Encoding of Planning Problems and their Optimizations in Linear Logic

Lukáš Chrpa, Pavel Surynek and Jiří Vyskočil

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague
{chrpa,surynek,vyskocil}@kti.mff.cuni.cz

Abstract. Girard's Linear Logic is a formalism which can be used to manage a lot of problems with consumable resources. Its expressiveness is quite good for an easily understandable encoding of many problems. We concentrated on expressing planning problems by linear logic in this paper. We observed a rich usage of a construct of consumable resources in planning problem formulations. This fact motivates us to provide a possible encoding of planning problems in linear logic. This paper shows how planning problems can be encoded in Linear Logic and how some optimizations of planning problems can be encoded. These optimizations can help planners to improve the efficiency of finding solutions (plans).

1 Introduction

Linear Logic is a formalism which can be used to formalize many problems with consumable resources [4]. There is a lot of such problems that handle with consumable and renewable resources in practice. Linear Logic gives us a good expressiveness that help us with formalization of these problems, because these problems can be usually encoded in formulae with linear size with respect to the length of the problems.

Previous research showed that planning problems can be encoded in many different formalisms, usually related to logic. It was showed in [16] that planning problems can be solved by reduction to SAT. This approach brings very good results (for example SATPLAN [17] won last two International Planning Competitions (IPC)[1] in optimal deterministic planning). However, there seems to be a problem with extending the SAT based planners to be able to solve planning problems with time and resources. One of the influential works regarding first order logic is a framework for programming reasoning agents called FLUX [26] which is based on Fluent Calculus. Another logic formalism which can be used in connection to planning problems is Temporal Logic (especially Simple Temporal Logic). It has been showed in [1] that Temporal Logic can be used for improvement of searching for plans. This approach is used in several planners (for example TALplanner [8] which gathered very good results in IPC 2002).

[1] http://ipc.icaps-conference.org

Temporal Logic is good for representing relationships between actions, but not so good to represent whole planning problems.

Unlike other logics Linear Logic has a good expressiveness for formalization of whole planning problems including planning problems with resources etc. Linear Logic related research made during the last twenty years [12] brought many interesting results, however mainly in a theoretical area. One of the most important results was a connectivity of Linear Logic with Petri Nets [23] which gave us more sence of the good expressiveness of Linear Logic.

Instead of the encoding of Petri nets in Linear Logic, there is a possibility of encoding of planning problems in Linear Logic. Planning problems are an important branch of AI and they are very useful in many practical applications. Previous research showed that planning problems can be simply encoded in Linear Logic and a problem of plan generation can be reduced to a problem of finding a proof in linear logic. First ideas of solving planning problems via theorem proving in Linear Logic has been shown at [3, 15, 22]. These papers showed that M?LL fragment of Linear Logic is strong enough to formalize planning problems. Another interesting work in this area is [7] and it describes a possibility of a recursive application of partial plans. Analyzing planning problems encoded in Linear Logic via partial deduction approach is described in [19].

In the area of implementations of Linear Logic we should mention Linear Logic programming. Linear Logic Programming is derived from 'classical' logic programming (Prolog based) by including linear facts and linear operators. There are several Linear Logic programming languages, for example Lolli [14] or LLP [2]. However, all Linear Logic Programming languages are based on Horn's clauses which means that only one predicate can appear on the right side of implication. In fact, this forbids the obtaining of resources (resources can be only spent) which causes an inapplicability of Linear Logic programming languages in solving planning problems. Instead of Linear Logic programming languages there are several Linear Logic provers, for example llprover [24]. Linear Logic provers seem to be strong enough to solve planning problems, but the existing ones are very slow and practically almost unusable.

In the other hand, one of the most important practical results is an implementation of a planning solver called RAPS [20] which in comparison between the most successful planners in IPC 2002 showed very interesting results (especially in Depots domain). RAPS showed that the research in this area can be helpful and can provide an improvement for planners.

This paper extends the previous research in this area [5, 22] by providing detailed description of encoding of planning problems into Linear Logic. In addition, we provide an encoding of many optimizations of planning problems, which helps a planner to find a solution more quickly. Furthermore, we designed two algorithms for actions assemblage that benefit from the Linear Logic encoding. Finally, we show how planning problems with resources can be encoded to Linear Logic and how Linear Logic can help in planning under uncertainty.

We provide a short introduction to Linear Logic and to planning problems in Section 2. Description of the pure encoding of planning problems in Linear

Logic and an extension of this encoding which works with negative predicates is provided in Section 3. Description of the encoding of optimizations such as encoding of static predicates, blocking actions, enforcing actions and assembling actions is provided in Section 4. Section 5 describes how planning with resources can be encoded in Linear Logic. Section 6 describes how planning under uncertainty can be encoded in Linear Logic. Section 7 concludes and presents a possible future research in this area.

2 Preliminaries

This section presents some basic information about Linear Logic and planning problems that helps the reader to get through this paper.

2.1 Linear Logic

Linear Logic was introduced by Girard in 1987 [10]. Linear Logic is often called 'logic of resources', because unlike the 'classical' logic, Linear Logic can handle with expendable resources. The main difference between 'classical' logic and Linear Logic results from an expression saying: "From A, A imply B we obtain B", in 'classical' logic A is still available after B is derived, but in Linear Logic A consumed after B is derived which means that A is no longer available (see $\mathbf{L}{\multimap}$ rule in table 2.1).

In addition to the implication ($\multimap$), there are more operators in linear logic. However we mention only those that are relevant for our encoding. One of the operators is a multiplicative conjunction ($\otimes$) whose meaning is consuming (on the left side of the implication, see $\mathbf{L}\otimes$ rule in table 2.1) or obtaining (on the right side of the implication, see $\mathbf{R}\otimes$ rule in table 2.1) resources together. Another operator is an exponential (!), which converts a linear fact (expendable resource) to a 'classical' fact (not expendable resource). At last, there are an additive conjunction (&) and an additive disjunction ($\oplus$) whose meaning is close to modalities. Exactly when the additive conjunction is used on the right side of the implication, then only one alternative must be proved (see $\mathbf{R}\&$ rule in table 2.1) , but when the additive disjunction is used on the right side of the implication, then all the alternatives must be proved (see $\mathbf{R}\oplus$ rule in table 2.1). If the additives are on the left side of the implication, proving is quite similar, only the meaning of the additive conjunction and additive disjunction is swapped (see $\mathbf{L}\&$ and $\mathbf{L}\otimes$ rules in table 2.1).

Proving in Linear Logic is quite similar to proving in the 'classical' logic, which is based on Gentzen's style. Part of Linear Logic calculus needed to follow this paper is described in Table 2.1. The complete calculus of Linear Logic can be found in [10, 11, 21].

2.2 Planning problems

This subsection brings us some basic introduction which is needed to understand basic notions frequently used in a connection to planning problems.

Id	$A \vdash A$	$\Gamma \vdash \top, \Delta$	$\mathbf{R}\top$
$\mathbf{L}\otimes$	$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, (A \otimes B) \vdash \Delta}$	$\frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2 \vdash B, \Delta_2}{\Gamma_1, \Gamma_2 \vdash (A \otimes B), \Delta_1 \Delta_2}$	$\mathbf{R}\otimes$
$\mathbf{L}\multimap$	$\frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2, (A \multimap B) \vdash \Delta_1 \Delta_2}$		
$\mathbf{L}\&$	$\frac{\Delta, A \vdash \Gamma}{\Delta, (A \& B) \vdash \Gamma} \quad \frac{\Delta, B \vdash \Gamma}{\Delta, (A \& B) \vdash \Gamma}$	$\frac{\Delta, A \vdash \Gamma \quad \Delta, B \vdash \Gamma}{\Delta, (A \oplus B) \vdash \Gamma}$	$\mathbf{L}\oplus$
$\mathbf{R}\&$	$\frac{\Delta \vdash A, \Gamma \quad \Delta \vdash B, \Gamma}{\Delta \vdash (A \& B), \Gamma}$	$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash (A \oplus B), \Delta} \quad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash (A \oplus B), \Delta}$	$\mathbf{R}\oplus$
W !	$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta}$	$\frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}$	**C !**
D !	$\frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}$	$\frac{\Gamma, A \vdash \Delta}{\Gamma, (\forall x) A \vdash \Delta}$	**Forall**

Table 1. Fragment of Linear Logic calculus.

In this paper we consider action-based planning problems like Block World, Depots, Logistics etc. These planning problems are based on worlds containing objects (boxes, robots, etc.), locations (depots, platforms, etc.) etc. Relationships between objects and places (Box 1 is on Box2, Robot 1 is in Depot 2, etc.) are described by predicates. The worlds can be changed only by performing of actions. The next definitions define notions and notations well known in classical action-based planning problems.

Definition 1. Assume that $L = \{p_1, \ldots, p_n\}$ is a finite set of predicates. *Planning domain* Σ over L is a 3-tuple (S, A, γ) where:

- $S \subseteq 2^L$ is a set of states. $s \in S$ is a state. If $p \in s$ then p is true in s and if $p \notin s$ then p is not true in s.
- A is a set of actions. Action $a \in A$ is a 4-tuple $(p^+(a), p^-(a), e^-(a), e^+(a))$ where $p^+(a) \subseteq L$ is a positive precondition of action a, $p^-(a) \subseteq L$ is a negative precondition of action a, $e^-(a) \subseteq L$ is a set of negative effects of action a and $e^+(a) \subseteq L$ is a set of positive effects of action a and $e^-(a) \cap e^+(a) = \emptyset$.
- $\gamma : S \times A \to S$ is a transition function. $\gamma(s, a) = (s - e^-(a)) \cup e^+(a)$ if $p^+(a) \subseteq s$ and $p^-(a) \cap s = \emptyset$.

Remark. When a planning domain contains only actions without negative precondition we denote such an action by 3-tuple $(p(a), e^-(a), e^+(a))$ where $p(a)$ is a positive precondition.

Definition 2. *Planning problem* P is a 3-tuple (Σ, s_0, g) such that:

- $\Sigma = (S, A, \gamma)$ is a planning domain over L.
- $s_0 \in S$ is an initial state.
- $g \subseteq L$ is a set of goal predicates.

Definition 3. *Plan* π is an ordered sequence of actions $< a_1, \ldots, a_k >$ such that, plan π *solves* planning problem P if and only if there exists an appropriate sequence of states $< s_0, \ldots, s_k >$ such that $s_i = \gamma(s_{i-1}, a_i)$ and $g \subseteq s_k$. Plan π is *optimal* if and only if for each $\pi' \mid \pi \mid \leq \mid \pi' \mid$ is valid (an optimal plan is the shortest plan solving a particular planning problem).

To get deeper insight about planning problems, see [9].

3 Planning in Linear Logic

In this section, we show that Linear Logic is a good formalism for encoding planning problems. Main idea of the encoding is based on the fact that predicates in planning problems can be represented as linear facts that can be consumed or obtained depending on performed actions.

3.1 Basic encoding of planning problems

Idea of a reduction of the problem of plan generation to finding a proof in Linear Logic was previously studied at [5, 15, 22]. At first we focus on such planning problems whose actions do not contain negative preconditions. As it was mentioned above, the predicates in planning problems can be encoded as the linear facts. Let $s = \{p_1, p_2, \ldots, p_n\}$ be a state, its encoding in Linear Logic is following:

$$(p_1 \otimes p_2 \otimes \ldots \otimes p_n)$$

Let $a = \{p(a), e^-(a), e^+(a)\}$ be an action, its encoding in Linear Logic is following:

$$\forall p_i \in p(a) \setminus e^-(a), 1 \leq i \leq l; \forall r_j \in e^-(a), 1 \leq j \leq m; \forall s_k \in e^+(a), 1 \leq k \leq n$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes r_1 \otimes r_2 \otimes \ldots \otimes r_m) \multimap (p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes s_1 \otimes s_2 \otimes \ldots \otimes s_n)$$

Performing of action a can be reduced to the proof in Linear Logic in following way (Γ and Δ represent multiplicative conjunctions of literals and the vertical dots represent the previous part of the proof):

$$\frac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, p_1, \ldots p_l, s_1, \ldots, s_n \vdash \Delta\end{array}}{\Gamma, p_1, \ldots p_l, r_1, \ldots, r_m, ((p_1 \otimes \ldots \otimes p_l \otimes r_1 \otimes \ldots \otimes r_m) \multimap (p_1 \otimes \ldots \otimes p_l \otimes s_1 \otimes \ldots \otimes s_n)) \vdash \Delta} \quad \dfrac{}{p_1, \ldots p_l, r_1, \ldots, r_m \vdash p_1, \ldots p_n, r_1, \ldots, r_m}(Id)}{}(L \multimap)$$

In most of planning problems it is not known how many times the actions are performed. This is the reason why the exponential ! is used for each rule representing the action. The proof must be modified in dependance of how many times the action is performed. There can be three cases:

– action a is not performed — then use $W!$ rule in a following way:

$$\frac{\Gamma \vdash \Delta}{\Gamma, !a \vdash \Delta}\ (W!)$$

– action a is performed just once — then use $D!$ rule in a following way:

$$\frac{\Gamma, a \vdash \Delta}{\Gamma, !a \vdash \Delta}\ (D!)$$

– action a is performed more than once — then use $D!$ and $C!$ rule in a following way:

$$\frac{\dfrac{\Gamma, !a, a \vdash \Delta}{\Gamma, !a, !a \vdash \Delta}\ (D!)}{\Gamma, !a \vdash \Delta}\ (C!)$$

The last thing which has to be explained is why a constant $\top$ must be used. The reason is that a goal state is reached when some state contains all the goal predicates. The state can certainly contain more predicates. The importance of the constant $\top$ can be seen in the following:

$$\frac{g_1, \ldots, g_n \vdash g_1 \otimes \ldots \otimes g_n\ (Id) \qquad s_1, \ldots, s_m \vdash \top\ (R\top)}{g_1, \ldots, g_n, s_1, \ldots, s_m \vdash g_1 \otimes \ldots \otimes g_n \otimes \top}\ (R\otimes)$$

Now it is clear that whole planning problem can be reduced to theorem proving in Linear Logic. Let $s_0 = \{p_{0_1}, p_{0_2}, \ldots, p_{0_m}\}$ be an initial state, $g = \{g_1, g_2, \ldots, g_q\})$ be a goal state and $a_1, a_2, \ldots, a_n$ be actions encoded as above. The whole planning problem can be encoded in a following way:

$$\begin{gathered}
p_{0_1}, p_{0_2}, \ldots, p_{0_m}, \\
!(p_1^1 \otimes p_2^1 \otimes \ldots \otimes p_{l_1}^1 \otimes r_1^1 \otimes r_2^1 \otimes \ldots \otimes r_{m_1}^1) \multimap (p_1^1 \otimes p_2^1 \otimes \ldots \otimes p_{l_1}^1 \otimes s_1^1 \otimes s_2^1 \otimes \ldots \otimes s_{n_1}^1), \\
!(p_1^2 \otimes p_2^2 \otimes \ldots \otimes p_{l_2}^2 \otimes r_1^2 \otimes r_2^2 \otimes \ldots \otimes r_{m_2}^2) \multimap (p_1^2 \otimes p_2^2 \otimes \ldots \otimes p_{l_2}^2 \otimes s_1^2 \otimes s_2^2 \otimes \ldots \otimes s_{n_2}^2), \\
\vdots \\
!(p_1^n \otimes p_2^n \otimes \ldots \otimes p_{l_n}^n \otimes r_1^n \otimes r_2^n \otimes \ldots \otimes r_{m_n}^n) \multimap (p_1^n \otimes p_2^n \otimes \ldots \otimes p_{l_n}^n \otimes s_1^n \otimes s_2^n \otimes \ldots \otimes s_{n_n}^n) \\
\vdash g_1 \otimes g_2 \otimes \ldots \otimes g_q \otimes \top
\end{gathered}$$

The plan exists if and only if the above expression is provable in Linear Logic. Obtaining of a plan from the proof can be done by checking of the $(L \multimap)$ rules from the bottom (the expression) to the top (axioms) of the proof.

3.2 Encoding of negative predicates

Previous subsection showed how planning problems can be encoded in Linear Logic. However, this encoding works with the positive precondition only. In planning problems there are usually used negative preconditions which means that an action can be performed if some predicate does not belong to the current state. However in Linear Logic the negative preconditions cannot be encoded directly. Fortunately, there are some possible approaches for bypassing of this problem.

The first approach can be used in propositional Linear Logic. The basic encoding of planning problems must be extended with linear facts representing negative predicates (each predicate p will obtain a twin $\overline{p}$ representing predicate $p \notin s$). It is clear that either p or $\overline{p}$ is contained in the each part of the

proof. The encoding of state s, where predicates $p_1, \ldots, p_m \in s$ and predicates $p_{m+1}, \ldots, p_n \notin s$:

$$p_1 \otimes \ldots \otimes p_m \otimes \overline{p_{m+1}} \otimes \ldots \otimes \overline{p_n}$$

Each action $a = \{p^+(a), p^-(a), e^-(a), e^+(a)\}$ from a given planning domain can be transformed to action $a' = \{p'(a'), e'^-(a'), e'^+(a')\}$, where $p'(a') = p^+(a) \cup \{\overline{p}\,|p \in p^-(a)\}$, $e'^-(a') = e^-(a) \cup \{\overline{p}\,|p \in e^+(a)\}$ and $e'^+(a') = e^+(a) \cup \{\overline{p}\,|p \in e^-(a)\}$. The encoding of the action a' is following:

$$\forall p_i \in p'(a') \setminus e'^-(a'), 1 \leq i \leq l; \forall \overline{p_{i'}} \in p'(a') \setminus e'^-(a'), 1 \leq i' \leq l'$$
$$\forall r_j \in e'^-(a'), 1 \leq j \leq m; \forall s_k \in e'^+(a'), 1 \leq k \leq n$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes \overline{p_1} \otimes \overline{p_2} \otimes \ldots \otimes \overline{p_{l'}} \otimes r_1 \otimes r_2 \otimes \ldots \otimes r_m \otimes \overline{s_1} \otimes \overline{s_2} \otimes \ldots \otimes \overline{s_n}) \multimap$$
$$(p_1 \otimes p_2 \otimes \ldots \otimes p_l \otimes \overline{p_1} \otimes \overline{p_2} \otimes \ldots \otimes \overline{p_{l'}} \otimes s_1 \otimes s_2 \otimes \ldots \otimes s_n \otimes \overline{r_1} \otimes \overline{r_2} \otimes \ldots \otimes \overline{r_m})$$

Second approach can be used in predicate Linear Logic. Each linear fact representing predicate $p(x_1, \ldots, x_n)$ can be extended by one argument representing if predicate $p(x_1, \ldots, x_n)$ belongs to the state s or not ($p(x_1, \ldots, x_n) \in s$ can be represented as $p(x_1, \ldots, x_n, 1)$ and $p(x_1, \ldots, x_n) \notin s$ can be represented as $p(x_1, \ldots, x_n, 0)$). Encoding of actions can be done in a similar way like in the first approach. The advantage of this approach is in the fact that the representation of predicates can be generalized to such a case that more than one (same) predicate is available. It may be helpful in encoding of some other problems (for example Petri Nets).

3.3 Example

In this example we will use a predicate extension of Linear Logic. Imagine a version of "Block World", where we have slots and boxes, and every slot may contain at most one box. We have also a crane, which may carry at most one box.

Objects: 3 slots (1,2,3), 2 boxes (a, b), crane

Initial state: $in(a, 1) \otimes in(b, 2) \otimes free(3) \otimes empty$

Actions:

$$PICKUP(Box, Slot) = \{ \quad p = \{empty, in(Box, Slot)\},$$
$$e^- = \{empty, in(Box, Slot)\},$$
$$e^+ = \{holding(Box), free(Slot)\}\}$$

$$PUTDOWN(Box, Slot) = \{ \quad p = \{holding(Box), free(Slot)\},$$
$$e^- = \{holding(Box), free(Slot)\},$$
$$e^+ = \{empty, in(Box, Slot)\}\}$$

Goal: Box a in slot 2, Box b in slot 1.

The encoding of the action $PICKUP(Box, Slot)$ and $PUTDOWN(Box, Slot)$:

$PICKUP(Box, Slot) : empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot)$

$PUTDOWN(Box, Slot) : holding(Box) \otimes free(Slot) \multimap empty \otimes in(Box, Slot)$

The whole problem can be encoded in Linear Logic in the following way:

$in(a, 1), in(b, 2), free(3), empty, !(empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot)),$
$!(holding(Box) \otimes free(Slot) \multimap empty \otimes in(Box, Slot)) \vdash in(b, 1) \otimes in(a, 2) \otimes \top$

3.4 Additional remarks

The basic encoding is accurate if the following conditions are satisfied:

- $e^-(a) \subseteq p(a)$, $\forall a \in A$
- $p(a) \subseteq s$ and $s \cap e^+(a) = \emptyset$, $\forall a \in A$ and s is a state

Even though there are not many domains violating these conditions, the violation may cause that the basic encoding is inaccurate. If the first condition is violated, it may happen that some actions normally performable will not be performable via Linear Logic, because all predicates from negative effects are placed on the left hand side of the implication which means that all predicates must be presented before the performance of the action (regarding the definition we can perform every action if all predicates from its precondition are presented in a certain state). If the second condition is violated, it may happen that after the performance of the action violating this condition some predicates will be presented more times than once. To avoid these troubles we have to convert the basic encoding into the encoding supporting negative predicates and we must put $(p \oplus \overline{p})$ into left side of implications for every predicate p which may cause the breaking of the conditions. It is clear that either p or $\overline{p}$ is presented in every state. $(p \oplus \overline{p})$ on the left side of the implication ensures that either p or $\overline{p}$ is removed from the particular state (depending on which one is available) and then there is added p (if p is in positive effects) or $\overline{p}$ (if p is in negative effects).

4 Encoding optimizations of planning problems in Linear Logic

In the previous section, we showed the pure encoding of planning problems in Linear Logic. To improve efficiency of the searching for a plan, it is needed to encode some optimizations described in the next subsections.

4.1 Handling with static predicates

Static predicates are often used in planning problems. Static predicates can be easily detected, because each static predicate in a planning problem is such predicate that belongs to an initial state and does not belong to any effect of any

action (static predicates appear only in preconditions). It is possible to encode the static predicates like 'classical' facts using the exponential !. Assume action $a = \{p(a) = \{p_1, p_2\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\}$ where p_2 is a static predicate. Action a can be encoded in a following way:

$$(p_1 \otimes !p_2) \multimap p_3$$

The encoding of static predicates is described in propositional Linear Logic for better understanding. This encoding has the purpose in predicate Linear Logic (in propositional Linear Logic static predicates can be omitted).

4.2 Blocking of actions

To increase efficiency of solving planning problems it is quite necessary to use some technique which helps a solver to avoid unnecessary backtracking. Typically, the way how a lot of unnecessary backtracking can be avoided is blocking of actions that are not leading to a solution (for example inverse actions).

The idea how to encode the blocking of actions rests in an addition of new predicate $can(a, x)$, where a is an action and $x \in \{0, 1\}$ is representing a status of action a. If $x = 0$ then action a is blocked and if $x = 1$ then action a is unblocked. Now it is clear that the encoding of action a can be done in the following way: (Assume $a = \{p(a) = \{p_1\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\}\}$)

$$\begin{gathered}(can(a,1) \otimes p_1) \multimap (can(a,1) \otimes p_3), \text{or} \\ (can(a,1) \otimes p_1) \multimap (can(a,0) \otimes p_3)\end{gathered}$$

The first expression means that a does not block itself and the second expression means that a blocks itself.

Assume that some action $b = \{p(b) = \{q_1\}, e^-(b) = \{q_1\}, e^+(b) = \{q_2\}$ can block action a. Encoding of action b is following ($can(b, ?)$ means $can(b, 1)$ or $can(b, 0)$ like in the previous encoding of action a):

$$\forall X : (can(b,1) \otimes can(a,X) \otimes q_1) \multimap (can(b,?) \otimes can(a,0) \otimes q_2)$$

The predicate $can(a, X)$ from the expression above represents the fact that we do not known if a is blocked or not. If a is already blocked then X unifies with 0 and anything remains unchanged ($can(a, 0)$ still holds). If a is not blocked then X unifies with 1 which means that a become blocked, because $can(a, 1)$ is no longer true and $can(a, 0)$ become true.

Now assume that action $c = \{p(c) = \{r_1\}, e^-(c) = \{r_1\}, e^+(c) = \{r_2\}$ can unblock action a. Encoding of action c can be done in a similar way like before. The encoding of action c is following ($can(c, ?)$ means $can(c, 1)$ or $can(c, 0)$ like in the previous encoding of action a):

$$\forall X : (can(c,1) \otimes can(a,X) \otimes r_1) \multimap (can(c,?) \otimes can(a,1) \otimes r_2)$$

The explanation how this encoding works is similar like the explanation in the previous paragraph.

4.3 Enforcing of actions

Another optimization which can help a solver to find a solution faster is enforcing of actions. Typically, when some action is performed it is necessary to perform some other action. It is possible to enforce some action by blocking of other actions, but it may decrease the efficiency, because each single action must be blocked in the way described in the previous subsection which means that formulae rapidly increase their length.

The idea how enforcing of actions can be encoded rests also in an addition of new predicate $can(a)$, where a is an only action which can be performed. Let $can(1)$ represent the fact that all actions can be performed. The encoding of action $a = \{p(a) = \{p_1\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\}$ which does not enforce any other action is following:

$$((can(a) \oplus can(1)) \otimes p_1) \multimap (can(1) \otimes p_3)$$

The expression $can(a) \oplus can(1)$ means that action a can be performed if and only if a is enforced by other action ($can(a)$ is true) or all actions are allowed ($can(1)$ is true).

Now assume that action $b = \{p(b) = \{q_1\}, e^-(b) = \{q_1\}, e^+(b) = \{q_2\}$ enforces the action a. The encoding of action b is following:

$$((can(b) \oplus can(1)) \otimes q_1) \multimap (can(a) \otimes q_2)$$

It is clear that in this encoding there is one or all actions allowed in a certain step. This idea can be easily extended by adding some new symbols representing groups of allowed actions.

4.4 Assembling of actions into a single action

Another kind of optimization in planning problems is assembling of actions into a single action, usually called macro-action (for deeper insight see [18]). This approach is based on a fact that some sequences of actions are used several times. Let $a_1, \ldots, a_n$ be a sequence of actions encoded in Linear Logic in the way described before. For further usage we use shortened notation of the encoding:

$$\bigotimes_{\forall l \in \Gamma_i} l \multimap \bigotimes_{\forall r \in \Delta_i} r \quad \forall i \in \{1, \ldots, n\}$$

Assume that action a is created by an assembling of a sequence of actions $a_1, \ldots, a_n$. Because a is also an action, the encoding is following:

$$\bigotimes_{\forall l \in \Gamma} l \multimap \bigotimes_{\forall r \in \Delta} r$$

The following algorithm shows how action a can be obtained from the sequence of actions $a_1, \ldots, a_n$.

Algorithm 1:
INPUT: $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ (see the previous encoding of actions $a_1, \ldots, a_n$)
OUTPUT: Γ, Δ (see the previous encoding of action a)

$\Gamma := \Delta := \emptyset$
for $i = 1$ **to** n **do**
  Λ:=$\Delta \cap \Gamma_i$
  Δ:=$(\Delta \setminus \Lambda) \cup \Delta_i$
  $\Gamma := \Gamma \cup (\Gamma_i \setminus \Lambda)$
endfor

Proposition 4. *Algorithm 1 is correct.*

Proof. The correctness of algorithm 1 can be proved inductively in a following way:

For $n = 1$, it is easy to see that algorithm 1 works correctly when only action a_1 is in the input. The for cycle on lines 2-6 runs just once. On line 3 it is easy to see that $\Lambda = \emptyset$, because $\Delta = \emptyset$. Now it can be seen that $\Delta = \Delta_1$ (line 4) and $\Gamma = \Gamma_1$ (line 5), because $\Lambda = \emptyset$ and Γ and Δ before line 4 (or 5) are also empty.

Assume that algorithm 1 works correctly for k actions. From this assumption imply existence of an action a that is created by algorithm 1 from some sequence of actions $a_1, \ldots, a_k$ after k steps of the for cycle in lines 2-6. Let s be a state and s' be a state which is obtained from s by applying action a (without loss of generality assume that a can be applied on s). It is true that $s' = (s \setminus \Gamma) \cup \Delta)$. Let a_{k+1} be an action which is applied on state s' (without loss of generality assume that a_{k+1} can be applied on s'). It is true that $s'' = (s' \setminus \Gamma_{k+1}) \cup \Delta_{k+1} = ((s \setminus \Gamma) \cup \Delta) \setminus \Gamma_{k+1}) \cup \Delta_{k+1}$. After $k+1$-th step of the for cycle (lines 2-6) action a' is created (from sequence $a_1, \ldots, a_k, a_k + 1$). When a' is applied on state s, $s''' = (s \setminus \Gamma') \cup \Delta')$. From lines 3-5 it can be seen that $\Gamma' = \Gamma \cup (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$ and $\Delta' = (\Delta \setminus (\Delta \cap \Gamma_{k+1})) \cup \Delta_{k+1}$. Now $s''' = (s \setminus (\Gamma \cup (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1})))) \cup ((\Delta \setminus (\Delta \cap \Gamma_{k+1})) \cup \Delta_{k+1})$. To finish the proof we neede to prove that $s'' = s'''$. $p \in s''$ iff $p \in \Delta_{k+1}$ or $p \in \Delta \wedge p \notin \Gamma_{k+1}$ or $p \in s \wedge p \notin \Gamma \wedge (p \notin \Gamma_{k+1} \vee p \in \Delta)$. $p \in s'''$ iff $p \in \Delta_{k+1}$ or $p \in \Delta \wedge p \notin (\Delta \cap \Gamma_{k+1})$ or $p \in s \wedge p \notin \Gamma \wedge p \notin (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$. It is easy to see that $p \in \Delta \wedge p \notin (\Delta \cap \Gamma_{k+1})$ is satisfied iff $p \in \Delta \wedge p \notin \Gamma_{k+1}$ is satisfied. $p \notin (\Gamma_{k+1} \setminus (\Delta \cap \Gamma_{k+1}))$ is satisfied iff $p \notin \Gamma_{k+1}$ or $p \in \Delta$ is satisfied. Now is clear that $s'' = s'''$.

□

Algorithm 1 works with planning problems encoded in propositional Linear Logic. The extension of the algorithm to predicate Linear Logic can be simply done by adding of constraints symbolizing which actions' arguments must be equal. This extension affect only a computation of intersection $(\Delta_i \cap \Gamma_j)$.

The following algorithm for assembling of actions into a single action is based on a paradigm divide and conquer which can support a parallel implementation.

Algorithm 2:
`INPUT:` $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ (see the previous encoding of actions $a_1, \ldots, a_n$)
`OUTPUT:` Γ, Δ (see the previous encoding of action a)

Function Assemble$(\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots \Delta_n)$:$\Gamma, \Delta$
  if $n = 1$ **then return**(Γ_1, Δ_1) **endif**
  Γ', Δ':=**Assemble**$(\Gamma_1, \ldots, \Gamma_{\lceil \frac{n}{2} \rceil}, \Delta_1, \ldots, \Delta_{\lceil \frac{n}{2} \rceil})$
  Γ'', Δ'':=**Assemble**$(\Gamma_{\lceil \frac{n}{2} \rceil + 1}, \ldots, \Gamma_n, \Delta_{\lceil \frac{n}{2} \rceil + 1}, \ldots, \Delta_n)$
  $\Lambda := \Delta' \cap \Gamma''$
  $\Gamma := \Gamma' \cup (\Gamma'' \setminus \Lambda)$
  $\Delta := \Delta'' \cup (\Delta' \setminus \Lambda)$
  return(Γ, Δ)
endFunction

Proposition 5. *Algorithm 2 is correct.*

Proof. The correctness of algorithm 2 can be proved in a following way:

$n = 1$ — It is clear that the assemblage of one-element sequence of actions (a_1) is equal to action a_1 itself.

$n > 1$ — Let $a_1, \ldots a_n$ be a sequence of actions. In lines 2 and 3 the sequence splits into two sub-sequences $(a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and $a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n)$ and algorithm 2 is applied recursively on them. Because $\lceil \frac{n}{2} \rceil < n$ when $n > 1$, it is easy to see that the recursion will finitely terminate (it happens when $n = 1$). Now it is clear that a' and a'' are actions obtained by the assembling of sequences $a_1, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and $a_{\lceil \frac{n}{2} \rceil + 1}, \ldots, a_n$. These actions are assembled into a single action a at lines 4-6. The proof of the correctness of this assemblage is done in the proof of proposition 1.

□

Algorithm 2 can be extended to predicate Linear Logic in a similar way like algorithm 1.

5 Linear Logic in planning with resources

Planning problems with resources becomes commoner, because it has more practical applicability than classical planning. Linear Logic itself as mentioned before is often called 'logic of resources'. Even though propositional Linear Logic does not seem to be good for representing planning problems with resources. We are able to represent the number of units of resources by linear facts, but handling with them is difficult, making the encoding much more complex. The biggest problem is refilling of the resources to some predefined level, because in Linear Logic we usually cannot find out how many units of the resources are remaining. Instead of propositional Linear Logic we can use predicate Linear Logic where we are able to use function symbols like + or −. In addition, we can use comparative operators that can be represented in Linear

Logic notation as binary predicates with exponential !. Here, all classical function symbols and comparative operators are written in classical infix form. Let $a = \{p(a) = \{p_1\}, e^-(a) = \{p_1\}, e^+(a) = \{p_3\}$ be an action which in addition requires at least 3 units of resource r and after performance of a 3 units of r will be consumed. The encoding of a is following:

$$\forall X : (p_1 \otimes r(X) \otimes X \geq 3) \multimap (p_3 \otimes r(X - 3))$$

Let $b = \{p(b) = \{q_1\}, e^-(b) = \{q_1\}, e^+(b) = \{q_2\}$ be an action which in addition refills resource r to 10 units. The encoding of b is following:

$$\forall X : (q_1 \otimes r(X)) \multimap (q_2 \otimes r(10))$$

Predicate r representing some resource must appear on the both sides of the implication, because in every state predicate r must be listed just once. The first rule can be applied if and only if p_1 is true and r contains at least 3 units. It is done in the left side of the implication where predicates p_1 and r are removed. In the right side of the implication we add predicates p_3 and r, where r is decreased by 3 units. The second rule can be applied if and only if q_1 is true (do not depend on r). It is done in the left side of the implication where predicates q_1 and r are removed. In the right side of the implication we add predicates q_2 and r, where r is set to 10 units.

6 Linear Logic in planning under uncertainty

The main difference between deterministic planning and planning under uncertainty is such that actions in planning under uncertainty can reach more states (usually we do not know which one). The main advantage of Linear Logic, which can be used in planning under uncertainty, are additive operators (&) and ($\oplus$). We have two options how to encode uncertain actions. In the following expressions we assume that after the performance of action a on state s we obtain one of $s_1, s_2, \ldots, s_n$ states in the certain steps (remember the encodings of states):

$$s \multimap (s_1 \& s_2 \& \ldots \& s_n)$$

$$s \multimap (s_1 \oplus s_2 \oplus \ldots \oplus s_n)$$

If we use additive conjunction (&) we want to find a plan which may succeed (with nonzero probability). Recall the rule $L\&$ which gives the choice which state will be set as following. If we use additive disjunction ($\oplus$) we want to find a plan which certainly succeeds. Recall the rule $L\oplus$ which tells that we have to proof that from all of the following states we can certainly reach the goal. Even though Linear Logic cannot handle probabilities well, we can use it for decision if there exists some plan which certainly succeed or if there is no chance to find some plan which may succeed.

7 Conclusions and future research

The previous research showed that Linear Logic is a good formalism for encoding many problems with expendable resources like planning problems, because in comparison to the other logics, Linear Logic seems to be strong enough to represent also planning problems with time and resources. This paper extends the previous research in this area by providing detailed description of encoding of planning problems into Linear Logic and by showing that many optimizations of planning problems, which helps a planner to find a solution more quickly, can be also easily encoded in Linear Logic. Main advantage of this approach also rests in the fact that an improvement of the Linear Logic solver leads to improved efficiency of the planner based on Linear Logic.

One of possible directions how to implement an efficient algorithm for solving planning problems encoded in Linear Logic is using the connection between Linear Logic and Petri Nets. It is not difficult to see that the encoding of the planning problems is similar to an encoding of Petri Nets. The implementation of an unfolding algorithm for reachability problem in Petri Nets for solving planning problems has been done by [13] and showed very good results. We will study the possibility of extending this algorithm in the way that the extended algorithm will support the encoding of planning problems in predicate Linear Logic. There is a possibility of extension of the presented encodings of planning problems in Linear Logic into Temporal Linear Logic (to learn more about Temporal Linear Logic, see [25]). It seems to be a very useful combination, because Temporal Logic can give us more possibilities for encoding of relationships between actions. Another possible way of our future research is using of the encodings of the optimizations for transformation of planning domains. Transformed planning domains can be used with existing planners and can reach better results, because the presented optimizations of planning problems can help the planners to prune the search space. We focused on getting knowledge from plan analysis which helps us to find out the optimizations. We studied it in [6] and results that were provided there are very interesting. At last, Linear Logic seems to be strong enough to encode planning problems with time. Basic idea of this extension comes from the fact that time can be also encoded into linear facts. The usage of predicate Linear Logic seems to be necessary in this case as it as in planning with resources.

8 Acknowledgements

We thank the reviewers for the comments. The research is supported by the Czech Science Foundation under the contracts no. 201/08/0509 and 201/05/H014 and by the Grant Agency of Charles University (GAUK) under the contract no. 326/2006/A-INF/MFF.

References

1. Bacchus F., Kabanza F.: Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 22:5-27. 1998.
2. Banbara M. *Design and Implementation of Linear Logic Programming Languages*. Ph.D. Dissertation, The Graduate School of Science and Technology, Kobe University. 2002.
3. Bibel W., Cerro L. F., Fronhofer B., Herzig A. Plan Generation by Linear Proofs: On Semantics. *In proceedings of GWAI*. 49–62. 1989.
4. Chrpa L. Linear Logic: Foundations, Applications and Implementations. *In proceedings of workshop CICLOPS*. 110-124. 2006.
5. Chrpa L. Linear logic in planning. *In proceedings of Doctoral Consorcium ICAPS*. 26-29. 2006.
6. Chrpa L., Bartak R. Towards getting domain knowledge: Plans analysis through investigation of actions dependencies *In proceedings of FLAIRS*. 531–536.
7. Cresswell S., Smaill A., Richardson J. Deductive Synthesis of Recursive Plans in Linear Logic. *In proceedings of ECP*. 252–264. 1999.
8. Doherty P., Kvanstrom J.: TALplanner: A temporal logic based planner. *AI Magazine* 22(3):95-102. 2001.
9. Ghallab M, Nau D., Traverso P. *Automated planning, theory and practice*. Morgan Kaufmann Publishers 2004. 2004.
10. Girard J.-Y. Linear logic. *Theoretical computer science* 50:1–102. 1987.
11. Girard J.-Y. *Linear Logic: Its Syntax and Semantics*. Technical report, Cambridge University Press. 1995.
12. Hodas J. *Linear Logic in Computer Science*. Cambridge University Press. 2004.
13. Hickmott S., Rintanen J., Thiebaux S., White L. Planning via Petri Net Unfolding *In proceedings of IJCAI*. 1904-1911. 2007
14. Hodas J. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. Ph.D. Dissertation, University of Pennsylvania, Department of Computer and Information Science. 1994.
15. Jacopin E. Classical AI Planning as Theorem Proving: The Case of a Fragment of Linear Logic *In proccedings of AAAI*. 62–66. 1993.
16. Kautz H. A., Selman B.: Planning as Satisfiability. *In proccedings of ECAI*. 359-363. 1992.
17. Kautz H. A., Selman B., Hoffmann J.: SatPlan: Planning as Satisfiability. *In proccedings of 5th IPC*. 2006.
18. Korf, R. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–77. 1985.
19. Küngas P. Analysing AI Planning Problems in Linear Logic - A Partial Deduction Approach. *IN proceedings of SBIA*. 52–61. 2004.
20. Küngas P. Linear logic for domain-independent ai planning. *Proceedings of Doctoral Consorcium ICAPS*. 2003.
21. Lincoln P. Linear logic. *Proceedings of SIGACT*. 1992.
22. Masseron M. Tollu C., Vauzeilles J. Generating plans in linear logic i-ii. *Theoretical Computer Science*. vol. 113, 349-375. 1993.
23. Oliet N. M., Meseguer, J. From petri nets to linear logic. *Springer LNCS 389*. 1989.
24. Tamura N. *User's guide of a linear logic theorem prover (llprover)* Technical report, Kobe University, Japan. 1998.

25. Tanabe M. Timed petri nets and temporal linear logic. *In proceedings of Application and Theory of Petri Nets.* 156-174. 1997
26. Thielsher M.: FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming* 5(4-5):533–565. 2005.