resource allocation in Lygon. We have also shown how to apply known permutability results in order to reduce the nondeterminism associated with selecting the active formula. Both these optimisations are incorporated in the Lygon interpreter[8]

Since both of these sources of nondeterminism are exponential avoiding them is essential for a non-toy implementation. Measurements confirm that these optimisations are significant.

Whilst the method presented for splitting resources between sub-branches is optimal (in that the $\otimes$ and $\top$ rules are deterministic) there is scope for further investigation regarding the selection of the active formula. Possibilities to be explored include a global analysis and having the user supply information.

The methods reported in this paper are applicable to linear logic programming languages other than Lygon. In particular they have an impact on the implementation of Forum and LinLog.

In addition the lazy splitting rules have application in theorem provers for linear logic [17], where they eliminate a significant potential inefficiency.

Other obvious areas of work include investigation into the compilation of Lygon and relevant analyses and user specified information such as types and modes.

Non-implementation related work includes investigations into natural applications for Lygon. These so far include graph algorithms, concurrent problems, AI search problems involving states and transitions, specifications and a variety of database related applications including the modelling of transactions and deductive and active databases. In addition there are a number of more theoretical issues that are raised, some of which relate to certain application. These include the nature of negation as failure in a linear logic context and bottom up evaluation of Lygon.

Acknowledgments

We would like to thank David Pym for interesting discussions. We would like to thank an anonymous referee for pointing us to Hodas' thesis and for suggesting the encoding of Lygon in Lolli. Michael Winikoff is supported by an Australian Postgraduate Award (APA) scholarship.

[8] Available from [19].

References

[1] V. Alexiev. Applications of linear logic to computation: An overview. *Bulletin of the IGPL*, 2(1):77–107, March 1994.

[2] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

[3] J.-M. Andreoli and R. Pareschi. LO and behold! concurrent structured processes. *SIGPLAN Notices*, 25(10):44–56, 1990.

[4] A. W. Bollen. Relevant logic programming. *Journal of Automated Reasoning*, 7:563–585, 1991.

[5] D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, December 1994.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] J. Harland and D. Pym. A note on the implementation and applications of linear logic programming languages. In G. Gupta, editor, *Seventeeth Annual Computer Science Conference*, pages 647–658, 1994.

[8] J. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation*. PhD thesis, University of Pennsylvania, 1994.

[9] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic (extended abstract). In *Logic in Computer Science*, pages 32–42. IEEE, 1991.

[10] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic logic. *Journal of Information and Computation*, 10(2):327–365, 1994.

[11] N. Kobayashi and A. Yonezawa. ACL – a concurrent linear logic programming paradigm. In *International Logic Programming Symposium*, pages 279–294, 1993.

[12] Y. Lafont. Introduction to linear logic. Lecture Notes for the Summer School in Constructive Logics and Category Theory, August 1988.

[13] D. Miller. A multiple-conclusion meta-logic. In *Logic in Computer Science*, pages 272–281, 1994.

[14] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In D. M. Gabbay and H. J. Ohlbach, editors, *First International Conference on Temporal Logic*, pages 445–479. Springer-Verlag, July 1994.

[15] D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[16] A. Scedrov. A brief guide to linear logic. *Bulletin of the European Association for Theoretical Computer Science*, 41:154–165, June 1990.

[17] T. Tammet. Proof search strategies in linear logic. Programming Methodology Group 70, University of Göteborg and Chalmers University of Technology, March 1993.

[18] P. Volpe. Concurrent logic programming as uniform linear proofs. In G. Levi and M. Rodríguez-Artalejo, editors, *Algebraic and Logic Programming*, pages 133–149. Springer-Verlag, September 1994.

[19] M. Winikoff. Lygon home page.
`http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html`.

[20] M. Winikoff and J. Harland. Deterministic resource management for the linear logic programming language Lygon. Technical Report 94/23, Melbourne University, 1994.

[21] M. Winikoff and J. Harland. Characterising logic programming languages. Technical Report 95/26, Melbourne University, 1995.

[22] M. Winikoff and J. Harland. Implementation and development issues for the linear logic programming language Lygon. In *Australasian Computer Science Conference*, pages 563–572, February 1995.

A The Lazy Splitting System

The symbols Σ , Π and Ξ represent multisets of formulae of the form $-^{\top}$. The symbols $\aleph$ (aleph) , $\beth$ (beth), $\gimel$ (gimel) and $\daleth$ (daleth) represent multisets of formulae of the form ¿−. The symbols δ , Δ , Γ , Λ and Ω represent multisets of formulae that are not in one of the two prior forms. a and b represent any formulae. p represents an atom. The notation $a^{\top_n}$ represents the formula where a is superscripted by n $\top$s. c represents a tagged formulae. The function f used in the $\top$ rule is given by

$$\begin{aligned} f\ A^{\top} &= (f\ A)^{\top} \\ f\ ¿A &= ¿A \\ f\ x &= ¿x \end{aligned}$$

The δ : is a nonlinear region where formulae of the form $?F$ are kept.

We define $-'$ to remove all tags. Thus $(¿a)^{\top'} = a$.

mintag removes the ¿ tag if exactly one of the formulae has a ¿ tag. Otherwise it leaves the formula unchanged. The & rule has the side conditions that $(x = true) \vee (\daleth = \Omega_1 = \emptyset)$ and $(y = true) \vee (\gimel = \Omega_2 = \emptyset)$ and $\Sigma' = \Xi'$.

$$\frac{}{\delta : p, p^{\perp}, \Pi, \beth \Rightarrow \Pi, \beth / false}\ \text{Ax} \qquad \frac{}{\delta : \mathbf{1}, \Pi, \beth \Rightarrow \Pi, \beth / false}\ \mathbf{1}$$

$$\frac{}{\delta : \top, \Gamma, \Pi, \beth \Rightarrow (f\ \Pi), \beth / true}\ \top \qquad \frac{\delta : \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : \perp, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ \perp$$

$$\frac{\delta : a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : a \oplus b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ \oplus_1 \qquad \frac{\delta : b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : a \oplus b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ \oplus_2$$

$$\frac{\delta : a \Rightarrow \aleph / x}{\delta : !a, \Pi, \beth \Rightarrow \Pi, \beth / false}\ ! \qquad \frac{\delta : a, b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : a \invamp b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ \invamp$$

$$\frac{\delta : a[t/x], \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : \exists x a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ \exists \qquad \frac{\delta : a[y/x], \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : \forall x a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ \forall$$

Where y is not free in Γ

$$\frac{\delta, a : \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : ?a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ ? \qquad \frac{a, \delta : a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{a, \delta : \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ ?D$$

$$\frac{\delta : c', \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}{\delta : c, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph / x}\ Use$$

$$\frac{\delta : a, \Gamma^{\top}, \Pi^{\top}, \beth^{\top} \Rightarrow \Delta^{\top}, \Xi^{\top}, \gimel, \daleth^{\top} / x \quad \delta : b, \Delta, \Xi, \daleth \Rightarrow \Sigma, \aleph / y}{\delta : a \otimes b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \gimel, \aleph / x \vee y}\ \otimes$$

$$\frac{\delta : a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \Omega_1, \aleph, \daleth / x \quad \delta : b, \Gamma, \Pi, \beth \Rightarrow \Xi, \Omega_2, \aleph, \gimel / y}{\delta : a \,\&\, b, \Gamma, \Pi, \beth \Rightarrow mintag(\Sigma, \Xi), \aleph / x \wedge y}\ \&$$