Deductive Synthesis of Recursive Plans in Linear Logic

Stephen N. Cresswell

Ph.D.
University of Edinburgh
2001

Dedicated to the memory
of my father,
Eric Cresswell.

Abstract

Conventionally, the problem of plan formation in Artificial Intelligence deals with the generation of plans in the form of a sequence of actions.

This thesis describes an approach to extending the expressiveness of plans to include conditional branches and recursion. This allows problems to be solved at a higher level, such that a single plan in such a language is capable of solving a class of problems rather than a single problem instance. A plan of fixed size may solve arbitrarily large problem instances.

To form such plans, we take a deductive planning approach, in which the formation of the plan goes hand-in-hand with the construction of the proof that the plan specification is realisable.

The formalism used here for specifying and reasoning with planning problems is Girard's Intuitionistic Linear Logic (ILL), which is attractive for planning problems because state change can be expressed directly as linear implication, with no need for frame axioms. We extract plans by means of the relationship between proofs in ILL and programs in the style of Abramsky.

We extend the ILL proof rules to account for induction over inductively defined types, thereby allowing recursive plans to be synthesised. We also adapt Abramsky's framework to partially evaluate and execute the plans in the extended language.

We give a proof search algorithm tailored towards the fragment of the ILL employed (excluding induction rule selection). A system implementation, Lino, comprises modules for proof checking, automated proof search, plan extraction and partial evaluation of plans.

We demonstrate the encodings and solutions in our framework of various planning domains involving recursion. We compare the capabilities of our approach with the previous approaches of Manna & Waldinger, Ghassem-Sani & Steel and Stephan & Biundo. We claim that our approach gives a good balance between coverage of problems that can be described and the tractability of proof search.

Acknowledgements

I would like to thank my supervisors, Dr. Alan Smaill and Dr. Julian Richardson for being extremely supportive and patient. Thanks also to Dr. Louise Pryor, who was a supervisor in the earlier stages of the work, and to members of the DReaM group at Edinburgh.

Dr. Maria Fox and Dr. Derek Long generously made allowances for me to complete the work whilst working in the Planning Group at Durham University.

I would like to thank my examiners, Dr. Paolo Traverso and Prof. Michael Fourman for giving valuable constructive feedback.

I would also like to thank my family and friends for all their support and encouragement.

This work was supported by an EPSRC studentship.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Stephen Cresswell
Edinburgh
November 20, 2001

Publications

Parts of this thesis have already appeared in print, have been submitted for publication, or have been made publicly available:

Stephen Cresswell, Alan Smaill, and Julian Richardson. Deductive synthesis of recursive plans in linear logic. In *Proceedings of the Fifth European Conference on Planning ECP-99*, pages 252–263, Durham, UK, 1999.

Contents

List of Figures

Chapter 1

Introduction

Planning is concerned with the problem of finding a course of action that will achieve a given goal. In conventional AI planning, the problem is specified by giving an initial state, some desired goal conditions and a repertoire of available actions that can be performed. The solution consists of a set of applications of the actions, together with a partial or total order for their execution.

This means that:

1. The planner must laboriously discover the whole plan step-by-step, even when it has a very simple repetitive structure. The performance of planners generally scales very badly with problem size.
2. The planner can only solve the problem if every detail is known about the initial state of the world.

We are interested in problems in which plans involving repetition are useful. A simple way to model repetitive plans is by the use of recursion.

Using recursion allows a some kinds of planning problem to be solved at a higher level, in which we know the structure but not necessarily the details. For instance, we could form a general plan to invert a tower of blocks without knowing in advance exactly how many blocks are involved. This tackles issue (1) and also addresses a form of (2).

This thesis describes a recursive planning approach based on linear logic [Girard 87]. Plans are constructed in a language that can express recursion, conditional branching,

and partial ordering of steps. The plan is found by searching for a linear logic proof that the plan specification is realisable. A system has been implemented which can solve a range of problems in a semi-automatic way.

1.1 Context

There have been several notable studies of recursive planning. It is natural to approach the task in the deductive planning framework, in which the formation of plans takes place as a by-product of forming a proof in some appropriate logic.

Manna and Waldinger [Manna & Waldinger 87] focussed on formulating a deductive system that was expressive enough to describe state-changing actions. Using a deductive approach based on situation calculus, they demonstrated the formation of recursive plans by building the corresponding inductive proofs. However, it is difficult for a search procedure to automatically control inference in their framework. They do not give an automatic search algorithm for their framework.

Ghassem-Sani and Steel [Ghassem-Sani & Steel 91] use an enhanced STRIPS representation in a non-linear planner. This gives a good solution to search control, but at the expense of limiting the representation to the extent that many interesting problems cannot be tackled.

In general, we have a trade-off between the expressiveness of the representation and the ease with which proof search can be carried out. Efficient search is easier for less expressive formalisms which, on the other hand, cannot be used to formulate the sorts of problems we are interested in.

In this way, previous work on formation of recursive plans has either given a powerful logical representation to the problem which is not amenable to automation, or has made search feasible by limiting the logic so much that many problems cannot be represented.

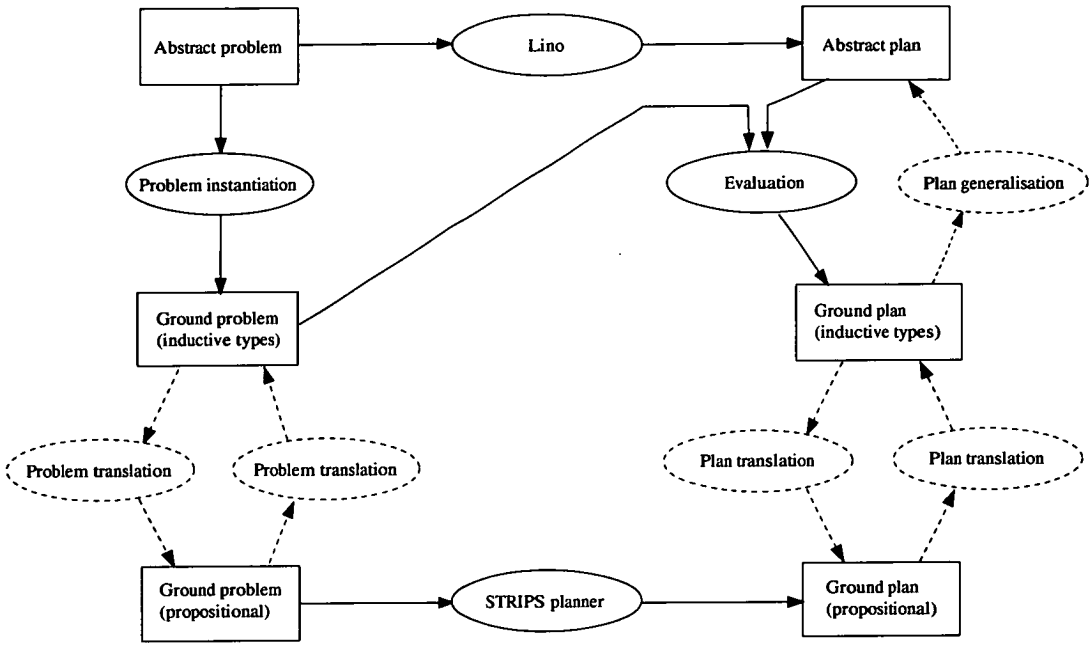


Figure 1.1: Problems, plans and processes.

1.2 Approach

One of the difficulties of describing actions in a logical system is that of describing exactly what changes and what remains the same as the result of performing an action. This is known as the *frame problem.*

Linear logic offers a simple built-in solution to this problem, as its own version of implication can be read directly as describing a state change. This makes it a particularly attractive logic on which to base our deductive planning system.

Linear logic has previously been applied to planning only as a small fragment for dealing with a limited kind of plans [Masseron *et al* 93]. We consider that its strength lies in using a larger fragment of the logic to talk about a more general sort of plan.

We show how plans can be formed in this language by associating terms representing constructs in the plan language directly with the deduction rules of the logic. This accounts for a plan language that can describe partial ordering of plan steps, conditional branches, and actions with uncertain effects. Such a language is given in [Abramsky 93], and we extend this for our purpose.

In order to set our work in the context of STRIPS planning we consider three repre-

sentations for problems, and their corresponding plan representations.

1. Planning problems stated in a STRIPS representation — i.e. propositional, with a finite set of objects in the domain.

2. Planning problems stated in a linear logic representation allowing inductively defined datatypes, but with plan specification fully instantiated.

3. Planning problems stated in a linear logic representation allowing inductively defined datatypes, in which we seek a general recursive solution.

Here (1) is the representation conventionally used for planning problems. (3) is the representation we have used, and (2) is an intermediate stage. Fig. 1.1 shows the methods that may be used to convert between plans and processes in these representations.The solid ellipses represent processes for which we have an algorithm. The dotted ellipses represent processes which have not been automated.

This thesis deals mainly with the generation of abstract solutions from abstract expressions of the problems. By also introducing induction rules into the logic we can form recursive plans. These ideas form the basis of an interactive proof checker, which can be used to synthesise plans interactively to meet to a given specification.

We describe a proof search strategy which can solve problems automatically if specified using a certain fragment of the logic. This fragment is chosen carefully to be just expressive enough to represent the features required for a good coverage of planning problems.

Novel aspects are:

- We extend linear logic with induction rules to allow the formation of recursive plans.

- We give a new proof procedure which is adequate for a fragment of intuitionistic linear logic chosen for expressing planning problems.

- We have a system that is expressive enough to model interesting recursive problems, but restricted enough for automated solution to be realistic.

1.3 Thesis structure

Chapter 2 reviews approaches to planning, considering the situation calculus and STRIPS representations. The compromise between expressiveness and tractability is considered. Efficient planners based on the STRIPS representation are described. Extensions to deal with the formation of conditional structures are reviewed.

Chapter 3 reviews previous work on systems for forming recursive plans. Manna and Waldinger's system is general but has never been automated. Ghassem-Sani and Steel implemented a more limited automatic system based on STRIPS planning, but no proof of its correctness. Stephan and Biundo implemented a deductive system in which interactive proof of recursive plans was a preparatory stage for a fully automatic planner.

Chapter 4 introduces linear logic, and explains how it was used by Masseron to account for simple planning problems. Masseron gives a plan extraction procedure which allows partially-ordered plans to be extracted from proofs.

In Chapter 5, we introduce our scheme for relating plans to linear logic proofs. This enables a larger fragment of the linear logic to be used in plan formation with a correspondingly richer plan language. For each construct in the plan language, we define what it means to execute that construct.

Chapter 6 introduces inductive datatypes, and shows how appropriate induction rules can be added to the proof system. The corresponding constructs in the plan language are defined. Examples are given of recursive planning problems involving various datatypes and induction rules.

Chapter 7 considers automated proof search in this framework. Techniques from inductive theorem proving such as rippling for controlling rewriting and generalisation are described. Other techniques from planning and linear logic programming are also considered. We then give a search algorithm which is specialised for planning problems expressed in an appropriately chosen fragment of linear logic.

Chapter 8 describes Lino, our implemented system incorporating proof checking, proof search, and the extraction, partial evaluation and execution of plans.

Chapter 9 evaluates the planning framework with respect to the range of problems on which it can be successfully used. Comparisons are made with other systems. Strengths and weaknesses are considered.

Chapter 10 is the conclusion. The findings are summarised and put in perspective. Opportunities for further development are considered.

Chapter 2

Background: Planning

An intelligent agent should be able to reason about its own actions and the changes that these will bring about in the world. In Artificial Intelligence, planning usually involves the formation of a sequence of primitive actions in order to change the world to a required goal state.

This chapter reviews the background of planning in Artificial Intelligence, looking at both representation and search algorithms.

We will see that there is generally a trade-off between the expressiveness of the representation used for the planning problem, and the tractability of search using that representation.

At one extreme lies situation calculus (Section 2.1), which is very expressive, but for which the intractability of proof search makes it impractical for planning. At the other extreme is the STRIPS representation, which is not very expressive, but has allowed practical search algorithms to be developed. We look at the most successful search algorithms for the STRIPS representations.

We then look at some of the intermediate stages — planners that extend the STRIPS representation back in the direction of more expressiveness, and correspondingly adapt the STRIPS search algorithm. We focus on the treatment of actions with uncertain outcomes, as this is one of the aspects which will be later treated in the linear logic framework.

2.1 Situation calculus

Situation calculus is a regime for representing and reasoning about changes of state using first order logic. This was first introduced by [McCarthy & Hayes 69]. We present its use for reasoning only about state here, but the original work suggested its use for reasoning about belief, knowledge, etc. Although this work dates from a long time ago, it was influential in setting up background assumptions that underpin much planning work since.

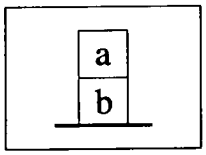
We can describe a static world by assuming it can be described only in terms of:

Objects Unique objects in the world can be represented by their names, e.g. we might have building blocks called *a*,*b*, and another object called *table*, etc.

Relations We need to also represent the relations between objects using relations, e.g. $on(a, b)$, $clear(a)$.

We could describe a static situation using only a conjunction of such statements, e.g.,

$on(a, b) \wedge on(b, table) \wedge clear(a)$

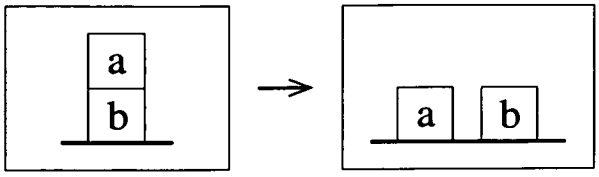


This is fine for describing a single, unchanging state, but we want to describe changes of state, so we need to consider also that there are states referred to by an extra argument in each relation.

So we could describe two situations as follows:

$$clear(a, s_0) \wedge \\ on(a, b, s_0) \wedge \\ on(b, table, s_0)$$

$$clear(a, s_1) \wedge \\ clear(b, s_1) \wedge \\ on(a, table, s_1) \wedge \\ on(b, table, s_1)$$



We also want to be able to talk about the relationships between the situations. For instance, s_1 is the state that results from s_0 by the execution of a command $move(a, table)$. We can express this in situation calculus by the use of the *result* function, which maps actions and situations to new situations. In this example, we would have $s_1 = result(move(a, table), s_0)$.

So now we have representation which is capable of describing situations and the relationships between situations. We can also describe sequences of actions (i.e. plans), as they simply correspond to nested applications of the result function.

In order to be able to reason about correct plans using this representation, we will also need a general way to describe what are the effects of an action when applied in any state.

We need to express:

- restrictions on when it is possible to perform actions,
- a specification saying how to derive the description of the state after the change from the description of the state before the change.

For instance, suppose we wish to describe the action of placing one block on another, we could write the description as:

$$\forall x, y, s \; clear(x, s) \wedge clear(y, s) \supset \begin{array}{l} on(x, y, result(puton(x, y, s))) \wedge \\ clear(x, result(puton(x, y, s))) \end{array}$$

On the left of the implication, we can give *preconditions* to the execution of an action. These are the conditions which must be satisfied for the action to be applicable. On the right, we give properties of the new situation (the *effects*).

Unfortunately, it is not sufficient to only describe those relations which change in the

new situation, it is also necessary to provide *frame axioms*, which allow the unchanged relations to also be deduced for the new state. For instance, if we know that a block is red, we need to state that it will be still be red after it is moved.

$$\forall x, y, s \; colour(x, red, s) \supset colour(x, red, result(puton(x, y)))$$

The fact that we also need to explicitly specify everything that persists as the result of an action is a manifestation of the *frame problem*. Any representation for a planning problem needs to find some way of coping with the frame problem. If we use situation calculus with frame axioms like the example above, then even small problems require a prohibitively large number of frame axioms, and potentially a large amount of inference is also required. As we will see Chapter 5, linear logic provides a simple way addressing the frame problem.

2.1.1 Search in the situation calculus

Having given some idea of how the situation calculus may be used to represent and reason about actions, we must now consider how it could be used to automatically form plans of action. This requires performing a search. In the simplest case, one could imagine a planner which searched through a space of possible states derived by non-deterministically selecting applicable actions, then deriving the properties of the new state from the axioms defining the actions.

Cordell Green's QA3 system [Green 69] was an ambitious and noteworthy early planning system based on the situation calculus. This was a resolution theorem prover which anticipated much of the more recent work on deductive planning. It was capable of forming plans with conditionals, loops, etc.

This system was not very practical because it had poor of control of inference, and no particularly good solution of the frame problem. A more interesting flaw was that references to explicit states could appear in the final plan, and this sometimes meant that the plan demanded tests on hypothetical states. This problem was subsequently taken on by Manna and Waldinger (Section 3.2).

2.2 STRIPS representation

The situation calculus is a very expressive representation language, but a consequence of this expressiveness is that making inference in this logic is computationally expensive. To counter this problem, The STRIPS planner introduced a more restrictive representation for plan operators, which allows search to be carried out more efficiently [Fikes & Nilsson 71].

The following assumptions are made in the treatment of planning work described in this section.

The STRIPS assumption: Nothing changes except by application of an action.

Atomic time: Execution of an action is indivisible and uninterruptable, so we need not consider the state of the world while execution is proceeding.

Deterministic effects: The effect of executing any action is a deterministic function of the action and the state of the world when the action executed.

A STRIPS operator is defined by a list of preconditions, a list of statements that are added by the action, and a list of literals that are deleted by the action.

```
operator puton(X,Y)

preconditions: clear(X)
               clear(Y)
               on(X,Z)

addlist:       on(X,Y)
               clear(Z)

deletelist:    on(X,Z)
               clear(Y)
```

The STRIPS operators describe purely syntactic manipulations on the world description. The STRIPS world description could be seen as a set of (implicitly conjoined) formulas. The operator definitions describe what formulas should be added and deleted from the world description when the actions are carried out.

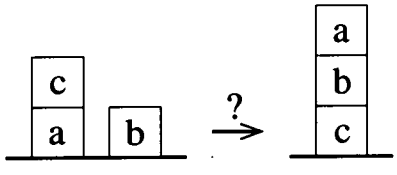
The STRIPS planner itself worked by *goal regression.* Starting with the goal conditions, an operator is selected that achieves an outstanding goal condition.

It is usually more efficient to start the planning process from the goal, and to try to connect it with the initial state, since the the initial state may contain information not relevant to the problem, whereas the goal state is typically only a partial specification. This means that only the important information about the goal state is used in the search process.

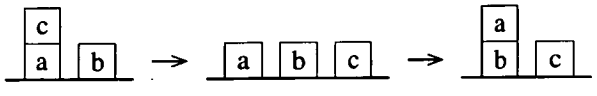
In the original STRIPS planner, the assumption was made that all goals were independent, i.e. that one goal could be completely solved before attempting to solve the next. This lead to incompleteness in the plan search space — some solutions could simply not be found by STRIPS, since it could not interleave steps required to solve different goals at the top level. This is sometimes called the *linearity assumption.*

The classic example of this problem is known as the Sussman anomaly problem [Sussman 73]. The problem can be solved with a three-step plan, but STRIPS could only find longer, redundant plans.

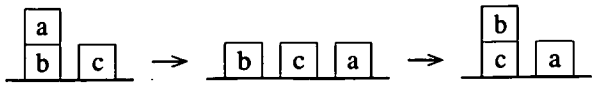
The goal state consists of only two statements $on(a, b)$ and $on(b, c)$.



If STRIPS attempts to satisfy the goal $on(a, b)$ first, it succeeds in finding a plan involving the following sequence of state transitions:



This makes it impossible to achieve $on(b, c)$ without undoing $on(a, b)$. Subsequent steps generated by STRIPS would be:



Similarly, if STRIPS attempted to satisfy the $on(b, c)$ first, it still would find a similar problem. To get around this problem, it is necessary to use a quite different approach.

2.3 Plan space planners

The existence of this sort of goal-interaction problem motivated another shift in the approach to planning problems.

Instead of searching through a space of world states, a search is performed through a space of partially-specified plans. Partially specified plans may contain unsatisfied goals, unbound variables, and only partial descriptions of step ordering. This allows a least-commitment approach, in which arbitrary choices in search are deferred until more information is available.

A partially-specified plan consists of:

- a set S of plan steps
- a set O of ordering constraints on elements of S
- a set B of binding constraints on variables
- a set C of causal links. Each causal link names a relation, a producer step (which adds the statement as effect) and a consumer step (which requires the statement as a precondition).

The search algorithms for this representation can be considered as a selection of plan refinement operations, which attempt to fill in more of the plan to resolve possible conflicts in the current partial plan.

Typically, plan refinement operations are;

- Select a subgoal.
- Choose an operator.
- Resolve threats.

- Threats are steps that might delete (*clobber*) a protected condition. This means that the present step ordering could allow the threatening step to delete some condition which is required as a precondition of another step.
- Threats can be resolved by one one of:

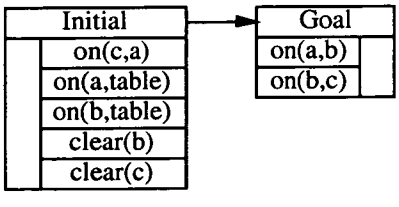
  promotion — add an ordering constraint to ensure that the clobbering action occurs after the consumer of the threatened link.

  demotion — add an ordering constraint to ensure that the clobbering action occurs before the producer of the threatened link.

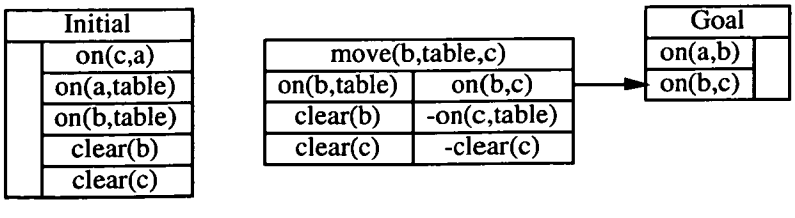
  Posting equality/inequality constraints — in the presence of uninstantiated variables, it is possible to add constraints on the binding of variables to ensure that the relation clobbered is not the same as the one protected.

2.3.1 Example — solution of the Sussman anomaly

Here we present a walk-through of such a planner solving the Sussman anomaly problem. This account is based on Dan Weld's tutorial [Weld 94]. The planning process starts off from dummy initial and goal nodes. The start node has effects and no preconditions, and the goal has preconditions and no effects. Here the plan steps are represented by boxes with the name of the step at the top, the preconditions in the left column, and the effects in right column.

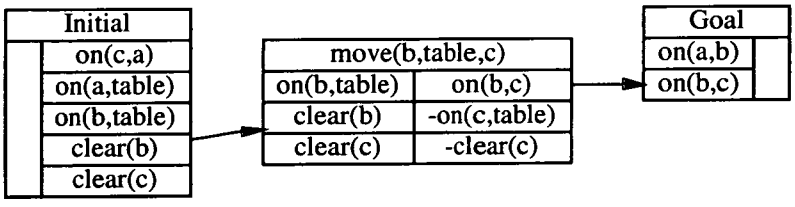


Add a step $move(b, table, c)$ to satisfy goal $on(b, c)$

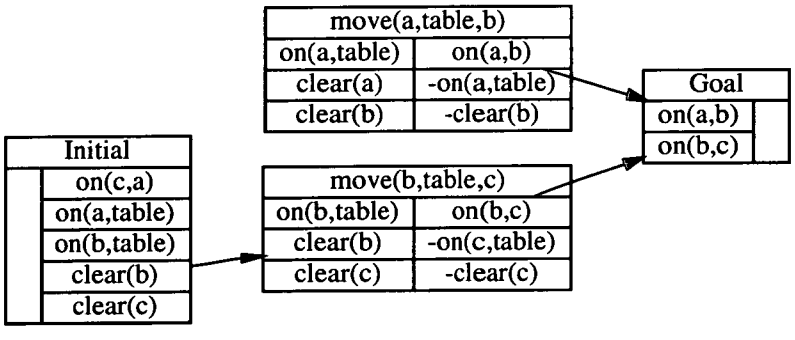


Causal links are shown by blue lines, and step ordering is shown by black lines. Step ordering is omitted where it is implied by causal links. The '-' before a statement indicates the effect is a delete effect.

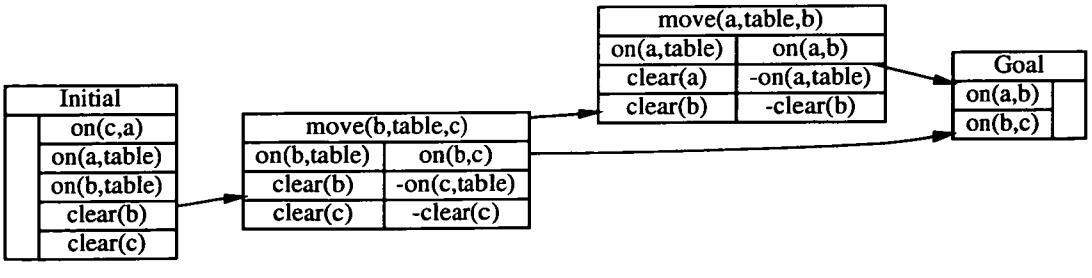
We can add a further causal link to satisfy the goal $clear(b)$. We happen to choose not to satisfy $clear(c)$ just yet.



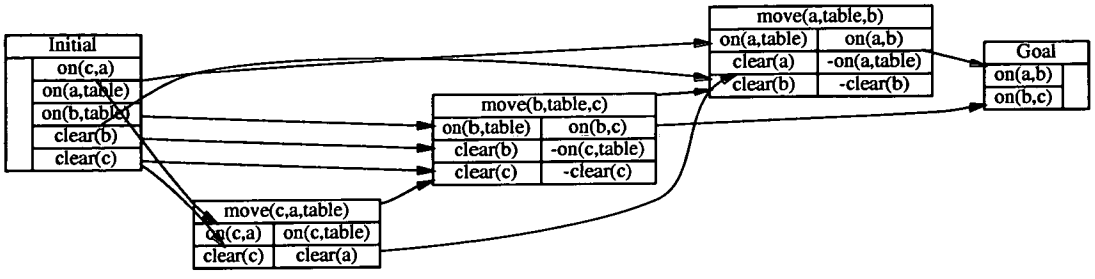
Now suppose we next choose to satisfy the goal $on(a, b)$. We could do this by adding a step $move(a, table, b)$.



But now we find that the link $clear(b)$ is threatened, since the new step could delete this relation before the execution of $move(b, table, c)$ and which requires $clear(b)$ as a precondition. Now we have a choice of how to resolve this threat. It turns out that it can be removed by promoting the threatening action, which means adding an ordering constraint (shown in black).



The final plan requires a further step and a further threat resolution, and is shown below.



2.3.2 Development of causal link planners

Causal link planning was first introduced by Sacerdoti [Sacerdoti 75], then given a more thorough analysis by Tate [Tate 77].

Chapman gave a more formal treatment, and proved soundness and completeness of a causal link planner, TWEAK [Chapman 87]. Chapman introduced the notion of the modal truth criterion, which is a test for determining the consistency of partially ordered plans, and which can itself be executed as a planner.

Pednault [Pednault 87] went back to situation calculus as a basis for his Action Description Language (ADL), which is between STRIPS and situation calculus in expressiveness, and provides a STRIPS-like representation.

A subset of ADL was implemented in the UCPOP [Penberthy & Weld 92] planner, a provably sound and complete refinement planner which allowed the extensions of universal quantification in effects and goals, and actions which have effects conditional on the context in which they are used.

2.4 Graphplan

In recent years, an alternative approach to planning emerged, marking a departure from the established tradition of causal link planning. Graphplan, [Blum & Furst 95], despite being a relatively simple idea, gave considerable performance improvements over the planners previously available. Graphplan makes use of a datastructure called

a *planning graph* for representing plans and world states, which allows constraints on the plan to be derived and exploited.

2.4.1 Representation

The philosophy is that the plan/states are not fully described, but we have a projection which gives us necessary conditions for finding a plan.

In Graphplan, the planning graph consists of alternating levels of *proposition nodes* and *action nodes.* The planning graph has three types of edge:

- precondition edges
- add-edges
- delete-edges

The planner works in two phases:

- In the first phase, the planning graph is constructed by projecting forwards from the initial state. The planning graph allows certain constraints to be recorded for later exploitation
- In the second phase the end of the planning graph is matched with the goal state, and the search is carried out by regression on the planning graph.

2.4.2 Algorithm

The initial phase of graph expansion overlays all actions and propositions which could be present in the graph at each level, together with constraints in the form of *exclusion relations*, recording pairs of nodes which cannot be simultaneously present.

There two types of exclusion relations between actions:

Interference If either of the actions deletes a precondition or add-effect of the other.

Competing needs If there is a precondition of a and a precondition of b which are marked as mutually exclusive at the previous level.

Additionally, exclusion relations can be added between two propositions if all ways of achieving one exclude all ways of achieving the other.

The graph is expanded level-by-level until a state is reached wherein the goal conditions are not excluded, and at this point it is worth attempting the solution extraction phase.

For each goal at time t, Graphplan attempts to find a set of actions compatible with exclusion constraints at time $t-1$ which achieve these goals. The preconditions for these actions then form a goal set for time $t-1$. By recursing backwards using this process, we have a procedure for finding a plan with t steps.

If no plan of that length can be found then the extraction phase fails and planning may continue by expanding the planning graph by a further level. Using this process, we can be assured that we always find the shortest plan (in Graphplan representation) first. A further criterion concerning the fixpoint of the planning graph allows detection and termination in cases where no solution can exist.

Note that the Graphplan representation allows for any number of non-interfering actions to be carried out in parallel in each time step.

2.5 Conformant and contingent planning

This section reviews work in the planning literature on extensions of STRIPS planning to deal with conformant and contingent planning, which are ways of handling actions with uncertain effects. This section is included because this is an area that we consider in our linear logic approach (Section 8.7).

We adopt the term *conformant* from [Smith & Weld 98], where it is defined to mean planning in the presence of uncertainty, but without the ability to resolve the uncertainty when the plan is executed — so a single plan (without conditional branches) must be formed that will work under all possible outcomes of the uncertainty. This type of problem is also called *fail-safe* planning in [Pryor & Collins 96]. Examples in this class include Moore's Bomb in the toilet problem [McDermott 87] and the matching socks problem [Bibel 86].

Contingent plans aim to take into account the idea that more information will be

available to any executing agent when the plan is being executed than when the plan is being formed. Contingent planners are able to gather information and choose between conditional branches of the plan at run time.

Both types of plan can cope with uncertainty about the initial state of the world (e.g. whether it is raining), and also with uncertainty in the effects of actions (e.g. the result of tossing a coin).

For our purposes, the most important point to consider is the way that uncertainty, sensing and decisions are modelled, rather than the problems of integrating this into partial order planning.

2.5.1 Warplan-C

The first planner that was capable of planning for contingencies was Warplan-C [Warren 76]. This system forms totally ordered plans with conditional branches. It is assumed that all information is available to the planner at runtime, and plans simply branch at the point at which the distinction needs to be made.

Here, uncertain effects are represented by conditional actions with two possible outcomes P or $\neg P$. It is assumed that the executing agent will know the result of the action as soon as it is carried out, so explicit sensing action is not required.

The strategy for handling uncertainty is first to assume a particular outcome, then later to add branches to the plan to account for other possibilities.

2.5.2 CNLP

CNLP [Peot & Smith 92] used a partial-order planning framework. Here the source of uncertainty is separated from the conditional action, but it is assumed that the consequences of any action are immediately known — i.e. there is no distinction between sensing and decision-making. This type of action has different consequences, depending on the uncertainty. These different consequences can give rise directly to different contingent sub-plans.

2.5.3 Cassandra

[Pryor 95], [Pryor & Collins 96] emphasised the need to separate the sensing actions and decision-making actions. Uncertainties are represented in a totally distinct way from ordinary literals describing the state of the world. Steps are labelled by the contingencies under which they apply.

The steps which only apply under certain contingencies must be preceded by a decision step which is able to distinguish those conditions. The decision step allows the agent executing the plan to deduce the outcome of uncertainties which cannot be perceived directly. The decision step comprises a set of rules with perceivable conditions as antecedents and contingencies (i.e. possible outcomes of the uncertainty) as conclusions.

During goal-directed planning, the antecedents of the decision rules give rise to knowledge goals. This requires explicit modelling of knowledge via *know-if* propositions. These know-if conditions are the effects of sensing actions, whose effects are dependent on the value of some uncertainty which is not directly perceivable.

2.5.4 Graphplan derivatives

Graphplan has been extended to model conformant plans in the SGP planner [Smith & Weld 98] and contingent plans in CGP [Weld *et al* 98]. Both these systems rely on modelling of possible worlds in a planning graph. Each time an uncertain effect is introduced, the planning graph is split into a new branch for each outcome. Planning then handles constraints both within and between possible worlds.

The SGP system, like Cassandra, adopts a representation for reasoning about knowledge. In this case, it takes the form of exclusion relations between worlds. For example $K\neg u : v$ represents a proposition that if an agent is in world w_v then it will know that it is not in world w_u.

2.6 Discussion

This chapter reviewed background material on planning representations and algorithms. We considered the very general situation calculus representation, in which

inference is awkward to control and frame axioms are required for action descriptions.

We then considered the STRIPS representation, which pragmatically uses a less expressive language, but allows for efficient search algorithms. We then described some extensions to STRIPS planning for handling actions with uncertain effects.

In chapters 4 and 5, we will show that the linear logic framework, like situation calculus, provides a clear formal description of action. Unlike situation calculus, we can neatly express state change without the use of frame axioms, and conveniently extract plans directly from proofs. We will see that linear logic approach represents a different compromise between expressiveness and tractability than those discussed in the present chapter.

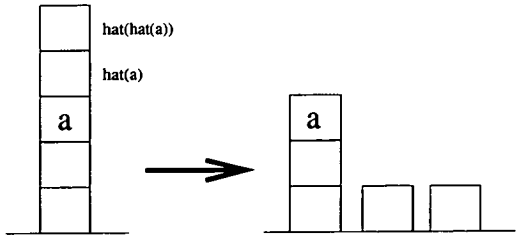
Chapter 3

Background: Formation of Recursive Plans

3.1 Introduction

By including control structures in plans, a planner can produce a single general plan which will deal with a class of situations. For example, in the blocks world, recursive plans can solve problems involving towers of unspecified size, e.g.

- Inverting a tower.
- Putting one tower on top of another.
- Inserting a block at the bottom of a tower.
- Clearing a tower from above a certain block — "How to clear a block".



This chapter reviews three approaches to the formation of recursive plans.

- by deductive planning with situation calculus (Manna and Waldinger)
- by planning with extended STRIPS operators (Ghassem-Sani and Steel)

- by deductive planning in modal logic (Stephan and Biundo)

These differ in:

- the choice of representation for problems and plans, and consequently the class of problems which can be represented and solved,
- the degree of automation of the proof process.

3.2 Manna and Waldinger

[Manna & Waldinger 87] viewed planning as being similar to the synthesis of imperative programs. This is quite similar to the approach used by Green in QA3 system [Green 69].

One particular problem with Green's approach is that, as a consequence of using situation calculus, the plans formed contained explicit references to states. This can have some unwanted consequences, since it allows the construction of some forms of plan which it is not possible to execute.

Manna and Waldinger give the example of the monkey, bomb, bananas problem. This is a problem in which a monkey must obtain the banana, given the fact that it is in one of two identical closed boxes, a and b. Whichever box does not contain the bananas contains a bomb, which will explode if approached.

There should be no solution to this problem, but the QA3 system allows the following plan:

```
getbanana(s0) <=
  if Hasbanana(goto'(s0,a))
     then goto'(s0,a)
     else goto'(s0,b)
```

This plan involves the monkey testing whether it gets the banana in the hypothetical state in which he chooses box a. This plan cannot actually be executed by the monkey.

Manna and Waldinger restrict their deductive system to producing plans which are executable, by restricting the use of case-splits so as to avoid the problem.

In order to construct usable plans, the plan should avoid explicit references to states, and this is a motivation behind their modification of the situation calculus. They propose *plan theory*, a form of situation calculus including *fluents*, which have no reference to states, but are given a state dependent interpretation by attaching a state variable with a *linkage operator*. Situational expressions and fluent expressions both exist in plan theory and they are related by the linkage operators. A fluent expression e in state s is written as

$$s : e,\ s :: e,\ \text{or}\ s\,;e$$

depending on whether e represents an object, a truth value, or a state respectively.

Static expressions refer to a state, e.g.
$hat'(s,b)$, $Clear(s,b)$, $put'(s,b,c)$. where s denotes a state and b,c denote blocks.
Fluent expressions designate elements w.r.t. an implicit state, e.g.
$hat(d)$, $clear(d)$, $put(d,e)$
They are related by linkage operators, e.g. in state w,

$$w{:}hat(u) \equiv hat'(w,\ w{:}u)$$

$$w{::}clear(u) \equiv Clear(w,\ w{:}u)$$

$$w;put(u,v) \equiv put'(w,\ w{:}u,\ w{:}v)$$

To construct a plan for achieving a condition $Q[s_0, a, z]$, where s_0 is the initial situation, a an input object, and z the final state, Manna and Waldinger prove the theorem:

$$\forall s_0\, \forall a\, \exists z_1\, Q[s_0, a, s_0; z_1]$$

z_1 is instantiated to a plan fluent term during the proof.

The solution plan for the "how to clear a block" problem is represented as:

```
makeclear(a) <=
  if clear(a)
  then skip
```

```
else makeclear(hat(a));
     put(hat(a)),table)
```

The deductive tableau method is used to form proofs. This combines non-clausal resolution, well-founded induction, and conditional term rewriting. An equational unification algorithm is used which has built-in equivalences between static and dynamic expressions.

In the tableau, each row may have either an assertion or goal, as well as a plan expression. A tableau is *true* whenever the following holds: *if all instances of each of the assertions are true, then some instance of at least one of the goals is true.*

The deduction rules add new rows to the tableau, such that if the new tableau is valid, then so is the original one. The tableau is proved valid when *true* appears as a goal or *false* appears as an assertion.

The proof and the plan are constructed side-by-side, with the restriction that static terms cannot appear inside plan terms.

Manna and Waldinger show how this approach can be used for the formation of conditional expressions and recursive plans. An example of a recursive plan is a general one which clears the top of particular block, no matter how many other blocks are stacked on top of it.

Some unresolved issues with their approach are:

- A proposal of how the frame problem should be handled is broadly outlined, but has not been fully developed.
- Their treatment of induction suggests that it has not been very successful in practice. The main areas which cause problems are the choice of a well-founded relation, and the difficulties of producing strengthened induction schemes in order to find an alternative proof choice.
- The axioms relating situational and fluent expressions are built into an equational unification algorithm. The redundancy caused by using two equivalent forms means that the unifier yields multiple solutions — i.e. there is no unique most

general unifier.

- They do not address the issue of search control. The scheme was not implemented as a fully automatic system. A related deductive tableau systems was implemented as an interactive system, in which a user guides the proof.

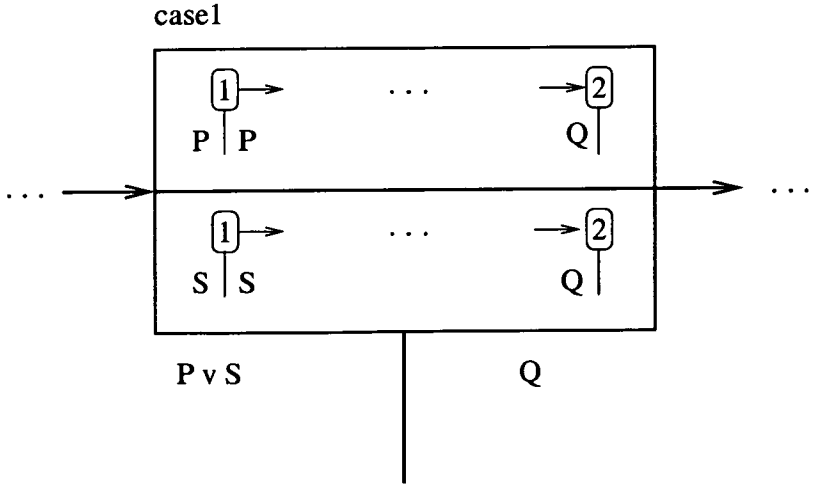
3.3 Ghassem-Sani and Steel's RNP

Ghassem-Sani and Steel's Recursive Nonlinear Planner (RNP) [Ghassem-Sani & Steel 91, Ghassem-Sani 92] is an extension of a classical partial order planner to generate recursive plans. This is a less general method than Manna and Waldinger's, but gives improved control over solution search, to the extent that a fully automatic system was implemented.

3.3.1 RNP — Plan representation

Plan representations using STRIPS operators cannot express recursive plans. Steel and Ghassem-Sani extend the plan representation by including three new kinds of plan step:

CASE nodes — Two (or more) subplans are contained inside one node. Each subplan has the same postconditions, but different preconditions. The preconditions for the case node include the disjunction of the preconditions of the components subplans.



PROC nodes — These are equivalent to procedure definitions in programming languages. They are identical to case nodes, but are invoked by call nodes.

CALL nodes — These are equivalent to procedure calls in programming languages, naming a PROC node to invoke.

3.3.2 RNP — Induction principle

To form a recursive plan, a proof by induction is used. The following form of Boyer and Moore's induction principle is used. The following form accounts for forming recursive plans with a single base and step case:

If

(a) P and Q are predicates;

(b) $<$ denotes a well-founded relation

(c) f is a function

(d) x is a variable

(e) $P(x) \rightarrow f(x) < x$ is theorem

then in order to prove that $Q(y)$ is theorem, it is sufficient to prove:

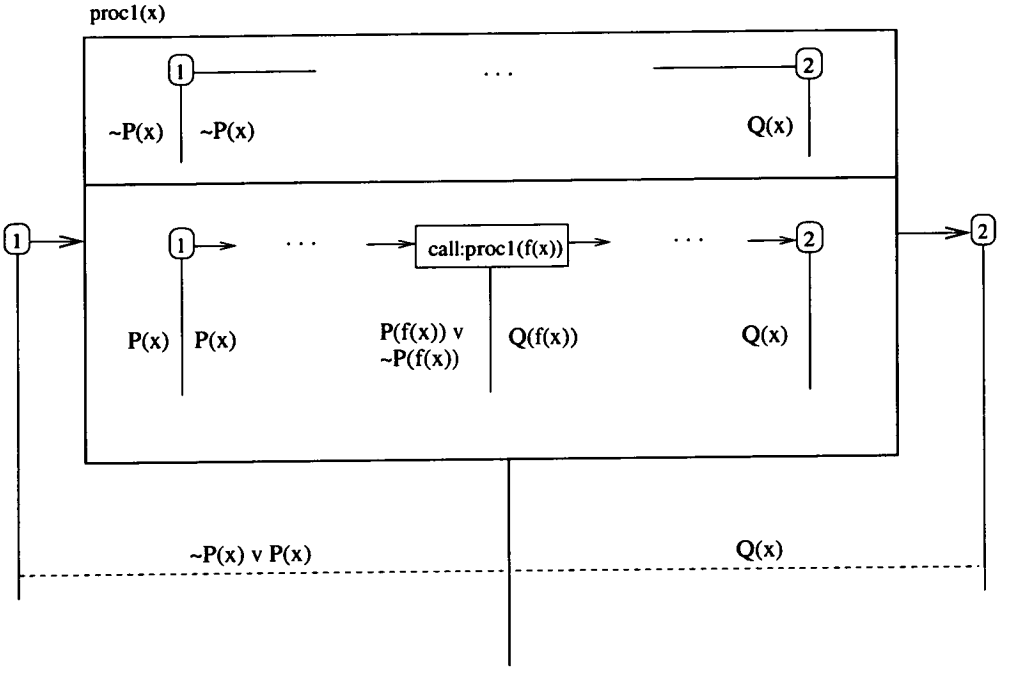
(1) $\neg P(x) \rightarrow Q(x)$ Base case

(2) $(P(x) \wedge Q(f(x))) \rightarrow Q(x)$ Step case

In forming recursive plans, the planner uses a database of theorems of the form of (e), defining well-founded relations. $P(x)$ here behaves as a condition which determines whether the base or step case applies.

The figure below gives the corresponding plan structure.

RNP — Form of recursive plan

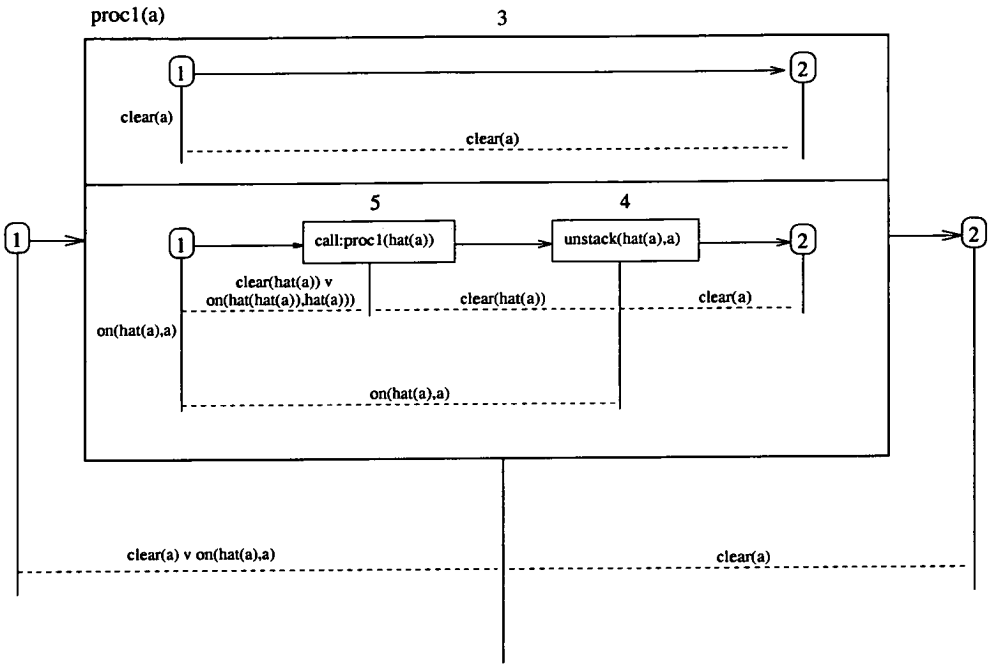


RNP — example plan

The plan formation process is an extension of a conventional partial-order causal-link planner. Recursive constructs are introduced when the planner detects a similarity between between a subgoal and a top level goal, e.g. a subgoal of the form $Q(f(x))$ and a goal of the form $Q(x)$. In these circumstances the planner can then check if the two forms are ordered by a known well-founded relation, such that the subgoal is a reduced form of the top level goal.

If this test succeeds, then RNP tries to solve the problem by recursion. The next step is to perform a case analysis, e.g. using the predicate P from (e) above to separate base and step cases. RNP then tries to find or create a new PROC node with subplans for the $P(x)$ and $P(x)$ cases.

The plan generated for the "how to clear a block" problem is given below.



A form of generalisation is also possible, where a constant may be replaced by a variable in order to make the induction work.

The detection of threats in RNP's plans is complicated due to the presence of recursive constructs. No equivalent of Chapman's modal truth criterion 2.3.2 was developed to demonstrate the soundness of the planner.

3.4 Stephan and Biundo

A group at DFKI in Saarbrücken has carried out recent work on deductive planning [Stephan & Biundo 93], [Stephan & Biundo 95], [Dengler 96], [Koehler 96].

Their approach is characterised by the use of programming logics - particularly an interval-based temporal logic called *logical language for planning* (LLP). The intervals are sequences of states, and the logic has modal operators:

$\bigcirc A$ — next A

$\Diamond A$ — sometimes A

$\Box A$ — always A

$X ; Y$ — X followed by Y

The operator ; (*chop*) which expresses the sequential composition of formulae. This allows plans and actions to be expressed as programming constructs in the logic, which have semantics defined on intervals.

A planning problem is formulated as a proof of the following formula:

$$pre \wedge Plan \rightarrow \Diamond goal$$

where *pre* represents initial conditions and *Plan* is a variable which is instantiated during the proof.

The changes brought about by action are expressed as operations on local variables, which may change their values between states, and this is extended to handle a STRIPS-like manipulation of logical relations.

An example definition of an action is given as:

```
rec unstack(x,y).
   if on(x,y) & clear(x)
      then add- table(x);
           add- clear(y);
           delete- on(x,y)
      else abort fi.
```

This syntax translates directly into formulas in the temporal logic. From such a description, axioms describing action effects and frame conditions can be derived automatically.

This example is taken from [Stephan & Biundo 93], which uses first-order dynamic logic. This paper shows how frame conditions can be efficiently handled in the logic by treating them as invariants of the actions or action sequences. The frame problem can be treated efficiently by generating frame assertions non-deductively. This is part of a domain modelling process in which action definitions are analysed to produce temporal logic conditions in a form to be used during planning.

3.4.1 Recursive planning in LLP

The LLP language is capable of expressing recursive constructs, but requires user interaction to do this. Abstract recursive plans are formed and proved correct as part of an interactive domain modelling process. After this domain modelling, the planner can work fully automatically. To automatically solve a concrete planning problem, the abstract recursive plans are unwound to simple action sequences. This is performed by a tactical theorem prover, KIV.

[Stephan & Biundo 95] develops the idea of planning as refinement, by which a non-executable specification is refined step-by-step into an executable plan. Stress is laid on the idea that the specification and the plan are expressed in the same language.

[Dengler 96] also considers planning as a process of refinement and shows how LLP can be used to perform non-linear planning. [Koehler 96] uses the framework as a basis for a system of plan reuse — i.e. stored plans are retrieved and adapted to solve a new planning problem. Koehler suggests that a powerful language of plan specification and a system for manipulating plans which guarantee to meet the specification are needed for a plan reuse system (*second principles planner*).

3.5 Inductive theorem provers

3.5.1 Introduction

In this section we consider proof techniques which are specialised for solving the problems present when constructing inductive proofs. These proofs are crucial to the formation of recursive plans in deductive planning, since recursive constructs in the plan correspond to the use of induction in the proof. Some particular problems in forming induction proofs are:

- The choice of the specific induction scheme to use.
- The choice of the induction variable.
- The control of rewriting steps in the induction step case. The special heuristic technique of rippling has been applied successfully to greatly reduce the amount of search in rewriting. The technique makes use of the similarity between the induction hypothesis in induction step cases. The notion of rewriting here includes certain deductive steps which yield to being treated as rewrite rules.
- The need for generalisation. Often the original theorem may be impossible to prove inductively, but a more general version of the theorem can be proved. Techniques exist for generating these generalisations automatically.

3.5.2 Boyer and Moore

The Boyer-Moore theorem prover [Boyer & Moore 79] was an early success at automatically generating inductive proofs. They use a first-order logic with a LISP-like syntax and no explicit quantifiers. Destructor syntax is used for inductive problems.

The prover makes use of a repertoire of separate theorem-proving components and heuristics.

Symbolic evaluation — simplification of an expression by application of rewrite rules.

Induction — The selection and application of an induction rule.

Fertilization — The use of the induction hypothesis to prove the induction conclusion.

Generalisation — Generation of generalised theorem, to allow successful induction.

Their system incorporated heuristics for dealing with the problems listed in 3.5.1 above. For example, the use of induction was attempted only after a failed attempt at a proof using only symbolic evaluation. The choice of induction variable was motivated by the manner in which the symbolic evaluation proof failed.

A large contribution towards the success of the Boyer-Moore prover is reasoning about about the applicability of these separate components and about the relationships between them.

3.5.3 Proof planning

A rational reconstruction of the Boyer-Moore heuristics led to the concepts of *proof planning* [Bundy 88]. In this section we give an overview of proof planning, with particular reference to its use in controlling inference in inductive proofs using the technique of *rippling*.

Proof plans are meta-level plans for guiding the object-level inference in constructing a proof.

Common strategies for applying object-level proof steps, such as symbolic evaluation, are expressed as tactics. Tactics are programs for constructing part of the proof at the object-level.

In order to reason about the application of tactics, meta-level specifications of the object-level tactics are used. These specifications are known as *methods*.

A method is described by:

Input Meta-level sequent to which the tactic applies.

Preconditions Conditions which must hold for the method to be applied.

Effects Conditions which will hold after the method has been applied.

Outputs Subgoals generated (list of meta-level sequents).

Tactic The name of the tactic to be applied.

In proof planning, the meta-level plan is constructed using a planning algorithm to search in a space of method applications. This meta-level search space is typically much smaller than the corresponding object-level search space.

3.5.4 Rippling

We will briefly discuss the technique of rippling. Although we do not use this in our work, we will refer to it in our discussion of techniques for generalisation (Section 3.5.6) and for dynamic creation of induction rules (Section 10.3.6). Rippling is a heuristic technique for controlling the application of rewrite rules in inductive proofs. In the *Clam* system, it has been implemented as a method within the proof planning framework.

In step cases of inductive proofs, there is typically a need to prove a theorem of the form:

$$P(x) \vdash P(f(x))$$

where $f(x)$ is a constructor function - e.g. for natural numbers it might be the successor function $s(x)$. Rippling exploits the fact that the presence of the constructor is the only difference between the induction hypothesis (on the left) and the induction conclusion (on the right). This difference is called a *wavefront.* Annotations are added to identify the presence of the wavefronts.

The proof involves applying a series of rewrite rules such that the wavefront can be moved out of the way, allowing a match between conclusion and hypothesis. Annotations allow the wavefront to be treated as separate entity, which moves on a fixed background formula, the *skeleton.* In the example above, $P(x)$ is the skeleton.

3.5.5 Choice of induction variable and rule

Rippling can also be used to inform the choice of induction variable and induction rule, so that the choice is made on the basis of what is most likely to work. This process is

known as *ripple analysis*

3.5.6 Generalisation

Often a generalisation step is necessary to successfully complete an inductive proof. As pointed out in [Manna & Waldinger 87], generalisation is also important in forming recursive plans. Techniques have been developed for automatically determining the generalisation. Here we sketch the technique used in [Hesketh *et al* 92].

This generalisation is based on middle-out reasoning. The precise form of the generalised theorem is not determined until midway through the proof. Its value is instantiated so as to make the proof work correctly. After creating the generalised theorem, and the proof of the generalised theorem, it is also necessary to construct a proof that the generalised form entails the original theorem.

We give an example from [Hesketh *et al* 92], in which the problem is to transform functions into equivalent tail-recursive functions. For example, consider a function for reversing a list, rev_n, defined by the following rules:

$$\begin{aligned} rev_n(nil) &= nil \\ rev_n(h :: t) &= append(rev_n(t), h :: nil) \end{aligned}$$

We start with a specification using the non-tail-recursive version. This will be used as a specification for the tail-recursive version.

$$\vdash \forall x.\exists z.\ z = rev_n(x)$$

Tail recursive algorithms can be extracted from proofs of a certain shape. To create a proof from which we can extract the tail recursive version, we need to generalise this proof to a version that involves an accumulator variable:

$$\vdash \forall x.\forall a.\exists z.\ z = append(rev_n(x), a)$$

The generalisation problem here is to perform this step automatically. How do we know that *append* is the correct function to use? The approach is to use a higher-order meta-variable as a placeholder for the unknown function.

$$\vdash \forall x.\forall a.\exists z.\ z = G(rev_n(x), a)$$

During the inductive proof of this theorem, we then allow G to be instantiated to anything that allows the rippling steps to pass through successfully. The appearance of *append* then comes from the application of the rewrite rules defining rev_n. For the analogous generalisation in a planning problem, this would correspond to a reformulation of the plan specification (initial and goal states).

3.5.7 Summary

Control of reasoning in inductive proofs raises some specific problems — i.e. choice of induction rule, choice of induction variable, control of rewriting, and generalisation.

These can often be tackled by a combination of strategies. Proof planning provides a coherent framework to handle high-level plans which talk about the application of such strategies. The specific technique of rippling is an important component in proof plans for induction, as it allows similarity between induction hypothesis and conclusion to be exploited to reduce search.

3.6 Conclusion

This chapter reviewed three different approaches to recursive planning problems.

Manna and Waldinger's deductive synthesis approach, based on a modified situation calculus, has good coverage. However, there is no known strategy for controlling inference so this has not been implemented as an automatic system.

Ghassem-Sani and Steel's RNP uses a STRIPS-like representation, and is more restricted in expressiveness, but has been implemented as a fully automatic planner.

Stephan and Biundo's deductive planning approach pushes the difficult theorem-proving tasks into the phase of planning domain modelling, which is done interactively. The result is a domain-specific planner which works fully automatically in a tactical theorem prover.

We also considered the problems which arise in inductive proving: choice of induction rule, choice of induction variable, control of rewriting, and generalisation. We described approaches to the problems using proof planning and rippling.

Proof planning is not used to guide theorem proving in our system. For larger or more complex problems than those considered, proof planning would provide a powerful and flexible framework for further development.

Chapter 4

Linear Logic

4.1 Introduction

In this chapter, we give an introduction to a fragment of linear logic and demonstrate its use in simple planning problems. See [Girard 95] for a full exposition of linear logic, and [Masseron *et al* 93] for a thorough treatment of its use in conjunctive planning problems.

4.2 Basics of linear logic

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.

For simple planning problems we will need the *multiplicative conjunction* connective, written $\otimes$. The formula $A \otimes B$ means that resources A and B are simultaneously present. The rules for $\otimes$ are given in the next section.

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 are simultaneously present. This is the form of conjunction which we use in STRIPS-like planning problems.

The second form *additive conjunction* means that both resources are 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, we regard it as a conjunction, since it would be equivalent to the conventional conjunction if weakening and contraction were allowed, i.e. the discarding and copying of resources.

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.

4.2.1 Exponentials

An important feature of linear logic the exponential operator !. This provides a means to mark out formulas to which weakening and contraction rules can be applied - so that they can be used as many or as few times as necessary in a proof. This is particularly important because it allows an embedding of intuitionistic logic into intuitionistic linear logic, e.g.

$$!a \multimap b$$

is equivalent to:

$$a \rightarrow b$$

However, we shall not use exponentials for the rest of this chapter.

4.3 Linear logic for planning

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 rules $Ax, cut, l\otimes, r\otimes$.

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

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

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 subplans 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 $hold(c), clr(a), clr(b), ontable(b)$.

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

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 [Masseron 93]. An alternative approach is described in Chapter 5.

4.4 Masseron's geometry of conjunctive actions

[Masseron *et al* 93] explains how planning problems using conjunction only can be represented and solved in linear logic. The linear logic proof does not directly provide us with a plan, although it does contain all the information needed to extract the plan. The companion paper [Masseron 93] completes this picture by explaining the mapping between proofs in a restricted form of linear logic and plans expressed as directed graphs. The account below is a summary of this second paper.

4.4.1 Overview

This approach is restricted to a certain fragment of intuitionistic linear logic, using only the $\otimes$ connective and the rules $l\otimes, r\otimes, Ax$ and *cut*, plus the transition axioms which are part of a planning domain description. Now we can sketch the relationship between the proof rules and plans as follows:

- identity axiom corresponds to the empty plan.
- A transition axiom represents the performance of an action, and is associated with a plan containing only that action.
- The $r\otimes$ rule combines two independent plans. The plans are independent in the sense that they affect disjoint multisets of resources, there is therefore no dependency between the plans and they can be executed in parallel.
- The cut rule combines two plans in series, introducing dependency between the postconditions of one, and the preconditions of the other.

4.4.2 Details

Now we will make this more concrete by defining a representation for plans.

Definition 1 *A* pseudo-plan is defined as a finite graph composed of vertices and oriented edges, where:

- Each vertex is labelled by the name of a transition axiom. An axiom of the form

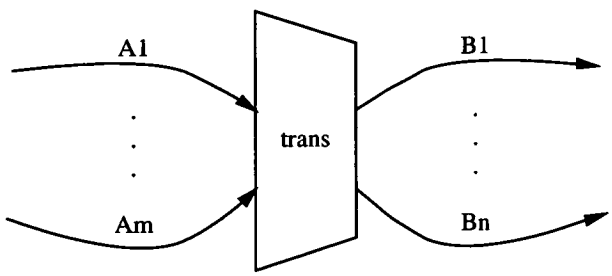
$$trans : A_1, \ldots, A_m \vdash B_1 \otimes \ldots \otimes B_n$$

is represented with a vertex with entries $iA_1, \ldots, iA_m$ and the exits $xB_1, \ldots, xB_n$.

- Each edge has an exit at its origin and an entry of the same type at its end. Edges represent linear resources used in the proof and are labelled as such.

- Entries of a pseudo-plan are entries of its component vertices which are not connected to the end of an edge.

- Exits of a pseudo-plan are exits of its component vertices which are not connected to the start of an edge.



A pseudo-plan D can be derived from a proof $\mathcal{D}$ by induction over proofs as follows (we use uppercase letters D, E, F to stand for proofs and corresponding curly letters $\mathcal{D}, \mathcal{E}, \mathcal{F}$ to stand for their associated pseudo-plans):

- If $\mathcal{D}$ is a proper axiom, D is a vertex, labelled by an occurrence of the name of the axiom. The vertex has entries and exits corresponding to either side of the axiom.

- If D is any other kind of axiom (e.g. identity axiom, Ax), $\mathcal{D}$ is empty.

- If D is obtained from E by an application of the $l\otimes$ rule, $\mathcal{D}$ is identical to $\mathcal{E}$.

- if D is obtained from E and F by an application of the $r\otimes$ rule, $\mathcal{D}$ is the union of $\mathcal{E}$ and $\mathcal{F}$.

- If D is obtained from E and F by an application of the cut rule on the formula $B_1 \otimes \ldots \otimes B_n$, $\mathcal{D}$ is obtained from the union of $\mathcal{E}$ and $\mathcal{F}$, with edges added from xB_j to iB_j for each j such that xBj is an exit of $\mathcal{E}$ and iB_j is an entry of $\mathcal{F}$.

The edges of the pseudo-plan can be used to define an ordering relation $<$ over the set of entries and exits of vertices:

$iA < xB$ if iA and xB are attached to the same vertex, and

$xA < iA$ if there exists an edge between xA and iA.

This also defines an ordering over the vertices. Vertices can be regarded as minimal pseudo-plans, and we can define a relation $\ll$ over pseudo-plans $\mathcal{E}$ and $\mathcal{F}$ as follows:

$\mathcal{E} \ll \mathcal{F}$ iff there exists $xA \in \mathcal{E}$ and $iB \in \mathcal{F}$ such that $xA < iB$.

4.4.3 Example

Given the blocks world proof below, the conversion would give the pseudo-plan illustrated in fig. 4.1.

$$\dfrac{clr(c) \vdash clr(c) \qquad \dfrac{\overbrace{empty, clr(b), ontable(b) \vdash hold(b)}^{take(b)} \quad \overbrace{hold(b), clr(a) \vdash empty \otimes clr(b) \otimes on(b,a)}^{stack(b,a)}}{empty, clr(b), ontable(b), clr(a) \vdash empty \otimes clr(b) \otimes on(b,a)}\ cut}{empty, clr(c), clr(b), ontable(b), clr(a) \vdash empty \otimes clr(b) \otimes on(b,a) \otimes clr(c)}\ \otimes r$$

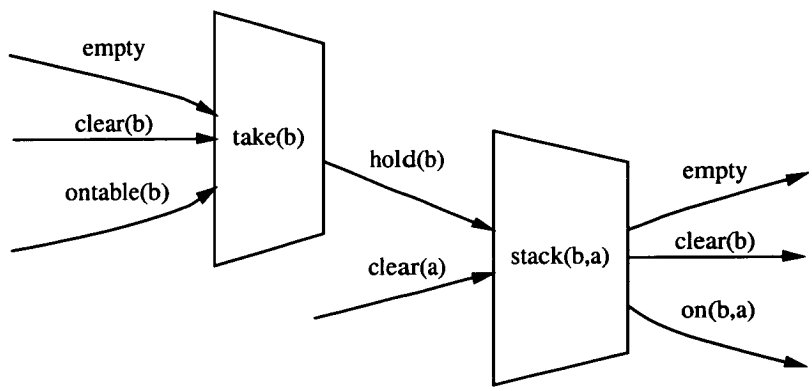


Figure 4.1: Example pseudo-plan

4.4.4 Discussion

This technique allows a pseudo-plan, represented as a directed graph, to be extracted from a linear logic proof. [1] The graph notation is less redundant in that a single graph

[1] The paper also considers the reverse translation

may represent many permutations of the proof. This can be attributed to the fact that the graph only contains edges directly between the producer of a resource (exit of a vertex) and its consumer. Notice that we would not add an edge for the case where a resource is transmitted through the cut rule untouched.

Note that this representation is very similar to that used in the linear connection method of [Bibel 86]. A comparison between these methods and an equational resolution method is given in [Große *et al* 96].

The main limitation is that the method is restricted to proofs using only the multiplicative conjunction connective, and would be difficult to extend to a larger fragment.

4.5 Summary

We have reviewed the linear logic approach to conjunctive planning as described by in [Masseron *et al* 93, Masseron 93]. We have looked at the representation of plan operators and plan specifications in intuitionistic linear logic. We have described how proofs can be built using an appropriate subset of linear logic deduction rules, and how plans can be extracted from the proofs. In Chapter 5, we will introduce an alternative representation for plans which allows a larger fragment of the logic to be used in representing and solving planning problems.

Chapter 5

Linear Logic Planning with Plan Terms

5.1 Introduction

This chapter introduces the notation for including plans as proof terms. This enables a larger fragment of the linear logic to be used in plan formation than that used in [Masseron *et al* 93]. We can then represent planning problems involving actions with uncertain effects and forms of partial ordering and quantification. We then define how the plan language is executed and indicate how plans may be partially evaluated. Chapter 6 extends this to include induction rules with extracts corresponding to recursive plans.

5.2 Constructing plan terms

Previous authors have extracted plans by using a procedure to recover plans from a 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 [Nordström *et al* 90]. 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 [Abramsky 93]. Proof terms can be seen as programs in Linear Lambda Calculus — a functional language in which there is a restriction of using each input exactly once.

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. An operational semantics, also from [Abramsky 93], defines how the constructs in the Linear Lambda Calculus are executed as plans.

In general, the inference rules describe how to build a plan for a given sequent out of plans for the subgoals associated with inference rule.

The logic we describe is based on Abramsky's version of Intuitionistic Linear Logic. We adapt this by omitting exponentials and additive conjunction, and by allowing a form of quantification (Section 5.2.7). We will refer to our version of the logic as Intuitionistic Linear Logic for Planning (ILL-P). The rules for ILL-P are given in Sections 5.2.1 — 5.2.8.

5.2.1 Transition axioms

Operator definitions now take the following form:

$$\vdash step : A \multimap C$$

5.2.2 Identity axiom and cut rule

The identity axiom says that required resource (or state) is available and we simply instantiate the label on the goal side to that on the resource side. Note that in linear logic, the Ax rule cannot apply if there are spare hypotheses on the left of the sequent.

$$\frac{}{x : A \vdash x : A}\ Ax$$

The *cut* rule allows us to attain a goal C via some intermediate state A. The plan for reaching A from the current state is u, though x acts as a placeholder for this plan as the plan u is formed, which accounts for the transition from the intermediate state to the final goal. In the final plan, we replace occurrences of x with t. This substitution is represented by the notation $u[t/x]$.

$$\frac{\Gamma \vdash t : A \quad \Gamma', x : A \vdash u : C}{\Gamma, \Gamma' \vdash u[t/x] : C}\ cut$$

5.2.3 Linear implication

The formula $A \multimap B$ corresponds to a single available action that can perform a transition from A to B.

The $l\multimap$ rule corresponds to the application of the action to its preconditions. Application of f to argument t is written here as $f \circ t$.

$$\frac{\Gamma \vdash t : A \quad x : B, \Gamma' \vdash u : C}{\Gamma, \Gamma', f : A \multimap B \vdash u[(f \circ t)/x] : C}\ l\multimap$$

We often wish to use *cut* and $l\multimap$ together. This enables us to cut in action from a (reusable) axiom, then apply the action. We call this combination of rules *lcut*.

$$\frac{\vdash step : A \multimap B \quad \dfrac{\Gamma \vdash t : A \quad \Gamma', y : B \vdash u : C}{x : A \multimap B, \Gamma, \Gamma' \vdash u[(x \circ t)/y] : C}\ l\multimap}{\Gamma, \Gamma' \vdash u[(step \circ t)/y]}\ cut$$

To prove $A \multimap B$ as a goal, we show that we can get B from A. The plan term $\lambda x.t$ indicates that the plan may be applied to an appropriate value for x. Here, λx indicates standard λ-abstraction.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ r\multimap$$

5.2.4 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 which may be executed in parallel.

$$\frac{\Gamma, x : A, y : B \vdash t : C}{\Gamma, z : A \otimes B \vdash \texttt{let } z \texttt{ be } x \otimes y \texttt{ in } t : C}\ l\otimes \qquad \frac{\Gamma \vdash t : A \quad \Gamma' \vdash u : B}{\Gamma, \Gamma' \vdash t \otimes u : A \otimes B}\ r\otimes$$

5.2.5 Disjunctive effects and conditionals

In some planning problems, it is 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.

[Masseron *et al* 93] 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).

$$\vdash \mathtt{pick} : hs \multimap (bs \oplus ws)$$

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 which may contain different actions in the different branches.

$$\frac{\Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma, z : A \oplus B \vdash \mathtt{case}\ z\ \mathtt{of}\ \mathtt{inl}(x)\ \mathtt{then}\ u, \mathtt{inr}(y)\ \mathtt{then}\ v : C}\ {}^{l\oplus}$$

This is equivalent to forms used in [Brüning *et al* 93].

The different approaches correspond to contingent and conformant planning, as discussed in Section 2.5. We discuss our restricted handling of conformant planning in Section 8.7.

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

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{inl}(t) : A \oplus B}\ {}^{r1\oplus}$$

$$\frac{\Gamma \vdash u : B}{\Gamma \vdash \mathtt{inr}(u) : A \oplus B}\ {}^{r2\oplus}$$

5.2.6 Special values

$\top$ *(top)* is used to avoid the need to fully specify a goal state. It consumes all resources present.

$$\frac{}{\Gamma \vdash \mathtt{erase} : \top}\ {}^{r\top}$$

In the example proof fragment below (shown without proof terms), the goal is to achieve $on(a,b)$, and we do not wish to specify what other resources are present. By including $\top$ in the goal specification, we can allow the goal state to match with any context that includes the goal condition. The $r\top$ rule allows the remaining resources to be consumed.

$$\dfrac{\dfrac{}{on(a,b) \vdash on(a,b)}\,Ax \quad \dfrac{}{clear(a), on(b,table), clear(c), on(c,table) \vdash \top}\,r\top}{on(a,b), clear(a), on(b,table), clear(c), on(c,table) \vdash on(a,b) \otimes \top}\,r\otimes$$

The value **0** is used to denote impossible situations. Thus any resource is allowed to be derived.

$$\dfrac{}{\Gamma, \mathbf{0} \vdash \mathtt{abort} : A}\,l\mathbf{0}$$

5.2.7 Quantifiers

To make recursive plans or schematic plans, we will need to be able to handle quantification in some form. We will consider quantification over unrestricted types only, not over linear types in the sense of [Pfenning 98].

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.

A universally quantified resource is one for which exactly one instance can be used, and we choose the instance.

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. Thus the rules are:

$$\dfrac{\Gamma, y : A[t/x] \vdash c : C}{\Gamma, z : \forall x{:}\tau\,.\,A \vdash c[(z \circ t)/y] : C}\,l\forall \qquad \dfrac{\Gamma \vdash z : A[a/x]}{\Gamma \vdash \lambda a.z : \forall x{:}\tau\,.\,A}\,r\forall$$

where a is not free in Γ.
τ is the type of the quantified variable (we omit typing constraints on terms t which are implicit).

An existentially quantified resource is one where we are assured that some instance is a resource, but we do not know which one.

$$\frac{\Gamma, t : A[a/x] \vdash c : C}{\Gamma, t : \exists x{:}\tau \,.\, A \vdash c : C}\ {}^{l\exists} \qquad \frac{\Gamma \vdash c : C[t/x]}{\Gamma \vdash c : \exists x{:}\tau \,.\, C}\ {}^{r\exists}$$

where a is not free in Γ, C.

5.2.8 Equality

We also sometimes need to reason about equality. Equality can be represented as a resource $a = h$. Such equations may appear on the left hand side of sequents with dummy labels (i.e. these labels can never appear in plan terms). To allow substitution we introduce a new deduction rule:

$$\frac{\Gamma \vdash c : C[b/a]}{\Gamma, (a = b) \vdash c : C}\ {}^{substitute(a=b)}$$

We suppose a standard background equality theory.

This concludes the rules for ILL-P.

5.2.9 Relationship to features of planning problems

The following table describes how the various connectives and special values described in this chapter realate to features of planning problems. Note that the sequent symbol, $\vdash$, corresponds to linear implication, $\multimap$. Simple actions and plan specifications can be written without using $\multimap$, as in [Masseron *et al* 93].

Connective	Problem features
$\otimes$	This is the only connective used in simple conjunctive STRIPS in the style of [Jacopin 93].
$\multimap$	Used in goal position, allows problem specification to be stated as a sequent with an empty LHS. This is the required form for extracting a plan term. Used in resource position, $\multimap$ allows an available action to be treated as a resource.
$\oplus$	Planning problems involving disjunction in goals or effects. Disjunction of resources in an action effect is used to specify actions with uncertain outcomes.
$\forall$	Universal quantifier around plan specification allows for synthesis of schematic plans.
$\exists$	In goal position, can be used to specify required value. In effect position, stands for an unspecified value.

Connective	Problem features
$\top$	Used in problems with partially-specified goal.
0	Used to denote impossible situations. Axioms with **0** in effects can be used as a way to allow the closing of branches of the proof which describe impossible situations.

5.2.10 Partial specification of initial state

The initial state must include all of the resources that are needed as action preconditions and goals. Some degree of flexibility is possible in describing the initial states as follows:

1. Uncertain initial states may be described using a disjunction of resources. If it is necessary that certain combinations of disjuncts need not be considered, this can be specified by inclusion of axioms that match those situations and produce **0** as an effect. This marks the situation as impossible and allows the proof branch to be closed.

2. Quantifiers may be used instead of giving explicit values.

5.3 Plan execution

To describe the order of execution of the plan language, we need an operational semantics. Abramsky provides a semantics for the Linear Lambda Calculus. This can be used to ensure that our primitive steps are executed in a correct order.

In the following, the letters t and u are used to represent arbitrary terms of the language, whereas c and d represent terms which have been evaluated to a canonical form. The relation $t \Downarrow c$ says that t evaluates to a canonical form c. The canonical forms are given by:

$$c \otimes d \quad \lambda x.t \quad \mathtt{inl}(c) \quad \mathtt{inr}(d)$$

The operational semantics rules define how the evaluation of a terms of the plan language is found from the evaluation its subterms. The canonical forms evaluate to themselves.

To use these rules to execute a plan, we must find a rule for which the term below matches the program, and for which evaluation of corresponding terms above the line also match.

In most of the operational semantics rules, a left to right order of evaluation of the terms above the line is enforced by dependencies between symbols on the top line. E.g. in the rule below that we require the derivation of canonical forms c and d before substitution into u.

$$\frac{t \Downarrow c \otimes d \quad u[c/x, d/y] \Downarrow c}{\texttt{let } t \texttt{ be } x \otimes y \texttt{ in } u \Downarrow c}$$

A notable exception to this is the rule for executing $t \otimes u$. Here the terms above the line are completely independent of each other, so the rules are not deterministic about execution order. This corresponds to two plans t and u which may be executed in parallel. The linear logic proof guarantees that there is no interaction between the two plans, by ensuring that no resource appears in both plans (see Sec. 5.5.1).

$$\frac{t \Downarrow c \quad u \Downarrow d}{t \otimes u \Downarrow c \otimes d}$$

The rules for handling lambda-terms and their application are:

$$\frac{}{\lambda.t \Downarrow \lambda.t} \qquad \frac{t \Downarrow \lambda x.v \quad u \Downarrow c \quad v[c/x] \Downarrow d}{t \circ u \Downarrow d}$$

The rules below give handling for conditionals.

$$\frac{t \Downarrow c}{\texttt{inl}(t) \Downarrow \texttt{inl}(c)} \qquad \frac{t \Downarrow c}{\texttt{inr}(t) \Downarrow \texttt{inr}(c)}$$

$$\frac{t \Downarrow \texttt{inl}(c) \quad u[c/x] \Downarrow d}{\texttt{case } t \texttt{ of inl}(x) \texttt{ then } u, \texttt{inr}(y) \texttt{ then } v \Downarrow d}$$

$$\frac{t \Downarrow \texttt{inr}(c) \quad u[c/y] \Downarrow d}{\texttt{case } t \texttt{ of inl}(x) \texttt{ then } u, \texttt{inr}(y) \texttt{ then } v \Downarrow d}$$

5.4 Partial evaluation

In this section we define partial evaluation on plan terms by giving rewrite rules for Linear Lambda Calculus. Exhaustive application of these rewrite rules on a plan term

with respect to an input initial situation results in the elimination of some of the control structures.

If some of the input data is available before execution, then application of the rewrite rules results in a partial evaluation of the plan — i.e. we specialise with respect to the available input situation.

One case of partial evaluation we are interested in is where a plan $\forall x(I(x) \multimap G(x))$ is evaluated for an instance $I(t) \multimap G(t)$. Here we apply rewrite rules to $plan \circ t$, resulting in a new plan, $plan'$, which is specialised for instance t.

We define a one-step partial evaluation relation as a rewriting relation based on the following four rules:

$$\texttt{let } c \otimes d \texttt{ be } x \otimes y \texttt{ in } u \longrightarrow u[c/x, d/y] \tag{5.1}$$

$$(\lambda x.t) \circ u \longrightarrow t[u/x] \tag{5.2}$$

$$\texttt{case inl}(c) \texttt{ of inl}(x) \texttt{ then } u, \ldots \longrightarrow u[c/x] \tag{5.3}$$

$$\texttt{case inr}(c) \texttt{ of} \ldots, \texttt{inr}(y) \texttt{ then } v \longrightarrow v[c/y] \tag{5.4}$$

The relation $\longrightarrow$ thus allows rewriting at arbitrary subexpressions of a plan term. Our rewriting respects the binding associated with λ-abstraction — it is sometimes necessary to rename variables in lambda-terms to prevent capture of free variables due to an accidental correspondence of variable names. This is a form of higher-order rewriting (see [Baader & Nipkow 98] for details).

5.4.1 Correctness of partial evaluation

Definition 2 *We say a relation R is* correct *with respect to a type system iff the following holds:*
if $a\,R\,a'$ and $\vdash a : C$ is a provable sequent, then $\vdash a' : C$ is also a provable sequent.

We state here but do not prove:

Claim 1 *the relation $\longrightarrow$ is correct w.r.t. the system ILL-P.*

It follows that the relation $\longrightarrow^*$ (the reflexive transitive closure of $\longrightarrow$) is also correct w.r.t. ILL-P. One way to show the correctness of the rewrite rules is to carry out transformations on the proofs corresponding to the transformations on the plan terms, by induction over the structure of proofs.

5.4.2 The relationship between $\longrightarrow$ and $\Downarrow$

The relationship between transformations by partial evaluation rules ($\longrightarrow$) and by operational semantics rules ($\Downarrow$) is as follows. We require, but do not prove that:

Claim 2

1. *If* $\vdash P : \forall x.Z$ *and* $P \circ t \longrightarrow^* P'$
   then $\vdash P' : Z$ *and* $\forall r[(P \circ t \Downarrow r)$ *iff* $(P' \Downarrow r)]$.
2. *If* $\vdash P : A \multimap B$ *and* $a : A$ *and* $P \circ a \longrightarrow^* P'$
   then $\vdash P' : B$ *and* $\forall r.[(P \circ a \Downarrow r)$ *iff* $(P' \Downarrow r)]$.
3. *If* $\vdash P \circ D_1 \circ \ldots \circ D_n : T$ *and* $P \circ D_1 \circ \ldots \circ D_n \longrightarrow^* E$
   then $E : T$ *and* $\forall r\,[(P \circ D_1 \circ \ldots \circ D_n \Downarrow r)$ *iff* $(E \Downarrow r)]$.

In each case, the claim is that the plan term resulting from partial evaluation is correctly typed, and that execution of the partially evaluated term yields the same result as execution of the original plan. Cases 2.1 and 2.2 cover the applications of single lambda-terms arising from universal quantifiers and from linear implication in goal position. Case 2.3 covers the nesting of lambda terms and their applications.

The typing claims of 2.1 and 2.2 can be obtained from Claim 1 and type inference for application terms. 2.3 can be proved by induction over the definitions of $\Downarrow$ and $\longrightarrow^*$.

Where there is a computational advantage, we may choose to work with a partially evaluated plan term and avoid some run-time cost. This may, however, result in larger use of space resource, as the plan term may grow larger on partial evaluation.

5.5 Correspondence of Linear Lambda Calculus programs to plans

Programs in Linear Lambda Calculus have the appearance of functional programs. However, they have the special feature that each linear argument is only used once, a

property which is guaranteed by the corresponding linear logic proof. Use of linear implication, which represents state change, corresponds to function application in Linear Lambda Calculus. A plan in our formalism is a program which accepts as arguments the resources defining an initial state, and returns the goal state.

In the following example, we look at a plan to solve the goal:

$$\vdash empty \otimes clr(b) \otimes on(b,c) \otimes clr(a) \multimap empty \otimes clr(b) \otimes on(b,a) \otimes clr(c)$$

with the transition axioms:

$$\vdash \texttt{take} : empty \otimes clr(X) \otimes on(X,Y) \multimap hold(X) \otimes clr(Y)$$

$$\vdash \texttt{stack} : hold(X) \otimes clr(Y) \multimap empty \otimes clr(X) \otimes on(X,Y)$$

The plan to solve this problem is:

```
let take∘h1*h2*h3 be h6*h7 in
  let stack∘h4*h6 be h9*h10 in
    let h10 be h11*h12 in
      h9*h11*h12*h7
```

It is found by building a proof using rules above, and then computing the resultant extract term.

The `h` values are labels for the hypotheses which appear in the linear logic proof.

$$\begin{aligned}
h1 &: \quad empty \\
h2 &: \quad clr(b) \\
h3 &: \quad on(b,c) \\
h4 &: \quad clr(a) \\
h5 &: \quad hold(b) \otimes clr(c) \\
h6 &: \quad hold(b) \\
h7 &: \quad clr(c) \\
h8 &: \quad empty \otimes clr(b) \otimes on(b,a) \\
h9 &: \quad empty \\
h10 &: \quad clr(b) \otimes on(b,a) \\
h11 &: \quad clr(b) \\
h12 &: \quad on(b,a)
\end{aligned}$$

To understand the program, we need to consider the values which are passed in and out of action steps. Fig. 5.1 shows the relationship between actions and values in the

plan. Arrows entering a box represent arguments and arrows leaving a box represent returned results. The splitting and joining of lines simply corresponds to the book-keeping operations of constructing and dismantling (multiplicative) conjunctions, and these operations are interleaved with the actions.

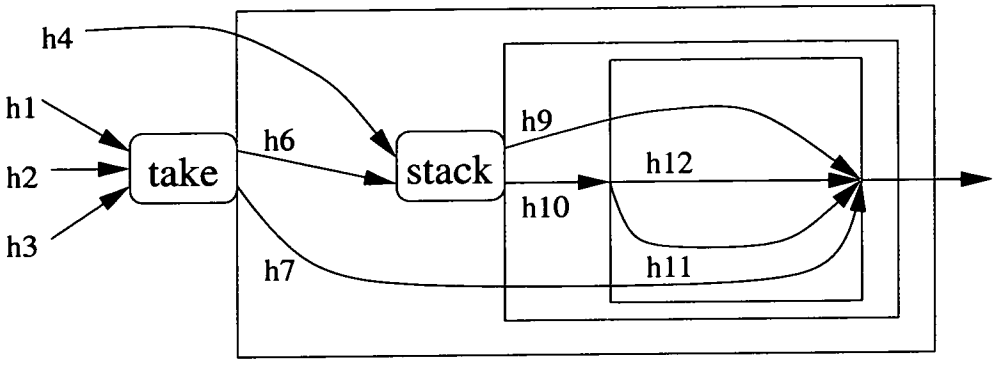


Figure 5.1: Example program/plan

5.5.1 Partially-ordered plans in Linear Lambda Calculus

Here we give an example of a simple plan in which the steps are partially ordered, and show how this is represented in both STRIPS and in Linear Lambda Calculus. The problem is one of opening a door which has two locks. The partial-order aspect is present because the two locks may be unlocked in either order. The STRIPS representation of the of the problem is given below, and graphical representation of the solution plan is given in Fig. 5.2.

```
operator:      unlock(L)
preconditions: locked(L)
deletelist:    locked(L)
addlist:       unlocked(L)
```

```
operator:      open_door
preconditions: unlocked(lock1)
               unlocked(lock2)
               door_closed
deletelist:    door_closed
addlist:       door_open
```

```
initial: locked(lock1)
         locked(lock2)
         door_closed

goal:    unlocked(lock1)
         unlocked(lock2)
         door_open
```

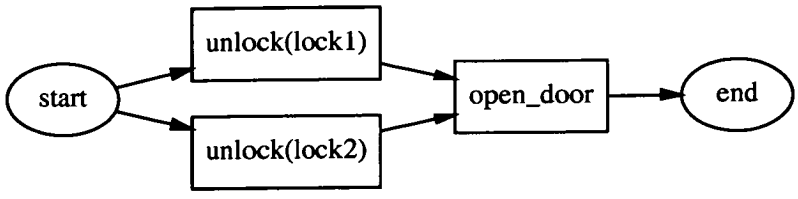


Figure 5.2: Example partially-ordered plan in STRIPS

In linear logic, the actions are represented by the following axioms:

$$\vdash \texttt{unlock(l)}: \quad locked(l) \multimap unlocked(l)$$

$$\vdash \texttt{open\_door}: \quad \left[\begin{array}{c} unlocked(lock1) \\ \otimes \\ unlocked(lock2) \\ \otimes \\ door\_closed \end{array}\right] \multimap \left[\begin{array}{c} unlocked(lock1) \\ \otimes \\ unlocked(lock2) \\ \otimes \\ door\_open \end{array}\right]$$

The problem specification is given by:

$$\vdash locked(lock1) \otimes locked(lock2) \otimes door\_closed \multimap unlocked(lock1) \otimes unlocked(lock2) \otimes door\_open$$

The plan is given below. Notice that `open_door` action is applied to a conjunction of resources, and it is in obtaining each conjunct that the execution order is not forced (Section 5.3). The applications of the `unlock` actions are within this conjunction, and hence may executed in any order (or in parallel). Fig. 5.3 gives a graphical representation of the plan.

```
λh1.
  let h1 be h2*h3 in
    let h3 be h4*h5 in
      open_door∘
        h5*
        (unlock∘h2)*
        (unlock∘h4)
```

The labels have the following types:

$$\begin{array}{ll} h1: & locked(lock1) \otimes locked(lock2) \otimes door\_closed \\ h2: & locked(lock1) \\ h3: & locked(lock2) \otimes door\_closed \\ h4: & locked(lock2) \\ h5: & door\_closed \end{array}$$

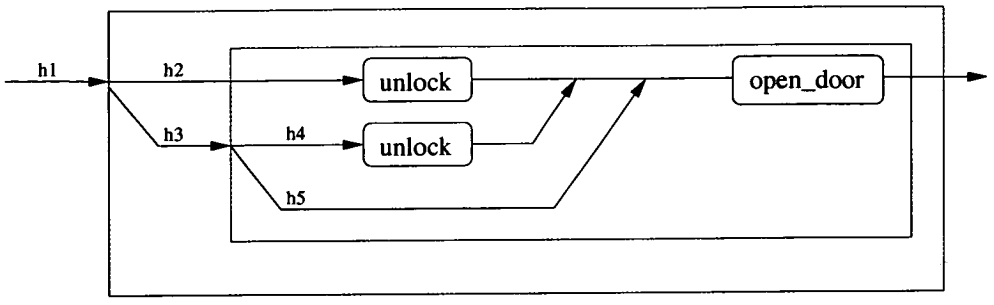


Figure 5.3: Example graphical representation of partially-ordered plan in Linear Lambda Calculus

5.5.2 Execution of primitive steps

The parts of the program which arise from the application of transition axioms have a special meaning. These are applications of actions, which bring about state changes. They may represent the actions performed by a robot to transform the state of the external world. Hence, these steps cannot be removed by partial evaluation, and it is very important that they are executed in a correct order. An order is enforced where necessary by the operational semantics.

5.6 Sources

This section is largely based on [Abramsky 93], though we omit rules for additive conjunction, exponentials, and the special value **1** given there, as they have not appeared useful in the planning problems. We have added the quantifier rules, which are given, without proof terms, in [Masseron *et al* 93]. The rules for $r\top$ and $l\mathbf{0}$, also omitted by Abramsky, are based on those in [Masseron *et al* 93] and [Pfenning 98]. The partial evaluation rules are similar to forms used in [Pfenning 98]. The handling of equality is our own, though this has not been fully developed.

5.7 Summary

We have explained the extraction of plans from proofs of linear logic specifications. The plans are built up as terms, whose construction is guided by the deduction rules of linear logic. Additionally, this chapter has described the partial evaluation and

execution of such plans.

Chapter 6

Induction and Formation of Recursive Plans

6.1 Introduction

In a traditional formulation of a planning problem, we have a set of primitive operators which must be used to form a plan to get from fully specified initial state to a goal state. The solution of a planning problem is a partially ordered sequence of fully ground operator applications.

A frequently used example of a planning problem is the blocks world. A hand-crafted procedure for solving blocks world problems can guarantee to always solve the problem in a time linear in the number of blocks [Slaney & Thiebaux 96]. Little attention has been given to the problem synthesising such procedures automatically, though limited recursive problems in the blocks-world have been addressed [Ghassem-Sani & Steel 91].

We believe that recursive structure is often present in conventional planning problems, but is usually hidden by the way that the domain is defined.

In fact, the inherent structure may allow a single abstract procedure to solve classes of problems in the domain once and for all. This is expected to be much more efficient in the case of large but structured problems, and also provides a solution to the problem of incomplete knowledge — we can guarantee a solution without knowing everything about the world.

In this chapter, we show how the superior expressiveness of linear logic over conven-

tional planning approaches enables recursive plans to be formed. By treating the planning process as a proof problem, we exploit the simple relationship between proof by induction and recursive procedures.

This chapter introduces induction, and relates induction in linear logic to the formation of recursive plans. induction on lists, natural numbers and trees are considered. We give induction rules, partial evaluation rules and operational semantics for these datatypes. The logic which we form by extending ILL-P to enable handling of recursive problems will be referred to as ILL-PR.

We deal with issues which arise in representing the planing problems, such as how to represent the problem specification so as to avoid fully specifying the goal state.

6.2 Induction

We choose to perform induction via inductively defined datatypes. For example, lists of untyped elements can be defined as follows:

$$list ::= (term :: list) \mid \texttt{nil}$$

The usual definition for structural induction on lists is:

$$\frac{\Gamma \vdash P(nil) \quad \Gamma \vdash P(l') \rightarrow P(h' :: l')}{\Gamma \vdash \forall l : list.P(l)}$$

Such inductive definitions are associated with a well-founded order — see [Luo 94] for details.

6.2.1 Special considerations for induction in Linear Logic

In linear logic, we can write a similar rule, but we must be careful about the treatment of Γ. Since it denotes linear resources which are to be used exactly once, it would be incorrect to allow its use in the step case of the proof. This would lead to a plan in which the same resources are consumed on each successive recursive call. Exponentials, which are tagged formulas that can be used any number of times, are permissible in Γ, but we omit them in our treatment.

However, if the form of the induction rule is such that the base case can only be reached once, this corresponds to a part of the plan that is only executed once, and it is therefore permissible to allow the Γ to be consumed in the base case.

$$\frac{\Gamma \vdash P(nil) \quad P(l') \vdash P(h' :: l')}{\Gamma \vdash \forall l : list.P(l)}$$

In the rest of this thesis, for consistency, we prefer to omit the Γ in our definitions of induction rules. This is not as restrictive as it may appear, since $P(l)$ itself is often of the form $I(l) \multimap G(l)$. In this case the $I(l)$ plays a role similar to the Γ, but this form of the induction rule demands that $I(l)$ should be replenished if used.

In the example given below, we will consider manipulating towers, which are represented as lists of blocks.

6.3 Formation of recursive plans

6.3.1 Inductive datatype and corresponding induction rule

Here we illustrate induction using lists to represent towers of blocks.

We define a datatype for towers:

$$tower ::= (block :: tower) \mid \texttt{empty}$$

In the inductive proof, the handling of plan terms presents special difficulties. The syntax for plans is extended by adding the notion of recursive plan, based on the treatment of recursion in type theory [Nordström *et al* 90].

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

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

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. Linear logic enforces that it must be used exactly once in the step case plan.

Other induction rules are possible, which correspond to different uses of resources — e.g. for natural numbers, two forms *nat_ind* and *nat_ind2* are given in Section 6.3.2.

Execution and partial evaluation of `twr_rec` is considered in Sections 6.3.4 and 6.3.5.

6.3.2 Other inductively defined datatypes

Natural numbers

The Peano formulation of natural numbers is given by the inductive datatype:

$$nat ::= \mathtt{s}(nat) \mid \mathtt{zero}$$

An induction rule for the datatype is:

$$\frac{\vdash \mathtt{bp} : F(\mathtt{zero}) \quad \mathtt{r} : F(n') \vdash \mathtt{sp} : F(\mathtt{s}(n'))}{\vdash \lambda \mathtt{n}.\mathtt{nat\_rec}(\mathtt{n}, \mathtt{bp}, \lambda \mathtt{r}.\lambda \mathtt{n}'.\mathtt{sp}) : \forall n : nat.F(n)}\ \textit{nat\_ind}$$

An alternative form of induction gives steps of 2:

$$\frac{\vdash \mathtt{bp}_0 : F(\mathtt{zero}) \quad \mathtt{bp}_1 : F(\mathtt{s}(\mathtt{zero})) \quad \mathtt{r} : F(n') \vdash \mathtt{sp} : F(\mathtt{s}(\mathtt{s}(n')))}{\vdash \lambda \mathtt{n}.\mathtt{nat\_rec}(\mathtt{n}, \mathtt{bp}_0, \mathtt{bp}_1, \lambda \mathtt{r}.\lambda \mathtt{n}'.\mathtt{sp}) : \forall n : nat.F(n)}\ \textit{nat\_ind2}$$

Trees

The tree datastructure in this example is a binary tree with values at the branching nodes, but not at the leaves.

$$tree ::= \mathtt{node}(tree, tree, value) \mid \mathtt{empty}$$

We can then write an induction rule for t of type tree:

$$\frac{\vdash P(\mathtt{empty}) \quad P(l), P(r) \vdash P(\mathtt{node}(l, r, v))}{\vdash \forall t : tree.P(t)}\ \textit{tree\_ind}$$

where l and r represent the left and right subtrees.

Note that we have any empty context on the left of the sequent. We cannot even use a context in the base case, since it may be executed more than once.

Adding plan terms, we have:

$$\frac{\vdash \mathtt{bp} : P(\mathtt{empty}) \quad \mathtt{recl} : P(l), \mathtt{recr} : P(r) \vdash \mathtt{sp} : P(\mathtt{node}(l, r, v))}{\vdash \lambda \mathtt{t.tree\_rec}(\mathtt{t}, \mathtt{bp}, \lambda \mathtt{recl}.\lambda \mathtt{recr}.\lambda \mathtt{l}.\lambda \mathtt{r}.\lambda \mathtt{v.sp}) : \forall t : tree.P(t)} \; tree\_ind$$

where *bp* and *sp* are the plans for base and step cases, and *rec* is the recursive procedure call.

6.3.3 Recursion versus iteration

We have chosen to use recursion to represent all forms of looping behaviour in our plan language. In some circumstances, it is beneficial to synthesise plans that use the more restrictive form of simple iteration. One reason to do this is that such plans can be executed by a simpler mechanism, with lower space requirements. Generally, when a recursive call is made, it is necessary to store the state of the calling function on a stack so that it can be resumed when the recursive call returns. Hence the space requirement for the computation increases with the number of nested calls.

However, in the case of recursive computation where the result of the recursive call will be also be returned as the result of the calling function, there is no need to store the state of the calling function. This is known as a *tail recursive* call.

A function which is tail recursive is equivalent to iteration. In the following, we describe restrictions that can be imposed on the form of our proofs to restrict synthesised plans to be in the tail-recursive form.

Tail recursion

Tail recursive plans can be found by restricting the form of the step cases in the induction.

For example, for natural numbers, if we are generating the proof for a plan specification of the form

$$\forall n.pre(n) \multimap \mathit{eff}(n)$$

then we can use an induction rule of the form:

$$\frac{\vdash pre(zero) \multimap \mathit{eff}(zero) \quad pre(n') \multimap \mathit{eff}(n') \vdash pre(s(n')) \multimap \mathit{eff}(s(n'))}{\vdash \forall n.pre(n) \multimap \mathit{eff}(n)}$$

The step case can be proved if the following two goals can be proved:

$$pre(s(n')) \vdash pre(n') \tag{6.1}$$

$$eff(n') \vdash eff(s(n')) \tag{6.2}$$

This can be shown by the following partial proof of the induction rule step case, in which 6.1 and 6.2 form the open sequents:

$$\dfrac{\dfrac{pre(s(n')) \vdash pre(n') \quad eff(n') \vdash eff(s(n'))}{pre(n') \multimap eff(n'), pre(s(n')) \vdash eff(s(n'))}\ {}^{l\multimap}}{pre(n') \multimap eff(n') \vdash pre(s(n')) \multimap eff(s(n'))}\ {}^{r\multimap}$$

If we read the induction hypothesis, $pre(n') \multimap eff(n')$ as the specification of a recursive function, then the plan corresponding to the sequent 6.1 is the plan to be executed before the recursive call is made, meeting the preconditions of the recursive call.

The plan corresponding to the sequent 6.2 is the plan to be executed after the recursive call is made. The criterion for making our plan tail-recursive is that the result of the recursive call is also the result of the calling procedure - i.e. this part of the plan is empty.

This happens when

$$eff(n') = eff(s(n'))$$

i.e. proof above this point does not contain any applications of actions.

In summary we can write the tailored form of the induction rule as follows:

$$\dfrac{\vdash pre(zero) \multimap eff(zero) \quad pre(s(n')) \vdash pre(n') \quad \overbrace{eff(n') \vdash eff(s(n'))}^{\text{No actions in this proof}}}{\vdash \forall n.pre(n) \multimap eff(n)}$$

An example of a plan which corresponds to the tail-recursive form (for towers) is given in Section 6.4.1.

6.3.4 Operational semantics of recursive plans

Section 5.3 described an operational semantics defining the execution of recursive plans, which is a subset of that described in [Abramsky 93]. In this section, we describe our

extensions to describe execution of recursive plans of the forms described in Sections 6.3.1 and 6.3.2.

In the following bp, sp, c, d and e are canonical.

Towers

First, we will give a solution for the case of towers. The general form of a plan for recursion over towers is:

$$\lambda \mathtt{t}.\mathtt{twr\_rec}(\mathtt{t}, \mathtt{bp}, \lambda \mathtt{x}.\lambda \mathtt{y}.\lambda \mathtt{z}.\mathtt{sp})$$

where `t` is a tower,

`bp` is the base case of the recursive plan,

`sp` is the step case of the recursive plan taking the following parameters:

`x` is the recursive plan,

`y` is the top block,

`z` is the rest of the tower.

When we apply this rule to a specific tower in place of t, we know from the definition of the datatype that t is either `empty` or is of the form $b' :: t'$. Hence, in reducing the plan term, we have two different cases to deal with:

If t evaluates to `empty`, then execute bp:

$$\frac{t \Downarrow \mathtt{empty} \quad bp \Downarrow c}{\mathtt{twr\_rec}(t, bp, sp) \Downarrow c}$$

Otherwise, evaluate the step case plan, sp.

$$\frac{t \Downarrow c :: d \quad sp \circ \mathtt{twr\_rec}(t, bp, sp) \circ c \circ d \Downarrow e}{\mathtt{twr\_rec}(t, bp, sp) \Downarrow e}$$

Note that the second (step case) rule must pass the entire recursive plan as an argument to the step case plan.

Natural numbers

The selection of the appropriate rule for execution of natural number depends on the evaluation of t

$$\frac{t \Downarrow \texttt{zero} \quad bp \Downarrow c}{\texttt{nat\_rec}(t, bp, sp) \Downarrow c}$$

$$\frac{t \Downarrow s(n) \quad sp \circ \texttt{nat\_rec}(n, bp, sp) \circ n \Downarrow c}{\texttt{nat\_rec}(t, bp, sp) \Downarrow c}$$

Trees

For trees, the operational semantics rules must handle two instances of recursive call — one for each subtree.

$$\frac{t \Downarrow \texttt{empty} \quad bp \Downarrow c}{\texttt{tree\_rec}(t, bp, sp) \Downarrow c}$$

$$\frac{t \Downarrow \texttt{node}(l, r, v) \quad sp \circ \texttt{tree\_rec}(l, bp, sp) \circ \texttt{tree\_rec}(r, bp, sp) \circ l \circ r \circ v \Downarrow c}{\texttt{tree\_rec}(t, bp, sp) \Downarrow c}$$

6.3.5 Partial evaluation of recursive plans

We must also define how plans are partially evaluated in the case of recursive plans. To do this, we define an extended partial evaluation relation $\Longrightarrow$.

Towers

$$\texttt{twr\_rec}(\texttt{empty}, bp, sp) \Longrightarrow bp \tag{6.3}$$

$$\texttt{twr\_rec}(b :: t, bp, sp) \Longrightarrow sp \circ \texttt{twr\_rec}(t, bp, sp) \circ b \circ t \tag{6.4}$$

Natural numbers

Similarly, for natural numbers, we can give the partial evaluation rules as:

$$\texttt{nat\_rec}(\texttt{zero}, bp, sp) \Longrightarrow bp \tag{6.5}$$

$$\texttt{nat\_rec}(\texttt{s}(n), bp, sp) \Longrightarrow sp \circ \texttt{nat\_rec}(n, bp, sp) \circ n \tag{6.6}$$

Trees

In the case of binary trees, we must pass two instances of the recursive call — one for each subtree.

$$\texttt{tree\_rec}(\texttt{empty}, bp, sp) \Longrightarrow bp \tag{6.7}$$

$$\texttt{tree\_rec}(\texttt{node}(l, r, v), bp, sp) \Longrightarrow sp \circ \texttt{tree\_rec}(l, bp, sp) \circ \texttt{tree\_rec}(r, bp, sp) \circ l \circ r \circ v \tag{6.8}$$

Correctness of partial evaluation

in Section 5.4 we claimed correctness of the partial evaluation relation $\longrightarrow$ w.r.t. the system ILL-P. For the extended logic ILL-PR and the extended relation $\Longrightarrow$, used in rules (6.4 - 6.8), we require, but do not prove:

Claim 3 $\Longrightarrow$ *is correct (def. 2) w.r.t. the system ILL-PR.*

It follows that the relation $\Longrightarrow^*$ (the reflexive transitive closure of $\Longrightarrow$) is also correct w.r.t. ILL-PR.

6.3.6 The relationship between $\Longrightarrow$ and $\Downarrow$

Here, we repeat the claim of Section 5.4.2 for the extended relation $\Longrightarrow$, that the result of executing the partially evaluated plan is the same as executing the original plan.

Claim 4

1. *If* $\vdash P : \forall x.Z$ *and* $P \circ t \Longrightarrow^* P'$
   then $\vdash P' : Z$ *and* $\forall r[(P \circ t \Downarrow r)$ *iff* $(P' \Downarrow r)]$.
2. *If* $\vdash P : A \multimap B$ *and* $a : A$ *and* $P \circ a \Longrightarrow^* P'$
   then $\vdash P' : B$ *and* $\forall r.[(P \circ a \Downarrow r)$ *iff* $(P' \Downarrow r)]$
3. *If* $\vdash P \circ D_1 \circ \ldots \circ D_n : T$ *and* $P \circ D_1 \circ \ldots \circ D_n \Longrightarrow^* E$
   then $E : T$ *and* $\forall r\,[(P \circ D_1 \circ \ldots \circ D_n \Downarrow r)$ *iff* $(E \Downarrow r)]$.

6.4 Examples

6.4.1 Example using towers

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. This cannot be be executed itself as a plan, but simply states properties of the specification language.

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:

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

which means that a tower t can reversed by execution of *plan*. *rev* is a reverse function, used in defining relationship between initial and goal states.

The following will be used as plan operators.

$$\begin{array}{lrcl} \vdash \mathtt{pick(b :: t)} : & twr(b :: t) \otimes hn & \multimap & twr(t) \otimes hold(b) \\ \vdash \mathtt{put(b, t)} : & twr(t) \otimes hold(b) & \multimap & twr(b :: t) \otimes hn \end{array}$$

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 specification is discussed in Section 3.5.6. The generalised plan specification is given below, and this is what we must prove in order to sythesise the plan:

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

The definitions of *rev* and *app* are given by:

$$rev(empty) \;=\; empty \tag{6.9}$$

$$rev(b :: t) = app(rev(t), b :: empty) \tag{6.10}$$

$$app(empty, u) = u \tag{6.11}$$

$$app(b :: t, u) = b :: app(t, u) \tag{6.12}$$

From this we can prove the associativity of *app*.

$$app(app(a, b), c) = app(a, app(b, c)) \tag{6.13}$$

Now we use the induction rule (for clarity, we will omit plan terms).

$$\frac{\langle base\rangle \quad \langle step\rangle}{\vdash \forall t.\forall a.twr(t) \otimes twr(a) \otimes hn \multimap twr(empty) \otimes twr(app(rev(t), a)) \otimes hn}\ twr\_ind(t)$$

Base case:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{twr(empty) \otimes twr(a_1) \otimes hn \vdash twr(empty) \otimes twr(a_1) \otimes hn}\ Ax}{twr(empty) \otimes twr(a_1) \otimes hn \vdash twr(empty) \otimes twr(app(empty, a_1)) \otimes hn}\ rrewrite(app1)}{twr(empty) \otimes twr(a_1) \otimes hn \vdash twr(empty) \otimes twr(app(rev(empty), a_1)) \otimes hn}\ rrewrite(rev1)}{\vdash twr(empty) \otimes twr(a_1) \otimes hn \multimap twr(empty) \otimes twr(app(rev(empty), a_1)) \otimes hn}\ r\multimap}{\vdash \forall a.twr(empty) \otimes twr(a) \otimes hn \multimap twr(empty) \otimes twr(app(rev(empty), a)) \otimes hn}\ r\forall}{\langle base\rangle}$$

Step case:

In searching for such a proof, we allow meta-variables (A in this proof) so as to delay the choice of witness term t in the use of the rule $l\forall$.

The step case involves applications of the *pick* and *put* actions, and the application of the recursive call represented by the induction hypothesis.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle 1\rangle \quad \langle 2\rangle}{\begin{array}{c}twr(t_1)\otimes twr(A)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),A))\otimes hn,\\ twr(t_1),\\ twr(b_1 :: a_2),\\ hn\end{array} \vdash twr(empty)\otimes twr(app(rev(b_1 :: t_1),a_2))\otimes hn}\,l\multimap}{\begin{array}{c}twr(t_1)\otimes twr(A)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),A))\otimes hn,\\ twr(t_1),\\ twr(b_1 :: a_2)\otimes hn\end{array} \vdash twr(empty)\otimes twr(app(rev(b_1 :: t_1),a_2))\otimes hn}\,l\otimes}{\begin{array}{c}twr(t_1)\otimes twr(A)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),A))\otimes hn,\\ twr(a_2),\\ twr(t_1),\\ hold(b_1)\end{array} \vdash twr(empty)\otimes twr(app(rev(b_1 :: t_1),a_2))\otimes hn}\,lcut(put)}{\begin{array}{c}twr(t_1)\otimes twr(A)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),A))\otimes hn,\\ twr(a_2),\\ twr(t_1)\otimes hold(b_1)\end{array} \vdash twr(empty)\otimes twr(app(rev(b_1 :: t_1),a_2))\otimes hn}\,l\otimes}{\begin{array}{c}twr(t_1)\otimes twr(A)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),A))\otimes hn,\\ twr(b_1 :: t_1),\\ twr(a_2),\\ hn\end{array} \vdash twr(empty)\otimes twr(app(rev(b_1 :: t_1),a_2))\otimes hn}\,lcut(pick)}{\begin{array}{c}twr(t_1)\otimes twr(A)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),A))\otimes hn,\\ twr(b_1 :: t_1)\otimes twr(a_2)\otimes hn\end{array} \vdash twr(empty)\otimes twr(app(rev(b_1 :: t_1),a_2))\otimes hn}\,l\otimes,l\otimes}{\begin{array}{c}twr(t_1)\otimes twr(A)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),A))\otimes hn\end{array} \vdash \begin{array}{c}twr(b_1 :: t_1)\otimes twr(a_2)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(b_1 :: t_1),a_2))\otimes hn\end{array}}\,r\multimap}{\forall a.\left[\begin{array}{c}twr(t_1)\otimes twr(a)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(t_1),a))\otimes hn\end{array}\right] \vdash \forall a.\left[\begin{array}{c}twr(b_1 :: t_1)\otimes twr(a)\otimes hn \multimap \\ twr(empty)\otimes twr(app(rev(b_1 :: t_1),a))\otimes hn\end{array}\right]}\,r\forall,l\forall$$

$$\langle step\rangle$$

The subproof $\langle 1\rangle$ deals with satisfying the preconditions of the recursive call. In doing so, the meta-variable A is instantiated to $b_1 :: a_2$.

$$\dfrac{\dfrac{\dfrac{}{twr(t_1)\vdash twr(t_1)}\,Ax \qquad \dfrac{\dfrac{}{twr(A)\vdash twr(b_1 :: a_2)}\,Ax,A=b_1::a_2 \qquad \dfrac{}{hn\vdash hn}\,Ax}{twr(A),hn\vdash twr(b_1 :: a_2)\otimes hn}\,r\otimes}{twr(t_1),twr(A),hn\vdash twr(t_1)\otimes twr(b_1 :: a_2)\otimes hn}\,r\otimes}{\langle 1\rangle}$$

The subproof $\langle 2\rangle$ takes us from the postconditions of the recursive call to the post-conditions of the step case. In this case, this part involves no applications of actions, which results in a tail-recursive plan. In this case, this was not achieved by forcing such a restriction, as described in Section 6.3.3, though that approach could have been used if it was specifically required that the plan must be tail recursive.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{twr(empty)\otimes twr(app(rev(t_1),b_1::a_2))\otimes hn}\,Ax}{\vdash}}{twr(empty)\otimes twr(app(rev(t_1),b_1::a_2))\otimes hn}}{twr(empty)\otimes twr(app(rev(t_1),b_1::a_2))\otimes hn}\,rrewrite(app1)}{\vdash}}{twr(empty)\otimes twr(app(rev(t_1),b_1::app(empty,a_2)))\otimes hn}}{twr(empty)\otimes twr(app(rev(t_1),b_1::a_2))\otimes hn}\,rrewrite(app2)}{\vdash}}{twr(empty)\otimes twr(app(rev(t_1),app(b_1::empty,a_2)))\otimes hn}}{twr(empty)\otimes twr(app(rev(t_1),b_1::a_2))\otimes hn}\,rrewrite(appassoc)}{\vdash}}{twr(empty)\otimes twr(app(app(rev(t_1),b_1::empty),a_2))\otimes hn}}{twr(empty)\otimes twr(app(rev(t_1),b_1::a_2))\otimes hn}\,rrewrite(rev2)}{\vdash}}{twr(empty)\otimes twr(app(rev(b_1::t_1),a_2))\otimes hn}}{\langle 2\rangle}$$

The plan can now be given as (after application of some simplification rules):

```
λt_2.
  twr_rec(t_2,
    λa_2.λh3.
        let h3 be h6*h7 in
          let h7 be h10*h11 in h6*h10*h11,
    λh1.λb_1.λt_1.λa_8.λh60.
              let h60 be h63*h64 in
                let h64 be h67*h68 in
                  let pick∘h63∘h68 be h75*h76 in
                    let put∘h67∘h76 be h92*h93 in
                      let h1∘b_1::a_8∘h75*h92*h93 be h97*h98 in
                        let h98 be h101*h102 in h97*h101*h102)
```

Plans like this one can be produced automatically by the Lino system (Chapter 8).

6.4.2 Example partial evaluation

Now we can show how this general plan can be partially evaluated to create a specific plan for a given initial state. In this example, the `b1::b2::b3::empty` represents a three-block tower, and `p*q*r` represent the labels of a conjunction of literals in the initial state.

```
Plan∘b1::b2::b3::empty∘empty∘p*q*r
```

When evaluated with the data available, the plan reduces to a form that can be executed a simple sequence of actions:

```
let pick∘p∘r be h107*h108 in
  let put∘q∘h108 be h109*h110 in
    let pick∘h107∘h110 be h123*h124 in
      let put∘h109∘h124 be h125*h126 in
        let pick∘h123∘h126 be h139*h140 in
          h139*put∘h125∘h140
```

6.4.3 Example with sets represented as lists

It is useful in describing many planning problems to handle finite sets. Unfortunately the axioms stating that order and duplication in sets are irrelevant cause some trouble. With *in* as the set constructor, these axioms can be stated for elements d and e in a set s as follows:

$$\begin{aligned} in(d, (in(d, s))) &= in(d, s) \\ in(d, in(e, s)) &= in(e, in(d, s)) \end{aligned}$$

To properly handle finite sets, it would be necessary to show that proofs are valid with respect to these axioms. Here we will do without these axioms and only simulate sets by the use of lists. This simply means that we impose an arbitrary order on the elements.

The following example makes use of lists of towers, which are themselves lists of blocks. The problem is to dismantle a tower to form a list of towers, each containing only a single block. To define this example, we define a a function *flatten*, which is used to the state the relation between the tower initially present, and its final state as a list of towers of blocks.

$flatten$ can be defined as:

$$flatten(h :: t) = (h :: empty) :: flatten(t) \tag{6.14}$$

$$flatten(empty) = nil \tag{6.15}$$

The problem specification is:

$$\forall t.[twr(t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(t))]$$

The following actions will be used:

$$\begin{array}{lll} \vdash \texttt{lop(h)}: & twr(h :: t) & \multimap \; twr(t) \otimes twr(h :: empty) \\ \vdash \texttt{gather(t,l)}: & twr(t) \otimes lst(l) & \multimap \; lst(t :: l) \end{array}$$

Where `lop` is an action which creates a new tower from a block removed from the top of t. `gather` adds a tower into a list of towers.

We need to be able to create and to destroy empty towers. There are several ways that we could go about doing this.

The approach we will take here is to add the necessary $twr(empty)$ resources to the specification.

Base case:

$$\dfrac{\dfrac{\dfrac{}{twr(empty) \otimes lst(nil) \vdash twr(empty) \otimes lst(nil)}\,Ax}{\vdash twr(empty) \otimes lst(nil) \multimap twr(empty) \otimes lst(nil)}\,r\multimap}{\vdash twr(empty) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(empty))}\,6.15$$

Step case:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{twr(empty) \otimes lst(flatten(h :: t)) \vdash twr(empty) \otimes lst(flatten(h :: t))}\,Ax}{twr(empty) \otimes lst((h :: empty) :: flatten(t)) \vdash twr(empty) \otimes lst(flatten(h :: t))}\,6.14}{twr(empty) \otimes lst(flatten(t)), twr(h :: empty) \vdash twr(empty) \otimes lst(flatten(h :: t))}\,lcut(gather)}{twr(t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(t)),\ \vdash twr(empty) \otimes lst(flatten(h :: t))}\,l\multimap}{\begin{array}{l} twr(t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(t)), \\ twr(t) \otimes twr(h :: empty), lst(nil) \end{array} \vdash twr(empty) \otimes lst(flatten(h :: t))}\,l\otimes}{\begin{array}{l} twr(t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(t)), \\ twr(h :: t), lst(nil) \end{array} \vdash twr(empty) \otimes lst(flatten(h :: t))}\,lcut(lop)}{\begin{array}{l} twr(t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(t)), \\ twr(h :: t) \otimes lst(nil) \end{array} \vdash twr(empty) \otimes lst(flatten(h :: t))}\,l\otimes}{\begin{array}{c} twr(t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(t)) \\ \vdash \\ twr(h :: t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flatten(h :: t)) \end{array}}\,r\multimap$$

The recursive plan extracted is:

```
λt_2.
  twr_rec(t_2,
    λh2.h2,
    λh1.λb_1.λt_1.λh3.
            let h3 be h4*h5 in
              let lop∘h4 be h7*h8 in
                let h1∘h7*h5 be h10*h11 in h10*(gather∘h8*h11))
```

6.4.4 Example using trees

In this section we give an example of induction on a binary tree datastructure. Suppose we have available actions `goright` and `goleft` to progress down either branch of the tree, and a test which allows us to expand the recursive definition of *intree* determine which branch to proceed down.

$$\begin{array}{lrcl}
\vdash \texttt{goleft}: & tree(node(l,r,v)) & \multimap & tree(l) \\
\vdash \texttt{goright}: & tree(node(l,r,v)) & \multimap & tree(r) \\
\vdash \texttt{testtree1}: & intree(b, empty) & \multimap & \mathbf{0} \\
\vdash \texttt{testtree2}: & intree(b, node(l,r,v)) & \multimap & (b = v) \oplus intree(b,l) \oplus intree(b,r)
\end{array}$$

The `testtree1` axiom specifies that $intree(b, empty)$ can never occur. If a matching term appears in a proof, the rule allows $\mathbf{0}$ to be derived, which signals an absurd state from which anything can be derived (see Section 5.2.6).

We can then create a proof and extract a plan which navigates the tree. The assumption that we can resolve the $\oplus$ is effectively an assumption that when the plan is executed, we we will always know which subtree to follow at each step.

We attempt to the prove the goal:

$$\forall t.\forall a.tree(t) \otimes intree(a,t) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top$$

First we apply the tree induction rule on t.

$$\frac{\langle base \rangle \quad \langle step \rangle}{\vdash \forall t.\forall a.tree(t) \otimes intree(a,t) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top}\ tree\_ind(t)$$

Base case

We deal with the base case showing that it assumes an absurdity — i.e. that a is in the empty tree. We can use the action `testtree1` to introduce the $\mathbf{0}$, which allows us to prove anything. From the point of view of the proof, this corresponds to showing that the situation will not occur. The corresponding plan term fails to execute.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{tree(empty), \mathbf{0} \vdash (\exists x.\exists y.tree(node(x,y,a_1))) \otimes \top}\ l\mathbf{0}}{tree(empty), intree(a_1, empty) \vdash (\exists x.\exists y.tree(node(x,y,a_1))) \otimes \top}\ lcut(testtree1)}{tree(empty) \otimes intree(a_1, empty) \vdash (\exists x.\exists y.tree(node(x,y,a_1))) \otimes \top}\ l\otimes}{\vdash tree(empty) \otimes intree(a_1, empty) \multimap (\exists x.\exists y.tree(node(x,y,a_1))) \otimes \top}\ r\multimap}{\vdash \forall a.tree(empty) \otimes intree(a, empty) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top}\ r\forall}{\langle base \rangle}$$

Step case

The proof for the step case proceeds by expanding the definition of *intree* and doing a case split into three possibilities: ⟨1⟩ the current node has value a, ⟨2⟩ the value a is in the left subtree, or ⟨3⟩ the value is in the right subtree.

$$\dfrac{\dfrac{\langle 1\rangle \quad \dfrac{\langle 2\rangle \quad \langle 3\rangle}{H_1, H_2, tree(node(l_1,r_1,v_1)), intree(a_2,l_1) \oplus intree(a_2,r_1) \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\, l\oplus}{H_1, H_2, tree(node(l_1,r_1,v_1)), \left[\begin{array}{c}(a_2 = v_1)\\ \oplus \\ intree(a_2,l_1)\\ \oplus \\ intree(a_2,r_1)\end{array}\right] \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\, l\oplus}{H_1, H_2, tree(node(l_1,r_1,v_1)), intree(a_2,node(l_1,r_1,v_1)) \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\, lcut(testtree2)$$

$$\dfrac{H_1, H_2, tree(node(l_1,r_1,v_1)) \otimes intree(a_2,node(l_1,r_1,v_1)) \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}{H_1, H_2 \vdash tree(node(l_1,r_1,v_1)) \otimes intree(a_2,node(l_1,r_1,v_1)) \multimap (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\, r\multimap$$

$$\dfrac{H_1, H_2 \vdash tree(node(l_1,r_1,v_1)) \otimes intree(a,node(l_1,r_1,v_1)) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top}{\langle step\rangle}\, r\forall$$

Where H_1 and H_2 are the two induction hypotheses:

$$H_1 \equiv \forall a.tree(l_1) \otimes intree(a,l_1) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top$$
$$H_2 \equiv \forall a.tree(r_1) \otimes intree(a,r_1) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top$$

In case ⟨1⟩ the equality allows us to make a substitution and we need only show that the value has been found. The presence of $\top$ allows us to dispose of the unwanted hypotheses.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{tree(node(l_1,r_1,v_1)) \vdash tree(node(l_1,r_1,v_1))}\, Ax}{tree(node(l_1,r_1,v_1)) \vdash \exists y.tree(node(l_1,y,v_1))}\, r\exists}{tree(node(l_1,r_1,v_1)) \vdash \exists x.\exists y.tree(node(x,y,v_1))}\, r\exists \quad \dfrac{}{H_1, H_2 \vdash \top}\, r\top}{H_1, H_2, tree(node(l_1,r_1,v_1)) \vdash (\exists x.\exists y.tree(node(x,y,v_1))) \otimes \top}\, r\otimes}{H_1, H_2, tree(node(l_1,r_1,v_1)), a_2 = v_1 \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\, substitute$$

$$\langle 1\rangle$$

In case ⟨2⟩, we have assumed that the value lies in the left subtree, so we can apply an action `goleft` to move down the left subtree. We can then use the $l\multimap$ rule to apply one of the induction hypotheses (i.e. make the recursive call) to reach the goal. Proof of ⟨3⟩ is omitted, since it is similar to ⟨2⟩, using the right subtree instead of the left.

$$\dfrac{\dfrac{\dfrac{}{tree(l_1) \vdash tree(l_1)}\,Ax \quad \dfrac{}{intree(a_2,l_1) \vdash intree(a_2,l_1)}\,Ax}{intree(a_2,l_1), tree(l_1) \vdash tree(l_1) \otimes intree(a_2,l_1)}\,r\otimes \quad \dfrac{\dfrac{\dfrac{}{\exists x.\exists y.tree(node(x,y,a_2)) \vdash \exists x.\exists y.tree(node(x,y,a_2))}\,Ax \quad \dfrac{}{H_2, \top \vdash \top}\,r\top}{H_2, \exists x.\exists y.tree(node(x,y,a_2)), \top \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\,r\otimes}{H_2, (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\,l\otimes}{\begin{array}{c} tree(l_1) \otimes intree(a_2,l_1) \multimap (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top, \\ H_2, \\ intree(a_2,l_1), \\ tree(l_1) \end{array} \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\,l\multimap$$

$$\dfrac{\begin{array}{c} \forall a.tree(l_1) \otimes intree(a,l_1) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top, \\ H_2, \\ intree(a_2,l_1), \\ tree(l_1) \end{array} \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}{\begin{array}{c} \forall a.tree(l_1) \otimes intree(a,l_1) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top, \\ H_2, \\ intree(a_2,l_1), \\ tree(l_1) \end{array} \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}\,l\forall$$

$$\dfrac{\begin{array}{c} \forall a.tree(l_1) \otimes intree(a,l_1) \multimap (\exists x.\exists y.tree(node(x,y,a))) \otimes \top, \\ H_2, \\ tree(node(l_1,r_1,v_1)), \\ intree(a_2,l_1) \end{array} \vdash (\exists x.\exists y.tree(node(x,y,a_2))) \otimes \top}{\langle 2 \rangle}\,cut(goleft)$$

The plan

From the proof we can extract the plan:

```
λt.
  tree_rec(t_2,
    λa_1.λh3.
        let h3 be h4*h5 in abort(apply(testtree1,h5)),
    λh1.λh2.λl_1.λr_1.λa_2.λh7.
                let h7 be h8*h9 in
                  case testtree2∘h9 of
                    inl(h11) then h8*erase
                    inr(h12) then
                      case h12 of
                        inl(h13) then
                          let h1∘a_2∘apply(goleft,h8)*h13
                            be h18*h19
                            in h18*erase
                        inr(h14) then
                          let h2∘a_2∘apply(goright,h8)*h14
                            be h23*h24
                            in h23*erase)
```

6.4.5 Example: Nim

Here we consider the game of Nim. This is a two-player game for which a winning strategy exists for one of the players. We show that a proof can be built of the winning strategy in the linear logic framework. The example involves the use of recursion and of actions with uncertain effects.

The rules

There are various forms of the game of Nim. Here we consider a simple version. At the start of the game there is a row of 21 counters. Players take turns to remove 1, 2 or 3 counters from the row. The aim of the game is to avoid taking the last counter.

The winning strategy

Either player can find themselves in one of only 21 possible states. It is fairly easy to categorise the states into winning and losing positions.

We can see our goal state as leaving the opponent with only a single counter. Reasoning backwards from the goal state, it is clear that this state is accessible if there are 2, 3 or 4 counters remaining at our turn. Furthermore, we can guarantee to reach this state if the opponent was left with 5 counters at the previous turn.

Similarly, every fourth position from 1 is also a losing position. In the 21-counters version of the game, the player to take the first move will always be the loser if the opponent plays perfectly. The winning and losing positions for 21 down to 0 counters are shown below.

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W	L	W

Solution to a fixed-size problem

For the problem with fixed size, the solution branches on the possible choices of move from the opponent. At our turn, we always play so as to leave the opponent in state where the number of remaining counters is $4n + 1$, for some n. So after branching on the opponent's move, our move brings us back to the same state in each branch. Hence the proof tree contains repeated subtrees (below nodes 7, 20 and 32).

In the 9-counter problem below, we attempt to prove:

$$\vdash them \otimes s(s(s(s(s(s(s(s(s(zero))))))))) \multimap them \otimes s(zero)$$

Fig. 6.1 gives the structure of a proof of the above specification.

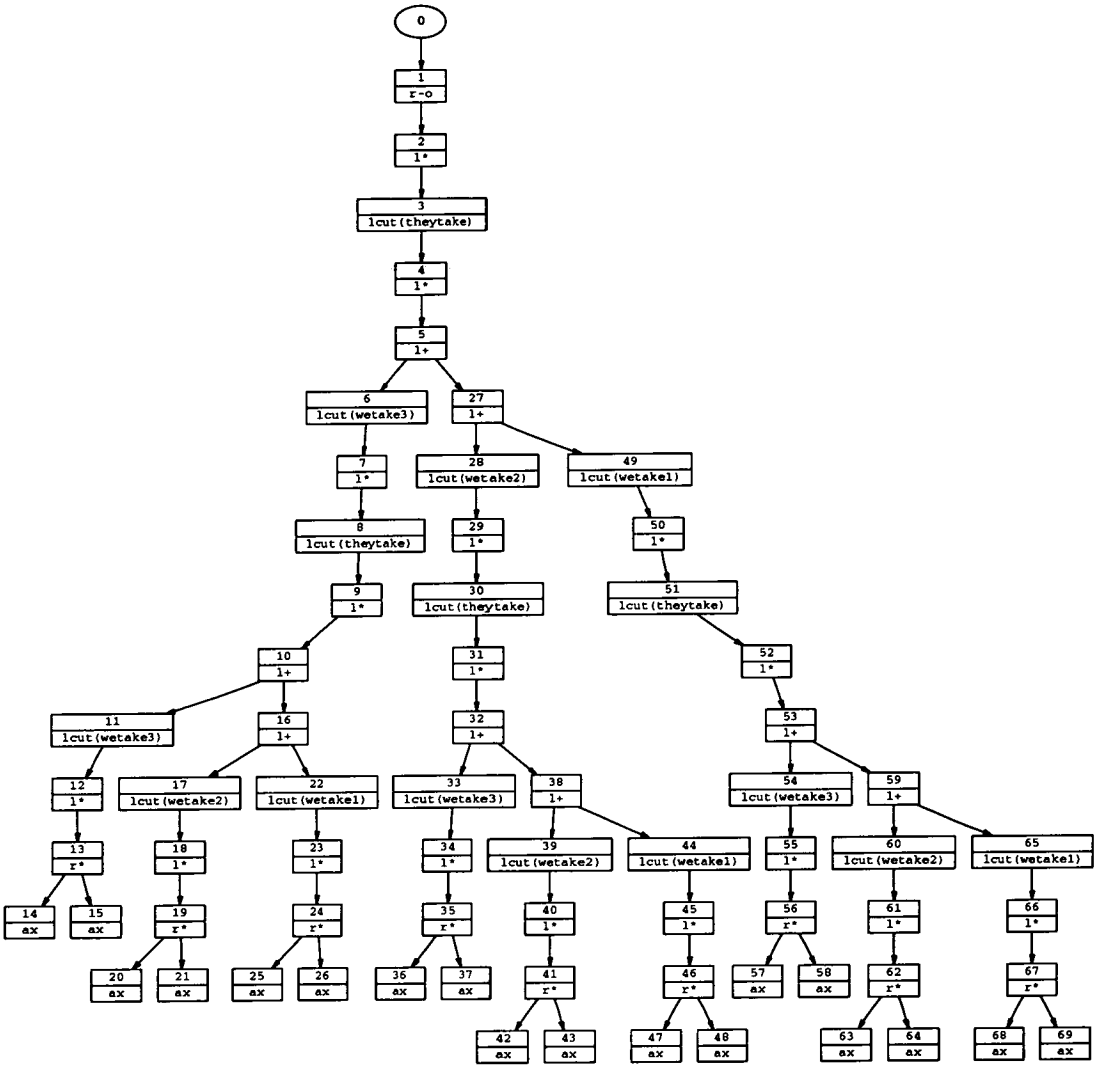


Figure 6.1: Proof tree for 9-counter Nim game.

This leads to a long plan which branches on every possible move by the opponent.

Inductive solution

Here we consider the solution for a more general problem. The winning strategy only works for problems with $4n + 1$ counters, for any n. We can write this goal in linear logic as:

$$\vdash \forall n.them \otimes s(mult(n, s(s(s(s(zero)))))) \multimap them \otimes s(zero)$$

In order to solve this problem, our prover needs to know about multiplication and

addition. These appear as the rewrite rules for *mult* and *plus*.

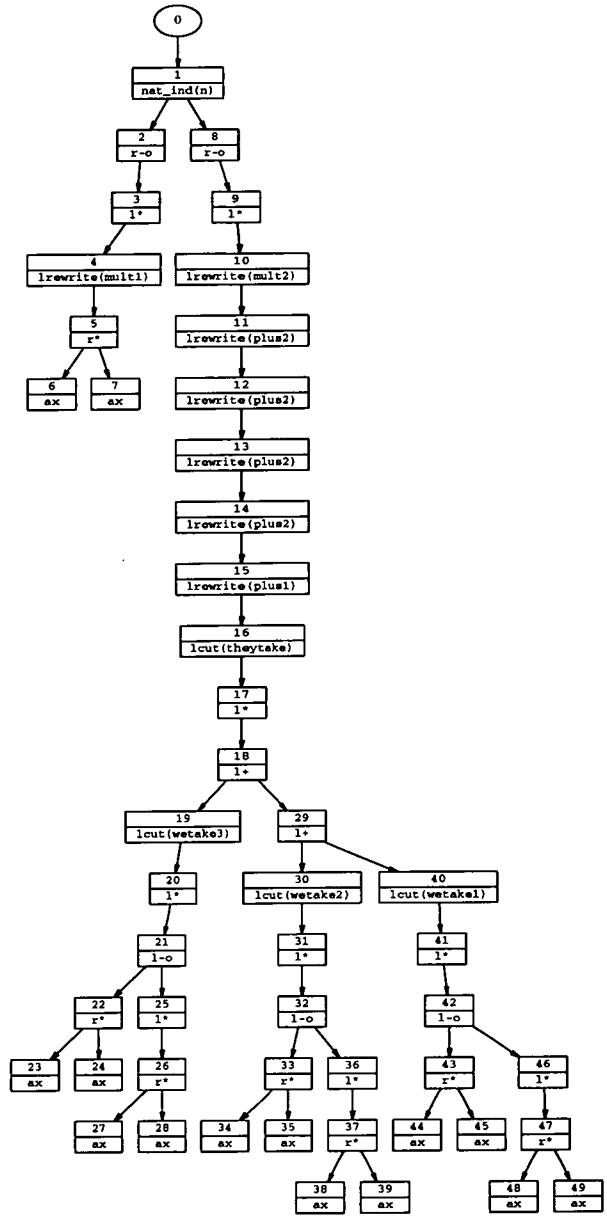


Figure 6.2: Proof tree for Nim game with $4n + 1$ counters.

This gives the following plan:

```
λn_2.
  nat_rec(n_2,
    λh3.
      let h3 be h6*h7 in h6*h7,
    λh1.λn_1.λh57.
          let h57 be h60*h61 in
            let theytake∘h60∘h61 be h66*h67 in
              case h67 of
                inl(h70) then
                  let wetake3∘h66∘h70 be h97*h98 in
                    let h1∘h97*h98 be h102*h103 in h102*h103
                inr(h71) then
                  case h71 of
                    inl(h106) then
                      let wetake2∘h66∘h106 be h118*h119 in
                        let h1∘h118*h119 be h123*h124 in h123*h124
                    inr(h107) then
                      let wetake1∘h66∘h107 be h129*h130 in
                        let h1∘h129*h130 be h134*h135 in h134*h135)
```

Note that this plan is much shorter than that generated specifically for the 9-counter game. It is also more general, as it applies to games for any choice of n. If we applied the partial evaluation rules to the recursive plan with the value $n = s(s(0))$, we would generate the same plan as for the 9-counter case.

6.5 Summary

This chapter has shown how inductive proofs can be used to synthesise recursive plans. We looked at the inductive datatypes of towers, natural numbers and binary trees and gave induction rules for each of these, defining also the formation of plan terms.

We have also defined how to partially evaluate and execute these recursive plans.

We have given various example planning problems specified using inductive datatypes, and shown solution as proofs and corresponding extracted plans.

Our examples have also demonstrated how to use function symbols in specifying planning problems and how to avoid giving a complete specification of a goal state.

Chapter 7

Automated Proof Search

7.1 Introduction

So far, we have considered the formalism for representing planning problems and their solutions (proofs). We have not said anything about how to find such proofs automatically. Here we consider previous approaches to proof search in linear logic. We then consider the issue of forward versus backward chaining when constructing proofs in ILL to solve planning problems, and find that forward chaining has advantages.

We then present a forward-chaining strategy which is complete for the fragment involving only the connectives $\multimap$, $\otimes$ and $\oplus$. However, this fragment does not include all the connectives that we need in order to handle all the planning problems which we want to be able to handle.

We then present a strategy which allows a carefully defined larger fragment of the logic to be used, but which is not complete.

Finally, the strategy for the application of rewrite rules in proof search is considered.

7.2 Search in linear logic

In this section we consider three approaches to proof search in linear logic — Jacopin's CSLL algorithm, connection-based approaches, and linear logic programming.

These approaches differ strongly in the following ways:

- The fragment of linear logic tackled, e.g. multiplicatives, additives, exponentials, quantifiers.
- The approach to context splitting and redundancy in proofs caused by permutability of rules.
- Whether emphasis is on representation of proof or automation of search.
- Applicability to planning problems.

7.2.1 Jacopin's CSLL algorithm

Jacopin's approach is a search algorithm to realise Masseron's proof system for constructing proofs of plans (called *formal actions* by Masseron) in linear logic.

Jacopin restricts proof search to use only the decidable fragment involving the rules, ax, $l\otimes$, $r\otimes$ and *cut*. The application of *cut* is restricted its use in applications of the transition axioms on the left side. The left side of the transition axioms (i.e. the action preconditions) is matched within the left side of open sequent in the proof (i.e. the current state). This effectively makes the search equivalent to a total-order planner using forward state-space search.

Plan extraction then uses a method which is based on Masseron's geometric approach.

The full algorithm from [Jacopin 93] is given below. The algorithm uses the following arguments:

- Axioms — set of axioms describing available transitions.
- E_i, E_f — Antecedent and Succedent of sequent, describing initial and final states.
- P — Proof constructed by CSLL.

$\sqsubseteq$ denotes multiset inclusion.

$\sqcup$ denotes multiset union.

(A, C) denotes a sequent $A \vdash C$

CSLL(Axioms,E_i,E_f,P)
If $E_i \vdash E_f$ is the identity Then
return P
Else
Choose (A_{as}, C_{as}) from Axioms such that $A_{as} \sqsubseteq E_i$
If $(E_i = A_{as}) \wedge (E_f = C_{as})$ Then
return P
Else
$(A_{as}, C_{as}) = (\Gamma, B)$
$(E_i, E_f) = (\Gamma \sqcup \Gamma', C)$
$\mathrm{P} \longleftarrow \mathrm{P} \cup \left\{ \dfrac{\Gamma \vdash B \quad \Gamma', B \vdash C}{\Gamma, \Gamma' \vdash C}\ cut \right\}$
$(A_1, C_1) \longleftarrow (\Gamma' \sqcup \{B\})$
While $(A_1, C_1) = (\Gamma' \cup \{B' \otimes B''\}, C)$ Do
$\mathrm{P} \longleftarrow \mathrm{P} \cup \left\{ \dfrac{\Gamma', B', B'' \vdash C}{\Gamma', B' \otimes B'' \vdash C}\ l\otimes \right\}$
$(A_1, C_1) \longleftarrow (\Gamma' \sqcup \{B', B''\}, C)$
End While
$(A_2, C_2) \longleftarrow (A_1, C_1)$
While $(A_2, C_2) = (C' \sqcup \Delta, C' \otimes C'')$ Do
Choose non-deterministically:
1. $\mathrm{P} \longleftarrow \mathrm{P} \cup \left\{ \dfrac{C' \vdash C' \quad \Delta \vdash C''}{C', \Delta \vdash C' \otimes C''}\ r\otimes \right\}$
$(A_2, C_2) \longleftarrow (\Delta, C'')$
2. Exit While
End While
$(A_3, C_3) \longleftarrow (A_2, C_2)$
CSLL(Axioms,A_3,C_3,P)
End If
End If

The algorithm applies rules with the following priority:

1. Identity axiom, Ax.
2. Exact match with a transition axiom.
3. Cut of a transition axiom (forward application of action),
and do the following to the resulting sequent:
    - Exhaustively apply $l\otimes$.
    - Nondeterministically choose whether to apply $r\otimes$.
    - Recursively call CSLL on resulting sequent.

Non-deterministic application of $r\otimes$ rule often causes redundant search. This problem has been addressed in our work by deferring application of $r\otimes$ until late in the proof, and adopting a lazy context splitting mechanism.

Jacopin's conclusions on linear logic are mainly pessimistic. One reason for this could be because the fragment used is very restrictive, yet the search algorithm has an unnecessary choice point into the search at the point of context splitting. Hence there is no advantage over a conventional planner in expressiveness or performance. In Jacopin's formulation it is necessary to fully specify the goal state of planning problems. However, we have already seen that the $\top$ operator this problem.

7.2.2 Connection-based methods

In this section we do not discuss a specific algorithm, but a family of algorithms which are based on a similar representation. Connection-based proof search has been used in planning problems since it was introduced by Bibel [Bibel 86]. This approach was not designed as a linear logic theorem proving method (it predates linear logic) but derived from a FOPC prover by engineering resource sensitivity into the representation used in proof search.

We take a brief look at Bibel's representation, and consider subsequent work that has used similar representations for linear logic theorem provers. The main advantage is the elimination of redundancies in the proof search space caused by permutabilities of sequent deduction rules. The method is based on making connections between positive and negative instances of formulae in the proof. In the planning problem, these correspond to effects and to preconditions and goals. The resource-sensitivity is enforced by imposing the restriction that each formula is connected at most once.

Bibel's LCM

We give a brief sketch example from Bibel's original paper on the Linear Connection Method (LCM) [Bibel 86]. Note that Bibel does not formalise a resource sensitive logic, but only adds a resource-sensitivity into an existing representation used in constructing proofs [Bibel 83]. Note also that Bibel's claim was that making a minor change to an

existing proof method (i.e. adding resource-sensitivity) allowed planning problems to be solved without changing the logic or adding frame axioms.

Initial situation:

$ontable(a) \wedge on(b,a) \wedge clear(b) \wedge handempty$

Goal situation:

$ontable(a) \wedge clear(a) \wedge hold(b)$

Description of action *pick*:

antecedent: $clear(x) \wedge on(x,y) \wedge handempty$

consequent: $hold(x) \wedge clear(y)$

We adhere to Bibel's notation as follows:

T	for ontable
O	for on
C	for clear
E	for handempty
H	hold

The problem is written in the logic as follows:

$$Ta \wedge Oba \wedge Cb \wedge E \wedge \forall xy(Cx \wedge Oxy \wedge E \rightarrow Hx \wedge Cy) \rightarrow Ta \wedge Ca \wedge Hb$$

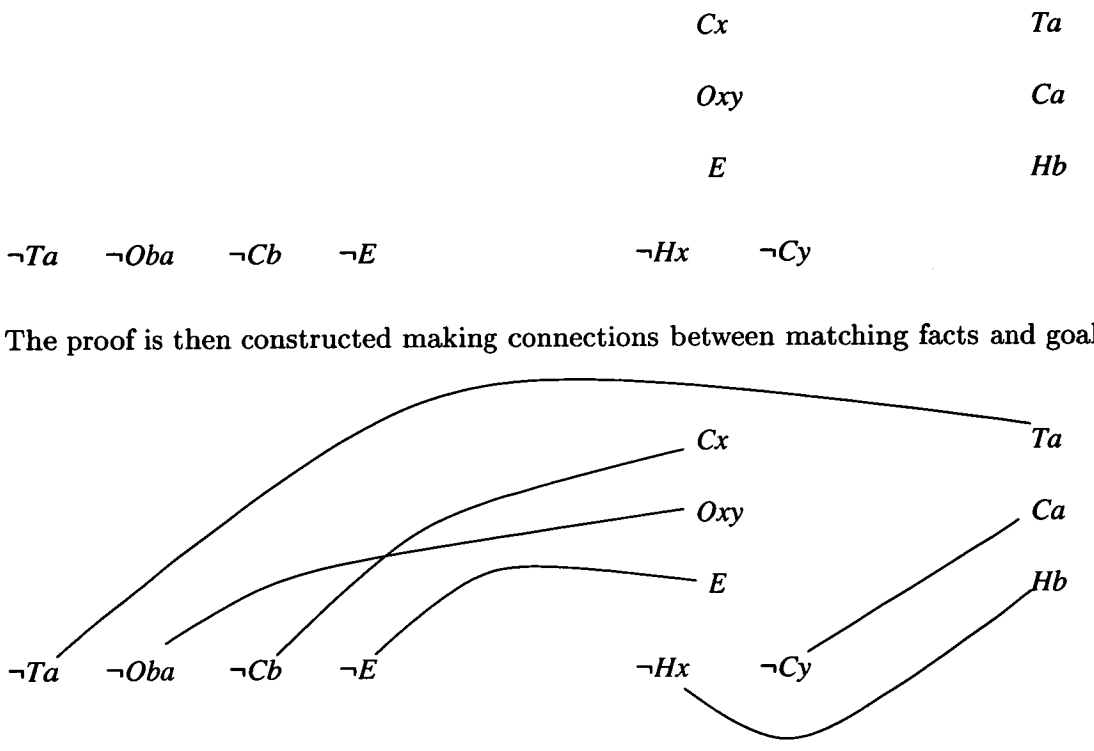
It is then converted into AND/OR form:

$$\neg Ta \vee \neg Oba \vee \neg Cb \vee \neg E \vee \exists xy(Cx \wedge Oxy \wedge E \wedge (\neg Hx \vee \neg Cy)) \vee Ta \wedge Ca \wedge Hb$$

Universal variables are skolemised and existential variables are replaced by free variables.

$$\neg Ta \vee \neg Oba \vee \neg Cb \vee \neg E \vee (Cx \wedge Oxy \wedge E \wedge (\neg Hx \vee \neg Cy)) \vee Ta \wedge Ca \wedge Hb$$

The problem is laid out as a matrix with the disjunctions laid out horizontally and the conjunctions laid out vertically.



The proof is then constructed making connections between matching facts and goals.

Completing the proof involves finding appropriate connections in the matrix, and also determining appropriate substitutions. The linearity restriction requires that each literal may be connected with at most one other literal.

The original paper does not give an algorithm, but Bibel used an adaptation of a proof search without the linearity assumption. Later publications refer to a goal-driven linear backward-chaining (LBC) algorithm, and to LIP$^+$ algorithm [Fronhöfer 97], which is a close relative of partial-order causal link planning.

Connection proofs and linear logic

The resource-sensitive nature of Bibel's method makes it sensible to relate it to linear logic. The equivalence to linear logic, at least for simple conjunctive problems has been shown by [Große *et al* 96]. This relates both approaches (and also another one based on equational resolution) to a common semantics. It is worth noting that since sequents are not handled directly, the context-splitting issue does not arise.

[Kreitz *et al* 96] presents a connection-based prover for multiplicative and exponential fragment of classical linear logic. This is based on the linear connection idea, and on

connection-matrix formulations for other logics [Wallen 90]. This formulation removes certain redundancies from the proof search space. There is also a strong relationship to Girard's notion of representing linear logic proofs as proof nets [Girard 95]. Proof nets behave very well for the multiplicative fragment, but need awkward extensions to deal with exponentials and additives. Even simple conjunctive planning problems cannot be handled by (multiplicative) proof nets, as we would need to know in advance how many times each action transition would be used.

[Brüning *et al* 93] presents an adaptation of a connection method which effectively adds handling for additive disjunction.

[Fronhöfer 97] provides a survey of research descendents of the linear connection method.

7.2.3 Linear logic programming

In logic programming, proof search itself is seen as a form of computation. Hence, logic programming languages perform systematic proof search, and that is why they are of interest to us.

Several formulations of Prolog-like languages for linear logic have been devised. Two which have given rise to practical implementations are *Lolli* [Hodas & Miller 94] and *Lygon* [Harland *et al* 96]. Here, we shall give an account of the principles behind Lolli, which is a summary of [Hodas & Miller 94].

Logic programming can be characterised as a declarative proof system that can be given an executable semantics in the style of [Miller *et al* 91].

Hodas and Miller also require that proofs should be goal directed, which is given a formal definition using the notion of *uniform proofs*. In a uniform proof, the right-introduction rules are applied to break down goals to atomic formulas before any left-introduction rules are applied. Since, in this setting, the left side of the sequent represents program and data, this means that goals are processed uniformly and independently from the program.

Lolli

The logic behind Lolli is derived by the following steps:

1. Define a fragment of intuitionistic linear logic for which uniform proofs are possible ($\mathcal{L}$).
2. Replace all the left-introduction rules with a single backchaining rule, giving $\mathcal{L}'$.
3. Extend the fragment by allowing missing connectives to appear in goals only ($\mathcal{L}''$).

Additionally, Hodas and Miller describe an approach to resource management which lazily handles the choice of how to split the context when applying rules such as $r\otimes$.

Proof system $\mathcal{L}$

In this section, we will consider a fragment of intuitionistic linear logic using only $\top, \&, \otimes, !, \multimap$, and $\forall$. In fact, even $\otimes$ and ! cause some difficulties, because uniform proofs involving these connectives may not be possible. For example $a \otimes b \vdash b \otimes a$ and $!a \vdash !a \otimes !a$ have proofs, but not uniform proofs, since both require a left rule to be applied before a right rule. We will ignore ! and $\otimes$ rules for the time being, though ! is handled in a limited way by the form of sequents that we use.

Consider sequents of the form: $\Gamma; \Delta \vdash B$ where B is a formula, Γ is set of formulas and Δ is a multiset of formulas. Such sequents have their context divided into an unbounded part Γ (a set) and a bounded part Δ, which is a multiset corresponding to left-hand side of sequents of the purely linear fragment.

The sequent

$$B_1, ..., B_n; C_1, ..., C_m \vdash B$$

is equivalent to the linear logic sequent

$$!B_1, ..., !B_n, C_1, ..., C_m \vdash B$$

It is now natural to make a second modification to linear logic by introducing a second

kind of implication. We will use intuitionistic implication $B \Rightarrow C$, which is directly equivalent to $!B \multimap C$.

The system $\mathcal{L}$, with these restrictions, is defined by the following proof rules:

$$\frac{\Gamma, B; \Delta, B \vdash C}{\Gamma, B; \Delta \vdash C}\ absorb$$

$$\frac{}{\Gamma; A \vdash A}\ identity \qquad \frac{}{\Gamma; \Delta \vdash \top}\ r\top$$

$$\frac{\Gamma; \Delta, B_i \vdash C}{\Gamma; \Delta, B_1 \& B_2 \vdash C}\ l\& \qquad \frac{\Gamma; \Delta \vdash B \quad \Gamma; \Delta \vdash C}{\Gamma; \Delta \vdash B \& C}\ r\&$$

$$\frac{\Gamma; \Delta_1 \vdash B \quad \Gamma; \Delta_2, C \vdash E}{\Gamma; \Delta_1, \Delta_2, B \multimap C \vdash E}\ l\multimap \qquad \frac{\Gamma; \Delta, B \vdash C}{\Gamma; \Delta \vdash B \multimap C}\ r\multimap$$

$$\frac{\Gamma; \emptyset \vdash B \quad \Gamma; \Delta, C \vdash E}{\Gamma; \Delta, B \Rightarrow C \vdash E}\ l\Rightarrow \qquad \frac{\Gamma, B; \Delta \vdash C}{\Gamma; \Delta \vdash B \Rightarrow C}\ r\Rightarrow$$

$$\frac{\Gamma; \Delta, B[t/x] \vdash C}{\Gamma; \Delta, \forall x B \vdash C}\ l\forall \qquad \frac{\Gamma; \Delta \vdash B[y/x]}{\Gamma; \Delta \vdash \forall x B}\ r\forall$$

Provided that y is not free in the lower sequent.

The proof system $\mathcal{L}'$

The various left-introduction rules can now be combined into a single backchaining rule. This is done by decomposing a linear formula B to define a set $\|B\|$ of triples $\langle \Gamma, \Delta, B' \rangle$ as follows:

1. $\langle \emptyset, \emptyset, B \rangle \in \|B\|$,
2. if $\langle \Gamma, \Delta, B_1 \& B_2 \rangle \in \|B\|$ then both $\langle \Gamma, \Delta, B_1 \rangle \in \|B\|$ and $\langle \Gamma, \Delta, B_2 \rangle \in \|B\|$
3. if $\langle \Gamma.\Delta, \forall x, B' \rangle \in \|B\|$, then for all closed terms t, $\langle \Gamma, \Delta, B'[t/x] \rangle \in \|B\|$,
4. if $\langle \Gamma, \Delta, B_1 \Rightarrow B_2 \rangle \in \|B\|$ then $\langle \Gamma \cup \{B\}, \Delta, B_2 \rangle \in \|B\|$ and

5. if $\langle \Gamma, \Delta, B_1 \multimap B_2 \rangle \in \|B\|$ then $\langle \Gamma, \Delta \uplus \{B_1\}, B_2 \rangle \in \|B\|$, where $\uplus$ denotes multiset union.

This is now used to define the backchaining rule:

$$\frac{\Gamma; \emptyset \vdash B_1 \ \ldots \ \Gamma; \emptyset \vdash B_n \quad \Gamma; \Delta_1 \vdash C_1 \ \ldots \ \Gamma; \Delta_m \vdash C_m}{\Gamma; \Delta_1, \ldots, \Delta_m, B \vdash A} \ BC$$

provided $n, m \geq 0$, A is atomic, and $\langle \{B_1, \ldots, B_n\}, \{C_1, \ldots, C_m\}, A \rangle \in \|B\|$

Hodas and Miller call this proof system $\mathcal{L}'$.

The proof system $\mathcal{L}''$

The connectives $\otimes$ and ! were omitted because of their bad behaviour when appearing on the left of the sequent. However, we have the possibility of allowing some connectives to appear in goals only. Hodas and Miller define $\mathcal{L}''$ using R-formulas (resource formulas), which can appear on either side of the sequent, and G-formulas (goal formulas), which can only appear only on the right of sequents.

$$R := \top \mid A \mid R_1 \& R_2 \mid G \multimap R \mid G \Rightarrow R \mid \forall x R$$

$$G := \top \mid A \mid G1 \& G2 \mid R \multimap G \mid R \Rightarrow G \mid \forall x G \mid G_1 \oplus G_2 \mid 1 \mid G1 \otimes G2 \mid !G \mid \exists x G$$

Now we can introduce rules for the missing forms $1, \oplus, \otimes, !$, and $\exists$.

$$\frac{}{\Gamma; \emptyset \vdash 1} \ r1$$

$$\frac{\Gamma; \emptyset \vdash B}{\Gamma; \emptyset \vdash !B} \ r!$$

$$\frac{\Gamma; \Delta \vdash B_i}{\Gamma; \Delta \vdash B_1 \oplus B_2} \ r \oplus (i = 1, 2)$$

$$\frac{\Gamma; \Delta \vdash B[x/t]}{\Gamma; \Delta \vdash \exists x.B} \ r\exists$$

$$\frac{\Gamma; \Delta_1 \vdash B_1 \quad \Gamma; \Delta_2 \vdash B_2}{\Gamma; \Delta_1, \Delta_2 \vdash B_1 \otimes B_2} \ r\otimes$$

Resource management

An extra problem that the proof system for linear logic faces is that some of the proof rules, e.g. for $\otimes$ require that the bounded context should be split in a non-deterministic way. The $r\otimes$ rule is one example.

$$\frac{\Gamma; \Delta_1 \vdash G_1 \quad \Gamma; \Delta_2 \vdash G_2}{\Gamma; \Delta \vdash G_1 \otimes G_2}\, r\otimes$$

When applying this rule in a bottom-up way, we need to decide how to partition Δ into Δ_1 and Δ_2. If Δ has cardinality n then there are 2^n possible partitions of Δ.

Fortunately, it is possible to perform this context splitting lazily. We first attempt a proof of $G1$ with all of the resources in Δ available. Then we can determine that Δ_1 comprises those resources actually used, and the remainder constitute Δ_2. Using this basic idea, Hodas and Miller formalise the notion by defining a 3-place relation of the form $I\{G\}O$, meaning that goal G can be proved from resources in I, with the resources O remaining. Using this representation it is possible to define the behaviour of the connectives using the following form:

$$\frac{I\{G_1\}M \quad M\{G_2\}O}{I\{G_1 \otimes G_2\}O}$$

Lolli for planning

Since the use of the $\otimes$ connective in resource formulas is forbidden in this language, we cannot model a planning action with multiple effects by a clause with a conjunction in its head. For this reason, planning problems cannot be directly presented to the Lolli interpreter. It is, however, possible to represent and solve planning problems by defining a very simple meta-interpreter, which uses forward reasoning instead of backward reasoning.

We give a minimal Lolli planner below, in which the clause `succeeds Plan Goal` is satisfied if `Plan` can be instantiated to a list of actions satisfying `Goal`. The operator `-o` represents linear implication, and attempting a goal of the form The goal `erase` stands for the constant $\top$, which is allowed to consume any resources left over after proving `Goal`.

```
succeeds nil Goal :-
   Goal,
   erase.

succeeds (Step :: Plan) Goal:-
   operator Step Pre Eff,
   Pre,
   Eff -o succeeds Plan Goal.
```

This program can be run to form a plan to get from `Initial` to `Goal` using the following query:

```
Initial -o succeeds Plan Goal.
```

With the in-built depth-first search strategy of Lolli, this program tends to loop, but by extending slightly to include a depth bound so we can do iterative-deepening search, we get a working but inefficient planner.

7.2.4 Summary

Jacopin's CSLL is a search algorithm for a very limited fragment of linear logic, sufficient to solve conjunctive planning problems using a forward-chaining strategy. The is no attempt to deal efficiently with non-determinism caused by context-splitting.

Connection-based methods are a family of methods in which redundancies in the proof search space (including context-splitting) are removed by considering direct relationships between facts and goals in the proof. So far, these methods have been extended to the fragment of linear logic including multiplicatives and exponentials , and to multiplicatives and additive disjunction.

linear logic programming in Lolli tackles proof search in a large fragment of linear logic, restricted in such a way that proofs are uniform. The fragment is made as large as possible by allowing some connectives to be used in goals only. A lazy context splitting mechanism deals efficiently with context splits.

7.3 Forward chaining versus backward chaining in ILL proof search

Before describing our own strategies for proof search in ILL, we consider the issue of forward versus backward chaining when constructing proofs to solve planning problems. Proofs may be constructed in the direction forwards from the initial state or backwards from the goal. We show that backwards search is problematic if $\top$ is used to avoid completely describing the goal state.

In this section, we use $Pre \vdash \mathit{Eff}$ representation for actions, which are applied by the cut rule.

7.3.1 Forward chaining without $\top$

For a proof in which actions are applied forwards, action preconditions are matched against current state. This is achieved by applying the cut rule so as to cut in action instances on the left. This form of cut application we call *fcut*. We then get proofs which have the following general shape:

$$\dfrac{\mathit{Action}_1 \qquad \dfrac{\mathit{Action}_2 \qquad \dfrac{\vdots}{\ldots \vdash G}}{\ldots \vdash G}\ \mathit{fcut} \qquad \dfrac{\vdots}{\ldots \vdash G}}{\ldots \vdash G}\ \mathit{fcut}$$

Note that the goal is preserved in righthand sequents.

7.3.2 Forward chaining with $\top$

If we wish to avoid fully specifying the goal, $\top$ may be used as a conjunct in the goal G. Application of the $r\top$ rule (Section 5.2.6) accounts for the disposal of any unused resources. The $r\top$ rule does not need be applied until after all the actions have been applied and goal conditions have been met.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\ldots \vdash G'} \quad \dfrac{}{\ldots \vdash \top}\, r\top}{\ldots \vdash G' \otimes \top}\, r\otimes \quad \dfrac{\vdots}{\ldots \vdash G' \otimes \top}}{\mathit{Action}_2 \quad \ldots \vdash G' \otimes \top}\, \mathit{fcut}}{\ldots \vdash G' \otimes \top} \quad \dfrac{\vdots}{\ldots \vdash G' \otimes \top}}{\mathit{Action}_1 \quad \ldots \vdash G' \otimes \top}\, \mathit{fcut}$$

7.3.3 Backward chaining without $\top$

Backward chaining involves checking the goal to see if it contains effects of a possible action. We can define a backward chaining (*bcut*) rule below, which is a specialisation of the standard cut rule. Here we assume matching on goal terms has built-in handling of associativity and commutativity of $\otimes$.

$$\dfrac{\Gamma \vdash \mathit{Pre} \otimes Z \quad \mathit{Pre} \vdash \mathit{Eff}}{\Gamma \vdash \mathit{Eff} \otimes Z}\, \mathit{bcut}$$

Note that we use a meta-variable Z to represent the part of the goal formula not achieved by the current action. The backward-chaining proof has the following shape:

$$\dfrac{\dfrac{\dfrac{\vdots}{I \vdash \ldots} \quad \mathit{Action}_1}{I \vdash \ldots}\, \mathit{bcut} \quad \dfrac{\vdots}{I \vdash \ldots}}{I \vdash \ldots \quad \mathit{Action}_2}\, \mathit{bcut}$$

Note that the initial state is preserved on each application of bcut. If $\top$ does not appear in the goal, the meta-variable in bcut is always immediately instantiated.

7.3.4 Backward chaining with $\top$

If we apply the $r\top$ rule to dispose of resources after the last action is applied, this corresponds to applying the $r\top$ near the root sequent of the proof. We must apply the $r\top$ rule before we know what resources it is to dispose of. In combination with

the *bcut* rule, this means we must allow an uninstantiated meta-variable into the goal. This is a problem, because the meta-variable allows the effects of any action to match the goal. Consider the following example:

Suppose we have a domain with resources a and b which are mutually exclusive. We encode this by including a single instance of resource a or b in the initial state, and making sure our action preserves the constraint that we have either one copy of a or one copy of b.

Suppose we have only one action

$$a \otimes c \multimap b \otimes c \tag{7.1}$$

Suppose we start from a specification

$$init \vdash b \otimes c \otimes \top \tag{7.2}$$

where *init* is a context representing the initial state.

Cutting in a meta-variable Z in place of $\top$, we get:

$$init \vdash b \otimes c \otimes Z \tag{7.3}$$

Applying backward-chaining *bcut* rule for the action, we get:

$$init \vdash a \otimes c \otimes Z \tag{7.4}$$

Applying *bcut* rule again, we get:

$$init \vdash a \otimes a \otimes c \otimes Z' \tag{7.5}$$

where $Z = b \otimes Z'$. Note that this instantiation means that the previous goal in 7.4 contains both a and b, which are intended to be mutually exclusive.

Applying *bcut* rule again, we get:

$$init \vdash a \otimes a \otimes a \otimes c \otimes Z'' \tag{7.6}$$

where $Z' = b \otimes Z''$.

Since linear logic allows us to have multiple copies of the same resource, and since we have a meta-variable in the goal, the goal may grow infinitely in the backward-chaining proof. This does not mean that an incorrect proof will ever be found, only that we expend a large amount of time searching a space that has no meaningful interpretation in the planning problem.

To get around this problem, we would need to impose some check, external to the logic, on whether the current goal is meaningful as a partial description of a state.

7.4 A complete proof search strategy for a fragment of ILL

In this section, we give a complete proof search strategy for a fragment of ILL using only $\otimes$, $\oplus$ and $\multimap$. This is sufficient for describing simple planning problems involving conjunction and disjunction.

We draw on two of the approaches to proof search in ILL mentioned earlier — Jacopin's CSLL algorithm [Jacopin 93] and linear logic programming systems, especially Lolli [Hodas & Miller 94].

We have forward chaining application of reusable transition axioms (in common with Jacopin's CSLL), handling for more of the connectives of ILL, and lazy context splitting in the style of [Harland & Pym 97].

7.4.1 Complete search

Since we forbid the use of exponentials, each of the allowed deduction rule, when applied bottom-up, results in smaller subgoals. We therefore have only a finite search space and the problem is decidable.

This search space exhibits some redundancy, as some orderings of rule applications are equivalent to others. We remove some of the redundancy be eagerly applying the 'safe' deduction rules.

We will then show how this generalises to the case where actions are reusable, by imposing a depth limit on the number of applications of the $l \multimap$ rule.

In Section 7.5.2 we describe the proof search strategy actually implemented. This further generalises the strategy described to use more connectives, i.e. quantifiers, constants $\top$, and $\mathbf{0}$, arbitrary applications of rewrite rules.

7.4.2 Invertible rules

Invertible rules are those for which the upper sequent is necessarily provable if the lower sequent is provable. Hence application of an invertible rule cannot be a mistake. One such rule is the $l\otimes$ rule.

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ l\otimes$$

We can show that this is invertible by showing that we can derive the upper sequent if we assume the lower sequent.

$$\frac{\dfrac{A \vdash A \quad B \vdash B}{A, B \vdash A \otimes B}\ r\otimes \qquad \Gamma, A \otimes B \vdash C}{\Gamma, A, B \vdash C}\ cut$$

See Appendix A for invertibility of other rules. Note about the use of cut: Since we used cut in these proofs, they are not necessarily valid in a proof search where we do not use cut. However, since we have a cut-elimination theorem for the logic, we know that an equivalent proof could be found (shown by transforming proofs). This argument does not remain valid for inductive proofs, as cut elimination no longer holds. Our search procedure does not encompass selection and application of induction rules.

By contrast, the $r\otimes$ rule is not invertible.

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

This rule relies on an appropriate splitting of the linear context into Γ and Γ'. In some cases an appropriate division is not possible, e.g.:

$$\frac{a \otimes b \vdash a \quad c \vdash b \otimes c}{a \otimes b, c \vdash a \otimes (b \otimes c)}\ r\otimes$$

In this case it is mistake to apply the $r\otimes$ rule without first applying $l\otimes$ to break the formula $a \otimes b$ on the left.

The invertible rules in the current fragment are: $l\otimes, l\oplus, r\multimap$:

Since application of these rules is always safe, it does not represent a real choice point in the search. It is sensible to apply these rules eagerly — i.e. always apply an invertible rule where possible. If more than one invertible rule is applicable, we can choose the order based on an arbitrary order. For our set of rules, we chose to apply $l\oplus$ with the lowest priority. This rule causes branching in the proof, so we prefer to apply it later.

7.4.3 Non-invertible rules

If there are no invertible rules to apply, we can attempt to apply the non-invertible rules $r\oplus, r\otimes, l\multimap$. If any of these are applicable, then we have a nondeterministic choice of which to apply and how to apply them. Choices are:

- Which of the possible rules to apply.
- How to split the context in the $r\multimap$ and $r\otimes$ rules.
- Which of the two possible $r\oplus$ rules to select — i.e. which goal to satisfy.

For $r\oplus$ and $r\otimes$, there can never be a choice over which to apply, since we have only a single formula on the right.

The application of the $l\multimap$ rule is more problematic, since this is where the choice of action occurs when reasoning about planning problems. There may be a choice of $\multimap$ formulae, corresponding to different available actions.

7.4.4 Strategy

The strategy is to apply eagerly the invertible rules $l\otimes, r\multimap, l\oplus$, and nondeterministically apply the other rules $r\otimes, l\multimap, r\oplus$ This is the general strategy used by e.g. [Galmiche & Boudinet 94].

7.4.5 Unbounded actions

In the discussion above, we considered only the use of bounded actions of the form $A\multimap B$. In the planning problems, we need to model actions that can be used any

number of times. Previously, we modelled this by the of the *cut* rule together with actions described by axioms. We can also describe unbounded actions by formulas of the form $!(A \multimap B)$. The exponential excuses the reuse, or non-use, of the formula in the proof.

With unbounded actions, we lose the termination property of our proof procedure. However, we can keep control by performing a search by iterative deepening. The problem of finding a proof with a specific number of action applications is equivalent to the bounded-actions case considered above, and is decidable. Here, we repeat the search, increasing the depth bound on each iteration.

7.4.6 Summary

We gave a complete strategy for applying sequent rules of ILL to solve planning problems. In Section 7.5, we present a search strategy for a wider fragment of ILL. That algorithm is not, however, complete.

7.5 Search strategy for a larger fragment

In this section we extend the search strategy described in Section 7.4. Here we will handle a larger fragment, at the expense of completeness. Previously, we only considered $\otimes, \oplus, \multimap$. The proof search procedure is defined here for a fragment using most of ILL, but we limit the use of cut, and omit explicit use of exponentials. However, the transition axioms representing planning operators are reusable, and so are a form of exponential formulas.

In the following, we consider actions to be represented by the $\multimap$ connective only. In the implementation we use an *lcut* rule to perform actions. The difference is that *lcut* allows actions to be treated as re-usable. Since our search algorithm contains the problem by bounding the number of times *lcut* can be applied, we can treat them equivalently here.

In Section 7.5.1 we analyse the invertibility or otherwise of the proof rules for this enlarged fragment of ILL. This analysis motivates the search strategy presented in

Section 7.5.2.

7.5.1 Categorisation of rules

Certain of the proof rules have the property of invertibility, meaning that the premises are derivable whenever the conclusion is derivable. It is possible to commit to the application of these rules without the danger of losing a proof. We now analyse the rules to determine a good strategy.

$l\otimes, r\multimap$ These (invertible) rules cause the context to increase. These rules should be applied with the highest priority, as they should be performed before any context splitting rules.

$l\exists, r\forall$ These rules are invertible. However, they introduce an eigenvariable and the side condition requiring that it does not appear elsewhere in the sequent.

$l\forall, r\exists$ These are not invertible, because they can interact with the side conditions of the $l\exists$ and $r\forall$ rules. For this reason, it is sensible to eagerly apply the invertible quantifier rules, and defer the application of $l\forall$ and $r\exists$.

$l\mathbf{0}$, $r\top$ If either of these rules can be applied, then the current branch of the proof is closed. There is an interaction with the context-splitting mechanism here, as we may have a delayed choice as to which resources in the context are actually present in this branch of the proof. It is never a mistake to apply these rules.

$l\oplus$ This rule is invertible, but has the unfortunate effect of causing the proof to break into two branches. Most of the effort after that is therefore going to be duplicated, so we are not keen to apply this early.

$r1\oplus, r2\oplus$ Here we have a nondeterministic choice point, since we need to apply one rule out of the pair. From an efficiency point of view, it would be preferable to defer these rules as long as possible.

$r\otimes$ splits the goal and the context. This can be a mistake, as some required splittings of the context may not be possible until context-increasing rules have been fully

applied. We therefore do not want to apply it until the context is fully decomposed. In fact, it is usually only useful when we are testing to see if we have achieved the goal.

$l \multimap$ Also causes a context split. In planning problems, its use corresponds to the application of an action which must be used exactly once — usually the induction hypothesis. Its use is potentially a mistake, if it splits the context too soon or if it is applied to the wrong resources.

We summarise this analysis in the following table.

Context increasing:	$l\otimes$	$r \multimap$	
Quantifiers:	$l\exists$	$r\forall$	Side conditions
	$l\forall$	$r\exists$	No side conditions
Decompose:	$l\mathbf{0}$	$r\top$	Close branch
	$l\oplus$		Causes branching
		$r1\oplus$ $r2\oplus$	Nondeterministic choice
Context splitting:		$r\otimes$	
	$l \multimap$		Actions
	Ax		Close branch

7.5.2 Search algorithm

The general strategy is to fully decompose formulas in the context before applying any context splitting rule. Note that this is unlike the strategy employed in e.g. Lolli (Section 7.2.3), where right rules are always applied ahead of left rules.

The only context splitting rules that we have are $l \multimap$, and $r\otimes$.

We define our strategy using three procedures:

1. Decompose

   Apply rules to remove connectives. This comprises application of the context-increasing, quantifier, and decompose rules, but not context-splitting rules from above table. Decompose contains no choice points — we commit to the selected rule.

2. TestGoal

   TestGoal is to check whether the goal has been reached already, without further actions. TestGoal is always applied after Decompose, and it is allowed to use all

the rules except $l \multimap$. Usually it will only need $r\otimes$ and Ax, which are applied with the lowest priority. TestGoal always terminates, with either success or failure.

3. Act

   Act attempts to select and apply an action using $l \multimap$, either failing or returning a new open sequent.

The strategy is to first apply Decompose, then apply TestGoal to each result. If TestGoal fails then apply Act and recurse on the result.

There are nondeterministic choices present which represent opportunities for making a mistake in the proof. These become choice points in a backtracking search. Notice that for the $r\oplus$ rules, the choice is only *which* of the two rules to apply. It is safe, however to commit to the order in which the rules are applied. Below, we give pseudo-code for our procedure, in which **choose** indicates a choice point to which execution may backtrack, and **select** indicates a choice to which we immediately commit. The **try ... else try ... end try** construct attempts each procedure in turn until one of them succeeds.

```
procedure LinoSolve( I ⊢ G ):
{
    try
        call Decompose( I ⊢ G)
    else try
        call TestGoal( I ⊢ G )
    else try
        call Act( I ⊢ G )
    end try
}

procedure Decompose( I ⊢ G ):
{
    select first applicable rule from the order:
        l⊗, r⊸, l0, l∃, r∀, r⊤, l⊕, l∀, r∃
    Apply rule to derive a set of unsatisfied sequents S
    for each sequent I' ⊢ G' in S do
        call LinoSolve( I' ⊢ G' )
}

procedure TestGoal( I ⊢ G ):
{
```

```
    try
    {
        select first applicable rule from:
            l⊗, r⊸, l0, l∃, r∀, r⊤, l⊕, l∀, r∃
        Apply rule to derive a set of unsatisfied sequents S
        for each sequent I' ⊢ G' in S do
            call TestGoal( I' ⊢ G' )
    }
    else try
        if dominant connective in G is ⊗
        then
        {
            Apply r⊗ to derive a set of unsatisfied sequents S
            for each sequent I' ⊢ G' in S do
                call TestGoal( I' ⊢ G' )
        }
        else fail
        endif
    else try
        if dominant connective in G is ⊕
        then
            choose either r1⊕ or r2⊕ to get I' ⊢ G'
            call TestGoal( I' ⊢ G') )
        else fail
        endif
    else try
        choose any possible application of ax and apply it
    end try
}

procedure Act( I ⊢ G ) :
{
    choose any formula in the context of the form a ⊸ b
    Apply l ⊸ to derive open sequents s1 and s2
    call TestGoal(s1) /* check action preconditions */
    call LinoSolve(s2) /* continue planning from modified state */
}
```

The use of lazy context-splitting means that there is no choice point for how to split a context at the point where a context-splitting rule is applied, e.g. by $r\otimes$ rule. However, there is a choice point wherever the commitment is made on how to split the context. This is done by ax and $l \multimap$ rules. Other left rules are forced to be applied before any context-splitting rules.

7.5.3 Reduced Fragment

Although we do not have complete proof search for the current fragment, we can identify some forms of failure of the proof search with some features of the sequent which we are attempting to prove. Here we define a restricted fragment of the logic for which these specific problems will not occur. This leaves us with a fragment which still allows all the connectives to be used in the form in which they most naturally appear in planning problems.

Some proofs will not be possible due to the restriction which we have placed on TestGoal — i.e. that we do not consider the application of actions during the TestGoal. For example, consider the following incomplete proof attempt for $a \vdash a \otimes (b \multimap c)$. When we apply TestGoal here, we get:

$$\dfrac{\dfrac{}{a \vdash a}\,Ax \quad \dfrac{b \vdash c}{\vdash b \multimap c}\,r\multimap}{a \vdash a \otimes (b \multimap c)}\,r\otimes$$

A proof of the open goal $b \vdash c$ may be possible by the application of transition axioms, but this specifically forbidden within TestGoal, so it will not be found.

We can prevent this situation from occurring by banning the use of $\multimap$ within a goal expression dominated by $\otimes$. The following grammar captures this idea by defining G_0 for goal expressions outside the scope of $\otimes$, and G_1 for goal expressions inside its scope. We will generally prove sequents restricted to be of the form $R \vdash G_0$

$$\begin{aligned} R &:= R \otimes R \mid G_0 \multimap R \mid R \oplus R \mid \forall x.R \mid \exists x.R \mid \top \mid \mathbf{0} \\ G_0 &:= G_1 \otimes G_1 \mid R \multimap G_0 \mid G_0 \oplus G_0 \mid \forall x.G_0 \mid \exists x.G_0 \mid \top \mid \mathbf{0} \\ G_1 &:= G_1 \otimes G_1 \mid G_1 \oplus G_1 \mid \forall x.G_1 \mid \exists x.G_1 \mid \top \mid \mathbf{0} \end{aligned}$$

None of our example planning problem specifications needed to be changed to conform to this reduced fragment.

7.5.4 Depth bound

An iterative-deepening strategy is used to control the proof search. The depth bound counts the number of applications of action axioms, (see Section 7.4.5).

7.6 Rewriting strategy

7.6.1 Introduction

We have previously described a proof strategy for a large fragment of intuitionistic linear logic. This fragment has been chosen to allow the handling of planning problems in an expressive language.

However, in order to deal with examples involving recursion, we make of use of auxiliary functions such as *reverse* in the problem specification. Such functions are specified recursively using equations.

This section considers the problems of controlling the application of these equations in testing equality between expressions.

7.6.2 Functions specified as rewrites

It would have been pleasing if the rippling strategy (Section 3.5.4) was successful in this case, but the interleaving with planning steps usually involves skeleton disruption, and cannot be handled by rippling. We have separately accounted for the application of actions during the proof search, and what we need is a strategy to use the rewrite rules in a cheap test of equality.

Since the equations represent function definitions, there is usually a natural orientation of the rule which corresponds to evaluation of the function. Such definitions usually yield rewrite rules which always terminate with a unique result. In such a case, to decide equality of ground terms, we need only apply the rules exhaustively.

In the presence of free variables, matters are unfortunately not so straightforward.

7.6.3 Summary

The inclusion of rewrite rules in planning domains introduces search problems for which there is no general, complete solution. However, a strategy of exhaustive rewriting is very often successful, so this approach is adopted.

7.7 Conclusion

We looked at three existing approaches to proof search in linear logic. For the requirements of the planning problems we wish to deal with, we introduced a complete algorithm, which combines forward search (as CSLL), with a large set of connectives extended to include $\multimap$, $l\oplus$ and lazy context splitting (as Lolli).

We then introduced an algorithm for a larger set of connectives, which is not shown to be complete, but for which we can define a restriction on logic which is relatively harmless in our applications, but which eliminates an important source of incompleteness.

We also described our approach to the problem of applying rewrite rules.

Chapter 8

The Lino Implementation

8.1 Introduction

In this chapter, we introduce the Lino system, and discuss its key features, particularly the lazy context splitting mechanism and proof search procedure.

8.2 Overview

In this section, we give an overview of the Lino system implementation. The Lino system comprises:

1. A proof checker, using standard proof rules of ILL, plus extended proof rules for recursion on lists, natural numbers and binary trees, and the application of term rewriting rules.

   Lazy context splitting is integrated and makes use of the CLP(FD) solver of SICStus Prolog [Carlsson *et al* 97]. Our implementation of lazy context-splitting is described in Section 8.3. Output of completed proofs is available formatted with LaTeX and as proof trees drawn using the *dot* package [Gansner & North 00].

2. An automated search procedure. The search procedure is implemented as a control layer on top of the prover. We discuss the procedure and its properties in Section 7.5.

3. Extraction of Abramsky (Linear Lambda Calculus) proof terms, as described in Chapter 5.

4. Partial evaluation of proof terms (Section 5.4).

5. Execution of proof terms (Section 5.3).

Notation

Some examples from the Lino system are given in this chapter. The following notation for linear logic symbols is used in Lino.

Girard	Lino
$\vdash$	`==>`
$\multimap$	`-<>`
$\otimes$	`*`
&	`&`
$\oplus$	`+`
$\top$	`top`
0	`0`
$\forall x{:}\tau.P$	`all(x,`P`)`
$\exists x{:}\tau.P$	`exists(x,`P`)`

The Lino versions of the quantifiers lack type restrictions. This is an omission in the current version of Lino, which potentially allows application of inappropriate induction rules. This flaw will be rectified in future versions.

8.3 Lazy context splitting

Any theorem prover for linear logic needs to deal with the choice arising from the need to split the context when applying multiplicative rules (Section 7.2.3).

The Lolli system uses an input-output model of resources. In splitting the context between two branches of the proof, first one branch is attempted, with all resources being made available to it, then the unused resources are passed to the second branch.

This is adequate some of the time, but extra complications may arise. Suppose the first branch attempted contains an occurrence of $\top$ in the goal position. The behaviour of the $r\top$ rule is such that it may consume as many or as few of the resources as we

wish. So when we attempt the right branch of the proof, we still have not fixed which resources are available.

A more general approach was given by [Harland & Pym 97]. A boolean variable is attached to each resource to indicate whether it is available in a given branch of the proof. Constraints are then established between these boolean variables.

By using the boolean constraints model, we do not make any assumptions about what order will be used for tackling branches, or what proof strategy is employed.

Below we give the $r\otimes$ rule, annotated for lazy context splitting. The variables in square brackets represent boolean values indicating whether the resource is available.

For example, the resource a must be present on the left side of exactly one of the upper sequents. Its occurrence in the first sequent is determined by a boolean value x_1, and its occurrence in the second sequent is indicated by the opposite value $\bar{x_1}$.

$$\frac{a[x_1], b[x_2] \vdash a \quad a[\bar{x_1}], b[\bar{x_2}] \vdash b}{a, b \vdash a \otimes b}\, r\otimes$$

In proving the left branch we need to set $x_1 = true$ and $x_2 = false$, and this automatically resolves the resources present in the right branch. This leads to the proof we want:

$$\frac{a \vdash a \quad b \vdash b}{a, b \vdash a \otimes b}\, r\otimes$$

When Lino is used interactively, a hypothesis under the control of the constraint solver is flagged with an @ symbol. The meaning is that the resource may optionally be used in the current branch of the proof.

In the following example, we illustrate the lazy context splitting by showing an interactive proof of $a, b \vdash a \otimes b$. Input from the user is in the form of rule names (the `r` input applies any possible right rule).

```
Welcome to Lino, version 1.4

  h1: a
  h2: b
```

```
  ==> a*b

|: r.
Applied r(*)

@ h1: a
@ h2: b

  ==> a

|: ax.
Applied ax

  h2: b

  ==> b

|: ax.
Applied ax

*** Completed subproof ***


*** QED ***
```

In the second example, we consider the behaviour of the $r\top$ rule. In this example, the use of the $r\top$ rule in one branch of the proof does not resolve the context split, and the hypotheses `h1` and `h2` remain under the control of the constraint solver until the second branch of the proof is completed.

```
Welcome to Lino, version 1.4

  h1: a
  h2: b

  ==> top*b

|: r.
Applied r(*)

@ h1: a
@ h2: b

  ==> top
```

```
|: r.
Applied r(top)


@ h1: a
@ h2: b

  ==> b

|: ax.
Applied ax

*** Completed subproof ***


*** QED ***
```

8.4 Problem specification

A problem specification consists of definitions of rewrite rules, axioms and goal theorem.

These are simply described using Prolog facts, e.g., the specification for the problem of reversing a tower of blocks described in Section 6.4.1.

```
probname(revblocks).

rr(rev1, [],    rev(empty) => empty ).
rr(rev2, [b,t], rev(b::t) => app(rev(t),b::empty) ).

rr(app1, [u],     app(empty,u) => u ).
rr(app2, [b,t,u], app(b::t,u) => b::app(t,u) ).

rr(appassoc, [a,b,c], app(app(a,b),c) => app(a,app(b,c))).

axiom( pick(b::t), [b,t], [twr(b::t),hn]   ==> twr(t)*hold(b) ).
axiom( put(b,t),   [b,t], [twr(t),hold(b)] ==> twr(b::t)*hn ).

goal(
[] ==>
   all(t,
     all(a,
       twr(t)*twr(a)*hn -<> twr(empty)*twr(app(rev(t),a))*hn ))
).
```

8.5 Proof scripts

A scripting facility is implemented. Scripts are stored in the form of Prolog lists specifying a sequence of rules to be used by Lino. An interactive Lino session generates a script by logging rule applications.

Generally, the steps that can be used in the script are invocations of primitive proof rules. However, the search algorithm of Section 7.5, which functions as a proof tactic, can be invoked in the same way at any point in a scripted or interactive proof.

8.6 Output from Lino

In Lino, two styles of output for proofs are supported:

- Proof trees in LaTeX, formatted using the package `proof.sty`. This is conventional presentation of sequent proofs, but is suitable only for small proofs.
- Proof trees presented graphically with cross-referencing to LaTeX-formatted sequents at individual proof steps. Graphical trees are generated using the *dot* package [Gansner & North 00].

Extracted plans in the form of Linear Lambda Calculus terms are output. These terms are also output to a file which be can be read directly into the partial evaluation and plan execution module.

8.7 Conformant plans

We considered conformant and contingent plans in Section 2.5. Actions with uncertain effects are modelled using a disjunction of possible effects. In our framework, we resolve this disjunction in the proof using the $l\oplus$ rule, branching the proof for each possible outcome. This results in a plan which conditionally branches on the disjunction. This is a contingent plan.

In the case of conformant planning, it is deemed to be impossible to perform a test a run-time to resolve the disjunction, and conditional branches should not appear.

We must resolve the disjunction in order to complete the proof. To ensure that this does not lead to a conditional branch in the resulting plan, we need to do one of two things:

1. Ensure that exactly the same plan is used in both branches, hence the case split exists in the proof but is redundant in the plan. This brings in issues about deciding equality of plan terms, which we would prefer to avoid. This is discussed further in Section 10.3.3.

2. Ensure that no actions at all are used in the branches for each disjunct. The conditional branches then only select between different ways of demonstrating that the goal has been reached, but do not effect actions to be performed by the executing agent.

In Lino, we take the second approach. We model the non-testable form of $\oplus$ using an alternative connective, $\oplus\!\!\oplus$. Thus we can solve problems such as the socks problem [Bibel 86].

The restriction for conformant plans is a restriction of the proof search only, not of the underlying logic. The proof rules for the new connective are the same as those for $\oplus$, though we modify the extract terms.

The proof search relegates the application of the $l\oplus\!\!\oplus$ rule into the TestGoal phase of the proof search. In this phase of the proof search, no applications of actions are allowed. Hence we enforce that both plans for both disjuncts can be formed with no further applications of actions.

This is a somewhat restricted form of conformant planning which means that the disjuncts must form the goal directly, and cannot be used to form preconditions to further actions.

When dealing with problems involving $\oplus\!\!\oplus$, we must be more careful about the form of the plan term constructed. The restriction on the proof guarantees that the same actions will work regardless of which disjunct holds, however the substitution operation used in building the plan terms allows the action invocation to be substituted inside the conditional branches. We can use the following alternative form, which fixes the

evaluation of the action outside the scope of the conditional.

$$\frac{\Gamma \vdash t : A \quad x : B, \Gamma' \vdash u : C}{\Gamma, \Gamma', f : A \multimap B \vdash \texttt{let } f \circ t \texttt{ be } x \texttt{ in } u : C} \; l\multimap$$

Since the conditional branches have no significance with regard to action execution, we do not wish them to appear in the plan at all.

The form of the $l\oplus$ rule is given by:

$$\frac{\Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma, z : A \oplus B \vdash \texttt{case } z \texttt{ of inl}(x) \texttt{ then } u, \texttt{inr}(y) \texttt{ then } v : C} \; l\oplus$$

For the $l\oplus\oplus$ rule, we use a form which simply says that an appropriate type will be returned:

$$\frac{\Gamma, x : A \vdash \_ : C \quad \Gamma, y : B \vdash \_ : C}{\Gamma, z : A \oplus\oplus B \vdash \texttt{yields}(C) : C} \; l\oplus\oplus$$

We need to extend our operational semantics with appropriate rules defining how these new forms are executed.

The full *socks* example appears in Section B.9.

8.8 Partial evaluation and plan execution

Partial evaluation is a direct implementation of the partial evaluation rewrite rules given in Section 5.4 and Section 6.3.5. The notion of substitution requires some care, as in the presence of λ-terms, we must avoid the capture of free variables.

The rules are applied exhaustively, starting with leftmost outermost applications.inwards.

Plan execution is a direct implementation of the operational semantics described in Sections 5.3 and 6.3.4.

8.9 Code Size

The table below gives a breakdown of the size of the code in the Lino system. The figures give the number of lines of Prolog code, excluding comments and blank lines.

We consider that this a relatively small program, given that it provides proof checking, a degree of automation, and the handling of plan terms.

Module	Lines
Theorem-proving shell and deduction rules	620
Plan extraction	234
Partial evaluation and execution	522
Search	81
Output formatting	437
Total	1894

8.10 Conclusions

In this chapter, we have described the Lino system, which is an implementation of the ideas introduced in Chapters 5, 6, and 7.

Lino is a proof checker for ILL, enhanced with induction rules and term rewriting rules. Lino handles interactive, scripted or automatic approaches for performing proofs. Unlike general-purpose proof checkers, Lino provides built-in handling for lazy splitting of linear contexts. It also provides extraction, partial evaluation and execution of plan terms. Automated proof search mixes application of left and right sequent rules to provide a forward-chaining search implemented by a strategy which eliminates much of the possible redundancy in handling application of proof rules. In summary, Lino is designed specifically to handle the various requirements of deductive planning in ILL.

Chapter 9

Evaluation

9.1 Introduction

In this chapter we consider the capabilities and performance of our planning technique in comparison to:

- Methods for planning or proof search in linear logic.
- Recursive planning methods introduced in Chapter 3, i.e. those of [Manna & Waldinger 87], [Ghassem-Sani 92], and [Stephan & Biundo 95]

9.2 Comparison with linear logic systems

The linear logic planning system and theorem provers vary in the logic they deploy, i.e. classical or intuitionistic and the fragment they handle, and the proof search technique they use.

9.2.1 Jacopin

Jacopin uses a very restricted fragment of intuitionistic linear logic. Our planning algorithm would behave like Jacopin's on any problems defined with that restricted fragment, except that our more careful application of the context-splitting rule $r\otimes$ would be expected to give us a performance advantage. One of Jacopin's key criticisms of the use of linear logic, i.e. the necessity of fully specifying the goal state, can be

overcome simply by the inclusion of $\top$ in the goal state. Jacopin prefers to use only the very limited fragment of the logic to avoid moving into a higher complexity class.

In fact, simply adding the $\top$ to Jacopin's minimal fragment makes almost no difference to the forward-chaining search procedure.

The example which Jacopin gives to back up this argument is that of swapping two registers, given an action for assignment. Here, the resource $cont(x, a)$ is used to represent that register x contains value a, and the problem is to exchange the values of registers x and y, with an extra register z available. In Jacopin's restricted fragment, the problem is represented as follows:

$$cont(x, a), cont(y, b), cont(z, zero) \vdash cont(x, b) \otimes cont(y, a) \otimes cont(z, b)$$

The problem that Jacopin points out is that the goal state must include a specification of the final state of the spare register, z, which is not so much part of the goal as a consequence of the solution.

In our framework, we would avoid this problem by using $\top$ as mentioned.

$$cont(x, a), cont(y, b), cont(z, zero) \vdash cont(x, b) \otimes cont(y, a) \otimes \top$$

An alternative is to use an existential quantifier to describe the value of the intermediate register.

$$cont(x, a), cont(y, b), cont(z, zero) \vdash cont(x, b) \otimes cont(y, a) \otimes \exists val.cont(z, val)$$

Jacopin's other criticism is the inability of linear logic to represent a situation where a single effect satisfies a precondition of two different actions, with both actions preserving the condition. In linear logic, any such an action must consume then replace the condition, so it would not be possible for them to exist in parallel plans.

9.2.2 Lolli

We have covered Lolli in some detail in Chapter 7.

Our proof search algorithm has been inspired by Lolli, but adopts a different set of restrictions placed on the fragment and a different strategy. The choice of strategy made in Lolli make it unsuitable for direct use as a planner.

In particular, our strategy is not goal driven, and allows the interleaving of the application of left and right rules. This has allowed the adoption of a set of connectives which is more appropriate for planning problems.

9.2.3 Lygon

Lygon documentation [Harland *et al* 96] devotes a little attention to its application to planning problems. Lygon is based on classical linear logic, and it is an interesting question whether this is suitable as a logic for plan formation. In fact, the problem which they give as an example involves deriving the final state resulting from executing a set of actions (without a specified order).

9.2.4 Hölldobler et al.

The work of Hölldobler's group has the following aspects: The basic approach has been to describe a system based on equational reasoning.

The correspondence was established between their approach and those of Bibel and Masseron. In a later paper by Brüning and others [Brüning *et al* 93], they have extended the method to deal with disjunction. This informs an adaptation of Bibel's linear connection method to solve disjunctive problems. They identify the $l\oplus$ rule with formation of conditional branches in the plan, and this is equivalent to the formulation that we have used. However, their notion of plan extraction does not allow for the formation of conformant plans in which uncertainty is handled without the use of conditional branches.

Some consideration has also been given to extending the method to recursive plans [Hölldobler & Störr 98]. This paper considers only the correctness of given recursive plans with respect to given initial states. Nevertheless, it is interesting because it defines a syntax and semantics for a recursive plan language.

9.3 Comparison with other recursive planners

The approaches of [Manna & Waldinger 87], [Ghassem-Sani 92] and [Stephan & Biundo 95] were described in Chapter 3.

9.3.1 Basis for comparison

There is some difficulty making a useful comparison between the behaviours of the various systems. We must rely only on published accounts of the systems, which only give us an overview, with limited examples. Each system uses a different representation for problems, with brings its own restrictions on what problems can be expressed.

We consider the following aspects of representation style to be important:

- The logic on which the system is based, e.g. situation calculus, modal logic, linear logic.
- The approach adopted to handling the frame problem.
- The use of auxiliary functions and/or predicates in defining problems, such as our use of a *reverse* function defined using rewrite rules.
- The use of inductively defined datatypes versus the use of induction over well-founded relations.
- The use of a fixed set of induction rules versus the dynamic creation of induction rules on demand.

Strategies and Implementations:

- If the systems were implemented at all, how much of the process was automatic?
- The problem solving strategy adopted by the planner — for example searching forwards from the initial state or backwards from the goal state.
- Automatic generalisation.

9.3.2 Manna and Waldinger

The approach of Manna and Waldinger is based on first order situation calculus. This is very expressive for representing problems, but correspondingly difficult to reason with.

Their logic is not constructive, so could be used to produce plans which are not executable. Manna and Waldinger guard against this by separately filtering out non-executable plans during plan construction in an ad hoc way.

No solution to the frame problem is given. Explicit frame axioms are required, and explicit inference steps are required to handle them. The problem is discussed in [Manna & Waldinger 87], pages 364–365.

Their plan theory, unlike our formalism, allows functions and identifiers to have different interpretations between states. This introduces extra complications in reasoning.

In their "how to clear a block" example, the only example they present, they do not make use of auxiliary functions to define the relationships between states.

Because of redundancy in the representation, which refers to entities with or without a state argument, the approach requires a form equational unification. This equational unification is troublesome, because it may yield an infinite number of unifiers. Although the example given in [Manna & Waldinger 87] does not make use of any auxiliary functions, in rewrite rules defining these functions could be built into the equational unifier.

The proof method is based on a deductive tableau system. There is no claim that the proof process can be automated, and there is no report of an implemented system.

Their example demonstrates the plan being constructed backwards from the goal, but the representation is capable is of building the plans in either direction. Plans themselves are represented as terms constructed during proof.

Induction is performed on the basis of well-founded relations. The selection and generalisation of induction rules are identified as difficult problems which are not solved automatically.

They use a general form of induction over relations. This must be customised for particular theories by hand. It part of the process of theory development to show that relations are well-founded. This results in a fixed set of well-founded relations that can be used in a generic induction rule.

In general, we believe that Manna and Waldinger's representation is more flexible than ours, but much more complex and not so amenable to automation.

9.3.3 Ghassem-Sani and Steel

Ghassem-Sani and Steel's RNP planner is based on a restricted form of Manna and Waldinger's plan theory. Explicit reference to state is avoided completely by the use of STRIPS operators. The STRIPS assumption is used to avoid the frame problem.

Plans are represented as partially-ordered networks of plan steps, in the style of planners such as Nonlin. This representation is augmented with special types of nodes to allow the representation of conditional and recursive plans.

The proof process is fully automatic, and is goal driven in the manner of conventional partial-order planners. The occurrence of a subgoal related to the final goal by a well-founded relation motivates a case analysis for the introduction of recursive constructs. Typically, the guard conditions of partially-defined destructor functions form the basis of the case analysis. Thus RNP has a principled approach to introducing induction into the proof and to forming an appropriate induction rule. A limited form of automatic generalisation is also performed by the planner.

In comparison to our system, we find that RNP has a more compelling solution to the problems of search control. However, this is achieved at a cost of restricting the language to the point where some of the more interesting problems cannot be expressed.

There is no equivalent in RNP of using auxiliary functions to express the relationship between initial and goal states using rewrite rules. In their example of reversing a list, they use plan operators to fulfill the role that is taken by rewrite rules in our system — i.e. defining equivalences instead of state transformations. Since they have no other way to define the relationship between initial and goal state, it seems impossible to specify a problem like our reversing a list example, in which the relationship between

initial and final list is defined using rewrite rules.

9.3.4 Stephan and Biundo

In the approach of Stephan and Biundo, a dynamic temporal logic is used. This is an expressive framework, but inference in this framework is a complicated process, with multiple proof steps required to handle frame conditions.

A tactical theorem prover is used to control reasoning. Plans are represented as terms which exist in the logic at the same level as terms describing world state.

Induction can be performed on the basis of inductively-defined datatypes. This is an interactive process in which a user must make key decisions. Having done this, the recursively defined procedures may be used in an automatic system.

In comparison with our system, the emphasis is somewhat differently placed. They provide a sophisticated environment for the modelling of planning domains, where work is done up front by a domain modeller. The end result is a system which can work automatically in that specific domain.

The process of synthesising the recursive plans themselves is not automatic, nor is generation of generalisations in induction.

9.3.5 Lino

Our approach makes use of a fragment of intuitionistic linear logic, which is carefully chosen to be expressive enough to handle interesting planning problems, but restrictive enough to enable a useful search procedure to be defined.

For expressing recursive planning problems, we allow the use of inductively-defined datatypes. We use predefined induction rules, making use of constructors only. Selection of the induction rule must be done manually, as must rule generalisation.

We do not allow functions and symbols to change their interpretations between states, and this makes it simple to define a relationship between initial and goal states using a recursively defined function.

The linear logic approach has a very tidy handling of the frame problem, and a very clear and direct relationship between proof steps and constructs in the plan language. We can represent partially-ordered, conditional plans and recursive plans. The logical system is considerably simpler than the other deductive approaches of [Manna & Waldinger 87] and [Stephan & Biundo 95].

9.4 Example problems

In this section we consider the planner with respect to various problems existing in the literature. Due to the small number of publications on recursive planners, there is no established corpus of problems. The greatest number of examples comes from Ghassem-Sani's thesis [Ghassem-Sani 92]. Below we consider the suitability of our planner on these problems. We also give a number of further problems devised as test cases for our planner.

9.4.1 Examples from Ghassem-Sani

The examples from Ghassem-Sani are as follows:

1. Factorial function
2. Division function
3. Fibonacci function
4. Ackermann function
5. Reversing a list
6. Clearing the base of a tower
7. Building a tower
8. Hammering a nail into a plank

Examples 1-5 need not be modelled as planning problems at all. They are proofs of mathematical theorems for which no notion of state change is necessary. In Ghassem-Sani's representation, there is no separate notion of a rewrite rule. The RNP plans in

these cases are effectively proofs that exhaustive application of these rules would lead to a solution of the problem.

The Ghassem-Sani representations tend to make use of destructor functions. This is the most natural representation to use in conjunction with goal-driven plan search process.

Our tower reverse is similar to Ghassem-Sani's list reverse, but our formulation does have a state-changing aspect.

We cannot model the example of hammering a nail into a plank. In Ghassem-Sani's modelling of the problem, the outcome of each hammer action is that the nail is either flush or not flush.

We cannot use the modelling in our framework, because there is no corresponding inductively defined datastructure, which we require to model recursion in our framework. Furthermore, we should not be able to synthesise this plan in our system as its termination cannot be proved.

9.4.2 Example problems for Lino

These problems have all been solved interactively, and most of them are solved successfully solved by automated search. However, in all cases the selection of induction rules and the generalisation of the theorem was done by hand.

Problem	Requires Generalisation	Solution found by automated search	Time/ms	Comments
revblocks	Yes	Yes	70	
flatten	No	Yes	50	Uses list-of-lists formulation.
division	No	No	-	Formulation uses exponentials.
factorial	No	Yes	67	
fibonacci	No	Yes	27	
gripper	Yes	Yes	1261	Finds inefficient plan.
gripper2	Yes	Yes	22702	Finds efficient plan for same problem, by the use of a customised induction rule.
ants	Yes	No	-	Fails due to rewrite strategy preconditions.
boat	Yes	No	-	Search space too large.
nim9	No	Yes	154	9 counter problem - no induction.
nim	No	Yes	105	Problem specified for all winnable games
tree	No	No	-	Search requires equality substitution not handled by search procedure.
socks	No	Yes	126	Contingent version.
conformant socks	No	Yes	626	Conformant version.
omelette_lemma	Yes	Yes	138	
omelette3	Yes	Yes	116	
omelette	Yes	No	-	

Generally, the modelling of the problem in a suitable form in linear logic was found to be a difficult process. Although this is true to some extent of all planning problems, there are many established test domains for STRIPS and ADL planning. The difficulty involved in constructing reasonable formulations for the domains cannot easily be separated from the difficulty of solving them.

Timings were obtained on a PC with a 550MHz Pentium III processor, running Linux. The code bytcode compiled using SICStus Prolog 3.7.1.

The times are generally low, but the problems are small in size. The poor time for the gripper2 problem shows that the times scale badly as the length of the required step case plan increases. It is likely that this could be improved with simple modificatons

to the search code, as little attention has been given to optimising it for speed.

9.5 Discussion

Since there is no established corpus of linear logic planning problems, it is not possible to make a clear comparison of our system with others.

However, we have shown that our representation is expressive enough to give a clean formalisation of some existing problems, and many new ones. Automated proof search is often successful, and where it is not, this can often be related to specific features of the problem formulation.

We can tackle some problems which are adapted from standard STRIPS and ADL planning problems. This illustrates the notion that by making use of the inherent recursive structure in the problem, a solution can be found relatively easily, which will deal with problems of any size.

Chapter 10

Conclusions and Further Work

10.1 Introduction

In this chapter we report the main conclusions of the work and consider directions for further work.

10.2 Conclusions

10.2.1 Contributions

Recursive plans in linear logic

We have extended the use of linear logic in planning to deal with problems involving recursion.

We have demonstrated that intuitionistic linear logic, with an appropriate induction principle, can be used to represent and solve a range of recursive planning problems.

The linear logic formalism avoids the need for frame axioms and gives a clear relationship between proofs and plans. Plans are represented as terms of Linear Lambda Calculus, which are extracted directly from the proof of a plan specification. In this formalism, it is possible to synthesise general recursive plans which can handle a family of instances of planning problems.

We can also describe problems involving action steps with uncertain outcomes which may or may not be observable. Our plan language can express:

- parallel branches,
- conditional branches,
- recursion.

Schematic plans can be specialised with respect to information about a problem instance by means of partial evaluation rules. We also describe how to execute the plan directly on a problem instance, according to the operational semantics for the language.

The correctness of the plans rests on the correctness of the underlying proof rules with respect to the extraction and evaluation mechanism. This differs from the typical situation with conventional planners, for which it is necessary to prove the correctness of the planning search algorithm. In our approach, planning is done by theorem proving in our chosen logic. The correctness of plans is independent of the search mechanism used to prove the planning goal.

For RNP [Ghassem-Sani & Steel 91], correctness of plans depends in part on correctness of conflict detection, which in some cases involves testing conditions inside a recursive plan with parallel steps outside the recursive plan. This is a non-trivial problem.

Search procedure

We have given a search strategy for a defined fragment of the logic, and shown that it is complete. This fragment has been chosen carefully to enable interesting planning problems to be expressed. The fragment comprises only the following: $\otimes, \multimap, \oplus$. In the implementation the algorithm is extended to deal with the use of $\forall, \exists, \top, \mathbf{0}$, with some restrictions.

For problems involving only conjunction, this corresponds to forward-chaining in the style of [Jacopin 93]. However, we are are able to use a much larger fragment of the logic, and adopt efficient solutions to the problem of splitting the linear context.

Implementation

The above ideas have been implemented in Prolog as a system, Lino, which demonstrates the feasibility of the approach. It is a semi-automatic linear logic proof checker,

which uses a set of rules including induction rules and rewrite rules. Lazy context splitting is implemented using constraints in a CLP(FD) solver. Lino may be used for:

- interactive proof checking,
- automated proof search, which is complete under restrictions defined in Section 7.4.1. In the implementation, the proof search strategy is extended to deal with a larger fragment of the logic, for which completeness of the strategy has not been shown. A simple strategy for application of rewrite rules is also employed.

Lino also encompasses the following:

- automatic extraction of plans from completed proofs,
- partial evaluation of plans,
- execution of plans.

Results

Our system allows a range of problems to represented. The degree to which automatic solution is possible varies. See Appendix B for a summary of problem examples. We believe that it has a wider coverage of problems than the RNP planner of [Ghassem-Sani & Steel 91], whilst being more amenable to automation than the approaches of [Manna & Waldinger 87] and [Stephan & Biundo 95].

10.3 Further Work

The area of automatic recursive plan formation is a challenging one. In this section we consider the opportunities for extending the current work.

10.3.1 Proofs of correctness

In Chapters 5 and 6 we made the following claims in relation to partial evaluation and execution of plans.

Claim 1 Correctness of partial evaluation w.r.t. ILL-P (Section 5.4.1).

Claim 2 The relationship between $\longrightarrow$ and $\Downarrow$ (Section 5.4.2).

Claim 3 Correctness of partial evaluation w.r.t. ILL-PR (Section).

Claim 4 The relationship between $\Longrightarrow$ and $\Downarrow$ (Section 6.3.6).

We have given no proofs of these claims, so it is important further work to produce such proofs.

10.3.2 Equality of values

We have made use of limited reasoning about equality in our system, without fully developing a theory of equality (Section 5.2.8). Important further work would be to develop this theory and demonstrate correct deduction rules.

Note that we distinguish in our system between equality of values (e.g. blocks, towers) which do not have linear types and the treatment of plan terms, which do.

10.3.3 Equality of plan terms and conformant planning

Our treatment of conformant planning (Section 8.7) is compromised because we are limited to plans in which the outcomes of plan steps with uncertain effects are never used as preconditions to further plan steps. This means that many problems in conformant planning cannot be solved by our approach.

In order to solve the more general problem, we must allow the proof to branch on the disjunction, and allow each branch to contain further actions. However, since we cannot form a conditional test in the plan, we must enforce that the two branches contain equivalent plans. We need to define what we mean by the notion of equality of plans in this context to study how we can enforce this equality. The work of Barber [Barber 97] may be of some relevance here.

The conformant plans that we can synthesise require altered forms of plan terms. An extension of the operational semantics was not given for these new forms, and is required as further work.

10.3.4 Formulation of problems in linear logic

Since there is no existing corpus of recursive planning problems, our example problems have been hand-coded into linear logic specifications, such that inductively defined types were present in the specification given to the theorem prover. An important question is whether it would be possible to work from conventional plan specifications, such as the corpus of planning problems available in the PDDL language [McDermott *et al* 98].

This requires the generation of recursive plans from an input in which there is no explicit use of inductive datatypes. In Fig. 1.1, this corresponds to performing the steps labelled *problem translation* and *problem generalisation.* This route could be beneficial in cases where we are presented with a large problem instance that has a recursive structure. We would therefore expect solution of the abstract plan to be cheap compared to the large problem instance.

Once the abstract problem is solved, we can cheaply generate a specific plan by taking the *evaluation* and *plan translation* route. We will discuss these how the processes of *problem translation* and *problem generalisation* may be realised.

Problem translation

We would need a system capable of recognising the presence of a an inductively defined type in a planning domain. The domain description would be supplied in e.g. STRIPS or PDDL. Translation would be done relative to a library of standard inductively defined datatypes. Domain analysis, such as that used in the TIM system [Fox & Long 98], could identify the signature of encodings of inductive types. An example is the *on* relation in blocks world problems, which effectively behaves as a list constructor.

It therefore seems plausible that the concept of a tower could be automatically derived from a planning domain description. If a system is able to automatically bring into play the notion of lists in a domain in which they are only implicit, it can also recognise that a library of list-related concepts may be useful — e.g. *member*, *append.*

Generalisation of ground problems

In our recursive planning examples, we have made use of recursively defined functions to specify the required relationship between structures appearing in initial and goal states. To automatically derive such a problem specification, it would be necessary to speculate the general relationship between initial and final state. This would need to be done on the basis of a library of predefined relations (e.g. *flatten*, *reverse*).

This approach is potentially useful in solving problems which are very large, but with uniformly structured problem instances.

This is similar to problems that have been considered in the machine learning literature, particularly in the area of Inductive Logic Programming (ILP), e.g. [Muggleton 91].

10.3.5 Search control in Lino

Although the step cases of recursive plans can often be very short, the undirected forward-chaining search strategy of Lino is not efficient compared to modern STRIPS planners. Here we suggest some improvements.

Generalisation of ground plans

Small, ground problem instances can be generated from the domain definition. A fast STRIPS planner could then be used to generate possible solutions to the ground problem. These solutions could then be analysed w.r.t. recursive structure and used to guide the choice of actions during recursive plan generation. A similar form of generalisation is considered in [Baker 94]. In Fig. 1.1, this corresponds to a route from *abstract problem* to *abstract plan* via *ground problem* and *ground plan.*

This may also be used to inform choice of induction rule and generalised theorem.

We could envisage an architecture which solves a large problem by extracting a general recursive structure in the problem, then generates and solves one or more ground problem instances. Such solutions may then be generalised to lead to a provably correct general solution. This general solution can then be evaluated to solve a given large problem, or executed directly.

10.3.6 Induction rule choice

The work described in this thesis has not addressed the important problems of selection of induction rules, but instead has relied on pre-selection of induction rules from a limited set. The literature on inductive theorem-proving contains a number of more sophisticated approaches which allow an induction rule to be synthesised or selected on the basis of a partially complete step case proof. Important recent approaches to this problem are:

- In [Kraan 94, Kraan *et al* 96] induction rules are chosen from a pre-stored set, but the choice is deferred until a step case proof has been attempted. The step case proof is performed with meta-variables in place of induction terms. Rippling is used to control the instantiation of the meta-variables, and the resulting step case is used to select a matching induction rule.
- In [Protzen 95] induction rules are not stored, but synthesised from scratch. The method can only deal with destructor-style induction, and generates appropriate restrictions on hypotheses such that the induction rule is sound.
- Gow [Gow 00] first finds a step case proof using a representation with meta-variables (as Kraan did). The other cases of the induction rule are then generated and proved automatically. Gow's approach allows rules to be synthesised from scratch without Protzen's restriction to destructor-style inductions.

10.3.7 Generalisation of specification

The use of the induction hypothesis often fails due to the absence of a matching resource. For example, in the cases where we consider towers of blocks, we usually require to move the blocks to a tower which may not be mentioned in the specification.

In this case, the solution is to build the proof for a strengthened specification with extra resources. This is the same class of problem which has been tackled successfully in [Hesketh *et al* 92, Ireland & Bundy 96] (see Section 3.5.6). In that work, the problem is solved by adding accumulator arguments to the specifications. In the planning case, failure to prove should suggest missing resource and generalise the initial final states

of the planning problem specification.

10.3.8 Complete search

In Section 7.4 we considered a complete search strategy for a fragment of ILL including only the connectives $\otimes, \oplus, \multimap$.

The fragment of the logic which we use in representing our planning problems is often larger than the fragment for which proof search is shown to be complete.

Further work is required to define an extended fragment for which proof search can be proved to be complete.

10.4 Summary

We have demonstrated that linear logic provides a suitable formalism for the expression of planning problems involving inductive types, which has enabled us to realise our aims of making a planner that can form plans involving conditional constructs and recursion.

Whilst our work has some limitations, we believe that the approach is very promising, and there are interesting opportunities to overcome these limitations.

We believe that this is a promising approach which opens several interesting possible areas for future research.

Appendix A

Invertible Rules in Intuitionistic Linear Logic

Original rule	Proof of inversion
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\, r\multimap$	$\dfrac{\Gamma \vdash A \multimap B \quad \dfrac{A \vdash A \quad B \vdash B}{A, A \multimap B \vdash B}\, l\multimap}{\Gamma, A \vdash B}\, cut$
$\dfrac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\, l\otimes$	$\dfrac{\dfrac{A \vdash A \quad B \vdash B}{A, B \vdash A \otimes B}\, r\otimes \quad \Gamma, A \otimes B \vdash C}{\Gamma, A, B \vdash C}\, cut$
$\dfrac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}\, l\oplus$	$\dfrac{\dfrac{A \vdash A}{A \vdash A \oplus B}\, r1\oplus \quad \Gamma, A \oplus B \vdash C}{\Gamma, A \vdash C}\, cut$
	$\dfrac{\dfrac{B \vdash B}{B \vdash A \oplus B}\, r2\oplus \quad \Gamma, A \oplus B \vdash C}{\Gamma, B \vdash C}\, cut$
$\dfrac{\Gamma \vdash A[a/x]}{\Gamma \vdash \forall x A}\, r\forall$	$\dfrac{\Gamma \vdash \forall x.A \quad \dfrac{A[a/x] \vdash A[a/x]}{\forall x.A \vdash A[a/x]}\, l\forall}{\Gamma \vdash A[a/x]}\, cut$
$\dfrac{\Gamma, A[a/x] \vdash C}{\Gamma, \exists x A \vdash C}\, l\exists$	$\dfrac{\dfrac{A[a/x] \vdash A[a/x]}{A[a/x] \vdash \exists x A}\, r\exists \quad \Gamma, \exists x A \vdash C}{\Gamma, A[a/x] \vdash C}\, cut$

Appendix B

Examples

B.1 Reverse blocks

This is the example of reversing a tower of blocks. It is solved automatically by the prover.

B.1.1 Problem specification

$$\vdash \forall t.\forall a.\left(\left[\begin{array}{c} twr(t) \\ \otimes \\ twr(a) \\ \otimes \\ hn \end{array}\right] \multimap \left[\begin{array}{c} twr(empty) \\ \otimes \\ twr(app(rev(t),a)) \\ \otimes \\ hn \end{array}\right]\right)$$

B.1.2 Axioms

$$\vdash \texttt{pick(b :: t)} : \quad twr(b :: t) \otimes hn \multimap twr(t) \otimes hold(b)$$

$$\vdash \texttt{put(b, t)} : \quad twr(t) \otimes hold(b) \multimap twr(b :: t) \otimes hn$$

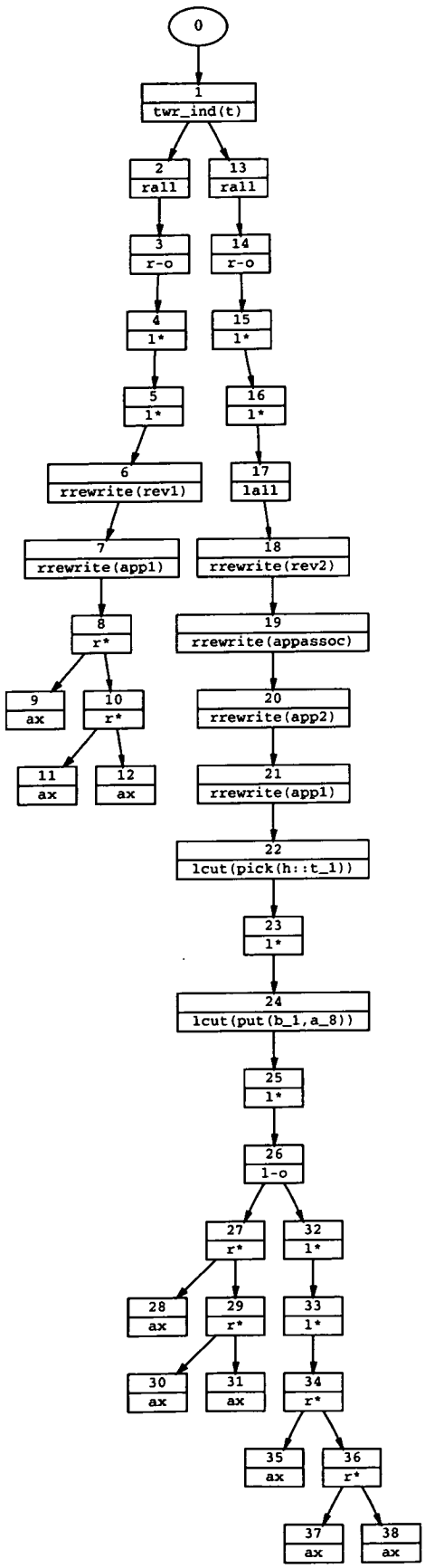
B.1.3 Rewrite rules

$$\begin{array}{rcll} rev(empty) & \rightarrow & empty & (rev1) \\ rev(b :: t) & \rightarrow & app(rev(t), b :: empty) & (rev2) \\ app(empty, u) & \rightarrow & u & (app1) \\ app(b :: t, u) & \rightarrow & b :: app(t, u) & (app2) \\ app(app(a, b), c) & \rightarrow & app(a, app(b, c)) & (appassoc) \end{array}$$

B.1.4 Plan

```
λt_2.
  twr_rec(t_2,
    λa_2.λh3.
        let h3 be h6*h7 in
          let h7 be h10*h11 in h6*h10*h11,
    λh1.λb_1.λt_1.λa_8.λh60.
              let h60 be h63*h64 in
                let h64 be h67*h68 in
                  let pick∘h63*h68 be h75*h76 in
                    let put∘h67*h76 be h92*h93 in
                      let h1∘b_1::a_8∘h75*h92*h93 be h97*h98 in
                        let h98 be h101*h102 in h97*h101*h102)
```

B.1.5 Proof tree



B.2 Flatten

This problem is to flatten a tower of blocks. The flattened tower is represented by a list of single-block towers.

B.2.1 Problem specification

$$\vdash \forall t.twr(t) \otimes lst(nil) \multimap twr(empty) \otimes lst(flattened(t))$$

B.2.2 Axioms

$$\vdash \texttt{lop(h)}: \quad twr(h :: t) \multimap twr(t) \otimes twr(h :: empty)$$

$$\vdash \texttt{gather(t,l)}: \quad twr(t) \otimes lst(l) \multimap lst(t :: l)$$

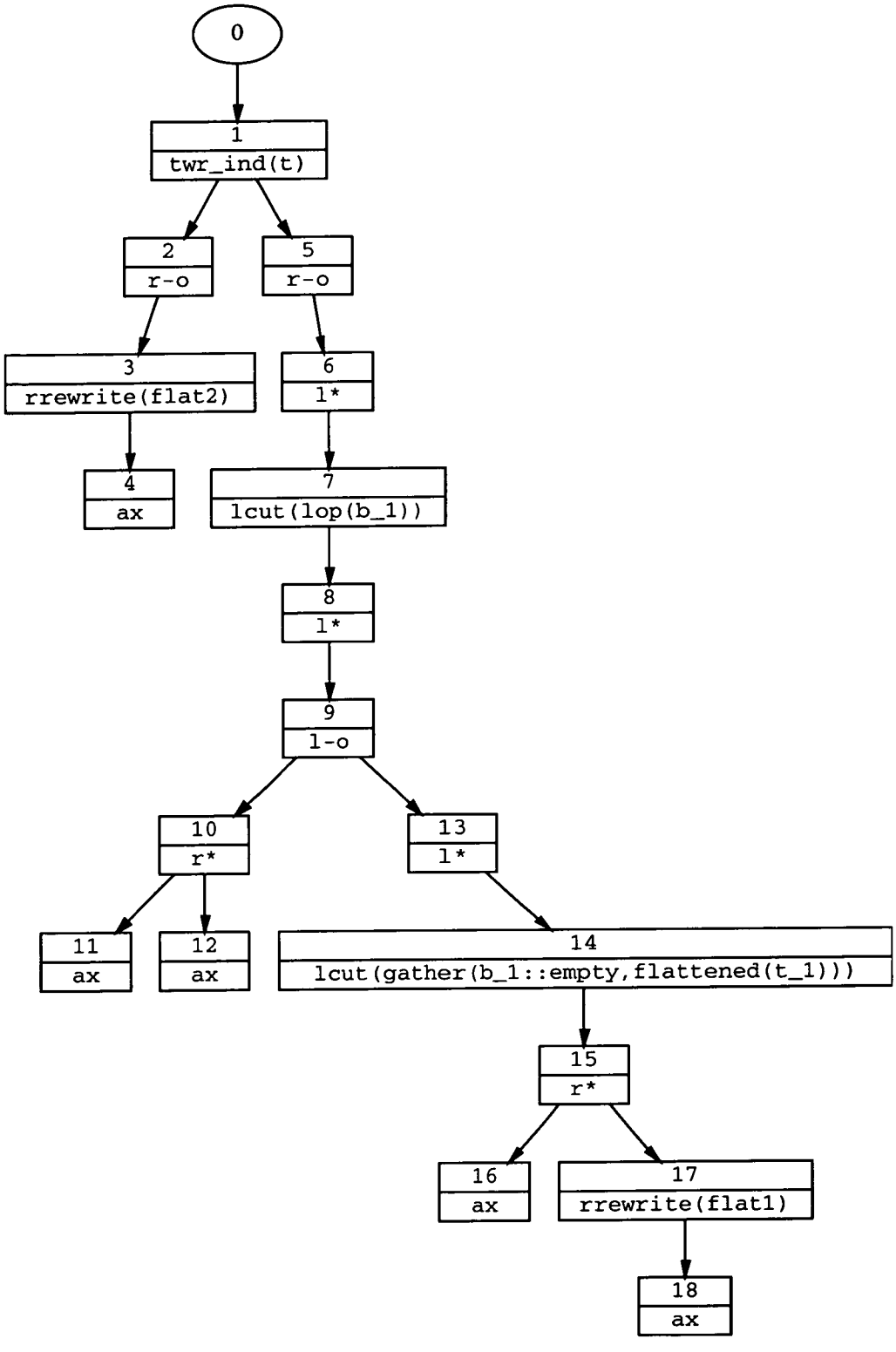
B.2.3 Rewrite rules

$$\begin{array}{rcll} flattened(h :: t) & \rightarrow & (h :: empty) :: flattened(t) & (flat1) \\ flattened(empty) & \rightarrow & nil & (flat2) \end{array}$$

B.2.4 Plan

```
λt_2.
  twr_rec(t_2,
    λh2.h2,
    λh1.λb_1.λt_1.λh3.
            let h3 be h4*h5 in
              let lop∘h4 be h7*h8 in
                let h1∘h7*h5 be h10*h11 in h10*gather∘h8*h11
```

B.2.5 Proof tree



B.3 Factorial

B.3.1 Problem specification

$$\vdash \forall x.nat(x) \multimap nat(fac(x)) \otimes \top$$

B.3.2 Axioms

$$\vdash \texttt{mult(x,y)}: \quad nat(x) \otimes nat(y) \quad \multimap \quad \left[\begin{array}{c} nat(x) \\ \otimes \\ nat(y) \\ \otimes \\ nat(mu(x,y)) \end{array}\right]$$

$$\vdash \texttt{sub1(x)}: \quad nat(s(x)) \quad \multimap \quad nat(x) \otimes nat(s(x))$$

$$\vdash \texttt{add1(x)}: \quad nat(x) \quad \multimap \quad nat(x) \otimes nat(s(x))$$

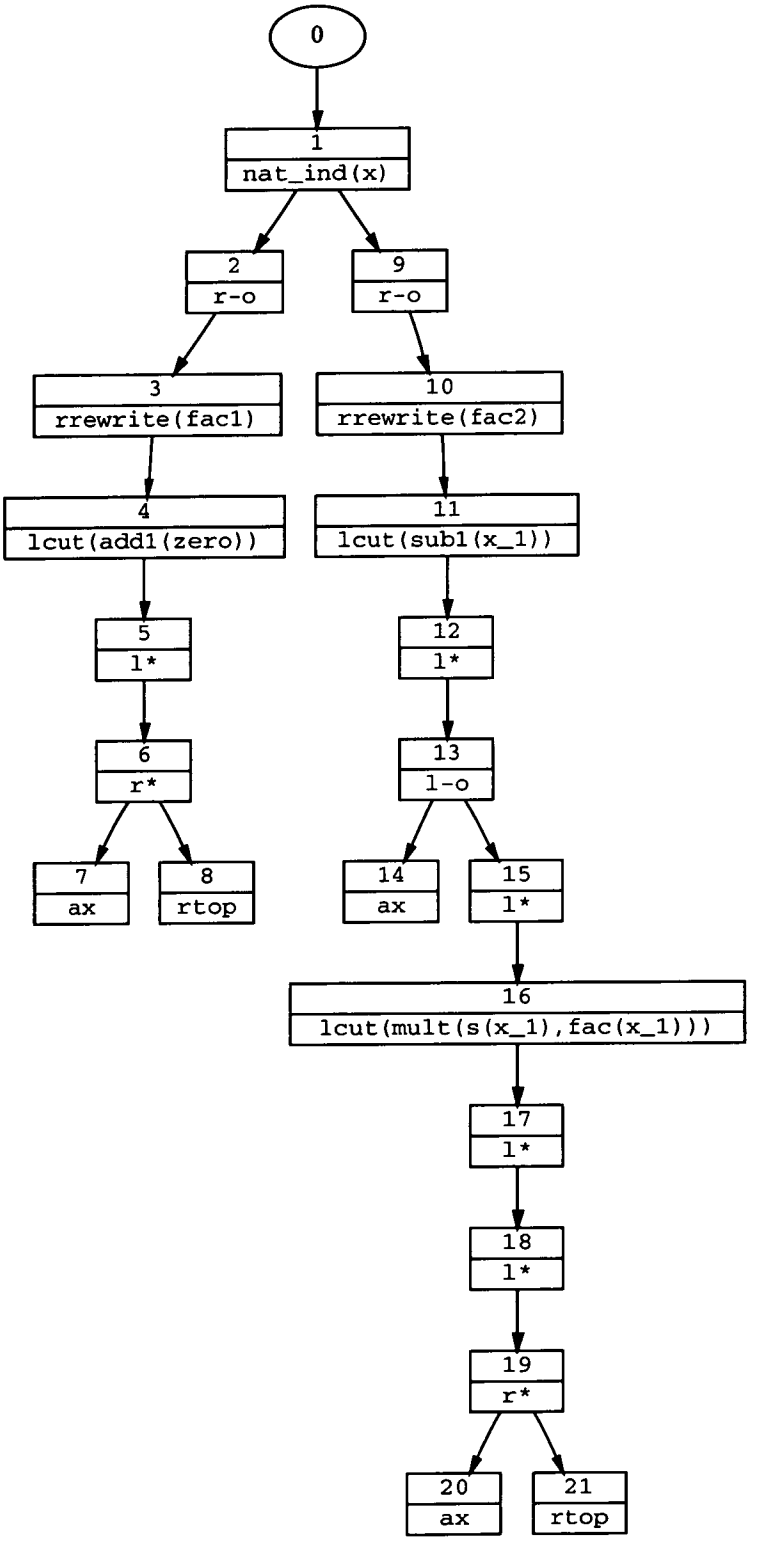
B.3.3 Rewrite rules

$$\begin{array}{lcll} nat(fac(zero)) & \rightarrow & nat(s(zero)) & (fac1) \\ nat(fac(s(x))) & \rightarrow & nat(mu(s(x), fac(x))) & (fac2) \end{array}$$

B.3.4 Plan

```
λx_2.
  nat_rec(x_2,
    λh3.
      let add1∘h3 be h7*h8 in h8*erase,
    λh1.λx_1.λh113.
          let sub1∘h113 be h118*h119 in
            let h1∘h118 be h123*h124 in
              let mult∘h119*h123 be h137*h138 in
                let h138 be h141*h142 in h142*erase)
```

B.3.5 Proof tree



B.4 Fibonacci

B.4.1 Problem specification

$$\vdash \forall x.(\exists y.fib(x,y)) \otimes (\exists z.fib(s(x),z)) \otimes \top$$

B.4.2 Axioms

$$\vdash \texttt{fib0} : \quad \multimap \quad fib(zero, zero)$$

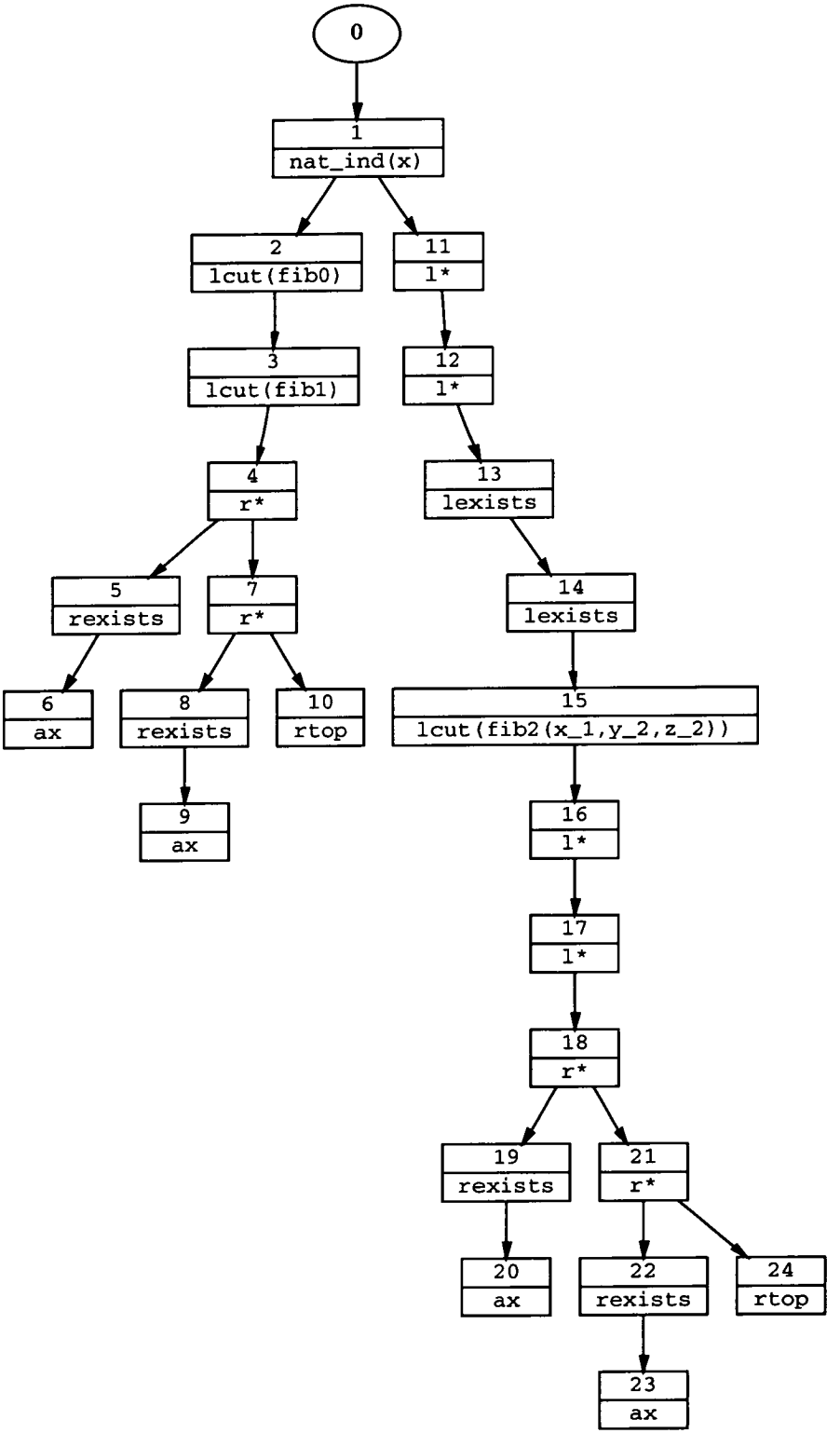
$$\vdash \texttt{fib1} : \quad \multimap \quad fib(s(zero), s(zero))$$

$$\vdash \texttt{fib2(x,y1,y2)} : \quad fib(x,y1) \otimes fib(s(x),y2) \quad \multimap \quad \left[\begin{array}{c} fib(x,y1) \\ \otimes \\ fib(s(x),y2) \\ \otimes \\ fib(s(s(x)),add(y1,y2)) \end{array}\right]$$

B.4.3 Plan

```
λx_2.
  nat_rec(x_2,
    fib0*fib1*erase,
    λh1.λx_1.
        let h1 be h9*h10 in
          let h10 be h13*h14 in
            let fib2∘h9*h13 be h20*h21 in
              let h21 be h24*h25 in h24*h25*erase)
```

B.4.4 Proof tree



B.5 Boat

This is a river crossing problem. There are n adults and 2 children wishing to cross from the west to the east bank of the river. A boat is available, which may only take 1 adult or 2 children.

B.5.1 Problem specification

$$\vdash \forall n. \left(\left[\begin{array}{c} inboat(nil) \\ \otimes \\ at(boat, west) \\ \otimes \\ at(crowd(n), west) \\ \otimes \\ at(crowd(zero), east) \\ \otimes \\ at(person(child), west) \\ \otimes \\ at(person(child), west) \end{array} \right] \multimap \left[\begin{array}{c} inboat(nil) \\ \otimes \\ at(boat, east) \\ \otimes \\ (\exists m. at(crowd(m), west)) \\ \otimes \\ at(crowd(n), east) \\ \otimes \\ at(person(child), east) \\ \otimes \\ at(person(child), east) \end{array} \right] \right)$$

B.5.2 Axioms

$$\vdash \texttt{leavecrowd} : \quad at(crowd(s(n)), pl) \multimap \left[\begin{array}{c} at(person(adult), pl) \\ \otimes \\ at(crowd(n), pl) \end{array}\right]$$

$$\vdash \texttt{joincrowd} : \quad \left[\begin{array}{c} at(person(adult), pl) \\ \otimes \\ at(crowd(n), pl) \end{array}\right] \multimap at(crowd(s(n)), pl)$$

$$\vdash \texttt{embark(pn, pl)} : \quad \left[\begin{array}{c} at(person(pn), pl) \\ \otimes \\ at(boat, pl) \\ \otimes \\ inboat(nil) \end{array}\right] \multimap \left[\begin{array}{c} at(boat, pl) \\ \otimes \\ inboat(person(pn) :: nil) \end{array}\right]$$

$$\vdash \texttt{embark(child, pl)} : \quad \left[\begin{array}{c} at(person(child), pl) \\ \otimes \\ at(boat, pl) \\ \otimes \\ inboat(person(child) :: nil) \end{array}\right] \multimap \left[\begin{array}{c} at(boat, pl) \\ \otimes \\ inboat(person(child) :: \\ person(child) :: \\ nil) \end{array}\right]$$

$$\vdash \texttt{disembark(pn, pl)} : \quad \left[\begin{array}{c} at(boat, pl) \\ \otimes \\ inboat(person(pn) :: rest) \end{array}\right] \multimap \left[\begin{array}{c} at(boat, pl) \\ \otimes \\ inboat(rest) \\ \otimes \\ at(person(pn), pl) \end{array}\right]$$

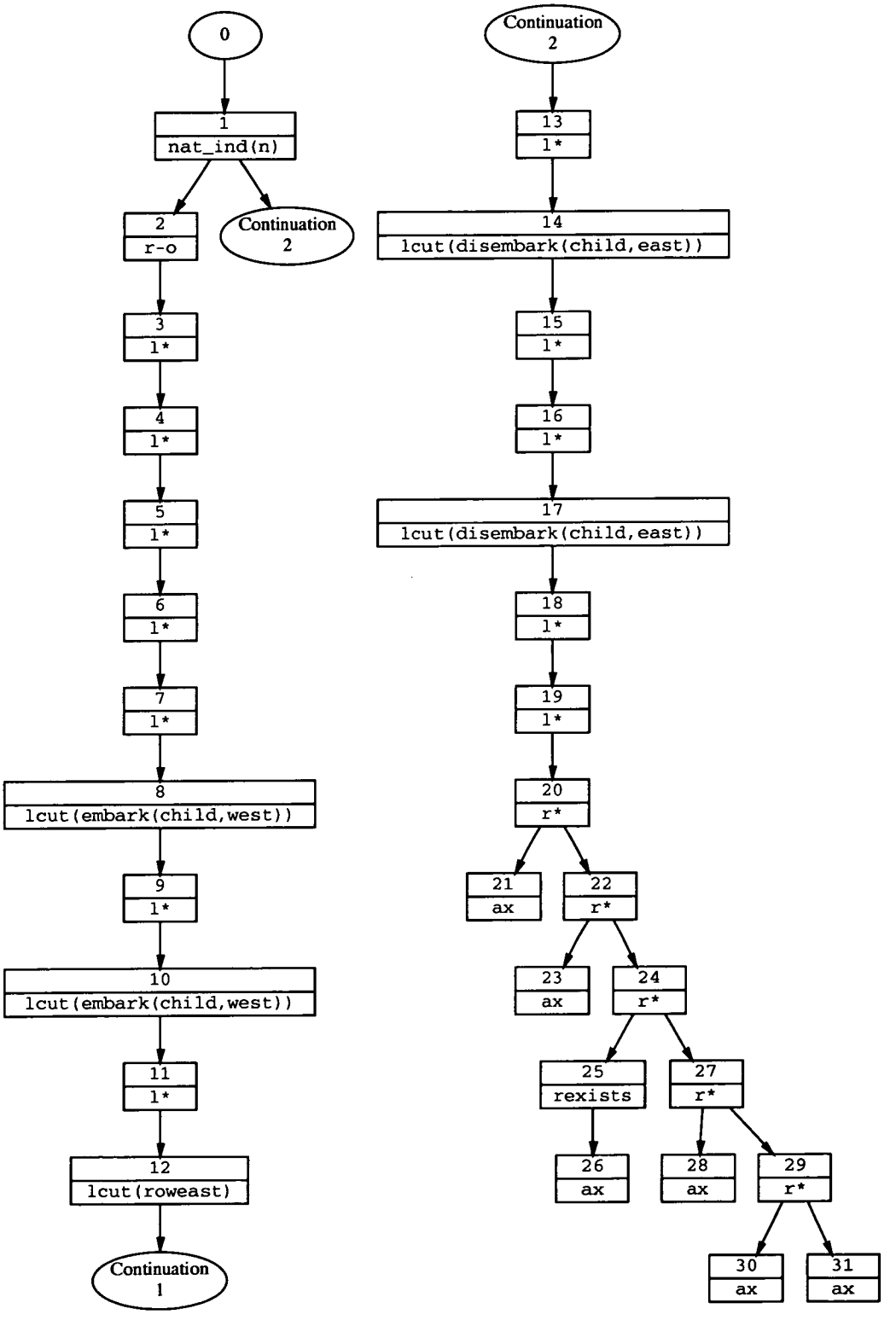
$$\vdash \texttt{roweast} : \quad \left[\begin{array}{c} at(boat, west) \\ \otimes \\ inboat(person(pn) :: rest) \end{array}\right] \multimap \left[\begin{array}{c} at(boat, east) \\ \otimes \\ inboat(person(pn) :: rest) \end{array}\right]$$

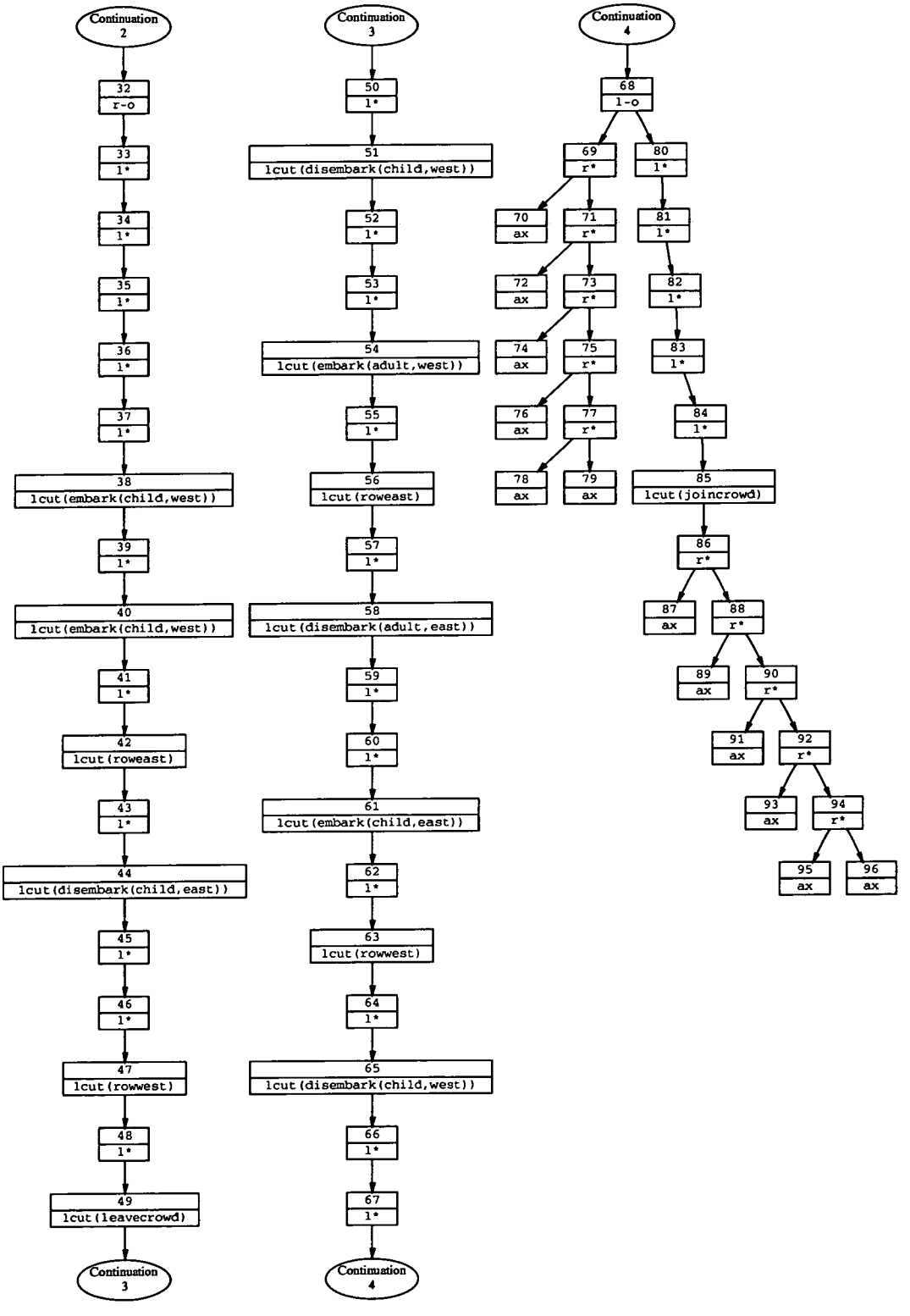
$$\vdash \texttt{rowwest} : \quad \left[\begin{array}{c} at(boat, east) \\ \otimes \\ inboat(person(pn) :: rest) \end{array}\right] \multimap \left[\begin{array}{c} at(boat, west) \\ \otimes \\ inboat(person(pn) :: rest) \end{array}\right]$$

B.5.3 Plan

```
λn_2.
 nat_rec(n_2,
  λh2.
   let h2 be h3*h4 in
    let h4 be h5*h6 in
     let h6 be h7*h8 in
      let h8 be h9*h10 in
       let h10 be h11*h12 in
        let embark∘h3*h5*h12 be h14*h15 in
         let embark∘h11*h14*h15 be h17*h18 in
          let roweast∘h17*h18 be h20*h21 in
           let disembark∘h20*h21 be h23*h24 in
            let h24 be h25*h26 in
             let disembark∘h23*h25 be h28*h29 in
              let h29 be h30*h31 in h30*h28*h7*h9*h31*h26,
  λh1.λn_1.λh32.
     let h32 be h33*h34 in
      let h34 be h35*h36 in
       let h36 be h37*h38 in
        let h38 be h39*h40 in
         let h40 be h41*h42 in
          let embark∘h33*h35*h42 be h44*h45 in
           let embark∘h41*h44*h45 be h47*h48 in
            let roweast∘h47*h48 be h50*h51 in
             let disembark∘h50*h51 be h53*h54 in
              let h54 be h55*h56 in
               let rowwest∘h53*h55 be h58*h59 in
                let leavecrowd∘h37 be h61*h62 in
                 let disembark∘h58*h59 be h64*h65 in
                  let h65 be h66*h67 in
                   let embark∘h61*h64*h66 be h69*h70 in
                    let roweast∘h69*h70 be h72*h73 in
                     let disembark∘h72*h73 be h75*h76 in
                      let h76 be h77*h78 in
                       let embark∘h56*h75*h77 be h80*h81 in
                        let rowwest∘h80*h81 be h83*h84 in
                         let disembark∘h83*h84 be h86*h87 in
                          let h87 be h88*h89 in
                           let h1∘h88*h86*h62*h39*h89*h67 be h91*h92 in
                            let h92 be h93*h94 in
                             let h94 be h95*h96 in
                              let h96 be h97*h98 in
                               let h98 be h99*h100 in
                                 h91*h93*h95*(joincrowd∘h78*h97)*h100*h99)
```

B.5.4 Proof tree




Continuation 2
32 r-o
33 1*
34 1*
35 1*
36 1*
37 1*
38 lcut(embark(child,west))
39 1*
40 lcut(embark(child,west))
41 1*
42 lcut(roweast)
43 1*
44 lcut(disembark(child,east))
45 1*
46 1*
47 lcut(rowwest)
48 1*
49 lcut(leavecrowd)
Continuation 3
Continuation 3
50 1*
51 lcut(disembark(child,west))
52 1*
53 1*
54 lcut(embark(adult,west))
55 1*
56 lcut(roweast)
57 1*
58 lcut(disembark(adult,east))
59 1*
60 1*
61 lcut(embark(child,east))
62 1*
63 lcut(rowwest)
64 1*
65 lcut(disembark(child,west))
66 1*
67 1*
Continuation 4
Continuation 4
68 1-o
69 r*
80 1*
70 ax
71 r*
81 1*
72 ax
73 r*
82 1*
74 ax
75 r*
83 1*
76 ax
77 r*
84 1*
78 ax
79 ax
85 lcut(joincrowd)
86 r*
87 ax
88 r*
89 ax
90 r*
91 ax
92 r*
93 ax
94 r*
95 ax
96 ax

B.6 Gripper

This is an adaptation of the gripper domain from the AIPS competition corpus. The domain features a robot with two grippers. The goal is to transfer a number of balls from one room to another.

We use natural numbers to represent the number of balls at each location, and provide an obvious generalisation. Given the simple induction rule for natural numbers, the planner is able to find the plan for the balls one-by-one, using only a single a hand.

B.6.1 Problem specification

$$\vdash \forall n.\forall m. \left(\left[\begin{array}{c} at(n, rooma) \\ \otimes \\ at(m, roomb) \\ \otimes \\ atrobby(rooma) \\ \otimes \\ free(left) \\ \otimes \\ free(right) \end{array} \right] \multimap at(plus(n,m), roomb) \otimes \top \right)$$

B.6.2 Axioms

$$\vdash \texttt{move(from, to)} : \quad atrobby(from) \multimap atrobby(to)$$

$$\vdash \texttt{pick(n, r, g)} : \quad \left[\begin{array}{c} at(s(n), r) \\ \otimes \\ atrobby(r) \\ \otimes \\ free(g) \end{array} \right] \multimap \left[\begin{array}{c} at(n, r) \\ \otimes \\ atrobby(r) \\ \otimes \\ holdsball(g) \end{array} \right]$$

$$\vdash \texttt{drop(n, r, g)} : \quad \left[\begin{array}{c} at(n, r) \\ \otimes \\ atrobby(r) \\ \otimes \\ holdsball(g) \end{array} \right] \multimap \left[\begin{array}{c} at(s(n), r) \\ \otimes \\ atrobby(r) \\ \otimes \\ free(g) \end{array} \right]$$

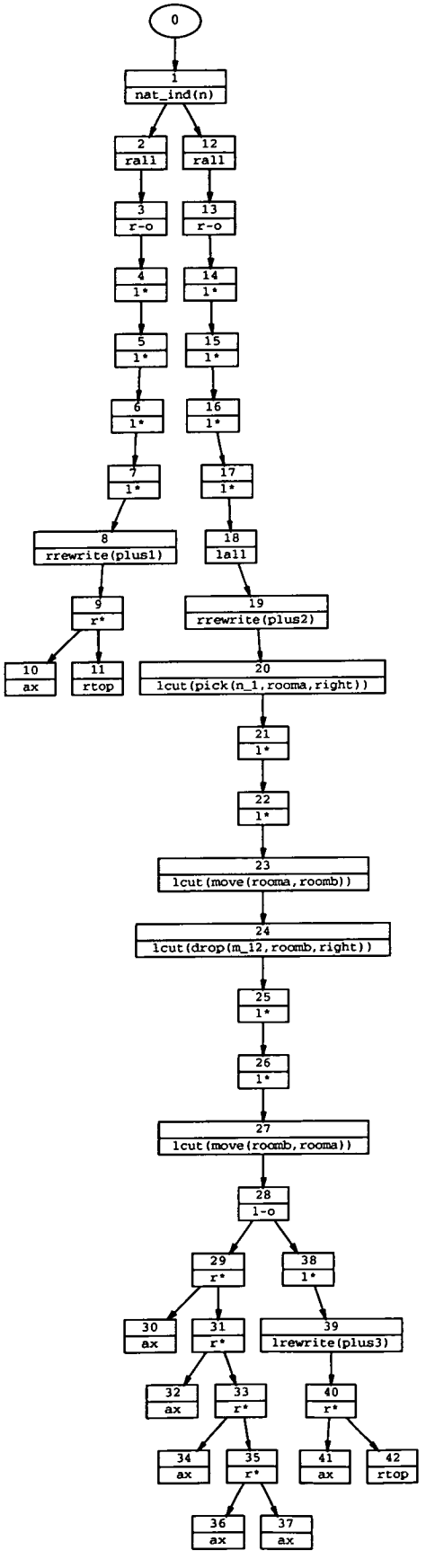
B.6.3 Rewrite rules

$$\begin{array}{lcll} plus(zero, y) & \rightarrow & y & (plus1) \\ plus(s(x), y) & \rightarrow & s(plus(x, y)) & (plus2) \\ plus(x, s(y)) & \rightarrow & s(plus(x, y)) & (plus3) \end{array}$$

B.6.4 Plan

```
λn_2.
 nat_rec(n_2,
  λm_2.λh3.
    let h3 be h6*h7 in
     let h7 be h10*h11 in
      let h11 be h14*h15 in
       let h15 be h18*h19 in h10*erase,
  λh1.λn_1.λm_12.λh877.
      let h877 be h880*h881 in
       let h881 be h884*h885 in
        let h885 be h888*h889 in
         let h889 be h892*h893 in
          let pick∘h880*h888*h893 be h1411*h1412 in
           let h1412 be h1415*h1416 in
            let drop∘h884*h1416*move∘h1415 be h1575*h1576 in
             let h1576 be h1579*h1580 in
              let h1∘s(m_12)∘
                  h1411*h1575*(move∘h1579)*h892*h1580)))
               be h1586*h1587
               in h1586*erase)
```

B.6.5 Proof tree



B.7 Gripper2

This version of the problem uses the *nat_ind2* induction rule (Section 6.3.2), which allows both robot hands to be used.

B.7.1 Problem specification

$$\vdash \forall n.\forall m. \left(\left[\begin{array}{c} at(n, rooma) \\ \otimes \\ at(m, roomb) \\ \otimes \\ atrobby(rooma) \\ \otimes \\ free(left) \\ \otimes \\ free(right) \end{array} \right] \multimap at(plus(n,m), roomb) \otimes \top \right)$$

B.7.2 Axioms

$$\vdash \texttt{move(from, to)} : \quad atrobby(from) \multimap atrobby(to)$$

$$\vdash \texttt{pick(n, r, g)} : \quad \left[\begin{array}{c} at(s(n), r) \\ \otimes \\ atrobby(r) \\ \otimes \\ free(g) \end{array} \right] \multimap \left[\begin{array}{c} at(n, r) \\ \otimes \\ atrobby(r) \\ \otimes \\ holdsball(g) \end{array} \right]$$

$$\vdash \texttt{drop(n, r, g)} : \quad \left[\begin{array}{c} at(n, r) \\ \otimes \\ atrobby(r) \\ \otimes \\ holdsball(g) \end{array} \right] \multimap \left[\begin{array}{c} at(s(n), r) \\ \otimes \\ atrobby(r) \\ \otimes \\ free(g) \end{array} \right]$$

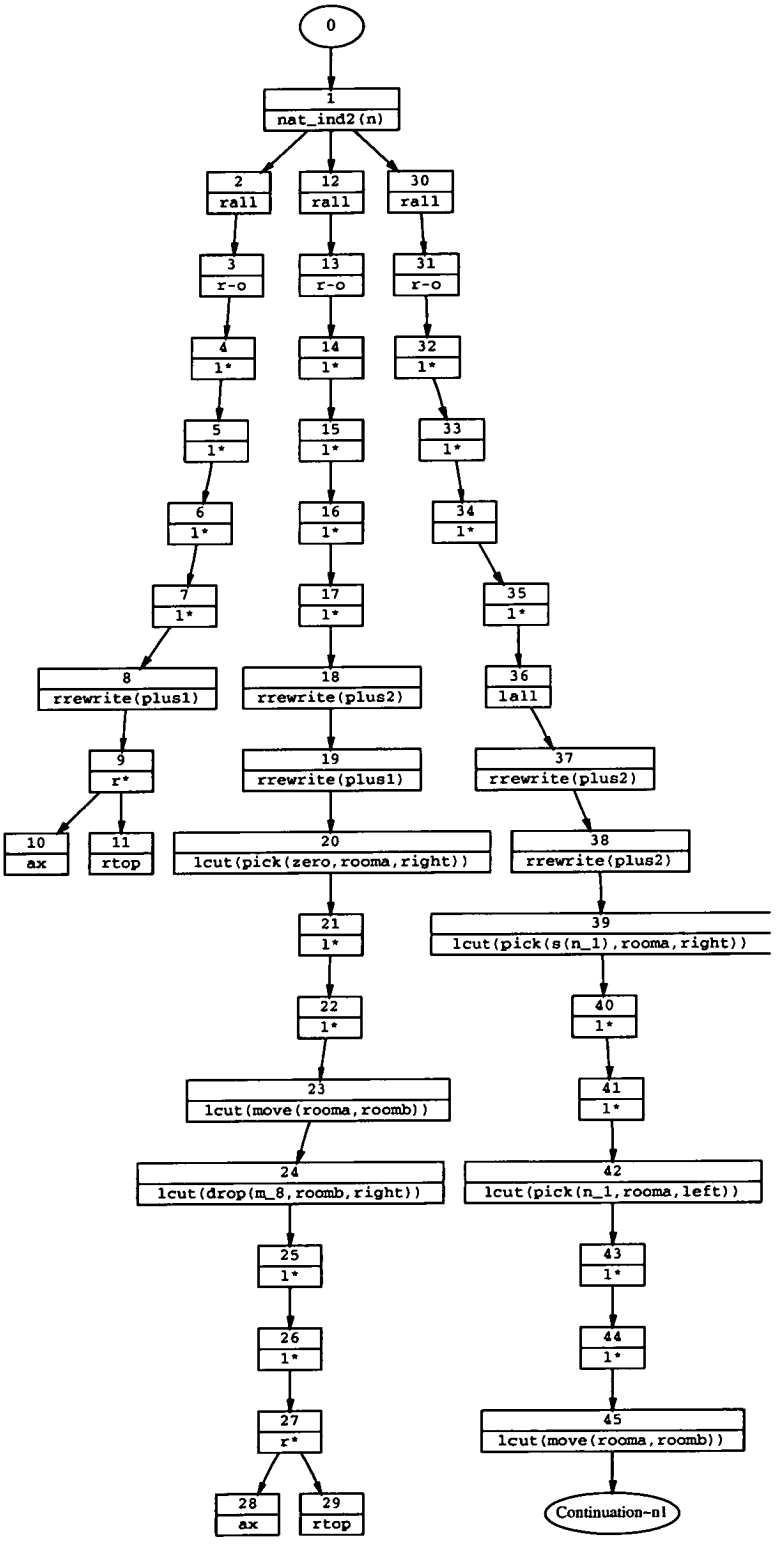
B.7.3 Rewrite rules

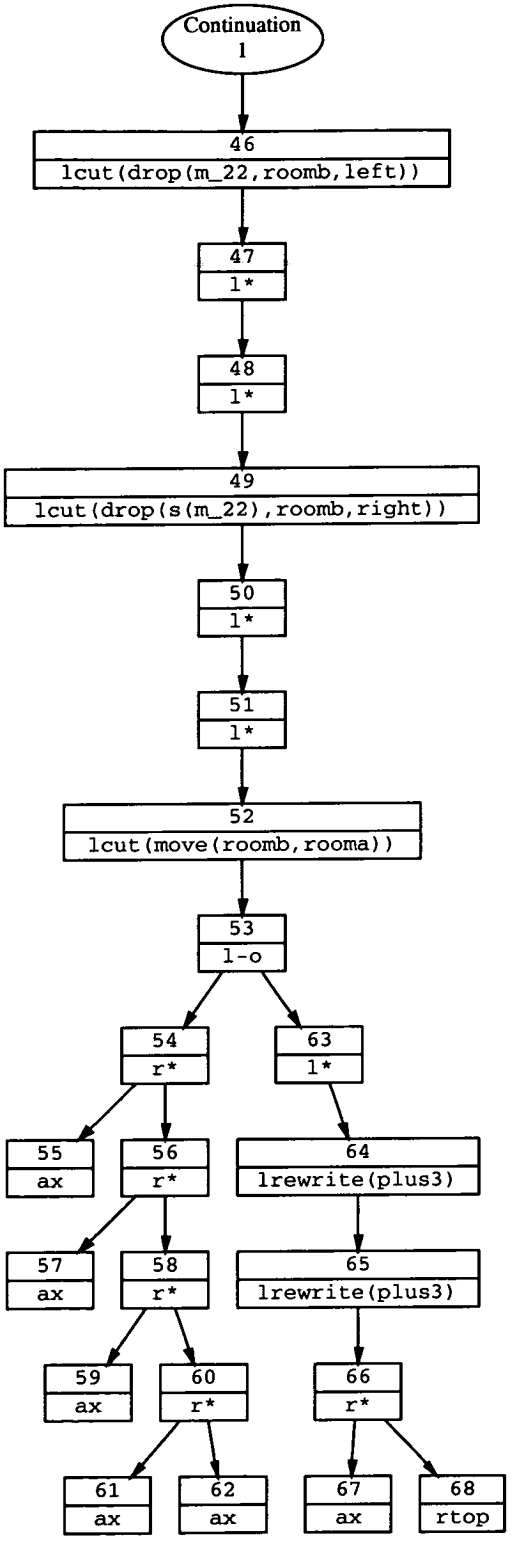
$$\begin{array}{lcll} plus(zero, y) & \rightarrow & y & (plus1) \\ plus(s(x), y) & \rightarrow & s(plus(x, y)) & (plus2) \\ plus(x, s(y)) & \rightarrow & s(plus(x, y)) & (plus3) \end{array}$$

B.7.4 Plan

```
λn_2.
 nat_rec2(n_2,
  λm_2.λh3.
    let h3 be h6*h7 in
     let h7 be h10*h11 in
      let h11 be h14*h15 in
       let h15 be h18*h19 in h10*erase,
  λm_8.λh134.
    let h134 be h137*h138 in
     let h138 be h141*h142 in
      let h142 be h145*h146 in
       let h146 be h149*h150 in
        let pick∘h137*h145*h150 be h210 in
         let h210 be h213*h214 in
          let h214 be h217*h218 in
           let move∘h217 be h219 in
            let drop∘h141*h218*h219 be h230 in
             let h230 be h233*h234 in
              let h234 be h237*h238 in h233*erase,
  λh1.λn_1.λm_22.λh14772.
      let h14772 be h14775*h14776 in
       let h14776 be h14779*h14780 in
        let h14780 be h14783*h14784 in
         let h14784 be h14787*h14788 in
          let pick∘h14775*h14783*h14788 be h24833 in
           let h24833 be h24836*h24837 in
            let h24837 be h24840*h24841 in
             let pick∘h14787*h24836*h24840 be h29167 in
              let h29167 be h29170*h29171 in
               let h29171 be h29174*h29175 in
                let move∘h29174 be h29177 in
                 let drop∘h14779*h29175*h29177 be h30380 in
                  let h30380 be h30383*h30384 in
                   let h30384 be h30387*h30388 in
                    let drop∘h24841*h30383*h30387 be h30616 in
                     let h30616 be h30619*h30620 in
                      let h30620 be h30623*h30624 in
                       let move∘h30623 be h30626 in
                        let h1∘s(s(m_22))∘
                            h29170*h30619*h30626*h30388*h30624
                         be h30630*h30631
                         in h30630*erase)
```

B.7.5 Proof tree




Continuation
1
46
lcut(drop(m_22,roomb,left))
47
l*
48
l*
49
lcut(drop(s(m_22),roomb,right))
50
l*
51
l*
52
lcut(move(roomb,rooma))
53
l-o
54
r*
63
l*
55
ax
56
r*
64
lrewrite(plus3)
57
ax
58
r*
65
lrewrite(plus3)
59
ax
60
r*
66
r*
61
ax
62
ax
67
ax
68
rtop

B.8 Socks

This is the socks problem described in [Bibel 86] and [Masseron *et al* 93]. Uncertain effects are involved, but not recursion.

Masseron's formulation has specific axioms for disposing of left-over resources (socks) when the goal has been acheived. The formulation used here makes of $\top$ instead.

B.8.1 Problem specification

$$\vdash hs \otimes hs \otimes hs \multimap \left[\begin{array}{c} bs \otimes bs \otimes \top \\ \oplus \\ ws \otimes ws \otimes \top \end{array}\right]$$

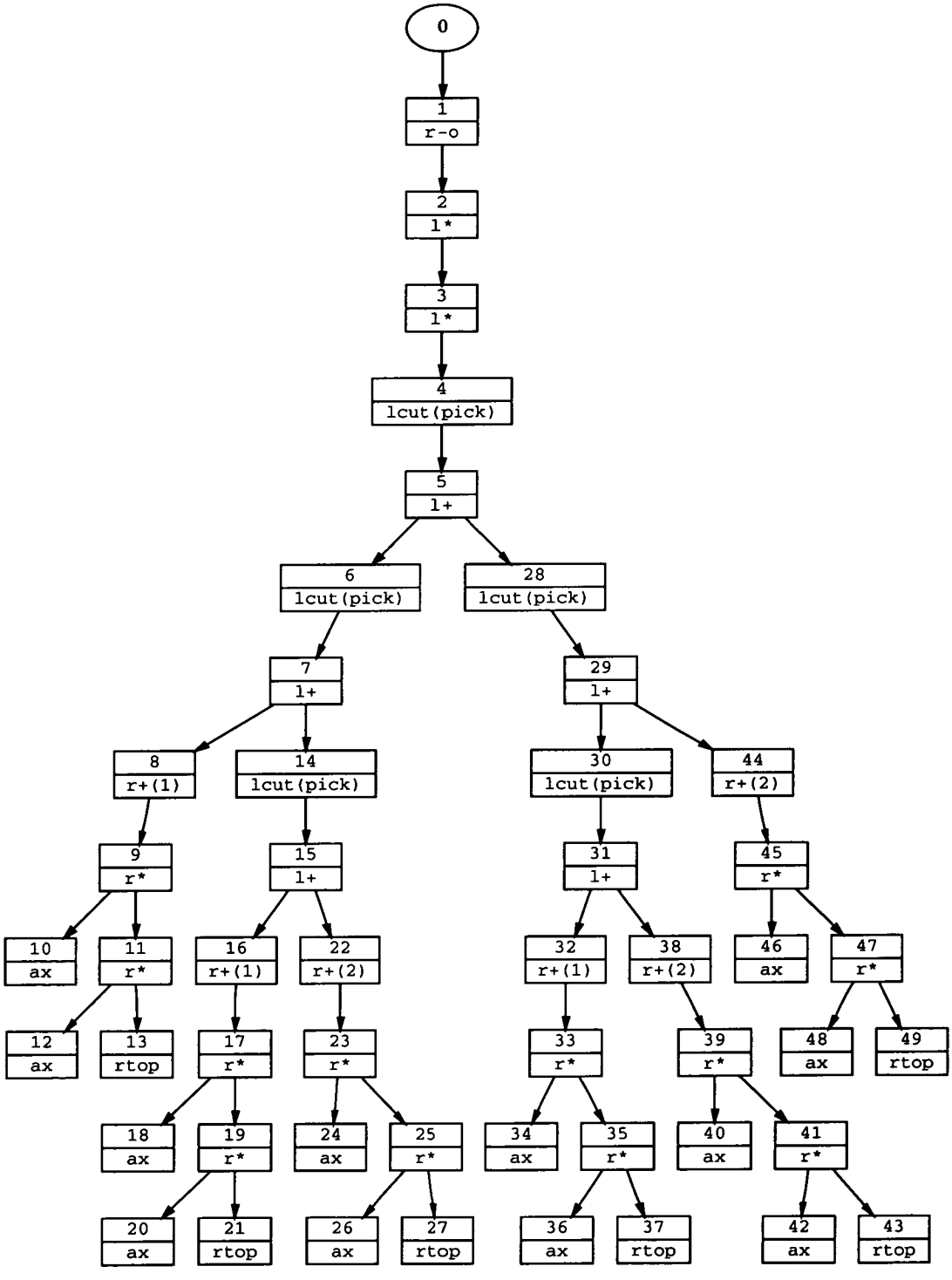
B.8.2 Axioms

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

B.8.3 Plan

```
λh82.
  let h82 be h85*h86 in
    let h86 be h89*h90 in
      case pick∘h90 of
        inl(h94) then
          case pick∘h89 of
            inl(h99) then inl(h99*h94*erase)
            inr(h100) then
              case pick∘h85 of
                inl(h104) then inl(h104*h94*erase)
                inr(h105) then inr(h105*h100*erase)
        inr(h95) then
          case pick∘h89 of
            inl(h109) then
              case pick∘h85 of
                inl(h114) then inl(h114*h109*erase)
                inr(h115) then inr(h115*h95*erase)
            inr(h110) then inr(h110*h95*erase)
```

B.8.4 Proof tree



B.9 Conformant socks

This is the socks problem in which we use actions with untestable conditional outcomes (See Section 8.7). This forces the planner to generate the conformant version of the plan.

B.9.1 Problem specification

$$\vdash hs \otimes hs \otimes hs \multimap \left[\begin{array}{c} bs \otimes bs \otimes \top \\ \oplus \\ ws \otimes ws \otimes \top \end{array}\right]$$

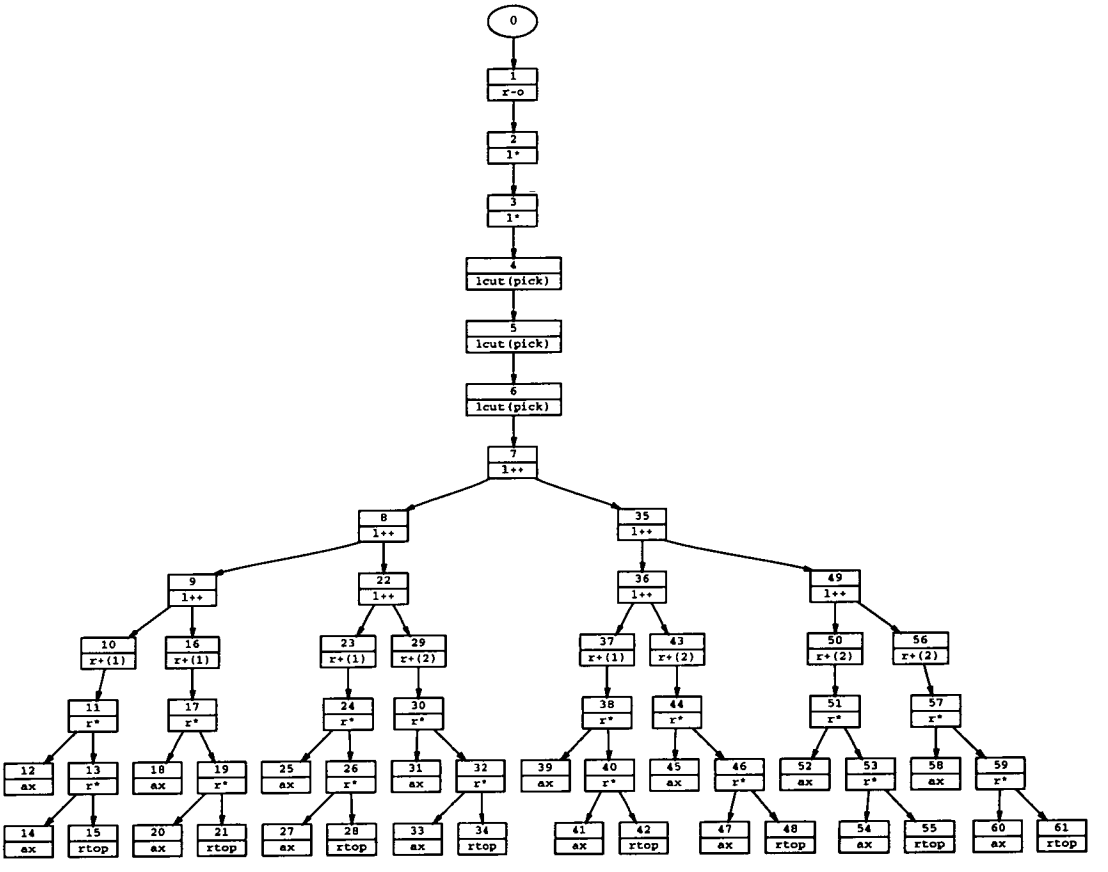
B.9.2 Axioms

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

B.9.3 Plan

```
λh574.
  let h574 be h577*h578 in
    let h578 be h581*h582 in
      let pick∘h582 be h583 in
        let pick∘h581 be h594 in
          let pick∘h577 be h675 in yields(bs*bs*top+ws*ws*top)
```

B.9.4 Proof tree



B.10 Omelette problem

This problem was introduced in [Levesque 96] as follows:

> We begin with a supply of eggs, some of which may be bad, but at least 3 of which are good. We have a bowl and a saucer, which can be emptied at any time. It is possible to break a new egg into the saucer, if it is empty, or into the bowl. By smelling a container, it is possible to tell if it contains a bad egg. Also, the contents of the saucer can be transferred to the bowl. The goal is to get 3 good eggs and no bad ones into the bowl.

In our representation, we do not separate the action with an uncertain outcome from the sensing action. A peano representation of natural numbers is used to count the eggs. The representation uses the form $eggs(place, n\_good, n\_total)$ to mean that there are (at least) n_good good eggs out of a total n_total eggs at $place$.

We (manually) separate the problem into the proof of a lemma, and the proof of the top level goal using the lemma. The lemma says that if there is at least one good egg in the fridge, then we can add it to the eggs in the bowl.

The `detect_impossible` action is used to eliminate the impossible situation where there are more good eggs than the total number of eggs.

B.10.1 Problem specification (for 3-egg problem)

$$\vdash \forall n\_total. \left(\begin{array}{c} \left[\begin{array}{c} eggs(fridge, s(s(s(zero))), n\_total) \\ \otimes \\ eggs(saucer, zero, zero) \\ \otimes \\ eggs(bowl, zero, zero) \end{array} \right] \\ \multimap \\ \left[\begin{array}{c} (\exists n\_remaining. eggs(fridge, zero, n\_remaining)) \\ \otimes \\ eggs(saucer, zero, zero) \\ \otimes \\ eggs(bowl, s(s(s(zero))), s(s(s(zero)))) \\ \otimes \\ \top \end{array} \right] \end{array} \right)$$

B.10.2 Axioms

$$\vdash \texttt{break\_into(cont)} : \left[\begin{array}{c} eggs(fridge, s(n1), s(n2)) \\ \otimes \\ eggs(cont, n3, n4) \end{array}\right] \multimap \left(\begin{array}{c} \left[\begin{array}{c} eggs(fridge, s(n1), n2) \\ \otimes \\ eggs(cont, n3, s(n4)) \end{array}\right] \\ \oplus \\ \left[\begin{array}{c} eggs(fridge, n1, n2) \\ \otimes \\ eggs(cont, s(n3), s(n4)) \end{array}\right] \end{array}\right)$$

$$\vdash \texttt{saucer\_to\_bowl} : \left[\begin{array}{c} eggs(saucer, s(zero), s(zero)) \\ \otimes \\ eggs(bowl, n3, n4) \end{array}\right] \multimap \left[\begin{array}{c} eggs(saucer, n1, n2) \\ \otimes \\ eggs(bowl, s(n3), s(n4)) \end{array}\right]$$

$$\vdash \texttt{saucer\_to\_bin} : eggs(saucer, n1, n2) \multimap eggs(saucer, zero, zero)$$

$$\vdash \texttt{detect\_impossible} : eggs(fridge, s(n), zero) \multimap \mathbf{0}$$

B.10.3 Lemma

This lemma is proved separately, and used as an axiom in the proof of the main goal. The lemma says that if we have at least one good egg in the fridge, we can get it into the bowl.

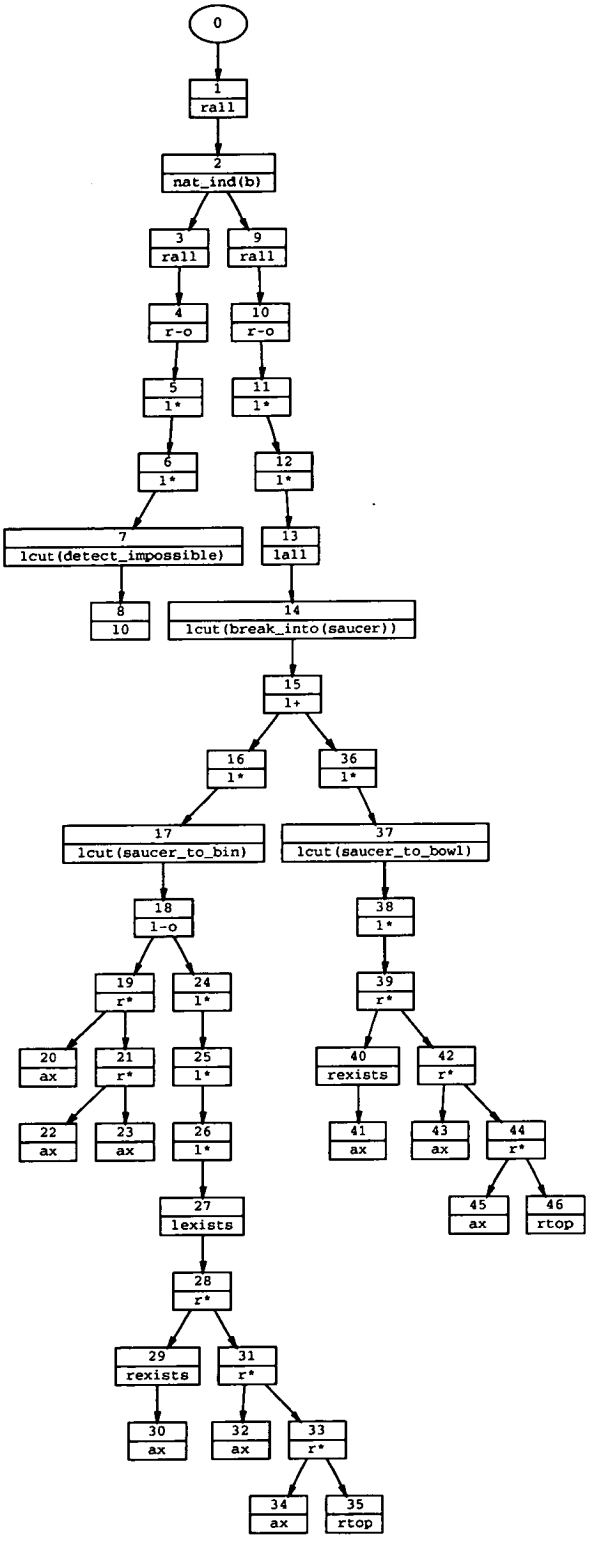
$$\vdash \texttt{lemma1} : \forall a.\forall b.\forall c. \left(\left[\begin{array}{c} eggs(fridge, s(a), b) \\ \otimes \\ eggs(saucer, zero, zero) \\ \otimes \\ eggs(bowl, c, c) \end{array}\right] \multimap \left[\begin{array}{c} (\exists d.eggs(fridge, a, d)) \\ \otimes \\ eggs(saucer, zero, zero) \\ \otimes \\ eggs(bowl, s(c), s(c)) \\ \otimes \\ \top \end{array}\right]\right)$$

B.10.4 Plan (for lemma)

The lemma was found automatically (excluding selection of induction rule).

```
λa_1.λb_2.
    nat_rec(b_2,
      λc_2.λh3.
          let h3 be h6*h7 in
            let h7 be h10*h11 in
              abort(detect_impossible∘h6),
      λh1.λb_1.λc_8.λh103.
              let h103 be h106*h107 in
                let h107 be h110*h111 in
                  case break_into∘h106*h110 of
                    inl(h131) then
                      let h131 be h135*h136 in
                        let
                          h1∘c_8∘h135*saucer_to_bin∘h136*h111
                        be h142*h143 in
                          let h143 be h146*h147 in
                            let h147
                            be h150*h151
                            in h142*h146*h150*erase
                    inr(h132) then
                      let h132 be h154*h155 in
                        let saucer_to_bowl∘h111*h155 be h160*h161 in
                          h154*h160*h161*erase)
```

B.10.5 Proof tree (for lemma)

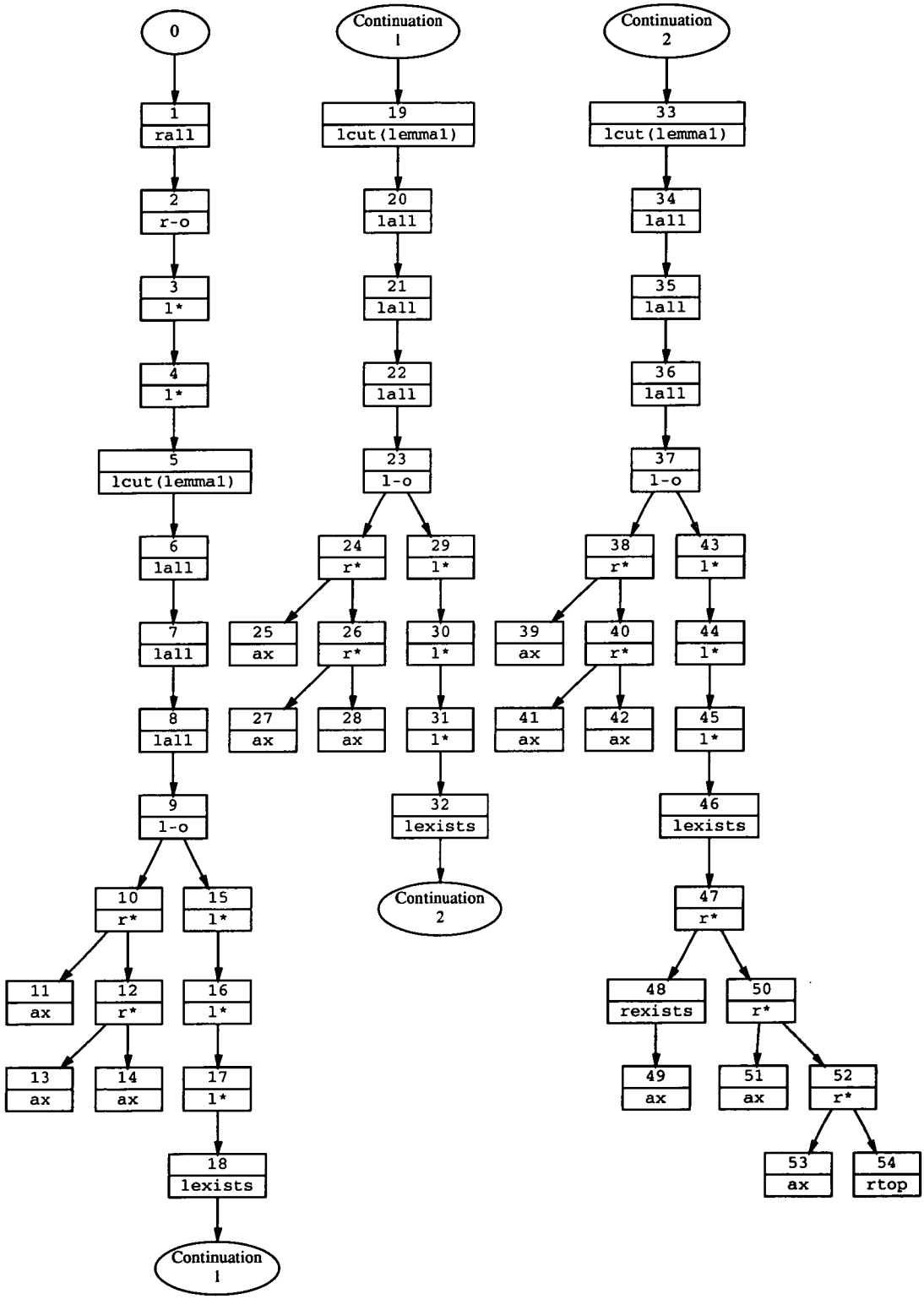


B.10.6 Plan (for 3-egg problem)

The 3-egg problem was solved automatically, given the lemma as an axiom.

```
λn_total_12.λh1914.
    let h1914 be h1917*h1918 in
     let h1918 be h1921*h1922 in
      let lemma1∘s(s(zero))∘n_total_12∘zero∘h1917*h1921*h1922
       be h3210*h3211 in
       let h3211 be h3214*h3215 in
        let h3215 be h3218*h3219 in
         let lemma1∘s(zero)∘d_282∘s(zero)∘h3210*h3214*h3218
          be h3360*h3361 in
          let h3361 be h3364*h3365 in
           let h3365 be h3368*h3369 in
            let lemma1∘zero∘d_294∘s(s(zero))∘h3360*h3364*h3368
             be h3389*h3390 in
             let h3390 be h3393*h3394 in
              let h3394 be h3397*h3398 in
               h3389*h3393*h3397*erase
```

B.10.7 Proof tree (for 3-egg problem)



B.11 Generalised omelette problem

This problem presents the more general version of the problem: to make an omelette from any number of good eggs. The proof relies on the same lemma as the 3-egg version of the problem (Section B.10.3).

This problem can be solved interactively, but the solution does not lie within the search space of the Lino search procedure.

The failure is caused by the interaction of quantifier rules. The Decompose phase of the planning algorithm commits too early to removing the universal quantifiers around the induction hypothesis. This prevents matching with the constant which is introduced by the existentially quantified effects.

B.11.1 Problem specification

Initially, the fridge contains a good eggs, out of a total of b eggs, and the bowl contains c good eggs. Introducing the variable c instead of using *zero* is a required generalisation.

$$\vdash \forall a.\forall b.\forall c. \left(\left[\begin{array}{c} eggs(fridge, a, b) \\ \otimes \\ eggs(saucer, zero, zero) \\ \otimes \\ eggs(bowl, c, c) \end{array} \right] \multimap \left[\begin{array}{c} (\exists d.eggs(fridge, zero, d)) \\ \otimes \\ eggs(saucer, zero, zero) \\ \otimes \\ eggs(bowl, plus(a, c), plus(a, c)) \\ \otimes \\ \top \end{array} \right] \right)$$

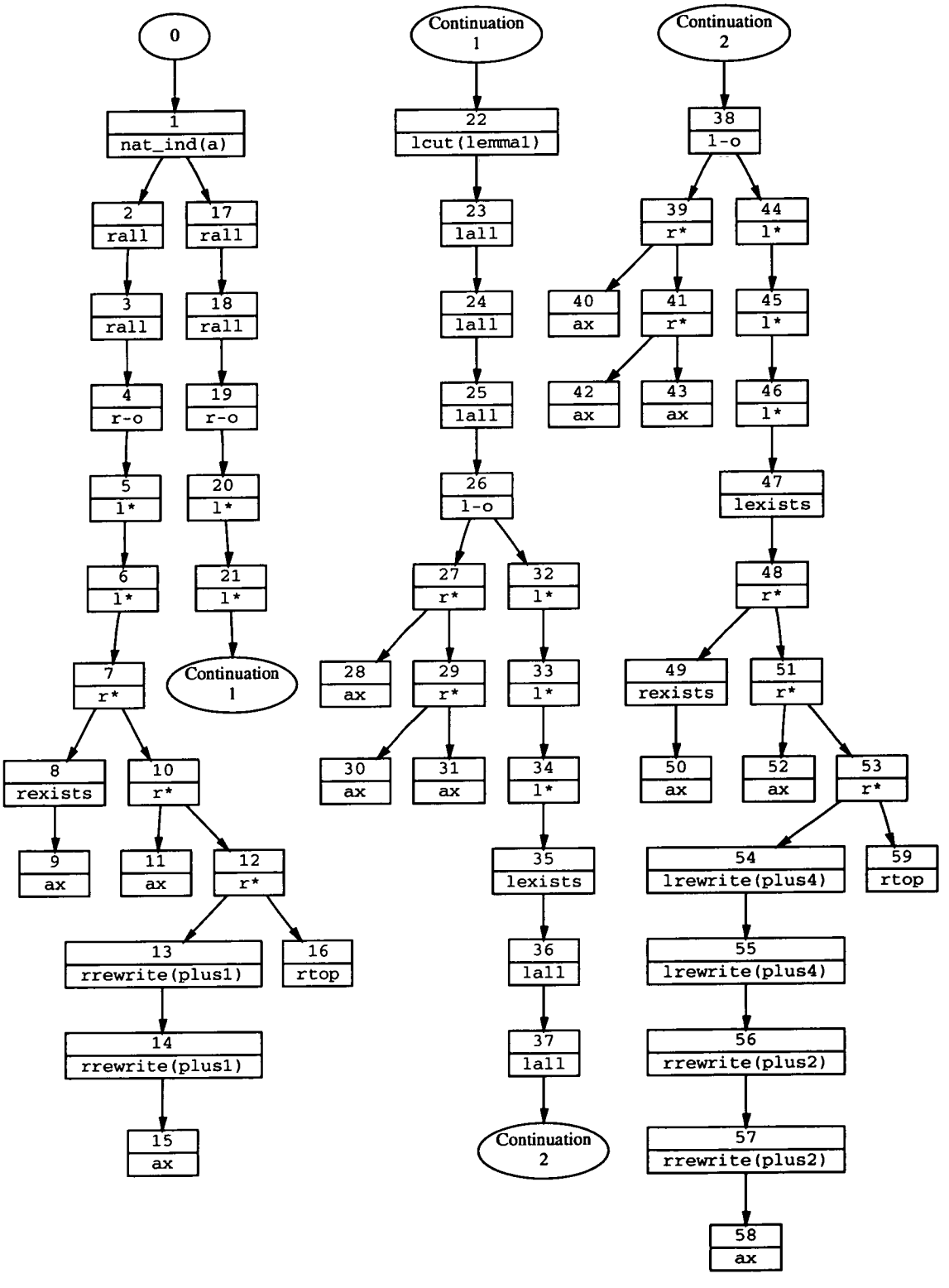
B.11.2 Rewrite rules

$$\begin{array}{lcll} plus(zero, y) & \rightarrow & y & (plus1) \\ plus(s(x), y) & \rightarrow & s(plus(x, y)) & (plus2) \\ plus(y, zero) & \rightarrow & y & (plus3) \\ plus(x, s(y)) & \rightarrow & s(plus(x, y)) & (plus4) \end{array}$$

B.11.3 Plan

```
λa_2.
  nat_rec(a_2,
    λb_1.λc_1.λh2.
          let h2 be h3*h4 in
            let h4 be h5*h6 in h3*h5*h6*erase,
    λh1.λa_1.λb_2.λc_2.λh7.
              let h7 be h8*h9 in
                let h9 be h10*h11 in
                  let lemma1∘a_1∘b_2∘c_2∘h8*h10*h11
                      be h17*h18 in
                    let h18 be h19*h20 in
                      let h20 be h21*h22 in
                        let h1∘d_1∘s(c_2)∘h17*h19*h21
                          be h26*h27 in
                          let h27 be h28*h29 in
                            let h29 be h30*h31 in h26*h28*h30*erase)
```

B.11.4 Proof tree



Bibliography

[Abramsky 93] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993. (Revised version of Imperial College Technical Report DoC 90/20).

[Baader & Nipkow 98] F. Baader and T. Nipkow. *Term Rewriting and All That*, chapter 11. Cambridge University Press, 1998.

[Baker 94] S. Baker. A new application for explanation-based generalisation withing automated reasoning. In Alan Bundy, editor, *12th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 177–191, Nancy, France, 1994. Springer-Verlag.

[Barber 97] A. Barber. *Linear Type Theories, Semantics and Action Calculi.* Unpublished PhD thesis, Department of Computing Science, University of Edinburgh, 1997.

[Bibel 83] W. Bibel. Matings in matrices. *CACM*, 26:844–852, 1983.

[Bibel 86] W. Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[Blum & Furst 95] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1636–1642, 1995.

[Boyer & Moore 79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Brüning *et al* 93] 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.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Carlsson *et al* 97] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.

[Chapman 87] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[Cresswell *et al* 99] S. Cresswell, A. Smaill, and J. Richardson. Deductive synthesis of recursive plans in linear logic. In *Proceedings of the Fifth European Conference on Planning ECP-99*, pages 252–263, Durham, UK, 1999.

[Dengler 96] D. Dengler. Customized plans transmitted by flexible refinement. In *12th European Conference on Artificial Intelligence*, 1996.

[Fikes & Nilsson 71] 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.

[Fox & Long 98] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367 – 421, 1998.

[Fronhöfer 97] B. Fronhöfer. Plan generation with the linear connection method. *Informatica*, 8(1), 1997.

[Galmiche & Boudinet 94] D. Galmiche and E. Boudinet. Proof search for programming in intuitionistic linear logic. In *Proceedings of CADE-12 Workshop on Proof Search in type-theoretic languages*, 1994.

[Gansner & North 00] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11), September 2000.

[Ghassem-Sani & Steel 91] 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.

[Ghassem-Sani 92] G. R. Ghassem-Sani. *Recursive Nonlinear Plans.* Unpublished PhD thesis, Department of Computer Science, University of Essex, 1992.

[Girard 87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Girard 95] J.-Y. Girard. Linear logic: its syntax and semantics. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Notes Series. Cambridge University Press, 1995.

[Gow 00] J. Gow. Position paper. In *Proceedings of 9th Workshop on Inductive Theorem Proving, 17th International Conference on Automated Deduction*, June 2000.

[Green 69] C. Green. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 219 – 239, 1969.

[Große *et al* 96] G. Große, S. Hölldobler, and J. Schneeberger. Linear deductive planning. *Journal of Logic and Computation*, 6(2):233–262, 1996.

[Harland & Pym 97] J. Harland and D. Pym. Resource-distribution via boolean constraint (extended abstract). In William McCune, editor, *Proceedings of CADE-14*, pages 222–336, Townsville, North Queensland, Australia, July 1997. Springer-Verlag LNCS 1249.

[Harland *et al* 96] J. Harland, D. Pym, and M. Winikoff. Programming in Lygon: An overview. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, pages 391–405, Munich, Germany, July 1996. Springer-Verlag LNCS 1101.

[Hesketh *et al* 92] 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*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 310–324, Saratoga Springs, NY, USA, June 1992.

[Hodas & Miller 94] 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.

[Hölldobler & Störr 98] S. Hölldobler and H.-P. Störr. Reasoning about complex actions. In *Reasoning about actions: foundations and applications, Tenth European Summer School on Logic, Language and Computation*, 1998.

[Ireland & Bundy 96] 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.

[Jacopin 93] 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.

[Japaridze 98] G. Japaridze. A formalisim for resource-oriented planning. IRCS Report 98-01, University of Pennsylvania, Institute for Cognitive Science, January 1998.

[Koehler 96] J. Koehler. Planning from second principles. *Artificial Intelligence*, 87:145 – 186, 1996.

[Kraan 94] I. Kraan. *Proof Planning for Logic Program Synthesis.* Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Kraan *et al* 96] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996. Also available from Edinburgh as DAI Research Paper 729.

[Kreitz *et al* 96] C. Kreitz, H. Mantel, J. Otten, and S. Schmitt. Connection-Based Proof Construction in Linear Logic. Technical Report AIDA–96–17, FG Intellektik, FB Informatik, TH Darmstadt, December 1996.

[Levesque 96] H. Levesque. What is planning in the presence of sensing?, 1996.

[Luo 94] Z. Luo. *Computation and Reasoning: A type theory for computer science.* International Series of Monographs on Computer Science. Oxford Science Publicatons, 1994.

[Manna & Waldinger 87] Z. Manna and R. Waldinger. How to clear a block: a theory of plans. *Journal of Automated Reasoning*, 3(4):343–377, 1987.

[Masseron 93] M. Masseron. Generating plans in linear logic II: A geometry of conjunctive actions. *Theoretical Computer Science*, 113:371–375, 1993.

[Masseron *et al* 93] M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic I: Actions and proofs. *Theoretical Computer Science*, 113(2):349–371, 1993.

[McCarthy & Hayes 69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence.* Edinburgh University Press, 1969.

[McDermott *et al* 98] D. McDermott *et al.* PDDL — the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[McDermott 87] D. McDermott. A critique of pure reason. *Computational Intelligence*, 3:151 – 160, 1987.

[Miller *et al* 91] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 1991.

[Muggleton 91] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[Nordström *et al* 90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf Type Theory.* Oxford University Press, 1990.

[Pednault 87] E. Pednault. Formulating multi agent dynamic world problems in the classical planning framework. In Georgeff and Lansky, editors, *Reasoning about actions and Plans*, 1987.

[Penberthy & Weld 92] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the third international conference on principles of knowledge representation and reasoning (KR-92)*, pages 103–114, Cambridge, MA, USA, October 1992.

[Peot & Smith 92] M. A. Peot and D. E. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, 1992.

[Pfenning 98] F. Pfenning. Graduate course on linear logic, 1998.

[Protzen 95] M. Protzen. *Lazy Generation of Induction Hypotheses and Patching Faulty Conjectures.* Unpublished PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, February 1995.

[Pryor & Collins 96] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[Pryor 95] L. Pryor. Decisions, decisions: Knowledge goals in planning. In *Proc. of AISB-95 Hyrid Problems, Hybrid Solutions*, pages 181–192, 1995.

[Sacerdoti 75] E. D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of IJCAI-75*, pages 206–214, Tbilisi, USSR, 1975. International Joint Conference on Artificial Intelligence.

[Schellinx 91] H. Schellinx. Some syntactical observations on linear logic. *Journal of Logic and Computation*, 1(4):537–559, September 1991.

[Slaney & Thiebaux 96] J. Slaney and S. Thiebaux. Linear time near-optimal planning in the blocks world. In *13th American National Conference on Artificial Intelligence (AAAI-96)*, pages 1208 – 1214, 1996.

[Smith & Weld 98] D. E. Smith and D. S. Weld. Conformant Graphplan. In *Proceedings of the 15th National Conference on A.I.*, 1998.

[Stephan & Biundo 93] W. Stephan and S. Biundo. A new logical framework for deductive planning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 32 – 38, 1993.

[Stephan & Biundo 95] W. Stephan and S. Biundo. Deduction-based refinement planning. DFKI Research Report RR-95-13, DFKI, Saarbrücken, Germany, 1995.

[Sussman 73] G. J. Sussman. *A Computational Model of Skill Acquisition.* Unpublished PhD thesis, Artificial Intelligence Laboratory, MIT, Cambridge, Ma., 1973.

[Tate 77] A. Tate. Generating project networks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 888 – 893, 1977.

[Troelstra 92] A. S. Troelstra. *Lectures in linear logic.* CSLI, 1992.

[Wallen 90] L. Wallen. *Automated Deduction in Non-Classical Logics.* MIT Press, 1990.

[Warren 76] D. Warren. Generating conditional plans and programs. In *Proceedings of AISB Summer Conference*, pages 344–354. University of Edinburgh, 1976.

[Weld 94] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[Weld *et al* 98] D. S. Weld, C. R. Anderson, and David E. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proceedings of AAAI-98*, 1998.