The Use of Proof Plans for Normalization

Alan Bundy

DAI Research Paper No.

July 7, 1992

Submitted to the Festschrift in Honour of Woody Bledsoe

Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland

The Use of Proof Plans for Normalization *

Alan Bundy

Abstract

We propose using proof plans to implement expression normalizers in automatic theorem proving. We outline some general-purpose proof plans and show how these can be combined in various ways to yield some standard normalizers. We claim that using proof plans facilitates the flexible application of these normalizers so that they can interact with the theorem prover in which they are embedded. We intend to extend this technique to decision procedures.

1 Introduction

In [Boyer & Moore 88], Boyer and Moore investigate a case study in the use of decision procedures in automated deduction, namely a decision procedure for linear arithmetic. Their conclusions were as follows.

- Such decision procedures have a vital role to play in reducing the combinatorial explosion, but
- they cannot be treated as black boxes. In practice, few sub-goals exactly fit the requirements of a decision procedure, but many almost do. In these cases it is necessary to augment the decision procedure with additional lemmas. So the operations of the decision procedure must be interleaved with those of the theorem prover, *i.e.* as well as the theorem prover calling the decision procedure, the decision procedure must be able to call the theorem prover.
- Organising this interaction between decision procedure and theorem prover dominates both the implementation time and the run time of the decision procedure. So the efficiency of the decision procedure itself is of secondary importance.

Boyer and Moore's original implementation took a decision procedure of Hodes, [Hodes 71], and allowed their theorem prover to call it when a sub-goal fell within its scope. Unfortunately, in a run of their standard corpus of several hundred theorems only a handful of sub-goals were of suitable form and the decision procedure had no significant impact on the performance of the theorem prover.

They then re-implemented Hodes' procedure so that it could interact with the theorem prover. For instance, their decision procedure applies a series of linear rules to its input expression. It can call on an expression rewriter to establish the hypotheses of these linear rules. This more flexible implementation was successfully used many times during a run on their standard corpus. It reduced the overall run-time of the theorem prover by 40%. Of this overall run-time, less than 3% was spent in the decision procedure itself, but 22% was spent in interaction between the decision procedure and the theorem prover. Thus the speed of the decision procedure was of

*The research reported in this paper was supported by SERC grant GR/E/44598 and an SERC Senior Fellowship to the author. I am grateful for feedback on this paper from David Basin, Frank van Harmelen and Toby Walsh, and for many conversations with other members of the mathematical reasoning group at Edinburgh.

relatively low significance. Far more important was having an implementation that simplified the process of interaction. The changes required to both Hodes' procedure and the theorem prover, in order to integrate them, were extensive and complex, and time-consuming to implement. It would not be a simple matter to modify this decision procedure or to swap it for another one.

In this paper we initiate an investigation into the use of *proof plans*, [Bundy 88], for implementing decision procedures that might help solve these problems. The hypothesis we are trying to test is that proof plans will provide a modular representation of decision procedures that will facilitate their synthesis and modification and their interaction with the theorem provers in which they are embedded. The cost may be some inefficiency in the implementation of the decision procedures, but the Boyer-Moore experience seems to suggest that this cost is not significant within the context of the complete system.

We focus on the use of proof plans for the implementation of *normalizers*, *i.e.* procedures for putting expressions into normal form. A normalizer takes an expression belonging to some syntactic class and finds an equivalent (in some sense) expression belonging to a strict sub-class. This equivalent expression is the *normal form* of the original expression.

Normalization plays an important role in theorem proving; many proofs include steps in which expressions are put into normal form. In particular, normalization plays an important role in many decision procedures. For instance, some decision procedures consist of the application of a sequence of normalization steps which reduce the original expression to something which is evaluable. So implementing normalizations would solve an important part of the problem of implementing decision procedures. At the end of the paper we will discuss what more is required to extend the ideas to a complete decision procedure, such as the Hodes one. We will restrict the discussion to the implementation of normalizers by the exhaustive or selective application of sets of rewrite rules. A large subset of normalizers can be implemented in this way, and it will simplify the application of proof plans to make this restriction. The cost will be in the efficiency of the implementation, but as discussed above, this cost is less important than is normally assumed in this context. It may be possible to lift the restriction at some future time.

This paper is organised as follows. We begin with some background information on proof plans. We then informally describe some simple proof plans for common normalization steps. We show how these proof plans can be combined to implement some standard normalizers and we discuss the meta-logic required to define them formally. We discuss how these normalization proof plans could be used to implement part of Hodes' decision procedure for linear arithmetic and the equality and inequality solving proof plans that are required to implement the rest of it. Finally, we illustrate the sort of flexible application of normalizers that the proof plans approach facilitates.

2 Proof Plans

A proof plan is a representation of the structure of all or part of a mathematical proof. In [Bundy *et al* 91] we describe the use of proof plans to control the search for proofs of mathematical conjectures. The key idea is that many proofs share common structure, either in whole or in part. We analyse the structure of a family of proofs and identify any common structure. We represent these common structures in general-purpose proof plans, and then use them as blueprints to guide the search for a proof in the same family. A special-purpose proof plan is developed for this new proof by putting together general-purpose proof plans, using techniques of plan formation.

A proof plan consists of two parts: a *tactic* and a *method*. A tactic is a computer program which constructs a proof by applying rules of inference to a conjecture. A method is a partial specification of a tactic. It consists of preconditions for the application of its tactic and a description of the effects of the successful application of the tactic. Both preconditions and effects are written in a meta-logic, whose domain of discourse includes expressions of the object-logic, whose functions manipulate these expressions and whose predicates describe syntactic properties of, or relations between, the expressions. AI plan formation techniques can be used to combine smaller proof plans into larger ones. Proof plans are linked together by inferring the preconditions of later ones from the effects of earlier ones. Plan formation can be used in two modes:

on-line To prove a particular conjecture the planner can put together some general-purpose proof plans into a special-purpose proof plan, customised for this conjecture. Our planner, CLAM, does this, [Bundy *et al* 91].

off-line To synthesise a new general-purpose proof plan without regard to any particular conjecture. Desimone has built a planner of this form, [Desimone 89].

In this paper we will outline some general-purpose proof plans which we claim are of general utility in putting expressions into normal form. By instantiating these proof plans with different sets of rewrite rules and by combining them in different ways we can represent a wide variety of different normalizers. Thus a proof plans implementation of normalizers provides a degree of generality and modularity that is not available from standard implementations.

To represent the preconditions and effects of these proof plans we will make extensive use of Backus-Naur Form (henceforth, BNF). We will write BNFs using the following notation:

$$Cls \quad := \quad Base_1|\ldots|Base_k|F_1(Cls^1_1,\ldots,Cls^1_{m_1})|\ldots|F_n(Cls^n_1,\ldots,Cls^n_{m_n})$$

which means that expressions of class Cls are either of class $Base_i$ for some $1 \leq i \leq k$ or of the form $F_i(E^i_1,\ldots,E^i_{m_i})$, for some $1 \leq i \leq n$. where E^i_j is of class Cls^i_j for all $1 \leq j \leq m_i$. Note that F_i need not be a function symbol, but might be a compound expression. In practice, the classes Cls^i_j are usually Cls, but in principle can be different classes.

The effect of most of the proof plans will be to take an expression belonging to one BNF and transform it to an equivalent (in some sense) expression belonging to another one. For instance, the procedure for putting propositional expressions into conjunctive normal form takes expressions of class, fm, defined by:

$$fm \quad := \quad prop|\neg fm|fm \vee fm|fm \wedge fm|fm \rightarrow fm|fm \leftrightarrow fm$$

and puts them into the class, fm', defined by:

$$\begin{aligned} fm' &\quad := \quad clause|fm' \wedge fm' \\ clause &\quad := \quad literal|clause \vee clause \\ literal &\quad := \quad prop|\neg prop \end{aligned}$$

Thus our meta-logic will be mostly concerned with defining such syntactic classes, manipulating them and describing their relationships with each other and with syntactic expressions.

We will discuss both the on-line and off-line synthesis of such proof plans. In off-line synthesis a new general-purpose normalizer will be built for a class of expressions. In on-line synthesis a new normalizer will be custom-built for a particular expression. This normalizer will be able to use any available definitions or lemmas about the symbols in the expression to be normalized. This will constitute our proposed solution to the problem of integrating decision procedures and theorem provers. Thus a proof plans implementation of normalizers facilitates their flexible integration with their host theorem prover to an extent not available from standard implementations.

3 Removing 'Defined' Functions

We will start by considering a step that occurs in many normalizers and is simple to represent as a proof plan: the removal of a particular function[1] F, say, by rewriting all occurrences of it with a rewrite rule of the form:

$$F(X_1, \ldots, X_m) \quad \Rightarrow \quad T(X_1, \ldots, X_m) \tag{1}$$

where the X_i are variables and $T(X_1, \ldots, X_m)$ is a term not containing F. Such rewrite rules might be based on either explicit definitions or lemmas of similar form, and applied left to right or right to left, as appropriate. Examples of such rewrite rules are:

$$\begin{aligned}
p \leftrightarrow q &\quad \Rightarrow \quad p \rightarrow q \wedge q \rightarrow p \\
p \rightarrow q &\quad \Rightarrow \quad \neg p \vee q \\
x \geq y &\quad \Rightarrow \quad y \leq x \\
x \leq y &\quad \Rightarrow \quad \neg y < x \\
\forall x.\, p &\quad \Rightarrow \quad \neg \exists x.\, \neg p
\end{aligned}$$

We will call this proof plan, *remove*. It will take *one* such rule and apply it exhaustively to the current expression until no occurrences of the function F remain. By repeatedly applying *remove*s a set of functions can be removed. Care must be taken not to reintroduce an earlier function during the removal of a later one. This can be done by partially ordering the functions in such a way that the right hand side, T, of the remove rewrite rule, (1), for function F, contains only functions which are lower in the order than F.

Preventing such loops is the job of the planner using the method of the proof plan. This means that the preconditions and effects of *remove* should note both the function removed and those in terms of which it is removed. This will be done implicitly by the BNFs describing the input and output of the tactic. Recall that we define such BNFs using the notation:

$$Cls \quad := \quad Base_1 | \ldots | Base_k | F_1(Cls^1_1, \ldots, Cls^1_{m_1}) | \ldots | F_n(Cls^n_1, \ldots, Cls^n_{m_n})$$

So to remove a function, F, from an expression, Exp, the *remove* tactic will take two inputs: Exp and the rewrite rule used to remove F, and output an expression, Exp', free of F. The input expression, Exp, will belong to some syntactic class, Cls, containing F, *i.e.*:

$$Cls \quad := \quad Base_1 | \ldots | Base_k | F(Cls^F_1, \ldots, Cls^F_m) | \ldots G(Cls^G_1, \ldots Cls^G_n) | \ldots$$

where the F disjunct has been placed first for notational convenience, but without loss of generality, and the G disjunct represents one of the other non-base disjuncts. The output expression, Exp', will belong to the syntactic class, Cls', which is Cls with F deleted, *i.e.*

$$Cls' \quad := \quad Base_1 | \ldots | Base_k | \ldots G(Cls^G_1 \phi, \ldots Cls^G_n \phi) | \ldots$$

where $\phi = \{Cls'/Cls\}$.

The remove rewrite rule for F must have the form:

$$F(X_1, \ldots, X_m) \quad \Rightarrow \quad T(X_1, \ldots, X_m)$$

where if X_i is of syntactic class $Cls_i \phi$ for all $1 \leq j \leq n$ then $T(X_1, \ldots, X_m)$ is of syntactic class Cls'.

[1]We use the word 'function' here in the lambda calculus sense to include predicates, connectives, quantifiers, *etc.*

Note that the above description does not restrict F to be a function symbol. It could be a compound expression.

Hodes' decision procedure for linear arithmetic contains an example of a more complex form of removal. The rewrite rules:

$$\neg x = y \quad \Rightarrow \quad x < y \vee y < x \tag{2}$$
$$\neg x < y \quad \Rightarrow \quad x = y \vee y < x \tag{3}$$

are used to transform expressions of the form fm, defined by the BNF:

$$\begin{aligned} fm &:= literal|fm \vee fm|fm \wedge fm \qquad (4) \\ literal &:= prop|\neg prop \\ prop &:= tm = tm|tm < tm \end{aligned}$$

into the expressions of the form fm', defined by the BNF:

$$fm' \quad := \quad prop|fm' \vee fm'|fm' \wedge fm' \tag{5}$$
$$prop \quad := \quad tm = tm|tm < tm \tag{6}$$

This transformation can be viewed in various ways. The following account establishs a clear connection to *remove*. The input BNF, (4), is first rewritten to the equivalent BNF:

$$fm \quad := \quad tm = tm|tm < tm|\neg tm = tm|\neg tm < tm|fm \vee fm|fm \wedge fm$$

Rewrite rule (2) is then used to *remove* $\neg tm = tm$ and rewrite rule (3) is used to *remove* $\neg tm < tm$. The new BNF is:

$$fm' \quad := \quad tm = tm|tm < tm|fm' \vee fm'|fm' \wedge fm'$$

which is equivalent to (5). Here we take advantage of the possibility of F being a compound expression.

What might motivate this transformation of the BNF for fm which enables *remove* to work? There seem to be at least three possibilities.

1. We could incorporate transformations of the BNFs into the search space at the meta-level. Unfortunately, this would significantly increase the amount of search.

2. We could put all BNFs into normal form. Unfortunately, It is not clear that we could devise a normal form that would satisfy the preconditions of all the various versions of *remove* that we want.

3. We could devise more powerful preconditions for *remove* that would be transparent to the various equivalent forms in which a BNF might appear. Unfortunately, these preconditions would be hard to write and might only internalise the additional search involved in possibility 1 above.

4 Stratification of Classes

Another simple and common proof plan is to stratify a class into a number of layers, with each layer containing only one function. For instance, during conjunctive normal form the class of formulae, fm, defined by:

$$fm \quad := \quad prop|\neg fm|fm \vee fm|fm \wedge fm$$

is stratified into the three layers:

$$\begin{aligned} fm' &:= clause | fm' \wedge fm' \\ clause &:= literal | clause \vee clause \\ literal &:= prop | \neg literal \end{aligned}$$

by moving all $\neg$s beneath all $\vee$s and $\wedge$s and all $\vee$s beneath all $\wedge$s. The fm' layer contains only $\wedge$, the *clause* layer contains only $\vee$, and the *literal* layer contains only $\neg$. This requires two different stratification operations.

Stratification requires a set of rewrite rules. Its object is to move one function, F, beneath each of the other functions in the class. So if the 'before' BNF class is of the form:

$$Cls \quad := \quad Base_1 | \ldots | Base_k | F(Cls_1^F, \ldots, Cls_m^F) | \ldots G(Cls_1^G, \ldots Cls_n^G) | \ldots$$

then it will be broken into two 'after' classes of the form:

$$\begin{aligned} Cls' &:= NewBase | \ldots G(Cls_1^G \phi, \ldots, Cls_n^G \phi) | \ldots \\ NewBase &:= Base_1 | \ldots | Base_k | F(Cls_1^F \psi, \ldots, Cls_m^F \psi) \end{aligned}$$

where $\phi = \{Cls'/Cls\}$ and $\psi = \{NewBase/Cls\}$.

One distributive law is required for each argument of F, for each of the other functions, G, in the definition of Cls, *i.e.* for each function, G, and for the i^{th} argument position of F, a distributive law is required of the form:

$$F(X_1, \ldots, G(Y_1, \ldots, Y_n), \ldots, X_m) \quad \Rightarrow \quad T(X_1, \ldots, Y_1, \ldots, Y_n, \ldots, X_m)$$

where $T(X_1, \ldots, Y_1, \ldots, Y_n, \ldots, X_m)$ is in the syntactic class Cls'. Examples of such distributive law sets are the following ones for moving $\neg$ beneath $\vee$ and $\wedge$,

$$\begin{aligned} \neg(p \vee q) &\Rightarrow \neg p \wedge \neg q \\ \neg(p \wedge q) &\Rightarrow \neg p \vee \neg q \end{aligned}$$

$\vee$ beneath $\wedge$,

$$\begin{aligned} p \vee (q \wedge r) &\Rightarrow (p \vee q) \wedge (p \vee r) \\ (q \wedge r) \vee p &\Rightarrow (q \vee p) \wedge (r \vee p) \end{aligned}$$

and $\exists$ beneath $\vee$.

$$\exists x.\, (p \vee q) \quad \Rightarrow \quad \exists x.\, p \vee \exists x.\, q$$

Using these ideas we can design a tactic, *stratify*, which takes an expression and a complete set of distributive laws, applies the rules exhaustively to the expression and returns a stratified expression.

Note that it is usually convenient to precede *stratify* with *remove* since this reduces the number of functions in the syntactic class and, hence, the number of distributive laws required in the *stratify* rule set.

5 Reorganising within a Class

Repeated stratification will often convert the BNF into a number of layers with only one function in each layer. We might then want to reorganise the arguments and applications of this function within the layer. The standard way to do this is by selective applications of laws like commutativity and associativity. There are a variety of ways of doing this according to the arity of the function and which laws are true of it. Below we consider some of the most common cases.

5.1 Left(Right) Association

Suppose a layer contains only one function, F, which is both binary and associative. This layer can be put into left or right associative form. Without loss of generality we consider a proof plan for left associative form, *left_assoc*. This proof plan will use the associative law as a rewrite rule in the form:

$$F(X, F(Y, Z)) \Rightarrow F(F(X, Y), Z)$$

Exhaustive application of which will have the effect of converting a BNF class:

$$Cls \quad := \quad Base_1 | \ldots | Base_k | F(Cls, Cls)$$

into the class:

$$Cls' \quad := \quad Base_1 | \ldots | Base_k | F(Cls', Base_1) | \ldots | F(Cls', Base_k)$$

5.2 Reordering

If F is also commutative then we can reorder its arguments. Suppose $\prec$ is some order on the members of $Base_1 \cup \ldots \cup Base_k$. We can use the rewrite rules:

$$\begin{aligned} F(X, Y) &\Rightarrow F(Y, X) \\ F(X, F(Y, Z)) &\Rightarrow F(F(X, Y), Z) \\ F(F(X, Y), Z) &\Rightarrow F(X, F(Y, Z)) \end{aligned}$$

selectively, to ensure that Exp is both left associated and for all base elements S and T that $S \prec T$ whenever S occurs before T in Exp.

Note that we cannot apply these three rules exhaustively, since the left/right and right/left associativity rules will cause a loop. Finding the right selective ordering is tricky. Such reordering is not usually implemented by the application of rewrite rules. It is usually done by some meta-function that, for instance, puts the arguments into a list and calls a sort algorithm on them. However, in this paper we have restricted ourselves to implementing tactics by the application of rewrite rules. In this case, reordering can be done by a set of 11 commutated associative laws, of which some typical examples are:

$$\begin{aligned} F(T, F(S, A)) &\Rightarrow F(F(S, T), A) \\ F(T, F(A, S)) &\Rightarrow F(F(S, T), A) \\ F(F(T, S), A) &\Rightarrow F(F(S, T), A) \end{aligned}$$

The right hand side of each of these 11 rules is $F(F(S, T), A)$. The 11 different left hand sides are the 6 permutations of $F(T, F(S, A))$ and the 6 permutations of $F(F(T, S), A)$ except $F(F(S, T), A)$. They are to be applied whenever S and T are in $Base_1 \cup \ldots \cup Base_k$ with $S \prec T$ and A is in Cls. Applied in this way they implement a kind of bubble sort. The 11 rules could be automatically generated from the standard associativity and commutativity laws.

We will call this proof plan *reorder*. It will have the effect of converting a BNF class:

$$Cls \quad := \quad Base_1 | \ldots | Base_k | F(Cls, Cls)$$

into the class:

$$Cls' \quad := \quad Base_1 | \ldots | Base_k | F(Cls', Base_1) | \ldots | F(Cls', Base_k)$$

just as with *left_assoc*, but with the additional condition that for all base elements S and T, whenever S occurs before T in any member of Cls', then $S \prec T$. So in this case it is not possible to describe the effect of the proof plan using standard BNF notation alone. To describe the ordering effect we need either an expanded BNF notation or additional meta-formulae.

5.3 Thinning

If F is unary and obeys a thinning rewrite rule, *i.e.* one of the form:

$$F(F(X)) \quad \Rightarrow \quad X$$

then we can remove multiple applications of F from the layer. Examples of such functions are $\neg$ and $-$, which obey the rules:

$$\begin{aligned} \neg\neg p \quad &\Rightarrow \quad p \\ - - x \quad &\Rightarrow \quad x \end{aligned}$$

The *thin* proof plan will apply such rules exhaustively. This will convert the BNF form:

$$Cls \quad := \quad Base_1|\ldots|Base_k|F(Cls)$$

into the form:

$$Cls' \quad := \quad Base_1|\ldots|Base_k|F(Base_1)|\ldots|F(Base_k)$$

Thinning could be generalised to the use of any rewrite rule of the form:

$$F^n(X) \quad \Rightarrow \quad F^m(X)$$

where $F^n(X)$ means $F(F(\ldots F(X)\ldots))$, n times, and $m < n$, but the case $n = 2$ and $m = 0$ seems to the only common one.

Thinning can also be seen as removing $\lambda x.F(F(x))$[2], but this requires a hard to motivate transformation of Cls to:

$$Cls \quad := \quad Base_1|\ldots|Base_k|F(Base_1)|\ldots|F(Base_k)|F(F(Cls))$$

so it is unclear whether this view of thinning is helpful.

6 Normalization

The proof plans outlined above are rather small scale. To obtain useful normalizers we must combine them together. This can be done in a variety of ways.

Special-Purpose : We can construct special-purpose proof plans, *e.g.* for conjunctive normal form or polynominal normal form, by instantiating the rewrite rule arguments of our tactics with particular rules and combining the resulting tactics in a fixed combination. For instance, a disjunctive normal form procedure could be constructed by combining *remove* $\leftrightarrow$, *remove* $\rightarrow$, *stratify* $\neg$ over $\vee$ and $\wedge$, *stratify* $\vee$ over $\wedge$, and *thin* $\neg$.

General-Purpose : We can construct a general-purpose normalizer by combining our tactics together into a super-tactic, but without instantiating their rewrite rule arguments. For instance, we might form a general-purpose super-tactic that applied repeated *remove*s, followed by repeated *stratify*s, followed by an appropriate reorganising tactic: *left_assoc*, *reorder* or *thin* to each layer. The preconditions of this super-tactic would have to determine how many repeats of each sub-tactic were required and with what rewrite rules.

Dynamic : We can leave it to the planner to determine how to combine the sub-tactics according to the situation it confronts at run-time. It would use the preconditions and effects of these sub-tactics to determine how best to combine them. See §10.1 for an example.

Each of these combination techniques is valid and has its role to play. Our CLAM system already employs aspects of each of them in the domain of inductive proofs.

[2] I am indebted to Toby Walsh for this observation.

7 The Meta-Logic

The preconditions and effects of the proof plans described above will be formulae of a meta-logic in which most of the functions and predicates will manipulate and describe expressions and BNFs. Among the functions and predicates required will be the following:

- $Exp{:}Cls$: means that expression Exp is in syntactic class Cls.
- $in(Disj, Cls)$: means that $Disj$ is one of the disjuncts of class Cls.
- $delete(Disj, Cls)$: is the result of deleting disjunct $Disj$ from Cls.
- $subst(New, Old, Cls)$: is the result of substituting New for Old in Cls.

Using this meta-language, the preconditions of:

$$delete(Exp, F(X_1, \ldots, X_m) \Rightarrow T(X_1, \ldots, X_m))$$

can be written:

$$\begin{array}{l} \exists Cls, Cls_1, \ldots, Cls_m. \\ \qquad\qquad Exp{:}Cls \wedge in(F(Cls_1, \ldots, Cls_m), Cls) \wedge \\ \forall X_1{:}Cls_1, \ldots, X_m{:}Cls_m.\ T(X_1, \ldots, X_m){:}delete(F(Cls_1, \ldots, Cls_m), Cls) \end{array}$$

This says that Cls, the syntactic class of Exp, contains a disjunct of the form $F(Cls_1, \ldots, Cls_m)$, and that the right hand side of the removal rule belongs to the class formed from Cls by removing this disjunct. We follow the standard naming convention that object-level variables are meta-level constants and that object-level expressions serve as their own meta-level names.

8 Linear Algebra

The Hodes decision procedure for linear arithmetic starts by applying a sequence of normalizers, which transform the input expression into a variable free form which can be evaluated. These normalizers can be implemented using the proof plans described above.

By linear arithmetic, we mean the expressions generated by the following BNF.

$$\begin{array}{rcl} fm & := & prop|\neg fm|fm \vee fm|fm \wedge fm|fm \rightarrow fm|fm \leftrightarrow fm|\exists var.\ fm|\forall var.\ fm \\ prop & := & tm = tm|tm < tm|tm > tm|tm \leq tm|tm \geq tm \\ tm & := & var|rat| - tm|tm + tm|tm - tm|rat.tm \\ var & := & x|y|z| \ldots \\ rat & := & nat/nat|(-nat)/nat \\ nat & := & 0|succ(nat) \end{array}$$

where fm stands for formulae, $prop$ for propositions, tm for terms, var for variables, rat for rational numbers and nat for natural numbers.

Hodes' procedure works by eliminating all quantifiers from formulae of class fm, without changing their free variables. If the input formula contains no free variables then the output formula will contain neither free nor bound variables, and so can be evaluated to determine its truth value. At each stage of the procedure an innermost existentially quantified sub-formula, is chosen, *i.e.* a sub-formulae of the form $\exists x.\ p(x)$, where $p(x)$ contains no quantifiers. This existential quantifier is then eliminated by transforming the sub-formulae into an equivalent one not containing x. If necessary, an innermost universally quantified formula is transformed into an innermost existentially quantified formula by removing $\forall$ in favour of $\exists$ and $\neg$.

The transformation of the sub-formula can be described using our proof plans as follows.

1. *remove* is used to remove $\leftrightarrow$, $\rightarrow$, $>$, $\geq$ and $\leq$, in that order.
2. *stratify* is used to move $\neg$ inside $\vee$ and $\wedge$.
3. *thin* is used to eliminate multiple $\neg$s.
4. The extended version of *remove*, described at the end of §3, is used to remove $\neg$.
5. *stratify* is used to put formulae in disjunctive normal form.
6. *stratify* is used to move $\exists$ inside $\vee$.
7. Various equation and inequality solving procedures are used to eliminate x from the sub-formula, so that the quantifier becomes redundant and can be dropped.

9 Equation and Inequality Solving

As can be seen from step 7 of the description above, in order to implement Hodes' linear arithmetic decision procedure using proof plans we will need to devise proof plans for solving equations and inequalities. Fortunately, much of this work has already been done. Our PRESS equation solving program, [Sterling *et al* 89], was implemented using a precurser of proof plans, which we called PRESS *methods*. The PRESS methods of *polysolve*, *collection* and *isolation* are more than adequate for the simple linear equation solving required in Hodes' procedure. They could easily be realised as proof plans. This would require the development preconditions and effects in the meta-language used for the normalization proof plans above. In particular, a normalizer for polynomial normal form, used by *polysolve*, could be implemented using *remove*, *stratify* and *reorder*. PRESS also contains some simple methods for manipulating inequalities, but these will need extending.

10 Facilitating Flexible Interaction

In devising proof plans for normalization we have taken trouble to make these as general-purpose as possible. For instance, rather than design separate proof plans for disjunctive normal form and polynomial normal form, we have recognised the common structure underlying these normalizers and represented these as the general-purpose proof plans *remove*, *stratify*, *thin* and *reorder*, from which these special-purpose normalizers can be constructed. The benefit of this organisation is that these general-purpose proof plans can be reused to build normalizers for new domains and can be applied flexibly to take account of additional information. To see this consider the following examples. For explanatory purposes they have been kept as simple as possible and are, as a consequence, rather artificial.

10.1 Application to New Domains

The following BNF describes a subset of list processing expressions.

$$tm \quad := \quad nil|var|palindrome(tm)|app(tm,tm)|rev(tm)$$

The existence of the rewrite rule:

$$rev(app(l,k)) \quad \Rightarrow \quad app(rev(k),rev(l))$$

suggests stratification of *app* above *rev*, provided *palindrome* can be removed first. This can be done with the rewrite rule:

$$palindrome(l) \quad \Rightarrow \quad app(l,rev(l))$$

So removing *palindrome* transforms the BNF to:

$$tm \quad := \quad nil|var|app(tm, tm)|rev(tm)$$

and stratification transforms it to:

$$\begin{aligned} tm \quad &:= \quad newbase|app(tm, tm) \\ newbase \quad &:= \quad nil|var|rev(newbase) \end{aligned}$$

Finally, multiple applications of *rev* can be thinned with the rewrite rule:

$$rev(rev(l)) \quad \Rightarrow \quad l$$

to get the BNF:

$$\begin{aligned} tm \quad &:= \quad newbase|app(tm, tm) \\ newbase \quad &:= \quad nil|var|rev(nil)|rev(var) \end{aligned}$$

This example shows that our general-purpose proof plans can be combined to make a normalizer for a new area of mathematics. This could be done on-line, *i.e.* built up dynamically, in response to a particular normalization problem, using previously proved lemmas or even the hypotheses of the current conjecture, as the required rewrite rules.

10.2 Application to Extended Domains

Suppose we have constructed a special-purpose normalizer, $poly\_form$, for polynominal normal form, which is designed to transform expressions of the BNF class:

$$tm \quad := \quad var|rat| - tm|tm + tm|tm - tm|rat.tm \tag{7}$$

into the BNF class:

$$\begin{aligned} tm \quad &:= \quad summand|tm + summand \\ summand \quad &:= \quad var|rat|rat.var \end{aligned}$$

The proof plan, $poly\_form$, is not restricted to expressions in the class (7). Consideration of the preconditions for *remove* given in §7 will show that the class of the input expression must have certain properties, but is not, in general, constrained to be a particular class. We design the preconditions of proof plans to be the minimum required for the tactic application to make sense. In particular, $poly\_form$ will apply to the expression[3] $max(a) + k$, where a is an array of numbers and $max(a)$ is the maximum element of this array. Suppose a is an expression of class $array\_tm$. The BNF of the input expression might have the form:

$$tm \quad := \quad var|rat| - tm|tm + tm|tm - tm|rat.tm|max(array\_tm)$$

Since it does not make a recursive call to tm, $poly\_form$, will treat $max(array\_tm)$ as if it were an additional base class, which will be carried through the sequence of BNFs, so that *summand* is defined as:

$$summand \quad := \quad var|rat|rat.var|max(array\_tm)$$

Thus the use of weakest preconditions in proof plans will enable $poly\_form$ to treat terms that lie outside its domain of knowledge. In this case such terms are treated as if they were variables.

This example shows that special-purpose proof plans can be applied outside their intended syntactic class, and will still behave as required.

[3] Taken from [Boyer & Moore 88].

10.3 Bridging between Proof Plans

If an additional disjunct in the BNF of a class *does* make a recursive call to the class, then it cannot be treated as a base case. For instance, if tm were defined as:

$$tm \quad := \quad var|rat|-tm|tm+tm|tm-tm|rat.tm|double(tm)$$

where $double(x) = x+x$, then $double(tm)$ cannot be treated as a base class. Thus the preconditions of $poly\_form$ will fail, so that it cannot be applied. In particular, the parts of its precondition which are inherited from one of its stratify sub-proof-plans will fail because the set of distributive laws will not be complete. At this stage the planner will back up and try to plan its way around the blockage. One way that it might find to do this will be to *remove double*, before the calls to *stratify*, using the rewrite rule:

$$double(x) \quad \Rightarrow \quad x+x$$

This example shows that special-purpose proof plans can fail if they are applied to expressions too far outside their intended syntactic class. In this case plan formation can be used, dynamically, to patch holes in the old proof plans. These patches can be composed of any combination of other proof plans. This permits a flexible interaction between normalizers and other parts of the theorem prover. This is similar to the interaction allowed in the Boyer-Moore implementation of the Hodes' procedure, but in a more modular form. In particular, we could add or delete proof plans from the store of those available to the proof planner and it will use the methods of those available to combine them in the most suitable form it can find. No major reprogramming will be required.

11 Conclusion

In this paper we have proposed the use of proof plans for representing normalizers, as a first step in using proof plans to represent decision procedures. We have outlined some general-purpose proof plans: *remove*, *stratify*, *left_assoc*, *reorder* and *thin*, from which a wide variety of normalizers can be constructed. The advantages of this method of representing normalizers are as follows:

- We can represent a wide variety of special-purpose normalizers in a uniform way.
- These special-purpose normalizers can be successfully applied to expressions just outside their intended domain of application.
- When such applications fail, these special-purpose normalizers can be dynamically repaired, by inserting new proof plans as patches. This allows a flexible interaction between those proof plans concerned with normalization and those concerned with other aspects of theorem proving.
- New normalizers can be constructed for new domains, either 'on-line', *i.e.* for a particular expression during a proof, or 'off-line' for a whole new class of expressions.

These advantages go some way to address the issue, raised in [Boyer & Moore 88], of how to integrate decision procedures into theorem provers in a way that facilitates their flexible interaction. In particular, it suggests a modular way to do this that facilitates modification of either decision procedure or theorem prover without major reprogramming.

However, these proposals are still some way from realisation, and the advantages listed above are only predictions. Much remains to be done before they can become a reality.

- The proof plans outlined above must be implemented in a system such as CLAM. Both the tactics and their associated methods must be built. The CLAM meta-language must be extended to cover the description and manipulation of BNFs required in these methods.

- In order to implement decision procedures like that of Hodes, our set of proof plans must be extended. Some of the required proof plans can be adapted from the PRESS equation solver, but others will also be needed.
- Further normalizers and decision procedures must be analysed to discover additional general-purpose proof plans.
- Empirical testing must be conducted to see whether the flexibility gained by implementing normalizers and decision procedures declaratively justifies the increased cost over the usual procedural implementation. It might be possible to avoid this cost by compilation of the declarative representation into a procedural one.

Nevertheless, even at this early stage this appears to be a very fruitful application of proof plans.

References

[Boyer & Moore 88] R.S. Boyer and J.S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In J.E. Hayes, J. Richards, and D. Michie, editors, *Machine Intelligence 11*, pages 83–124, 1988.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy *et al* 91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Desimone 89] R.V. Desimone. *Learning Control Knowledge within an Explanation-Based Learning Framework*. Unpublished PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1989.

[Hodes 71] L. Hodes. Solving problems by formula manipulation. In *Proceedings of the Second IJCAI*, pages 553–559. The British Computer Society, 1971.

[Sterling *et al* 89] L. Sterling, A. Bundy, L. Byrd, R. O'Keefe, and B. Silver. Solving symbolic equations with PRESS. *J. Symbolic Computation*, 7:71–84, 1989. Also available from Edinburgh as DAI Research Paper 171.