Verified Planning by Deductive Synthesis in Intuitionistic Linear Logic

Planning as Deductive Synthesis in Intuitionistic Linear Logic

Lucas Dixon, Alan Smaill, and Alan Bundy

{L.Dixon, A.Smaill, A.Bundy}@ed.ac.uk
University of Edinburgh, Informatics, UK

Abstract. We describe a new formalisation in Isabelle/HOL of Intuitionistic Linear Logic and consider the support this provides for constructing plans using deductive synthesis of the proof terms. 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 the proof assistant. 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.

1 Introduction

Linear Logic was introduced in [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 can aid proof search by helping guide the synthesis process. 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 must go through the logical kernel of the proof system. However, because we have implemented the full logic, where deducibility is undecidable [Lincoln *et al.*, 1992], it allows us to work in a much richer language. This motivates our use of a flexible environment for developing incremental improvements and extensions to the automation. For example, heuristics mixing forwards and backwards planning can also be built into search without worrying about compromising the soundness of the results.

We illustrate the approach with one example from within a decidable fragment, for which we have developed an automatic planner, and one which incorporates constraints on the desired plan that make it conformant.

Overview. In §2 we describe related work, both in implementation and in relation to planning. We then introduce the Isabelle implementation (§3) and associated reasoning techniques (§4). Search strategies for automated planning are discussed in §5. §6 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, for example [Kalvala and de Paiva, 1995]. Our work extends these by providing a representation of proof terms, needed for the synthesis of plans, and improves on the automation by supporting lazy context splitting as described in §4.

A Prolog implementation of ILL with proof terms was used to synthesise plans in [Cresswell, 2001]. This includes forms of induction with associated proof terms that extend the ILL framework. It also provided a good level of automation but lacks higher-order features and the various libraries present in Isabelle, which simplify the presentation and aid further automation.

There are several logic programming languages that automate reasoning where theories and queries 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 §6 falls outside this subset. However, it does allow simple meta-interpreters to be written for the full language.

Framework logics that support linear connectives, such as [Ishtiaq and Pym, 1998; Cervesato and Pfenning, 2002], provide a suitable form of executable proof term and could be thus used to embed a representation of plans. Implementation of these provides 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 encoded in the framework, or issues such as contingent versus conformant plans.

Relationship to Planning

AI planning systems based on the STRIPS approach work with a procedural manipulation of statements that are taken to correspond to a description of the state of the system being modelled. To make the reasoning involved explicit, a standard approach is to make use of a version of the situation calculus, where notions of *state* or *situation* appear explicitly in the object language; Levesque *et al.* [1998] provides 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 itself. 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 §6.1), and distributed execution, without any additional machinery. It also allows us to reason easily about the dynamic introduction and elimination of entities associated with actions.

The situation calculus uses notions of relational and functional *fluents* for properties that may change from state to state, but does not directly support a notion of entities that are dynamically created. It is possible to some extent to encode this idea 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.

A comparison of Linear Logic with other formalisms for planning has been presented by [Große *et al.*, 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 aim to further this approach by the use of proof terms with a well-defined operational semantics.

3 Intuitionistic Linear Logic in Isabelle/HOL

We have formalised the dual context account of ILL following the work of [Barber, 1997]. In this paper, we view ILL as characterising how resources are used in a plan. 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.

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 new theorems is a higher-order version of resolution. Additional proof tools can be written in ML, following the LCF methodology [Gordon *et al.*, 1979]. This requires that all functions that derive a new theorem 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 particular to Isabelle with which we will present our formalisation.

ILL Sequents

A sequent of our embedded linear logic is characterised by the `turnstyle` constant which has type `ires multiset` ⇒ `lres multiset` ⇒ `res` ⇒ `bool` where

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: $[\![\ assumption_1\texttt{;}\ \ldots\texttt{;}\ assumption_n\ ]\!] \Longrightarrow conclusion$

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

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

Fig. 1. The Isabelle syntax used in this paper.

`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.

ILL Types

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
```

An ILL type of the form `A ⊸ B` expresses a resource which will use up `A` and produce `B`. `A ⊕ 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 ⊗ B` we get both `A` and `B`. Lastly, `!A` lets us get arbitrarily many `A`'s.

ILL Terms

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 ⊢ (app (lam y (b y)) 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].

4 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 ⇒ foodstuff ⇒ ltyp
⋀ p x. {} ⊢ eats p x : x ⊗ hungry p ⊸ full p
```

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

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 *et al.*, 1972] fragment of ILL then it acts in a similar way to a traditional planner.

4.1 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.

Backward reasoning about actions can be handled easily by the following derived rule: $[\![$ `(r : A ⊸ B) ∈ D ; D | T ⊢ A` $]\!] \Longrightarrow$ `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:

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

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 to deal with this issue in [Harland and Pym, 2003; Cresswell, 2001; Cervesato *et al.*, 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 on the goal "`{B, C}` $\vdash$ `X` $\otimes$ `(A` $\multimap$ `Y)`". This would result in the following subgoals: "`?S, ?T` $\vdash$ `X` $\otimes$ `(A` $\multimap$ `Y)` $\Longrightarrow$ `{B, C}` $\vdash$ `X` $\otimes$ `(A` $\multimap$ `Y)`"$^{(1)}$, "`?S` $\vdash$ `X`"$^{(2)}$, and "`?T` $\vdash$ `A` $\multimap$ `Y`"$^{(3)}$. Subgoal (3) can then be further decomposed using the $\multimap$-introduction rule to get the subgoal "`{A}, ?T` $\vdash$ `Y`"$^{(4)}$. To illustrate forward reasoning, we consider having an action `r : (A` $\otimes$ `B)` $\multimap$ `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 "`{E}, ?T'` $\vdash$ `Y`".

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.

4.2 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.

5 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, it will find the smallest plan. 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.

5.1 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 *et al.*, 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 $\texttt{A} \multimap \texttt{B}$, 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 ⊸ cnf x
otter : cnf x ⊸ res_proof x
vampire : cnf x ⊸ res_proof x
prex : conj x ⊗ res_proof x ⊸ 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 ",". 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.

6 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:

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

Using this derived rule instead of the usual $\oplus$-elimination and checking that the constraint subgoals `conformant` c_1 c_2 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.

6.1 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
⊢ ?p : (black ⊗ black ⊗ top) ⊕ (white ⊗ white ⊗ 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)
```

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.

7 Further Work

The work presented in this paper can be extended in several ways, e.g. by presenting the formalisation of execution of proof terms within Isabelle; this can be related to normalisation of plans. Another interesting avenue of further work is to add to our formalisation of ILL. For example, by including a notion of iteration to the formalisation, following the work of [Cresswell, 2001], or by adding quantifiers to the formalisation of ILL.

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].

Another area of future work is to improve automation by including heuristic information in the synthesis of plans. 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].

8 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.

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.

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.

Bibliography

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

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

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

A. Barber. *Linear Type Theories, Semantics and Action Calculi*. PhD thesis, University of Edinburgh, 1997.

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

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

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

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

S. Cresswell. *Deductive Synthesis of Recursive Plans in Linear Logic*. PhD thesis, University of Edinburgh, 2001.

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

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

Yolanda Gil, Ewa Deelman, Jim Blythe, Carl Kesselman, and Hongsuda Tangmunarunkit. Artificial intelligence and grids: Workflow planning and beyond. *IEEE Intelligent Systems*, 19(1):26–33, 2004.

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

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

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

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

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

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

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*, pages 62–66, 1993.

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

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

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

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

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

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

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

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

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

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

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

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