Making Logic Programs Reactive

James Harland
Department of Computer Science
RMIT
GPO Box 2476V
Melbourne, 3001
Australia
jah@cs.rmit.edu.au
Phone: +61 3 9925 2348
Fax: +61 3 9662 1617

Michael Winikoff
Department of Computer Science
University of Melbourne
Parkville, 3052
Australia
winikoff@cs.mu.oz.au
+61 3 9344 9100
+61 3 9348 1184

June 11, 1998

Abstract

Logic programming languages based on linear logic have been of recent interest, particularly as such languages provide a logical basis for programs which execute within a dynamic environment. Most of these languages are implemented using standard resolution or *backward-chaining* techniques. However, there are applications for which the use of *forward-chaining* techniques within a dynamic environment are appropriate, such as genetic algorithms, active databases and agent-based systems, and for which it is difficult or impossible to specify an appropriate goal in advance. In this paper we discuss the foundations for a forward-chaining approach (or in logic programming parlance, a bottom-up approach) to the execution of linear logic programs, which thus provides forward-chaining within a dynamic environment. In this way it is possible not only to execute programs in a forward-chaining manner, but also to combine forward- and backward-chaining execution. We describe and discuss the appropriate inference rules for such a system, the formal results about such rules, the role of search strategies, and applications.

1 Introduction

In recent years there has been a significant amount of interest in logic programming languages based on linear logic [7], a logic designed with bounded resources in mind. Due to the resource-sensitive nature of linear logic, such languages such as Lygon[8], Lolli[9], Forum[12], ACL[10], LC[19], and LO[3] have been applied to concurrency, updates, knowledge representation, logical interpretations of actions and graph search algorithms. These languages are generally implemented in a top-down manner,[1] using the standard techniques of resolution and unification, in that given a program P and a goal $\mathcal{G}$, the key question is to determine whether or not $P \vdash \mathcal{G}$, and hence computation proceeds by decomposing $\mathcal{G}$ into a sufficiently simple form, and then applying resolution to generate a new goal. The precise details of the computational method vary from system to system, depending on

[1]Here we use the term in the usual logic programming sense, in which top-down is synonymous with backward-chaining. However, as proof trees are written upside-down by logic programmers, and the right way up by proof theorists, such terms can be quite confusing. In this paper, we adhere to the logic programming usage, i.e. that top-down execution refers to backward-chaining techniques, and bottom-up to forward-chaining ones.

exactly what restrictions are placed on the formulae in P and $\mathcal{G}$, and what search strategy is used. However, these methods, apart from ACL, all generally use the "query-and-answer" paradigm, in that a query (the goal) is posed to the program, and if the goal succeeds, then some kind of answer information that has been calculated in the course of the proof is returned.

When compared with traditional logic programming languages such as Prolog, these languages have the advantage of introducing dynamic behaviour within a logical framework. However, there are some applications in which such *backward-chaining* techniques[2] are not as natural as *forward-chaining* ones[3]. For example, in [8] it is discussed how problems which involve rule-based changes of state, such as the blocks world, bin-packing problems, or the Yale shooting problem, fit more naturally into a paradigm in which the output of the computation is a multiset of formulae (i.e. resources) rather than an answer substitution or something similar. The most natural outcome of a bin-packing program is a representation of the final state of the bins, or for the blocks world, the most natural outcome is a representation of the final state of the world. This approach to logic programs is somewhat different from the query-and-answer paradigm, in that there is no real notion of success or failure, but rather a series of state changes which result in some particular outcome. This approach is more akin with that of ACL; in our case, we are interested in the ability to mix the paradigms together, and for the largest possible class of formulae.

The combination of forward-chaining and a dynamic environment provides a way for logic programs to be *reactive*, i.e. able to take action depending on the circumstances found at the time, rather than adhering to a predetermined plan. Applications such as genetic algorithms, active databases and agent systems appear to fit naturally into this paradigm, as they generally do not have a specific goal in mind, such as "move these blocks to that location" or "sell 10% of IBM stock", but are more open-ended, such as requiring that the movement of blocks be managed to minimise storage requirements, or that stock be bought and sold in order to maximise profits. For such applications it would seem that a bottom-up execution model would be appropriate. Clearly, it will be useful to be able to combine both bottom-up and top-down execution, which may be thought of as providing both reactivity and *rationality* within a dynamic environment, so that both unplanned and planned actions may be performed. Similar observations have been made about Mixlog [16], a system which uses aspects of linear logic to model both top-down and bottom-up execution methods in the same computational framework. However, in our case we are interested in allowing an interaction between both execution methods, rather than incorporating both into the one computational mechanism as is done in Mixlog.

Hence the key technical question is then to find the appropriate computational framework for bottom-up evaluation of linear logic programs. Unlike the situation for languages based on classical logic, in which there is no internal notion of dynamics, the linear logic case involves evaluating a program in a dynamic environment, and hence the program will generally change during computation. Hence it is not so much a matter of converting implicit information into explicit information (as, for example, happens in the well-known T_P^ω construction [5]) as changing the program itself to reflect the outcomes of the computation.

One of the key differences between the top-down and bottom-up approaches is that backtracking is used in the top-down paradigm to implement the "don't know" non-deterministic choices that need to be made. In the bottom-up case, it is not clear that such action is appropriate, but there does need to be some mechanism to cope with such choices. As we shall see, this comes down to an appropriate choice of the result of evaluating the program.

In addition, it seems important to permit top-down execution where appropriate, as is done

[2] i.e. given $A \to B$, conclude that to prove B it is sufficient to prove A

[3] i.e. given $A \to B$, conclude that if A has been derived, then B can be derived.

in Mixlog and in deductive databases systems which use bottom-up execution, such as Aditi [18]. As we shall see, our framework allows significant flexibility in the balance between top-down and bottom-up execution.

This paper is organised as follows. In Section 2 we give a brief introduction to linear logic, and in Section 3 we discuss the appropriate form of the inference rules of the system and in particular the role of the implication rule. In Section 4 we give a formal presentation of the inference rules and informal discussion of their properties. Section 5 contains a discussion of transitivity issues and Section 6 contains the formal soundness and completeness results. Section 7 presents the possibilities for normalization of derivations in our system, and the following section has a brief discussion of search strategies and applications. Finally we present our conclusions and possibilities for further work in Section 9.

2 Linear Logic

Linear logic contains two forms of conjunction: one which is "cumulative", i.e. for which $p \otimes p \not\equiv p$, and one which is not, i.e. $p \& p \equiv p$. Roughly speaking, the former is what allows linear logic to deal with resource issues, whilst the latter allows for these issues to be overlooked (or, more precisely, for an "internal" choice to be made between the resources used), as, by default, each formula in linear logic represents a resource which must be used exactly once.

Consider the following menu from a restaurant:

fruit or seafood (in season)
main course
all the chips you can eat
tea or coffee

Note that the first choice, between fruit and seafood, is a classical disjunction; we know that one or the other of these will be served, but we cannot predict which one, which may be thought of as an "external" choice, in that someone else makes the decision. On the other hand, the choice between tea and coffee is an "internal" choice — the customer is free to choose which one shall be served. Note the internal choice is a conjunction; in order to satisfy this, the restauranteur has to be able to supply *both* tea and coffee, and not just one of them. The chips course clearly involves a potentially infinite amount of resources, in that there is no limit on the amount of chips that the customer may order. We represent this situation by prefixing such formulae with a !. Note also that the meal consists of four components, and hence we connect the components with $\otimes$. Hence we have the following representation of the menu:

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

where we write $\oplus$ for the classical disjunction. Note that the use of ! makes it possible to recover classical reasoning, as formulae beginning with ! with ? in a succedent behaves classically, in that such formulae may used arbitrarily many times, including 0, rather than exactly once. Hence `chips` corresponds to *exactly* one serving of chips, ! `chips` corresponds to an arbitrary number of servings (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.

Linear logic also contains a negation, which behaves in a manner reminiscent of classical negation. The negation of a formula F is written as $F^\perp$. As there are two conjunctions, there are two corresponding disjunctions, as well as a dual to ! denoted as ?. The following laws, reminiscent of the de Morgan laws, all hold:

$$\begin{array}{l}(F_1 \otimes F_2)^\perp \equiv (F_1)^\perp \parr (F_2)^\perp \\ (F_1 \parr F_2)^\perp \equiv (F_1)^\perp \otimes (F_2)^\perp \\ (F_1 \oplus F_2)^\perp \equiv (F_1)^\perp \,\&\, (F_2)^\perp \\ (F_1 \,\&\, F_2)^\perp \equiv (F_1)^\perp \oplus (F_2)^\perp\end{array}$$

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 $\parr$ and $\oplus$ are written as $\perp$ and $\mathbf{0}$, and which may be thought of as generalisations of the boolean value false.

There is far more to linear logic than can be discussed in this paper; for a more complete introduction see the papers [7, 17, 15, 1], among others.

3 Specifying Bottom-up Computation

A bottom-up evaluation of a program involves producing a new program based on the old one. Hence, unlike the top-down case, the notion of a goal is of minimal interest per se; the main technical point is to determine how the new program is derived from the old. Hence we will denote by $P \leadsto P'$ the statement that P' can be derived in an appropriate manner from P. Naturally $\leadsto$ must respect soundness, i.e. that if $P \leadsto P'$ then $P \vdash \otimes P'$. The key point is then to find the appropriate inference system for $\leadsto$.

It should be noted that another significant difference from the top-down case is that whilst we will be a given a program P to evaluate, there will (generally) be no specification of P'. Hence the evaluation will be a forward-chaining one, and the rules for $\leadsto$ will be in the style of Plotkin's Structural Operational Semantics [13], in that given a rule such as

$$\frac{P \leadsto P''}{P \leadsto P'}\ R$$

[4] where P is known but P' is not, we will use the premise (and any appropriate sub-proofs) to evaluate P to P'', and then using the rule R we will then evaluate P to P'. Hence we will generally use the rules of the system below in such a way that P is known before the proof is attempted, and, if the proof is successful, P' is calculated in the course of the proof.

In the top-down case, the root of the proof is $\mathcal{P} \vdash \mathcal{G}$, where both $\mathcal{P}$ and $\mathcal{G}$ are known, and hence whatever calculations arise from the proof are in the form of witnesses for existentially quantified variables, or choices for disjunctive branches in the proof (and are hence dependent, at least partially, on $\mathcal{G}$). In the bottom-up case, the outcome of the computation is a multiset of formulae (P'), and this is dependent only on the original program P, and not on any other formulae. Hence the bottom-up approach encompasses a different computational paradigm that the top-down approach.

One aspect of this difference may be found in the way that "unused" resources are treated.[5] For example, we expect that $p, q, p \multimap r \leadsto q, r$; a top-down execution of the same program with

[4]Note that we assume, for simplicity, that R is a unary rule.

[5]Note that in linear logic, we do not have that $p, q \vdash p$, as the q has not been accounted for. We do, however, have that $p, q \vdash p \otimes q$.

the goal $q \otimes r$ would succeed, but both of the goals r and q would fail. Hence in the bottom-up case unused resources are not a source of failure, but are "added extras" in the outcome of the computation.

It should be noted that the above form of rules is similar to the notion of conditional rewriting systems, in that we may think of the above rule as stating that P can be re-written to P' under the condition that the premise of the rule holds.

Our technical aim, then, is to find the appropriate rules for $\rightsquigarrow$. Clearly we want the evaluation process to be complete, in that for some class of goal formulae, we have that there is a proof of a goal G from the original program just in the case that there is a proof of G from the evaluated program.

A natural starting point for this analysis is the rule of *modus ponens*. The linear form of this rule may be stated as $A, A \multimap B \vdash B$. An important point to note is that both A and $A \multimap B$ are consumed by this process, and hence the linear form has a different effect in the context of a proof than the corresponding rule in either classical or intuitionistic logic. In particular, whilst the rule looks the same, in the classical or intuitionistic case, the presence of the structural rules of weakening and contraction mean that we can actually use a stronger form of the rule, i.e. $A, A \rightarrow B \vdash A \wedge B$. Note that $A \wedge (A \rightarrow B) \equiv A \wedge B$ (in both classical and intuitionistic logics), and hence a bottom-up execution process based on this rule will preserve equivalence. This strong property ensures that such a process will never need to backtrack, as the preservation of equivalence ensures that anything provable from the original program is provable from the evaluated one. Hence, the T_P construction [5] and similar constructions which are used to give semantics to logic programs may be used without any consideration of either the order in which the rules of the program are used, or the completeness of the evaluation mechanism.

In the linear case, the equivalence property does not hold. For example, the program $p, p \multimap q$ will clearly evaluate to q, and whilst $p, p \multimap q \vdash q$, it is clearly not the case that $q \vdash p \otimes (p \multimap q)$. As a result, we need to consider how we may achieve an appropriate completeness property for the linear version.[6]

Note that the issue of completeness is complicated by the interaction between strictly linear formulae and !. For example, consider the program $p, !(p \multimap q)$. Note that $p, !(p \multimap q) \vdash p$ and $p, !(p \multimap q) \vdash q$ (but of course $p, !(p \multimap q) \not\vdash p \otimes q$). Hence we have that $p, !(p \multimap q) \vdash p \& q$, and so completeness requires that $p, !(p \multimap q) \rightsquigarrow p \& q$. Note that this is a way of integrating the choice of whether or not to make use of a particular rule with a means of making the rewriting system confluent.

Note also that we would want $!p, !(p \multimap q) \rightsquigarrow !q$, rather than the weaker form $!p, !(p \multimap q) \rightsquigarrow q$. One way to achieve this is to use the weaker form, but to have an explicit rule to achieve the stronger one where possible. We will return to this point below.

Hence the main computational metaphor will be that given a program P containing a clause $G \multimap D$, we wish to find P_1, P_2 such that $P = P_1 \cup P_2 \cup \{G \multimap D\}$ and $P_1 \vdash G$ so that we have $P_1, P_2, G \multimap D \rightsquigarrow P_2, D$. The main focus is then on the appropriate form of the implication rule. Note that we do not assume that D is atomic; as we shall see, this is a natural choice for bottom-up evaluation, as well as being very useful for applications.

It should be noted that the rules for $\rightsquigarrow$ are similar to the notion of conditional rewriting systems, in that we may think of the above rules as stating that P can be re-written to P' under the condition that the premises hold.

One key technical issue is how to determine whether $P_1 \vdash G$ or not. Clearly this will generally require right rules, such as in the program $p, q, (p \otimes q) \multimap r$, for which we require that $p, q, (p \otimes q) \multimap$

[6] As noted above, any reasonable evaluation should be sound, i.e. if $P \rightsquigarrow P'$, then $P \vdash \otimes P'$.

$r \rightsquigarrow r$. It is clear that in this case we have $P_2 = \emptyset$, $P_1 = \{p, q\}$, and it is straightforward to show that $p, q \vdash p \otimes q$. However, we would expect the same result from the program $D_1, D_2, G_1 \multimap p, G_2 \multimap q, (p \otimes q) \multimap r$ where $D_1 \vdash G_1$ and $D_2 \vdash G_2$. Note that in this case the proofs of $D_1 \vdash G_1$ and $D_2 \vdash G_2$ may be arbitrarily large.

The question is then how we write the implication rule. One possibility is the one below.

$$\frac{P_1 \vdash G}{P_1, P_2, G \multimap D \rightsquigarrow P_2, D}$$

However, this is not particularly satisfactory. This does not provide for any bottom-up evaluation of P_1, and whilst we wish to retain the possibilities for interaction between top-down and bottom-up computations, we also want to allow the possibility that our computations are as bottom-up as possible. Hence we propose the rule below:

$$\frac{P_1 \rightsquigarrow P' \quad P' \vdash G}{P_1, P_2, G \multimap D \rightsquigarrow P_2, D} \multimap$$

When compared to the rules of the linear sequent calculus, this rule corresponds to a particularly localised form of the $\multimap$L rule, which may be specified as below:

$$\frac{P_1 \vdash G, \Delta_1 \quad P_2, D \vdash \Delta_2}{P_1, P_2, G \multimap D \vdash \Delta_1, \Delta_2} \multimap L$$

Clearly our rule corresponds to the case in which the body of the clause can be evaluated without any external context (i.e. Δ_1). As we shall see, this will require some restrictions on the proofs (and hence classes of formulae) that we consider.

Note also the flexibility of the $\multimap$ rule when it comes to the balance between top-down and bottom-up evaluation. For example, we may choose to maximally evaluate P_1 before attempting to prove $P' \vdash G$, and hence make the proof of $P' \vdash G$ as simple as possible. On the other hand, we may choose not to evaluate P_1 at all, and hence choose $P_1 = P'$, and proceed in an entirely top-down manner. Naturally, we may choose a more intermediate course, as well.

Another possibility for the implication rule is given below.

$$\frac{P \rightsquigarrow P' \quad P_1 \vdash G \quad P_2, D \rightsquigarrow P''}{P_1, P_2, G \multimap D \rightsquigarrow P''} \multimap'$$

This version of the rule encodes the transitivity of $\rightsquigarrow$ into the rule, and as we shall see, $\multimap'$ is a derived rule of the system given below.

The rules for the other connectives which may appear in programs are generally straightforward copies of those for the left rules in the linear sequent calculus. The most interesting one of these is the rule for $\invamp$. The rule in the sequent calculus is

$$\frac{\Gamma_1, F_1 \vdash \Delta_1 \quad \Gamma_2, F_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, F_1 \invamp F_2 \vdash \Delta_1, \Delta_2}$$

This suggests the following rule for $\rightsquigarrow$.

$$\frac{P_1, D_1 \rightsquigarrow P_1' \quad P_2, D_2 \rightsquigarrow P_2'}{P_1, P_2, D_1 \invamp D_2 \rightsquigarrow (\otimes P_1') \invamp (\otimes P_2')} \invamp$$

Here we split the program, as indicated by the sequent calculus rule, evaluate each part separately, and then combine the results.

Another point to note is that we do not require that the heads of clauses (i.e. the D in $G \multimap D$) be atomic. In some top-down systems, this assumption is made in order to simplify the goal-reduction process, i.e. given a sequent $\mathcal{P} \vdash \mathcal{G}$, one wants to reduce $\mathcal{G}$ as much as possible, and hopefully to a multiset of atomic formulae, before applying left rules, including the resolution rule [14, 9]. There are some top-down systems in which the heads of clauses need not be atomic, such as Forum and LO, in which there may be occurrences of $⅋$, but this still corresponds to a simple method of determining the next goal. In the bottom-up case, the emphasis is more on finding connections within the program, and producing the appropriate results. This may involve the derivation of atomic formulae, but there is no obvious reason for restricting clauses to contain only atomic heads (or, for that matter, to any other subclass of definite formulae) . For example, in a genetic algorithm application, it is often desirable to replace a pair of chromosomes with a new pair; in this case, it is clearly more natural to express this change in the form $(A \otimes B) \multimap (C \otimes D)$ than in a fragment in which the right hand side of $\multimap$ is restricted to atoms.

We need also to consider some extra properties, and hence rules, for our system. For example, we would insist that the system be transitive, so that if $P_1 \leadsto P_2$ and $P_2 \leadsto P_3$, then $P_1 \leadsto P_3$. It may be possible to establish such a result by proving directly the appropriate properties of the rules, but it seems appropriate to add the necessary rule to the system, and then consider how it may be eliminated, in the manner of cut elimination [6]. In addition, we would expect that the system would allow weakening in some global sense, in that if $P_1 \leadsto P_2$, then $P_1, P_3 \leadsto P_2, P_3$. This suggests the following rules:

$$\frac{P_1 \leadsto P_2 \quad P_2 \leadsto P_3}{P_1 \leadsto P_3}\ Cut \qquad \frac{P_1 \leadsto P_2}{P_1, P_3 \leadsto P_2, P_3}\ Weak$$

Note that these rules ensure that $\leadsto$ is transitive and monotonic (essentially localising the evaluation of program clauses). Note also that Cut corresponds to the cut rule in the linear sequent calculus, and Weak to a particular form of weakening. As we shall see, these rules play an important role in the system.

There are two other rules of note. As noted above, we will require that $p, !(p \multimap q) \leadsto p \,\&\, q$. In order to achieve this, it seems reasonable to allow a "backtrackable" choice to be made at some point in the derivation, and then "collect" the different choices together when necessary. Hence if we find that $P \leadsto P_1$ and $P \leadsto P_2$, then we can conclude that $P \leadsto (\otimes P_1) \,\&\, (\otimes P_2)$. In effect this allows us to choose to execute a particular branch when desired, and to draw together various different branches as required. Another way to think of this rule is that if $P_1 \leadsto P_2$ and $P_3 \leadsto P_4$ then it seems reasonable that $(\otimes P_1) \,\&\, (\otimes P_2) \leadsto (\otimes P_3) \,\&\, (\otimes P_4)$. When $P_1 = P_2$, this is just $P_1 \leadsto (\otimes P_3) \,\&\, (\otimes P_4)$, as $(\otimes P_1) \,\&\, (\otimes P_1) \equiv (\otimes P_1)$. As noted above, it is this rule that addresses the problem of confluence in the presence of "don't know" non-determinism.

The other rule of note is the one which deals with !. As noted above, we wish to have a means of recovering the stronger conclusion where appropriate, such as determining that $!p, !(p \multimap q) \leadsto !q$ from $!p, !(p \multimap q) \leadsto q$. Hence we add an appropriate rule to the system, based on the corresponding rule in the sequent calculus, and address the eliminability of this rule below.

Hence the main technical points are concerned with implication, and "meta-properties", such as transitivity, confluence and the strength of the conclusions reached. The following section contains a formal definition of the inference system.

4 Inference Rules

Below we present the inference rules for the system.

Definition 1 *A* derivation tree *is proof tree governed by the following rules:*

$$\frac{}{P \rightsquigarrow P}\ \text{Axiom} \qquad \frac{P_1 \rightsquigarrow P' \quad P' \vdash G}{P_1, P_2, G \multimap D \rightsquigarrow P_2, D}\ \multimap$$

$$\frac{P, D_i \rightsquigarrow P'}{P, D_1 \,\&\, D_2 \rightsquigarrow P'}\ \& \qquad \frac{P_1, D_1 \rightsquigarrow P_1' \quad P_2, D_2 \rightsquigarrow P_2'}{P_1, P_2, D_1 \parr D_2 \rightsquigarrow (\otimes P_1') \parr (\otimes P_2')}\ \parr$$

$$\frac{P, D_1, D_2 \rightsquigarrow P'}{P, D_1 \otimes D_2 \rightsquigarrow P'}\ \otimes \qquad \frac{P, D \rightsquigarrow P'}{P, !D \rightsquigarrow P'}\ ! \qquad \frac{P, !D, !D \rightsquigarrow P'}{P, !D \rightsquigarrow P'}\ C!$$

$$\frac{P \rightsquigarrow P'}{P, \mathbf{1} \rightsquigarrow P'}\ \mathbf{1} \qquad \frac{P, D[t/x] \rightsquigarrow P'}{P, \forall x D \rightsquigarrow P'}\ \forall$$

$$\frac{P_1 \rightsquigarrow P_2 \quad P_2 \rightsquigarrow P_3}{P_1 \rightsquigarrow P_3}\ Cut \qquad \frac{P \rightsquigarrow P'}{P, Q \rightsquigarrow P', Q}\ Weak$$

$$\frac{P \rightsquigarrow P_1 \quad \ldots \quad P \rightsquigarrow P_n}{P \rightsquigarrow (\otimes P_1) \,\&\, \ldots \,\&\, (\otimes P_n)}\ collect \qquad \frac{!P \rightsquigarrow P'}{!P \rightsquigarrow !P'}\ !M$$

We will often use $P \rightsquigarrow P'$ as a shorthand for the statement that $P \rightsquigarrow P'$ has a derivation tree.

Note that for the right-hand premise of the $\multimap$ rule, we assume that the standard rules of the linear sequent calculus are used.

Note also that the following rule for implication is equivalent to the one above in the presence of Weak:

$$\frac{P_1 \rightsquigarrow P' \quad P' \vdash G}{P_1, G \multimap D \rightsquigarrow D}$$

Whilst this may appear to be a simpler form of the rule, we use the above form for ease of comparison with the corresponding rule in the sequent calculus.

It is interesting to note that the $\multimap$ rule is akin to a directed cut, in that P' does not appear anywhere in the conclusion of the rule, but that there is a computational means of deriving P' from P. Hence P' corresponds to an interpolant formula for P and G [4].

Note also that as written, the rules for $\rightsquigarrow$ do not necessarily specify a bottom-up computation; the generality of the $\multimap$ rule means that the left-hand premise could be trivial. The bottom-up interpretation comes from a derivation tree together with a particular execution strategy, such as requiring that the $P \rightsquigarrow P'$ premise of $\multimap$ be maximally evaluated. Such a strategy can be specified by a syntactic rule such as P' being required to be a multiset of formulae of a particular kind, or by requiring that the proof of $P \vdash G$ have a particular property, such as bounding the number of right rules that can be used by the number of connectives in G.

Another consideration (foreshadowed above) is that the implication rule seems to require some form of restriction. In particular, it seems necessary to insist that in the $\multimap$L rule of the linear sequent calculus, i.e.

$$\frac{\Gamma_1 \vdash F_1, \Delta_1 \quad \Gamma_2, F_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, F_1 \multimap F_2 \vdash \Delta_1, \Delta_2}$$

we have that $\Delta_1 = \emptyset$. This means that we can look entirely within the program to satisfy F_1, rather than having some external context (Δ_1) make a contribution. The precise properties of this rule are then clearly of interest. Hence we come to the definition below.

Definition 2 *Let Θ be a proof in the linear sequent calculus. We say Θ is* $\multimap$-localised *if every occurrence of the rule $\multimap L$ in Θ is of the form*

$$\frac{\Gamma_1 \vdash F_1 \quad \Gamma_2, F_2 \vdash \Delta}{\Gamma_1, \Gamma_2, F_1 \multimap F_2 \vdash \Delta}$$

Our particular interest is to determine an appropriate class of formulae for which $\multimap$-localised proofs are complete (i.e. a proof exists iff a $\multimap$-localised proof exists). One way in which to obtain such a result is to restrict our attention to intuitionistic linear logic, i.e. the fragment in which every succedent contains at most one formula. However, this seems too drastic, especially as the restriction only applies to one rule, rather than to all rules. In addition, it is possible to re-write certain proofs which contain multiple formulae in succedents into $\multimap$-localised proofs. Consider the sequent $p \parr q \parr r, r \multimap s \vdash s, p, q$ which has a proof as below:

$$\dfrac{\dfrac{\dfrac{p \vdash p \quad q \vdash q}{p \parr q \vdash p, q}\parr L \qquad r \vdash r}{p \parr q \parr r \vdash r, p, q}\parr L \qquad s \vdash s}{p \parr q \parr r, r \multimap s \vdash s, p, q}\multimap L$$

Note that the occurrence of $\multimap$L is not $\multimap$-localised, as $\Delta_1 = \{p, q\}$, $\Delta_2 = \{s\}$. However, note that there is an $\multimap$-localised proof of this sequent, given below:

$$\dfrac{\dfrac{p \vdash p \quad q \vdash q}{p \parr q \vdash p, q}\parr L \qquad \dfrac{r \vdash r \quad s \vdash s}{r, r \multimap s \vdash s}\multimap L}{p \parr q \parr r, r \multimap s \vdash p, q, s}\parr L$$

Note also that we have $p \parr q \parr r, r \multimap s \leadsto p \parr q \parr s$, as below:

$$\dfrac{p \parr q \leadsto p \parr q \qquad \dfrac{r \leadsto r \quad r \vdash r}{r, r \multimap s \leadsto s}\multimap}{p \parr q \parr r, r \multimap s \leadsto p \parr q \parr s}\parr$$

Hence it is possible to show by a permutation argument (sketched below) that $\multimap$-localised proofs are complete for goal formulae in which formulae of the form $?G$ and $\bot$ are absent. The reason for this restriction is that if we allow the corresponding rules of the sequent calculus to arbitrarily add formulae to the succedent, then we cannot be guaranteed to rearrange the left-hand premise of the $\multimap$L rule to be a singleton multiset. Consider for example the following proof:

$$\dfrac{\dfrac{\dfrac{q \vdash q}{q \vdash ?p, q}W?R}{q \vdash ?p \oplus r, q}\oplus R \qquad s \vdash s}{q, (?p \oplus r) \multimap s \vdash s, q}\multimap L$$

It should be clear that there can be no $\multimap$-localised proof of this sequent. The problem is that $?p$ is needed for the construction of the formula in the endsequent, and that in order for this to happen, we require the presence of q in the succedent of the left premise of $\multimap$L. Note that when considering proof-search which begins at the root of the proof, the W?R and $\bot$R rules may be thought of as reducing the number of formulae in the succedent. However, if we do not allow such rules, then the only way in which a succedent which contains more than one formula can be reduced to the succedent of an axiom (which must be a singleton) is for each occurrence of $\parr$R to be matched by an occurrence of $\parr$L which reduces the size of the succedent.

To see an example of this, consider the provable sequent $p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s \parr t$ which has the following proof (which is not $\multimap$-localised):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{q \vdash q \quad r \vdash r}{r, q \vdash q \otimes r}\otimes R \qquad t \vdash t}{r, t \parr q \vdash q \otimes r, t}\parr L \qquad p \vdash p}{p \parr r, t \parr q \vdash p, q \otimes r, t}\parr L}{p \parr r, t \parr q \vdash p \parr (q \otimes r), t}\parr R \qquad s \vdash s}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s, t}\multimap L}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s \parr t}\parr R$$

Note that the upper occurrence of $\parr$L can be permuted downwards past the $\multimap$L rule, whereas the lower occurrence of $\parr$L cannot be permuted past the occurrence of $\parr$R, as the formulae p and $q \otimes r$ are on different branches of the occurrence of $\parr$L. Hence we can rearrange this proof into the one below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p \vdash p \qquad \dfrac{r \vdash r \quad q \vdash q}{r, q \vdash q \otimes r}\otimes R}{p \parr r, q \vdash p, q \otimes r}\parr L}{p \parr r, q \vdash p \parr (q \otimes r)}\parr R \qquad s \vdash s}{p \parr r, q, (p \parr (q \otimes r)) \multimap s \vdash s}\multimap L \qquad t \vdash t}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s, t}\parr L}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s \parr t}\parr R$$

Note that this sequent has the following derivation in our system:

$$\dfrac{\dfrac{\dfrac{p \leadsto p \quad q, r \leadsto q, r}{p \parr r, q \leadsto p \parr (q \otimes r)}\parr \qquad p \parr (q \otimes r) \vdash p \parr (q \otimes r)}{p \parr r, q, (p \parr (q \otimes r)) \multimap s \leadsto s}\multimap \qquad t \leadsto t}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \leadsto s \parr t}\parr$$

A formal statement of the correctness of this process is in Section 6.

5 Transitivity Rules

As mentioned above, we want $\leadsto$ to be transitive and monotonic. However, it is not clear that Cut and Weak are the only rules which will serve this purpose (although it should be clear that they are reasonable choices). In addition, we would also expect some slightly stronger properties of $\leadsto$ to hold, in that we would expect both of the following rules to be admissible:

$$\frac{P_1 \leadsto P_1' \quad P_2 \leadsto P_2'}{P_1, P_2 \leadsto P_1', P_2'}\ \text{Mix} \qquad \frac{P_1 \leadsto P_2 \quad P_2, P \leadsto P_3}{P_1, P \leadsto P_3}\ \text{ICut}$$

Note that there is a counterpart to Mix in the classical sequent calculus, which is often used to simplify proofs, such as cut elimination. Note also that ICut is similar to the version of the cut rule used in the intuitionistic sequent calculus LJ.

It is easy to see that Cut is an instance of ICut, and that Weak is an instance of Mix (as $P \leadsto P$). Hence it is clear that the combination of Mix and ICut is at least as powerful as that of Cut and Weak. Interestingly, the converse is also the case, in that Mix and ICut are derived rules in the presence of Cut and Weak as below:

$$\frac{P_1 \leadsto P_1' \quad P_2 \leadsto P_2'}{P_1, P_2 \leadsto P_1', P_2'}\ \text{Mix} \qquad \frac{\dfrac{P_1 \leadsto P_1'}{P_1, P_2 \leadsto P_1', P_2}\ Weak \quad \dfrac{P_2 \leadsto P_2'}{P_1', P_2 \leadsto P_1', P_2'}\ Weak}{P_1, P_2 \leadsto P_1', P_2'}\ Cut$$

$$\frac{P_1 \leadsto P_2 \quad P_2, P_3 \leadsto P_4}{P_1, P_3 \leadsto P_4}\ \text{ICut} \qquad \frac{\dfrac{P_1 \leadsto P_2}{P_1, P_3 \leadsto P_2, P_3}\ Weak \quad P_2, P_3 \leadsto P_4}{P_1, P_3 \leadsto P_4}\ Cut$$

Hence we get the result below.

Proposition 1 *Cut + Weak $\equiv$ Mix + ICut*

Hence the combination of Cut and Weak seems to be a good way to get simple, elegant rules with the appropriate power. We would want both of these to be ultimately eliminable, however, in order to simplify the process of finding proofs. However, note that due to the forward-chaining (or "rewriting") nature of the rules, any lack of eliminability is not as drastic as it may seem. For example, with the Cut rule, given P_1, it is possible to calculate P_2 via the left premise, and once P_2 is known, it is possible to calculate P_3 from the right premise. Hence, whilst eliminability is desirable, a lack of it does not necessarily mean that the system is inherently intractable.

6 Results

In this section we give the formal results about $\leadsto$.

In what follows, we assume that implications are not allowed in goals. This is for reasons of simplicity. For example, if at some stage in the derivation there was an occurrence of the implication rule of the form

$$\frac{P_1 \leadsto P' \quad P' \vdash D' \multimap G}{P_1, (D' \multimap G) \multimap D \leadsto D}$$

then the proof of $P' \vdash \multimap G$ could conclude with $\multimap$R [7], making the premise $P', D \vdash G$. Clearly this sequent is not problematic per se, but in a completeness proof based on an induction on the size of the proof, this means that we can "push" the bottom-up behaviour into the right-hand premise of the $\multimap$ rule as well, in that as the proof of $P', D \vdash G$ is necessarily shorter than that of the endsequent, we can then deduce that $\exists P''$ such that $P', D \leadsto P''$ and $P'' \vdash G$. Note that whatever the hypothesis implies about $P', D \vdash G$ will also be true for $P'' \vdash G$, and so we can recursively apply the hypothesis all the way up the proof tree. This suggests the addition of the following rule to the inference system:

$$\frac{P \leadsto P' \quad P' \vdash \mathcal{G}}{P \vdash \mathcal{G}}$$

[7]In fact, as the $\multimap$R rule is asynchronous (in the terminology of [2]) then the proof may be assumed to be of this form.

Note that in this form, this rule is precisely a directed cut. This may be thought of as a "top-down" conclusion from the premises $P \leadsto P'$ and $P' \vdash \mathcal{G}$, whereas the implication rule gives $P, G \multimap D \leadsto D$, which may be considered the corresponding "bottom-up" conclusion. However, a full analysis and discussion of this point would take us beyond the scope of this paper, and so for now we omit the possibility that goals may contain implications to simplify the discussion.

Hence we arrive at the class of formulae below.

Definition 3 *We define the following classes of formulae:*

Goal *formulae:*	G	$::=$	$A \mid G \otimes G \mid G \invamp G \mid G \mathbin{\&} G \mid G \oplus G \mid \mathbf{1} \mid \bot \mid \forall x G \mid \exists x G \mid !G \mid ?G$
Strong goal *formulae:*	G	$::=$	$A \mid G \otimes G \mid G \invamp G \mid G \mathbin{\&} G \mid G \oplus G \mid \mathbf{1} \mid \forall x G \mid \exists x G \mid !G$
Definite *formulae:*	D	$::=$	$A \mid D \otimes D \mid D \invamp D \mid D \mathbin{\&} D \mid \mathbf{1} \mid \bot \mid \forall x D \mid !D \mid G \multimap D$

where G is a goal formula.

Strong definite *formulae are definite formulae in which all formulae of the form $G \multimap D$ are such that G is a strong goal formula.*

In all cases above, A is an atomic formula.

Definition 4 *We define the* depth *of a formula F as follows:*

depth(A) =   *0 where A is atomic*
depth($\clubsuit F$) =   *1 + depth(F) where $\clubsuit$ is a unary connective*
depth($F_1 \heartsuit F_2$) =   *1 + max(depth(F_1), depth(F_2)) where $\heartsuit$ is a binary connective*
We define depth($\mathcal{F}$) = $\Sigma_{F \in \mathcal{F}} \mathrm{depth}(F)$.

Proposition 2 (Soundness) *Let P be a multiset of definite formulae. Then if $P \leadsto P'$ then $P \vdash \otimes P'$.*

Corollary 1 *Let P be a multiset of definite formulae and let $\mathcal{G}$ be a multiset of goal formulae. Then if $P \leadsto P'$ and $P' \vdash \mathcal{G}$ then $P \vdash \mathcal{G}$.*

Proof: By the above proposition, $P \vdash \otimes P'$, and as $\otimes P' \vdash \mathcal{G}$, we have that $P \vdash \mathcal{G}$. □

The statement of a completeness result is more subtle. It is tempting to state the completeness of $\leadsto$ as follows: if $P \vdash \mathcal{G}$ then $\exists P'$ such that $P \leadsto P'$ and $P' \vdash G$. However, this is trivially true, as $P \leadsto P$. Hence the statement of completeness need to be somewhat more careful, and will depend upon the precise execution properties desired. Below we present one version of completeness which seems appropriate for a "maximally bottom-up" computation.

Proposition 3 (Definite Completeness) *Let P be a multiset of strong definite formulae and $\mathcal{G}$ be a multiset of strong goal formulae. If $P \vdash \mathcal{G}$ has an $\multimap$-localised proof, then there is a multiset P' of D formulae such that $P \leadsto P'$ and $P' \vdash \mathcal{G}$ where the number of right rules in the proof of $P' \vdash \mathcal{G}$ is not more than depth($\mathcal{G}$).*

It is clear that the amount of mandatory bottom-up evaluation is proportional to the degree of freedom allowed in the proof of $P' \vdash \mathcal{G}$; the greater the freedom, the more scope there is for top-down evaluation. Hence strategies which are more intermediate than the strict bottom-up approach will require an appropriate statement of the requirements for this proof.

Note that the restriction to strong definite formula is necessary, as the presence of contraction on the right may violate the property about right rules and the depth of the succedent. Whilst this may seem a little strong, the proposition below introduces another reason for restricting our attention to this class of formula, as we require that weakening not be present in order to show the completeness of $\multimap$-localised proofs.

Proposition 4 *Let P be a multiset of strong definite formulae and $\mathcal{G}$ be a multiset of strong goal formulae. Then $P \vdash \mathcal{G}$ is provable iff $P \vdash \mathcal{G}$ has an $\multimap$-localised proof.*

Hence we have that $\leadsto$ is complete for strong definite formulae. As noted above, there may be other forms of completeness which are of interest, but the one given above seems appropriate for bottom-up execution.

Note that in order to determine search strategies for our inference rules, we consider their permutation properties, i.e. the ability to interchange adjacent rules without altering the overall proof. Many of these follow immediately from the properties of the corresponding rules in the linear sequent calculus [11]. Our interest is in the properties of the rules peculiar to our system, and in particular the consequences of the permutation properties for the eliminability of Cut and Weak (and for that matter, collect and !M) and the possibilities for normal forms for proofs.

A full analysis is beyond the scope of this paper, but we briefly discuss these below. Our Cut rule generally cannot be permuted upwards, but can always be permuted downwards, expect past other occurrences of Cut. Hence the Cut rule is not eliminable, but we can limit its occurrences to being closer to the endsequent than any other rule. As mentioned above, this does not mean that our rules are inherently intractable, as it is possible to use P to calculate P'. In fact, this may be considered as an appropriate proof-theoretic characterization of the transitivity of $\leadsto$, in that a derivation of $P_1 \leadsto P_2$ can always be "supplemented" by the derivation of $P_2 \leadsto P_3$ and hence $P_1 \leadsto P_3$, no matter how long or complicated the derivation of $P_1 \leadsto P_2$ may be.

The Weak rule also cannot be generally eliminated. It is eliminable at the axioms, and permutes upwards past all rules except $\otimes$, collect and !M. Hence this suggests that the three rules $\otimes$, collect and !M should be modified to incorporate the Weak rule. In such a modified system, the Weak rule is eliminable (note also that our form of the $\multimap$ rule effectively already has Weak encoded into it). Note also that Weak is a more general rule than the corresponding one in the linear sequent calculus, as the Weak rule does not place any restrictions on the formulae which may occur in the rule. In this sense, and particularly given the impermutabilities of the Weak rule, our inference system has some characteristics of *affine logic*, i.e. linear logic with arbitrary weakening rules added. It is interesting to note that in affine logic, the weakening rules cannot always be permuted upwards, akin to the lack of upward permutability of Weak past !M. In a computational sense, this indicates when it is necessary to "localise" the current search.

The collect rule has the interesting property of permuting downwards past all rules except Weak, and on the right-hand side of Cut. Hence in a system in which Weak is eliminable, collect can always be permuted downwards, except past certain occurrences of Cut. This suggests that a normal form would position the occurrences of collect immediately above the occurrences of Cut.

It should also be noted that the standard problems of contraction apply here, and hence so can the standard solutions, such as requiring proofs not to contain occurrences of weakening immediately above contraction where the two rules do not occur in permutation position, i.e. that there is a formula introduced by the weakening rule which is eliminated by the contraction.

We also note that the permutation behaviour of Mix and ICut is significantly more restrictive than that of Cut and Weak, which acts as a post-fact justification of our choice of the latter two rules.

Using these observations, it is not hard to come up with an equivalent "normalized" system, in which the above suggestions are incorporated, and in which:

- there are no occurrences of Weak.
- for each occurrence of Cut, the only rules between the Cut and the endsequent are other occurrences of Cut.
- for each occurrence of collect, the only rules between the collect and the endsequent are other occurrences of collect or Cut.

7 Computational Issues

As noted above, the normal form for proofs consists of an arbitrary number of occurrences of the Cut rule at the root of the proof. From a computational point of view, this means that we may arbitrarily extend any derivation of $P_1 \leadsto P_2$ to $P_1 \leadsto P_3$ provided that we can show that $P_2 \leadsto P_3$, and so that computation can continue whenever the appropriate reductions can be found. This indefiniteness of termination is clearly appropriate for a forward-chaining application such as a genetic algorithm, in which it is desirable to continue computation until a certain threshold is reached.

The properties of the collect rule pose some interesting implementation issues. Clearly if it is desired to determine all possible consequences of a particular set of circumstances (such as finding all possible states which would result from a particular setting of valves in a power plant), then it will be necessary to have many occurrences of the collect rule, and hence a potentially large number of different computational branches. On the other hand, if the aim of the computation is to find some appropriate final state (as distinct from all such possible states), such as an appropriate serialisation of a set of banking transactions, then it may not be appropriate to use the collect rule at all, as the entire set of possibilities does not need to be examined. Hence it would seem natural to allow the user to explore the different branches lazily (i.e. each branch is generated on demand) rather than eagerly (i.e. all branches are evaluated in advance).

The issues concerning the other rules are fairly standard for the implementation of linear logic programming languages, and hence will not be discussed here. One exception to this is the !M rule, which, in this case, will function as a trigger of its own: whenever the formulae on the left of $\leadsto$ contain only formulae of the form $!D$, then "promote" the current outcome. As computation will proceed by having P specified in advance, it is straightforward to determine when this promotion should occur. For example, given the program $!D, !(G' \multimap D')$, it should be clear that for any P' such that $!D, !(G' \multimap D') \leadsto P'$, then $!D, !(G' \multimap D') \leadsto !P'$. Hence the !M rule could be implemented as a flag of some sort, rather than as an explicit search rule per se.

There are some remaining sources of non-determinism, including the choice of the formula to be used in rules such as $\multimap$. An interesting observation that can be made is that there are some cases in which a "breadth-first" approach would be desired. For example, in a genetic algorithm, the use of aggregate functions (such as sum, count, maximum, minimum, etc.) implies that the use of atoms must be breadth-first within a predicate; for example, the fitness of the overall is generally required, and may be used as an input to the next step in the algorithm. Clearly then the data has to be processed in a breadth-first manner to achieve this. On the other hand, in an application such as an active database, a depth-first approach may be more appropriate, as it is generally important to follow through on a particular course of action rather than allowing alternatives to be pursued.

8 Acknowledgements

The authors are indebted to David Pym for many stimulating discussions about this paper.

References

[1] V. Alexiev. Applications of Linear Logic to Computation: An Overview. *Bulletin of the IGPL* 2(1):77-107, 1994.

[2] J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *J. Logic Computat.* 2(3), 1992.

[3] 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.

[4] P. Andrews. *An Introduction to Mathematical Logic and Type Theory.* Academic Press, 1986.

[5] M.H. van Emden and R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *Journal of the Association for Computing Machinery* 23:4:733-742, October, 1976.

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

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

[8] J. Harland, D. Pym, and M. Winikoff. Programming in Lygon: An Overview. Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology 391-405, Munich, July, 1996.

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

[11] P.D. Lincoln. *Computational Aspects of Linear Logic.* PhD thesis, Stanford University, August 1992.

[12] D. Miller. A multiple-conclusion meta-logic. Proc. of the IEEE Symposium on *Logic in Computer Science* 272–281, Paris, June, 1994.

[13] G. Plotkin. Structural Operational Semantics (lecture notes). Technical Report DAIMI FN-19, Aarhus University, 1981 (reprinted 1991).

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

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

[16] D. Smith. Mixlog: A Generalised Rule-based Language. Manuscript, 1997.

[17] A. Troelstra. Lectures on Linear Logic. CSLI Lecture Notes No. 29, 1992.

[18] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask and J. Harland. The Aditi Deductive Database System. *VLDB Journal* 3:2:245-288, April, 1994.

[19] P. Volpe. Concurrent Logic Programming as Uniform Linear Proofs. In G. Levi and M. Rodríguez-Artalejo (eds.), *Algebraic and Logic Programming*, 133–149. Springer, 1994.