Implementation and Development Issues for the Linear Logic Programming Language Lygon

Michael Winikoff
Department of Computer Science
University of Melbourne
Parkville, 3052

winikoff@cs.mu.oz.au

James Harland
Department of Computer Science
Royal Melbourne Institute of Technology
GPO Box 2476V
Melbourne, 3001
jah@cs.rmit.oz.au

Abstract

We describe and discuss the implementation of a new logic programming language called *Lygon*. This language is based on linear logic, a logic designed with bounded resources in mind. Linear logic may be thought of as a generalisation of classical logic, and as a result *Lygon* contains various features which do not exist in (pure) Prolog, whilst maintaining all the features of (pure) Prolog. In this paper we describe the implementation of this language, which posed a variety of programming challenges. The operational model for the language is based on the notion of *goal-directed* provability, a notion which has been much studied in the literature. However, there is a significant amount of non-determinism in this notion of proof. Hence the task of developing a systematic and deterministic manner in which to search for proofs requires some intricate and novel implementation techniques. We describe and discuss our particular solution, as well as the features of the language and its applications.

Keywords: Linear Logic, Logic Programming, Goal Directed Proof, Lazy Evaluation, Implementation Techniques.

1 Introduction

There has been a significant amount of interest of late in the design and development of logic programming languages based on linear logic. This is perhaps not surprising, given that linear logic is a often described as a logic of resources, which makes it directly applicable to many computer science tasks, including concurrency, updates and natural language processing.[1]

Traditional logic programming languages, such as Prolog, are based on classical logic. Linear logic can be seen as a refinement of classical logic, in that there is a fragment of linear logic which has precisely the same properties as classical logic; at the same time however, linear logic contains features which are not present in classical logic. In essence, these features are due to the property of linear logic that formulae generally have to be used exactly once in a linear proof. Hence each formula must be examined (i.e. one cannot simply ignore certain formulae, as can be done in classical logic), and cannot be copied. There are some exceptions to this general rule which allow linear logic to recapture classical features, but by having this default behaviour, a variety of concepts which are either awkward or impossible to deal with in classical logic can be captured in linear logic simply and elegantly. As a result, logic programming languages based on linear logic contain a variety of constructs which are not present in (pure) Prolog, thus providing novel and interesting extensions to the language.

There have been various proposals for linear logic programming languages, including LO! [2], Lolli [9] and ACL [11]. In this paper we describe the implementation of the language *Lygon*, which is based on the proof-theoretic analysis of [7]. This analysis identifies a fragment of linear logic for which the search strategy of *goal-directed*, or *uniform*, proofs is complete. This strategy forms the conceptual starting point of an

[1] A full introduction to linear logic is beyond the scope of this paper — see [4, 14], amongst others, for more details.

implementation of the language. The notion of goal-directed proof is defined in terms of cut-free proofs in the linear sequent calculus (the rules for which are given in appendix A); essentially, a goal-directed proof is one in which the next step in the search is determined by the formulae in the succedent (i.e. the formulae on the right of $\vdash$) , but not by those in the antecedent (i.e. on the left of $\vdash$). This seems to correspond to the way that logic programs execute, in that a program is written to specify what computations are to be performed, but the overall shape of the derivation is very much determined by what goal is asked. It is shown in [7] that there is a large fragment of linear logic for which goal-directed proofs are complete, and hence forms a logic programming language, following the methodology of [12].

This may be thought of as using a particular method of proof search, which appears to be particularly applicable to computational tasks, to determine a class of formulae (known as *definite* formulae) which may be used as a logic programming language. This class of programs not only contains the usual Horn clauses, but also various features which are peculiar to linear logic. Such features include global variables, a mutual exclusion operator, a notion of state and various constructs for manipulating clauses. A fuller description of these features than is possible in this paper may be found in [8].

Having found the appropriate fragment of linear logic, it then remains to determine how to implement an interpreter for the language. Whilst the notion of a *resolution proof* was used in [7] as a specialisation of the sequent calculus to definite formulae, there remains a significant amount of non-determinism. For example, consider the following proof in the linear sequent calculus.

$$\dfrac{\dfrac{}{p \vdash p}\,Ax \qquad \dfrac{}{q \vdash q}\,Ax}{p, q \vdash p \otimes q}\,\otimes$$

Whilst this is a simple enough proof, when searching for a proof of a sequent of this form — say $P \vdash G_1 \otimes G_2$ — it is, in general, necessary to split the program up into two mutually exclusive and exhaustive sub-programs P_1 and P_2 such that both $P_1 \vdash G_1$ and $P_2 \vdash G_2$ are provable. The sequent calculus does not specify how such a decomposition of the program might be found; only that it needs to be done. Hence it is the task of the implementor to develop a systematic and deterministic manner in which to find such proofs.

Note that a simple brute force approach to this problem results in an exponential algorithm; there are an exponential number of sub-(multi)sets of a given (multi)set. A better way to solve the problem of splitting the program is to do it *lazily* and *sequentially*, which is the approach taken in Lolli [9]. Using this method, the entire program is passed to the first conjunct, which uses whatever resources are necessary, and passes any excess resources to the second conjunct. Using this technique, when attempting to find a proof of the sequent $p, q \vdash p \otimes q$, we first attempt to find a proof of the sequent $p, q \vdash p$, which we discover is provable if the q is discounted. Hence we pass the "excess" resource q to the other conjunct, and hence attempt to find a proof of $q \vdash q$, which is clearly provable, and without any resources remaining unused. Hence we conclude that the original sequent is provable.

This may appear to be a simple enough technique. However, there are a number of details which, when combined together, make this process somewhat intricate, and hence care and precision is needed. One may think of this process as removing the restriction that formulae used in the $\otimes$-R and $\bindnasrepma$-L rules (which we often refer to as *multiplicative* rules) must go to exactly one of the two branches. The advantage in doing this is that we do not need to know in advance to which branch a given formula needs to be allocated. The disadvantage is that we have to find some means to ensure that this process remains sound, i.e. that each formula ends up in the "right" place, and exactly one place at that. In particular, we need to be able to reconstruct a proof in the sequent calculus from the proof search process. This requires us to ensure that certain formulae are *not* passed to another branch via the lazy mechanism. For instance, in the example above we decomposed the sequent $p, q \vdash p \otimes q$ into $p, q \vdash p$, and $\vdash q$, on the understanding that the excess from the first conjunct was to be used in the second. However, as the p in the succedent must appear on the left-hand branch in order for the formula $p \otimes q$ to appear in the appropriate part of the proof, this p cannot be passed to the second conjunct. On the other hand, the q can be passed to the other conjunct, as it is not required to construct any formula which *must* be on this part of the proof tree. One way to think of this restriction is that only formulae which exist at the time of the decomposition of the goal containing $\otimes$ may be passed to the second branch. Hence, whilst this lazy sequential technique appears to be simple enough, there is a certain amount of devil in the detail.

It should also be said that this is not necessarily the only way of calculating the way that the program should be split. Another possibility is to allow both conjuncts to execute in parallel, whilst maintaining the constraint that no formula may be used by both conjuncts. This achieves an effect like co-routining, in that the two processes may execute independently except for access to shared resources, which need to be allocated by a central server.

Whilst this method is technically sound, it is conceptually more complex than the lazy sequential one, and arguably less useful. As discussed in [9] and [8], the lazy sequential technique is very useful for introducing a notion of state to logic programming languages, in that the excess resources passed from one state to another may be seen as a state resulting from one sub-computation, and passed to another. This technique has some very useful applications, such as natural language parsing, updates and bin-packing programs [9, 8].

In this paper, we describe an implementation of Lygon which addresses these and other problems. In particular, this implementation uses the lazy sequential technique to deal with multiplicative rules. A recurrent theme throughout this exercise has been the principle of making the proof search process deterministic. This not only applies to the lazy mechanism used to implement $\otimes$, but also to constructs which require the proof search process to make a choice, and possibly to backtrack over this choice. Whilst this is a familiar feature of Prolog implementations, the range of possible choices is increased in Lygon.

This paper is organized as follows: in § 2 we discuss the logical background of Lygon, and in § 3 its operational model. § 4 explains the meaning of the connectives in a logic programming context. In § 5 we describe and discuss the precise way that the non-determinacies of the operational model are addressed, and give various examples of executable Lygon programs in § 6. Finally in § 7 we present our conclusions and discuss some possibilities for further work.

2 Logical Preliminaries

One of the most common examples of the use of linear logic is money. If a single dollar is represented by the predicate `dollar`, then the property of having two dollars may be specified by the conjunction of `dollar` with itself, i.e. `dollar` $\otimes$ `dollar` (pronounced "dollar cross dollar"). In classical logic, this would be represented by `dollar` $\wedge$ `dollar`, which is equivalent to `dollar`, i.e. that having two dollars is equivalent to one dollar, which is clearly nonsensical. However, in linear logic, `dollar` $\otimes$ `dollar` and `dollar` are not equivalent, which is more appropriate. For this reason, linear logic is often described as a logic of resources rather than a logic of truth (such as classical logic) in that different amounts of the same thing are considered to be different.

Now as any schoolchild knows, having two dollars to spend means that it is possible to buy up to two dollars worth of goods, but not three. If, for example, a drink costs a dollar, then one can convert `dollar` $\otimes$ `dollar` into `drink` $\otimes$ `dollar`, and, if a packet of sweets also costs a dollar, we can also arrive at `drink` $\otimes$ `sweets`. However, we should not be able to arrive at `drink` $\otimes$ `drink` $\otimes$ `dollar`, or `sweets` $\otimes$ `sweets` $\otimes$ `sweets`, as this would require more than two dollars. (Similarly we should not be able to derive `drink` on its own, as we would still have a dollar to spend). Finally, we would also be able to derive `happy_child` from `drink` $\otimes$ `sweets`.

It is straightforward to capture this scenario in linear logic. The conversion of a dollar into a sweet or a drink is represented by a linear implication, which is written as $\multimap$. The rules for buying a drink or a packet of sweets are then

$$\texttt{dollar} \multimap \texttt{drink}$$
$$\texttt{dollar} \multimap \texttt{sweets}$$

and the rule that a child with a drink and a packet of sweets is happy is written as follows:

$$(\texttt{drink} \otimes \texttt{sweets}) \multimap \texttt{happy\_child}$$

Using these rules, it is straightforward to derive `happy_child` from `dollar` $\otimes$ `dollar`, as expected, or from equivalent combinations such as `sweets` $\otimes$ `dollar`.[2] However, it is not possible to derive `happy_child` from `dollar` alone.

[2] Note that $\otimes$ is commutative, so that the order does not matter, and hence $p \otimes q$ is linearly equivalent to $q \otimes p$.

In this way, linear logic allows the direct and simple statement of problems which involve the notion of resources, in a way that classical logic cannot match. As a result, linear logic has been applied to the study of concurrency, as well as to knowledge representation.

However, reasoning in linear logic need not be totally different from classical logic. It is possible to re-introduce classical logic by the means of two connectives ! and ?. Essentially, a formula beginning with ! in an antecedent or with ? in a succedent behaves classically, in that such a formula may copied arbitrarily many times, and may be ignored in a proof, if desired. Hence whilst **`dollar`** corresponds to *exactly* one dollar, ! **`dollar`** corresponds to an arbitrary number of dollars, including 0. In this way we may think of a formula $!F$ in linear logic as representing an unbounded resource, i.e. one that may be used as many times as we like. Thus classical logic may be seen as a particular fragment of linear logic, in that there is a class of linear formulae which precisely matches classical formulae. This makes linear logic a conservative extension of classical logic, in that anything that can be done with classical logic can be done with linear logic (but not vice-versa, of course).

An example of the use of ! may be found if we return to the drinks and sweets example. As written above, the rules for turning a dollar into either a drink or a packet of sweets are themselves linear formulae, and hence have to be used *exactly* once. However, there is nothing in the problem to state that the dollars have to be converted into drinks or sweets, or that such a conversion may not take place twice or more. Hence a better representation of the problem is to precede each of the rules with !, so that we have

$$\begin{gathered} !\,(\texttt{dollar} \multimap \texttt{drink}) \\ !\,(\texttt{dollar} \multimap \texttt{sweets}) \\ !\,((\texttt{drink} \otimes \texttt{sweets}) \multimap \texttt{happy\_child}) \end{gathered}$$

From these rules and the information **`dollar`** $\otimes$ **`dollar`** we can then derive information such as **`drink`** $\otimes$ **`drink`**, or that from four dollars we can have two happy children. This process may be thought of as "localising" the classical part of the reasoning, in that we can use some formulae an arbitrary number of times, whereas other formulae can be used only once. In this sense linear logic can be said to make finer distinctions than is possible in classical logic.

The other way that linear logic can make finer distinctions than classical logic is in the area of the binary connectives. In classical logic, there is one conjunction and one disjunction; in linear logic, there are two of each. We have already seen one conjunction, $\otimes$, which may be thought of as an accumulator of resources: $p \otimes q$ means the resource p together with the resource q, so that **`dollar`** $\otimes$ **`sweet`** signifies that we found the resources necessary to obtain both one dollar and a packet of sweets. The other conjunction, denoted as $\&$ and pronounced "with", is not accumulative: $p \& q$ expresses that both p and q may be derived, but not necessarily by separate resources. For example, from **`dollar`** it is possible to derive both **`drink`** and **`sweet`**, and so we can derive **`drink`** $\&$ **`sweet`** from a single dollar. Hence $\&$ represents an *internal choice*; as both formulae may be derived, we are free to choose either one.

For similar reasons, there are two disjunctions in linear logic, each of which is the dual of one of the conjunctions. The dual of $\&$ is often written as $\oplus$ ("either"), and represents an *external choice*: if all we know is that $p \oplus q$ can be derived, then we know that one of p and q must be derivable, but we do not know which, per se. This has a similar feel, in many ways, to the standard classical disjunction. The other disjunction is the dual of $\otimes$, and is sometimes written $⅋$ (and sometimes other ways as well). $⅋$ is pronounced as "tensum" or "par". If $p \otimes q$ represents an accumulation of the resources p and q, the dual of this notion may be thought of as the accumulation of *promises to provide resources*, or debts. In this sense, $p ⅋ q$ represents the "consolidation" of the debts p and q.

Each of these four connectives also has a unit, which, for $\otimes$ and $\&$ are written as $\mathbf{1}$ and $\top$, and which may be thought of as generalisations of the boolean value true, and for $⅋$ and $\oplus$ are written as $\perp$ and $\mathbf{0}$, and which may be thought of as generalisations of the boolean value false.

Linear logic also contains a negation, which behaves in a manner reminiscent of classical negation. The negation of a formula F is written as $F^{\perp}$. The following laws, reminiscent of the de Morgan laws, all hold:

$$\begin{gathered} (F_1 \otimes F_2)^{\perp} \equiv (F_1)^{\perp} ⅋ (F_2)^{\perp} \\ (F_1 ⅋ F_2)^{\perp} \equiv (F_1)^{\perp} \otimes (F_2)^{\perp} \\ (F_1 \oplus F_2)^{\perp} \equiv (F_1)^{\perp} \& (F_2)^{\perp} \\ (F_1 \& F_2)^{\perp} \equiv (F_1)^{\perp} \oplus (F_2)^{\perp} \end{gathered}$$

Hence linear logic has many symmetric properties reminiscent of classical logic. A common example of these symmetries is the following restaurant menu:

fruit or seafood (in season)
main course
sweets
tea or coffee

From the point of view of the customer, this corresponds to the formula below:

$$(\texttt{fruit} \oplus \texttt{seafood}) \otimes \texttt{main} \otimes \texttt{sweets} \otimes (\texttt{tea} \,\&\, \texttt{coffee})$$

The decision about whether the entree is fruit or seafood will be made by the chef, depending on what is in season, current price etc., and so for the customer, this choice is external. However, the choice about tea or coffee is made by the customer, and hence is an internal choice. The addition of courses to make up a meal is clearly cumulative, in that each extra course requires extra resources (and hence extra cost).

The view of the menu for the restauranteur, however, is given by the formula below:

$$(\texttt{fruit}^\perp \& \texttt{seafood}^\perp) \parr \texttt{main}^\perp \parr \texttt{sweets}^\perp \parr (\texttt{tea}^\perp \oplus \texttt{coffee}^\perp)$$

Each formula here is negated, as the restauranteur has to *supply* each of the named items, rather than acquire each one. For similar reasons, the courses are joined together with $\parr$, rather than $\otimes$. The internal and external choices are also, of course, swapped.

There is far more to linear logic than can be discussed in this paper; for a more complete introduction see the papers [4, 14], among others.

3 Linear Logic Programming

As far as logic programming languages are concerned, it should be clear that linear logic provides scope for a significant extension to the features of languages such as Prolog. However, in order to identify a logic programming language in a given logic, it is necessary to have some criterion for identifying such languages. One such criterion is the completeness of a class of proofs known as *goal-directed* proofs, which is defined in terms of proofs in the sequent calculus. A precise definition of this class of proofs is beyond the scope of this paper, but the basic idea is that the goal (i.e. the succedent) alone determines the next step in the search for a proof, and not the program (i.e. the antecedent). A logic programming language is then a class of formula for which such proofs completely characterize logical consequence [12]. An analysis along these lines for linear logic was given in [7]. This leads to the following class of formulae forming a logic programming language:

$$\begin{array}{lcl} D & ::= & A \mid \mathbf{1} \mid \perp \mid D \& D \mid D \otimes D \mid D \parr D \mid \forall x\,.\,D \mid !\,D \mid G \multimap A \mid G \multimap \perp \mid G \multimap \mathbf{1} \\ G & ::= & A \mid \mathbf{1} \mid \perp \mid \top \mid G \otimes G \mid G \oplus G \mid G \parr G \mid G \& G \mid D \multimap G \mid \forall x\,.\,G \mid \exists x\,.\,G \mid !\,G \mid ?\,G. \end{array}$$

where A ranges over atomic formulae.

In [7] it is shown how the rules of the sequent calculus may be specialized for this class of formulae, and we refer to such proofs as *resolution proofs*. In particular, such proofs only allow the left rules of the sequent calculus to be used under certain circumstances, which thus form the linear version of the resolution rule.

Whilst resolution proofs provide a basic strategy for finding proofs for the above class of formulae, there is still a significant amount of non-determinism in them. In particular, there is no mechanism for splitting programs, and issues such as unification are not addressed. As discussed above, we have implemented a mechanism for splitting the program using a lazy sequential evaluation of $\otimes$. In order to make this systematic, we will use not just a sequent $\mathcal{P} \vdash \mathcal{G}$, but a *hypersequent* $\mathcal{P} \vdash \mathcal{G} \Rightarrow \mathcal{P}' \vdash \mathcal{G}'$, with the intuitive meaning that the sequent $\mathcal{P} \vdash \mathcal{G}$ is derivable if the "excess" formulae $\mathcal{P}'$ and $\mathcal{G}'$ are deleted from $\mathcal{P}$ and $\mathcal{G}$ respectively. Hence we can specify the behaviour of this lazy mechanism by writing down the appropriate rules for the hypersequents. However, the lazy mechanism means that at any point in the proof, the current list of resources may be an overestimate of the resources that are "really" available at any one point, and so the rules for the hypersequents need to take this into account. In particular, the rule for $\&$ needs to be careful, as otherwise it is possible to find lazy proofs of sequents which do not have proofs in the sequent calculus. For example, consider the sequent $p, q \vdash p\&q$. This does not have a proof in the linear sequent calculus. However, there are lazy proofs of the hypersequents

$$p, q \vdash p \Rightarrow q \vdash \perp$$

and

$$p, q \vdash q \Rightarrow p \vdash \perp$$

Hence the rule for $\&$ must insist that the "returned" resources on each branch are the same. Full details of the rules for hypersequents are beyond the scope of this paper. However, some important properties of the system of hypersequents are discussed in § 5.

4 What do all these connectives mean?

The idea behind uniform proofs can be summarised with the slogan

Connectives as instructions

We consider connectives in the goal as instructions which when followed dictate the shape of the proof.

A problem which arises in the context of Lygon due to the support of multiple conclusion sequents is the question of *which* formula we choose to handle first. For instance the formula $(a \oplus b)\parr(a^\perp \& b^\perp)$ has a proof but only if the $\&$ is done first ...

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, a^\perp}\,Ax}{\vdash a \oplus b, a^\perp}\,\oplus \quad \dfrac{\dfrac{}{\vdash b, b^\perp}\,Ax}{\vdash a \oplus b, b^\perp}\,\oplus}{\vdash a \oplus b, a^\perp \& b^\perp}\,\&}{\vdash (a \oplus b)\parr(a^\perp \& b^\perp)}\,\parr \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, a^\perp}\,Ax \quad \vdash a, b^\perp}{\vdash a, a^\perp \& b^\perp}\,\&}{\vdash a \oplus b, a^\perp \& b^\perp}\,\oplus}{\vdash (a \oplus b)\parr(a^\perp \& b^\perp)}\,\parr$$

An implementation of Lygon must have a complete strategy for selecting the next formulae. Our strategy is explained in section 6.1; for now simply assume that when there are multiple formulae on the right the system magically selects one that will work.

The Connectives as Instructions

- $\otimes$ This is a conjunction – both subformulae need to be proven. The rest of the sequent is split between them. This can be thought of as a sequential conjunction, however $\otimes$ is nonetheless commutative.

$$\vdash G_1 \otimes G_2, \Gamma_1, \Gamma_2 \ \text{ if } \ \vdash G_1, \Gamma_1 \text{ and } \vdash G_2, \Gamma_2$$

- $\&$ Another conjunction – both subformulae need to be proven. Both subformulae however, get the current context. This can be thought of as a `fork()` operation.

$$\vdash G_1 \& G_2, \Gamma \ \text{ if } \ \vdash G_1, \Gamma \text{ and } \vdash G_2, \Gamma$$

- $\parr$ Adds both formulae to the context.

$$\vdash G_1 \parr G_2, \Gamma \ \text{ if } \ \vdash G_1, G_2, \Gamma$$

- $\oplus$ Selects (nondeterministically) one of the formulae to replace it. The standard disjunction.

$$\vdash G_1 \oplus G_2, \Gamma \ \text{ if } \ \vdash G_1, \Gamma \text{ or } \vdash G_2, \Gamma$$

$\perp$ This constant is simply dropped from the context.

$$\vdash \perp, \Gamma \text{ if } \vdash \Gamma$$

0 This constant is false and unprovable.

1 This is provable *provided* the context is empty

$$\vdash \mathbf{1}, \Gamma \text{ if } \Gamma = \emptyset$$

$\top$ This is provable in any context.

$$\vdash \top, \Gamma$$

$\multimap$ We treat $\multimap$ using the equivalence $a \multimap b \equiv (a^{\perp}) \parr b$

? Formulae of the form $?F$ are *nonlinear*. They can be copied and deleted freely. A standard trick is to automatically delete nonlinear formulae at the leaves of a proof rather than during a proof. This means that the only thing we can do with a nonlinear formula during a proof is copy it.

$$\vdash ?F, \Gamma \text{ if } \vdash F, ?F, \Gamma$$

! This connective can be dropped *provided* there are no linear formulae around.

$$\vdash !F, \Gamma \text{ if } \vdash F, \Gamma$$

provided Γ contains only formulae of the form $?F$.

$\forall$ The quantifiers are the same as in classical logic.

$$\vdash \forall x F, \Gamma \text{ if } \vdash F[y/x], \Gamma$$

where y is not free in Γ.

$\exists$ Same as in classical logic.

$$\vdash \exists x F, \Gamma \text{ if } \vdash F[t/x], \Gamma$$

$A^{\perp}$ In the single sided sequent calculus negation is only ever applied to atoms (this can be ensured through applications of de Morgan laws). Negated atoms are facts and cannot be viewed as instructions.

A Atoms are treated using either the Axiom rule or the Resolution rule

$$\vdash A, A^{\perp}, \Gamma \text{ if } \Gamma = \emptyset$$

OR

$$\vdash A, \Gamma \text{ if } \vdash P, A, \Gamma$$

where the program contains the clause P

5 The Implementation

Having familiarised ourselves with the intended meaning of the connectives as dictated by linear logic we now come to consider how they may be implemented in an efficient manner.

To begin with, we shall use a couple of standard tricks to make life easier and to gain some efficiency. These are:

1. Isolating nonlinear formulae into a separate area in the sequent. This enables us to duplicate them easily at a $\otimes$ rule and to ignore them at axioms.

2. Using a one sided sequent calculus to reduce the number of rules considered.

As mentioned above we need to implement lazy resolution in order to avoid exponential complexity on the size of the context for goals involving $\otimes$. It turns out that implementing lazy resolution soundly requires the modification of the $\top$ and $\&$ rules. The modifications required to implement lazy resolution in a sound and complete manner are more subtle then one might expect.

A naive implementation of lazy resolution would simply extend sequents with an extra field for returned formulae and modify rules to pass on excess formulae. Unfortunately this naive implementation is unsound – the formula $(\mathbf{1}\parr b)\otimes b^\perp$ is unprovable as the standard sequent proof attempt shows. However, if we naively pass unused resources along (the b) then a lazy proof is possible:

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

To avoid this problem we need to introduce a "proof of purchase" marker (denoted by a superscript $\top$). When a formula is passed into the left branch of a $\otimes$ rule we mark it. Any attempt to use the formula must first remove the tag. When the axioms attempt to pass on unneeded formulae they can only pass on formulae with an intact tag. Note that the principal formulae (the G_i in $G_1 \otimes G_2$) are not tagged.

In the above example tags are not actually used – their absence makes the passing of b impossible and hence prevents us from proving the nontheorem. The following example show the use of tags and show that tags have to be nestable.

Consider the formulae

$$((a\parr(\mathbf{1}\otimes a^\perp))\otimes b)\parr b^\perp$$

It is provable with sequent proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1} \qquad \dfrac{}{\vdash a, a^\perp}\,Ax}{\vdash a, \mathbf{1}\otimes a^\perp}\,\otimes}{\vdash a\parr(\mathbf{1}\otimes a^\perp)}\,\parr \qquad \dfrac{}{\vdash b, b^\perp}\,Ax}{\vdash (a\parr(\mathbf{1}\otimes a^\perp))\otimes b, b^\perp}\,\otimes}{\vdash ((a\parr(\mathbf{1}\otimes a^\perp))\otimes b)\parr b^\perp}\,\parr$$

And with lazy resolution proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}, a^\top, {b^\perp}^{\top\top} \Rightarrow a^\top, {b^\perp}^{\top\top}}\,\mathbf{1} \qquad \dfrac{}{\vdash a, a^\perp, {b^\perp}^\top \Rightarrow {b^\perp}^\top}\,Ax}{\vdash a, \mathbf{1}\otimes a^\perp, {b^\perp}^\top \Rightarrow {b^\perp}^\top}\,\otimes}{\vdash a\parr(\mathbf{1}\otimes a^\perp), {b^\perp}^\top \Rightarrow {b^\perp}^\top}\,\parr \qquad \dfrac{}{\vdash b, b^\perp \Rightarrow \emptyset}\,Ax}{\vdash (a\parr(\mathbf{1}\otimes a^\perp))\otimes b, b^\perp \Rightarrow \emptyset}\,\otimes}{\vdash ((a\parr(\mathbf{1}\otimes a^\perp))\otimes b)\parr b^\perp \Rightarrow \emptyset}\,\parr$$

Note that the top left hand side sequent contains $a^\top, {b^\perp}^{\top\top}$.

Below we describe some of the intricacies unique to linear logic programming. Other subtleties such as the implementation of nested mixed quantifiers (for example $\forall x \exists y \forall z \ldots$) are not unique to linear logic programming and so are elided.

5.1 The $\top$ rule

The $\otimes$ rule works, however the use of lazy resolution has implications for the $\top$ rule. This rule simply states that anything containing a $\top$ is provable.

$$\dfrac{}{\vdash \top, \Gamma}\,\top$$

In attempting to prove a formulae of the form

$$\vdash \top \otimes G, \Gamma$$

we pass Γ into the left branch which contains a $\top$. This branch is trivially provable – the question however arises as to what exactly should be returned?

As is demonstrated by the following two sequent proofs, this choice depends on the rest of the proof.

In the first sequent, the $\top$ rule must not consume anything since both a and b are needed for the rest of the proof. In the second sequent the $\top$ rule must consume the $b^\perp$.

$$\frac{\dfrac{}{\vdash \top}\,\top \qquad \dfrac{\dfrac{}{\vdash a^\perp, a}\,Ax \quad \dfrac{}{\vdash b^\perp, b}\,Ax}{\vdash a^\perp, b^\perp, a \otimes b}\,\otimes}{\vdash a^\perp, b^\perp, \top \otimes (a \otimes b)}\,\otimes \qquad \frac{\dfrac{}{\vdash b^\perp, \top}\,\top \qquad \dfrac{\dfrac{}{\vdash a^\perp, a}\,Ax \quad \dfrac{}{\vdash \mathbf{1}}\,\mathbf{1}}{\vdash a^\perp, a \otimes \mathbf{1}}\,\otimes}{\vdash a^\perp, b^\perp, \top \otimes (a \otimes \mathbf{1})}\,\otimes$$

The $\top$ rule then, must mark all of the formulae that it can pass on (ie those with tags) as *potentially deletable* and return them. If the rest of the proof requires these formulae they can be used; if not they can be deleted. We introduce a rule allowing us to strip away the “potentially deletable” marker (denoted by a prefix ¿) so we can use the formulae and modify the axiom rules to delete unneeded formulae of the form $\text{¿}F$.

$$\frac{}{\vdash a, a^\perp, \text{¿}\Gamma, \Delta^\top \Rightarrow \Delta^\top}\,Ax \qquad \frac{\vdash F, \Gamma \Rightarrow \Delta}{\vdash \text{¿}F, \Gamma \Rightarrow \Delta}\,\text{¿}D$$

It should be noted that delaying $\top$ is not a general solution for this problem since we can't know where $\top$s will come from. For example the formulae

$$((\top \oplus a(x)) \otimes (\top \oplus b(x))) \parr a(pizza)^\perp \parr b(pasta)^\perp$$

has a number of proofs, at the time when we choose which subformulae of the $\otimes$ to process first we can't tell which will become a $\top$.

5.2 The problem with &

The intended meaning of a formulae tagged as deletable is that *it has already either been deleted or not, we just don't know which.* Such formulae are either present and undeletable or simply not there – the rest of the proof makes the choice as to which is the case. The thorn is that the rest of the proof must be consistent about this choice. This is the difference between $\text{¿}F$ and $F \oplus \perp$.

As an example of this consider the sequent $\vdash a^\perp, b^\perp, \top \otimes (a \& b)$. As the following proof attempt shows it is not provable.

$$\frac{\dfrac{}{\vdash \top}\,\top \qquad \dfrac{\vdash a^\perp, b^\perp, a \quad \vdash a^\perp, b^\perp, b}{\vdash a^\perp, b^\perp, a \& b}}{\vdash a^\perp, b^\perp, \top \otimes (a \& b)}\,\otimes$$

However, by allowing the different branches of the & rule to make different choices as to which of the formulae passed on by $\top$ are actually there we can find a lazy resolution proof of the sequent.

$$\frac{\dfrac{}{\vdash {a^\perp}^\top {b^\perp}^\top, \top \Rightarrow (\text{¿}(a^\perp))^\top (\text{¿}(b^\perp))^\top}\,\top \qquad \dfrac{\dfrac{\dfrac{}{\vdash a^\perp, \text{¿}(b^\perp), a \Rightarrow \emptyset}\,Ax}{\vdash \text{¿}(a^\perp), \text{¿}(b^\perp), a \Rightarrow \emptyset}\,\text{¿}D \quad \dfrac{\dfrac{}{\vdash \text{¿}(a^\perp), b^\perp, b \Rightarrow \emptyset}\,Ax}{\vdash \text{¿}(a^\perp), \text{¿}(b^\perp), b \Rightarrow \emptyset}\,\text{¿}D}{\vdash \text{¿}(a^\perp), \text{¿}(b^\perp), a \& b \Rightarrow \emptyset}\,\&}{\vdash a^\perp, b^\perp, \top \otimes (a \& b) \Rightarrow \emptyset}\,\otimes$$

In order for the & rule to be sound, the two subtrees must make use of the same contexts – they must hence make the same choices with respect to which of the deletable formulae are actually deleted. In order to have the necessary accounting information to enforce this condition we must modify the axioms to return an indication of which choices were made.

The modified rules are too long to be given here and appear in appendix B.

Whilst this isn't a complete list of the issues involved in the implementation they indicate those most specific to Lygon.

6 Examples

As with other declarative languages, the formalism behind the language is only half the story. Actually programming in the language reveals uses for the constructs and features of the language. Programming idioms such as difference lists in Prolog and `map` and `fold` in functional languages are discovered only by actually using the language. Some of these programming idioms are decidedly nontrivial – witness for instance the recent development of monads as a method of structuring functional programs. On the other hand, familiarity with the formalism, while desirable in understanding the language, is not essential and is not often used in practical programming.

This section firstly introduces Lygon as it appears to a programmer who is not necessarily versed in linear logic and then considers some examples of programming idioms made possible by the use of linear logic.

6.1 The Lygon System

The current implementation of Lygon is an interpreter written in NU-Prolog[3]. It represents a first stab to enable practical experimentation with programming idioms opened up by linear logic programming. The interpreter comprises some 700 odd lines of code and was developed over a period of a couple of weeks. It currently supports a limited form of program – all program clauses are implicitly nonlinear. This is not a severe limitation since linear clauses in the program can be negated and added to the goal as a simple syntactical preprocessing step.

The concrete syntax used by the system is

$$
\begin{array}{rcl}
\mathcal{G} & ::= & A \mid \texttt{neg}A \mid \texttt{!}\mathcal{G} \mid \texttt{1} \mid \texttt{top} \mid \texttt{bot} \mid \mathcal{G}\texttt{*}\mathcal{G} \mid \mathcal{G}\texttt{\#}\mathcal{G} \mid \mathcal{G}\texttt{@}\mathcal{G} \mid \mathcal{G}\texttt{\&}\mathcal{G} \mid \mathcal{G}\texttt{->}\mathcal{G} \\
 & \mid & \texttt{forall(}x\texttt{,}\mathcal{G}\texttt{)} \mid \texttt{exists(}x\texttt{,}\mathcal{G}\texttt{)} \mid \texttt{query}\ [x_1,\ldots,x_n]\ \mathcal{G} \mid \texttt{quant}\ [x_1,\ldots,x_n]\ \mathcal{G} \mid \texttt{?}\mathcal{F}' \\
\mathcal{D} & ::= & !\mathcal{D}' \\
\mathcal{D}' & ::= & \texttt{forall(}x\texttt{,}\mathcal{D}'\texttt{)} \mid A \mid A\texttt{:-}\mathcal{G} \mid \texttt{quant}\ [x_1,\ldots,x_n]\ \mathcal{D}' \\
\mathcal{F}' & ::= & \texttt{exists(}x\texttt{,}\mathcal{F}'\texttt{)} \mid \texttt{neg}A \mid (\texttt{neg}A)\texttt{*}\mathcal{G} \mid \texttt{query}\ [x_1,\ldots,x_n]\ \mathcal{F}'
\end{array}
$$

`query` and `quant` are short hand for nested `exists` and `forall` respectively.

Predicates (A) are as in Prolog. Variables are *lower case* and are distinguished as variables by their explicit quantification.

Negation can only be applied to atoms. Again, this is not a loss since negation can be pushed inwards using de Morgan laws.

The correspondence between the concrete and abstract syntax is

$\otimes$	$\Rightarrow$	`*`
$\&$	$\Rightarrow$	`&`
$⅋$	$\Rightarrow$	`#`
$\oplus$	$\Rightarrow$	`@`
$\multimap$	$\Rightarrow$	`->`
$-^{\perp}$	$\Rightarrow$	`neg` $-$

The interpreter's strategy for selecting the next formula to be reduced is based on the notion of synchronous and asynchronous formulae [1]. A formula is described as (a)synchronous if its top connective is (a)synchronous. Asynchronous formulae[4] are those that, due to the permutability properties of linear logic, can be selected and committed to without loss of completeness. That is, no backtracking over the choice of formula will be necessary. The synchronous formulae[5] on the other hand must be selected nondeterministically since the wrong order can prevent a proof from being found. Note that resolution is synchronous since it can introduce synchronous formulae. Thus the next formulae at any point is a committed choice

[3] Available by email request from the authors.
[4] $\perp$ $\top$ $\&$ $⅋$ $?$ $\forall$
[5] $\mathbf{1}$ $\mathbf{0}$ $\otimes$ $\oplus$ $!$ $\exists$

to an arbitrary asynchronous formulae if one exists, otherwise a nondeterministic choice of a synchronous formulae.

6.2 Examples

We begin with a few formulae taken from section 5

```
prompt% lygon
Lygon V0.0.1
Type ``help.''

Lygon (neg a) # (neg b) # (top * (a & b)).
Failed.
Lygon (1 # b) * (neg b).
Failed.
Lygon query [x] neg a(pizza) # neg b(pasta) # (top @ a(x)) * (top @ b(x)).
Succeeded.
x = C
More? (y/n) y
Succeeded.
x = pasta
More? (y/n) y
Succeeded.
x = pizza
More? (y/n) y
Failed.
Lygon bye.

Bye!
prompt%
```

6.2.1 States and Memory

The rest of the sequent can be used to store state information which can be modified.

For example, consider the toggling of a switch. We define the program

```
toggle :- off * neg on.
toggle :- on * neg off.
```

The effect of a call to `toggle` in a context containing the fact `off` is to retract the fact and assert the fact `on`. For example:

```
Lygon [toggle]. % Load the code above from a file.
Lygon (neg off) # toggle # on.
Succeeded.
More? (y/n) n
Lygon (neg off) # toggle # off.
Failed.
Lygon (neg off) # toggle # toggle # off.
Succeeded.
More? (y/n) n
Lygon (neg off) # toggle # toggle # on.
Failed.
```

Note that since the program is implicitly nonlinear we must supply the initial state as part of the goal. Adding `off` as a program clause would mean that even after a toggle we could still prove `off` from the clause.

6.2.2 Global Variables

Although the quantifier rules are identical to those of classical logic there are some new uses for variables due to their interaction with linear logic. When unifying a clause head with an atom we should bind variables in the clause. It is the nonlinearity of the clause that makes this redundant by allowing us to copy the clause first and operate on the copy. If the clause is linear we cannot copy and must bind variables in the clause.

Consider the formula

$$\exists x ? \exists x p(x, y)^{\perp}$$

This formula can be used for a number of different values of y by taking copies. However there can only be one value of x since x's introduction occurs outside the scope of the ?.

```
Lygon exists(x,? exists(y, neg p(x,y))) # p(a,b) * p(a,c).
Succeeded.
More? (y/n) n
Lygon exists(x,? exists(y, neg p(x,y))) # p(b,a) * p(c,a).
Failed.
```

Such unique variables are referred to as "global".

6.2.3 Mutual Exclusion

A similar idiom to global variables is mutual exclusion – by using a linear disjunction we can make the choice binding. Given the formula

$$(?p(a)) \oplus (?p(b))$$

once we have chosen to use $p(a)$ all future goals are limited to $p(a)$.

```
Lygon ((? neg p(a)) @ (? neg p(b))) # p(a).
Succeeded.
More? (y/n) n
Lygon ((? neg p(a)) @ (? neg p(b))) # p(b).
Succeeded.
More? (y/n) n
Lygon ((? neg p(a)) @ (? neg p(b))) # (p(a) * p(a)).
Succeeded.
More? (y/n) n
Lygon ((? neg p(a)) @ (? neg p(b))) # (p(a) * p(b)).
Failed.
```

6.2.4 Other applications

The semantics of & suggests that Lygon has potential to be applied to concurrent applications. This aspect of linear logic programming has been explored by [11] and [2].

Further discussion of applications of Lygon and a Bin Packing example can be found in [8]. In addition work on the application of Lolli to natural language processing carries over to Lygon [9].

7 Conclusions and Further Work

Lygon's origins were in a proof-theoretic analysis of goal-directed provability in linear logic, and hence the task of the implementor is to find a deterministic way to search for proofs in a certain fragment of linear logic. The implementation described above does this, but it represents a "first cut"; whilst the solutions used in this interpreter are reasonable, only experience with an implementation and testing it on various examples will enable us to determine the best techniques to use. We intend to experiment with this preliminary implementation to gain experience in writing Lygon programs, and to implement, test and measure various applications of linear logic programming in a real system.

Much work remains to be done, of course. Experience with various programs will presumably suggest optimizations that can be done, and there is a large and growing body of literature on the optimization of logic programming languages. It is reasonable to predict that working with this interpreter for a few months will enable us to write a more efficient interpreter later. One significant area of optimization is choice of which formula to reduce next. There are certain constraints implied by the rules of the sequent calculus, as well as various pragmatic considerations (such as minimising the number of choice points) which will have an influence on the optimal order of reduction. We intend to conduct both a theoretical and empirical analysis of various reduction strategies to determine what is best.

Another area of research is the precise relationship between derivations and proofs. For example, it is straightforward to find a proof of a sequent containing $\top$, as such a sequent is an axiom in the sequent calculus. However, there are in fact many proofs of such a sequent, as one may reduce other formulae before "finding" the $\top$. Thus if $\top$ has sufficiently low priority as a goal, we may continue computation until $\top$ is the only alternative, in which case we must succeed. Hence if we are concerned with the variety of proof found by the search process, rather than just the existence or non-existence of a proof, then we may use a wider variety of computations than may otherwise be the case.

We are also interested in developing a programming methodology for Lygon. Due to the more sophisticated features of the language, this may vary significantly from that of Prolog. In addition. work is already underway on a toolkit for debugging support for Lygon.

References

[1] J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. Journal of Logic and Computation 2(3) 1992.

[2] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. Proceedings of the International Conference on Logic Programming 496-510, Jerusalem, June, 1990.

[3] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift* 39, 176-210, 405-431, 1934.

[4] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* 50, 1-102, 1987.

[5] J.A. Harland, D.J. Pym. The Uniform Proof-theoretic Foundation of Linear Logic Programming (Extended Abstract). Proceedings of the International Logic Programming Symposium 304-318, San Diego, October, 1991.

[6] J.A. Harland, D.J. Pym. Resolution in Fragments of Classical Linear Logic (Extended Abstract). Proceedings of the Russian Conference on Logic Programming and Automated Reasoning 30-41, St. Petersburg, July, 1992.

[7] J.A. Harland, D.J. Pym. A Uniform Proof-theoretic Investigation of Linear Logic Programming. Journal of Logic and Computation 4:2:175-207, April, 1994.

[8] J.A. Harland, D.J. Pym. A Note on the Implementation and Applications of Linear Logic Programming Languages. Proceedings of the Seventeenth Annual Computer Science Conference 647-658, Christchurch, January, 1994.

[9] J. Hodas, D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic: Extended Abstract. Proceedings of the Symposium on Logic in Computer Science 32-42, Amsterdam, July 1991.

[10] S.C. Kleene. *Introduction to Metamathematics.* North-Holland, 1952.

[11] N. Kobayash, A. Yonezawa. ACL - A Concurrent Linear Logic Programming Paradigm. Proceedings of the International Logic Programming Symposium, 279-294, Vancouver, October, 1993.

[12] D. Miller, G. Nadathur, F. Pfenning, A. Ščedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic* 51, 125-157, 1991.

[13] G. Nadathur and D.A. Miller. An Overview of λProlog. Proceedings of the International Conference and Symposium on Logic Programming 810-827, Seattle, August, 1988.

[14] A. Ščedrov. A Brief Guide to Linear Logic. in *Current Trends in Theoretical Computer Science*. G. ROzenberg and A. Salolmaa (eds.), World Scientific, 1993.

[15] D.J. Pym. Errata and Remarks. University of Edinburgh LFCS Report, ECS-LFCS-93-265, 1993.

[16] S. Read. *Relevant Logic: A Philosophical Examination of Inference*. Blackwells, 1988.

A The Linear Sequent Calculus

$$\frac{}{\phi \vdash \phi}\ \text{axiom}$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Gamma', \phi \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \text{cut}$$

$$\frac{\Gamma, \phi, \psi, \Gamma' \vdash \Delta}{\Gamma, \psi, \phi, \Gamma' \vdash \Delta}\ \text{X-L} \qquad \frac{\Gamma \vdash \Delta, \phi, \psi, \Delta'}{\Gamma \vdash \Delta, \psi, \phi, \Delta'}\ \text{X-R}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \mathbf{1} \vdash \Delta}\ \mathbf{1}\text{-L} \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1}\text{-R}$$

$$\frac{}{\bot \vdash}\ \text{L}^{\bot} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}\ \text{R}^{\bot}$$

$$\frac{}{\Gamma \vdash \top, \Delta}\ \top\text{-R}$$

$$\frac{}{\Gamma, \mathbf{0} \vdash \Delta}\ \mathbf{0}\text{-L}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \phi^{\bot} \vdash \Delta}\ \bot\text{-L} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \phi^{\bot}, \Delta}\ \bot\text{-R}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \otimes \psi \vdash \Delta}\ \otimes\text{-L} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma' \vdash \psi, \Delta'}{\Gamma, \Gamma' \vdash \phi \otimes \psi, \Delta, \Delta'}\ \otimes\text{-R}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \phi \& \psi \vdash \Delta} \qquad \frac{\Gamma, \psi \vdash \Delta}{\Gamma, \phi \& \psi \vdash \Delta}\ \&\text{-L} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \& \psi, \Delta}\ \&\text{-R}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \oplus \psi \vdash \Delta}\ \oplus\text{-L} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \oplus \psi, \Delta} \qquad \frac{\Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \oplus \psi, \Delta}\ \oplus\text{-R}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Gamma', \psi \vdash \Delta'}{\Gamma, \Gamma', \phi \parr \psi \vdash \Delta, \Delta'}\ \parr\text{-L} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \parr \psi, \Delta}\ \parr\text{-R}$$

$$\frac{\Gamma \vdash \phi, \Delta \qquad \Gamma', \psi \vdash \Delta'}{\Gamma, \Gamma', \phi \multimap \psi \vdash \Delta, \Delta'}\ \multimap\text{-L} \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \multimap \psi, \Delta}\ \multimap\text{-R}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ !\text{-L (!D)} \qquad \frac{!\Gamma \vdash \phi, ?\Delta}{!\Gamma \vdash !\phi, ?\Delta}\ !\text{-R (!S)}$$

$$\frac{!\Gamma, \phi \vdash ?\Delta}{!\Gamma, ?\phi \vdash ?\Delta}\ ?\text{-L (?S)} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash ?\phi, \Delta}\ ?\text{-R (?D)}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ W!\text{-L (!W)} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?\phi, \Delta}\ W?\text{-R (?W)}$$

$$\frac{\Gamma, !\phi, !\phi \vdash \Delta}{\Gamma, !\phi \vdash \Delta}\ C!\text{-L (!C)} \qquad \frac{\Gamma \vdash ?\phi, ?\phi, \Delta}{\Gamma \vdash ?\phi, \Delta}\ C?\text{-R (?C)}$$

$$\frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, \forall x \,.\, \phi \vdash \Delta}\ \forall\text{-L} \qquad \frac{\Gamma \vdash \phi[y/x], \Delta}{\Gamma \vdash \forall x \,.\, \phi, \Delta}\ \forall\text{-R}$$

$$\frac{\Gamma, \phi[y/x] \vdash \Delta}{\Gamma, \exists x \,.\, \phi \vdash \Delta}\ \exists\text{-L} \qquad \frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash \exists x \,.\, \phi, \Delta}\ \exists\text{-R}$$

where y is not free in Γ, Δ.

B The Modified Rules

The symbols Σ , Π and Ξ represent multisets of formulae of the form $-^{\top}$. These are formulae being passed through and returned from a branch of the $\otimes$ rule.
The symbols $\aleph$, $\beth$, $\gimel$ and $\daleth$ represent multisets of formulae of the form $¿-$. These are the "potentially deletable" formulae returned by the $\top$ rule.
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. c represents a formula whose topmost connective is not a superscript $\top$.
The notation $a^{\top n}$ represents the formula where a is superscripted by n $\top$s.
The function f used in the top is given by

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

The intention is that formulae satisfy the grammar:

$$
\begin{aligned}
\mathcal{F} &::= F^{\top} \mid F' \\
\mathcal{F}' &::= ¿\mathcal{F}'' \mid \mathcal{F}'' \\
\mathcal{F}'' &::= \ldots
\end{aligned}
$$

These rules are modified from the rules in the previous appendix in the following ways:

1. We use a one sided sequent calculus.
2. We segregate the nonlinear part of the sequent for easy identification. (This is the δ on the left of the rule). Axiomatic rules (Ax, $\mathbf{1}$ and $\top$) are modified to ignore nonlinear formulae.
3. Rules are modified to return unused formulae (Π , Σ etc.)
4. Rules are modified to return an indication of which potentially deletable formulae ($¿F$) were not used ($\aleph$, $\beth$ etc.)
5. We introduce rules for eliminating the tags. *Steal* eliminates the proof of purchase tag ($F^{\top}$) and $¿D$ chooses not to delete a potentially deletable formula.

$$\dfrac{}{\delta : p, p^{\perp}, \Pi, \beth \Rightarrow \Pi, \beth}\ \mathrm{Ax} \qquad \dfrac{}{\delta : \mathbf{1}, \Pi, \beth \Rightarrow \Pi, \beth}\ \mathbf{1}$$

$$\dfrac{}{\delta : \top, \Gamma, \Pi, \beth \Rightarrow (f\ \Pi), \beth}\ \top$$

$$\dfrac{\delta : c, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}{\delta : \Gamma, c^{\top n}, \Pi, \beth \Rightarrow \Sigma, \aleph}\ \mathrm{Steal} \qquad \dfrac{\delta : \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}{\delta : \perp, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}\ \perp$$

$$\dfrac{\delta : a, \Gamma^{\top}, \Pi^{\top}, \beth^{\top} \Rightarrow \Delta^{\top}, \Xi^{\top}, \gimel, \daleth^{\top} \qquad \delta : b, \Delta, \Xi, \daleth \Rightarrow \Sigma, \aleph}{\delta : a \otimes b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \gimel, \aleph}\ \otimes$$

$$\dfrac{\delta : a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph \qquad \delta : b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}{\delta : a \& b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}\ \&$$

$$\dfrac{\delta : a_k, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}{\delta : a_1 \oplus a_2, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}\ \oplus \quad k \in \{1, 2\}$$

$$\dfrac{\delta : a \Rightarrow}{\delta : !a, \Pi, \beth \Rightarrow \Pi, \beth}\ !S \qquad \dfrac{\delta : a, b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}{\delta : a \parr b, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}\ \parr$$

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

$$\dfrac{\delta : a, \Gamma, \Pi, \beth \Rightarrow \Sigma, \aleph}{\delta : \Gamma, \Pi, ¿a, \beth \Rightarrow \Sigma, \aleph}\ ¿D$$