A Note on the Implementation and Applications of Linear Logic Programming Languages

J.A. Harland
University of Melbourne, Australia

D.J. Pym
University of Birmingham, UK

Abstract

Traditional logic programming languages, such as Prolog, have their origins in classical logic. Recently it has been shown how logic programming languages can be based on linear logic, a logic designed with bounded resources in mind. In this paper we discuss the implementation issues for the linear logic programming language *Lygon*, and describe some of the novel features of the language. These include global variables, a notion of state, mutual exclusion operators and constructs for the manipulation of clauses. We also show how a bin packing problem can be solved simply and elegantly in Lygon.

1 Introduction

In recent work [6, 4, 5, 11], the present authors have shown, via an analysis of those fragments of Girard's *linear logic* [3] for which a certain notion of goal-directed provability completely characterizes logical consequence, how certain fragments of linear logic provide an appropriate logical (proof-theoretic) foundation for logic programming languages.[1]

In this paper, we investigate how such languages give simple, logical accounts of a variety of programming techniques that must be coded explicitly in languages such as Prolog. We briefly describe a particular implementation strategy for such languages, which forms the operational basis for the logic programming language *Lygon*[2]. The language Lygon has, *inter alia*, several novel features for a logic programming language: global variables; a notion of state; a mutual exclusion operator and some useful constructs for manipulating clauses. We describe these features and their use, showing, in particular, how such features extend the logically characterized functionality of logic programming languages. That these features arise as properties of the logic programming language Lygon is directly attributable to the basis of the language in linear logic.

A key feature of linear logic programming languages is that once a clause has been used, it must be deleted, and every clause in the program must be be used *exactly* once. It is this particular property that gives rise to many of the novel features of Lygon. There are certain exceptions to this rule, associated with formulae that commence with the *exponentials*, ! (of course) and ? (why not). Such formulae can be used any number of times (including 0) in a proof; such usage corresponds to the use of Horn clauses in classical logic programming languages, such as Prolog. Specifically, the use of this particular connective enables Lygon to encode Horn clauses, and hence this language is a strict generalization of (pure) Prolog.

Some other approaches to the use of linear logic in a logic programming framework include those of Hodas and Miller [7] and Andreoli and Pareschi [1]. However, it seems that no other linear logic programming language has the breadth of features found in Lygon.

This paper is organized as follows: in § 2 we discuss some of the relevant features of linear logic; in the following section we describe the language Lygon; § 4 contains an overview of the operational aspects of Lygon; in § 5 we describe some of the more novel and useful features of the language; in § 6, we give an example of the use of Lygon for a bin packing problem; finally, in § 7, we discuss our conclusions and some prospects for further work. For the convenience of the general reader, we give the classical and linear sequent calculi in the appendices.

[1] Linear logic is an important example of the family of *relevance logics* [12].

[2] This name is not acronym; its origin is gastronomic.

2 Logical Preliminaries

An elegant and convenient proof system for classical logic is provided by the *sequent calculus* [2, 8, 3].[3]

Each derivation in the calculus yields a sequent $\Gamma \longrightarrow \Delta$, which can be thought of as stating that if the conjunction of the formulae in (the antecedent) Γ is true, then the disjunction of the formulae in (the succedent) Δ is true. For each connective, the calculus includes two rules: one each for introducing it to the antecedent (a left rule) and succedent of the sequent (a right rule). For example, conjunction, $\wedge$, has the following two rules:

$$\frac{\Gamma, F_1, F_2 \longrightarrow \Delta}{\Gamma, F_1 \wedge F_2 \longrightarrow \Delta} \wedge\text{-L} \qquad \frac{\Gamma \longrightarrow F_1, \Delta \quad \Gamma \longrightarrow F_2, \Delta}{\Gamma \longrightarrow F_1 \wedge F_2, \Delta} \wedge\text{-R.}$$

A *proof* in the sequent calculus consists of a finite tree regulated by such rules. We write $\Gamma \vdash \Delta$ to denote that the sequent $\Gamma \longrightarrow \Delta$ has a proof.

An important property of *classical* proofs is that they permit the *structural* inferences of *weakening* and *contraction*, on the left,

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, F \longrightarrow \Delta} \quad \text{W-L} \quad \frac{\Gamma, F, F \longrightarrow \Delta}{\Gamma, F \longrightarrow \Delta} \quad \text{C-L,}$$

together with similar ones on the right.

It should be clear that both of these rules preserve classical truth, *i.e.* that if the premiss is derivable then so is the conclusion. The effect of these rules is that a formula can be used any number of times in a proof. The classical sequent calculus is given in Appendix A.

The main difference between classical and linear logic is that in linear logic, these two rules (and the corresponding right rules) are omitted. This seems straightforward to justify if we think not in terms of truth, but in terms of resources. In such a case it seems reasonable to omit both of these rules, as having two or zero units of a resource is different from having one unit of the resource. A full introduction to linear logic is beyond the scope of this paper; for now, it is sufficient to note that the rules of weakening and contraction cannot be generally applied, and as a result there are more connectives and less equivalences in linear logic than in the classical case. In particular, linear logic contains two conjunctions $\otimes$ and $\&$, two disjunctions $\oplus$ and ⅋, two special operators ! and ? to which weakening and contraction can be applied, the usual two quantifiers written $\bigwedge$ (universal) and $\bigvee$ (existential), an implication written as $\multimap$ and negation written as $-^{\perp}$. The conjunctions and disjunctions divide into two kinds, one of each being *multiplicative* and one of each being *additive*.[4] A two-sided version of the linear sequent calculus is given in Appendix B.

Our view of logic programming is that it is a form of proof-search: given a program $\mathcal{P}$ and goal $\mathcal{G}$, we attempt to satisfy $\mathcal{G}$ by searching for a proof of the sequent $\mathcal{P} \longrightarrow \mathcal{G}$, using the inference rules of the sequent calculus as *reduction operators*, in the sense of Kleene [8]. Under this interpretation, the contraction rule states that a formula can be copied, and the weakening rule states that a formula need not be used in (the construction of) a proof. The lack of these rules in linear logic can have some subtle effects. For example, in classical logic, universal quantifiers distribute over conjunctions, and hence $\forall x.\, p(x) \wedge q(x)$ is classically equivalent to $\forall x.\, p(x) \wedge \forall x.\, q(x)$. This means that in the classical theory of Horn clauses we can consider all variables as universally quantified at the front of the clause.

An important effect of this in Prolog is that substitutions, arising from unification, need not be applied to the program, but only to the goal. For example, consider the program $P, \forall x.\,(G \supset p(x))$ and the goal $p(t) \wedge G'$. Due to the presence of the contraction rule, when we unify $p(x)$ and $p(t)$, we can make a copy of the clause (in fact, this is necessary to maintain completeness), and so the next step is to attempt to find a proof of the sequent $P, \forall x.\,(G \supset p(x)), G[t/x] \supset p(t) \longrightarrow G'[t/x]$. However, it is not hard to show that the "stronger" clause above overrides the "weaker" one, and hence this is provable iff $P, \forall x.\,(G \supset p(x)) \vdash G'[t/x]$. Thus the substi-

[3] We are concerned here with the cut-free sequent calculus.

[4] In multiplicative rules, the side formulae of the premisses are combined to form larger collections of side formulae of the conclusion, *e.g.*

$$\frac{\Gamma_1 \longrightarrow F_1, \Delta_1 \quad \Gamma_2 \longrightarrow F_2, \Delta_2}{\Gamma_1, \Gamma_2 \longrightarrow F_1 \otimes F_2, \Delta_1, \Delta_2} \quad \otimes\text{-R.}$$

However, in additive rules, the side formulae of each premiss and of the conclusion are identical, *e.g.*

$$\frac{\Gamma \longrightarrow F_1, \Delta \quad \Gamma \longrightarrow F_2, \Delta}{\Gamma \longrightarrow F_1 \,\&\, F_2, \Delta} \quad \&\text{-R.}$$

In the presence of structural rules discussed above, these two conjunctions are equivalent (and similarly for the disjunctions).

tution need only be applied to the goal.

When performing a similar derivation step in the linear calculus, it is not generally possible to copy the clause, and so it is necessary to instantiate the clause using the substitution generated by the unification. For example, given the sequent $\bigwedge x.\,(p(x)\otimes q(x)) \longrightarrow p(a), G$, the unification between $p(x)$ and $p(a)$ produces the substitution $x \leftarrow a$, which must be applied to $q(x)$ as well as to $p(x)$. Hence the sequent becomes $q(a) \longrightarrow G$, as the lack of contraction means that $p(a)$ must be deleted from the program. Thus, in general, the unification mechanism of Lygon is more intricate than that of Prolog.

It should be noted that the weakening and contraction rules are recovered in linear logic by means of the connectives ! and ?. Essentially, the weakening and contraction rules can only be applied to a formula in the antecedent when the formula commences with a !. This means that we can think of clauses commencing with a ! as infinite resources, in that they can be used any number of times (including zero). This feature means that we can recover the behaviour of Prolog in Lygon, provided that we insert ! at the appropriate places.

This also means that we can group together formulae in interesting ways. For example, consider the clause $!(p\otimes q)$. Due to the occurrence of !, we can make an arbitrary number of copies of the formula. However, no matter how many times the formula is used, the same number of copies of p and q must be made; in other words, for every p used, we must use exactly one q. Hence the connective $\otimes$ provides us with an interesting way of balancing resources. In particular, it is often useful to exploit our particular implementation of the balancing of resources, which involves passing information from one computation branch to another. For example, consider a (trivial) bin packing problem in which there is only one type of bin, and each bin must be either empty or contain exactly two objects. The particular instance of the problem is to determine a bin assignment for three objects (which clearly cannot exist). Now consider the sequent $!\,(p\otimes p) \vdash p\otimes p\otimes p$. The proof-search procedure for this sequent would initially make the entire program available to the first conjunct p. In order for this goal to be provable, we need to "extract" two copies of p, so that the antecedent becomes $p, p, !\,(p\otimes p)$, and hence we pass the resources $\{p,\ !(p\otimes p)\}$ on to the next conjunct. Now the second conjunct is provable from these resources without addition, which means that we pass $\{!\,(p\otimes p)\}$ to the third branch of the computation. Clearly we need to extract copies of p again in order to prove p, but this requires the generation of two copies of p, and as there are no further goals onto which we can pass excess resources, we conclude that there is no proof of $!\,(p\otimes p) \longrightarrow p\otimes p\otimes p$, and hence the corresponding bin packing problem has no solution.

In § 6 we show how a non-trivial bin packing problem can be implemented using similar features of Lygon.

3 Definitions

We define below the class of formulae for which a suitable, logically complete notion of goal-directed proof is available [4, 5, 6]; such formulae comprise the basis of *programs* and *goals*.

Definition 3.1 *Let A range over atomic formulae. We define the classes of* definite formulae *and* goal formulae *as follows:*

$$\begin{array}{rcl} D & ::= & A \mid \mathbf{1} \mid \bot \mid D \,\&\, D \mid D\otimes D \mid D \mathbin{⅋} D \\ & & \mid \bigwedge x \,.\, D \mid !\,D \\ & & \mid G \multimap A \mid G \multimap \bot \mid G \multimap \mathbf{1} \end{array}$$

$$\begin{array}{rcl} G & ::= & A \mid \mathbf{1} \mid \bot \mid \top \mid G\otimes G \mid G\oplus G \\ & & \mid G \mathbin{⅋} G \mid G \,\&\, G \mid D \multimap G \\ & & \mid \bigwedge x \,.\, G \mid \bigvee x \,.\, G \mid !\,G \mid ?\,G. \end{array}$$

Programs *are linear antecedents that consist of just closed definite formulae and* goals *are linear succedents that consist of just closed goal formulae. We assume that all quantified variables are* standardized apart. □

This class of formulae is similar to that of hereditary Harrop formulae due to Miller *et al.* [9]. It is possible to "normalize" this class of formulae somewhat, in particular by writing $\{D_1\otimes D_2\}$ as $\{D_1, D_2\}$ and by omitting the outermost level of universal quantifiers. (Note, however, that the other connectives require more delicate treatment.) A mapping, $[-]$, which does this is given in [6]; for brevity it is omitted here. The effect of this mapping is that each program $\mathcal{P}$ has an equivalent multiset $[\mathcal{P}]$ of *clauses*, defined below.

Definition 3.2 *We define classes of* basic clauses, subclauses *and* clauses *by the following grammar:*

$$\begin{array}{lcl} B & ::= & \mathbf{1} \mid \bot \mid A \mid G \multimap A \mid G \multimap \bot \mid G \multimap \mathbf{1} \\ S & ::= & B \mid S \otimes S \mid S \maltese S \mid S \,\&\, S \mid\, !\, S \\ C & ::= & B \mid S \maltese S \mid S \,\&\, S \mid\, !\, S. \end{array} \qquad \Box$$

Note that due to the absence of weakening and contraction in linear logic, programs and goal are multisets of formulae, rather than sets. Also, as universal quantification does not distribute over $\otimes$, in linear logic the same variable can appear in more than one clause; such variables we call *global variables*.

Definition 3.3 *Let $\mathcal{P}$ be a program. If x is a free variable in $[\mathcal{P}]$, then it is a* global variable *if it is bound outside the scope of any ! in P. Otherwise, x is a* local variable. $\Box$

For example, consider the definite formula $\bigwedge x.\, !\bigwedge y.\ p(x,y)$. It corresponds, under the mapping $[-]$, to the clause $!\,p(x,y)$ where x is a global variable and y is a local variable.

Classical Horn clauses are obtained when clauses and goals are restricted as specified below, and all variables are local:

$$\begin{array}{lcl} C & ::= & !A \mid !(G \multimap A) \\ G & ::= & A \mid G \,\&\, G \mid \bigvee x\,.\,G. \end{array}$$

The definitions of clauses and the mapping $[-]$ are such that sequents of the form $[\mathcal{P}] \longrightarrow \mathcal{G}$ have *resolution* proofs, *i.e.* proofs that are not only goal-directed, but also clause-directed, via a special left rule of *resolution*. The details of resolution proof and its properties are given in [4, 5, 6, 11].

4 Operational Model

The full definition of the operational semantics of Lygon is beyond the scope of the present paper. However, we outline the important aspects in this section.

In general, the state of a Lygon computation is given by a multiset of sequents of the form $\Gamma \longrightarrow \Delta$, where Γ is a multiset of clauses and Δ is a multiset of goal formulae. Note that, by the reversibility of the $\otimes$-L and $\maltese$-R rules, we can think of the sequent $C_1, \ldots, C_n \longrightarrow G_1, \ldots, G_m$ as the sequent $C_1 \otimes \ldots \otimes C_n \longrightarrow G_1 \maltese \ldots \maltese G_m$.

We emphasize that the characteristic feature of our approach to logic programming is that each connective in a goal corresponds to a particular proof-search operation, and that a special operation, known as the resolution rule, is used for atomic goals. Hence we need to find an appropriate search operation for each right rule, and a way of encoding the appropriate left rules into a single rule.

As far as the first task is concerned, the main technical difficulties are to do with the connective $\otimes$. Consider the $\otimes$-R rule:

$$\frac{\Gamma_1 \longrightarrow F_1, \Delta_1 \quad \Gamma_2 \longrightarrow F_2, \Delta_2}{\Gamma_1, \Gamma_2 \longrightarrow F_1 \otimes F_2, \Delta_1, \Delta_2} \otimes\text{-R}.$$

When using this rule for finding proofs, it is necessary to split up the antecedent and succedent of the given sequent. In particular, we need to balance the allocation of resources to each branch of the computation. For example, we have that $p, q \vdash p \otimes q$, but neither of the sequents $p \otimes p \longrightarrow p$ and $p \longrightarrow p \otimes p$ is provable. However, there is no way to tell in advance precisely which parts of the program are needed on which branch, and as there are an exponential number of sub-multisets of a given multiset, it is not feasible to test all possibilities. Hence the fundamental problem, given the sequent $\mathcal{P} \longrightarrow G_1 \otimes G_2, \mathcal{G}$, is to determine how to balance the resources on each branch. Our solution, which is not necessarily the only one, is to allocate such the resources in a lazy sequential fashion; in other words, we allow one conjunct to have access to all of the resources, and once it has used all that it needs, the remainder is passed to the other conjunct. Hence we begin with $\mathcal{P} \longrightarrow G_1, \mathcal{G}$, and this is allowed to run to completion, leaving behind the unused resources $\mathcal{P}' \longrightarrow \mathcal{G}'$, which are then allocated to the second conjunct, *i.e.* we must show that $\mathcal{P}' \longrightarrow G_2, \mathcal{G}'$ is provable. Note that in order for a proof to exist, the second conjunct must consume all remaining resources. In this way we lazily calculate multisets $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{G}_1'$, $\mathcal{G}_2'$ such that $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ and $\mathcal{G} = \mathcal{G}_1' \cup \mathcal{G}_2'$ and $\mathcal{P}_1 \vdash G_1, \mathcal{G}_1'$ and $\mathcal{P}_2 \vdash G_2, \mathcal{G}_2'$, as required by the $\otimes$-R rule.

For example, when evaluating $p, q \longrightarrow p \otimes q$, we initially allocate all of the program to the first sub-goal, so that we have $p, q \longrightarrow p$, and we note that q is an unused resource; more precisely, that the deletion of this formula results in a provable sequent. Hence we then pass this "remnant" to the other sub-goal, resulting in the sequent $q \longrightarrow q$ which is clearly provable. Thus we enforce the constraint of balancing resources across different branches by sequentially evaluating each branch in this lazy manner. This idea is similar to the so-called "input/output model of resource consumption", described for a simpler language in [7]. We can think of this as a *sequential* form of conjunction, in that resources are passed sequentially from one computation to another.

The other binary connectives are generally less problematic. For the other conjunction $\&$, there is no need for splitting up the program and goal, but we need to ensure that exactly the same resources are available for each conjunct. In addition, we must have that the "leftover" resources are the same in each case. Hence there is some copying and checking to be done in this case, but it does not appear to be too onerous. For the disjunctions, $⅋$ is particularly simple, as we can replace a goal $G_1 ⅋ G_2$ in the succedent by G_1, G_2. However, note the symmetry between $⅋$ in a succedent and $\otimes$ in an antecedent; the presence of this connective in a goal means that some interesting effects can occur. Goals of the form $G_1 \oplus G_2$ or $D \multimap G$ present no new problems; we handle the former by choosing one subgoal and backtracking when necessary, and the other by adding $[D]$ to the program and evaluating G.

The quantifiers also present no new problems, as we can handle the goal $\bigvee x. G$ by unification methods, and the goal $\bigwedge x. G$ by the well-known technique of replacing the universal quantifier by a new constant (this technique is used in λProlog for example [10]). For simplicity we shall omit discussion of the goals $!G$ and $?G$. This leaves only the linear constants $\bot$, $\top$ and $\mathbf{1}$, and the resolution rule.

The constant $\bot$ can be thought of as an empty formula when it occurs in a succedent, and can be ignored. However, it can be useful at times, as we shall see, to interpret it as a distinguished atom, and hence use the resolution rule to good effect.

The constant $\mathbf{1}$ can be thought of as the empty program, and so when it occurs as a goal, it can only succeed when resolved against an empty program. However, there are times when the program can be "emptied out" to meet this criterion by means of the resolution rule — for example, $p, p \multimap \mathbf{1} \vdash \mathbf{1}$. Hence we can think of this goal as requiring that all resources be used.

The constant $\top$ is somewhat different, as $\Gamma \vdash \top, \Delta$ for any Γ, Δ. Hence we can think of $\top$ as an erase function, as it deletes any resources "left over" from elsewhere. This should be used with caution, and presumably have the lowest priority as a goal, as otherwise we can never get to do useful work. For this reason we will assume that $\top$ has the lowest priority of any goal, and acts only as a "mop up" operator after all other choices have been exhausted.

In order to determine the resolution rule, we need to encode the rules $⅋$-L,$\bigwedge$-L, $\&$-L, $\otimes$-L, $\multimap$-L, !-L, W!-L and C!-L into a single one. This corresponds to breaking down clauses into basic clauses, so that a unification can be performed with the head of a basic clause. Due to the presence of the $⅋$-L rule, it can be necessary for branching to occur in order to find such a basic clause, and so a similar mechanism to that described above for the $\otimes$-R rule is used. Clearly this branching cannot be done at compile time, and so resolution in this context is more intricate than merely unifying and generating a new goal. In particular, a resolution step can involve making a copy of a clause commencing with ![5], branching the computation via the connective $⅋$, decomposing a clause $C_1 \otimes C_2$ into the multiset $\{C_1, C_2\}$ or replacing $C_1 \& C_2$ with either C_1 or C_2. Typically a combination of these operations will be required, and in any case the clause used must be deleted from the program.

Below we present an example of the application of the resolution rule, where we write c for the formula $!\,(q(x) \otimes (r(x) \multimap p(x)))$ in which x is local.

$$\frac{P, q(a), c \longrightarrow r(a), G}{P, c \,\&\, s \longrightarrow p(a), G}$$

Note that the steps in this process include choosing between the clauses c and s, and that as x is local, a "new" local variable is generated before the unification is done.

5 Programming Techniques

State and Memories As mentioned above, when attempting to find a proof of $P \longrightarrow G_1 \otimes G_2$, we use the technique of passing the unused resources from one conjunct to the other. This can be used as a

[5]Note that this involves renaming all local variables.

kind of state-mechanism, in that the first conjunct can pass on a different state 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 $m(2,x) \otimes (m(2,x) \multimap G)$, 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 $\multimap$ connective, before G is executed. In a similar manner, writing the value t' into the memory can be achieved by the goal $m(2,x) \otimes (m(2,t') \multimap G)$, 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 $D_1, D_2 \longrightarrow G_1 \otimes G_2$ and $D_1 \maltese D_2 \longrightarrow G_1, G_2$. 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, thus giving the programmer explicit control over the way that resources are used.

Global variables As can be seen from the definitions above, any universally quantified variable which appears in a clause outside the scope of any occurrence of ! is denoted as a *global variable*. Such a variable can occur in more than one clause, and such occurrences cannot be standardised apart, as is the case in Prolog. This is due to the fact that in linear logic the universal quantifier does not distribute over $\otimes$, *i.e.* that the two formulae $\bigwedge x.\, (p(x) \otimes q(x))$ and $(\bigwedge x.\, p(x)) \otimes (\bigwedge x.\, q(x))$ are not equivalent. As mentioned above, this is in contrast to the case in classical or intuitionistic logic, in both of which 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 can 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, and so 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, and hence 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 communicate by means of this shared message-list.

Note also that a mixture of global and local variables can have some interesting effects. For example, consider the definite formulae $\bigwedge x.\, ! \bigwedge y.\, p(x,y)$, corresponding to the clause $!\, p(x,y)$, in which x is global and y is local. In order 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 $!\, p(x,y) \vdash p(a,b) \otimes p(c,d)$.

"Soft" Deletes Another way in which the lack of weakening and contraction in linear logic can be put to good use is in the area of updates to programs. Often in systems such as transaction processing systems, it is necessary to "softly" perform changes until a commit phase is reached, and it is known that the transaction will be allowed to complete. Only then are the real changes made, with the list of changes having been stored in some manner in the meantime.

We can model such "soft" deletes in Lygon as follows. Let C be a clause in the program that we wish to delete as part of a transaction, so that the program is initially $\mathcal{P} \cup \{C\}$.[6]

[6] Clearly C cannot commence with a !. For simplicity, we will also assume that C contains no global

Rather than deleting C directly, we *add* to the program the clause $C \multimap \mathbf{1}$, so that we now have $\mathcal{P} \cup \{C, C \multimap \mathbf{1}\}$. Clearly this change is reversible, in that if it is discovered that the transaction needs to be aborted, we simply delete the clause $C \multimap \mathbf{1}$, and we are back to where we started. If it is determined that the transaction will succeed, then we can delete both C and $C \multimap \mathbf{1}$, as this corresponds to the reduction of the sequent $\mathcal{P}, C, C \multimap \mathbf{1} \vdash \mathbf{1} \otimes G$ to the sequents $\mathcal{P} \vdash G$ and $C, C \multimap \mathbf{1} \vdash \mathbf{1}$. Note that as $\mathbf{1} \otimes G$ is equivalent to G, we can perform this operation as often as necessary to "clean up" all such pairs.

In this manner we can use the properties of the linear constant $\mathbf{1}$ to mark clauses as deleted, and then later actually delete them.

"Soft" Additions A similar technique can be used for additions to the program. In this case, rather than adding a clause C directly to the program, we add the clause $(C \multimap \bot) \multimap \bot$ to the program, so that the intermediate state is $\mathcal{P} \cup \{(C \multimap \bot) \multimap \bot\}$.[7] Again, if the transaction needs to be aborted, this change can easily be reversed. If the transaction succeeds, then we can replace $(C \multimap \bot) \multimap \bot$ with just C, which corresponds to the reduction of the sequent $\mathcal{P}, (C \multimap \bot) \multimap \bot \vdash \bot, G$ to the sequent $\mathcal{P} \vdash G, C \multimap \bot$ and then to $\mathcal{P}, C \vdash G$.

Mutual Exclusion The connective $\&$ can be thought of as specifying internal choice; in other words, when faced with $C_1 \& C_2$, 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 certain occurrences of Prolog's cut-operator. 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.

Affine Mode 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, 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. We refer to this as affine mode as that is the name of the logic obtained when arbitrary weakening (but not contraction) is added to linear logic.

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 $\top$.

Relevant Mode A similar property to that described above is to insist that a certain infinitary resource *must* be used in the computation, in the manner of relevance logic[12]. It is not hard to see that we can simulate such an effect in linear logic by replacing such a clause $!C$ with $!C \otimes C$. Note that this is similar to allowing the contraction rule, but not weakening. This is similar to the properties of a relevant logic, and hence the name.

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 list of resources. This list is 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.

6 A Bin Packing Problem

In this section we consider an extended example, illustrating the use of Lygon. Our partic-

variables.

[7]The careful reader will note that the formula added is the double negation of C, and hence is equivalent to C.

ular example is a bin packing problem, circulated recently by Andre Vellino.

There are five types of objects, being glass, plastic, steel, wood and copper, and three types of bins, being red, blue and green. Red bins may contain up to three objects, which may be glass, wood or copper. Blue bins can contain only one object, which may be glass, steel or copper. Green bins may contain up to four objects, which may be plastic, wood or copper. No bin may to contain both glass and copper objects, or both copper and plastic objects. Finally, a red bin may contain at most one wooden object, and a green bin may contain at most two wooden objects. A sample bin packing problem is to find the minimum number of bins need to store one glass object, two plastic objects, one steel object, three wooden objects and two copper objects.

There are two main design decisions to be made here: one is the data structure used for the bins; the other is to determine how to satisfy the constraints on the contents of each bin. One solution in Lygon is to model each bin as a predicate of the appropriate arity; for example, an empty red bin corresponds to the atomic clause `red(empty, empty, empty)`. We then use the state transition features of the language to gradually fill up each bin. For example, inserting a glass object into an empty red bin involves deleting the atom `red(empty, empty, empty)` and inserting the atom `red(glass, empty, empty)`. Naturally this requires an unbounded source of empty bins; so the program contains the clauses `!red(empty, empty, empty)` and `!green(empty, empty, empty, empty)`.[8] When a new (empty) bin is needed, a copy of the appropriate bin is made, and added to the current state. If there is room in an existing bin, and the constraints on the contents are satisfied, it is straightforward to write code to "fill" the empty slot in a bin. It is not, however, necessary to include the clause `!blue(empty)` to the program, as blue bins can only contain one object, and hence there is never any need to check the content of a blue bin before assigning an object to it. This means that the containment constraints for blue bins are trivial, and hence the code for insertion of objects into blue bins is particularly simple, and is given below. The code to insert an object `Obj` in a bin is given in the definition of the predicate `insert(Obj)`, which places `Obj` in an appropriate bin, if one can be found. Hence, inserting a glass object should result in the addition of `blue(glass)` to the current state. The clauses which encode this behaviour are as follows:

```
! (insert(glass) || blue(glass))
! (insert(steel) || blue(steel))
! (insert(copper) || blue(copper))
```

where we write || for the linear connective ⅋. Note that here (and in all cases that follow) the scope of the ! is the entire clause.

In each case, a new full blue bin is added to the current state, with the appropriate object in the bin. Note that we could write the first clause above as

```
! insert(glass) :- not(blue)(glass)
```

and similarly for the other clauses, where we write `:-` for the linear connective $\circ\!-$ and we use $F \multimap \perp$ as $F^\perp$ for simplicity, even though $F^\perp$ is not strictly a goal formula.[9] We write `not(p)` for $p^\perp$.

For the red and green bins, we need to check what the current contents are in order to satisfy the containment constraints. Hence we need to interrogate the current state to determine which bins have free slots and what the contents of the occupied slots are, and possibly create a new empty bin. Once that is done, we can then insert elements as appropriate.

The code for insertion into red bins is given below. Note that in this case we need to check that there is room in the bin, and that copper and glass objects are placed in different bins. We assume the existence of the predicates `member` and `sub`: `member` checks that the first argument is an element of the second; `sub` checks that the first argument, with, for simplicity, all occurrences of `empty` deleted, is a sub-multiset of the second argument. We also assume the existence of the predicate `assign(Obj, List, NewList)`, which inserts the object `Obj` into the list `List` in place of an occurrence of `empty`, producing the list `NewList`.

[8] Here we take `!` to have higher precedence than the binary connectives.

[9] The reader versed in linear logic will note that $F^\perp$ and $F \multimap \perp$ are equivalent.

```
! insert(glass):-
   red(X,Y,Z)
   (x) member(empty,[X,Y,Z])
   (x) sub([X,Y,Z],[glass,glass,wood])
   (x) assign(glass,[X,Y,Z],[X1,Y1,Z1])
   (x) not(red)(X1,Y1,Z1)

! insert(wood) :-
   red(X,Y,Z)
   (x) member(empty,[X,Y,Z])
   (x) sub([X,Y,Z],[glass,glass,copper,copper])
   (x) assign(wood,[X,Y,Z],[X1,Y1,Z1])
   (x) not(red)(X1,Y1,Z1)

! insert(copper) :-
   red(X,Y,Z)
   (x) member(empty,[X,Y,Z])
   (x) sub([X,Y,Z],[copper,copper,wood])
   (x) assign(glass,[X,Y,Z],[X1,Y1,Z1])
   (x) not(red)(X1,Y1,Z1) .
```

where we write `(x)` for the connective $\otimes$.

Note that when inserting a wooden object into a red bin we need to ensure that there is no other wooden object already present, as there can be at most one wooden object in each red bin.

The code for the green bin is similar, and hence omitted.

When all objects have been inserted into bins, the current state records an assignment of objects into bins satisfying the given constraints, and represents a solution to the original problem. Note that as the bin assignment is given by the "excess" resources and that a linear proof requires that all resources be used, we will use $\top$ (written as `consume`) in order to actually complete the proof. In practice, this would also involve displaying the resulting state to the user.

The goal corresponding to our particular instance of the problem is given below.

```
?- insert(glass) (x) insert(plastic)
(x) insert(plastic) (x) insert(steel)
(x) insert(wood) (x) insert(wood) (x)
insert(wood) (x) insert(copper) (x)
insert(copper) (x) consume
```

Some variations on this problem can also be easily handled in Lygon. For example, we can alter the specification of the problem so that wooden objects are sub-divided into beech and oak objects, but that all red bins must contain the same type of wood, *i.e.* all red bins must contain beech, or all red bins must contain oak. This can be achieved by the use of the additive conjunction $\&$; in particular, by adding the clause below to the program

```
(!wood_type(beech)) & (!wood_type(oak))
```

as well as modifying the code for insertion into red bins to have the following two clauses in place of the one for wood given above:

```
!insert(beech) :-
   red(X,Y,Z)
   (x) wood_type(beech)
   (x) member(empty,[X,Y,Z])
   (x) sub([X,Y,Z],[glass,glass,copper,copper])
   (x) assign(beech,[X,Y,Z],[X1,Y1,Z1])
   (x) not(red)(X1,Y1,Z1) .

!insert(oak) :-
   red(X,Y,Z)
   (x) wood_type(oak)
   (x) member(empty,[X,Y,Z])
   (x) sub([X,Y,Z],[glass,glass,copper,copper])
   (x) assign(oak,[X,Y,Z],[X1,Y1,Z1])
   (x) not(red)(X1,Y1,Z1) .
```

Initially, it is possible to put either type of wood into a red bin. However, once `wood_type(beech)` is executed, the clause `(!wood_type(beech))&(!wood_type(oak))` becomes `!wood_type(beech))` and henceforth only beech may be inserted into red bins. A similar property holds in the case that oak is inserted first.

It is clear that the latter two clauses above are only superficially different (as is the case with several other clauses above). It is possible to use the additive disjunction, $\oplus$ (written here as `;`) together with an equality test to merge the above two clauses into one. This requires writing the argument to `insert` as a variable, and using the disjunction and equality to limit it to the two cases of `beech` and `oak`. The modified clause is given below.

```
!insert(Wood) :-
   (Wood = beech; Wood = oak)
   (x) red(X,Y,Z)
   (x) wood_type(Wood)
   (x) member(empty,[X,Y,Z])
   (x) sub([X,Y,Z],[glass,glass,copper,copper])
   (x) assign(Wood,[X,Y,Z],[X1,Y1,Z1])
   (x) not(red)(X1,Y1,Z1) .
```

Note that we follow Prolog syntax for variables, and hence the capital letter in the first letter of `Wood` indicates that `Wood` is a variable. Note also that the use of the disjunction and equality ensures that this code is only applicable to objects of type beech or oak; without this check, the clause above would be applicable to all objects.

This example serves to illustrate how such problems can be coded simply and elegantly in Lygon. Note also that a Prolog solution using the same methodology would need to use asserts and retracts (or something similar) to maintain the current state of the bins.

7 Conclusions and Further Work

We have seen how the properties of linear logic lead to a logic programming language which is more powerful than one based on classical logic. In particular, Lygon contains features such as global variables and mutual exclusion operators which are a direct result of the basis of the language on linear logic. We have also seen how a particular method of dealing with goals such as $G_1 \otimes G_2$ leads to a useful notion of state. For these reasons we believe that Lygon offers a useful combination of features that are not found in languages based on classical logic. In addition, Lygon contains a richer class of formulae than other linear logic programming languages, such as Lolli [7].

Current research in progress includes, *inter alia*, the complete definition of the operational semantics of the full Lygon language, together with the development of a proof-theoretic characterization of such semantics via the notion of a *path* (related to that of a proof net) introduced in [6, 4, 5, 11].

References

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

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

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

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

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

[6] J.A. Harland, D.J. Pym. A Uniform Proof-theoretic Investigation of Linear Logic Programming. To appear in Journal of Logic and Computation.

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

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

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

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

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

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

A Classical Sequent Calculus

$$\frac{}{\phi \longrightarrow \phi}\ \text{axiom} \qquad \frac{\Gamma \longrightarrow \phi, \Delta \qquad \Gamma, \phi \longrightarrow \Delta}{\Gamma \longrightarrow \Delta}\ \text{cut}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, \phi \longrightarrow \Delta}\ \text{W-L} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \phi, \Delta}\ \text{W-R}$$

$$\frac{\Gamma, \phi, \phi \longrightarrow \Delta}{\Gamma, \phi \longrightarrow \Delta}\ \text{C-L} \qquad \frac{\Gamma \longrightarrow \phi, \phi, \Delta}{\Gamma \longrightarrow \phi, \Delta}\ \text{C-R}$$

$$\frac{\Gamma, \phi, \psi \longrightarrow \Delta}{\Gamma, \psi, \phi \longrightarrow \Delta}\ \text{I-L} \qquad \frac{\Gamma \longrightarrow \phi, \psi, \Delta}{\Gamma \longrightarrow \psi, \phi, \Delta}\ \text{I-R}$$

$$\frac{\Gamma, \phi, \psi \longrightarrow \Delta}{\Gamma, \phi \wedge \psi \longrightarrow \Delta}\ \wedge\text{-L} \qquad \frac{\Gamma \longrightarrow \phi, \Delta \qquad \Gamma \longrightarrow \psi, \Delta}{\Gamma \longrightarrow \phi \wedge \psi, \Delta}\ \wedge\text{-R}$$

$$\frac{\Gamma, \phi \longrightarrow \Delta \qquad \Gamma, \psi \longrightarrow \Delta}{\Gamma, \phi \vee \psi \longrightarrow \Delta}\ \vee\text{-L} \qquad \frac{\Gamma \longrightarrow \phi, \psi, \Delta}{\Gamma \longrightarrow \phi \vee \psi, \Delta}\ \vee\text{-R}$$

$$\frac{\Gamma \longrightarrow \phi, \Delta \qquad \Gamma, \psi \longrightarrow \Delta}{\Gamma, \phi \supset \psi \longrightarrow \Delta}\ \supset\text{-L} \qquad \frac{\Gamma, \phi \longrightarrow \psi, \Delta}{\Gamma \longrightarrow \phi \supset \psi, \Delta}\ \supset\text{-R}$$

$$\frac{\Gamma \longrightarrow \phi, \Delta}{\Gamma, \neg\phi \longrightarrow \Delta}\ \neg\text{-L} \qquad \frac{\Gamma, \phi \longrightarrow \Delta}{\Gamma \longrightarrow \neg\phi, \Delta}\ \neg\text{-R}$$

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

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

where y is not free in Γ, Δ.

B Linear Sequent Calculus

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

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

$$\frac{\Gamma, \phi, \psi, \Gamma' \longrightarrow \Delta}{\Gamma, \psi, \phi, \Gamma' \longrightarrow \Delta}\ \text{X-L}$$

$$\frac{\Gamma \longrightarrow \Delta, \phi, \psi, \Delta'}{\Gamma \longrightarrow \Delta, \psi, \phi, \Delta'}\ \text{X-R}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, \mathbf{1} \longrightarrow \Delta}\ \mathbf{1}\text{-L}$$

$$\frac{}{\longrightarrow \mathbf{1}}\ \mathbf{1}\text{-R}$$

$$\frac{}{\bot \longrightarrow}\ \text{L}^{\bot}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \bot, \Delta}\ \text{R}^{\bot}$$

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

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

$$\frac{\Gamma \longrightarrow \phi, \Delta}{\Gamma, \phi^{\bot} \longrightarrow \Delta}\ \bot\text{-L}$$

$$\frac{\Gamma, \phi \longrightarrow \Delta}{\Gamma \longrightarrow \phi^{\bot}, \Delta}\ \bot\text{-R}$$

$$\frac{\Gamma, \phi, \psi \longrightarrow \Delta}{\Gamma, \phi \otimes \psi \longrightarrow \Delta}\ \otimes\text{-L}$$

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

$$\frac{\Gamma, \phi \longrightarrow \Delta}{\Gamma, \phi \mathbin{\&} \psi \longrightarrow \Delta} \qquad \frac{\Gamma, \psi \longrightarrow \Delta}{\Gamma, \phi \mathbin{\&} \psi \longrightarrow \Delta}\ \&\text{-L}$$

$$\frac{\Gamma \longrightarrow \phi, \Delta \qquad \Gamma \longrightarrow \psi, \Delta}{\Gamma \longrightarrow \phi \mathbin{\&} \psi, \Delta}\ \&\text{-R}$$

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

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

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

$$\frac{\Gamma \longrightarrow \phi, \psi, \Delta}{\Gamma \longrightarrow \phi \parr \psi, \Delta}\ \parr\text{-R}$$

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

$$\frac{\Gamma, \phi \longrightarrow \psi, \Delta}{\Gamma \longrightarrow \phi \multimap \psi, \Delta}\ \multimap\text{-R}$$

$$\frac{\Gamma, \phi \longrightarrow \Delta}{\Gamma, !\phi \longrightarrow \Delta}\ !\text{-L}$$

$$\frac{!\Gamma \longrightarrow \phi, ?\Delta}{!\Gamma \longrightarrow !\phi, ?\Delta}\ !\text{-R}$$

$$\frac{!\Gamma, \phi \longrightarrow ?\Delta}{!\Gamma, ?\phi \longrightarrow ?\Delta}\ ?\text{-L}$$

$$\frac{\Gamma \longrightarrow \phi, \Delta}{\Gamma \longrightarrow ?\phi, \Delta}\ ?\text{-R}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, !\phi \longrightarrow \Delta}\ W!\text{-L}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow ?\phi, \Delta}\ W?\text{-R}$$

$$\frac{\Gamma, !\phi, !\phi \longrightarrow \Delta}{\Gamma, !\phi \longrightarrow \Delta}\ C!\text{-L}$$

$$\frac{\Gamma \longrightarrow ?\phi, ?\phi, \Delta}{\Gamma \longrightarrow ?\phi, \Delta}\ C?\text{-R}$$

$$\frac{\Gamma, \phi[t/x] \longrightarrow \Delta}{\Gamma, \bigwedge x\,.\,\phi \longrightarrow \Delta}\ \bigwedge\text{-L c}$$

$$\frac{\Gamma \longrightarrow \phi[y/x], \Delta}{\Gamma \longrightarrow \bigwedge x\,.\,\phi, \Delta}\ \bigwedge\text{-R}$$

$$\frac{\Gamma, \phi[y/x] \longrightarrow \Delta}{\Gamma, \bigvee x\,.\,\phi \longrightarrow \Delta}\ \bigvee\text{-L}$$

$$\frac{\Gamma \longrightarrow \phi[t/x], \Delta}{\Gamma \longrightarrow \bigvee x\,.\,\phi, \Delta}\ \bigvee\text{-R}$$

where y is not free in Γ, Δ.