PROGRAMMING IN LYGON: AN OVERVIEW

James Harland
Department of Computer Science
Royal Melbourne Institute of Technology
GPO Box 2476V
Melbourne, 3001
Australia

jah@cs.rmit.edu.au
phone: +61 3 660 2045
fax: +61 3 662 1617
http://www.cs.rmit.edu.au/~jah

David Pym
Department of Computer Science
Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS
England, UK
pym@dcs.qmw.ac.uk
phone: +44 171 975 5237
fax: +44 181 980 6533
http://www.dcs.qmw.ac.uk/~pym

Michael Winikoff
Department of Computer Science
University of Melbourne
Parkville, 3052
Australia
winikoff@cs.mu.oz.au
phone: +61 3 344 9147
fax: +61 3 348 1184
http://www.cs.mu.oz.au/~winikoff

Abstract

Recently, there has been much interest in the derivation of logic programming languages based on linear logic, a logic of resource-consumption. Such languages provide a notion of resource-oriented programming, often leading to programs that are more elegant and concise than their equivalents in languages, such as Prolog, based on classical logics. We discuss, with examples, the design, implementation and applications of *Lygon*, a linear logic programming language. Lygon is based on a proof-theoretic analysis of which occurrences of the linear connectives provide an adequate basis for programming. 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. Indeed, it appears that resource-sensitivity leads to significant differences between the natural programming methodologies in Lygon and Prolog. 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 *global variables*, a theoretically transparent notion of *state*, *mutual exclusion operators* and various constructs for manipulating clauses. All of these follow from the basis of Lygon in linear logic and *do not require* extra-logical features for their definition.

Keywords: Linear logic, resource-oriented programming, applications.

PROGRAMMING IN LYGON: AN OVERVIEW

James Harland
RMIT
Australia

David Pym
Queen Mary & Westfield College
University of London

Michael Winikoff
University of Melbourne
Australia

Abstract

Recently, there has been much interest in the derivation of logic programming languages based on linear logic, a logic of resource-consumption. Such languages provide a notion of resource-oriented programming, often leading to programs that are more elegant and concise than their equivalents in languages, such as Prolog, based on classical logics. We discuss, with examples, the design, implementation and applications of *Lygon*, a linear logic programming language derived from a proof-theoretic analysis of which occurrences of the linear connectives provide an adequate basis for programming. 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 *global variables*, a theoretically transparent notion of *state*, *mutual exclusion operators* and various constructs for manipulating clauses. All of these follow from the basis of Lygon in linear logic and *do not require* extra-logical features for their definition.

1 Introduction

Logic programming languages are based upon the observation that certain sequents of classical logic can be interpreted as a program together with a goal. This has been done by giving an appropriate interpretation to the mechanism of proof-search for a given fragment of formulae known as Horn clauses, giving rise to general-purpose programming languages such as Prolog. For example, from a logical point of view [18], 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 (an answer substitution) together with a proof of the sequent $\Gamma \vdash \phi[t/x]$. This view of logic programming is examined logically in [22].

For a given extension of classical logic, it is a natural to look for a corresponding extension of the logic programming paradigm which makes available to the logic programmer the features of the extended logic. One such extension is *linear logic*, a resource-sensitive refinement of classical logic. Linear logic has many applications in computer science, including the semantics of concurrency, natural language processing and updates [1, 14].

When looking for logic programming languages, the fundamental question of what is meant by a logic programming language naturally arises. 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, [20]). It is natural, then, to consider design principles that limit the non-determinism in the search for proofs.

One such principle is that of *goal-directed provability*. Here the shape of the proof is determined by the goal, so that the program performs the function of supplying the context of the proof, rather than determining the search strategy needed to find it. Once we have determined an appropriate notion of goal-directed provability, we can then 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 [20], higher-order logics [20] and linear logic [21, 14, 27]. 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 based on classical logic such as Prolog. 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 (unlike in Prolog, where such cycles either cause non-termination or require some significantly more complex programs and/or the use of extra-logical constructs such as assert and retract). 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*, building on the theoretical framework provided by Pym and Harland [21]. In common with other linear logic programming languages, Lygon by default allows clauses to be used exactly once in a computation, which allows the elegant specification of programs which are somewhat more awkward in a language like Prolog. However, 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 pure logic programs are retained in Lygon, and extended with new ones based on linear logic. Such features include *global variables*, a *notion of state*, *mutual exclusion operators* and various constructs for manipulating clauses. Note that all of these follow from the basis of Lygon in linear logic and *do not* require any extra-logical features for their definition.

2 Foundations

We briefly review the main ideas of linear logic from a proof-theoretic perspective, this view being the basis of our analysis of logic programming. We illustrate the ideas with an extended example. (The logically sophisticated reader may wish to skip over this Section.) We briefly consider how an analysis of goal-directed provability applies to linear logic [21].

2.1 Linear logic

The *sequent calculus* [7] is a presentation of the proofs of classical logic in which sequents of the form $\Gamma \vdash \Delta$ are manipulated. Such sequents are to be read as "the conjunction of the formulae from the set Γ entails the disjunction of the formulae from the set Δ". In this setting, there are two alternative presentations of conjunction: the rule for introducing conjunction on the right hand side of a consequence can be either

$$\frac{\Gamma \vdash \phi_1, \Delta \quad \Gamma \vdash \phi_2, \Delta}{\Gamma \vdash \phi_1 \wedge \phi_2, \Delta} \tag{1}$$

or

$$\frac{\Gamma_1 \vdash \phi_1, \Delta_1 \quad \Gamma_2 \vdash \phi_2, \Delta_2}{\Gamma_1, \Gamma_2 \vdash \phi_1 \wedge \phi_2, \Delta_1, \Delta_2} \tag{2}$$

The difference is that in the former is *additive* — the antecedent and succedent of the conclusion (respectively the left and right hand sides of $\vdash$) are copied to each premiss — and the latter is *multiplicative* — the antecedent and succedent of the conclusion (respectively Γ_1, Γ_2 and Δ_1, Δ_2) are divided between the two premisses. A similar situation obtains for the other connectives.

An important property of classical proofs is that they admit the structural rules of *weakening*, allowing formulae to be added to each side of a sequent, and *contraction*, allowing multiple copies of formulae to be removed.[1] In the presence of these rules, the two forms of conjunction given in (1) and (2) are interderivable. In linear logic, these structural rules are available for use only with special classes of formulae: one can weaken and contract on the left with formulae of the form $!\phi$, read as "of course ϕ" and on the right with formulae of the form $?\phi$, read as "why not ϕ". Consequently, linear logic distinguishes two kinds of conjunction: $\&$, read as "with", for (1) and $\otimes$, read as "tensor", for (2).

A consequence of the definition of linear logic is that the provable consequences are dependent upon the number of copies of a formula that are available. A commonly given example of this situation is that of buying food with money. If a single ecu is denoted by the predicate `ecu`, then the property of having two ecus can be specified by the conjunction of `ecu` with itself, *i.e.*, `ecu` $\otimes$ `ecu`. In classical logic, this would have to be represented by `ecu` $\wedge$ `ecu`, which is classically equivalent to `ecu`, so that having one ecu would be the same as having two ecus. If exactly one ecu can be used to buy exactly one drink, this is clearly nonsensical. However, in linear logic, `ecu` $\otimes$ `ecu` and

[1] That is, when proofs are constructed in a top-down manner — from leaves to root.

ecu are not equivalent. For this reason, linear logic is often described as a logic of resources.

Conditional assertions, such as the conversion of an ecu into a drink or into sweets, can be represented using linear implication. The rules for buying a drink and for buying sweets are then, respectively,

$$\mathsf{ecu} \multimap \mathsf{drink} \quad \text{and} \quad \mathsf{ecu} \multimap \mathsf{sweets}$$

where $\multimap$ is the linear implication. The rule that a person is happy if he has a drink and has sweets is then

$$(\mathsf{drink} \otimes \mathsf{sweets}) \multimap \mathsf{happy_person}$$

Using these rules, we can derive happy_person from ecu $\otimes$ ecu but not from ecu.

However, reasoning in linear logic need not be totally different from classical logic. Classical logic can be recovered by a controlled reinstatement of weakening and contraction. A formula beginning with ! in an antecedent or with ? in a succedent behaves classically: such a formula can be copied arbitrarily many times (contraction) and can be ignored in a proof (weakening) if desired.[2] Hence whilst ecu corresponds to *exactly* one ecu, ! ecu corresponds to an arbitrary number of ecus, including 0. In this way we can think of a formula $!F$ in linear logic as representing an unbounded resource, *i.e.*, one that can be used as many times as we like. Thus classical logic can 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 ! can be found if we return to the drinks and sweets example. As written above, the rules for turning a ecu 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 ecus have to be converted into drinks or sweets, or that such a conversion cannot take place twice or more. Hence a better representation of the problem is to precede each of the rules with an of course, so that we have

$$!(\mathsf{ecu} \multimap \mathsf{drink})$$
$$!(\mathsf{ecu} \multimap \mathsf{sweets})$$
$$!((\mathsf{drink} \otimes \mathsf{sweets}) \multimap \mathsf{happy_person})$$

[2] Thus, for sequents consisting entirely of such *exponentiated* formulae, the additive (*e.g.*, (1)) and multiplicative (*e.g.*, (2)) rules for the connectives are again interderivable.

From these rules and the information ecu $\otimes$ ecu we can then derive information such as drink $\otimes$ drink, or that from four ecus we can have two happy people. This process can be thought of as "localizing" 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 resides in the distinction drawn between the additive and multiplicative connectives. As we have seen, 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 can be thought of as an accumulator of resources: $p \otimes q$ means the resource p together with the resource q, so that ecu $\otimes$ sweets signifies that we found the resources necessary to obtain both one ecu and a packet of sweets. The other conjunction, &, is not accumulative: $p \& q$ expresses that both p and q can be derived, but by using the same resources. For example, from ecu it is possible to derive both drink and sweets, and so we can derive drink & sweets from a single ecu. Hence & represents an *internal choice*; as both formulae can be derived, we are free to choose either one.

Similarly, 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 is in many ways similar to the usual classical disjunction. The other disjunction is the dual of $\otimes$, and is often written $\parr$, pronounced as "par". If $p \otimes q$ represents an accumulation of the resources p and q, the dual of this notion can be thought of as the accumulation of *promises to provide resources*, or debts. In this sense, $p \parr q$ 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 $\top$ (which can be thought of as generalizations of the boolean value true) and for $\parr$ and $\oplus$ are written as $\bot$ and $\mathbf{0}$ (which can be thought of as generalizations 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 $F^{\bot}$. The following de Morgan laws all hold:

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

Hence linear logic has many symmetric properties reminiscent of classical logic. A common example of these symmetries is the following restaurant menu:[3]

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

From the point of view of the customer, the menu corresponds to the formula

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

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

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

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

There is far more to linear logic than can be discussed in this paper; for a more complete introduction see, for example, [8, 24, 23, 1].

2.2 Goal-directedness in linear logic

In order to exploit the features of linear logic in a logic programming context, it is necessary to have some criterion for determining what a logic programming language is in a given logic. 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 [20, 21].

This notion has been used to identify logic programming languages in intuitionistic logic and higher-order logic [20]. A particular advantage of using this approach, rather than, say, looking to apply the resolution principle to linear logic, is that it attempts to define a principle which is independent of a given logic, and hence provides a valuable theoretical framework for the study and comparison of the notion of, and languages for, logic programming. This means that we can use the proof-search strategy which will be used to implement the language to determine the class of formulae which will serve as a logic programming language. It is this proof-search strategy which can be taken to characterize the difference between a general theorem prover and a logic programming language. Thus this approach has the potential to exploit the properties of the given logic as much as possible, imposing as few limitations as possible on the resulting language.

We briefly review the main idea. Recall our description, in the introduction, of a logic program together with a goal as a sequent. It follows that if the shape of the proof-search is to be determined by the structure of the goal, then the operational interpretation of the goal, *i.e.*, the way in which the goal is processed by the interpreter, must correspond to the meaning determined by the rules of the sequent calculus. In particular, the rules that introduce structure to the goal will determine the way the search proceeds. Hence each connective of the goal can be considered to determine a search-instruction. To see this, consider some examples. Let Γ be a fixed choice of (classical) program.

- $\phi_1 \wedge \phi_2$ is a consequence of Γ *only if* both ϕ_1 and ϕ_2 are consequences of Γ. It should be clear that this amounts to a reading of this instance of the right rule for classical conjunction,

  $$\wedge\mathrm{R} \quad \frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \wedge \phi_2}$$

  as a reduction operator, from conclusion to premisses [16].[4]

- $\phi_1 \vee \phi_2$ is a consequence of Γ *only if* either ϕ_1 or ϕ_2 is a consequence of Γ. It should be clear

[3] The origin of the name Lygon is gastronomic.

[4] Note that, for simplicity, we consider here only single-conclusion sequents.

that this amounts to a reading of this instance of the right rule for classical disjunction,

$$\vee\mathrm{R} \quad \frac{\Gamma \vdash \phi_i}{\Gamma \vdash \phi_1 \vee \phi_2} \quad i = 1, 2$$

as a reduction operator.

- Implication is a little more interesting. The goal $\phi \supset \psi$ is a consequence of Γ *only if* ψ is a consequence of $\{\Gamma, \phi\}$, *i.e.*, Γ augmented with the formula ϕ. This transfer of a formula from the goal to the program imposes a constraint on the shape of implicational goal formulae: ϕ must be of the form permitted in programs. This shape is constrained by the requirement that the search strategy be minimally non-deterministic. As usual, the operational interpretation corresponds to the right rule of the sequent calculus, read as a reduction operator,

$$\supset \mathrm{R} \quad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \supset \psi},$$

subject to the constraint on the shape of the goal.

- Finally, consider a goal containing a logical variable, expressed as $\exists x \,.\, \phi$. Such a goal is provable from Γ *only if* there is a term t such that $\phi[t/x]$ is provable from Γ. Again, this corresponds to the appropriate right rule of the sequent calculus, read as a reduction operator,

$$\exists \mathrm{R} \quad \frac{\Gamma \vdash \phi[t/x]}{\Gamma \vdash \exists x \,.\, \phi}.$$

Similar interpretations of other connectives as search-instructions are possible. Note that many provability relations do not satisfy these conditions. To do so, a sequent calculus for the relation must enjoy a certain strength of *permutation theorem*, *q.v.* [20, 21]. Note also that the conditions do not fully specify an interpreter: they give no instruction as to how to solve atomic goals. A typical choice is a form of resolution, or backchaining, which amounts to a specialized combination of the left rules for each of the connectives permitted to occur in programs. This delicate notion is discussed at some length in [21].

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. In order to do this, according to the above approach, we need to determine an appropriate notion of goal-directed proof.

We follow the direction of [21], which provides as broad as possible an interpretation of goal-directedness and thereby allows the broadest possible linear logic programming language.

One of the key points to note is our approach to left rules. Roughly, we want to be able to incorporate all appropriate left rules into one particular inference, *i.e.*, the resolution rule. So we need to ensure that the formulate in a program are always in a particular clausal form. In [21], a mapping $[-]$ was introduced which converts a multiset of linear definite formulae (see below) into a multiset of clauses in such a way that logical consequence is preserved. For a sequent of the form $[\mathcal{P}] \vdash \mathcal{G}$, where the program $\mathcal{P}$ is a multiset of definite formulae and the goal $\mathcal{G}$ is a multiset of linear goal formulae (see below), the only left rule required in order to determine the consequence is the resolution rule [21], provided the uniform proof strategy is followed.[5]

The details of the analysis that identifies the classes of definite formulae and goal formulae for which this arrangement works can be found in [21]. The main point to note is that this analysis derives the following class of formulae as a logic programming language:

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

where A ranges over atomic formulae.[6] This class of formulae will be compared with that used in the language Forum [19] in § 5.

To understand the mapping $[-]$ consider a few examples. Firstly, $[p \otimes q] =_{\mathrm{def}} \{\, p, q \,\}$, 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 uniform proof. Secondly, $[(\forall x \,.\, (p \multimap q(x))) \otimes (r \multimap s)] =_{\mathrm{def}} \{\, p \multimap q(x), r \multimap s \,\}$. Finally, $[!D] =_{\mathrm{def}} \{\, !\bigotimes_{C \in [D]} C \,\}$. Full detail is provided in [21].

Just as in the classical case, the goal formulae G have an operational interpretation. The interpretation of $D \multimap G$ is similar to that of classical implication: note that the formula that moves from the goal to the program is of the form $[D]$. Similarly, the

[5] The Lygon interpreter implements this idea via one-sided sequents.

[6] Slight extensions to this class are possible. However, such extensions are not proof-theoretically significant. Note that definite formulae of the form $G \multimap (D_1 \parr D_2)$ can be written as $(G \multimap \bot) \parr D_1 \parr D_2$. $D ::= G^{\perp}$ and $G ::= D^{\perp}$ are implicit.

existential goal behaves as above. Of the other goal formulae, we describe a selection. As before, their operational meanings are characterized by the right rules of linear sequent calculus [21]. Let Γ denote a fixed choice of D-formulae (to which we assume the mapping $[-]$ to clausal form has been applied).

- $G_1 \otimes G_2$ 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 (G is a consequence of Γ *only if* G is a consequence of $\Gamma, [D]$. The (clausal form of the) formula D must be added to the program Γ.

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

- $!G$ is a consequence of Γ *only if* G is a consequence of Γ and each formula of Γ is of the form $!D$, *i.e.*, all of the resources required to prove G are unbounded.

The reader is invited to consult [21, 29] for discussions of the remaining connectives.

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, 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 [14] 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.[8] In this way 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 is beyond the scope of this paper and may be found in [29].

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 Computational features and examples

In this section, we discuss various programming techniques which distinguish Lygon from Prolog (see also [12, 30]), including several examples of Lygon programs.

3.1 Lygon syntax

The current implementation of Lygon (Version 0.4) is an interpreter written in BinProlog[9].

The interpreter comprises about 450 lines of code, including comments and whitespace. It currently supports a limited form of program — all program clauses are implicitly nonlinear. This is not a severe limitation since linear clauses in the program can be negated and added to the goal as a simple syntactical preprocessing step.

We use the following mapping $\Rightarrow$ between logical connectives and ASCII:

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

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

[8] The so-called "input/output model of resource con- sumption".

[9] Version 0.4 of Lygon is available upon request from the authors by email or via the World Wide Web from the URL `http://www.cs.mu.oz.au/~winikoff/lygon/lygon.html`.

Predicates and variables are as in Prolog. The Lygon equivalent of the Prolog `:-` is written as `<-`.

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 the 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 $C \,\&\, \top$. 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 $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ and both $\mathcal{P}_1 \vdash G$ 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, and there are many applications of graph problems, such as laying cable networks, evaluating dependencies, designing circuits and optimization problems. In this section we show how graph problems can 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 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 [28], or a bottom-up system such as Aditi [26]. 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, 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 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)`.

```
Lygon graph # path(a,_,P).
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

```
Lygon graph # path(X,X,P).
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))`., 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 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 [14] 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 code to do this assertion is below.

```
perm([X|L1],L2) <- neg list1(X) #
                   perm(L1,L2).
perm([], L2) <- perm1(L2).

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

Note that this can be thought of as a data conversion, from a list format into a predicate format. 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 succeed only if there are no linear resources so that 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 can 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 ordered in which items are represented is not important. Lists have the property that access to elements is linear in the length of the list. 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. Hence, 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 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 behaves as follows:

```
Lygon go(P).
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 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 travelling 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.

3.3 Representing states and actions

As mentioned above, 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 two cells only, which can be represented by two instances of the predicate m, with the first argument being the address and the second the contents of the cell. Thus 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 $m(2,x) \otimes (m(2,x)\ (\ G)$, where G is to be executed after the read. The states in this computation are that $m(2,x)$ is unified with $m(2,t_2)$, the latter atom is deleted from the program, and then added again via the (connective, before G is executed. Note that a similar sequence occurs for the goal $m(2,x) \otimes m^{\perp}(2,x)$.[10] In a similar manner, writing the value t' into the memory can be achieved by the goal $m(2,x) \otimes (m(2,t')\ (\ G)$, where it is possible that t' can contain x, and hence 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.

Note also the duality between $D_1, D_2 \vdash G_1 \otimes G_2$ and $D_1 \parr D_2 \vdash G_1, G_2$. The former case can be thought of as having the resource-passing paradigm imposed by the goal, whereas in the latter case it is specified by the programmer, thereby giving the programmer explicit control over the allocation of resources.

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

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 [9] is a prototypical example of a problem 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 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 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.

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

Note that the rules for `load` and `shoot` can be written in the following manner:[11]

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

This can be thought of as allowing non-atomic formulae as the heads of clauses. When written 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`, 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 into order 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 [14], lacking this facility, the second clause, `shoot # (dead * unloaded) <- alive * loaded.`, would have to be written as two special cases. Although the language Forum [19] 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 [3].

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 the following:

- `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.

```
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.

[11] Note that the first clause `load # loaded <- unloaded.`, can be considered an abbreviation for `( bot <- unloaded ) # loaded # load.`. See also the footnote to the grammars defining definite and goal formulae.

```
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.
```

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 # take(c) # put(c) # take(a)
      # stack(a,b) # showall(R))
      * print(R) * nl.
```

```
Lygon go.
[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 some actions, as 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.

As in the previous example, by allowing clause heads to contain multiple formulae (in the manner of LO [3]) 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. In particular, both examples exploit the positive occurrences of `#` in program clauses to achieve concise, elegant code.

3.4 Global variables

One of the subtle features of linear logic is that as formulae cannot be copied unless they commence with a !, variables which appear outside the scope of a ! cannot be standardized apart, and hence variable names can persist across clauses. As a result, 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 such occurrences cannot be standardized apart, as is the case in Prolog. 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 both of the formulae $\forall x.\,(p(x) \wedge q(x))$ and $(\forall x.\,p(x)) \wedge (\forall x.\,q(x))$ are equivalent, and hence in the case of linear logic, the substitution generated by a unification may need to be applied to several 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. In particular, it seems natural to maintain a pointer to a common location for each occurrence of a global variable, so that when the variable is instantiated, the common value is readily available. This feature would be particularly useful for large shared objects, such as dictionaries, as each occurrence of the instantiated variable would act as a pointer to the dictionary: the dictionary need not be passed around from procedure to procedure.

Another useful application of global variables is message passing. In this case the variable is 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 com-

municate 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 $!p(x,y) \vdash p(a,b) \otimes p(c,d)$.

3.5 Mutual exclusion

The connective & can be thought of as specifying internal choice; in other words, when faced with a 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. Hence this can be thought of as a mutual exclusion operator, in that the use of one formulae precludes the use of the other. In an operational sense, it seems most natural to implement this via backtracking, which means that this 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.

3.6 Preserving context

As noted above, one of the key features of this language is the operational 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.

3.7 Counting clauses

Another example in which the properties of linear logic make a significant simplification is in the following problem, which has been discussed as a motivation for the use of embedded implications in the presence of Negation-as-Failure [4, 5].

The problem can be stated thus: 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, though, 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)).
```

We then run the program with a goal such as

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

which returns the answer `X = even`, whilst the goal

```
?- neg count(even) # neg r(1) #
   neg r(2) # neg r(3) # check(X).
```

returns the answer `X = odd`.

Whilst this is hardly a deep example, it does suggest that a linear version of aggregrates may be interesting, as counting exercises such as the one above are very simple to code in Lygon. More generally, the resource-sensitive nature of linear logic makes the notion of meta-programming in linear logic particularly interesting.

Another potential application for "counting" applications is in solving puzzles such as the famed SEND + MORE = MONEY. Tarau [25] has shown how an improvement of the Prolog solution can be given by using a linear predicate for the digits. Thus each digit can be used only once, and this reduces the number of impossible combinations of digits that need to be considered, such as having all of the digits equal to 2. Consider also the possible application of the linearity of predicates in Lygon is games, particularly card games, in which each card must be played exactly once. However, a full solution to this kind of problem would presumably need a committed-choice language (backtracking during games such as poker is not generally permitted).

4 Other programming constructs

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 can occur independently of whether or not other predicates, such as `edge` are exhausted or not. Whilst we can use a combination of `one` (which can 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 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.

As we have seen, there are other similar constructs which would also prove useful, 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 travelling salesman problem above, this kind of facility allows a very powerful programming methodology.

Given the natural way in which planning problems can 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, *i.e.*, 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.

5 Other linear logic programming languages

There are various other logic programming languages based on linear logic. These include LO [3], LinLog [2], ACL [17], $\mathcal{LC}$ [27], Lolli [14, 13] and Forum [19].

LinLog is based on the idea of focusing proofs, which is an alternative formulation to one based on goal-directed proofs, and some of the results of this line of work have been used to make the Lygon interpreter more efficient. However, LinLog has not, to the authors' knowledge, been implemented.

LO is an early system, in which object-oriented features are introduced into a logic programming language by means of the connective $\parr$. There are a variety of interesting programs which can be written in this language; however, it should be noted that the class of formulae used in LO is subsumed by Lygon.

ACL was designed for concurrent programming, using the formalism of linear logic. Hence the operational model is somewhat different from that of Lygon, although the class of formulae is more restricted than that of Lygon. In particular, the only uses of the connective $\otimes$ are such that no splitting, lazy or otherwise, need be done.

$\mathcal{LC}$ is a more recent development, again designed for concurrency, using a class of formulae which is a subclass of the Lygon class. In particular, the log-

ical connectives must permute over each other, an assumption which is not made in Lygon.

The linear logic programming language which most resembles Lygon is Lolli, which is also based on a notion of goal-directed proof, but for a somewhat more restrictive class of formulae than Lygon. In particular, the sequents in Lolli are single-conclusioned, which effectively means that the connective $\parr$ cannot be used. In addition, there are some restrictions on the placement of the ! connective which do not apply in Lygon. So the class of formulae in Lygon fully subsumes that of Lolli (in the language of [10], this was not so; that fragment, as pointed out in [14], is different from the class of formulae used in Lolli).

Forum is a more recent development which is an amalgamation of linear and higher-order logics. Forum is designed as a specification language, in which various logical and programming systems may be expressed, such as the inference rules of the sequent calculus, the operational semantics of a functional language, and hence the motivation behind it is somewhat different to ours. Nevertheless, there is a notion of goal-directed proof in Forum which is more restricted than that of Lygon. In particular, whilst sequents in Forum may contain multiple conclusions, all of the right rules must permute over each other (similar to $\mathcal{LC}$, a restriction that is not made in Lygon). Hence Forum may contain only asynchronous connectives in conclusions. Forum is complete for full classical linear logic (CLL) in the sense that any formula of CLL is linearly equivalent to a formula permitted in Forum. However, regarding full CLL as a logic programming language in this way raises interesting questions about what is meant by a logic programming language. If a sequent must first be translated into a Forum sequent, the operational semantics of its execution is not necessarily goal-directed with respect to its original connectives. So although Forum is complete with respect to CLL and uses only uniform proofs, comparison with the hereditary Harrop-like fragment used in Lygon is misleading: we have been concerned to consider resolution proofs of explicit occurrences of connectives. Further development of these issues is beyond the scope of this paper (although see, for example, [31]).

6 Discussion

The development of the Lygon interpreter is continuing, particularly with regard to issues of optimization. There is a large and growing body of literature on optimization of logic programming languages, as well as a significant body of work on proof search in linear logic, which we hope to exploit. In addition, we plan to implement a compiler for the language once the interpreter has matured somewhat; and we have experimented with various optimization techniques on a variety of programs. Work is also already underway on a toolkit for debugging support for Lygon.

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

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, the notion of resources is critical, as is perhaps not unexpected. For example, in the blocks world program above, the information that we wish to extract from the computation is the final state of the blocks, which is given explicitly by the state of the linear predicates, and not by any particular answer substitution. This suggests that a more meaningful way to answer the goal is to merely output this final state, rather than being concerned about finishing off the proof in linear logic (which we know may be done by using a $\top$). In other words, the important information is the state of the resources, and not any particular substitution.

Similar tendencies can be found in the path program. There, the edges of the graph are represented in a predicate, whereas the cycles, *i.e.*, the answers to the goal, are, as is traditional, given as terms, and returned via an answer substitution. One could argue that the linear resources given to the program can be viewed as the input, which suggest that the output should be a similar multiset of formulae of a given form, *i.e.*, clauses. In such a scenario the cycles would be given as a subgraph of the original one, and in the same data structure, rather than returned in a list through a variable. In other words, programming in Lygon would seem to be *resource-oriented* [15] rather than *substitution-oriented*, in that it is easy to envisage input and output as formula, rather than terms. Note that the traditional T_P semantics of logic programs [6] can be thought of as treating the program as an operator, and the facts of the program as input,

with the output being the program's semantics.

Another perspective on this paradigm is given by transactions. Linear logic has been used to describe the semantics of concurrency, as well as updates. Hence it would seem a good candidate for the incorporation of transaction processing features into logic programming languages (or perhaps more appropriately, deductive databases). Viewing inputs and outputs as resources in the manner described above simplifies such an incorporation, in that it seems natural to regard a transaction as a mapping between states subject to certain criteria, which will presumably be given by an appropriate collection of formulae of linear logic. Hence a logical interpretation of transactions, and perhaps other dynamic features such as active databases and agents, seems to be made simpler in a resource-oriented framework.

7 Acknowledgements

We thank John Crossley, Graham White and many other colleagues for interesting discussions related to this work. The 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 (APA).

References

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

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

[3] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. Proc. ICLP'90, 496-510, Jerusalem, June, 1990.

[4] A. Bonner and L. McCarty. Adding Negation-as-Failure to Intuitionistic Logic Programming. Proc. NACLP, 681-703, Austin, October, 1990.

[5] P. Dung. Hypothetical Logic Programming. Proceedings of the Third International Workshop on Extensions of Logic Programming 61-73, Bologna, February, 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] G. Gentzen. Untersuchungen über das logische Schliessen. *Math. Zeit.* 39, 176-210, 405-431, 1934.

[8] J.-Y. Girard. Linear Logic. *Theoret. Comp. Sci.* 50, 1-102, 1987.

[9] S. Hanks and D. MacDermott. Nonmonotonic Logic and Temporal Projection. *Artif. Intell.* 33:3:379-412, 1987.

[10] J. Harland and D. Pym. The Uniform Proof-theoretic Foundation of Linear Logic Programming (Extended Abstract). Proc. ILPS'91, V. Saraswat and K. Ueda (eds.), 304-318, MIT Press, 1991.

[11] J. Harland and D. Pym. Resolution in Fragments of Classical Linear Logic (Extended Abstract). Proc. LPAR, A. Voronkov, (ed.), 30-41, MIT Press, 1992.

[12] 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.

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

[14] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Inform. and Computat.* 110:2:327-365, 1994.

[15] J.-M. Jacquet and L. Monteiro. Towards Resource Handling in Logic Programming: the PPL Framework and its Semantics. Proc. JELIA'94 379-399, York, 1994.

[16] S. Kleene. Mathematical Logic. Wiley and Sons, 1968.

[17] N. Kobayash and A. Yonezawa. ACL - A Concurrent Linear Logic Programming Paradigm. Proc. ILPS'93, D. Miller (ed.), 279-294, MIT Press, 1993.

[18] J. Lloyd Foundations of Logic Programming. Springer-Verlag, Berlin, 1984.

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

[20] 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.

[21] D. Pym and J. Harland. A Uniform Proof-theoretic Investigation of Linear Logic Programming. *J. Logic Computat.* 4:2:175-207, 1994.

[22] D. Pym and L. Wallen. Logic Programming via Proof-valued Computations. In: ALPUK92, Proc. 4th U.K. Conference on Logic Programming, K. Broda (editor), 253-263. Springer-Verlag, Workshops in Computing Series, 1992.

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

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

[25] P. Tarau. The BinProlog Manual. Distributed with the BinProlog system.

[26] 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.

[27] 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.

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

[29] M. Winikoff and J. Harland. Implementing the Linear Logic Programming Language Lygon. To appear: Proc. ILPS'95, J. Lloyd (ed.), MIT Press, 1995.

[30] M. Winikoff and J. Harland. Some Applications of the Linear Logic Programming Language Lygon. To appear: Proc. ACSC'96, Melbourne, January, 1996.

[31] M. Winikoff and J. Harland. Characterising Logic Programming Languages. Technical Report 95/26, Dept. of Computer Science, University of Melbourne, 1995.