Linear Logic for Domain-Independent AI Planning (Extended Abstract)

Peep Küngas
Norwegian University of Science and Technology
Department of Computer and Information Science
peep@idi.ntnu.no

Abstract

Although during the last decade in several papers applicability of Linear Logic (LL) theorem proving to AI planning has been emphasised, there is still no experimental data nor prototype available, which would allow to justify performance of LL planning. In this paper we present results demonstrating that performance of a Linear Logic planner may be comparable to state-of-the-art domain-independent planners. While planning with LL, we first abstract a planning problem propositionally. Then the propositional problem is translated to a Petri net and skeleton plans are computed by applying Petri net reachability checking. After a skeleton plan, consisting of not instantiated operators, has been generated, we instantiate the plan. Suitable constants for operators are determined by heuristic-driven constraint solving.

Introduction

Linear Logic (LL) (Girard 1987) is a refinement of classical logic introduced by J.-Y. Girard to provide a means for keeping track of "resources"—two assumptions of propositional constant A are distinguished from a single assumption of A. Since its introduction LL has enjoyed increasing attention both from proof theorists and computer scientists.

Several authors (Masseron, Tollu, & Vauzeilles 1993; Jacopin 1993; Cresswell, Smaill, & Richardson 1999; Grosse, Hölldobler, & Schneeberger 1996; Kanovich & Vauzeilles 2001) have emphasised the contribution LL could have for AI planning. LL enriches problem domains with resource consciousness and allows to represent nondeterminism emerging from interaction with an environment. There exist (Kanovich & Ito 1997) even work extending LL with certain features of temporal logic. Therefore LL provides a comprehensive formalism for expressing most of the rigorous aspects of planning problems—resources, time, nondeterminism, etc.

Usage of LL, similarly to other logics, provides sound and complete AI planning mechanism. Soundness means that plans found by applying LL inference rules are correct wrt to initial problem specifications. Completeness guarantees that if there is a solution, it will be found. Moreover, by fixing the fragment of LL for planning, we determine computational complexity of planning as well, since complexities of different LL fragments have been already studied.

However, there are at the moment by author's knowledge no experimental results nor implementations of (first-order) LL planners available, which would allow us to evaluate the performance of LL planning. This paper tries to fill this gap partially by presenting promising results for a fragment of LL, which provides us with means of keeping track of resources and allows represening STRIPS-like planning operators. Although we consider here only a fragment of LL, we hope to extend the framework to include different disjunctions and other constructions as well provided by LL. Then it becomes possible to express and reason also about non-deterministic effects of actions as well.

In this article we describe design of Resource-Aware Planning System (RAPS[1]), which performs LL planning. RAPS generates plans in two stages—first a LL planning problem is transformed into a propositional problem, this in turn is compiled into a Petri net, and then propositional planning is performed with the PNPlanner (Küngas 2002) algorithm. As a second step achieved propositional plans[2] are instantiated by applying a constraint solving technique described later.

The original problem is transformed into the propositional one by applying propositional abstraction (Plaisted 1981), which means that predicates in a planning operator description are abstracted to propositional constants by removing predicates' arguments. For instance, predicate *at(x,y)* is transformed to propositional constant *at* by propositional abstraction. This approach contrasts with the one adopted by SAT-based planning community, where a propositional problem representation is acquired by instantiating operators from a original problem with all possible combinations of constants. While our approach leaves the number of operators in the problem unchanged, SAT approach generates

[1] There is another planner called RAPS (Reactive Action Planning System), which is available at http://people.cs.uchicago.edu/~firby/raps/. Unfortunately the author was not aware of it when gave the same name for the planner described here.

[2] Plans based on a propositionally abstracted planning problem are called also *skeleton plans* in this article.

exponentially more operators compared to the original problem.

Representing STRIPS operators with LL formulae

While considering AI planning within LL, one of intriguing issues is how to represent planning domains and problems. This section reflects a resource-oriented representation of STRIPS-like operators as adopted by (Bibel 1998) for Transition Logic and (Masseron, Tollu, & Vauzeilles 1993; Jacopin 1993; Grosse, Hölldobler, & Schneeberger 1996; Kanovich & Vauzeilles 2001) for LL framework, where planning operators are encoded as extralogical axioms and planning problem as a sequent which correctness has to be proved.

We consider here (first-order) multiplicative intuitionistic fragment of LL (MILL) consisting of multiplicative conjunction $\otimes$ and linear implication $\multimap$. In terms of resource acquisition the logical expression $A \otimes B \multimap C \otimes D$ means that resources C and D are obtainable only if both A and B are available. Thus $\otimes$ is a sort of conjunction in LL and connective $\multimap$ defines deterministic relations between resources consumed and produced. After literals A and B are consumed, literals C and D are generated. Reasoning is reduced to the notion of resource—the *number* of occurrences of literals determines whether an operator can be applied or not.

It is straightforward to encode STRIPS-like operators directly as LL formulae with connectives $\otimes$ and $\multimap$. For an example, let us consider the STRIPS operator in Figure 1. An appropriate LL extralogical axiom representing semantics of that operator is $\vdash \forall x.(BALL(x) \otimes NEAR(x) \multimap BALL(x) \otimes IN(x) \otimes MOVE\_OK)$.

Thus every element in the *pre*-list of a STRIPS operator is inserted to the left hand side of $\multimap$. Additionally, all elements of the *delete*-list, which do not already exist there, are inserted. To the right hand side of $\multimap$, the *add*-list elements are inserted plus all elements from the *pre*-list, which have to be preserved. This is due to the resource-consciousness property of LL, meaning literally that everything in the left hand side of $\multimap$ are consumed and resources in the right hand side of $\multimap$ are generated, if an axiom is applied.

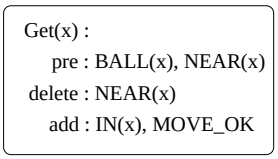

Figure 1: A STRIPS operator.

Some researcher, more familiar with classical logic, tend to adapt the idea that if a literal is deleted from a given world state, it symbolises that the value of the literal became false. Indeed, it is not so in LL—absense of a literal symbolises that no information is known about the literal and therefore this literal cannot be used during reasoning—LL is about computation, not about truth.

To give a reader better overview how we describe planning domains and problems with LL, let us consider the following example[3]:

$$\Gamma = \begin{array}{l} \vdash \forall x.(BALL(x) \otimes NEAR(x) \multimap_{Get} IN(x) \otimes MOVE\_OK) \\ \vdash \forall x.(MOVE\_OK \multimap_{MoveTo} NEAR(x)) \\ \vdash \forall x \forall y.(IN(x) \otimes IN(y) \multimap_{Fill} FULL) \end{array}$$

There the set of extralogical axioms Γ represents a problem domain, where a robot has to collect balls (*BALL*) and fill a box. There are 3 operators—*Get*, *MoveTo* and *Fill*. The planning problem is presented then with the following sequent:

$$\Gamma \vdash BALL(u) \otimes BALL(v) \otimes MOVE\_OK \multimap FULL \otimes MOVE\_OK$$

Subformulae $BALL(u) \otimes BALL(v) \otimes MOVE\_OK$ and $FULL \otimes MOVE\_OK$ represent above respectively the initial and the final state of the world.

LL planning in two stages

In (Küngas 2002) PNPlanner algorithm has been described, which first translates propositional LL planning problems to Petri nets and then applies on-the-fly Petri net reachability checking for planning. In order to take advantage of that algorithm we apply propositional abstraction (Plaisted 1981) to original planning problems and domains in first-order MILL, thereby converting predicates in planning problems to propositional constants. To give an example about propositional abstraction let us consider the planning domain and problem from the previous section—after propositional abstraction they would look as follows:

$$\Gamma' = \begin{array}{l} \vdash BALL \otimes NEAR \multimap_{Get} IN \otimes MOVE\_OK \\ \vdash MOVE\_OK \multimap_{MoveTo} NEAR \\ \vdash IN \otimes IN \multimap_{Fill} FULL \end{array}$$

$$\Gamma' \vdash BALL \otimes BALL \otimes MOVE\_OK \multimap FULL \otimes MOVE\_OK$$

Then we use PNSolver algorithm on propositional LL problems and start computing skeleton solutions to the initial problem. These skeleton solutions are enriched with suitable individuals, if possible. The cycle lasts until a final solution is found. Our LL planning process is summarised in Figure 2. Such approach may be viewed as late binding of variables, because variables are transferred to solutions in the late phase of the problem solving process.

Using skeleton solutions prunes the search space for the second planning stage, where the skeleton plans are instantiated. Skeleton solutions determine the structure of the final plan. Therefore less time is spent for unifying variables during planning.

In following we describe first the PNPlanner algorithm and then we show how propositional plans found can be enriched with individuals to form final plans. Our methodology is complete in the sense that, if there exists a final plan

[3]We write $\multimap_L$ to indicate that an implication represents planning operator L.

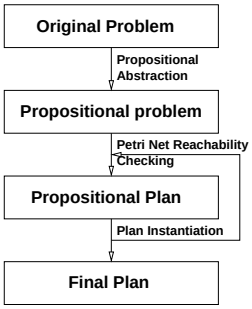

Figure 2: Our LL planning approach.

for a particular LL planning problem, it is found. It must be noted that although absence of skeleton solutions means absence of final solutions, converse is not true.

The PNPlanner algorithm

As told before, PNPlanner represents propositional LL planning problems as Petri nets and applies Petri net reachability checking for planning. Although computational complexity of Petri net reachability checking has been proved to be EXPSPACE-hard (Lipton 1976), PNPlanner overcomes this complexity by constraining the search state space while collapsing several states to a single one and not proceeding search from the same state twice. Moreover, PNPlanner recognises subplans, which, when repeatedly applied, can lead to the goal state. Informally explaining, having two planning operators $\mathcal{O} : \vdash A \multimap B$ and $\mathcal{P} : \vdash B \multimap A \otimes C$, a subplan is constructed merging them to $< \mathcal{O}; \mathcal{P} >: \vdash A \multimap A \otimes C$. Then, if our planning problem is $\mathcal{O}, \mathcal{P} \vdash A \multimap A \otimes C^5$, we can numerically compute that $< \mathcal{O}; \mathcal{P} >$ has be applied 5 times. Thereby routine tasks, where subplans are applied repeatedly, are solved quite efficiently by PNPlanner. This is one of the main factors, which allowed us to achieve results presented in this paper later.

Instantiating skeleton plans

After a skeleton plan has been computed, unification has to be applied to particular propositional planning operators occurring in the skeleton solution. Now propositional operators are replaced with original operators in the skeleton solution and, while trying to unify variables there, it is determined whether there exits a final plan.

We start with plan instantiation from the first operator in the skeleton plan using the initial state of the world as a basis and proceed until all operators have been instantiated. If as a result the final state is reached, the plan has been found. Otherwise we analyse the achieved state and jump back to an operator, where a first possible fault was generated. That kind of backjumping may be viewed as biasing search with heuristics, while heuristics is extracted by comparing a reached and an expected state.

At every operator instantiation step a set of possible operator instantiation solutions is computed by constructing first for every predicate in the operator's *delete*-list a list of matching variable combinations from the current state of the world. Then one of possible solutions is selected, particular operator is instantiated, elements in its *delete*-list are deleted from and elements in its *add*-list are inserted to the current world state. After that search proceeds with the next operator.

All predicates inserted to the world state by instantiated operators are indexed with the operator position number in a plan. If it should happen that an operator cannot be instantiated or the final state reached is not a superset of a goal, then it indicates that a wrong substitution for instantiation was performed somewhere before. Therefore we jump back to the operator, which generated a conflicting predicate. Since several predicates may be conflicting, we jump back as far as possible. This kind of backjumping is another factor why our planner performs well.

We say that a predicate is conflicting with a goal state, if there should exist a predicate with the same name but with different arguments in the goal state. For instance, if the goal state is $\{IN(a), IN(b)\}$, but we have achieved $\{IN(c), IN(b)\}$ then $IN(c)$ is conflicting. We say that a predicate is conflicting with an operator, if because of that predicate no solutions are found for the operator.

Experimental results

To evaluate the performance of our planner, we decided to compare achieved results with the ones of International Planning Competition 2002 (IPC 2002). We solved 22 planning problems for untyped STRIPS version of Depots domain.

The ideas presented here were implemented in a planner called RAPS. While performing our experiments we employed Java HotSpot Virtual Machine 1.4.1, which is distributed with J2SE 1.4.1. Results of IPC 2002 were achieved on a PC, using an Athlon 1800MHz with 1GB of RAM and running Mandrake 8.2. Our results were obtained by running RAPS on a PC with Pentium P4 2533MHz, 1GB of RAM and RedHat 8.0. Although the performance difference between these two configurations is in average 1.4 times, our implementation of RAPS was fully written in Java, while programs at IPC 2002 were mostly implemented in C or C++. Therefore we assume, that slower performance of Java programs compensates usage of a faster machine.

We modified[4] original problems and the Depots domain a bit, by including domain-dependent information. This has been done because our planner is based on propositional abstraction. This means that operator definitions like $\vdash \forall x \forall y \forall z.(at(x, y) \multimap_{Move} at(x, z))$ would first be abstracted to $\vdash (at \multimap_{Move} at)$. Unfortunately this operator is not considered by PNPlanner (Küngas 2002)—since the state of the world does not change as a result of applying this operator, the operator would not be included in the skeleton plan and would not be included in the final plan either. Therefore no solution would be found for the original problem representation either.

Inserted domain-dependent information basically includes additional literals, which change planning opera-

[4]These files are available through the RAPS homepage `http://www.idi.ntnu.no/~peep/RAPS`.

tors' descriptions so that subplan detection could be applied. These literals also give some hints for PNPlanner about the semantics of original problems and operators, which otherwise due to propositional abstraction was lost. Although we are currently looking for ways to generate that information automatically, so far it has been done manually.

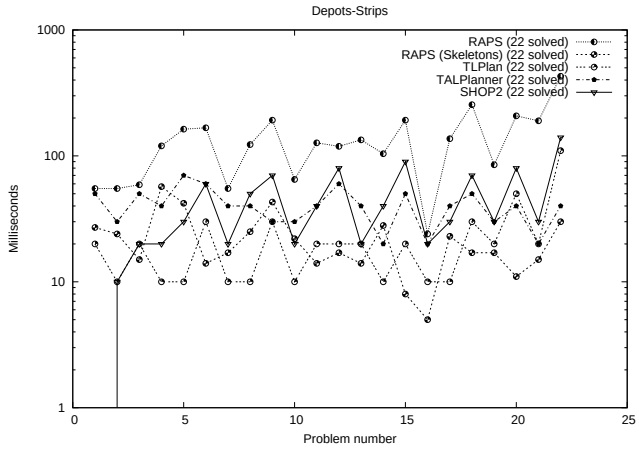

Figure 3: Comparison with IPC 2002 results (typed Depots domain).

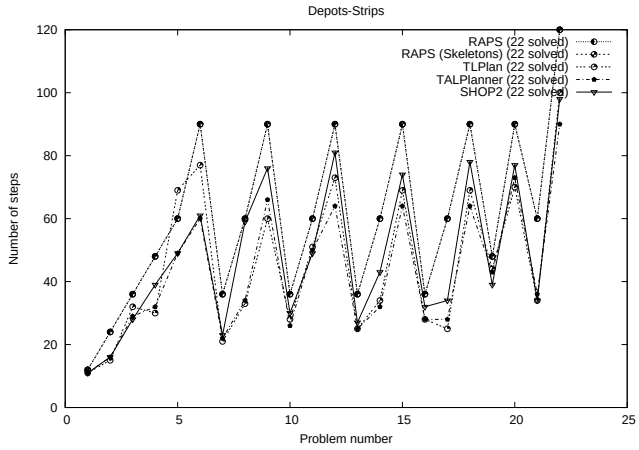

Figure 4: Comparison with IPC 2002 results (solution length in typed Depots domain).

Time spent for computing both, skeleton and full plans, is depicted in Figure 3. There time spent for translating LL problems to Petri nets is included. To keep the figure readable we included data only about the most efficient planners of IPC2002. Although our planner performed computations based on untyped problem specifications only, its performance is still quite close to SHOP2, TLPlan and TALplanner, which were leaders of IPC 2002. Presumably the speed of our planner can be increased, if also type information would be taken into account, since it allows to decrease the number constants considered during unification. Additionally one can see, that skeleton plans, which determine the structure of full plans, are computed generally at the same speed, or even faster, than competition leaders do for problems within the typed version of Depots domain.

In Figure 4 comparison of plan lengths is depicted. By analysing Figure 3 and Figure 4 one can see that there is not much correlation between our plan lengths and skeleton plan computation time. This is due to subplan detection in PNPlanner, where subplan repetitions are computed numerically. However, one of the drawbacks of subplan usage is that the final plan length is not optimal.

Related work

Several works have considered theoretical issues of LL planning. The first of them is presented in (Masseron, Tollu, & Vauzeilles 1993), where a demonstration of robot planning system has been given. Influenced by (Masseron, Tollu, & Vauzeilles 1993), LL theorem proving has been used by Jacopin (Jacopin 1993) as an AI planning kernel for STRIPS-like problem presentations. In (Cresswell, Smaill, & Richardson 1999) a formalism has been proposed for deductively generating recursive plans in LL. This advancement is a step further to more general plans, which are capable of solving instead of a single problem a class of problems. Another approach to deductive planning in a resource-conscious logic (Transition Logic) is given in (Bibel 1998).

In manufacturing automation Petri nets have been used for assembly, disassembly and task sequence planning. Anyway Petri nets used there have specific properties and do not support domain-independent planning. An overview of Petri net usage in manufacturing automation is given in (Moore & Gupta 1996). In (Murata, Nelson, & Yim 1991) general Predicate/Transition nets were used for AI planning. However, while using Place/Transition nets, like in PNPlanner, we are able to recognise and reuse subplans. Propositional planning with Petri nets or within our LL fragment may be viewed as multiset rewriting. Rewriting has been proposed for AI planning in (Vere 1977).

TokenPlan (Fabiani & Meiller 2000) and Petriplan (Silva, Castilho, & Künzle 2000) are domain-independent planners, which incorporate Graphplan (Blum & Furst 1997) and Petri net representation. Then corresponding Petri net analysing methods are applied in a certain stage of planning.

State-of-the-art domain-independent planners include Graphplan (Blum & Furst 1997), SATPLAN (Kautz & Selman 1992), Blackbox (Kautz & Selman 1998) and their derivates. SATPLAN and Blackbox use propositional SAT encodings for representing planning problems and therefore fast algorithms for satisfiability testing can be applied to solve AI planning problems. Further advancements include HSP, HSP-R (Bonet & Geffner 2000) and FF (Hoffmann & Nebel 2001). In (Do & Kambhampati 2001) it has been shown that compiling the Graphplan planning graph into CSP yields additional advancements.

FF, HSP and HSP-R are based on "planning as heuristic search" paradigm, where heuristic information is derived from the specification of the planning instance and

used for guiding the search through the search space. Both time and consumable/renewable resources have been embedded (Haslum & Geffner 2001) to heuristic planning to combine expressivity and performance.

Conclusions

In this article we described how LL planning can be performed efficiently and presented experimental results, which indicate that LL is suitable for domain-independent AI planning. Our planning method performs in two stages. First we abstract a planning problem propositionally. Then a skeleton plan is computed by applying Petri net reachability checking to the propositional problem. After the skeleton plan, consisting of not instantiated operators, has been generated, we instantiate the plan by heuristic-driven constraint solving.

Our planner performs relatively well basically because of three reasons. First, while computing skeleton plans, subplans are detected and numerically their repetition time is computed. Second, by computing first skeleton solutions, a lot of local search is avoided later since the general structure of a plan is already known. And third, by instantiating skeleton plans we perform backjumping according to the current state of the world. This activity may be viewed as heuristic-driven search, whereas heuristic is extracted by comparing the current and an expected (goal) state of the world.

Readers are encouraged to experiment with our LL planner RAPS evaluation tool, which is available at `http://www.idi.ntnu.no/~peep/RAPS`. Also all datasets, used for evaluating performance of our planner, are presented there.

References

Bibel, W. 1998. Let's plan it deductively. *Artificial Intelligence* 103:183–208.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Bonet, B., and Geffner, H. 2000. Planning as heuristic search: New results. In Biundo, S., and Fox, M., eds., *Proceedings of ECP'99, Durham, UK, September 1999*, volume 1809 of *Lecture Notes in Artificial Intelligence*, 360–372. Springer-Verlag.

Cresswell, S.; Smaill, A.; and Richardson, J. 1999. Deductive synthesis of recursive plans in linear logic. In *Proceedings of the Fifth European Conference on Planning*, 252–264.

Do, M. B., and Kambhampati, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence* 132:151–182.

Fabiani, P., and Meiller, Y. 2000. Planning with tokens: an approach between satisfaction and optimisation. In *Proceedings of the 14th Workshop "New Results in Planning, Scheduling and Design" (PuK2000), Berlin, Germany, August 21–22, 2000*, 26–35.

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

Grosse, G.; Hölldobler, S.; and Schneeberger, J. 1996. Linear deductive planning. *Journal of Logic and Computation* 6:232–262.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *Proceedings of IJCAI-01 Workshop on Planning with Resources.*

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Jacopin, É. 1993. Classical AI planning as theorem proving: The case of a fragment of linear logic. In *Proceedings of AAAI Fall Symposium on Automated Deduction in Nonstandard Logics*, 62–66. AAAI Press.

Kanovich, M., and Ito, T. 1997. Temporal linear logic specifications for concurrent processes (extended abstract). In *Proceedings of the Twelfth Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, 29 June–2 July 1997*, 48–57. IEEE Computer Society Press.

Kanovich, M. I., 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):689–716.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, 359–363.

Kautz, H., and Selman, B. 1998. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98, Pittsburgh, PA, June 1998*, 58–60.

Küngas, P. 2002. Resource-conscious AI planning with conjunctions and disjunctions. *Acta Cybernetica* 15:601–620.

Lipton, R. J. 1976. The reachability problem requires exponential space. Research Report 62, Department of Computer Science, Yale University.

Masseron, M.; Tollu, C.; and Vauzeilles, J. 1993. Generating plans in linear logic I–II. *Theoretical Computer Science* 113:349–375.

Moore, K. E., and Gupta, S. M. 1996. Petri net models of flexible and automated manufacturing systems: a survey. *International Journal of Production Research* 34:3001–3035.

Murata, T.; Nelson, P. C.; and Yim, J. 1991. A predicate-transition net model for multiple agent planning. *Information Sciences* 57–58:361–384.

Plaisted, D. A. 1981. Theorem proving with abstraction. *Artificial Intelligence* 16:47–108.

Silva, F.; Castilho, M.; and Künzle, L. A. 2000. Petriplan: a new algorithm for plan generation (preliminary report). In Monard, M. C., and Sichman, J. S., eds., *Proceedings of IBERAMIA-SBIA 2000, Atibaia, SP, Brazil, November 19–22, 2000*, volume 1952 of *Lecture Notes in Computer Science*, 86–95. Springer-Verlag.

Vere, S. A. 1977. Relational production systems. *Artificial Intelligence* 8:47–68.