

# Partial Deduction for Linear Logic—The Symbolic Negotiation Perspective

Peep K ungas<sup>1</sup>, Mihhail Matskin<sup>2</sup>

<sup>1</sup> Norwegian University of Science and Technology  
Department of Computer and Information Science  
Trondheim, Norway  
`peep@idi.ntnu.no`

<sup>2</sup> Royal Institute of Technology  
Department of Microelectronics and Information Technology  
Kista, Sweden  
`misha@imit.kth.se`

**Abstract.** It has been demonstrated earlier [8] how symbolic negotiation could be presented using Partial Deduction (PD) in Linear Logic (LL). However, the previous papers didn't provide formalisation of the PD process in LL. In this paper we fill the gap by providing formalisation of PD for !-Horn fragment of LL. The framework can be easily adapted to other fragments of LL. We consider soundness and completeness of the formalism. It turns out that, given a certain PD procedure, PD for LL in !-Horn fragment is sound and complete.

## 1 Introduction

Partial Deduction (PD) (or partial evaluation of logic programs, which was first introduced by Komorowski [7]) is known as one of optimisation techniques in logic programming. Given a logic program, PD derives a more specific program while preserving the meaning of the original program. Since the program is more specialised, it is usually more efficient than the original program.

For instance, let  $A$ ,  $B$ ,  $C$  and  $D$  be propositional variables and  $A \rightarrow B$ ,  $B \rightarrow C$  and  $C \rightarrow D$  computability statements in a logical framework. Then possible partial deductions are  $A \rightarrow C$ ,  $B \rightarrow D$  and  $A \rightarrow D$ . It is easy to notice that the first corresponds to forward chaining (from facts to goals), the second to backward chaining (from goals to facts) and the third could be either forward or backward chaining or even their combination.

Although the original motivation behind PD was deduction of specialised logic programs with respect to a given goal, our motivation for PD is a bit different. Namely, it turns out that PD could be applied to finding partial solutions of problems written in logical formalisms. In our case, given the formal specification of a problem, if we fail to solve the entire problem, we apply PD to generate partial solutions.

This approach supports detection of subgoals during distributed problem solving. If a single agent fails to solve a problem, PD is applied to solve the problem partially. As a result subproblems are detected, which could be solved further

by other agents. This would lead to a distributed problem solving mechanism, where different agents contribute to different phases in problem solving—each agent applies PD to solve a fragment of the problem and forwards the modified problem to others. As a result the problem becomes solved in the distributed manner. Usage of PD in such a way provides foundations for advance interactions between agents.

As a logical formalism for application of PD we use Linear Logic [2]. LL has been advocated [4] to be a computation-oriented logic and, because of its computation-oriented nature, LL has been applied to symbolic multi-agent negotiation in [8].

Although PD has been formalised for several frameworks, including fluent calculus [9], normal logic programs [13], etc., it turns out that there is no work considering PD for LL. Our goal is to fill this gap by providing a formal foundation of PD for LL as a framework for symbolic negotiation between agents such as it was introduced in [8].

The rest of the paper is organised as follows. Section 2 gives a short introduction to LL. Section 3 gives basic definitions of PD. Section 4 focuses on proofs of soundness and completeness of PD for !-Horn fragment of LL (HLL) [4]. Section 5 demonstrates the relationship between PD and symbolic negotiation. In Section 6 we review some of the PD strategies, which could be applied for guiding PD. Section 7 reviews the related work and Section 8 concludes the paper and discusses further research directions.

## 2 Linear logic

LL is a refinement of classical logic introduced by J.-Y. Girard to provide means for keeping track of “resources”. In LL two assumptions of a propositional constant  $A$  are distinguished from a single assumption of  $A$ . This does not apply in classical logic, since there the truth value of a fact does not depend on the number of copies of the fact. Indeed, LL is not about truth, it is about computation.

We consider !-Horn fragment of LL (HLL) [4] consisting of multiplicative conjunction ( $\otimes$ ), linear implication ( $\multimap$ ) and “of course” operator (!). In terms of resource acquisition the logical expression  $A \otimes B \vdash C \otimes D$  means that resources  $C$  and  $D$  are obtainable only if both  $A$  and  $B$  are obtainable. After the sequent has been applied,  $A$  and  $B$  are consumed and  $C$  and  $D$  are produced.

While implication  $A \multimap B$  as a computability statement clause in HLL could be applied only once,  $!(A \multimap B)$  may be used an unbounded number of times. When  $A \multimap B$  is applied, then literal  $A$  becomes deleted from and  $B$  inserted to the current set of literals. If there is no literal  $A$  available, then the clause cannot be applied. In HLL ! cannot be applied to formulae other than linear implications.

In order to illustrate some other features of LL, not presented in HLL, we can consider the following LL sequent from [11]— $(D \otimes D \otimes D \otimes D \otimes D) \vdash (H \otimes C \otimes (O \& S) \otimes !F \otimes (P \oplus I))$ , which encodes a fixed price menu in a fast-food restaurant: for 5 dollars ( $D$ ) you can get an hamburger ( $H$ ), a coke ( $C$ ), either

onion soup  $O$  or salad  $S$  depending, which one *you* select, all the french fries ( $F$ ) you can eat plus a pie ( $P$ ) or an ice cream ( $I$ ) depending on availability (restaurant owner selects for you). The formula  $!F$  here means that we can use or generate a resource  $F$  as much as we want—the amount of the resource is unbounded.

Since HLL could be encoded as a Petri net, then theorem proving complexity in HLL is equivalent to the complexity of Petri net reachability checking and therefore decidable [4]. Complexity of other LL fragments have been summarised by Lincoln [12].

### 3 Basics of partial deduction

In this section we present the definitions of the basic concepts of partial deduction for HLL.

#### 3.1 Basic definitions

**Definition 1.** *A program stack is a multiplicative conjunction*

$$\bigotimes_{i=1}^n A_i,$$

where  $A_i, i = 1 \dots n$  is a literal.

**Definition 2.** *Mapping from a multiplicative conjunction to a set of conjuncts is defined as follows:*

$$\left[ \bigotimes_i^n A_i \right] = \{A_1, \dots, A_n\}$$

**Definition 3.** *Consumption of formula  $A_i$  from a program stack  $S$  is a mapping*

$$A_1 \otimes \dots \otimes A_{i-1} \otimes A_i \otimes A_{i+1} \otimes \dots \otimes A_n \mapsto_{S, A_i} A_1 \otimes \dots \otimes A_{i-1} \otimes A_{i+1} \otimes \dots \otimes A_n,$$

where  $A_j, j = 1 \dots n$  could be any valid formula in LL.

**Definition 4.** *Generation of formula  $A_i$  to a program stack  $S$  is a mapping*

$$A_1 \otimes \dots \otimes A_{i-1} \otimes A_{i+1} \otimes \dots \otimes A_n \mapsto_{S, A_i} A_1 \otimes \dots \otimes A_{i-1} \otimes A_i \otimes A_{i+1} \otimes \dots \otimes A_n,$$

where  $A_j, j = 1 \dots n$  and  $A_i$  could be any valid formulae in LL.

**Definition 5.** *A Computation Specification Clause (CSC) is a LL sequent*

$$\vdash I \multimap_f O,$$

where  $I$  and  $O$  are multiplicative conjunctions of any valid LL formulae and  $f$  is a function, which implements the computation step.  $I$  and  $O$  are respectively consumed and generated from the current program stack  $S$ , when a particular CSC is applied.

It has to be mentioned that a CSC can be applied only, if  $[I] \subseteq [S]$ . Although in HLL CSCs are represented as linear implication formulae, we represent them as extralogical axioms in our problem domain. This means that an extralogical axiom  $\vdash I \multimap_f O$  is basically equal to HLL formula  $!(I \multimap_f O)$ .

**Definition 6.** A *Computation Specification (CS)* is a finite set of CSCs.

**Definition 7.** A *Computation Specification Application (CSA)* is defined as

$$\Gamma; S \vdash G,$$

where  $\Gamma$  is a CS,  $S$  is the initial program stack and  $G$  the goal program stack.

**Definition 8.** *Resultant* is a CSC

$$\vdash I \multimap_{\lambda a_1, \dots, a_n. f} O, n \geq 0,$$

where  $f$  is a term representing a function, which generates  $O$  from  $I$  by applying potentially composite functions over  $a_1, \dots, a_n$ .

CSA determines which CSCs could be applied by PD steps to derive resultant  $\vdash S \multimap_{\lambda a_1, \dots, a_n. f} G, n \geq 0$ . It should be noted that resultants are derived by applying PD steps to the CSAs, which are represented in form  $A \vdash B$ . The CSC form is achieved from particular programs stacks by implicitly applying the following inference figure:

$$\frac{\frac{\overline{\vdash A \multimap B} \text{ resultant} \quad \frac{\overline{A \vdash A} \text{ Id} \quad \overline{B \vdash B} \text{ Id}}{A, A \multimap B \vdash B} \text{ L } \multimap}}{A \vdash B} \text{ Cut}}$$

While resultants encode computation, program stacks represent computation pre- and postconditions.

### 3.2 PD steps

**Definition 9.** *Forward chaining PD step*  $\mathcal{R}_f(L_i)$  is defined as a rule

$$\frac{B \otimes C \vdash G}{A \otimes C \vdash G} \mathcal{R}_f(L_i)$$

where  $L_i$  is a labelling of CSC  $\vdash A \multimap_{L_i} B$ .  $A, B, C$  and  $G$  are multiplicative conjunctions.

**Definition 10.** *Backward chaining PD step*  $\mathcal{R}_b(L_i)$  is defined as a rule

$$\frac{S \vdash A \otimes C}{S \vdash B \otimes C} \mathcal{R}_b(L_i)$$

where  $L_i$  is a labelling of CSC  $\vdash A \multimap_{L_i} B$ .  $A, B, C$  and  $S$  are multiplicative conjunctions.

PD steps  $\mathcal{R}_f(L_i)$  and  $\mathcal{R}_b(L_i)$ , respectively, apply CSC  $L_i$  to move the initial program stack towards the goal stack or vice versa. In the  $\mathcal{R}_b(L_i)$  inference figure formulae  $B \otimes C$  and  $A \otimes C$  denote respectively an original goal stack  $G$  and a modified goal stack  $G'$ . Thus the inference figure encodes that, if there is an  $\text{CSC} \vdash A \multimap_{L_i} B$ , then we can change goal stack  $B \otimes C$  to  $A \otimes C$ . Similarly, in the inference figure  $\mathcal{R}_f(L_i)$  formulae  $B \otimes C$  and  $A \otimes C$  denote, respectively, an original initial stack  $S$  and its modification  $S'$ . And the inference figure encodes that, if there is a  $\text{CSC} \vdash A \multimap_{L_i} B$ , then we can change initial program stack  $A \otimes C$  to  $B \otimes C$ .

In order to manage access to unbounded resources, we need PD steps  $\mathcal{R}_{C_i}$ ,  $\mathcal{R}_{L_i}$ ,  $\mathcal{R}_{W_i}$  and  $\mathcal{R}_{!_i}(n)$ .

**Definition 11.** PD step  $\mathcal{R}_{C_i}$  is defined as a rule

$$\frac{!A \otimes !A \otimes B \vdash C}{!A \otimes B \vdash C} \mathcal{R}_{C_i}$$

where  $A$  is a literal, while  $B$  and  $C$  are multiplicative conjunctions.

**Definition 12.** PD step  $\mathcal{R}_{L_i}$  is defined as a rule

$$\frac{A \otimes B \vdash C}{!A \otimes B \vdash C} \mathcal{R}_{L_i}$$

where  $A$  is a literal, while  $B$  and  $C$  are multiplicative conjunctions.

**Definition 13.** PD step  $\mathcal{R}_{W_i}$  is defined as a rule

$$\frac{B \vdash C}{!A \otimes B \vdash C} \mathcal{R}_{W_i}$$

where  $A$  is a literal, while  $B$  and  $C$  are multiplicative conjunctions.

**Definition 14.** PD step  $\mathcal{R}_{!_i}(n)$ ,  $n > 0$  is defined as a rule

$$\frac{!A \otimes A^n \otimes B \vdash C}{!A \otimes B \vdash C} \mathcal{R}_{!_i}(n)$$

where  $A$  is a literal, while  $B$  and  $C$  are multiplicative conjunctions.  $A^n = \underbrace{A \otimes \dots \otimes A}_n$ , for  $n > 0$ .

Considering the first-order HLL we have to replace PD steps  $\mathcal{R}_f(L_i)$  and  $\mathcal{R}_b(L_i)$  with their respective first-order variants  $\mathcal{R}_f(L_i(\underline{x}))$  and  $\mathcal{R}_b(L_i(\underline{x}))$ . Other PD steps can remain the same. We also require that the initial and the goal program stack are ground.

**Definition 15.** First-order forward chaining PD step  $\mathcal{R}_f(L_i(\underline{x}))$  is defined as a rule

$$\frac{B \otimes C \vdash G}{A \otimes C \vdash G} \mathcal{R}_f(L_i(\underline{x}))$$







**Lemma 2.** *Resultants in a derivation are nodes in the respective HLL proof tree and they correspond to partial proof trees, where leaves are other resultants.*

*Proof.* Since each resultant  $\vdash A \multimap B$  in a derivation is achieved by an application of a PD step, which is defined with a respective LL inference figure, then it represents a node  $A \vdash B$  in the proof tree, whereas the derivation of  $\vdash A \multimap B$  represents a partial proof tree.

**Theorem 1 (Soundness of propositional PD).** *PD for LL in propositional HLL is sound.*

*Proof.* According to Lemma 1 and Lemma 2 PD for LL in propositional HLL is sound, if we apply propositional PD steps. The latter derives from the fact that, if there exists a derivation  $\vdash S \multimap G \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} \vdash S' \multimap G'$ , then the derivation is constructed by PD in a formally correct manner.

**Theorem 2 (Completeness of propositional PD).** *PD for LL in propositional HLL is complete.*

*Proof.* When applying PD with propositional PD steps, we first generate all possible derivations until no derivations could be found, or all proofs have been found. If  $\text{CSC} \vdash S' \multimap G'$  is executable then according to Lemma 1, Lemma 2 and Definition 19 there should be a path in the HLL proof tree starting with  $\text{CSC} \vdash S \multimap G$ , ending with  $\vdash A \multimap A$  and containing  $\text{CSC} \vdash S' \multimap G'$ . There is no possibility to have a path from  $\text{CSC} \vdash S' \multimap G'$  to  $\vdash A \multimap A$  without having a path from  $\text{CSC} \vdash S \multimap G$  to  $\text{CSC} \vdash S' \multimap G'$  in the same HLL proof tree.

Then according to Lemma 1 and Lemma 2, derivation  $\vdash S \multimap G \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} \vdash S' \multimap G'$  would be either discovered or it will be detected that there is no such derivation. Therefore PD for LL in HLL fragment of LL is complete.

**Theorem 3 (Soundness of PD of a first-order CSA).** *PD for LL in first-order HLL is sound.*

*Proof.* The proof follows the pattern of the proof for Theorem 1, with the difference that instead of applying PD steps  $\mathcal{R}_b(L_i)$  and  $\mathcal{R}_f(L_i)$ , we apply their first-order counterparts  $\mathcal{R}_b(L_i(\underline{x}))$  and  $\mathcal{R}_f(L_i(\underline{x}))$ .

**Theorem 4 (Completeness of PD of a first-order CSA).** *PD for LL in first-order HLL is complete.*

*Proof.* The proof follows the pattern of the proof for Theorem 2, with the difference that instead of applying PD steps  $\mathcal{R}_b(L_i)$  and  $\mathcal{R}_f(L_i)$ , we apply their first-order counterparts  $\mathcal{R}_b(L_i(\underline{x}))$  and  $\mathcal{R}_f(L_i(\underline{x}))$ .

In the general case first-order HLL is undecidable. However, Kanovich and Vauzeilles [5] determine certain constraints, which help to reduce the complexity of theorem proving in first-order HLL. By applying those constraints, theorem proving complexity could be reduced to PSPACE. Propositional HLL is equivalent to Petri net reachability checking, which is according to Mayr [15] decidable.

## 5 Application of PD to symbolic negotiation

In this section we demonstrate usage of PD symbolic negotiation. We consider here communication only between two agents and show only offers, which are relevant to demonstration of our framework. However, in more practical cases possibly more agents can participate and more offers can be exchanged. In particular, if agent  $A$  cannot help agent  $B$  to solve problem then  $A$  might consider contacting agent  $C$ , to get help in solving  $B$ -s problem. This would lead to many concurrently running negotiations.

**Definition 22.** *An agent is defined with a CSA  $\Gamma; S \vdash G$ , where  $\Gamma$ ,  $S$  and  $G$  represent agent's capabilities, what the agent can provide, and what the agent requires, respectively.*

**Definition 23.** *An offer  $A \vdash B$  is a CSC with  $\Gamma \equiv \emptyset$ .*

In our scenario we have two agents—a traveller  $\mathcal{T}$  and an airline company  $\mathcal{F}$ . The goal of  $\mathcal{T}$  is to make a booking (*Booking*). Initially  $\mathcal{T}$  knows only its starting (*From*) and final (*To*) locations. Additionally the agent has two capabilities, *findSchedule* and *getPassword*, for finding a schedule (*Schedule*) for a journey and retrieving a password (*Password*) from its internal database for a particular Web site (*Site*). Goals, resources and capabilities of the traveller  $\mathcal{T}$  are described in LL with the following formulae.

$$G_{\mathcal{T}} = \{Booking\},$$

$$S_{\mathcal{T}} = \{From \otimes To\},$$

$$\Gamma_{\mathcal{T}} = \begin{array}{l} \vdash From \otimes To \multimap_{findSchedule} Schedule, \\ \vdash Site \multimap_{getPassword} Password. \end{array}$$

For booking a flight agent  $\mathcal{T}$  should contact a travel agent or an airline company. The airline company agent  $\mathcal{F}$  does not have any explicit declarative goals that is usual for companies, whose information systems are based mainly on business process models. The only fact  $\mathcal{F}$  can expose, is its company Web site (*Site*). Since the *Site* is unbounded resource (includes !), it can be delivered to customers any number of times.

$\mathcal{F}$  has two capabilities—*bookFlight* and *login* for booking a flight and identifying customers plus creating a secure channel for information transfer. Goals, resources and capabilities of the airline company  $\mathcal{F}$  are described in LL as the following formulae.

$$G_{\mathcal{F}} = \{1\},$$

$$S_{\mathcal{F}} = \{!Site\},$$

$$\Gamma_{\mathcal{F}} = \begin{array}{l} \vdash \text{SecureChannel} \otimes \text{Schedule} \multimap_{\text{bookFlight}} \text{Booking}, \\ \vdash \text{Password} \multimap_{\text{login}} \text{SecureChannel}. \end{array}$$

Given the specification agent  $\mathcal{T}$  derives and sends out the following offer:

$$\text{Schedule} \vdash \text{Booking}.$$

The offer was deduced by PD as follows:

$$\frac{\text{Schedule} \vdash \text{Booking}}{\text{From} \otimes \text{To} \vdash \text{Booking}} \mathcal{R}_f(\text{findSchedule})$$

Since  $\mathcal{F}$  cannot satisfy the proposal, it derives a new offer:

$$\text{Schedule} \vdash \text{Password} \otimes \text{Schedule}.$$

The offer was deduced by PD as follows:

$$\frac{\frac{\text{Schedule} \vdash \text{Password} \otimes \text{Schedule}}{\text{Schedule} \vdash \text{SecureChannel} \otimes \text{Schedule}} \mathcal{R}_b(\text{login})}{\text{Schedule} \vdash \text{Booking}} \mathcal{R}_b(\text{bookFlight})$$

Agent  $\mathcal{T}$  deduces the offer further:

$$\frac{\text{Schedule} \vdash \text{Site} \otimes \text{Schedule}}{\text{Schedule} \vdash \text{Password} \otimes \text{Schedule}} \mathcal{R}_b(\text{getPassword})$$

and sends the following offer to  $\mathcal{F}$ :

$$\text{Schedule} \vdash \text{Site} \otimes \text{Schedule}.$$

For further reasoning in symbolic negotiation, we need the following definitions. They determine the case where two agents can achieve their goals together, by exchanging symbolic information.

**Definition 24.** *An offer  $A \vdash B$  is complementary to an offer  $C \vdash D$ , if  $A \otimes D \vdash B \otimes C$  is a theorem of LL.  $A$ ,  $B$ ,  $C$  and  $D$  represent potentially identical literals.*

The logical justification to merging complementary offers could be given from the global problem solving/theorem proving viewpoint. Having two complementary offers means that although two problems were locally (at a single agent) unsolvable, they have a solution globally (the problems of many agents together).

**Proposition 9.** *If two derived offers are complementary to each-other, then the agents who proposed the initial offers (which led to the complementary offers) can complete their symbolic negotiation by merging their offers.*

*Proof.* Since the left hand side of an offer encodes what an agent can provide and the right hand side of the offer represents what the agent is looking for, then having two offers, which are complementary to each other, we have found a solution satisfying both agents, who sent out the initial offers and whose derivations led to the complementary offers.

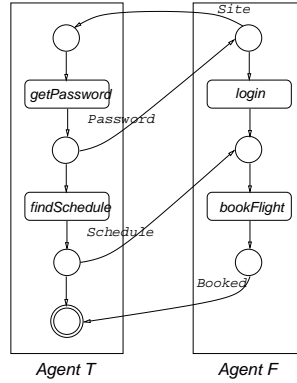
Now agent  $\mathcal{F}$  constructs a new offer:

$$\frac{Site \vdash 1}{!Site \vdash 1} \mathcal{R}_{L_i}$$

However, instead of forwarding it to  $\mathcal{T}$ , it merges the offer with the received complementary offer:

$$\frac{\frac{Site \otimes Schedule \vdash Site \otimes Schedule}{Site \otimes Schedule \vdash Site \otimes Schedule \otimes 1} Id \quad \frac{}{\vdash 1} Axiom}{R \otimes}$$

Thereby  $\mathcal{T}$  composed (with the help of  $\mathcal{F}$ ) a composite service, which execution achieves the goals of agents  $\mathcal{T}$  and  $\mathcal{F}$  (in the current example, the goal of  $\mathcal{F}$  is represented as constant 1). The resulting plan (a side effect of symbolic negotiation) is graphically represented in Figure 1. The rectangles in the figure represent the agent capabilities applied, while circles denote information collection/delivery nodes. The arrows denote symbolic information flow.



**Fig. 1.** The distributed plan.

## 6 Partial deduction strategies

The practical value of PD is very limited without defining appropriate PD strategies. These are called tactics and refer to selection and stopping criteria. Successful tactics depend generally quite much on a specific logic application. Therefore

we only list some possible tactics here. From agent negotiation point of view the strategies represent to some extent agents' policies—they determine which offers are proposed next.

Tammet [18] proposes a set of theorem proving strategies for speeding up LL theorem proving. He also presents experimental results, which indicate a good performance of the proposed strategies. Some of his strategies remind usage of our inference figures. Thus some LL theorem proving strategies are implicitly handled in our PD framework.

We also would like to point out that by using LL inference figures instead of basic LL rules, PD, as we defined it here, could be more efficient than pure LL theorem proving.

**Definition 25.** *Length  $l$  of a derivation is equal to the number of the applications of PD steps  $\mathcal{R}$  in the derivation.*

**Definition 26.** *Two derivations are computationally equivalent, regardless of the length of their derivations, if they both start and end with the same resultant.*

## 6.1 Selection criteria

Selection criteria define which formulae and PD steps should be considered next for derivation of a resultant. We consider the following selection criteria.

- Mixed backward and forward chaining—a resultant is extended by interleaving backward and forward chaining.
- Different search methods—depth-first, breadth-first, iterative deepening, etc could be used. While breadth-first allows discovering shorter derivations faster, depth-first requires less computational overhead, since less memory is used for storing the current search status.
- Prefer resultants with smaller derivation length—the strategy implicitly leads to breadth-first search.
- Apply only one PD step at time.
- Combine several PD steps together. The approach is justified, if there is some domain knowledge available, which states that certain CSCs are executed in sequence.
- Priority-based selection—some literals have a higher weight, which is determined either manually by the user or calculated by the system according to predefined criteria. During PD literals/resultants having higher weights are selected first.

We would like to emphasise that the above criteria are not mutually exclusive but rather complementary to each other.

## 6.2 Stopping criteria

Stopping criteria define when to stop derivation of resultants. They could be combined with the above-mentioned selection criteria. We suggest the following stopping criteria:

- The derived resultant is computationally equivalent to a previous one—since the resultants were already derived and used in other derivations, proceeding PD again with the same resultant does not yield neither new resultants nor unique derivations (which are not computationally equivalent with any previously considered one).
- A generative cycle is detected—if we derived a resultant  $\vdash A \multimap B \otimes C$  from a resultant  $\vdash A \multimap C$ , then by repeatedly applying PD steps between the former resultants we end up with resultants  $\vdash A \multimap B^n \otimes C$ , where  $n > 1$ . Therefore we can skip the PD steps in further derivation and reason analytically how many instances of literal  $B$  we need. The approach is largely identical to Karp-Miller [6] algorithm, which is applied for state space collapsing during Petri net reachability checking. A similar method is also applied by Andreoli et al [1] for analysing LL programs.
- Maximum derivation length  $l$  is reached—given that our computational resources are limited and the time for problem solving is limited, we may not be able to explore the full search space anyway. Then setting a limit to derivation length helps to constrain the search space.
- The resultant is equal to the goal—since we found a solution to the problem, there is no need to proceed further, unless we are interested in other solutions as well.
- Stepwise—the user is queried before each derivation in order to determine, which derivations s/he wants to perform. This stopping criterion could be used during debugging, since it provides the user with an overview of the derivation process.
- Exhaustive—derivation stops, when no new resultants are available.

## 7 Related work

Although PD was first introduced by Komorowski [7], Lloyd and Shepherdson [13] were first who formalised PD for normal logic programs. They showed PD's correctness with respect to Clark's program completion semantics. Since then several formalisations of PD for different logic formalisms have been developed. Lehmann and Leuschel [9] developed a PD method capable of solving planning problems in the fluent calculus. A Petri net reachability checking algorithm is used there for proving completeness of the PD method.

Analogically Leuschel and Lehmann [10] applied PD of logic programs for solving Petri net coverability problems while Petri nets are encoded as logic programs. De Schreye et al [17] presented experiments related to the preceding mechanisms by Lehmann and Leuschel, which support evaluation of certain PD control strategies.

Matskin and Komorowski [14] applied PD to automated software synthesis. One of their motivations was debugging of declarative software specification. The idea of using PD for debugging is quite similar to the application of PD in symbolic agent negotiation [8]. In both cases PD helps to determine computability statements, which cannot be solved by a system.

Our formalism for PD, through backward chaining PD step, relates to abduction. Given the simplification that induction is abduction together with justification, PD relates to induction as well. An overview of inductive logic programming (ILP) is given by Muggleton and de Raedt [16].

Forward and backward chaining for linear logic have been considered by Harland et al [3] in the logic programming context. In this article we define backward and forward chaining in PD context. Indeed, the main difference between our work and the work by Harland et al could be characterised with a different formalism for different purposes.

There is a similarity between the ideology behind an inductive bias in ILP and a strategy in PD. This means that we could adapt some ILP inductive biases as strategies for PD. In ILP  $\theta$ -subsumption is defined to order clauses partially and to generate a lattice of clauses. For instance clause  $parent(X, Y) \leftarrow mother(X, Y), mother(X, Z)$   $\theta$ -subsumes clause  $parent(X, Y) \leftarrow mother(X, Y)$ . The approach could be useful a PD strategy in our formalism. However, the idea has not been evaluated yet.

## 8 Conclusions

In this paper we formalised PD for LL, more specifically for !-Horn fragment of LL. The main reason for choosing the particular LL fragment was that (!)Horn fragment of LL has been designed for rule-based applications. Therefore it suits well for formalising symbolic negotiation.

We proved that for both propositional and first-order HLL the PD method is sound and complete. It was also demonstrated how PD could be applied in symbolic negotiation. The theorems proposed here can be easily adapted for other fragments of LL, relevant to symbolic negotiation.

Indeed, we have implemented an agent system, where PD is applied for symbolic negotiation. The system is based on JADE and can be download from <http://www.idi.ntnu.no/~peep/symbolic>. Although in the current version of the agent software the derived offers are broadcasted, we are working on heuristics, which would allow us to limit the number of offer receivers. In the long term we would like to end up with a P2P agent software where a large number of agents would apply symbolic negotiation for concurrent problem solving.

## Acknowledgements

This work was partially supported by the Norwegian Research Foundation in the framework of Information and Communication Technology (IKT-2010) program—the ADIS project. Additionally I would like to thank the anonymous referees for their comments.

## References

1. J.-M. Andreoli, R. Pareschi, T. Castagnetti. Static Analysis of Linear Logic Programming. *New Generation Computing*, Vol. 15, pp. 449–481, 1997.

2. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, Vol. 50, pp. 1–102, 1987.
3. J. Harland, D. Pym, M. Winikoff. Forward and Backward Chaining in Linear Logic. In *Proceedings of the CADE-17 Workshop on Proof-Search in Type-Theoretic Systems*, Pittsburgh, June 20–21, 2000. *Electronic Notes in Theoretical Computer Science*, Vol. 37, 2001.
4. M. I. Kanovich. Linear Logic as a Logic of Computations. *Annals of Pure and Applied Logic*, Vol. 67, pp. 183–212, 1994.
5. M. I. Kanovich, J. Vauzeilles. The Classical AI Planning Problems in the Mirror of Horn Linear Logic: Semantics, Expressibility, Complexity. *Mathematical Structures in Computer Science*, Vol. 11, No. 6, pp. 689–716, 2001.
6. R. M. Karp, R. E. Miller. Parallel program schemata. *Journal of Computer and Systems Sciences*, Vol. 3, No. 2, pp. 147–195, May 1969.
7. J. Komorowski. A Specification of An Abstract Prolog Machine and Its Application to Partial Evaluation. PhD thesis, Technical Report LSST 69, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1981.
8. P. Küngas, M. Matskin. Linear Logic, Partial Deduction and Cooperative Problem Solving. To appear in *Proceedings of the First International Workshop on Declarative Agent Languages and Technologies (in conjunction with AAMAS 2003), DALT'2003*, Melbourne, Australia, July 15, 2003, Springer-Verlag.
9. H. Lehmann, M. Leuschel. Solving Planning Problems by Partial Deduction. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning, LPAR'2000*, Reunion Island, France, November 11–12, 2000, *Lecture Notes in Artificial Intelligence*, Vol. 1955, pp. 451–467, 2000, Springer.
10. M. Leuschel, H. Lehmann. Solving Coverability Problems of Petri Nets by Partial Deduction. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'2000*, Montreal, Canada, September 20–23, 2000, pp. 268–279, 2000, ACM Press.
11. P. Lincoln. Linear Logic. *ACM SIGACT Notices*, Vol. 23, No. 2, pp. 29–37, Spring 1992.
12. P. Lincoln. Deciding Provability of Linear Logic Formulas. In J.-Y. Girard, Y. Lafont, L. Regnier (eds). *Advances in Linear Logic*, London Mathematical Society Lecture Note Series, Vol. 222, pp. 109–122, 1995.
13. J. W. Lloyd, J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, Vol. 11, pp. 217–242, 1991.
14. M. Matskin, J. Komorowski. Partial Structural Synthesis of Programs. *Fundamenta Informaticae*, Vol. 30, pp. 23–41, 1997.
15. E. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM Journal on Computing*, Vol. 13, No. 3, pp. 441–460, 1984.
16. S. Muggleton, L. de Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, Vol. 19/20, pp. 629–679, 1994.
17. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, M. H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorithms and Experiments. *Journal of Logic Programming*, Vol. 41, No. 2–3, pp. 231–277, 1999.
18. T. Tammet. Proof Strategies in Linear Logic. *Journal of Automated Reasoning*, Vol. 12, pp. 273–304, 1994.