Some Applications of 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.edu.au

Abstract

We describe and discuss the applications of a 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 various applications of this language, which include graph problems, and problems involving states and actions. In logic programming languages based on classical logic, it is possible to write elegant solutions for problems over acyclic graphs. By making use of properties of linear logic, it is possible to write similarly elegant solutions for cyclic graphs as well. As linear logic enables changes of state to be neatly expressed, it is straightforward to give Lygon solutions to problems such as the Yale shooting problem and the blocks world, and we give example solutions to each.*

Keywords Linear Logic, Logic Programming, Graph Problems, Applications, Actions and State.

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 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 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 [8], ACL [10], Forum [12] and Lygon [6, 7, 19, 20]. In this paper we describe some novel and interesting applications of Lygon, with particular reference to graph problems, and problems which involve reasoning about actions. Such applications make particular use of Lygon's basis in linear logic, which allows the natural specification of certain constraints, the constraint that each edge in a graph should appear at most once in a cycle, or that an action such as shooting a loaded gun causes the gun to become unloaded.

The basic features and implementation techniques used in Lygon have been explained in some detail in [19, 20]; in particular, the technique of *lazy splitting* is used to determine the allocation of resources to particular branches of the proof. Lygon may also be thought of as an extended version of Prolog, in which the default behaviour is that each formula must be used exactly once, and hence the execution model has to enforce this behaviour. This is what gives Lygon its particular expressive power, as the operational model has to maintain an internal notion of state in order to determine which formulae have been used so

[1] A full introduction to linear logic is beyond the scope of this paper — see [4, 16], amongst others, for more details.

Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 31–February 2 1996.

far, and which have not. This internal notion can be thus used to keep track of which edges in a graph have not been used yet, or whether a gun is currently loaded or unloaded.

The default behaviour can be overridden by the use of the special operators ! and ?, which, essentially, specify that formulae may be used any number of times, including zero, and hence the behaviour of (pure) Prolog can be recovered in Lygon.

This paper is organized as follows: in Section 2 we give a brief introduction to linear logic, and in Section 3 we briefly discuss its application to logic programming. In Section 4 give a brief overview of the features of Lygon, and in Section 5 we describe several applications of Lygon to graph problems. In Section 6 we discuss some further applications to problems involving actions and state, such as the Yale shooting problem. In Section 7 we look at some desirable programming constructs for Lygon and finally, in Section 8 we discuss our conclusions and 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.

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 be 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

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

From these rules and the information `dollar` $\otimes$ `dollar` we can then derive information such as

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

$\texttt{drink} \otimes \texttt{drink}$, 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 $\texttt{dollar} \otimes \texttt{sweet}$ 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 $\texttt{drink} \& \texttt{sweet}$ 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:

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

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:

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

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 $⅋$, 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, 16], 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 [13]. An analysis along these lines for linear logic was given in [6]. This leads to the following class of formulae forming a logic programming language:

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

where A ranges over atomic formulae.

In [6] 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 $p, q \vdash p \Rightarrow q \vdash \bot$ and $p, q \vdash q \Rightarrow p \vdash \bot$

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; for more detail see [20].

4 The Lygon Language

The current implementation of Lygon (version 0.4) is an interpreter written in BinProlog and is available by email request from the authors or from the URL `http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html`

The interpreter comprises about 500 lines of code (including comments and whitespace) for the version without the debugger and around 900 lines for the version with the debugger.

The class of formulae permitted is a limited form of the full Lygon language presented in the previous section. Goals are mostly unlimited, however programs are limited to clauses. The language accepted by the current implementation can be summarised by the following BNF:

$$
\begin{array}{lcl}
G & ::= & G \mathbin{\texttt{*}} G \mid G \mathbin{\texttt{@}} G \mid G \mathbin{\texttt{\&}} G \mid G \mathbin{\texttt{\#}} G \mid\ \texttt{!}\ G \\
 & & \mid \texttt{?}\ \texttt{neg}\ D \mid \texttt{one} \mid \texttt{bot} \mid \texttt{top} \mid A \mid \texttt{neg}\ A \\
D & ::= & [\texttt{!}]\ (A \mathbin{\texttt{<-}} G)
\end{array}
$$

Where the mapping between logical connectives and ASCII is:

$\otimes$	$\Rightarrow$	`*`	$\&$	$\Rightarrow$	`&`	$\multimap$	$\Rightarrow$	`->`
$\oplus$	$\Rightarrow$	`@`	$\parr$	$\Rightarrow$	`#`	$-^\bot$	$\Rightarrow$	`neg` $-$
$\mathbf{1}$	$\Rightarrow$	`one`	$\top$	$\Rightarrow$	`top`	$\bot$	$\Rightarrow$	`bot`

For example the toggle program in Lygon is

```
toggle <- (on * neg off) @ (off * neg on).
```

This toggles the state of a switch from on to off, and vice-versa, being equivalent to the linear formula

$$((\texttt{on} \otimes \texttt{off}^\bot) \oplus (\texttt{off} \otimes \texttt{on}^\bot)) \multimap \texttt{toggle}$$

This program is due to Hodas and Miller [8]. Operationally, given the state `off` and the goal `toggle`, the execution of the goal replaces the state `off` with the state `on` (and vice-versa).

Variables are denoted as in Prolog, using atoms beginning with an uppercase letter. The syntax for atoms is also inherited from BinProlog.

5 Graph Problems and Lygon

Graphs are an important data structure in computer science, and there are many applications of graph problems, such as laying cable networks, evaluating dependencies, designing circuits and optimisation problems. In this section we look at a number of graph problems, and show how they may be solved simply and elegantly in Lygon. In particular, the ability of Lygon to naturally state and satisfy constraints, such as that every edge in a graph can be used at most once, means that the solution to these problems in Lygon is generally simpler than in a language such as Prolog. The solutions presented are, we feel, concise and lucid.

The problems we examine are path finding and variants including Hamiltonian cycles and the related traveling salesman problem, and topological sorting (which has applications in job scheduling and is used for resolving dependencies in (for example) Makefiles)

One of the simplest problems involving graphs is finding paths. The standard Prolog program for path finding is the following one, which simply and naturally expresses that the predicate `path` is the transitive closure of the predicate `edge`, in a graph.

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Whilst this is a simple and elegant program, there are some problems with it. For example, the order of the predicates in the recursive rule is important, as due to Prolog's computation rule, if the predicates are in the reverse order, then goals such as `path(a,Y)` will loop forever. This problem can be avoided by using a memoing system such as XSB [18], or a bottom-up system such as Aditi [17]. However, it is more common to re-write the program above so that the path found is returned as part of the answer. In such cases, system such as XSB and Aditi will only work for graphs which are acyclic. For example, consider the program below.

```
path(X,Y,[X,Y]) :- edge(X,Y).
path(X,Y,[X|Path]) :-
       edge(X,Z), path(Z,Y,Path).
```

If there are cycles in the graph, then Prolog, XSB and Aditi will all generate an infinite number of paths, many of which will traverse the cycle in the graph more than once.

The main problem is that edges in the graph can be used an arbitrary number of times, and hence we cannot "mark" an edge as used, which is what is done in many imperative solutions to graph problems. However, in a linear logic programming language such as Lygon, we can easily constrain each edge to be used at most once on any path, and hence eliminate the problem with cycles causing an infinite number of paths to be found.

The code is simple; the main change to the above code is to load a "linear" copy of the `edge` predicate, and use the code as above, but translated into Lygon. Most of this is mere transliteration, and is given below.

```
graph <- neg edge(a,b) # neg edge(b,c) #
         neg edge(c,d) # neg edge(d,a).

trip(X,Y,[X,Y]) <- edge(X,Y).
trip(X,Y,[X|P]) <- edge(X,Z) * trip(Z,Y,P).

path(X,Y,P) <- top * trip(X,Y,P).
```

The extra predicate `trip` is introduced so that not every path need use every edge in the graph. As written above, `trip` will only find paths which use every edge in the graph (and so `trip` may be used directly to find Eulerian circuits, i.e. circuits which use every edge in the graph exactly once). However, the `path` predicate may ignore certain edges, provided that it does not visit any edge more than once.

It should be noted that this way of writing the `trip` is essentially using *affine* logic, which is similar to linear logic, except that each formula can be used *at most* once.

The goal `graph` is used to load the linear copy of the graph, and as this is a non-linear rule, we may load as many copies of the graph as we like; the important feature is that within each graph no edge can be used twice. We can then find all paths, cyclic or otherwise, starting at node `a` in the graph with the goal `graph # path(a,Y,P).` which yields the solutions below.

```
P = [a,b,c,d,a]
P = [a,b,c,d]
P = [a,b,c]
P = [a,b]
```

We can also find all cycles in the graph with a query such as `graph # path(X,X,P).` which yields the solutions

```
P = [c,d,a,b,c]
P = [d,a,b,c,d]
P = [b,c,d,a,b]
P = [a,b,c,d,a]
```

Note that we are not restricted to only one copy of the graph; if we wanted to do some further graph processing we could use a goal such as `(graph # (q1 * top)) & (graph # (q2 * top))`, as this provides one copy of the graph to the subgoal `q1`, and a separate, independent copy to the other subgoal `q2`.

This example suggests that Lygon is an appropriate vehicle for finding "interesting" cycles, such as Hamiltonian cycles (a cycle which visits every node in the graph exactly once). We can write such a program in a "generate and test" manner by using the `path` predicate above, and writing a test to see if the cycle is Hamiltonian. The key point to note is that we may delete any edge from a Hamiltonian cycle and we are left with an acyclic path which includes every node in the graph exactly once. Assuming that the cycle is represented as a list, then the test routine will only need to check that the "tail" of the list of nodes in the cycle (i.e. the returned list minus the node at the head of the list) is a permutation of the list of nodes in the graph. Hodas and Miller [8] have shown that such permutation problems may be simply solved in linear logic programming languages by "asserting" each element of each list into an appropriately named predicate (such as `list1` and `list2`), and testing that `list1` and `list2` have exactly the same solutions. The code to do this assertion is below; note that this may be thought of as a data conversion, from a list format into a predicate format.

```
perm([X|L],L2) <- neg list1(X) # perm(L,L2).
perm([], L2) <- perm1(L2).

perm1([X|l]) <- neg list2(X) # perm1(L).
perm1([]) <- check.
```

Once both lists are loaded, we call the predicate `check`, which will ensure that the two lists are permutations of each other. This predicate is particularly simple to write, and is given below. Note that if the input is given as predicates rather than lists, then this is all the code that we would need to write:

```
check <- list1(X) * list2(X) * check.
check.
```

Note that the second clause will only succeed if there are no linear resources, and hence this predicate is deterministic (we could also have written `check <- one`). The first clause will succeed provided that there is a common solution to both `list1` and `list2`; if this is not the case, then the only way to succeed is for both predicates to be exhausted. Hence, the only way for the test to succeed is if the two arguments to `perm` contain the same elements, and, moreover, the same number of times, but in a possibly different order, i.e. a permutation.

Whilst this is hardly a vast improvement on the standard Prolog predicate for permutations, it indicates the way in which more sophisticated routines may be written, such as sorting, or checking that one list is a sublist of the other, and it is particularly simple piece of code.

An interesting point raised by this example is that the representation of the data as resources, rather than in the traditional list, is potentially more efficient. The most common data structure used in Prolog programs is a list (and by a considerable margin, it would seem), and it seems that Prolog programmers have a tendency to overuse lists. It is not uncommon to find that some general collection of objects has been implemented as a list, even when the order in which items are represented is not important. Lists have the property that access to the nth element takes n operations. However, if the same collection is represented as a multiset of formulae, then there is the possibility for more flexible access, as there is no order of access imposed by the framework. Furthermore, by careful utilization of indexing techniques, it would seem that it is possible to provide constant or near-constant time access to an arbitrary element of the collection, and so Lygon has the potential for a more efficient representation of collections where the sequence of elements is not important.

The only remaining task to complete the code for the Hamiltonian cycle program is to write the code to extract the list of nodes in the graph from its initial representation. As this code is not particularly insightful, we omit it for the sake of brevity and assume that the nodes of the graph are given.

The Lygon program for finding Hamiltonian cycles in its entirety is given below:

```
go(P) <- graph # (top * (nodes # hamilton(P))).

graph <- neg edge(a,b) # neg edge(b,c)
       # neg edge(c,d) # neg edge(d,a).
nodes <- neg node(a) # neg node(b)
       # neg node(c) # neg node(d).

trip(X,Y,[X,Y]) <- edge(X,Y).
trip(X,Y,[X|P]) <- edge(X,Z) * trip(Z,Y,P).

all_nodes([]).
all_nodes([Node|Rest]) <- node(Node) *
                          all_nodes(Rest).

hamilton(Path) <- trip(X,X,Path) *
           eq(Path,[_|P]) * all_nodes(P).

eq(X,X).
```

The role of the `top` in `go` is to make the `edge` predicate affine (i.e. not every edge need be used). Given the query `go(P)` the program behaves as follows:

```
P = [c,d,a,b,c]
P = [d,a,b,c,d]
P = [b,c,d,a,b]
P = [a,b,c,d,a]
```

A related problem to the Hamiltonian path is that of the traveling salesman. In the traveling salesman problem we are given a graph as before. However each edge now has an associated *cost*. The solution to the traveling salesman problem is the (or a) Hamiltonian cycle with the minimal total edge cost.

Given an aggregate facility (such as Prolog's `findall` or `bagof`) which will enable all solutions to a given goal to be found, we can use the given program for finding Hamiltonian cycles as the basis for a solution to the traveling salesman problem. This is done by simply finding a Hamiltonian cycle and computing its cost. This computation is placed within a `findall`, which has the effect of finding all the Hamiltonian cycles in the graph, as well as the associated cost of each. We then simply select the minimum cost and return the associated cycle. Note that as this is an NP-complete problem, there is no better algorithm known than one which exhaustively searches through all possibilities.

In order to directly implement the solution described above, aggregate operators in Lygon are needed. As these are not yet present (but their effect can be simulated by some more lengthy code), we do not give the code for this problem here. However, it should be clear that this is not a difficult program to write.

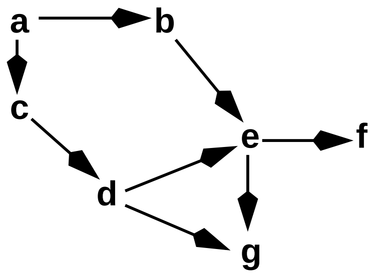
Another useful algorithm operating on graphs is the topological sort, which is often applied to the resolution of dependency problems. One application of topological sorting is to find an ordering of a set of dependent actions such that each action is preceded by the actions on which it depends.

The input to the algorithm is a directed acyclic graph (DAG), and the output is an ordering of the nodes in the graph. The computation is essentially a postorder traversal of the graph, where a node is only processed recursively and output the first time it is encountered.

There are two main predicates, `topsort` and `tsl` (short for topsortlist). The first terminates if there are no nodes left to process, otherwise it selects an arbitrary node and calls `tsl` on that node's links then postpends the node to the result before calling itself recursively.

The `getnode` and `getlink` predicates together conspire to ignore links emanating from nodes which have already been processed. Note the use of the operator `once`; the goal `once(G)` is operationally the same as the goal `G`, except that `once(G)` will succeed at most once. In this context the `once` is used to simulate an `if-then-else` construct.

For a given graph there are many orderings which are valid topological orderings. For example consider the following graph:



Given this graph, we can apply a topological sort using the query `topsort(X)` which returns the first solution

```
X = [g,f,e,d,c,b,a]
```

If we ask the system to find alternative solutions it comes up with the following distinct solutions:

```
X = [g,f,e,d,c,b,a]
X = [g,f,e,d,b,c,a]
X = [g,f,e,b,d,c,a]
X = [f,g,e,d,c,b,a]
X = [f,g,e,d,b,c,a]
X = [f,g,e,b,d,c,a]
```

The code is given below. It introduces a new connective: `linear`. By default in Lygon program clauses are re-usable (that is, non-linear). This gives behaviour similar to Prolog. The prefix unary connective `linear` specifies that a program clause is linear. Such clauses must be used *exactly* once, that is, they are consumed when they are used and they cannot be left over.

```
linear node(a,[a]).   linear link(a,[b,c]).
linear node(b,[b]).   linear link(b,[e]).
linear node(c,[c]).   linear link(c,[d]).
linear node(d,[d]).   linear link(d,[g,e]).
linear node(e,[e]).   linear link(e,[g,f]).
linear node(f,[f]).   linear link(f,[]).
linear node(g,[g]).   linear link(g,[]).

topsort(R) <- node(_,[N]) * link(N,L) *
              tsl(L,R1) * topsort(R2) *
              append(R1,[N|R2],R).
topsort([]).

ts(Node,R) <- getnode(Node,N) * getlink(N,L)
            * tsl(L,R1) * append(R1,N,R).
tsl([],[]).
tsl([N|Ns],R) <- ts(N,R1) * tsl(Ns,Rs)
               * append(R1,Rs,R).

getlink([],[]).
getlink([N],L) <- link(N,L).

getnode(N,R) <- once(node(N,R) @ eq(R,[])).
```

We have seen how Lygon allows concise and elegant solutions to a variety of problems involving graph manipulation where the graph is represented as linear facts. This seems to make Lygon a very useful tool for the solution of graph problems, as the ability to use each edge in the graph at most once means that simple and elegant programs will terminate even in the presence of cycles in the graph.

6 State Problems and Lygon

One of the features which distinguishes linear logic programming languages is the presence of a linear context. In this section we look at a number of problems which use this context to store state. This application of the context is useful in solving planning type problems where there is a notion of a state and operators which change the state.

The Yale shooting problem [5] is a prototypical example of a planning problem. The main technical challenge in the Yale shooting problem is to model the appropriate changes of state, subject to certain constraints. In particular:

1. Loading a gun changes its state from unloaded to loaded

2. Shooting a gun changes its state from loaded to unloaded

3. Shooting a loaded gun at a turkey changes its state from alive to dead

To model this situation in Lygon, we have predicates `alive, dead, loaded,` and `unloaded`, which will represent the given states, and two other predicates `load` and `shoot`, which, when executed, change the appropriate states. The initial state is to assert `alive` and `unloaded`, as initially the turkey is alive and the gun unloaded. The actions of loading and shooting are governed by the rules below:

```
load <- unloaded * neg loaded.
shoot <- alive * loaded * (neg dead
           # neg unloaded).
```

Hence given the initial resources `alive` and `unloaded`, the goal `shoot # load` will cause the state to change first to `alive` and `loaded`, as `shoot` cannot proceed unless `loaded` is true, and then `shoot` changes the state to `dead` and `unloaded`, as required.

Similar problems, such as the murder mystery problem, the fragile object problem, and the stolen car problem may be handled in a similar way.

An interesting point to note is that the rules for `load` and `shoot` may be written in the following manner:

```
load # loaded <- unloaded.
shoot # (dead * unloaded) <- alive * loaded.
```

This may be thought of as allowing non-atomic formulae as the heads of clauses. When written this way, the rules above may be considered as stating that if the state is `unloaded`, then on a request to `load`, the state is updated to `loaded`, and that if the state is `alive` and `loaded`, then on a request to `shoot`, then on a request to shoot, update the state to `dead` and `unloaded`. Hence this way of writing the rules makes the state changes a little more explicit than the previous one, and provides the possibility of a more flexible execution strategy, such as determining that in order to change the state of the gun from `unloaded` to `loaded`, it is necessary to perform a `load` action.

A slightly less artificial planning problem is blocks world. Blocks world consists of a number of blocks sitting either on a table or on another block and a robotic arm capable of picking up and moving a single block at a time. We seek to model the state of the world and of operations on it.

The predicates used to model the world in the Lygon program below are:

- `empty` The robotic arm is empty
- `hold(A)` The robotic arm is holding block A
- `clear(A)` Block A does not support another block
- `ontable(A)` Block A is supported by the table
- `on(A,B)` Block A is supported by block B

There are a number of operations that change the state of the world. We can `take` a block. This transfers a block that does not support another block into the robotic arm. It requires that the arm is empty and the block in question be supported by the table.

```
take(X) <- (empty * clear(X) * ontable(X))
             * neg hold(X).
```

We can `remove` a block from the block beneath it, which must be done before picking up the bottom block.

```
remove(X,Y) <- (empty * clear(X) * on(X,Y))
       * (neg hold(X) # neg clear(Y)).
```

We can also `put` a block down on the table or `stack` it on another block.

```
put(X) <- hold(X) * (neg empty # neg clear(X)
        # neg ontable(X)).
stack(X,Y) <- (hold(X) * clear(Y)) * (neg
   empty # neg clear(X) # neg on(X,Y)).
```

Finally, we can describe the initial state:

```
initial <- neg ontable(a) # neg ontable(b)
         # neg on(c,a) # neg clear(b)
         # neg clear(c) # neg empty.
```

We need to have some way of displaying the state of the world. We do this by collecting the state of the world into a single structure which can then be printed.

```
showall([ontable(X)|R]) <-  ontable(X)
      * showall(R).
showall([clear(X)|R]) <-  clear(X)
      * showall(R).
showall([on(X,Y)|R]) <-  on(X,Y) * showall(R).
showall([hold(X)|R]) <-  hold(X) * showall(R).
showall([empty|R]) <-  empty * showall(R).
showall([]).
```

And finally, we can start moving blocks around:

```
go <- (initial # remove(c,a) # put(c)
      # take(a) # stack(a,b)
      # showall(R)) * print(R) * nl.
```

The query `go.` outputs the result

```
[empty,on(a,b),clear(a),clear(c),ontable(c),
  ontable(b)]
```

Note that the order of the instructions `take`, `put` etc. is not significant; there are some actions, as specified by the rules, which cannot take place from the initial state (such as `put(c)`), and others, such as `take(b)` which may. It is the problem of the implementation to find an appropriate order in which to execute the instructions, and come up with the final state.

As in the previous example, by allowing clause heads to contain multiple formulae (in the manner of LO[2]) the rules describing state transitions become clearer to state and to read, and allow the possibility of more flexible execution.

For example the `put` rule becomes:

```
put(X) # (empty * clear(X) * ontable(X))
                                <- hold(X).
```

This rule can be read as "given that we are `hold`ing block X, we can do a `put`. The resulting state has the `hold` fact deleted and contains the facts `empty`, `clear(X)` and `ontable(X)`".

These two examples should make clear the potential for Lygon in the area of planning problems, which often make use of intricate sets of rules and some form of update.

7 Features that would help

Having written a variety of programs in Lygon, and particularly those concerning graph problems and planning problems, we have come to appreciate the power of the language. In addition, there are various programming constructs which would further simplify the code in places, and generally make the programming task simpler. In this section we discuss some of these constructs.

One of the most interesting inference rules used in logic programming is *Negation as Failure*, i.e. that if a goal fails, then its negation succeeds. This has been put to good use in various applications of Prolog. In the case of Lygon, it would seem that this construct would be useful as well, particularly to indicate when a particular resource has been exhausted. For example, in the Hamiltonian circuit program above, we want to ensure that all of the nodes are on the circuit, and so we want to be able to tell when the predicate `node` has been exhausted, but this may occur independently of whether or not other predicates, such as `edge` are exhausted or not. Whilst we can use a combination of `one` (which may be thought of as indicating that all resources are exhausted), `&` (which copies resources) and a sub-program to use up remaining resources for particular predicates to achieve the same effect, it would clearly be simpler (and potentially more efficient) to have a direct test for this circumstance, i.e. a connective for Negation as Failure in Lygon. A full answer to the problem of Negation as Failure within linear logic programming languages is yet to be found; however, it should be clear that a simple test to determine whether or not there are any "facts" remaining in a given predicate would clearly be useful.

There are other similar constructs which would also prove useful, which are generally known as *aggregates*. These often include facilities for finding all solutions to a given goal (such as `findall`, `bagof` and `setof` in Prolog), or more specific properties such as a maximum, minimum or a count of some kind. Such facilities generally require all possible solutions to a goal to be found, and hence often greatly simplify code by eliminating the necessity for the programmer to specify the way in which the various alternative solutions are found and the appropriate information stored internally. As seen in the traveling salesman problem above, this kind of facility allows a very powerful programming methodology.

Given the natural way in which planning problems may be implemented in Lygon, it would seem natural to provide macros to simplify this task even further. For example, given a problem such as the blocks world, we may wish to allow the programmer to write the transition rules in a manner such as `action: OldState -> NewState`, where `OldState` represents the old state of the world (or at least the relevant part of it), and `NewState` the updated state, and have a preprocessor generate code similar to that above. This would also help facilitate the use of such problems in an "any two out of three" manner; given two of the initial state, final state and desired action, compute the third.

A paradigm that this way of programming suggests is to think of resources as the input and output of a computation. Traditionally in logic programming languages, inputs and outputs are substitutions, i.e. particular values of logical variables. However, there are some circumstances where a more general notion of input and output would be useful, such as the blocks world problem, where it seems natural to view the current state of the world as the input, and the final state as the output, with computation being the way of applying the actions to go from one to the other. Hence it seems it would be useful to be able to simply calculate the final state and output it, rather than having to gather it up into an answer substitution, which is the case at present.

8 Conclusions and Further Work

We have seen how Lygon may be applied to various problems, but particular those in which it is useful to apply constraints such as "this resource can be used at most (or exactly) once", such as graph problems, and those which involve a notion of state (such as the blocks world). The Lygon system is freely available, and may be obtained, together with all of the example programs mentioned in this paper and various other documents, from the URL `http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html` or via email from the authors.

There remains much further work to be done. We have already mentioned how facilities such as Negation as Failure and aggregates would enhance the system. We also believe that Lygon is an excellent vehicle for planning problems in general, and we hope to investigate further applications along this line. In addition, linear logic has often been applied to the study of concurrency [2], and so it seems interesting to explore how languages such as Lygon may be used to combine both planning features and concurrency facilities, and hence this may be an appropriate vehicle for agent-oriented programming.

Another possibility is the application of bottom-up execution methods. Unlike Prolog systems, in which *backward reasoning* is used (i.e. this conclusion is true if this premise is true), bottom-up execution methods use *forward reasoning* (i.e., this premise is true, and so this conclusion is true). These have often been applied in logic programming systems to deductive database problems, whose queries may have large numbers of answers. However, the bottom-up execution of linear logic programming languages appears to have applications to areas such as active databases, as there would be a need to maintain a notion of state, just as in the top-down case.

Another area in which languages like Lygon may be applied is in the area of transactions. Often transaction processing systems involve complex concurrent algorithms, which must maintain certain properties, such as atomicity, and serializability. Clearly a system in which updates may be reconciled with logical reasoning has the potential to simplify the programming effort needed to implement a transaction processing system. We intend to explore this direction more fully in the near future.

Acknowledgments

The topological sort program was inspired by the work of Yi Xiao Xu. We would like to thank David Pym for stimulating discussions.

The support of the Australian Research Council, the Collaborative Information Technology Research Institute and the Centre for Intelligent Decision Sytems is gratefully acknowledged. Michael Winikoff is supported by an Australian Postgraduate Award (APA) scholarship.

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] S. Hanks and D. MacDermott. Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence* 33:3:379-412, 1987.

[6] 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.

[7] 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.

[8] J. Hodas, D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. Journal of Information and Computation 110:2:327-365, 1994.

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

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

[11] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. Workshop on Extensions to Logic Programming, E. Lamma and P. Mello (eds.), Springer Verlag, 1992.

[12] D. Miller. A Multiple-Conclusion Meta-logic. Proceedings of the Symposium on Logic in Computer Science 272-281, 1994.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] David S. Warren. Programming the PTQ Grammar in XSB. in *Applications of Logic Databases*, Raghu Ramakrishna (ed.), Kluwer Academic, 1994.

[19] M. Winikoff and J. Harland. *Implementation and Development Issues for the Linear Logic Programming Language Lygon.* Proceedings of the Eighteenth Australasian Computer Science Conference 562-573, Adelaide, February, 1995.

[20] M. Winikoff and J. Harland. *Implementing the Linear Logic Programming Language Lygon* Proceedings of the International Logic Programming Symposium, Portland, Oregon, December, 1995.