Linear Logic Semantics for Concurrent Prolog

Jiří Zlatuška

Institute of Computer Science, Masaryk University
Burešova 20, 602 00 Brno, Czech Republic
zlatuska@muni.cz

Abstract. This paper develops a proof-theoretic semantics based on linear logic for Concurrent Prolog. Several fragments of linear logic are used in order to provide for a suitable translation of Concurrent Prolog clauses, with the committed-choice concurrency features captured by the properties of linear logic provability rather than by some additional mechanism and without the need of using the interleaving approach for defining concurrent behaviour. Linear logic operations from different fragments arise naturally from the need for specific features of the translation. It is demonstrated that the relationship between a logic programming language, such as Prolog, and a committed-choice concurrent logic one, such as Concurrent Prolog, can be understood as a change of the underlying logic inference, rather than by resorting to extra-logical machinery.

1 Introduction

Logic programming has gained attraction as a general programming paradigm primarily because of the identification it introduced between the *specification* (in the language of logic) and the *program* itself (as a set of Horn clauses). The traditional view of the underlying abstract computational machine, as based on the SLD-resolution proof tree search for the *answer substitution*, provided for understanding logic programming languages as primarily *declarative* languages based on *logic*, with the addition of certain *control* information supporting efficient proof search — this is the *"program=logic+control"* of [8]. Practical logic programming languages, such as Prolog, depart from this ideal by incorporating extra-logical elements which play an essential rôle in real programming. The declarative semantics of the core parts of these languages is still essential for us to call these languages "logic programming languages".

The situation is rather different when it comes to concurrent logic programming languages such as Concurrent Prolog [11] or Parlog [5]. Only the syntax of definite clauses is retained (with expanded control elements), while the semantics of these languages no longer refers to logic or "declarativeness": instead, their semantics is phrased entirely procedurally in terms of transformation systems of parallel processes, as if the correspondence with sequential logic programming was nothing more than the convenience

of Horn clause notation. In fact, when trying to find a common denominator, it has been easier to turn it around, and to define the sequential logic programming semantics in procedural terms as in [12], and to identify the difference between sequential and committed choice concurrent logic programming languages as that between the kinds of nondeterminism used in the underlying operational mechanisms.

In this paper we try to show that the logic content across these two families can be retained. We show that when using the proof-theoretic semantics of logic programming languages based on sequent systems, we can retain the declarative view of both of them by locating the distinction in the choice of the *logic* used when reading the programs. We show that linear logic sequent calculus allows straightforward interpretation of a concurrent logic programming language which serves as a core of Flat Concurrent Prolog (FCP) [12]. This can be seen as a justification of the idea that committed choice languages are indeed logic programming languages in the full meaning of the term.

The relationship of linear logic to concurrency has been shown earlier in the context of Petri nets, cf. [9]. A general relationship between logic model theory and concurrency which can overcome the limitations of the interleaving semantics has been studied in [10] in the context of functional and object-oriented systems. So far, only an interleaving semantics has been used for concurrent logic programming languages. Based on linear logic, this paper develops a proof-theoretic partial-order semantics for a fragment of Concurrent Prolog which retains all the aspects of "logic programming" as known from the sequential case, just switching to another logic is enough to ensure correctness for the backward-chaining execution of the logic specification. An alternative to this would be a forward-chaining formulation based on interpretation of the provability relation as process transformation; this approach has been used in [13] employing the symmetries of linear implication with respect to linear negation.

2 Logic programs

2.1 Syntax of logic programs

We will use the basic model of logic programs as sets of Horn clauses with terminology based mostly on that of Prolog.

A *term* is a variable or a *function symbol* of arity $n \geq 0$ applied to the corresponding number of terms:

$$T := X \mid f(T_1, \ldots, T_n).$$

For p a predicate symbol of arity $n \geq 0$, an *atom* is a formula of the form

$$p(T_1, \ldots, T_n).$$

For most of this paper, just the propositional case will be enough, that is the case of $n = 0$. With H (the *head*) and $B_1, \ldots, B_n$ with $n \geq 0$ (the *body* part) atoms, a *clause* is a formula

$$H \leftarrow B_1, \ldots, B_n.$$

For the concurrent version we add atoms $G_1, \ldots, G_m$, with $m \geq 0$ (the *guards*), and form a *guarded clause*

$$H \leftarrow G_1, \ldots, G_m | B_1, \ldots, B_n,$$

provided that all variables occurring in the guards also occur in the head. When considering guards, we will restrict our attention to just flat languages, where the set of *guard predicates* is fixed by the language, and distinct from the user-defined predicates which can occur in the head or in the body. A set of (guarded) clauses with the same predicate symbol in their head is called a *procedure*. A *logic program* is a set of clauses. A *goal* is a sequence of atoms:

$$A_1, \ldots, A_k, \qquad k \geq 1.$$

A *sequential logic program* is a set of clauses, a *concurrent logic program* is a set of guarded clauses. Because of the special rôle of the "=" predicate in concurrent logic programming, we assume the clause

$$= (X, X) \leftarrow$$

to be present as the only clause with the = predicate in its head. Similarly, *fail* is a 0-ary predicate which doesn't appear in any clause head.

The basic structure of computations in which we are interested can be captured by using just the propositional case, i.e. just nullary predicates are needed, with no variables. Within the sequential case nothing essential is lost by this. When dealing with concurrency, variables are used as communication channels between processes, so it does make sense to return to them at some point.

When considering variables, we will use substitutions as functions instantiating variables by terms. We employ postfix notation as usual, i.e. the use of substitution ϑ on A will be denoted as $A\vartheta$. Composition of substitutions means ordinary composition of functions, i.e. $(A\vartheta)\theta$ is the same as $A\theta \circ \vartheta$; we also say that $\theta \circ \vartheta$ is an *instance* of ϑ. Two operations on atoms returning substitutions are of interest to us: the first of them is the *most general unifier*: $mgu(A, B)$ returns the most general substitution (i.e. every other with the same property is an instance of it) ϑ such that $A\vartheta = B\vartheta$, or reports failure if such a substitution does not exist. The second operation is *matching*: $match(A, B)$ returns the most general ϑ such that $A = B\vartheta$, reports failure if $\mathrm{mgu}(A, B)$ doesn't exist, and reports *suspension* if neither of those apply (this possibility is for synchronisation control). When defining the operational semantics using SLD-resolution, we also need clause renaming, which is just a special kind of substitution.

2.2 Operational semantics

The standard way of defining the operational semantics of (sequential) logic programs is that of interpreting the program clauses and goals as formulas of the language of Horn clauses and employing SLD-resolution in order to get the *answer substitution* as the result computed by the program when started by a given goal.

Transition systems are used in [12] for defining the operational semantics of both sequential and concurrent logic programs in a similar fashion, distinguished by the way of selecting applicable transitions.

A *transition system* for a logic program $\mathcal{P}$ is a system consisting of (i) a set of *states*, each of those being a pair $<C;\vartheta>$ with a goal C and a substitution ϑ; and (ii) a set of *transitions*, i.e. mappings from states into sets of states. For transition t and states S, S' such that $S' \in t(S)$ we denote $S \rightarrow_t S'$. A transition t is *enabled* on a state S when $t(S) \neq \emptyset$. A state with no transition enabled is called a *terminal* state; a terminal state of the form $<true;\vartheta>$ is called a *success* state, otherwise it is a *failure* state.

A *computation* of a program $\mathcal{P}$ on a goal C is a sequence of states

$$S_0, S_1, \ldots,$$

such that $S_0 = <C;\epsilon>$ (with ϵ being the empty substitution), for every k, $S_k \rightarrow_t S_{k+1}$ for some transformation t. We say that the computation is *terminated* if this sequence is finite with length l and S_l is a terminal state. For any terminated computation ending with $<true;\vartheta_l>$, we call ϑ_l the *answer substitution* of such a computation.

2.3 Sequential programs

The actual behaviour of the transformation system depends on the particular set of transformations. For a (sequential) logic program $\mathcal{P}$ the following two rules can be defined [12]:

reduce: $<A_1,\ldots,A_l,\ldots,A_m;\vartheta> \rightarrow <A_1\theta,\ldots,B_1\theta,\ldots,B_n\theta,\ldots,A_m\theta;\theta\circ\vartheta>$, for any renamed apart clause $A \leftarrow B_1,\ldots,B_n$ of program $\mathcal{P}$, for which $\theta = mgu(A, A_l)$.

fail: $<A_1,\ldots,A_m;\vartheta> \rightarrow <fail;\vartheta>$, whenever there is no renamed apart clause $A \leftarrow B_1,\ldots,B_n$ of P for which $mgu(A, A_1)$ exists.

It is easy to see that the above reductions when plugged into the definition of the notion of transition system allow the generation of SLD-resolution proofs: indeed, every "reduce" transformation corresponds to resolving one of the atoms of the current goal part of the most recent state, with the resolvent becoming the new current goal. In this case we interpret the clauses and goals by the standard Horn clause interpretation. As a consequence, it holds true [7, 12]:

Theorem 1. *Let P be a logic program and C a goal.*
(Soundness): if <C, ε> starts a computation on program $\mathcal{P}$ with ϑ as its answer substitution, then any instance of $C\vartheta$ is a logical consequence of the universal closure of $\mathcal{P}$.
(Completeness): if the universal closure of an instance C′ of a goal C is a logical consequence of the universal closure of $\mathcal{P}$, then there is a computation of the program $\mathcal{P}$ on the goal C with answer substitution ϑ such that C′ is an instance of $C\vartheta$.

2.4 Concurrent programs

The operational semantics of concurrent logic programs only needs to change the pair of reduce/fail transformations in order to obtain the *interleaving* semantics. The actual computation nondeterministically selects one of the clauses with successful matching of its head with the current goal. The idea is to try the alternatives in parallel, and after one of the clauses reaches the *commit* operator, "|", the alternatives are discarded, and it proceeds by evaluating the body goals of the selected clause in parallel.

The process of clause selection is captured by the *try* function applicable to an atom and a clause, defined by:

$$try(A, H \leftarrow \vec{G} \mid \vec{B}) = \begin{cases} \vartheta & \text{if } match(A,H) = \vartheta \text{ and } \vec{G}\vartheta \text{ succeed,} \\ fail & \text{if mgu(A,H) doesn't exist, or } \vec{G}\vartheta \text{ fail,} \\ suspend & \text{otherwise.} \end{cases}$$

Based on the *try* function, the following two rules have been introduced by Shapiro [12] for a version of Flat Concurrent Prolog, FCP(|):

reduce: $\langle A_1, \ldots, A_l, \ldots, A_m; \vartheta \rangle \rightarrow \langle A_1\theta, \ldots, B_1\theta, \ldots, B_n\theta, \ldots, A_m\theta; \theta \circ \vartheta \rangle$, for any renamed apart clause $A \leftarrow \vec{G} \mid B_1, \ldots, B_n$ of program $\mathcal{P}$, for which $\theta = try(A_l, A \leftarrow \vec{G} \mid B_1, \ldots, B_n)$.

fail: $\langle A_1, \ldots, A_m; \vartheta \rangle \rightarrow \langle fail; \vartheta \rangle$, whenever there is no renamed apart clause $A \leftarrow \vec{G} \mid B_1, \ldots, B_n$ of P for which $try(A_l, A \leftarrow \vec{G} \mid B_1, \ldots, B_n) \neq fail$.

The transition system semantics based on these two transformations allows the interpretation of every atom in a goal as a separate *process state*, and the interpretation of the computation sequence as a sequentialised "trace" of state transformations of parallel processes. Failing computations can be divided into two classes: one of them constituting genuine failures, that is the atom *fail* appears as the goal in the terminal state, and the other one being a "deadlocked" computation ending because of *suspend* results of *match* or *try* functions. It is obvious that for every deadlock-free concurrent computation there is a corresponding sequential computation of the same program with guards interpreted as goals and *commit* replaced by a standard goal conjunction, but the contrary is not true because of the possibility of deadlocked computations in the concurrent case [12].

Although the transformation semantics of concurrent logic programs based on interleaving may be suitable for basic visualisations of concurrent logic program behaviour, the need for a "global clock" obscures the parallel semantics. Independently running and-parallel processes are forcibly synchronised. To a given set of choices made by them, there corresponds a set of different computation sequences distinguished only by that artificial serialisation. Clearly, some sort of branching structure would be needed in order to eliminate the need for interleaving parallel computations, but the splits would make it difficult to accommodate the answer substitution construction in a well-defined way.

3 Proof-theoretic semantics

There is a viable alternative to the operational point of view when defining the semantics of logic programs. *Proof-theoretic* semantics views logic program execution as a process of *proof synthesis*. As well as employing objects with better structure than just a sequence of states, this may also provide for a more adequate treatment of practical incarnations of the logic programming paradigm as used, e.g., in Prolog, cf. [6]. In [4] a view of Horn clause programming is presented based on atomic intuitionistic sequents, and the underlying proof-search strategy is derived from the cut-elimination theorem. In [1] the idea of "proofs as answers" instead of "substitutions as answers" is articulated in the presence of sequent calculus.

For the sake of brevity we will only deal with propositional sequential logic programs as variable handling can be added to the system in a fairly standard way.

3.1 Sequent calculus

We will not be dealing with negation, so it is enough to consider only formulas in negation normal form (negations applied only to atoms, not complex formulas). The classical sequent calculus can then be defined for sequents of the form

$$\vdash X_0, \ldots, X_n,$$

which can also be understood as the result of taking a classical sequent $\Gamma \vdash \Delta$, and reverting formulas in Γ by the negation operator $^\perp$ we get $\vdash \Gamma^\perp, \Delta$. ($\Gamma$ and Δ will be used as meta variables for sequences of formulas.)

The sequent calculus G for propositional logic is given by the following set of inference rules (X, Y are formulas):

Axioms and the cut:

$$\frac{}{\vdash X^\perp, X}\ \textit{identity}, \qquad \frac{\vdash \Gamma, X \quad \vdash X^\perp, \Delta}{\vdash \Gamma, \Delta}\ \textit{cut}.$$

Structural rules (exchange, weakening, and contraction):

$$\frac{\vdash \Gamma, X, Y, \Delta}{\vdash \Gamma, Y, X, \Delta}\, E, \qquad \frac{\vdash \Gamma}{\vdash \Gamma, X}\, W, \qquad \frac{\vdash \Gamma, X, X}{\vdash \Gamma, X}\, C.$$

Logical rules (for each logical connective used in the language):

$$\frac{\vdash \Gamma, X}{\vdash \Gamma, X \vee Y}\, \vee_l, \qquad \frac{\vdash \Gamma, Y}{\vdash \Gamma, X \vee Y}\, \vee_r, \qquad \frac{\vdash \Gamma, X \quad \vdash \Gamma, Y}{\vdash \Gamma, X \wedge Y}\, \wedge, \qquad \frac{\vdash \Gamma, X^{\perp}, Y}{\vdash \Gamma, X \supset Y}\, \supset.$$

A *proof* of a sequent $\vdash \Gamma$ (from a set of formulas Σ - the *hypotheses*) is a finite tree of sequents generated by the inference rules above, the leaves of which are generated by the identity rule (or a (unary) sequent of the form $\vdash X$ for $X \in \Sigma$).

From the classical result of Gentzen one has ([4, 1]):

Theorem 2. *(i) For $\Sigma = \emptyset$ the cut rule can be eliminated from the calculus. (ii) The cut rule only needs to be used at the leaves with one of its premises being a sequent formed from a hypothesis from Σ.*

The use of sequent-calculus as a semantics for logic programs is a consequence of the above. In [1] it is shown that program clauses can be built into the cut rules for specific clauses. This allows us to provide for a backward-chaining proof construction which we will employ later for concurrent logic programs. The general idea is that in order to prove a goal

$$\vdash C,$$

the proof construction proceeds as follows (here we take just atomic C)

$$\dfrac{\vdash a, b^{\perp} \qquad \dfrac{\vdash b, c^{\perp} \qquad \dfrac{\vdots}{\vdash c}}{\vdash b}}{\vdash a}\,.$$

where each of the steps uses the cut with a hardwired program clause in it.

When generating such a tree (it and-branches for conjunctive subgoals) with $\vdash C$ in its root, each partial version of it can be viewed as a proof of $\vdash C$ for $C \in \Sigma \cup \{A_1, \ldots, A_n\}$, for $\vdash A_1, \ldots, \vdash A_n$ being non-axiom sequents from its leaves. This gives a link to the transformation-based SLD-resolution semantics, with $<A_1, \ldots, A_n>$ as the current goal of the computation.

Note that the "Standard Prolog" system from [1] technically does not just enforce SLD-resolution. This could be improved by also considering specialised cut rules also for each "clause" (i.e. an instance of the sequent schema of the form $\vdash \Gamma, X, \Gamma'$) as appropriate instances of

$$\frac{\vdash \Gamma, X, \Gamma' \quad \vdash X^{\perp}, \Delta}{\vdash \Gamma, \Delta, \Gamma'}\ \textit{cut with selection for}\ \Gamma, X, \Gamma'.$$

After this, only SLD-proofs are generated. In this context it seems to be a better choice to stick with sequences when introducing sequents, compared to the use of sets as employed by [1], in order to get as close to Prolog as possible.

3.2 Concurrent logic programs

When dealing with concurrent logic programs, the intended meaning of the atoms occurring in the transition system states would be concurrently evolving processes. In the propositional case, i.e. just asynchronous parallel processes without mutual communication via variables, the proof-theoretic equivalent of the state transition from S to S' might be provability of $h \vdash h'$, where h, h' are the symbols corresponding to the state S, S', respectively.

The basic model can be provided by taking the multiplicative fragment of linear logic [2, 3], i.e. two connectives $\otimes$ (times) and $\parr$ (par). Besides the identity axiom for atomic sequents, the cut, and the exchange,

$$\frac{}{\vdash A^{\perp}, A}\ \textit{identity}, \qquad \frac{\vdash \Gamma, X \quad \vdash X^{\perp}, \Delta}{\vdash \Gamma, \Delta}\ \textit{cut}, \qquad \frac{\vdash \Gamma, X, Y, \Delta}{\vdash \Gamma, Y, X, \Delta}\ E,$$

the corresponding logical rules for these two connectives and their neutral elements are

$$\frac{\vdash \Gamma, X \quad \vdash Y, \Delta}{\vdash \Gamma, X \otimes Y, \Delta}\ \textit{times}, \qquad \frac{\vdash \Gamma, X, Y}{\vdash \Gamma, X \parr Y}\ \textit{par},$$

$$\frac{}{\vdash 1}\ \textit{one}, \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \perp}\ \textit{false}.$$

De Morgan equations define negation $^{\perp}$ on them:

$$(X \otimes Y)^{\perp} = X^{\perp} \parr Y^{\perp}, \qquad (X \parr Y)^{\perp} = X^{\perp} \otimes Y^{\perp}, \qquad 1^{\perp} = \perp, \qquad \perp^{\perp} = 1,$$

and linear implication is defined in terms of *par*:

$$X \multimap Y = X^{\perp} \parr Y.$$

Now define translation from guarded clauses into formulas of the multiplicative fragment of linear logic by ($X \multimap Y$ stands for $Y \multimap X$):

$$(H \leftarrow G_1, \ldots, G_m | B_1, \ldots, B_n) \longmapsto H \multimap (G_1 \otimes \ldots \otimes G_m \otimes B_1 \otimes \ldots \otimes B_n).$$

When $m = 0$ and $n = 0$, the right-hand side becomes the neutral element of $\otimes$, i.e. 1.

The backward-chaining proof-theoretic model is given by the following theorem:

Theorem 3. *Let Σ be the set of linear logic formulas obtained from the guarded clauses of a concurrent program $\mathcal{P}$, and let Ω be the set of formulas defining the guard symbols. Then there is a (partial) computation of program $\mathcal{P}$ on a goal $p_1, \ldots, p_k$ ending in state $<q_1, \ldots, q_l>$ iff there is a proof of*

$$\vdash p_1 \otimes \ldots \otimes p_k, q_1^{\perp} \parr \ldots \parr q_l^{\perp}$$

from the set of hypotheses $\Sigma \cup \Omega$.

Proof: By the proof of cut-normalisation for linear logic, we only need to be interested in proofs where cuts are employed on hypotheses from $\Sigma \cup \Omega$. The correspondence we are looking for is between

$$<H;> \rightarrow_t <B_1, \ldots, B_n;>$$

and

$$\dfrac{\vdash H \multimap (G_1 \otimes \ldots \otimes G_m \otimes B_1 \otimes \ldots \otimes B_n) \qquad \dfrac{\vdash (H \multimap (G_1 \otimes \ldots \otimes G_m \otimes B_1 \otimes \ldots \otimes B_n))^{\perp}, H, B_1^{\perp} \parr \ldots \parr B_n^{\perp}}{\vdash H, B_1^{\perp} \parr \ldots \parr B_n^{\perp}}}{\dfrac{\vdash H^{\perp}, H \qquad \dfrac{\vdash G_2 \otimes \ldots \otimes G_m \otimes B_1 \otimes \ldots \otimes B_n, B_1^{\perp} \parr \ldots \parr B_n^{\perp}}{\vdash G_1 \otimes \ldots \otimes G_m \otimes B_1 \otimes \ldots \otimes B_n, B_1^{\perp} \parr \ldots \parr B_n^{\perp}}}{\vdash G_1 \qquad \vdots}}{\dfrac{\vdash B_1 \otimes \ldots \otimes B_n, B_1^{\perp} \parr \ldots \parr B_n^{\perp}}{\vdots}}{\vdash B_1, B_1^{\perp} \qquad \vdots}}{\vdash B_n, B_n^{\perp}}$$

This gives the *only if* part of the proof.

The *if* direction follows by induction on the length of the proof with cuts involving only hypotheses by rearrangement of the generating steps and constructing the corresponding computation.

The above construction could be further refined so that the proof structure corresponding to a single computation step is embedded into the sequent calculus by introducing new rules in the style of [1], so that no hypotheses would be needed in the proofs.

Note that it is essential to use linear logic with its restrictions on the use of structural rules in order to have a system which is sound. The number of atom occurrences matters in concurrent logic programming goals. This expresses the idea of resources within a concurrent system, mimicked well by the properties of linear logic. In order to illustrate this point, assume sequent $\vdash p^{\perp}, p'$ as a proper axiom. Then $p \vdash p' \otimes p'$ is not provable from the axiom in linear logic, but in a similar situation in intuitionistic logic, $p \vdash p' \wedge p'$ is indeed provable from $p \vdash p'$. In the proof of the theorem above, the expansion of the node $\vdash B_1 \otimes \ldots \otimes B_n, B_1^{\perp} \parr \ldots \parr B_n^{\perp}$ depends on this.

The above does not deal with built-in "extra-logical" predicates used as elementary actions by the programming language. The machinery which concerns them can be included by adding hypotheses of the form

$$b \multimap d$$

for each built-in "b" with defining condition "d". (This only makes real sense in the presence of variables, of course; a set of ground instance sequents covering the behaviour of each of the built-ins would be used then.)

We can, however, do more within just standard linear logic inference rules when adding also the additive connectives & (with) and ⊕ (plus) allowing for explicit handling of parallelism:

$$\frac{\vdash \Gamma, X \quad \vdash \Gamma, Y}{\vdash \Gamma, X \& Y}\ with, \qquad \frac{\vdash \Gamma, X}{\vdash \Gamma, X \oplus Y}\ plus_l, \qquad \frac{\vdash \Gamma, Y}{\vdash \Gamma, X \oplus Y}\ plus_r,$$

and modal operations ! (of course) and ? (why not) allowing controlled use of weakening, contraction and reusability of certain expressions:

$$\frac{\vdash ?\Gamma, X}{\vdash ?\Gamma, !X}\ !, \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?X}\ W, \qquad \frac{\vdash \Gamma, X}{\vdash \Gamma, ?X}\ D, \qquad \frac{\vdash \Gamma, ?X, ?X}{\vdash ?\Gamma, !X}\ C,$$

and negation on them:

$$(X \& Y)^{\perp} = X^{\perp} \oplus Y^{\perp}, \qquad (X \oplus Y)^{\perp} = X^{\perp} \& Y^{\perp}, \qquad (!X)^{\perp} = ?X^{\perp}, \qquad (?X)^{\perp} = !X^{\perp}.$$

Now program $\mathcal{P}$ can be translated into a single linear logic expression P_{LL}, instead of just a bunch of sequents: after translating clauses of a concurrent logic program into formulas $s_1, \ldots, s_j$, form a linear logic expression

$$P_{LL} = s_1 \& \ldots \& s_j.$$

State transitions will not only explicitly involve the states, but also the program. State transition from S to S' will be provability of

$$!P_{LL}, h' \vdash h$$

from the set of sequents defining guards. Now the nondeterministic choice inference rules for $\oplus_l$ and $\oplus_r$ handle clause selection on the level of proof search, without resorting to moving the program clauses away from the expression level itself; no extra proof-search limiting machinery is needed. It is not difficult to see that an analogy of the previous theorem holds true:

Theorem 4. *Let P_{LL} be the linear logic expression obtained from the guarded clauses of a concurrent program $\mathcal{P}$ and connected together by the "with" connective, and let Ω be the set of formulas defining the guard symbols. Then there is a (partial) computation of program $\mathcal{P}$ on a goal $p_1, \ldots, p_k$ ending in state $<q_1, \ldots, q_l>$ iff there is a proof of $\vdash ?P_{LL}^{\perp}, p_1 \otimes \ldots \otimes p_k, q_1^{\perp} \parr \ldots \parr q_l^{\perp}$ from the set of sequents Ω.*

3.3 Variables and communication channels

The picture for atoms being just propositional symbols gives the overall structure of the corresponding linear logic proofs, but it still misses the interaction between the concurrent processes, realized via variables.

The extension aiming to cover this case means introduction of yet other operators from linear logic — the quantifiers:

$$\frac{\vdash \Gamma, X}{\vdash \Gamma, \bigwedge x X} \textit{ for all, } x \text{ not free in } \Gamma, \qquad \frac{\vdash \Gamma, X[x := t]}{\vdash \Gamma, \bigvee x X} \textit{ there is,}$$

$$(\bigwedge xA)^{\perp} = \bigvee xA^{\perp}, \qquad (\bigvee xA)^{\perp} = \bigwedge xA^{\perp}.$$

Now the "program" becomes

$$!\bigwedge \vec{x} P_{LL}$$

and the existence of a transition between parallel system states is linked to provability of

$$!\bigwedge \vec{x} P_{LL}, h' \vdash h,$$

that is when using just one-sided sequents,

$$\vdash ?\bigvee \vec{x} P_{LL}^{\perp}, h, h'^{\perp}.$$

Note that now a program together with a process goal executing over this program can be understood as a *module*, quite in the sense of Girard's idea of plugging of units together [4]. Indeed, this is done by sharing a common variable between the formulas denoting a process to be executed: provability of

$$\vdash ?\bigvee \vec{x} P_{LL}^{\perp}, h(x), h'^{\perp}(\ldots), ?\bigvee \vec{x} Q_{LL}^{\perp}, f(x), f'^{\perp}(\ldots)$$

is thus linked with

$$\vdash ?\bigvee \vec{x} (P_{LL}^{\perp} \oplus Q_{LL}^{\perp}), ((h \otimes f)(x)), (h'(\ldots) \otimes f'(\ldots))^{\perp}.$$

(To a certain extent, the built-in set of predicates constituting the additional part of the inference machinery can also be specified as a "module" of this kind.)

Of course, this model still covers just the asynchronous part of the computations (no synchronization aspects) and it is only correct for deadlock-free programs.

3.4 Example

Let us show a simple example of the outlined proof-theoretic semantics compared to the interleaving semantics based on transformation systems. We will consider a factorial program defined via a pair of producer-consumer processes:

$$
\begin{aligned}
&fact(N,R) \leftarrow seq(1,N,Ss), times(Ss,1,R).\\
&seq(F,T,Os) \leftarrow F > T \mid Os = [\,].\\
&seq(F,T,Os) \leftarrow F \le T \mid Os = [F \mid OS1],\\
&\qquad\qquad\qquad\qquad F1 := F+1,\\
&\qquad\qquad\qquad\qquad seq(F1,T,Os1).\\
&times([X \mid Xs],B,R) \leftarrow B1 := B \times X,\\
&\qquad\qquad\qquad\qquad\qquad\quad times(Xs,B1,R).\\
&times([\,],B,R) \leftarrow R = B.
\end{aligned}
$$

When executing, e.g., $\leftarrow fact(3,X)$, there are several possible execution sequences within the transformation system interleaving approach here, two extreme cases being (we skip the output substitution part of the states):

- $fact(3,R) \Rightarrow seq(1,3,Ss), times(Ss,1,R) \Rightarrow$
  $seq(2,3,Ss1), times([1 \mid Ss1],1,R) \Rightarrow$
  $seq(2,3,Ss1), times(Ss1,1,R) \Rightarrow$
  $seq(3,3,Ss2), times([2 \mid Ss2],1,R) \Rightarrow$
  $seq(3,3,Ss2), times(Ss2,2,R) \Rightarrow$
  $seq(4,3,Ss3), times([3 \mid Ss3],2,R) \Rightarrow$
  $seq(4,3,Ss3), times(Ss3,6,R) \Rightarrow times([\,],6,R) \Rightarrow R = 6.$
- $fact(3,R) \Rightarrow seq(1,3,Ss), times(Ss,1,R) \Rightarrow$
  $seq(2,3,Ss1), times([1 \mid Ss1],1,R) \Rightarrow$
  $seq(3,3,Ss2), times([1,2 \mid Ss2],1,R) \Rightarrow$
  $seq(4,3,Ss3), times([1,2,3 \mid Ss3],1,R) \Rightarrow times([1,2,3],1,R) \Rightarrow$
  $times([2,3],1,R) \Rightarrow times([3],2,R) \Rightarrow times([\,],6,R) \Rightarrow R = 6.$

Within the proof-theoretic semantics using linear logic, the clauses translate into

$$
\begin{aligned}
&Fact(n,r) \multimap (Seq(1,n,s) \otimes Times(s,1,r))\\
&Seq(f,t,o) \multimap (f > t \otimes o = [\,])\\
&Seq(f,t,o) \multimap (f \le t \otimes o = [f \mid o_1] \otimes f_1 = f+1 \otimes Seq(f_1,t,o_1))\\
&Times([x \mid y],b,r) \multimap (b_1 = b \times x \otimes Times(y,b_1,r))\\
&Times([\,],b,r) \multimap (r = b).
\end{aligned}
$$

The proof tree corresponding to parallel computation of the factorial of 3 constitutes a partial order semantics comprising all possible interleavings. The very basic structure of it, indicating just the "processes" involved, without the rest of the steps of the inference formalism, can be depicted

as follows (additional leaves would correspond to clause selection, guard checking, and built-ins):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash Seq(4,3,[])}\quad\dfrac{}{\vdash Times([],2,6)}}{\vdash Seq(3,3,[3])}\quad\dfrac{}{\vdash Times([3],2,6)}}{\vdash Seq(2,3,[2,3])}\quad\dfrac{}{\vdash Times([2,3],1,6)}}{\vdash Seq(1,3,[1,2,3])}\quad\dfrac{}{\vdash Times([1,2,3],1,6)}}{\vdash Fact(3,6)}$$

4 Conclusion

We have presented a simple proof-theoretic definition of a partial-order semantics for concurrent logic programs based on a variant of Flat Concurrent Prolog. We have shown that when using linear logic instead of the classical one, this semantics retains the "logical" reading of concurrent logic programs when linear logic is used as the underlying mechanism. The correspondence is a suprisingly close one, and the tools for proof control offered by linear logic fit well for this class of programs.

References

1. J.-M. Andreoli and R. Pareschi: Logic programming with sequent systems. A linear logic approach. In: Extensions of Logic programming, P. Schroeder-Heister, ed., LNAI 475, Springer, Berlin, 1991.
2. J.-Y. Girard: Linear logic, Theoretical Computer Science, 50(1), 1987.
3. J.-Y. Girard: Linear logic: A survey. Logic, Algebra and Computation, Marktoberdorf Summer School, 1991.
4. J.-Y. Girard, Y. Lafont, and P. Taylor: Proofs and Types, Cambridge University Press, Cambridge, 1989.
5. S. Gregory: Parallel Logic Programming in PARLOG: The language and its implementation. Addison-Wesley, Reading, MA, 1987.
6. G. Huet: A uniform approach to type theory, in: Logical Foundations of Functional programming, Addison-Wesley, Reading, MA, 1990, 337-397.
7. J. W. Lloyd: Foundations of Logic Programming. Springer, Berlin, 1984.
8. R. Kowalski: Logic for Problem Solving. North-Holland, Amsterdam, 1979.
9. N. Martí-Oliet and J. Meseguer: From Petri nets to linear logic, in: Category Theory and Computer Science, D. H. Pitt, et. al., ed., LNCS 389, Springer, 1989, 313-340.
10. J. Messeguer: Rewriting as a unified model of concurrency, in: Proc. CONCUR'90, J. C. M. Baeten and J. W. Klop, eds., LNCS 458, Springer, 1990, 384-400.
11. E. Shapiro, (ed.): Concurrent Prolog: Collected Papers., Vols 1 & 2, MIT Press, 1987.
12. E. Shapiro: The family of concurrent programming languages, ACM Computing Surveys, 1989, 413-510.
13. J. Zlatuška: Committed-choice logic programming in linear logic, in: Computational Logic and Proof Theory (Third Kurt Gödel Colloquium, KGC'93; G. Gottlob, A. Leitsch, and D. Mundici, eds.), LNCS 713, Springer, 1993, 337-348.