Deductive Synthesis of Recursive Plans in Linear Logic

Stephen Cresswell, Alan Smaill, and Julian Richardson

Division of Informatics, University of Edinburgh,
80 South Bridge, Edinburgh, EH1 1HN, U.K.
Fax: +44 131 650 6516
{s.cresswell,a.smaill,julian.richardson}@ed.ac.uk

Abstract. Linear logic has previously been shown to be suitable for describing and deductively solving planning problems involving conjunction and disjunction. We introduce a recursively defined datatype and a corresponding induction rule, thereby allowing recursive plans to be synthesised. In order to make explicit the relationship between proofs and plans, we enhance the linear logic deduction rules to handle plans as a form of proof term.

1 Introduction

We are interested in the automated synthesis of recursive plans. In many planning domains, it is possible to construct plans at a more general level than is normally done, e.g. by building plans that can solve a whole class of problems, rather than solving specific ground instances of problems. Typically, a recursive plan is short, but can solve problem instances of arbitrary size — e.g. consider a plan to invert a tower of 1000 blocks.

In the deductive planning approach, plans are formed by constructing proofs in some appropriate logic. These formal approaches to planning, such as situation calculus and modal logic, bring with them a problem of representing frame conditions. In contrast, practical planning techniques are usually based on the STRIPS representation [5], which is hard to extend in a declarative way.

One approach to deductive planning is to make use of Girard's linear logic [7]. This is appealing for planning because it is inherently resource-sensitive. We can use a logical planning framework while retaining a STRIPS-like approach to dealing with the frame problem.

The use of linear logic in planning was first explored by Masseron et al. [15] and [14] and has also been considered by Jacopin [12] and Grosse et al. [8]. However, authors have mainly concentrated on adequacy for simple STRIPS-like plan representations. We think its true potential in planning lies in using its full expressiveness. In this paper, we extend linear logic with an appropriate form of induction that allows us to reason about recursive plans.

Section 2 gives a brief introduction to a fragment of intuitionistic linear logic, focussing on its application in planning problems.

Section 3 gives a plan representation which allows a plan to be built directly in the course of a linear logic proof. We then extend this scheme to give deduction rules with plan terms for a larger fragment of linear logic, allowing conditional and conformant plans to be represented.

Section 4 extends the linear logic with an appropriate induction rule for a recursive datatype, allowing proofs to be developed for recursive plans. Section 5 discusses automated proof search (i.e. a planner) using this framework. Sections 6 and 7 relate this work to existing work in the field, and draw conclusions.

2 Linear Logic

We briefly describe linear logic here; see [7] for a full exposition, and [15] for an account in relation to a semantics for planning problems.

Linear logic is resource-sensitive. This gives us the ability to model change of state directly using the linear version of implication, written as $\multimap$. The usual example here is that we can model the scenario that we can buy a drink for a pound as follows:

$$have\_pound \multimap have\_drink.$$

The notion of implication here is that if we have a pound, we can have a drink, but (unlike conventional implication) we won't still have the pound. The resource on the left of the implication is used up in producing the resource on the right.

These rules can only be seen as transitions if the logic itself restricts the copying and discarding of resources.

Now if we consider resource-limited versions of familiar connectives such as conjunction, we find that there are two different versions possible, differing in the way the resources are handled.

The *multiplicative conjunction* $A \otimes B$ means that resources A and B are simultaneously present. This is the form of conjunction which is used in STRIPS-like planning problems.

The second form, *additive conjunction*, means that both resources are individually available, but they are exclusive. Only one or the other may be used and the choice is ours.

$$have\_pound \multimap have\_tea \;\&\; have\_coffee$$

Although this looks somewhat more like a disjunction, it is regarded a conjunction, since it would be equivalent to the conventional conjunction if copying and discarding of resources were allowed.

This may be can contrasted with the additive disjunction, $\oplus$.

$$have\_pound \multimap have\_tea \oplus have\_coffee$$

This would correspond to using an erratic drinks machine, which will deliver either tea or coffee, but we cannot choose which.

2.1 Planning with Linear Logic

A planning problem can be represented as a sequent of the form $I \vdash G$ where I represents initial conditions and G represents goal conditions. In this intuitionistic version, I is a multiset of formulae (implicitly joined by $\otimes$). The $\vdash$ behaves as linear implication, so we can read this as meaning that the resources I should be consumed in deriving G. Similarly, we use transition axioms of the form $P \vdash E$ to represent operators. These axioms can be reused as many times as necessary in the proof, each use corresponding to an action. For instance, we could represent an operator $stack(X, Y)$ for placing a block as follows:

$$hold(X), clr(Y) \vdash empty \otimes clr(X) \otimes on(X, Y)$$

To see the correspondence with STRIPS operators, consider the STRIPS version of $stack(X, Y)$. This can be written as:

```
operator:      stack(X,Y)
preconditions: hold(X)
               clr(Y)
deletelist:    hold(X)
               clr(Y)
addlist:       clr(X)
               on(X,Y)
               empty
```

Note that the main difference between the two renditions is that there is no equivalent of the delete list in the linear logic description. This is not needed because anything used as a precondition will automatically be consumed by the linear logic version of implication. This can simply represent problems from the STRIPS notation, since any preconditions which are required but not consumed by an action can simply be added back onto the right hand side of the $\vdash$ in the action definition.

Another significant difference is that in linear logic, multiple instances of the same entity are regarded as distinct. For example, we could represent the situation of having two pounds as $have\_pound \otimes have\_pound$.

We can create proofs for solving simple STRIPS-like planning problems using only the $\otimes$ connective and the rules $Ax, cut, l\otimes, r\otimes$.

$$\frac{}{A \vdash A}\ Ax \qquad\qquad \frac{\Gamma \vdash A \quad \Gamma', A \vdash C}{\Gamma, \Gamma' \vdash C}\ cut$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ l\otimes \qquad\qquad \frac{\Gamma \vdash A \quad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \otimes B}\ r\otimes$$

The cut rule is crucial here, as it can now be seen as a rule which allows the transition between two states to be made via some intermediate state, and this accounts for the composition of a plan by combining two sub-plans in sequence.

An example of the use of the cut rule is given below. This shows the transition from a state described by $empty, clr(c), on(c,a), clr(b), ontable(b)$ by the application of a $remove$ action to a state described by $hold(c), clr(a), clr(b), ontable(b)$.

$$\dfrac{\overbrace{empty, clr(c), on(c,a) \vdash hold(c) \otimes clr(a)}^{remove(c,a)} \quad \dfrac{\vdots}{hold(c) \otimes clr(a), clr(b), ontable(b) \vdash G}}{empty, clr(c), on(c,a), clr(b), ontable(b) \vdash G}\ cut$$

If the cut rule is always applied to sequents in which the hypothesis list is a superset of the formulae on the left of a transition axiom, the proof corresponds to a plan built forwards from the initial state.

So we can build a proof tree with a statement of the planning problem at its root, and with instances of transition axioms and Ax at its leaves. If we can build such a proof, we can be satisfied that there is a plan that solves the problem, where applications of the transition axioms correspond to simple actions in the plan. However, some work is still needed to extract the plan from the proof tree. One way to do this is by analysis of the proof, as proposed in [14]. An alternative approach is described in section 3.

3 Extraction of Plans

In previous publications, authors have used a procedure for extracting plans from the completed proof. Here, we will make the relationship of the proof to the plan more concrete by attaching proof terms directly to the deduction rules in the style of type theory [16]. This makes the relationship of deduction rules and plan formation clearer, and is easier to extend to deal with a larger subset of linear logic.

A type theory has been defined for linear logic in [1]. Proof terms can be seen as programs in linear lambda calculus — a functional language in which there is a restriction of using each input once.

However, this allows the construction of a more powerful language than we want to deal with in a planning problem. We wish to restrict our consideration to the proof terms which have a simple imperative interpretation. We describe such a system below, in which sequents of the form $A \vdash \texttt{plan} : C$ should be interpreted as meaning that *plan* gets us from a state described by resources A to a state described by C.

The plan terms are built with the following syntax:

plan ::=	*step*	a single-step action
	| `nop`	do nothing
	| *plan*;*plan*	execute plans sequentially
	| `par`(*plan*, *plan*)	execute plans in parallel
	| $\lambda v.plan$	plan parameterised by v

For notational convenience, we will often omit plan terms on the left side of the sequent.

3.1 Transition Axioms

Operator definitions now take the following form:

$$A \vdash \texttt{step} : C$$

3.2 Identity Axiom and Cut Rule

Identity corresponds to a transition to the same state, so the corresponding plan is *nop*. The *cut* rule corresponds to the sequential composition of two plans.

$$\frac{}{A \vdash \texttt{nop} : A}\ Ax \qquad\qquad \frac{\Gamma \vdash \texttt{a} : A \quad \Gamma', A \vdash \texttt{c} : C}{\Gamma, \Gamma' \vdash (\texttt{a}; \texttt{c}) : C}\ cut$$

3.3 Multiplicative Conjunction

The $\otimes$ rule allows a planning problem to be broken down into two independent sub-problems. Since the sub-problems rely on disjoint sets of resources, the plan term consists of two sub-plans in parallel.

$$\frac{\Gamma, A, B \vdash \texttt{c} : C}{\Gamma, A \otimes B \vdash \texttt{c} : C}\ l\otimes \qquad\qquad \frac{\Gamma \vdash \texttt{a} : A \quad \Gamma' \vdash \texttt{b} : B}{\Gamma, \Gamma' \vdash \texttt{par}(\texttt{a}, \texttt{b}) : A \otimes B}\ r\otimes$$

3.4 Quantifiers

To make recursive plans, we will need to be able to handle quantification in some form. The quantifier rules for linear logic are the same as a version of the standard ones. However, the meaning of $\forall$ is *for any* rather than *for every* — a distinction which is not meaningful in constructive or classical logic. The resultant plan is parameterised, and will provide a plan for a specific instance when supplied with a value for the parameter. We assume β-reduction is used to compute the plan instances.

$$\frac{\Gamma, \texttt{j} : P(t) \vdash \texttt{A} : C}{\Gamma, \texttt{i} : \forall u.P(u) \vdash \texttt{A}[\texttt{i}(\texttt{t})/\texttt{j}] : C}\ l\forall \qquad\qquad \frac{\Gamma \vdash \texttt{c} : C[a/x]}{\Gamma \vdash \lambda\texttt{a}.\texttt{c} : \forall x.C}\ r\forall$$

where a is not free in Γ, C.

3.5 Disjunctive Effects and Conditionals

In some planning problems, it may be desirable to represent plans with indeterminate outcomes. These can be represented by actions with disjunctive effects. Here it is appropriate to use the additive disjunction operator $\oplus$. A formula $A \oplus B$ should be interpreted as saying that either resource A or resource B is available, and in a plan, we must cope with both possibilities.

[15] gives an example of a problem in which socks are blindly taken from a drawer. This action is represented by a transition in which a hidden sock (hs) becomes either a black sock (bs) or a white sock (ws).

$$hs \vdash \texttt{pick} : bs \oplus ws$$

In resolving these disjunctions during the planning process, there are two possibilities: the agent which will execute the plan may or may not be capable of performing a test to resolve the disjunction at runtime.

If the agent can perform a test at runtime, it can select between two different plans, i.e. we can build a conditional structure.

$$\frac{\Gamma, A \vdash \texttt{Plan1} : C \quad \Gamma, B \vdash \texttt{Plan2} : C}{\Gamma, A \oplus B \vdash \texttt{if}^{\texttt{A then Plan1}}_{\texttt{B then Plan2}} : C} \; l\oplus (test)$$

This is equivalent to forms used by Brüning et al. [3] and by Abramsky [1].

If the agent cannot perform a test (cannot look at the sock), we must ensure that the same plan will work regardless of which condition holds (black or white sock). This form of planning problem has been called *conformant* or *fail-safe* planning. Here we use the following form of the $l\oplus$ rule, in which the same $Plan$ will work for both A and B.

$$\frac{\Gamma, A \vdash \texttt{Plan} : C \quad \Gamma, B \vdash \texttt{Plan} : C}{\Gamma, A \oplus B \vdash \texttt{Plan} : C} \; l\oplus$$

Note that the proof term is here being used to restrict the kind of proof which can be extracted, i.e. to enforce the condition that the plan for each branch is identical. This would not be possible in other linear logic planning schemes, in which the term representing the plan itself is not present in the proof.

For a disjunction of goals, we must simply prove one or the other:

$$\frac{\Gamma \vdash \texttt{p} : B}{\Gamma \vdash \texttt{p} : B \oplus C} \; r1\oplus$$

$$\frac{\Gamma \vdash \texttt{p} : C}{\Gamma \vdash \texttt{p} : B \oplus C} \; r2\oplus$$

4 Induction and Formation of Recursive Plans

We can introduce a simple recursive datatype and synthesise recursive plans by performing a proof by induction over that datatype. Note that recursive plans assume the ability to test in the sense of the rule $l \oplus (test)$, to recognise when base and step case actions apply.

4.1 Recursive Datatype and Corresponding Induction Rule

We define a datatype for towers:

$$tower := (block :: tower) \mid empty$$

Now we define an induction rule for this datatype. Note that we must be quite careful about how this is defined. Since a given resource may only be consumed once, we must make sure that formulas used in the step case are either dependent on the datatype or are replenished if used. Γ is made available only in the base case, which corresponds to a plan that is only executed once.

The syntax for plans is extended by adding the notion of recursive plan, based on the treatment of recursion in type theory [16]. If t is a tower, $\mathtt{bp}$ is a plan and $\lambda\mathtt{x}.\lambda\mathtt{y}.\lambda\mathtt{z}.\mathtt{sp}$ is a plan with three parameters, then $\mathtt{rec\_twr}(\mathtt{t}, \mathtt{bp}, \lambda\mathtt{x}.\lambda\mathtt{y}.\lambda\mathtt{z}.\mathtt{sp})$ is a recursive plan. Recursive plans are executed as follows. If $t = empty$, then execute $\mathtt{bp}$; otherwise t is of the form $b' :: t'$, so evaluate the plan

$$\mathtt{sp}[(\mathtt{rec\_twr}(\mathtt{t}', \mathtt{bp}, \lambda\mathtt{x}.\lambda\mathtt{y}.\lambda\mathtt{z}.\mathtt{sp}))/\mathtt{x}, \mathtt{b}'/\mathtt{y}, \mathtt{t}'/\mathtt{z}]$$

and execute it.

The inference rule that relates induction to the formation of recursive plans is as follows:

$$\frac{\vdash \mathtt{bp} : F(empty) \quad \mathtt{r} : F(t') \vdash \mathtt{sp} : F(b' :: t')}{\vdash \lambda\mathtt{t}.\mathtt{rec\_twr}(\mathtt{t}, \mathtt{bp}, \lambda\mathtt{r}.\lambda\mathtt{t}'.\lambda\mathtt{b}'.\mathtt{sp}) : \forall t.F(t)}$$

In general, the $F(t)$ will be a plan specification of the form $I(t) \multimap G(t)$. The term r attached to the induction hypothesis will behave as an available action, and its appearance in the plan for the step case signifies the application of the recursive call. It must be used exactly once in the step case plan.

It is helpful to introduce a custom rule in this scenario. This rule is derivable from the standard rules for handling linear implication.

$$\frac{P_1 \vdash \mathtt{c} : P_2}{\mathtt{j} : P_2 \multimap Q \vdash (\mathtt{c}; \mathtt{j}) : P_1 \multimap Q}\ c\multimap$$

This is not the most general form, but it will simplify the presentation of the following example significantly.

4.2 Example

The following example shows a solution to the problem of inverting a tower of blocks. The specification of the problem itself requires the use of a recursively defined function.

As we shall see in the example below, our proof procedure requires both the application of deduction rules, contributing to the instantiation of a plan term, and the application of rewrite rules, to transform equivalent expressions into the same syntactic form.

For this example, we wish to solve a problem of the form:

$$\vdash \texttt{plan} : \forall t.\, [twr(t) \multimap twr(rev(t))]$$

which means that a tower t can be reversed by execution of *plan*. $twr(t)$ should be read meaning that a tower t of unknown height is present. rev is a reverse function — it allows the relationship between initial and goal states to be described. It does not describe an action.
The following will be used as plan operators.

$$twr(B :: T), hn \vdash \texttt{pickup(B)} : twr(T) \otimes hold(B)$$
$$twr(T), hold(B) \vdash \texttt{put(B,T)} : twr(B :: T) \otimes hn$$

where hn stands for "holding nothing". Since we are going to need to pass through intermediate steps in the plan where we have two separate towers, it is necessary to include a reference to a second tower throughout. The problem of finding this generalisation of the original problem is addressed in section 5.

$$\forall t, a.twr(t) \otimes twr(a) \otimes hn \multimap twr(empty) \otimes twr(app(rev(t), a)) \otimes hn$$

The definitions of rev and app are given by:

$$rev(empty) = empty \tag{1}$$
$$rev(b :: t) = app(rev(t), b :: empty) \tag{2}$$

$$app(empty, u) = u \tag{3}$$
$$app(b :: t, u) = b :: app(t, u) \tag{4}$$

We will also make use of associativity of app.

$$app(app(a, b), c) = app(a, app(b, c)) \tag{5}$$

We can begin the proof by using an induction on t (for clarity, we will omit plan terms).

For the base case, we must prove:

$$\vdash \forall a.twr(empty) \otimes twr(a) \otimes hn \multimap twr(empty) \otimes twr(app(rev(empty), a)) \otimes hn$$

After application of rewrites 1,3 we get:

$$\vdash \forall a.twr(empty) \otimes twr(a) \otimes hn \multimap twr(empty) \otimes twr(a) \otimes hn$$

which can be proved trivially.

For the step case we will walk through the proof in the direction of its construction i.e. from bottom to top. The full sequent proof is given as an appendix. The initial goal is:

$$\forall a.twr(t) \otimes twr(a) \otimes hn \multimap twr(empty) \otimes twr(app(rev(t), a)) \otimes hn \vdash \forall a.twr(b :: t) \otimes twr(a) \otimes hn \multimap twr(empty) \otimes twr(app(rev(b :: t), a)) \otimes hn$$

We can eliminate the universal quantifiers: first on the right, then on the left. We allow meta-variables (A in this proof) so as to delay the choice of witness term t in the use of the rule $l\forall$

$$\begin{array}{c} twr(t) \otimes twr(A) \otimes hn \multimap \\ twr(empty) \otimes twr(app(rev(t), A)) \otimes hn \end{array} \vdash \begin{array}{c} twr(b :: t) \otimes twr(a') \otimes hn \multimap \\ twr(empty) \otimes twr(app(rev(b :: t), a')) \otimes hn \end{array}$$

Now we apply rewrite rules 2,5,4,3 to the R.H.S.

$$\begin{array}{c} twr(t) \otimes twr(A) \otimes hn \multimap \\ twr(empty) \otimes twr(app(rev(t), A)) \otimes hn \end{array} \vdash \begin{array}{c} twr(b :: t) \otimes twr(a') \otimes hn \multimap \\ twr(empty) \otimes twr(app(rev(t), b :: a')) \otimes hn \end{array}$$

By instantiating the meta-variable to $b :: a'$, we can now apply the $c \multimap$ rule, which leads to:

$$twr(b :: t), twr(a'), hn \vdash twr(t) \otimes twr(b :: a') \otimes hn$$

Now, we can complete the proof by cutting in applications of the *pickup* and *put* actions (see appendix for complete proof of step case).
The plan can now be given as:

$$\begin{array}{l} \lambda \mathtt{t}.\mathtt{rec\_twr}(\mathtt{t}, \mathtt{nop}, \\ \quad \lambda \mathtt{r}.\lambda \mathtt{t}'.\lambda \mathtt{b}'.\lambda \mathtt{a}'.(\mathtt{par}(\mathtt{nop}, \mathtt{pickup}(\mathtt{b}')); \mathtt{par}(\mathtt{nop}, \mathtt{put}(\mathtt{b}', \mathtt{a}')); \mathtt{r}(\mathtt{t}', \mathtt{a}'))) \end{array}$$

which can be simplified to:

$$\lambda \mathtt{t}.\mathtt{rec\_twr}(\mathtt{t}, \mathtt{nop}, \lambda \mathtt{r}.\lambda \mathtt{t}'.\lambda \mathtt{b}'.\lambda \mathtt{a}'.(\mathtt{pickup}(\mathtt{b}'); \mathtt{put}(\mathtt{b}', \mathtt{a}'); \mathtt{r}(\mathtt{t}', \mathtt{a}')))$$

As we are synthesising a plan with two parameters (corresponding to the two universal quantifiers in the original specification), the recursive call $\mathtt{r}(\mathtt{t}', \mathtt{a}')$ requires values t', a' for the parameters.

5 Automated Proof Search

An attractive approach to proof search in linear logic is to use a form of logic programming based on linear logic rather than on classical (or constructive) logic. We have used the language Lolli [10] as a theorem prover in our experiments. These languages are based on intuitionistic linear logic, with some restrictions to permit a uniform proof procedure to be carried out (for example, expressions of the form $A \otimes B$ are not allowed as the head of a clause). These restrictions prevent us from writing operators directly as clauses in Lolli, and finding a plan by executing the query that asks whether the initial resources can be transformed into the goal resources. However, a plan-forming routine can be written to interpret declared operators and queries in the style of a meta-interpreter [18], while building up the extracted plan, and allowing an induction rule to be used. This provides a tidy representation of the planning problem, and allows bookkeeping

of the resources to be handled by the logic programming machine, but is not particularly efficient for finding a solution.

In the recursive planning problems, the process of synthesising the plan (and the proof) now consists not only of manipulations which correspond to plan steps, but also rewrites of terms to equivalent forms. Uniform search is not enough in this situation, and we lean on work in guiding inductive proof in standard logic, described in [4]. Heuristics are available to help at the difficult choice points, namely which induction to use (there may be several schemes possible, and variables to choose from), and choice of rewrites to apply.

The generalisation step in the example in section 4.2 is amenable to automation. A similar problem is discussed in [9, 11]: when the problem cannot be solved in its given form, the failed proof attempt gives rise to information about the form of generalisation that would allow the proof to be completed. In program synthesis, this may suggest the need for an accumulator variable, and an appropriate specification of the generalised procedure derived in the course of the proof. A similar analysis applies for plan synthesis.

6 Related Work

6.1 Linear Logic Planners

The work of Masseron et al. [15, 14] has been the starting point for this work. Masseron showed the adequacy of linear logic for simple conjunctive planning problems, for which a procedure for extracting plans in the form of directed graphs was given. Their representation for partially-ordered plans is less redundant than ours, but more awkward to extend to a larger fragment of the logic. Problems involving disjunction are covered by the representation but not by the geometric argument relating proofs to plans. A procedure for proof search using the conjunctive fragment is given in [12].

[8] makes clear the close relationship between their technique of planning with equational resolution, Bibel's linear connection method [2], and linear logic. Techniques based on the connection method have been used both as deductive planners and as linear logic theorem provers.

6.2 Recursive Planning

The approach of Manna and Waldinger [13] uses a version of the situation calculus. Their formalism accounts for the derivation of conditional and recursive plans. Their work identifies the need for generalisation in performing inductive proof as being a major problem.

The RNP planner of Ghassem-Sani and Steel [6] described RNP, an implementation of a recursive planner based on goal-driven partial order planning. This allowed for a principled method for deciding when to introduce a recursive construct. The plans generated by this planner used a conventional representation of a plan as a directed graph, and conditional and recursive constructs were

considered as special types of node containing sub-plans. An interesting feature of their approach is that induction rules are constructed dynamically from a well-founded order.

The approach described in this paper differs from these works in that our induction is based on a given rule for an inductive datatype, rather than constructing rules from a well-founded order. We believe our formulation allows stronger control of the proof search, while obviating the need to reason about partially defined functions.

The deductive planning work of Stephan and Biundo [17] uses a temporal logic to model problems and plans. There is a sophisticated theorem-proving environment for domain modelling and plan formation. In their approach, formation of recursive procedures in considered part of the domain modelling, where such procedures can be developed and proved interactively with the theorem prover. After this stage, plans can then be constructed automatically by refinement of specifications.

Unlike Stephan and Biundo, we are interested in automating the recursive plan formation process.

7 Conclusion

We believe the combination of linear logic and induction is a promising avenue for the formation of recursive plans in a declarative way. It allows us to bring experience in the control of proof search in inductive proofs to bear on this problem. Furthermore the expressive nature of linear logic allows a natural and concise formulation of several interesting extensions to conventional planning that can be incorporated in our framework.

In order to demonstrate the usefulness of this approach, we are currently devising a set of examples using different inductively defined datatypes, for example, trees.

As other authors have noted [13], the generalisation problem is central to full automation. As was mentioned in section 5, this problem is amenable to automation, but we have not yet explored the applicability of that approach in this context.

Acknowledgements

Thanks to Louise Pryor for discussions on this and related material. Stephen Cresswell was supported by an EPSRC studentship. Alan Smaill and Julian Richardson are supported by EPSRC grant GR/M45030.

References

1. S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993. (Revised version of Imperial College Technical Report DoC 90/20).

2. W. Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.
3. S. Brüning, S. Hölldobler, J. Schneeberger, U. Sigmund, and M. Thielscher. Disjunction in resource-oriented deductive planning. In *Proceedings of the International Symposium on Logic Programming*, page 670, 1993.
4. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.
5. R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
6. G. R. Ghassem-Sani and S. W. D. Steel. Recursive plans. In *Proc. of the European Workshop on Planning EWSP-91*, pages 53–63, St. Augustin, Germany, 1991.
7. J. -Y. Girard. Linear logic: its syntax and semantics. In J. -Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Notes Series. Cambridge University Press, 1995.
8. G. Große, S. Hölldobler, and J. Schneeberger. Linear deductive planning. *Journal of Logic and Computation*, 6(2):233–262, 1996.
9. J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, pages 310–324, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.
10. J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.
11. A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.
12. E. Jacopin. Classical AI planning as theorem proving: The case of a fragment of linear logic. In *AAAI Fall Symposium on Automated Deduction in Nonstandard Logics, Technical Report FS-93-01*, pages 62–66. AAAI Press Publications, Palo Alto, 1993.
13. Z. Manna and R. Waldinger. How to clear a block: a theory of plans. *Journal of Automated Reasoning*, 3(4):343–377, 1986.
14. M. Masseron. Generating plans in linear logic II: A geometry of conjunctive actions. *Theoretical Computer Science*, 113:371–375, 1993.
15. M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic I: Actions and proofs. *Theoretical Computer Science*, 113(2):349–371, 1993.
16. B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf Type Theory*. Oxford University Press, 1990.
17. W. Stephan and S. Biundo. Deduction based refinement planning. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 213–220. AAAI Press, 1996.
18. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, second edition, 1994.

Appendix: Step Case Proof

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{twr(a') \vdash twr(a')}\ Ax \quad \dfrac{}{twr(b :: t), hn \vdash twr(t) \otimes hold(b)}\ (pickup)
}{twr(b :: t), twr(a'), hn \vdash twr(t) \otimes twr(a') \otimes hold(b)}\ r\otimes
}{twr(b :: t) \otimes twr(a') \otimes hn \vdash twr(t) \otimes twr(a') \otimes hold(b)}\ l\otimes
\quad
\dfrac{
\dfrac{
\dfrac{}{twr(t) \vdash twr(t)}\ Ax \quad \dfrac{}{twr(a'), hold(b) \vdash twr(b :: a') \otimes hn}\ (put)
}{twr(t), twr(a'), hold(b) \vdash twr(t) \otimes twr(b :: a') \otimes hn}\ r\otimes
}{twr(t) \otimes twr(a') \otimes hold(b) \vdash twr(t) \otimes twr(b :: a') \otimes hn}\ l\otimes
}{
\dfrac{
twr(b :: t), twr(a'), hn \vdash twr(t) \otimes twr(b :: a') \otimes hn
}{
twr(b :: t) \otimes twr(a') \otimes hn \vdash twr(t) \otimes twr(b :: a') \otimes hn
}\ l\otimes, l\otimes
}\ cut
$$

$$
\dfrac{
\begin{array}{c}
twr(t) \otimes twr(A) \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), A)) \otimes hn
\end{array}
\vdash
\begin{array}{c}
twr(b :: t) \otimes twr(a') \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), b :: a')) \otimes hn
\end{array}
}{
\begin{array}{c}
twr(t) \otimes twr(A) \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), A)) \otimes hn
\end{array}
\vdash
\begin{array}{c}
twr(b :: t) \otimes twr(a') \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(b :: t), a')) \otimes hn
\end{array}
}\ r\forall, l\forall
$$

$$
\dfrac{
\begin{array}{c}
\forall a. twr(t) \otimes twr(a) \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), a)) \otimes hn
\end{array}
\vdash
\begin{array}{c}
\forall a. twr(b :: t) \otimes twr(a) \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(b :: t), a)) \otimes hn
\end{array}
}{
\begin{array}{c}
twr(t) \otimes twr(A) \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), A)) \otimes hn
\end{array}
\vdash
\begin{array}{c}
twr(b :: t) \otimes twr(a') \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), b :: a')) \otimes hn
\end{array}
}\ 2, 5, 4, 3
$$

$$
\dfrac{
\begin{array}{c}
twr(t) \otimes twr(A) \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), A)) \otimes hn
\end{array}
\vdash
\begin{array}{c}
twr(b :: t) \otimes twr(a') \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(b :: t), a')) \otimes hn
\end{array}
}{
\begin{array}{c}
twr(t) \otimes twr(A) \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(t), A)) \otimes hn
\end{array}
\vdash
\begin{array}{c}
twr(b :: t) \otimes twr(a') \otimes hn \multimap \\
twr(empty) \otimes twr(app(rev(b :: t), a')) \otimes hn
\end{array}
}\ c\multimap, A = b :: a'
$$