Verified Planning by Deductive Synthesis in Intuitionistic Linear Logic

Lucas Dixon and **Alan Smaill** and **Alan Bundy**
School of Informatics, University of Edinburgh, UK
{L.Dixon, A.Smaill, A.Bundy}@ed.ac.uk

Abstract

We describe a new formalisation in Isabelle/HOL of Intuitionistic Linear Logic and consider the support this provides for constructing plans by proving the achievability of given planning goals. The plans so found are provably correct, by construction. This representation of plans in linear logic provides a concise account of planning with sensing actions, allows the creation and deletion of objects, and solves the frame problem in an elegant way. Within this setting, we show how planning algorithms are implemented as search strategies within a theorem proving system. This allows us to provide a flexible methodology for developing search strategies that is independent of soundness issues. This feature is illustrated in two ways. Firstly, following ideas from logic programming, we show how a significant symmetry in search, caused by context splitting, can be pruned by using a derived inference rule. Secondly, we show how domain specific constraints on synthesis are supported and how they can be used to find contingent or conformant plans. We illustrate the approach with example planning scenarios.

1. Introduction

Linear Logic was introduced by Girard (1987) and is called a resource sensitive logic because assumptions can be consumed during inference. The intuitionistic version of the logic can be used to formalise planning problems in a way that elegantly solves the frame problem and provides a concise logical account of planning. It also provides a more expressive framework for planning: new objects can be created and deleted, non-deterministic and sensing actions can be expressed, and the exponentials in linear logic can be used to capture the notion of cached results.

In this paper, we describe a formalisation of Intuitionistic Linear Logic (ILL) within the higher-order logic of the Isabelle proof assistant. This includes support for incremental deductive synthesis by combining tools for proof automation with a representation of proof terms which serve as synthesised plans.

A language for such proof terms and semantics for their execution was proposed in Abramsky (1993). His work provides a computational interpretation of proofs in the logic, of which we use a variation. The formalisation, therefore, provides the machinery to function as a planner, where plans are synthesised by deduction from specifications in ILL.

Our development provides a platform for the further exploration of planning via deductive synthesis, and into the relationship between search algorithms for planning and search in theorem proving. This is done by building an embedding of ILL within a proof system with a small fixed logical kernel, namely Isabelle/HOL. This has a number of advantageous features:

1. It provides a soundness-preserving methodology for exploring the automation of planning by developing proof tactics for the theorem prover. This allows new search strategies to be considered without modification to any logical machinery. Existing tools for automation can also be used, such as Isabelle's simplifier and classical reasoner.
2. It allows domain specific constraints in the meta-logic to be attached to the derivations made within the linear logic. This additional expressivity can also be used to the guide proof search and plan synthesis. It also simplifies the formalisation by using theories from Isabelle/HOL. For example, we define a sequent's premises using the existing theory of multisets.
3. Derived theorems about ILL, such as the cut rule, can be proved and used within the formalisation and as part of the synthesis process, while the soundness of all such derivations rests only on the proof assistant's logical kernel.

The trade-off is that steps in planning are slowed down as they must go through the logical kernel of the proof system. We have implemented the full ILL, which is expressive, but undecidable in general (Lincoln et al., 1992). Search algorithms can be developed as 'proof-tactics' that improve proof automation by combining basic inference rules. For example, heuristics mixing forwards and backwards planning can also be built into search. This allows us to implement decision procedures for selected fragments of logic without worrying about compromising the correctness of found plans.

We illustrate the approach with one example from within a decidable fragment, for which we have developed an automatic planner, and two examples showing the use of con-

straints, to ensure conformant plans and to bound execution time respectively.

Overview. In §2 we describe related work, both in implementation and in relation to planning. We then describe the relationship between theorem proving in ILL and planning in §3, the Isabelle implementation and validation of plans in §4, and associated reasoning techniques in §5. Search strategies for automated planning are discussed in §6. §7 illustrates contingent and conformant planning by attaching constraints to plans. Finally we mention future work and conclude.

2. Related Work

Implementations

There have been previous formalisations of various fragments of Linear Logic in proof assistants. Sara Kalvala and Valeria de Paiva provided a basic implementation of intuitionistic propositional linear logic in Isabelle (Kalvala and de Paiva, 1995). They did not provide proof terms, and did not attempt significant automation. Without proof terms, there is no explicit representation of the plan. Philippe de Groote formalised the constant-free fragment of classical multiplicative additive linear logic within Isabelle (de Groote, 1995). He provided tactics in Isabelle to automate proof search in classical Linear Logic, with various heuristics, but no proof terms are involved, which makes it unsuitable for our purposes as plans cannot be extracted. Our work extends existing formalisations by providing a representation of proof terms, needed for the synthesis of plans, and improves on the automation by supporting lazy context splitting (see §5.) as well as additional constraints.

A Prolog implementation of ILL with proof terms was used to synthesise plans by Cresswell (2001). This includes forms of induction with associated proof terms that extend the ILL framework. It also provided a good level of automation. However, it lacks higher-order features and does not have a fixed logical kernel. It also lacks the various libraries present in Isabelle which simplify the presentation and aid further automation.

There are several logic programming languages which support theories within an appropriately restricted subset of the logic, e.g. Hodas and Miller (1994). This subset does not allow a direct formulation of planning where actions correspond to program clauses. For instance, the second example we present in §7. falls outside this subset. However, it does allow simple meta-interpreters to be written for the full language, as used by Dixon, Smaill, and Tsang (2009).

Framework logics that support linear connectives provide a suitable form of executable proof term to represent plans (Ishtiaq and Pym (1998); Cervesato and Pfenning (2002)). Implementation of these provide automated type checking and inference via a logic programming style of search. They have not so far been used to investigate the various search strategies available for ILL, or issues such as contingent versus conformant plans.

Relationship to Planning

A comparison of Linear Logic with other formalisms for planning has been presented by (Große, Hölldobler, and Schneeberger, 1996). The relationship between linear logic and planning has been explored on various occasions since the introduction of linear logic by Girard (1987). Work on the geometry of conjunctive actions by Masseron (1993) showed how a fragment of the logic given below can be used to build plans, represented as directed graphs, from proofs in the logic. An algorithm for realising Masseron's approach using a small decidable fragment of the language is described by Jacopin (1993); as with Masseron, a richer language is needed to deal with a realistic range of planning problems.

More recently, various authors have suggested that the formalism of ILL is well suited to the field of AI planning (Cresswell, 2001; Kanovich and Vauzeilles, 2001; Küngas, 2002). We extend this approach by the use of proof terms with a well-defined operational semantics formalised in a proof system.

AI planning systems based on the STRIPS approach manipulate a description of the state in a procedural fashion. To formalise the reasoning involved, a standard approach is use of a version of the situation calculus, where notions of *state* or *situation* appear explicitly in the object language; Levesque, Pirri, and Reiter (1998) provide such a language together with an associated programming language where state is not explicitly represented.

The situation calculus representation requires axioms to take care of the frame problem. These deal with fluents that are needed when coding a planning problem into a propositional satisfiability problem (Kautz and Selman, 1992). When using ILL, no such axioms are needed as the notion of resource consumption is built in to the logic. Thus we have a combination of reasoning in a well-defined logic, while not reasoning explicitly about state; indeed the basic language allows us to reason about non-deterministic or sensing actions (see §7.), and distributed execution, without any additional machinery. ILL also allows us to reason easily about the dynamic introduction and elimination of new entities associated with actions. It is possible, to some extent, to encode this idea in the situation calculus by means of entities that that have a fluent property of "existing", but note that this requires that all such entities that may appear at any time must then be made available statically in the problem specification — this is likely to be clumsy at best.

Compared to efficient planning techniques, our approach is much slower. There are two reasons for this: firstly, there has been relatively little work on efficient proof search for ILL, and secondly, because our approach verifies the correctness of the plans, it necessarily takes place within an interpreted system thus results a in linear, but large slowdown. We suggest ways to improve the efficiency, in §9., as further work.

3. Planning with Intuitionistic Linear Logic

In this work we tackle planning problems by treating them as theorem proving tasks.

ILL allows us to specify the possible actions and initial state concisely as statements written in ILL. The theorem proving task is then to show that some goal resources can be realised from the given resources, using actions as basic inference steps. Furthermore, this proof, that the goal can be achieved, has a computational reading: proofs are mapped mechanically in a typed (linear) functional programming language. For our purpose, these are executable plans. The details of how these programs are extracted from proofs can be found for example in Abramsky (1993).

The reading of "propositions as types" applies here: the result of theorem proving is that we have a proof that the program is well-typed, and this constitutes a proof that the plan is correct (provided, of course, that the planning problem has been accurately formalised).

We now review briefly the main ideas from linear logic and their connection to planning.

Formulas are treatred as resources which can be used up; thinking in terms of functional programming, a resource is information or data given to a program, and is used just once. A resource of the form `A` $\multimap$ `B` expresses an instance of an action that can use up `A` and produce `B`. `A` $\oplus$ `B` is a resource from which one of `A` or `B` can be produced, but we may not know which one. In contrast to this, `A` & `B` lets us choose which of `A` and `B` we would like, but we can only get one of them. From `A` $\otimes$ `B` we get both `A` and `B`. Lastly, `!A` lets us get arbitrarily many `A`'s.

In linear logic, the connectives typically come in pairs of *multiplicative* and *additive* depending on the way the the contexts are combined. For conjunction:

- if we want to use *both* of the conjuncts, we must split the resources used to establish the conjunction; this is the *multiplicative* connective, $\otimes$, which has the rules:

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

- if we want that just *one* of the conjuncts is used, we get the *additive* connective & which has the introduction rule:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}$$

The linear implication is written $\multimap$ and has the rules:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma, \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}$$

and disjunction (which is additive) has the rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}$$

An important case to consider when looking at the correspondence between proofs and plans is the following inference rule, with plan extraction notations added:

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash t \otimes u : A \otimes B}$$

This says that if u, t are plans that achieve goals from Γ, Δ respectively, then the *plan* (written overloading the symbol " $\otimes$ ") $t \otimes u$ will achieve the goal $A \otimes B$

Notice that the two subproblems are therefore logically *independent*. The corresponding execution can thus be distributed.

Linear logic is made more expressive by adding a unary connective "!" (called bang): $!A$ allows as many copies of A as may be needed (maybe none). This connective allows actions which can be performed arbitrarily many times to be naturally described. For example, to see how STRIPS operators are expressed in this language, consider a standard example:

```
operator:       stack(X,Y)

preconditions:  hold(X)
                clr(Y)

deletelist:     hold(X)
                clr(Y)

addlist:        clr(X)
                on(X,Y)
                empty
```

This corresponds to an ILL statement indicating which resources must be present before (and are thus used up), and which present afterwards:

```
stack(X,Y) : !(hold(X) ⊗ clr(Y) ⊸
                  empty ⊗ clr(X) ⊗ on(X,Y))
```

Here the claim is made that the action $stack(X, Y)$ uses up resources $hold(X) \otimes clr(Y)$ and generates resources $empty \otimes clr(X) \otimes on(X, Y)$. We use the bang connective to let the action be performed as many times as needed.

Non-determinism is treated with the $\oplus$ connective:

```
pick : hidden ⊸ (black ⊕ white)
```

characterises a "pick" action which uses up a hidden resource, generating either a black or a white (sock), but we do not know which will be returned.

In the ILL approach, no frame axioms are needed; the notion of state or situation is internalised in the semantics, so does not appear explicitly, unlike for example in the situation theory approach.

In summary, to perform planning using ILL we construct a sequent which describes our problem. This will have the form:

$$Actions, State \vdash Goals$$

Proving this shows the realisability of the goals from the resources available in the initial state. The corresponding proof can be viewed as plan/program which if executed successfully will produce the desired resources and so achieve the goal.

4. Valid Plans with Intuitionistic Linear Logic in Isabelle/HOL

Isabelle is a proof assistant that supports formal reasoning in a number of object logics (Paulson, 1994). These are formed and manipulated by Isabelle's intuitionistic, higher-order meta-logic, which supports polymorphic typing and performs type-inference. The basic operation for deriving

Type Expressions:
$type_1 \Rightarrow \ldots \Rightarrow type_n$ is the type of $\lambda(x_1 : type_1) \ldots \lambda(x_{n-1} : type_{n-1}).\ (y : type_n)$

Datatype Definitions:
`datatype` *type_name* `=`
constructor$_1$ *type_expr* ... *type_expr* (*syntax*)
`|` *constructor*$_2$ ...

Sequents:
$[\![\ assn_1\texttt{;}\ \ldots\texttt{;}\ assn_n\ ]\!] \Longrightarrow conclusion$

Quantification:
$\bigwedge x.$ `P(`x`)` is used for meta-level universal quantification.

Meta Variables: a meta variable x is written as "`?x`".

Figure 1: The Isabelle syntax used in this paper.

new theorems is a higher-order version of resolution. Additional proof tools can be written in ML, following the LCF methodology (Gordon, Milner, and Wadsworth, 1979). This requires that all functions that derive new theorems are decompose into applications of functions within the logical kernel.

The Isabelle system provides a library for higher-order logic including a simplifier and classical reasoner, as well as definitional packages to create datatypes and inductively defined sets. In Fig. 1, we show the syntax (taken from Isabelle) used in our formalisation.

Formalising ILL as a representation for plans in Isabelle/HOL helps ensure the correctness of plans in two important and distinct ways:

- Our formalisation is built as a conservative extension of Isabelle/HOL. Thus its validity, assuming the formalisation is correct, rests only on the implementation and axioms of Isabelle/HOL. By following the LCF methodology (Gordon, Milner, and Wadsworth, 1979), Isabelle's implementation has a strong argument for its correctness: it has a small fixed logical kernal on which all extensions are developed.
- Isabelle produces proof terms for which a small and independent proof checker can be written (Berghofer and Nipkow, 2000). The use of ILL also separates the search for a plan from the plan's correctness. To check that a plan does indeed solve the problem it is claimed to, the plan simply needs to be type-checked, which also can be done by a small independent type-checker. This allows the generated plans to be checked easily and efficiently, in much the same way as Isabelle's proofs. Such checking is independent of the correctness of the details of our formalisation and search machinery.

ILL Sequents, Types and Terms

We have formalised the dual context account of ILL following the work of (Barber, 1997). Each sequent expresses how a plan uses up some resources to produce some others. These sequents have two kinds of context: the first captures resources of which there are arbitrarily many and is called the non-linear or intuitionistic context; the second kind expresses ordinary resources which can be used up during planning and is called the linear context. Both kinds of context are formalised as a multiset of resources. This use of the existing multiset theory in Isabelle/HOL removes the need for explicit exchange rules.

A sequent of our embedded linear logic is characterised by the $\vdash$ constant which has type:

```
ires mset ⇒ lres mset ⇒ res ⇒ bool
```

where `mset` is the multiset type constructor, `ires` is a resource in the non-linear context, `lres` is one in the linear context, and `res` is the produced resource with its corresponding plan.

We use Isabelle's syntax machinery to support the usual notation. For example, we may write `{A,B} | {C} ⊢ p:D` to express that from arbitrarily many `A`'s and `B`'s, and a single `C` we can use up the `C` to get a `D` by performing the plan `p`. We will sometimes omit proof terms and the non-linear context for the sake of clarity.

The connectives of ILL express different kinds of resources. We characterise these using a HOL datatype:

```
datatype ltyp = 1 | ltyp & ltyp | ltyp ⊗ ltyp
     | ltyp ⊕ ltyp | ltyp ⊸ ltyp | ! ltyp
```

We use a HOL nominal datatype, which provides tools for managing the freshness of names (Urban and Tasson, 2005), to represent linear logic terms. This allows the datatype, trm, to expresses the different constructors from which a plan can be made:

```
nominal_datatype trm =
    top | var var | ivar ivar | star
  | let_star trm trm | trm ⊗ trm
  | let_tensor trm (≪var≫ ≪var≫ trm)
  | app trm trm | lam (≪var≫ trm)
  | choice trm trm
  | choosel trm (≪var≫ trm)
  | chooser trm (≪var≫ trm)
  | inl trm | inr trm
  | case_or trm (≪var≫ trm) (≪var≫ trm)
  | ! trm | let_bang trm (≪ivar≫ trm)
```

where $\ll X \gg\ Y$ expresses that there is a name for subterm of type X within Y. We use Isabelle's meta-level universal quantifier to express variable substitution. For example, the cut rule is traditionally presented as:

$$\frac{T \vdash a : A \qquad x : A,\ S \vdash b : B}{T, S \vdash b[a/x] : B}\ \text{Cut, x must be fresh}$$

In our formalisation, which also makes freshness conditions explicit, this becomes:

```
⟦ T ⊢ a : A;
  ⋀x. x freshin S ⟹ {x : A}, S ⊢ (b x) : B;
  y freshin (T and S) ⟧
⟹ T, S ⊢ (b a) : B
```

where $\bigwedge$ is Isabelle's meta-level universal quantifier, as shown in Figure 1. Because `b` occurs both within the context of the bound `x` and outside it, the subterm `x` stands for precisely every occurrence of the variable and, correspondingly, the term `b` can be viewed as the rest of the term. This

allows the framework to perform substitution of every occurrence of the variable in the plan. Using this approach we characterise ILL as an inductively defined set of derivations which correspond to the well-formed plans. The full set of rules in our formalisation can be found online[1].

5. Reasoning Techniques

We have developed tool support for planning with our formalisation. This assumes planning is performed by proving a goal sequent of the form:

```
initial_state ⊢ ?p : goal_state
```

where the meta-variable `?p` will become instantiated to the plan as the proof proceeds.

The available actions for planning are given by defining a new constant for the action and by providing an axiom that describes its effect. For example, we might specify that when a person eats food, it causes the eaten foodstuff to disappear and changes the person from being hungry to being full. The axiom specifying the eats action would then be:

```
const eats :: person ⇒ food ⇒ ltyp
⋀ p x. {} ⊢
   eats p x : x ⊗ hungry p ⊸ full p
```

The main tool support that has been developed provides tactics to support reasoning about linear logic by applying actions to the current state. In particular, one tactic to perform a forward planning step and one for performing a backward step. These provide a basis for automatic as well as interactive proof. We have also developed a tactic that checks to see if the current state can solve the goal by rearranging and decomposing the available resources without performing any further actions.

By combining these tactics we have also developed a simple automatic planner. When problems are within the STRIPS (Fikes, Hart, and Nilsson, 1972) fragment of ILL then it acts in a similar way to a traditional planner. However, we note that, being based on ILL, it can find a much richer set of plans. In particular, plans may contain conditional branching and may introduce new objects. Furthermore, arbitrary side conditions can be attached to planning. This allows a natural integration of constraint solving with planning. We give an example of this in §8..

Backward Reasoning

Backward reasoning involves applying a rule whose conclusion unifies with the conclusion of the goal sequent. We distinguish between two common kinds of backward reasoning:

Decomposition: breaks up the conclusion of a sequent. For example, a goal of the form `T ⊢ A & B` can be decomposed into the two subgoals `T ⊢ A` and `T ⊢ B`.

Actions: show how the conclusion could have been arrived at using an action axiom or resource in the non-linear context. For example, consider a goal of the form `T ⊢ B` and an action of the form `r : A ⊸ B`. Backward reasoning reduces the goal sequent to `T ⊢ A`, which corresponds to a backward step in planning.

[1] `http://dream.inf.ed.ac.uk/projects/e-Science/wfs.php`

Backward reasoning about actions can be handled easily by the following derived rule:

```
⟦ (r : A ⊸ B) ∈ D ; D | T ⊢ A ⟧
⟹ D | T ⊢ B
```

However, decomposition is more complicated as we describe below.

Lazy Context Splitting

The main difficulty with decomposition is having to split the context. This happens when the linear context must be split into several parts which are used in different subgoals. For example, this occurs with:

$$\frac{S \vdash A \qquad T \vdash B}{S, T \vdash A \otimes B} \otimes\mathbb{R}$$

The problem is that, at the point the rule is applied, it is not always clear which resources are needed to prove which goal. Searching over all possible ways to split the context is exponential. A better approach is to decide in a lazy fashion how the resources are allocated. This effectively removes a significant source of symmetry in the search space while maintaining completeness.

Boolean constraints have previously been used to deal with this issue in Harland and Pym (2003); Cresswell (2001); Cervesato, Hodas, and Pfenning (2000). The basic idea is that each resource is paired with a unique boolean variable indicating if it has been used. An extension of this, specially designed for logic programming proof search has been presented in López and Polakow (2004). We propose another solution which is independent of search and thus allows forwards and backwards planning steps to be interleaved. It also avoids expanding the trusted kernel with a constraints package or other complex efficiency measures by using following derived rule:

$$\frac{^{(1)}\ S, T \vdash A \otimes B \Longrightarrow M \vdash A \otimes B \quad ^{(2)}\ S \vdash A \quad ^{(3)}\ T \vdash B}{M \vdash A \otimes B}$$

When used backwards, this introduces `S` and `T` as new meta-variables. It allows us to perform backwards steps on subgoals (2) and (3), while also continuing forward reasoning on the resources in M of subgoal (1). Forward reasoning on subgoals (2) and (3) can also be performed by considering the linear context as including any resources associated through the meta variables S and T, namely those in M.

For example, consider using this rule to prove the goal:

```
{B, C} ⊢ X ⊗ (A ⊸ Y)
```

This would result in the following subgoals:

(1) `?S, ?T ⊢ X ⊗ (A ⊸ Y)`
`⟹ {B, C} ⊢ X ⊗ (A ⊸ Y)`

(2) `?S ⊢ X`

(3) `?T ⊢ A ⊸ Y`

Subgoal (3) can then be further decomposed using the ⊸-introduction rule to get the subgoal:

(4) `{A}, ?T ⊢ Y`

To illustrate forward reasoning, we consider having an action:

$$\texttt{r : (A} \otimes \texttt{B)} \multimap \texttt{E}$$

When forward reasoning on subgoal (4), any linear resources from the subgoal associated with ?T (subgoal 1) can be used, namely B and C. This is done by maintaining with each meta-variable the possible goals and thus resources it is associated with. In this example, forward reasoning reduces subgoal (4) to:

$$\{\texttt{E}\},\ \texttt{?T'} \vdash \texttt{Y}$$

As illustrated in this example, when forward reasoning is applied, a new meta variable must be introduced in order to maintain the lazy division of the linear context. This done by the ?T′ in the above example, and is performed by our proof machinery.

Decomposition

We decompose the conclusion of a goal to identify if the linear context contains the resources necessary to solve it. Our decomposition algorithm is defined as a recursive tactic that breaks up the conclusion to try to show that it is made up of the linear context. Depending on the syntactic form of the conclusion, it does the following:

1. If the conclusion is not made up of $\otimes$, $\multimap$, $\oplus$, or $\&$, then it looks at the resources associated with this goal to find one that *realises it* (see below). If no resource in the context realises the given one, then the subgoal is left to be solved later.
2. If the conclusion is of the form A $\multimap$ B it applies the $\multimap$-introduction rule which adds A to the context and requires that B is then shown. In this case no further decomposition is performed. Instead it results in a subgoal to be solved.
3. If the conclusion is of the form A $\otimes$ B, it splits the context lazily, as describe above, and then continues decomposition on both goals.
4. If the conclusion is of the form A $\oplus$ B, both the $\oplus$-introduction rules can be applied. The tactic searches over further decomposition of both possibilities.
5. If the conclusion is of the form A & B, it apples the &-introduction rule which results in two subgoals. Decomposition continues on both subgoals.

A resource A realises B if they unify, or if A is of the form $\texttt{A}_1$ & $\texttt{A}_2$ and B realises either $\texttt{A}_1$ or $\texttt{A}_2$. This effectively allows the choice of between $\texttt{A}_1$ and $\texttt{A}_2$ to be performed lazily. This is supported by the following derived rules:

fstL: "{A}, T ⊢ C ⟹ {A & B}, T ⊢ B"
sndL: "{B}, T ⊢ C ⟹ {A & B}, T ⊢ B"

Similarly, we also allow searching for resources in the context to include those in the non-linear context.

Forward Reasoning

Forward reasoning involves applying a rule whose premises unify with some (or all) of the resources available in a goal sequent and introduces the conclusion of the rule as a new resource. It is performed using the following derived variation of the modus-ponens rule for $\multimap$:

$$\frac{^{(1)}\{\} \vdash A \multimap B \quad ^{(2)}T \vdash A \quad ^{(3)}S, T = M \quad ^{(4)}\{B\}, S \vdash C}{M \vdash C}$$

Subgoal (1) is typically proved trivially by using axiom that specifies the action. Subgoal (2) is proved by decomposition, which instantiates T. Subgoal (3) is solved using a generic multiset equation solver which instantiates S to be the remaining resources. This leaves only subgoal (4) remaining, on which the proof attempt then continues.

As well as forward reasoning with actions, we also perform some decomposition of compound resources in the linear context. In particular, when this context contains resources of the form A $\oplus$ B, we apply the $\oplus$-elimination rule which results in two further subgoals, one with A in the context and the other with B. Resources of the form A $\otimes$ B are always decomposed using $\otimes$-elimination to give both resources. Those of the form A & B, are not decomposed. Instead, as mentioned earlier, they are included in the search for resources during backward decomposition of the conclusion. We assume that actions and resources in the non-linear context are either already decomposed, or are representing actions and thus of the form A $\multimap$ B, and therefore are applied by forward reasoning.

6. Planning by Proof Search with Tactics

Using the forward and backward reasoning tactics, we can easily define a tactic that acts as a simple planner. This searches by forward chaining interleaved with attempted decomposition to see if the goal sequent can be solved.

Our planning tactic searches breadth-first, depth-first, or using iterative deepening, over all possible applications of forward reasoning. The breadth-first search is complete for problems in the STRIPS fragment of ILL extended with non-deterministic resources: if there is a plan, a plan is found. Furthermore, it finds the smallest such plan, in terms of the number of actions, first.

A modification of this approach suggested by Kanovich and Vauzeilles (2001) provides a decision procedure. This highlights one of the main features of our approach: because planning is proof search, it can be developed incrementally while avoiding soundness issues. This separation of concerns gives rise to succinct code for planning search strategies that can easily be extended. For example, to write the breadth first search strategy required only 8 lines of ML code.

Example: The Synthesis of Workflows

Many web and grid-service workflows are currently created by writing programs in scripting languages that move data between the services. When such workflows involve expensive combinations of services, it is important to avoid errors in the composition. Providing machinery to create robust verified workflows is thus an important area of research (Bundy, Smaill, and Yang, 2003).

Planning has previously been used to automate Grid-service composition (Gil et al., 2004). However, we observe that limitations in the expressivity of planning systems

requires hand-coding characteristics of the solution in the problem specification. In particular, they do not allow the creation of new objects. Furthermore, planning systems typically are not engineered so that the correctness of the output depends only on a small trusted logical kernel. Thus the whole system must be trusted. Our proposed approach allows some extra expressiveness, and has a small fixed trusted code base thus giving an improved guarantee of the correctness of the synthesised plans.

Workflows for Proof Transformation Services. We look at an example workflow problem, presented in Zimmer et al. (2004), that integrates different proof tools. The approach they take is to give each proof system a formal description by specifying it in terms of its input and output parameters and conditions. A brokering agent then attempts to meet a service request by creating a workflow that combines the different proof tools available. It is this brokering agent which we model in our framework. Although Zimmer et al. (2004) were able to synthesise suitable workflows using a planner, to do this they had to manually introduce a number of unique dummy objects before planning. These were needed to represent the objects created by services. If this number is not chosen correctly, then no plan can be found.

In our framework, services are represented by actions of the form `A` $\multimap$ `B`, where `A` is the kind the input used and `B` is the result. Following the presentation of Zimmer et al. (2004), we consider the following services: CNF conversion, provided by the `tramp` service; first order resolution theorem proving, using `otter` and `vampire`; and the conversion of a resolution proof into a natural language one, which is done by the `prex` system. The brokering agent starts by placing the goal within a context containing these services. For example, to synthesis a service that given a conjecture, will find a natural language description of its proof, the following goal is given to our system:

$\{\} \vdash$ `?p :` `(! conj` x`)` $\multimap$ `nl_proof` x

where x is a formula and `conj` x denotes an resource that states this as a conjecture. For its part, `nl_proof` x is a resource that describes a proof of x in natural language. We make the conjecture a non-linear resource as we want to be able to use it arbitrarily many times. The available resources are:

`tramp : conj` x $\multimap$ `cnf` x
`otter : cnf` x $\multimap$ `res_proof` x
`vampire : cnf` x $\multimap$ `res_proof` x
`prex : conj` $x \otimes$ `res_proof` x $\multimap$ `nl_proof` x

This allows our planner to find both of the obvious workflows, which once pretty printed give the following instantiations for the variable `?p`:

```
(1)lam c. let! c' = c in
  prex (ivar c', otter (tramp (ivar c')))
(2)lam c. let! c' = c in
  prex (ivar c', vampire (tramp (ivar c')))
```

where we write the ILL "`app`" as an infix space, tensor as ",". Also, we use `let!` x `=` p `in` t for `let_bang p x t`, as it more closely resembles the pattern matching style of functional programming.

7. Conformant and Contingent Planning using Constraints

When planning in the presence of indeterminacy, a distinction is made between *conformant* planning, where the resultant plan is executed regardless of the indeterminate outcomes, and where, therefore, no sensing is needed; and *contingent* planning, where plan execution is made conditional on the outcomes of the various sensor output.

In our formalisation, this distinction can be made easily by attaching a condition on the extracted proof terms. The key rule is that of $\oplus$-elimination:

$$\frac{\{a : A\},\ T \vdash (c_1\ a) : C \qquad \{b : B\},\ T \vdash (c_2\ b) : C}{\{z : A \oplus B\},\ T \vdash \texttt{case\_or}\ a'\ b'.\ (\texttt{var}\ z)\ (c_1\ a')\ (c_2\ b')\ :\ C}\ \oplus E$$

where a, b, and z are fresh in T. This deals with case analysis during execution on a resource of the form `A` $\oplus$ `B`; the rule allows distinct proof terms, c_1 and c_2 above, to be used as depending on which of A or B is the case. This naturally provides contingent planning. We introduce a new meta-logical condition to require a conformant plan. This is expressed as a derived form of $\oplus$-elimination:

$$\frac{\{a : A\},\ T \vdash (c_1\ a) : C \qquad \{b : B\},\ T \vdash (c_2\ b) : C}{\{z : A \oplus B\},\ T \vdash \texttt{case\_or}\ a'\ b'.\ (\texttt{var}\ z)\ (c_1\ a')\ (c_2\ b')\ :\ C}\ \oplus E2$$

Using this derived rule instead of the usual $\oplus$-elimination and checking that the constraint subgoals `conformant c1 c2` is proved ensures that the synthesised plans are conformant. This illustrates the ability to attached constraints beyond those arising from higher-order unification.

For its part, the constant `conformant` is an recursively defined predicate over terms and can be varied depending on the desired notion of conformant. A selective use of $\oplus$-elimination with the conformant version allows us to mix the two approaches if required, for example if sensors are available for some contingencies and not for others.

Example: The Socks Problem

We now illustrate conformant and contingent planning with a simple example. The problem is to get a pair of socks from the back of a chest. Because of the location of the socks, their colour cannot be seen until they are taken. The two-colour version of this problem is when there are only black and white socks.

We formalise this problem by having a sequent where the available linear resources are the socks at the back of the chest. We have a single action which is that of picking a hidden sock the effect of which is to remove a hidden sock and add either a black sock or a white one. The conclusion of the goal sequent is the desired state, namely to have either two black socks or two white socks. For instance, the problem with three hidden socks is formalised as the following goal sequent:

`h1 : hidden, h2 : hidden, h3 : hidden`
$\vdash$ `?p : (black` $\otimes$ `black` $\otimes$ `top)`
$\oplus$ `(white` $\otimes$ `white` $\otimes$ `top)`

where we use top to allow solutions containing more socks than are needed. The action of picking a sock is specified as:

```
pick : hidden ⊸ (black ⊕ white)
```

When we use contingent planning, we get many different possible instantiation for ?p. For instance, the normalised version of a plan that looks at each sock after picking it is:

```
case_or (pick h1)
(λb1. case_or (pick h2) (λb2. b1⊗b2⊗h3)
 (λw2. case_or (pick h3)
   (λb3. b1⊗b3⊗w2) (λw3. w2⊗w3⊗b1)))
(λw1. case_or (pick h2)
 (λb2. case_or (pick h3)
   (λb3. b2⊗b3⊗w1) (λw3. w1⊗w3⊗b2))
 (λw2. w1⊗w2⊗h3))
```

If we only allow conformant planning, we find a strict subset of the contingent plans. For the above problem, the following plan is found:

```
case_or (pick h1)
(λb1. case_or (pick h2)
  (λb2. case_or (pick h3)
     λb3. inl b1⊗b2⊗b3)
       λw3. inl b1⊗b2⊗w3))
  (λw2. case_or (pick h3)
    (λb3. inl b1⊗b3⊗w2)
      (λw3. inr w2⊗w3⊗b1)))
(λw1. case_or (pick h2)
  (λb2. case_or (pick h3)
    (λb3. inl b2⊗b3⊗w1)
      (λw3. inr w1⊗w3⊗b2))
  (λw2. case_or (pick h3)
    (λb3. inr w1⊗w2⊗b3)
      (λw3. inr w1⊗w2⊗w3)))
```

This plan picks three socks and then checks the possible outcomes. Because in each case there will either be two black socks or two white ones, this plan solves the specification. Contingent planning would also find the plans where each sock is examined after it is picked.

8. Attaching Constraints to Synthesis

Nareyek et al. have argued for a closer integration between planning and constraint satisfaction (Nareyek et al., 2005). Our formalism provide an approach to this integration by allowing constraints to be attached to planning as extra subgoals. These extra constraint goals can then be checked during planning to prune the search, or at the end of planning to remove certain results and force backtracking.

Example: Scheduling a Fried Breakfast

We show the integration of constraints with an example of a scheduling problem for cooking a fried breakfast. We consider the problem of having two frying pans and wanting to cook eggs, bacon, tomatoes, and mushrooms, within seven minutes. We represent a frying pan as being free at a time x with pan(x). Cooking each item of the breakfast then takes a pan at some point in time, the ingredient being cooked and gives back the cooked ingredient and the pan noted as free at a later time:

```
cook_egg : pan(x) ⊗ egg
 ⊸ pan(x+1) ⊗ c_egg
cook_bacon : pan(x) ⊗ bacon
 ⊸ pan(x+3) ⊗ c_bacon
cook_tomatoes : pan(x) ⊗ tomato
 ⊸ pan(x+4) ⊗ c_tomatoes
cook_mushrooms : pan(x) ⊗ mushrooms
 ⊸ pan(x+3) ⊗ c_mushrooms
```

We use c_X to represent that X is cooked. Because our formalism does not contain a notion of universal quantification within ILL, we represent these actions as constants and assume that they are derivable without any context. For example, the assumption that we can cook eggs is:

```
∀x. {} ⊢ cook_egg : pan(x) ⊗ egg
 ⊸ pan(x+1) ⊗ c_egg
```

With an ILL universal quantifier the above can also be represented in the non-linear context.

We then use our planner to tackle the goal which is represented as:

```
{egg, bacon, tomatoes, mushroom,
 pan(0), pan(0)}
⊢ ?p : c_egg ⊗ c_bacon ⊗ c_tomatoes
 ⊗ c_mushrooms ⊗ pan(?y) ⊗ pan(?z)
 ∧ ?y < 7 ∧ ?z < 7
```

When ?y and ?z are instantiated, we check the two additional goals to ensure that the final plan meets the added constraint. To constrain the search during synthesis we can define a HOL predicate that examines the goal sequent and ensures that in every occurrence of pan(x), the constraint x < 7 is true. We use Isabelle simplifier to check if the constraint holds.

Another approach we have examined is to constrain the search by adding timing information into the proof term and then constrain ?p. This requires fully instantiating ?p with a dummy constant in order that the HOL predicate can then be computed.

9. Further Work

The work presented in this paper can be extended by formalising the execution of proof terms, which can be related to normalisation of plans. Another interesting avenue of further work would be to extend our formalisation of ILL by adding quantifiers and iteration, following the work of Cresswell (2001).

Our planner could be applied to various applications and could be combined with a system for executing workflows. One suitable candidate is Zimmer's mathematical services system. Another application is in the parsing of natural language, following the approach proposed by (Steedman, 2002).

We intent to further explore the automation of planning problems which allow entities to be created and destroyed by actions. This can be easily expressed in ILL and takes us outside the formalism of STRIPS with non-deterministic actions; §8. gives such an example.

Another area of future work is to improve the efficiency of automation. Although our framework is more expressive,

the speed of proof search in Isabelle results in planning being many orders of magnitude slower than other planners. Improvements to efficiency can be made by including further heuristic information in the synthesis of plans, or by using existing planners as an 'oracle' and verifying the result. The 'oracle' approach (Harrison and Théry, 1998) would use an efficient planner to search for the plans and simply check the plans are valid using our proof machinery. For proof search within Isabelle, we also intend to implement further symmetry removing techniques along the lines of Andreoli (1992).

There has been significant work on interfaces for interactive proof assistants, with various approaches to managing user interaction, such as Aspinall and Kleymann (2004); Dixon (2005). Thus, a natural avenue of further work is to consider how such interfaces could be used for interaction with a planner, and more generally in the field of mixed initiative planning (Burstein and McDermott, 1996).

10. Conclusions

We have formalised ILL as an embedding in Isabelle/HOL where both terms and types are HOL datatypes and derivability in ILL is defined as membership of an inductively defined set.

We interpret the ILL proof terms as plans and provide tactics to perform basic planning steps within our formalisation. This extends other planning formalisms by allowing the introduction of new objects as well as their removal, supporting non-deterministic resources, and allowing conditions to be attached to planning. Unlike previous work using linear logic for planning, we use the proof terms for the non-deterministic resources to support synthesis of both contingent and conformant plans. Moreover, our synthesis framework separates the proof search from the logical representation which allows it to employ the LCF methodology for extending automation while preserving soundness. The synthesised plans have been verified by a small logical kernel of trusted code and they can easily be checked by a small independent type-checking program.

Tactics for forward and backward reasoning have been defined and combined to provide fully automatic planners. These have been applied to the synthesis of workflows for combining theorem proving systems. We have also shown how integrating constraints on the derived plans can be done using the existing theories of Isabelle/HOL. We also apply these techniques to solve the socks problem illustrating how plans with disjunctions can be handled in both a contingent and conformant manner. This has also been used to solve a scheduling problem where the attached constraints were used to reduce the size of the search space during planning.

Although our framework is very expressive, planning is slow. This is because we are working in an interpreted environment and, although we avoid context splitting, our proof search algorithm is otherwise naive. Approaches to improve this include the use of existing planners as oracles (Harrison and Théry, 1998), so that verifiation of the plan would simply be type-checking, and the development of more efficient proof search for ILL.

We have thus provided a platform for the exploration of the relationship between ILL specifications, proof terms, planning problems and planning algorithms implemented as proof search.

References

Abramsky, S. 1993. Computational interpretations of linear logic. *Theor. Comput. Sci.* 111(1–2):3–57.

Andreoli, J. M. 1992. Logic programming with focusing proofs in linear logic. *J. Log. Comp.* 2(3):297–347.

Aspinall, D., and Kleymann, T. 2004. *Proof General Manual*. University of Edinburgh, proofgeneral-3.5 edition.

Barber, A. 1997. *Linear Type Theories, Semantics and Action Calculi*. Ph.D. Dissertation, University of Edinburgh.

Berghofer, S., and Nipkow, T. 2000. Proof terms for simply typed higher order logic. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, 38–52. London, UK: Springer-Verlag.

Bundy, A.; Smaill, A.; and Yang, B. 2003. Formalising the grid - the 1st step to automate grid application assembly using deductive synthesis. In *Proceedings of UK e-Science Second All Hands Meeting*, 337–341.

Burstein, M. H., and McDermott, D. V. 1996. Issues in the development of human-computer mixed-initiative planning systems. *Cognitive Technology: In Search of a Human Interface* 20.

Cervesato, I., and Pfenning, F. 2002. A linear logical framework. *Information & Computation* 179(1):19–75.

Cervesato, I.; Hodas, J. S.; and Pfenning, F. 2000. Efficient resource management for linear logic proof search. *Theor. Comput. Sci.* 232(1-2):133–163.

Cresswell, S. 2001. *Deductive Synthesis of Recursive Plans in Linear Logic*. Ph.D. Dissertation, University of Edinburgh.

de Groote, P. 1995. Linear logic with Isabelle: pruning the proof search tree. In *CADE'95*. Springer-Verlag LNCS 814.

Dixon, L.; Smaill, A.; and Tsang, T. 2009. Plans, actions and dialogue using linear logic. *Journal of Logic, Language and Information* 18(2):48.

Dixon, L. 2005. Interactive and hierarchical tracing of techniques in IsaPlanner. *ENTCS: User Interfaces For Theorem Provers* 13.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Some new directions in robot problem solving. In *Machine Intelligence 7*. Edinburgh University Press. 405–430.

Gil, Y.; Deelman, E.; Blythe, J.; Kesselman, C.; and Tangmunarunkit, H. 2004. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems* 19(1):26–33.

Girard, J.-Y. 1987. Linear logic. *Theor. Comput. Sci.* 50:1–102.

Gordon, M. J.; Milner, A. J.; and Wadsworth, C. P. 1979. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *LNCS*. Springer-Verlag.

Große, G.; Hölldobler, S.; and Schneeberger, J. 1996. Linear deductive planning. *J. Log. Comput.* 6(2):233–262.

Harland, J., and Pym, D. J. 2003. Resource-distribution via boolean constraints. *ACM Trans. Comput. Log* 4(1):56–90.

Harrison, J., and Théry, L. 1998. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning* 21:279–294.

Hodas, J. S., and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation* 110(2):327–365.

Ishtiaq, S. S., and Pym, D. J. 1998. A relevant analysis of natural deduction. *J. Log. Comput.* 8(6):809–838.

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

Kalvala, S., and de Paiva, V. 1995. Mechanizing linear logic in Isabelle. In *10th International Congress of Logic, Philosophy and Methodology of Science*.

Kanovich, M. I., and Vauzeilles, J. 2001. The classical AI planning problems in the mirror of horn linear logic: semantics, expressibility, complexity. *Mathematical Structures in Computer Science* 11(6):689–716.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *ECAI'92*.

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

Levesque, H.; Pirri, F.; and Reiter, R. 1998. Foundations for a calculus of situations. *Linkoping Electronic Articles in Computer and Information Science* 3(18).

Lincoln, P.; Mitchell, J. C.; Scedrov, A.; and Shankar, N. 1992. Decision problems for propositional linear logic. *Ann. Pure Appl. Logic* 56(1-3):239–311.

López, P., and Polakow, J. 2004. Implementing efficient resource management for linear logic programming. In *LPAR*, volume 3452 of *LNCS*, 528–543. Springer.

Masseron, M. 1993. Generating plans in linear logic II: A geometry of conjunctive actions. *Theor. Comput. Sci.* 113:371–375.

Nareyek, A.; Freuder, E. C.; Fourer, R.; Giunchiglia, E.; Goldman, R. P.; Kautz, H.; Rintanen, J.; and Tate, A. 2005. Constraints and ai planning. *IEEE Intelligent Systems* 20(2):62–72.

Paulson, L. C. 1994. Isabelle: A Generic Theorem Prover. *LNCS* 828.

Steedman, M. 2002. Plans, affordances, and combinatory grammar. *Linguistics and Philosophy* 25(5-6):723–753.

Urban, C., and Tasson, C. 2005. Nominal techniques in Isabelle/HOL. In *CADE*, volume 3632 of *LNCS*, 38–53. Springer.

Zimmer, J.; Meier, A.; Sutcliffe, G.; and Zhang, Y. 2004. Integrated proof transformation services. Technical report, RISC.