Programming in Lygon: an overview

Harland, James; Pym, David; Winikoff, Michael

Technical Report No. 720

Department of Computer Science

Programming in Lygon: an overview

James Harland
David Pym
Michael Winikoff

Queen Mary and Westfield College

1996

To appear: Proc. AMAST 96, M. Wirsing (ed.), LNCS, Springer, 1996.

Programming in Lygon: an overview

James Harland[1] David Pym[2] Michael Winikoff[3]

[1] RMIT, GPO Box 2476V, Melbourne 3001, Australia
[2] Queen Mary & Westfield College, University of London, UK
[3] University of Melbourne, Parkville 3052, Australia

Abstract. For many given systems of logic, it is possible to identify, via *systematic* proof-theoretic analyses, a fragment which can be used as a basis for a logic programming language. Such analyses have been applied to linear logic, a logic of *resource-consumption*, leading to the definition of the linear logic programming language *Lygon*. It appears that (the basis of) Lygon can be considered to be the largest possible first-order linear logic programming language derivable in this way. In this paper, we describe the design and application of Lygon. We give examples which illustrate the advantages of resource-oriented logic programming languages.

1 Introduction

Logic programming languages are based upon the observation that certain sequents can be interpreted as a program together with a goal: if Γ is a set of program clauses and $\exists x \,.\, \phi$ is a goal clause, the existentially bound variable x being a "logical variable", then the sequent $\Gamma \vdash \exists x \,.\, \phi$ is interpreted as a request to search for a term t, *i.e.*, an answer substitution, together with a proof of the sequent $\Gamma \vdash \phi[t/x]$.

In order to identify the fragment of a given system of logic which can form the basis of a logic programming language, one must have an independent notion of what is meant by a logic programming language. Whilst any definition should include the use of Horn clauses in classical logic as a special case, it is not clear exactly what characterizes a logic programming language. For example, precisely how does a logic programming system differ from a theorem prover ? An obvious point of departure is a consideration of the amount of non-determinism inherent in the search for a proof. Implementations of (logic) programming languages must be tolerably efficient: roughly speaking, the lesser the non-determinism, the greater the efficiency. Furthermore, the requirement of "minimal" non-determinism is also motivated by certain "definiteness" requirements (see, for example, [16]). So it is appropriate to consider design principles that limit the non-determinism in the search for proofs.

One such principle is that of *goal-directed provability*, in which the search strategy is determined by the structure of the goal and which the program supplies the context of the (putative) proof. Once we have determined an appropriate notion of goal-directed provability, we can look for classes of formulae for which goal-directed provability is complete for the given consequence relation. Such classes of formulae then form logic programming languages. The analysis of logic programming languages based on this criterion has been carried out for various logics and classes of formulae, including intuitionistic logic [16], higher-order log-

ics [16] and linear logic [7, 17, 12, 19]. Logic programming languages based on linear logic allow a notion of *resource-oriented* programming: by default, a clause in such a program must be used exactly once. This makes the writing of many programs simpler and more intuitive than in a language, such as Prolog, based on classical logic. For example, the resource-sensitive nature of linear logic means that path-problems in graphs can be solved simply and elegantly even in the presence of cycles in the graph. Moreover, the standard transitive closure predicate can be used, with a very minor modification, which will find any path in a graph, whether it is cyclic or not.

In this paper we give an overview of the linear logic programming language *Lygon*. Lygon is based on a fragment of classical linear logic [7] (of which we assume a basic knowledge) identified as a basis for logic programming via a systematic proof-theoretic analysis [17].

In common with other linear logic programming languages, Lygon allows clauses to be used exactly once in a computation, thereby avoiding the need for the explicit resource-counting often necessary in Prolog-like languages. Just as linear logic is a strict extension of classical logic, Lygon is a strict extension of (pure) Prolog: all (pure) Prolog programs can be executed by the Lygon system. Hence all the features of classical pure logic programs are available in Lygon, together with new ones based on linear logic. These include a theoretically transparent notion of *state*, a notion of resources and a form of concurrency. All of these follow from the basis of Lygon in linear logic and *do not require* extra-logical features for their definition.

2 Goal-directedness and resolution in linear logic

In order to obtain a logic programming language based on linear logic, we must, according to the criteria of § 1, identify a class of formulae for which an appropriate notion of resolution proof is complete. Our analysis, presented in detail in [17], appears to provide as broad as possible an interpretation of goal-directedness and thereby appears to allow the broadest possible linear logic programming language. Although the details are beyond the scope of this paper, we briefly review the key ideas.

In the cut-free (linear) sequent calculus, goal-directed proof can be achieved via *uniform proof* [16, 17]. The basic idea, introduced in [16], is to use the left- and right-rules of the sequent calculus as *reduction operators* (in the sense of [13]) and proceed as follows: if, at any stage, some right-rules are applicable, then one of them must be applied; otherwise, *i.e.*, if all goal-formulae are atomic, proceed to apply a left-rule. In linear logic with multiple conclusions, this basic notion is not quite adequate. A slightly weaker notion of goal-directedness, characterized by *simple locally LR proofs* [17], is required. In simple locally LR proofs, certain, highly restricted, occurrences of $\multimap$L are permitted below occurrences of right rules.[4]

Resolution proof is a refinement of uniform (simple locally LR) proof so that only one left-rule, the resolution rule, is required. Completeness of uniform proof depends on a restriction to hereditary Harrop formulae, *i.e.*, just definite-formulae on the left and just

[4] In [17], uniform proofs are defined to be simple locally LR proofs.

goal-formulae on the right. Completeness of resolution depends on definite formulae being expressed in a suitable *clausal form*. Full detail of the resolution rule and its proof-theoretic properties can be found in [17]; a sketch is provided below.

2.1 Definite formulae and goal formulae

Goal-directed (uniform [17]) proof is sound and complete for linear hereditary Harrop sequents, $\Gamma \vdash \Delta$, in which Γ and Δ are composed, respectively, of the following classes of D-formulae and G-formulae:

$$\begin{aligned} D ::= {} & A \mid \mathbf{1} \mid \bot \mid D \,\&\, D \mid D \otimes D \\ & \mid D \parr D \mid \forall x \,.\, D \mid {!}\, D \\ & \mid G \multimap A \mid G \multimap \bot \mid G \multimap \mathbf{1} \end{aligned}$$

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

where A ranges over atomic formulae.[5][6] This class of formulae is compared the language Forum [15] in § 5.

For single-conclusioned sequents, the goal-directed interpretation of linear hereditary Harrop formulae is determined by the right-rules of the linear sequent calculus and is explained in detail in [17]. We review a few important cases. (Note that for multiple-conclusioned sequents, which are forced by the presence of $\parr$ in goals, the situation is complicated by the need to consider locally LR proofs [17].).

- $G_1 \otimes G_2$ is a consequence of Γ *only if* G_1 is a consequence of Γ_1, G_2 is a consequence of Γ_2 and $\Gamma = \Gamma_1, \Gamma_2$. The resources in Γ must be divided into those available to solve G_1 and those available to solve G_2.[7]
- $G_1 \,\&\, G_2$ is a consequence of Γ *only if* G_1 and G_2 are consequences of Γ. The resources in Γ must be used to solve each of G_1 and G_2..
- $D \multimap G$ is a consequence of Γ *only if* G is a consequence of $\Gamma, [D]$. The clausal form $[D]$ of the formula D must be added to the program Γ. The definition of the mapping $[-]$ is related to our definition of resolution: it is discussed below.
- $G_1 \parr G_2$ is a consequence of Γ *only if* G_1, G_2 is a consequence of Γ. The program must have sufficient resources to solve both G_1 and G_2 simultaneously. For example, Γ includes a clause of the form $D_1 \parr D_2$, then a resolution step (see below) driven by such a clause will cause both program (and goal) to be split, thereby dividing the program into those resources available

[5] Slight extensions to this class are possible. Note that definite formulae of the form $G \multimap (D_1 \parr D_2)$ are equivalent to $(G \multimap \bot) \parr D_1 \parr D_2$. If D_1 and D_2 are atomic, then the equivalence is goal-directed. Weaker notions of goal-directedness, parameterized on classes of definite formulae, are possible. For example, we can choose not to enforce the reduction of a tensor product of atomic formulae, so obtaining goal-directedness "up to $A_1 \otimes A_2$". Under such a choice, the equivalence above is goal-directed even if D_1 and D_2 are permitted to be of the form $A_1 \otimes A_2$. See also Footnote 12.

[6] $D ::= G^{\perp}$ and $G ::= D^{\perp}$ are implicit.

[7] In general, for multiple-conclusioned sequents, we must divide the resources in the succedent (other than $G_1 \otimes G_2$) as well, but we omit this for simplicity here.

to solve each of G_1 and G_2. A similar splitting is required for the formulae in the succedent.

Discussions of the remaining connectives can be found in [17, 22].

So the right-rules of the linear sequent calculus provide us with an operational interpretation of goal formulae. For resolution proof, we must also provide an operational account of definite formulae.

Essentially, in goal-directed proof, we invoke a left-rule only if no right-rule is applicable, with an exception in the case of $\multimap$L. Consider, then, $\multimap$L,

$$\frac{\Gamma_1 \vdash \phi, \Delta_1 \quad \Gamma_2, \psi \vdash \Delta_2}{\Gamma_1, \Gamma_2, \phi \multimap \psi \vdash \Delta_1, \Delta_2}.$$

In resolution proof, we require a special case of this rule in which the right-hand premiss is an axiom,

$$\frac{\Gamma \vdash \phi, \Delta \quad \overline{\psi \vdash \psi}}{\Gamma, \phi \multimap \psi \vdash \psi, \Delta}.$$

Note that although resolution proof requires that ψ be atomic, non-atomic formulae are permitted in Δ. In this sense, resolution proof is characterized by locally LR proof.

We also require that resolution be the only left-rule required. This is achieved by restricting formulae in the antecendent to be *clausal*. In our setting, the resolution rule codes up instances of $\multimap$L, $\parr$L, &L, $\forall$L and !L, as well as contraction on the left (via !) that can occur in locally LR proofs. Note that $\otimes$L is absent from this list. Outermost occurrences of $\otimes$ on the left are removed by the mapping $[-]$ which reduces (multisets of) linear definite formulae to clausal form in such a way that logical consequence is preserved. Briefly, the general form of the resolution rule is

$$\frac{\Gamma_1 \vdash \Delta_1 \ldots \Gamma_m \vdash \Delta_m}{\Gamma \vdash \Delta},$$

where $\bigcup_{i=1}^{m} \{ \Gamma_i \vdash \Delta_i \}$ is a *resolvant* of $\Gamma \vdash \Delta$, a resolvant being a multiset of sequents that decomposes the structure of the conclusion of the resolution rule according to the structure of a *selected clause*. Propositionally, the basic units of clauses are atomic formulae and the implicational definite formulae, *e.g.*, $G \multimap A$. However, in order for resolution to code up the required instances of &L and $\parr$L, these basic units must be combined by certain instances of &, $\parr$ and $\otimes$. For example, if we let c denote the clause $!((p \multimap r) \parr ((q_1 \otimes q_2) \multimap s)$, then

$$\frac{c, p, p \multimap r \vdash r \quad c, q_1, q_2 \vdash q_1 \otimes q_2}{p, q_1, q_2, !((p \multimap r) \parr ((q_1 \otimes q_2) \multimap s) \vdash r, s}$$

is an instance of resolution, in which c is the selected clause. In this case, the resolvant has two components, *i.e.*, $m = 2$ because the clause c has as subformulae two basic clauses, $p \multimap r$ and $(q_1 \otimes q_2) \multimap s$, combined by a $\parr$. In this instance of resolution, the basic clause $(q_1 \otimes q_2) \multimap s$ has driven the rule, by matching its head s with the s in the succedent. Since this basic clause is connected in c by a $\parr$ to the basic clause $p \multimap r$, this latter must now also be used. In this example, it is available for use on the left-hand branch. Since c has a ! as its outermost connective, it can continue to be available on both branches, *i.e.*, resolution builds in contractions. The details of clauses and resolution can be found in [17].

To understand the mapping $[-]$, defined fully in in [17], consider a few examples. Firstly, $[p \otimes q] =_{\text{def}} \{ p, q \}$, so

that when querying the program $p \otimes q$ with the goal $p \otimes q$, we search for a proof of the sequent $p, q \vdash p \otimes q$, which has a goal-directed proof. Secondly, $[(\forall x\,.\,(p \multimap q(x))) \otimes (r \multimap s)] =_{\text{def}} \{\, p \multimap q(x), r \multimap s \,\}$. Finally, $[!D] =_{\text{def}} \{\, !\bigotimes_{C \in [D]} C \,\}$.

Resolution proof is complete for consequences of the form $[\mathcal{D}] \vdash \mathcal{G}$. [8]

2.2 Searching for resolution proofs

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, the problem of "splitting" programs, as specified by the following rule for introducing $\otimes$ on the right, when read as a reduction operator from conclusion to premisses, is not addressed:

$$\frac{\Gamma_1 \vdash G_1, \Delta_1 \quad \Gamma_2 \vdash G_2, \Delta_2}{\Gamma_1, \Gamma_2 \vdash G_1 \otimes G_2, \Delta_1, \Delta_2}.$$

The main problem here is how to "split" a program and goal along the lines specified by this rule. As there is an exponential number of sub-multisets of a given multiset, an exhaustive approach is not feasible. Hodas and Miller [12] have proposed a *lazy sequential* approach to this problem, in which the first conjunct is given all the resources, and those which are not consumed are passed to the second, which must consume all remaining resources.[9]

In the input/output model, the constraint that each formula must appear in exactly one branch is maintained by evaluating the conjuncts sequentially. As the class of formulae considered by Hodas and Miller is somewhat more restrictive than the class of formulae above, it is not obvious how this technique can be systematically applied to the class of formula used in Lygon. It turns out that this approach requires some significant revisions to the proof system. A full description of this process, which is beyond the scope of this paper, can be found in [22]. A brief summary can also be found in [10].

An important point to note that this method of implementation provides us with a method of "state-passing", in that we can think of the passage of "excess" resources from one branch to another as a means of state transition. As we shall see, there are some restrictions placed on this process, in accordance with the rules of the logic, but nonetheless this process has many interesting and useful applications.

3 Programming techniques and examples

In this section, we discuss various programming techniques which distinguish Lygon from Prolog (see also [9, 23, 22, 21]), including several examples of Lygon programs. We begin with brief remarks on Lygon syntax [22].

3.1 Lygon syntax

The current implementation of Lygon (Version 0.4) is an interpreter written in BinProlog[10] The interpreter com-

[8] The current Lygon interpreter [22] implements resolution proofs via one-sided sequents [7].

[9] The so-called "input/output model of resource consumption".

[10] Version 0.4 of Lygon is available from the authors by email or via the World

prises about 500 lines of code, including comments and whitespace. Version 0.4 supports a limited form of program: clauses are limited to the forms atoms, $\forall \overline{x}.(G \multimap A)$ and $!(\forall \overline{x}.(G \multimap A))$. Universal quantifiers are excluded from goals. Universal and existential quantifiers in programs and goals, respectively, need not be written explicitly, Prolog-style (upper case) logical variables being acceptable. We use the following mapping $\Rightarrow$ between logical connectives and ASCII:

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

Predicates are as in Prolog. The Lygon equivalent of the Prolog :- is <-.

3.2 Resource allocation

It can be useful sometimes to ignore certain resources, so that rather than requiring that they be used *exactly* once, we only require that they be used *at most* once. Whilst in general this would require a different logic (known as *affine logic*, *i.e.*, linear logic with an unrestricted weakening rule), the effect of this less restrictive approach can be simulated in linear logic. For example, to ensure that the clause C can be used at most once, we can write $C \& \mathbf{1}$. Then we can either use C as normal, or instead use the linear constant $\mathbf{1}$, which we can interpret as the empty clause. Note that this is similar to allowing the weakening rule but not contraction, so that we refer to this as *affine mode*.

A similar trick can be used to specify that a goal need not consume all of the available resources. Consider a program $\mathcal{P}$ and the goal $G \otimes \top$. In order for this goal to succeed, we must be able to divide up the program so that $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$, $\mathcal{P}_1 \vdash G$ and $\mathcal{P}_2 \vdash \top$. Now as the latter sequent is provable for any program $\mathcal{P}_2$ (including the empty program), the goal G need not consume all of the resources of $\mathcal{P}$, as any "leftovers" will be accounted for by $\top$.

An important application of this notion of resource allocation is graph problems. Graphs are an important data structure in computer science. Indeed, there are many applications of graph problems, such as laying cable networks, evaluating dependencies, designing circuits and optimization problems. 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 consider, concise and lucid.

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 [20], or a bottom-up system such

Wide Web [21].

as Aditi [18]. However, it is common to re-write the program above so that the path found is returned as part of the answer. In such cases, systems 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 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` can used directly to find Eulerian circuits, *i.e.*, circuits which use every edge in the graph exactly once). However, the `path` predicate can ignore certain edges, provided that it does not visit any edge more than once, and so the `path` predicate may be considered the affine form of the predicate `trip`.

The goal `graph` is used to load the linear copy of the graph, and as this is a non-linear rule, we can 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,_,P).
```

This goal yields the solutions below. Note that in the following and in other interactions with the Lygon system we elide irrelevant system responses such as `More? (y/n)`.

```
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:

```
X = c, P = [c,d,a,b,c]
X = d, P = [d,a,b,c,d]
X = b, P = [b,c,d,a,b]
X = a, 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))`..

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, *i.e.*, those visiting every node in the graph exactly once, which involve counting. 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 can 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 [12] have shown that such permutation problems can be solved simply 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 full Lygon program for finding Hamiltonian cycles 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 rôle 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 gives 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]
```

A problem related to the Hamiltonian path is that of the travelling salesman. In the travelling salesman problem we are given a graph as before. However each edge now has an associated *cost*. The solution to the travelling salesman problem is the (or a) Hamiltonian cycle with the minimal total edge cost. Given a facility for finding aggregates, such as `findall` or `bagof` in Prolog, 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 travelling salesman problem. This would be done by simply finding a Hamiltonian cycle and computing its cost. This computation would be placed within a `findall`, which would have the effect of finding all the Hamiltonian cycles in the graph, as well as the associated cost of each. We would 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.

3.3 Representing states and actions

When attempting to find a proof of $\mathcal{P} \vdash G_1 \otimes G_2, \mathcal{G}$, we use the technique of passing the unused resources from one conjunct to the other. This can be used as a kind of state-mechanism, in that the first conjunct can pass on information to the second. In particular, we can use this feature to simulate a memory. For example, consider a memory of just two cells, represented by two instances of the predicate m, the first argument being the address and the second the contents of the cell. The state in which these two cells contain the values t_1 and t_2 would then be represented by the multiset of clauses $\{m(1,t_1), m(2,t_2)\}$. A (non-destructive) read for cell 2, say, would be given by the goal $m(2,x) \otimes (m(2,x) \multimap G)$, where G is to be executed after the read. The states in this computation are (i) that $m(2,x)$ is unified with $m(2,t_2)$, (ii) that the latter atom is deleted from the program, and then (iii) added again via the $\multimap$ connective, before G is executed. Note that a similar sequence occurs for the goal $m(2,x) \otimes m^{\perp}(2,x)$.[11] Similarly, writing the value t' into the memory can be done using the goal $m(2,x) \otimes (m(2,t') \multimap G)$, where it is possible that t' can contain x, so that either the new value can be dependent on the old, or t' can be totally independent of the old value. In this way we can use the "delete after use" property of the linear system to model a certain form of destructive assignment.

Using a continuation passing style to encode sequentiality, with a predicate `call` to invoke continuations, we can create an abstract data type for memory cells using the operations `newcell/3`, `lookup/3` and `update/3`.

```
newcell(Id,Value,Cont) <-
 neg m(Id,Value) # call(Cont).
lookup(Id,Value,Cont) <-
 m(Id,Value) * (neg m(Id,Value)
 # call(Cont)).
update(Id,NewValue,Cont) <-
 m(Id,_) * (neg m(Id,NewValue)
 # call(Cont)).
```

For example, consider summing a list using a variable which is updated: the `top` in the second clause is needed to consume the cell once it is no longer needed.

```
sum(List,Result) <-
 newcell(sum,0,
         sumlist(List,Result)).
sumlist([],Result) <-
 lookup(sum,Result,top).
sumlist([N|Ns],Result) <-
 lookup(sum,S,(is(S1,S+N) *
 update(sum,S1,
        sumlist(Ns,Result)))).
```

We can then run the program using a goal such as `sum([1,5,3,6,7],X)` which yields the solution `X = 22`.

The notion of state present in Lygon can also be applied in planning type problems where there is a notion of a state and operators which change the state.

The Yale shooting problem [8] is a prototypical example of a problem

[11] We write $m^{\perp}(2,x)$, rather than $m(2,x) \multimap \perp$, for brevity.

involving actions. 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 in Lygon, we have predicates `alive`, `dead`, `loaded`, and `unloaded`, representing the given states, and 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 following rules:

```
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.

Note that the rules for `load` and `shoot` can be written in the following manner:[12]

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

[12] The first clause `load # loaded <- unloaded.` can be considered an abbreviation for `( bot <- unloaded ) # loaded # load.`. See Footnote 5.

Written in this way, the rules above can be considered as stating that if the state is `unloaded`, then on a request to `load`, the state is updated to `loaded`; that if the state is `alive` and `loaded`, 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 the order into which to change the state of the gun from `unloaded` to `loaded`, it is necessary to perform a `load` action.

Note that this program makes essential use of the possibility afforded by Lygon of positive occurrences of # in clauses. In languages, such as Lolli [12], lacking this facility, the second clause, `shoot # (dead * unloaded) <- alive * loaded.`, would have to be written as two special cases. Although the language Forum [15] can represent the first clause, `load # loaded <- unloaded.`, directly, its representation of the second clause would necessarily be more complex and less transparent than Lygon's. Similar remarks obtain for the language LO [2].

A (slightly) less artificial planning problem is the blocks world. The 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 the following:

- `empty`: the robotic arm is empty;
- `hold(A)`: the robotic arm is holding block A;
- `clear(A)`: block A does not sup-

port 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.

```
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 of the blocks.

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

Lygon (initial # take(c) # put(c)
      # take(a) # stack(a,b)
      # showall(R)).
[empty,on(a,b),clear(a),clear(c),
 ontable(c), ontable(b)]
Succeeded.
```

The order of the instructions `take`, `put` *etc.* is not significant: there are actions, specified by the rules, such as `put(c)`, which cannot take place from the initial state, and others, such as `take(b)` which can. It is the problem of the implementation to find an appropriate order in which to execute the instructions, so giving the final state.

By allowing clause heads to contain multiple formulae (in the manner of LO [2]), the rules describing state transitions become more clearly stated 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 `holding` 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)`".

3.4 Concurrency

Our next example is the classical dining philosophers (or logic programmers) problem and illustrates the use of Lygon to model concurrent behaviour.[13] This solution is adapted from [4].

For N logic programmers there are $N-1$ "room tickets". Before entering the room each logic programmer must take a roomticket from a shelf beside the door. This prevents all of the programmers from being in the room at the same time.

The program uses a number of linear predicates: `rm` represents a roomticket, `log(X)` represents the Xth programmer and `ch(X)` the Xth chopstick.

[13] This problem is particularly apt — Lygon's name is of gastronomic origin.

Since the Lygon implementation is basically unfair, one of the logic programmers dominates the action. It is a simple matter (five lines of code) to modify the Lygon interpreter to use a fairer strategy which allows all of the programmers a chance to dine.

```
go <-  log(a) # neg ch(a)
# neg rm # log(b) # neg ch(b)
# neg rm # log(c) # neg ch(c)
# neg rm # log(d) # neg ch(d)
# neg rm # log(e) # neg ch(e).

log(N) <- hack(N) * rm *
 succmod(N,N1) * ch(N) * ch(N1)
 * eat(N) * (neg ch(N) #
 neg ch(N1) # neg rm # log(N)).
```

Procedurally, this code is read as: get a room ticket; get the chopsticks in sequence; eat; return the chopsticks and room ticket; go back to hacking.

```
succmod(a,b).     succmod(d,e).
succmod(b,c).     succmod(e,a).
succmod(c,d).

eat(N) <- print('log(')
 * print(N) * print(')
 eating') * nl.
hack(N) <- print('log(')
 * print(N) * print(')
 hacking') * nl.
```

We have the following interaction with the modified system:

```
Lygon ['phil.lyg'].
.............................
Lygon go.
log(e) hacking    log(e) eating
log(d) hacking    log(d) eating
```

3.5 Counting clauses

Another problem, in which the properties of linear logic make a significant simplification and which has been discussed as a motivation for the use of embedded implications in the presence of Negation-as-Failure [3, 5], is the following: given a number of clauses $r(1)$, $\ldots r(n)$, how can we determine whether n is odd or even ? The program below has been used for this purpose.

```
even :- not odd.
odd :- select(X),
       (mark(X) => even).
select(X) :- r(X), not mark(X).
```

Note the dependence on the co-existence of Negation-as-Failure and embedded implications. In the linear case, there is no need to do the explicit marking, as this will be taken care of by the Lygon system. This can be thought of as a simple aggregate problem; a good solution to this would indicate potential for more involved problems (and possibly some meta-programming possibilities). Clearly the marking step can be subsumed by the linear properties of Lygon, resulting in a conceptually simpler program, which is given below.

```
check(Y) <-
 r(X) * (toggle # check(Y)).
check(X) <- count(X).

toggle <- (count(even) *
 neg count(odd)) @ (count(odd)
 * neg count(even)).
```

The goal

```
neg count(even) # neg r(1)
# neg r(2) # check(X).
```

returns the answer `X = even`.

4 Other techniques

We briefly review the possibility of programming in Lygon with notions of *glo-*

bal variables, *mutual exclusion* and *preserving context*. Other possible notions, not considered here but briefly discussed in [9], include "soft" deletes and additions.

4.1 Global variables

In linear logic, formulae cannot be copied unless they commence with a !, variables which appear outside the scope of a ! cannot be standardized apart; hence variable names can persist across clauses. Consequently, we denote any universally quantified variable which appears in a clause outside the scope of any occurrence of ! as a *global variable*. Such a variable can occur in more than one clause and, in contrast to Prolog, such occurrences cannot be standardized apart. This is because in linear logic the universal quantifier does not distribute over $\otimes$, *i.e.*, that the two formulae $\forall x.\,(p(x) \otimes q(x))$ and $(\forall x.\,p(x)) \otimes (\forall x.\,q(x))$ are not equivalent. This contrasts with the case in classical logic, in which the formulae $\forall x.\,(p(x) \wedge q(x))$ and $(\forall x.\,p(x)) \wedge (\forall x.\,q(x))$ are equivalent. So in linear logic, the substitution generated by a unification must sometimes be propagated to other clauses.

Thus when a global variable is involved in a unification, the resulting substitution must be applied to other parts of the program. In programming terms, this property can be interpreted as a (restricted) form of a pointer.

Another possible application of global variables is to message passing. In this case, the variable would be instantiated to a non-ground term, usually a list, with the ground parts of the instantiating term being interpreted as a message. The last element of the list would always be a variable, thus ensuring that the message-list can always have a further message appended to it. In this way different clauses can communicate by means of this shared message-list.

A mixture of global and local variables can have some interesting effects. For example, consider the definite formulae $\forall x.\,!\forall y.\ p(x,y)$, corresponding to the clause $!p(x,y)$, in which x is global and y is local. For the formula $p(t_1,t_2) \otimes p(u_1,u_2)$ to be provable, we must have that $t_1 = u_1$; but there is no restriction on t_2 and u_2. Hence we have that $!p(x,y) \vdash p(a,b) \otimes p(a,c)$ but not $!p(x,y) \vdash p(a,b) \otimes p(c,d)$.

4.2 Mutual exclusion

The connective & can be thought of as specifying internal choice; in other words, when faced with a linear program clause such as $C_1 \,\&\, C_2$, one can replace it with either C_1 or C_2. However, it is not (generally) possible to replace it with *both* subformulae. Thus & can be interpreted as a mutual exclusion operator — the use of one formulae precludes the use of the other. Logically, this arises from the form of the &L rule, read as a reduction operator. For example, in

$$\frac{\Gamma, G_i \multimap A_i \vdash \Delta}{\Gamma, (G_1 \multimap A_1) \,\&\, (G_2 \multimap A_2) \vdash \Delta} \quad i = 1, 2,$$

once we have chosen one of the basic clauses $G_i \multimap A_i$, the other is no longer available.

Operationally, it seems most natural to implement this via backtracking, which means that & behaves in a similar manner to pruning operators. For example, given the formula $C_1 \& C_2$ and

the goal G, we first try to prove G using the formula C_1. If we find that G succeeds, we are done. If G fails, then we can backtrack and use C_2 instead. However, at no time are both C_1 and C_2 available.

4.3 Preserving context

As noted above, one of the key features of Lygon is the linear context that must be maintained, *i.e.*, the current resources. These resources are generally updated by one goal and passed to another. However, if it is desired to maintain the same resources, then & in goals can be used. For example, to determine if G_1 succeeds and then restore the same original context for the goal G_2, we need only ask the goal $G_1 \,\&\, G_2$. In general, it can be better to use the goal $(G_1 \otimes \top) \,\&\, G_2$, so that not all resources need be consumed by the test.

5 Other linear logic programming languages

As well as Lygon, there are various other logic programming languages based on linear logic. These include LO [2], LinLog [1], ACL [14], $\mathcal{LC}$ [19], Lolli [12, 11] and Forum [15].

An adequate comparison of Lygon with these languages is beyond the scope of this paper. However, we emphasize the following points: (i) Lygon is the result of a systematic proof-theoretic analysis of linear logic with respect to the goal-directed account of logic programming [17, 16]; (ii) Consequently, Lygon is (based on) the largest fragment of linear logic of all the languages given above. Although Forum is described (in [15]) as a logic programming language for all of linear logic, only a fragment of Forum is a logic programming language according to the goal-directed account of logic programming [16, 17], the rest of linear logic being obtained via equivalences which lack goal-directed proofs.[14] J. Hodas and J. Polakow have recently made similar observations.

6 Discussion

One of the main lessons that we have learnt from writing programs in Lygon is that the programming methodology seems to have some significant differences from Prolog. In particular, resource-sensitivity is critical. For example, in the blocks world program in § 3.3, the information that we wish to extract from the computation is the final state of the blocks. This is given explicitly by the state of the linear predicates rather than by any particular answer substitution. This suggests that programming in Lygon would seem to be *resource-oriented* rather than just *substitution-oriented* — it is easy to envisage input and output as formulae, rather than just as terms.

More examples in which the linear logic basis of Lygon facilitates elegant solutions to problems are given in [21]. Included are modelling exceptions, parsing visual diagrams using multiset grammars and solving bin-packing problems. Other possibilities for further investigation include the use of Lygon as an object-oriented language (*cf.* [2]) and the use of Lygon for agent-based applications which combine planning and concurrency.

The operational model on which the current implementation [22, 21] of Lygon is based represents only one choice

[14] See also Footnotes 5 and 12.

among many. For example, many variations on the basic input/output model are possible. One possibility would be a concurrent model in which different multiplicative branches of a search, *e.g.*, created by a $\otimes$R, would have access to the same collection of resources, thereby necessitating explicit communication between branches as resources be consumed. Other variations would include ones motivated entirely by efficiency concerns.

Acknowledgements

We thank the referees and many colleagues for their comments this work. The partial support of the Australian Research Council, the Collaborative Information Technology Research Institute, the Centre for Intelligent Decision Systems and the UK EPSRC is gratefully acknowledged. Michael Winikoff is supported by an Australian Postgraduate Award.

References

1. J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *J. Logic Computat.* 2(3), 1992.
2. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Gen. Comp.*, 9:445–473, 1991.
3. A. Bonner and L. McCarty. Adding Negation-as-Failure to Intuitionistic Logic Programming. Proc. NACLP, 681-703, Austin, October, 1990.
4. N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.
5. P. Dung. Hypothetical Logic Programming. Proc. 3rd. International Workshop on Extensions of Logic Programming 61-73, LNCS, Springer, 1992.
6. M. van Emden and R. Kowalski. The Semantics of Predicate Logic as a Programming Language. *J.ACM* 23:4:733-742, 1976.
7. J.-Y. Girard. Linear Logic. *Theoret. Comp. Sci.* 50, 1-102, 1987.
8. S. Hanks and D. MacDermott. Nonmonotonic Logic and Temporal Projection. *Artif. Intell.* 33:3:379-412, 1987.
9. J. Harland and D. Pym. A note on the implementation and applications of linear logic programming languages. *Australian Computer Science Communications* 16(1), 647-658, 1994.
10. J. Harland, D. Pym and M. Winikoff. Programming in Lygon: a system demonstration. This volume.
11. J. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation.* PhD thesis, University of Pennsylvania, 1994.
12. J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Inform. and Computat.* 110:2:327-365, 1994.
13. S.C. Kleene. *Mathematical Logic.* Wiley and Sons, 1968.
14. N. Kobayash and A. Yonezawa. ACL - A Concurrent Linear Logic Programming Paradigm. Proc. ILPS'93, D. Miller (ed.), 279-294, MIT Press, 1993.
15. D. Miller. A multiple-conclusion meta-logic. Proc. *LICS'94*, 272–281, IEEE, 1994.
16. D. Miller, G. Nadathur, F. Pfenning and A. Ščedrov. Uniform Proofs as a Foundation for Logic Programming. *Ann. Pure Appl. Logic* 51 (1991) 125-157.
17. D. Pym and J. Harland. A Uniform Proof-theoretic Investigation of Linear Logic Programming. *J. Logic Computat.* 4:2:175-207, 1994.
18. J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask and J. Harland. *The Aditi Deductive Database System.* VLDB J. 3:2:245–288, 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.
20. D.S. Warren. Programming the PTQ Grammar in XSB. in *Applications of Logic Databases*, Raghu Ramakrishna (ed.), Kluwer Academic, 1994.
21. M. Winikoff. Lygon home page (subject to alteration). `http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html`.
22. M. Winikoff and J. Harland. Implementing the linear logic programming language Lygon. In: J. Lloyd (ed.), *Proc. ILPS'95*, 66–80, MIT Press, 1995.
23. M. Winikoff and J. Harland. Some applications of the linear logic programing language Lygon. *Australian Computer Science Communications*, 18(1), Kotagiri Romamohanarao (editor), 1996.

This article was processed using the LaTeX macro package with LLNCS style.