Linear Logic in Planning

Lukas Chrpa
Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague
chrpa@kti.mff.cuni.cz

Abstract

Linear Logic is a powerful formalism used to manage a lot of problems with resources. Linear Logic can also be used to formalize Petri Nets and to solve simple planning problems (for example 'Block World'). Research goes ahead also in Linear Logic Programming, which means that we have tools, that can solve Linear Logic problems. In this paper I will show the possible connection between solving planning problems and Linear Logic Programming.

Introduction

Planning problems can be solved by translation into another formalism like SAT, CSP, BDD, etc. Linear Logic is another formalism to which a planning problem can be translated. There already exists a planning system based on Linear Logic called RAPS (Küngas 2003). RAPS was introduced at Doctoral Consortium at ICAPS 2003. The author of RAPS compared RAPS with the best planners that participated at IPC 2002. This comparison showed very interesting results, a Skeleton version of RAPS showed almost the best computation time in computing the plans (the typed Depots domain), but on the other hand the solution length (in the typed Depots domain) was almost the highest, which means that the plans weren't optimal. The Skeleton version of RAPS first converts a planning problem into propositional Linear Logic (which means that predicates in a planning operator description are abstracted to propositional constants by removing predicates' arguments) and from that it calculates skeleton plans, which means that we obtain a sequence of actions needed to reach the goal. The final plan is obtained from the skeleton plan by unification with corresponding arguments. RAPS (including the Skeleton version) exploits the fact that a planning problem coded into Linear Logic is easily converted to the problem of Petri Net reachability so the system mainly exploits the algorithms for Petri Net reachability.

Instead of coding the planning problems in Linear Logic and solving the Petri Net Reachability problem like in RAPS, I propose to study possibilities of solving the planning problems using Linear Logic Programming tools (described bellow). I believe that Linear Logic Programming tools can achieve better efficiency than 'classical' Logic Programming tools and hence might be more appropriate for solving planning problems (Banbara 2006). Nevertheless I will also study the possibilities of efficient solving Linear Logic problems in Prolog. Prolog itself has many extensions which may support some techniques for optimization of planning problems.

In the next part of this paper I will give a short introduction to Linear Logic and Linear Logic Programming. Then I will describe how to solve Petri Net reachability problem using Linear Logic and how to convert planning problems to Linear Logic. Finally I will present my future research plans in this area.

Linear Logic

Linear Logic was introduced by J.Y. Girard at 1987 (Girard 1987; 1995). Unlike the 'classical' logic we can handle resources in Linear Logic. The basic operator in Linear Logic is a (linear) implication ($A \multimap B$), which is defined as B is obtained by using one resource A. Linear Logic defines more operators (not only implication), but I will describe here only the multiplicative conjunction $\otimes$ and the additive disjunction $\oplus$ (the description of other operators can be found in (Girard 1987; 1995)). The expression $(A \otimes B) \multimap (C \otimes D)$ means that C and D are obtained using A and B. The expression $A \multimap (B \oplus C)$ means that B or C (we don't know which one) is obtained using A. Proving in Linear Logic is quite similar to proving in the 'classical' logic (hypotheses $\Rightarrow$ conclusion) , but the calculus of Linear Logic is more complicated. To find out more about proving in Linear Logic and the whole calculus of Linear Logic, see (Girard 1987; 1995).

Linear Logic Programming

Linear Logic Programming is derived from classical logic programming (Prolog based) by including linear facts and linear operators. Syntax of common Linear Logic Languages is quite similar to Prolog syntax (Banbara 2006). As I mentioned above, the efficiency of these languages in solving problems describable in Linear Logic is better than in Prolog. The good efficiency of Linear Logic Programming languages is reached by using optimization techniques based on the theory of proving in Linear Logic, more in (Banbara 2002).

In my diploma thesis I proposed a Linear Logic Programming language SLLL (Chrpa 2005), which was constructed as a compiler to Prolog. However, the problem of this language is a low computational efficiency caused by emulating the linear facts as lists. Fortunately, there are other Linear Logic Programming languages: Lolli (Hodas 1994; 1992), LLP (Banbara 2002), Lygon (Winikoff 1996), LTL[1] and more. Lolli is possibly the strongest Linear Logic Programming language, which contains almost all of Linear Logic features. LTL and LLP are possibly the most effective Linear Logic Programming languages today.

Solving Petri Net reachability problem by Linear Logic

In the next paragraphs, I will describe how Linear Logic can be used in solving the Petri Net reachability problem, that is, the problem of finding whether a given marking is reachable from the initial marking. To find more about Petri Nets (and the problem of Petri Net reachability), see (Reisig 1985).

Now I explain how the problem of Petri Net reachability can be easily encoded using Linear Logic. Tokens in places are encoded as linear facts (resources), in particular the initial marking (in this case: one token in each place $p_1, \ldots, p_k$) is encoded in the following way:

$$\vdash p_1 \otimes p_2 \otimes \ldots \otimes p_k$$

Transitions are also encoded as axioms. For transition t and places $p_{i_1}, \ldots, p_{i_m} \in$ IN(t)...input places and $p_{o_1}, \ldots, p_{o_n} \in$ OUT(t)...output places we get:

$$\vdash (p_{i_1} \otimes \ldots \otimes p_{i_m}) \multimap (p_{o_1} \otimes \ldots \otimes p_{o_n})$$

If the goal (in this case: one token in each place $p_{g_1}, \ldots, p_{g_l}$) ($p_{g_1} \otimes \ldots \otimes p_{g_l}$) is provable (by using the above axioms), the marking (one token in each place $p_{g_1}, \ldots, p_{g_l}$) is reachable.[2] More about the topic can be found in (Oliet & Meseguer 1989).

Planning with Linear Logic

Problem of using Linear Logic in planning have been studied by several authors (Masseron, Tollu, & Vauzeilles 1993; Kanovich & Vauzeilles 2001). Encoding of planning problems in Linear Logic is quite similar to encoding of Petri Nets (planning problems can also be encoded directly using Petri Nets). In planning we have states, that are represented by the set of predicates, that are true in the given state. We can encode these states as a multiplicative conjunction of (true) predicates, that belong to the corresponding state. The encoding of state s:

$$(p_1 \otimes p_2 \otimes \ldots \otimes p_n), \qquad s = \{p_1, p_2, \ldots, p_n\}$$

Actions in planning contain preconditions p (must be satisfied before preforming the action), negative effects e^-(removed after the action), and positive effects e^+ (added after the action). The action $a = \{p, e^-, e^+\}$ is encoded as:

$$\forall i \in \{1, 2, \ldots, k\}, l_i \in p \cup e^-$$
$$\forall j \in \{1, 2, \ldots, m\}, r_j \in e^+ \cup (p - e^-)$$
$$(l_1 \otimes l_2 \otimes \ldots \otimes l_k) \multimap (r_1 \otimes r_2 \otimes \ldots \otimes r_m)$$

This expression means that the predicates on the left side of the implication will no longer be true after performing action a and the predicates on the right side of the implication will become true after performing action a. The plan exists if and only if the encoding of the goal state is provable from the encoding of the initial state using the actions encoded as axioms.[3]

Encoding negative predicates

The above formalism worked only with positive predicates. However sometime we also need to encode negative predicates.[4] We extend the encoding of predicates with symbols for negative predicates (p will obtain a twin $\overline{p}$ which represents a negative form of predicate p). The encoding of state s, where predicates $p_1, \ldots, p_m$ are true in s and $p_{m+1}, \ldots, p_n$ are false in s:

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

For every action $a = \{p, e^-, e^+\}$, we create an action $a' = \{p, e'^-, e'^+\}$, where $e'^- = e^- \cup \{\overline{p}\,|p \in e^+\}$ and $e'^+ = e^+ \cup \{\overline{p}\,|p \in e^-\}$. Now we can encode all actions a' in the same way as described above.

Example

Let us present now an example of the conversion (without negative predicates). Imagine the 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.

Initial state: 3 slots (1,2,3), 2 boxes (a, b), empty crane, box a in slot 1, box b in slot 2, slot 3 is free.

Actions:

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

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

[1]developed by Dr. Arnost Vecerka, my diploma supervisor

[2]multiple tokens in places or multiple (input, output) places can be easily encoded as n-times $p \otimes \ldots \otimes p$

[3]To obtain a full plan, we must keep information about used axioms (encoded actions) during proving.

[4]Negative predicates often appear in preconditions.

Goal: Box a in slot 2, Box b in slot 1, empty crane, free slot 3.

The encoding of the problem:

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

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

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

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

A solution is a sequence of actions. In this case, this sequence looks like: $PICKUP(a,1)$, $PUTDOWN(a,3)$, $PICKUP(b,2)$, $PUTDOWN(b,1)$, $PICKUP(a,3)$, $PUTDOWN(a,1)$.

Possible optimizations

In the previous subsections I described the pure encoding of planning problems to Linear Logic. In this subsection I will show that we are able to encode some optimizations to Linear Logic as well.

In the above example we have two actions: $PICKUP(Box,Slot)$ and $PUTDOWN(Box,Slot)$. These actions are inverse, which means that if we perform these actions with same parameters $Box,Slot$ consecutively, we obtain the state that we had before performing these actions. The main idea how to block the consecutive performing of inverse actions is an extension of the encoding of the actions. The encoding of the action $PICKUP(Box,Slot)$ from the above example id shown bellow (encoding of the action $PUTDOWN(Box,Slot)$ is analogical):

$PICKUP(Box,Slot)$:
$canpick(Box,Slot) \otimes canput(Box,Slot) \otimes nopick(X,Y) \otimes empty \otimes in(Box,Slot) \multimap holding(Box) \otimes free(Slot) \otimes canpick(Box,Slot) \otimes noput(Box,Slot) \otimes canpick(X,Y)$

The predicates $canpick(Box,Slot)$ ($canput(Box,Slot)$) mean that actions $PICKUP(Box,Slot)$ ($PUTDOWN(Box,Slot)$) can be performed (allowed). The predicates $nopick(Box,Slot)$ ($noput(Box,Slot)$) mean that actions $PICKUP(Box,Slot)$ ($PUTDOWN(Box,Slot)$) can't be performed (blocked). The encoding of the action $PICKUP(Box,Slot)$ means that this action can be performed if and only if the predicate $canpick(Box,Slot)$ is true. After performing this action the predicate $canput(Box,Slot)$ becomes false, the predicate $noput(Box,Slot)$ becomes true, the predicate $nopick(X,Y)$ (represents exactly one blocked action $PICKUP(X,Y)$) becomes false and the predicate $canpick(X,Y)$ becomes true. In the other words this means that performing some (allowed) action blocks the inverse action and unblocks the action blocked by the previously performed inverse action.

Another optimization of the previous example is blocking the action $PICKUP(Box,Slot)$ forever if the predicate $in(Box,Slot)$ is true in goal state. The action $PICKUP(Box,Slot)$ is blocked when both predicates $canpick(Box,Slot)$ and $nopick(Box,Slot)$ are false. This is obtained by removing the predicate $nopick(Box,Slot)$ from the right side of the linear implication in the encoded $PUTDOWN(Box,Slot)$ action.

I showed that Linear Logic can easily encode some optimizations for the planning problems. Using these optimizations may lead the to better efficiency.

Comparing to SAT

Linear Logic itself has some advantages that can be exploited in the encoding of planning problems. The main advantage is the linear size of the encoding of the planning problems. For example the size of a SAT encoding of planning problems can be exponential. On the other way, SAT problems are in general NP-complete unlike the undecidability of whole Linear Logic. In planning we are using only a part of Linear Logic, but we still have no evidence about decidability and complexity of this restricted problem.

Future Research

In my future research, I will study the problem of efficient usage of Linear Logic in planning problems (for example encoding optimizations). I will study the possibilities of using Linear Logic Programming tools and possibilities of emulating Linear Logic in Prolog. I will also make a comparison to some models (Gelfond & Lifschitz 1993). The following paragraphs will present my future research plans in more detail:

Using Linear Logic Programming Tools

As I have mentioned above, we have several tools that could solve Linear Logic problems efficiently. The preliminary experiments showed that the existing Linear Logic Programming tools are not powerful enough to solve the planning problems, because these tools can't still handle the linear implication well. This means that I am still emulating Linear Logic in Prolog, which isn't much efficient. Nevertheless, I believe that these tools may be useful as a support to other planning techniques. I also believe that possible improvements of these tools may help with solving the planning problems. I will study the possibilities of using these tools to solve the planning problems.

Emulating Linear Logic in Prolog

Linear Logic can be easily emulated in Prolog.[5] Linear facts are in a special list. We must define two predicates, one for deleting the facts from the list (lin_del) and one for adding the facts to the list (lin_add):

```
lin_del(V,[V|L],L).
lin_del(V,[H|L],[H|NL]):-lin_del(V,L,NL).
lin_add(V,L,[V|L]).
```

[5]We don't need whole Linear Logic, we need to emulate only the support for the operators $\otimes, \oplus, \multimap$.

Emulation of the multiplicative conjunction $\otimes$ and the additive disjunction $\oplus$ is very easy, because we can replace them by ‘classical‘ conjunction and disjunction which are presented in Prolog. Emulation of the linear implication $\multimap$ is also easy. All linear facts on the left side of the linear implication are deleted from the list and all linear facts on the right side of the linear implication are added to the list. A formula $a \otimes b \multimap c \oplus d$ can be written in Prolog like this:

```
(lin_del(a,L1,L2),lin_del(b,L2,L3)),
(lin_add(c,L3,L4);lin_add(d,L3,L4))
```

Variables $L1, L2, L3, L4$ represent the list of linear facts, because we must keep this list consistent and vulnerable to backtracking.

This emulation isn’t very efficient, but we can do some optimizations. If we have the list of linear facts sorted, we don’t need to use the predicates lin_del n-times consequently. We can improve the lin_del predicate such that it will accept a sorted list of n facts and make the same effect like using the old predicates lin_del n-times consequently. This approach will result in a fact that the list of linear facts can be explored only once. To keep the list sorted we must also improve the lin_add fact. In future I will try to find out more and better optimizations in emulating Linear Logic in Prolog.

Temporal Logic extensions

There are also Linear Logic Programming tools that support Temporal Logic extensions. For example the extension of LLP is called TLLP (Banbara 2002). This could provide a formalism to time extensions, especially for qualitative modeling of time. With Temporal Logic we can also model features like an action that must be performed before another action. This may lead to PSP (Plan-Space Planning).

Using Linear Logic in probabilistic planning

When performing an action in probabilistic planning we could reach more states (instead of one like in deterministic planning). Reachability of a particular state depends on probability of obtaining that state after performing the planned action. The main advantage of Linear Logic is additive disjunction, so we are able to encode the actions in probabilistic planning in the following way ($s, s_1, s_2, \ldots, s_n$ are states, A is the action):

$$s \times A \rightarrow \{s_1, s_2, \ldots, s_n\}$$

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

This expression means that only one state from $s_1, s_2, \ldots, s_n$, could be reached after performing action A from state s in a certain step, but we don’t know which one (depends on probability). Unfortunately the main disadvantage of Linear Logic is that it can’t handle probabilities directly. Nevertheless, there is still an option, which consists of possible cooperation with other techniques. This problem needs to be more studied, so in future I will also try to find out more about this extension.

Conclusion

The paper showed that Linear Logic can be used to encode planning problems. Like for other encodings, the advantage of this approach is that an improvement of the Linear Logic solver leads to improved efficiency of the planner based on Linear Logic. Still, the efficiency of current Linear Logic solvers applied to planning problems should be explored in more detail.

Acknowledgements

I thank to my supervisor Roman Bartak for help with writing this paper. The research is supported by the Czech Science Foundation under the contract no. 201/04/1102 and by the Grant Agency of Charles University (GAUK) under the contract no. 326/2006/A-INF/MFF.

References

Banbara, M. 2002. *Design and Implementation of Linear Logic Programming Languages*. Ph.D. Dissertation, The Graduate School of Science and Technology, Kobe University.

Banbara, M. 2006. http://bach.istc.kobe-u.ac.jp/llp/.

Chrpa, L. 2005. Linearni logika. Master’s thesis, Department of Computer Science, Palacky University, Olomouc. (in Czech).

Gelfond, M., and Lifschitz, V. 1993. Representing actions and change by logic programs. *Journal of Logic Programming* 17(2,3,4):301–323.

Girard, J.-Y. 1987. Linear logic. *Theoretical computer science* 50:1–102.

Girard, J.-Y. 1995. *Linear Logic: Its Syntax and Semantics*. Cambridge University Press.

Hodas, J. 1992. Lolli: An extension of lambdaprolog with linear logic context management. *Proceedings of the 1992 Workshop on the lambdaProlog Programming Language*.

Hodas, J. 1994. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation*. Ph.D. Dissertation, University of Pennsylvania, Department of Computer and Information Science.

Kanovich, M., and Vauzeilles, J. 2001. The classical ai planning problems in the mirror of horn linear logic: Semantics, expressibility, complexity. *Mathematical Structures in Computer Science 11(6)*.

Küngas, P. 2003. Linear logic for domain-independent ai planning. *Proceedings of Doctoral Consorcium ICAPS*.

Masseron, M.; Tollu, C.; and Vauzeilles, J. 1993. Generating plans in linear logic i-ii. *Theoretical Computer Science*.

Oliet, N. M., and Meseguer, J. 1989. From petri nets to linear logic. *Springer LNCS 389*.

Reisig, W. 1985. *Petri Nets, An Introduction*. Springer Verlag, Berlin.

Winikoff, M. 1996. Hitch hiker’s guide to lygon 0.7. Technical Report 96/36, The University of Melbourne, Australia.