The Improvement of Prolog Program Efficiency by Compiling Control: A Proof-Theoretic View

Geraint A Wiggins

DAI Research Paper No. 455

Based on paper
presented to META-90 Workshop,
April 1989,
Leuven, Belgium

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

The Improvement of Prolog Program Efficiency by Compiling Control: A Proof Theoretic View

Geraint A Wiggins

Abstract

In this paper, I report on progress in applying *Proof Planning* techniques developed by Bundy *et al* at Edinburgh to the ideas of *Compiling Control* proposed by Bruynooghe, de Schreye *et al* at Leuven.

The overall theme of the paper is the application of a proof-theoretic view of Prolog program execution to the issues involved in performing Compiling Control. In particular, I show how the use of strict data-typing can help to guide an inductive proof, analogous to the execution of a recursive Prolog program for some given computation rule, to produce an execution tree equivalent (in many cases) to that required for the compilation of control under the Leuven régime.

I conclude that this is a worthwhile direction to follow, though there are are some cases not covered by the proof-theoretic approach which are dealt with by the techniques of abstract interpretation involved in Compiling Control.

1 Acknowledgements

This paper presents work in progress under the ESPRIT "Computational Logic" project #3012. I gratefully acknowledge the input of Maurice Bruynooghe, John Gallagher, Danny de Schreye and Kristof Verschaetse during recent discussions in Leuven and the continuing support of my colleagues in the DReAM group in Edinburgh, especially Alan Smaill and Frank van Harmelen. The software built as part of this project is based on work by Julian Richardson, described in [Richardson 89].

2 Introduction

In this paper, I will discuss the relationship between work carried out in the Katholieke Universiteit Leuven on the compilation of non-standard execution rules for Prolog programs and continuing work in the University of Edinburgh on planning for (mathematical) proofs.

I will begin with a brief exposition of the Compiling Control technique, and briefly reproduce an example given in [Bruynooghe *et al* 89]. Next, I will introduce the idea of Proof Planning in the mathematical domain, and show how we can adapt it and the research tools currently used to produce a theorem prover for a Prolog-like language under resolution.

I will then return to the example cited above, and show how the Proof Planning technique can reproduce the Compiling Control results, in some cases in a more directly justifiable way than that suggested in the Compiling Control literature. In particular, I will suggest that the use of inductive theorem proving techniques (based on the DReAM[1] group ideas normally applied to general mathematical proofs) can lead to a more formal treatment of recursive programs than

[1] "Discovery and REAsoning in Mathematics"

in Compiling Control. This will require the use of a strong type theory for Prolog, and it will be seen that such a requirement leads to some limitations not present in the original Compiling Control work.

3 Compiling Control

3.1 Introduction

The primary goal of K U Leuven's Compiling Control work ([Bruynooghe *et al* 89, De Schreye & Bruynooghe 89, De Schreye & Bruynooghe 88]) is to improve the execution speed of a given Prolog program, by manipulation only of the computation rule under which it is executed[2], and not by logical transformation like that between naïve and accumulating reverse. The execution tree obtained includes co-routining behaviour based on instantiation, which is not available under the standard computation rule. The computation rule corresponding with the new tree is implemented as a meta-interpreter specialised to admit only the efficient execution of the program, even under the standard computation rule.

The procedure we must follow to perform this transformation is in two phases. First, we must produce a *symbolic trace tree* (like that shown in 2 representing the efficient execution. Second, we must produce a new program, the meta-interpreter mentioned above, which runs under the standard computation rule to give the same success set as the original[3].

3.2 The "Symbolic Trace Tree"

The process of producing the symbolic trace tree has been partially automated as part of the Leuven work. It involves successive execution of the program to be improved for a (small) number of concrete queries. At each step in the execution of a particular query the user decides which of the current subgoals should be dealt with next, and whether it should be unfolded (in the usual sense) or fully executed. Each time a new query is executed, the new execution tree is combined with that (if any) already existing, in some abstraction including both, which is generated on the fly. If the queries are well chosen, the abstraction introduced covers all the data for which the program can succeed, and the resulting tree is a correct abstract representation for all possible successful executions of the program (and possibly non-terminating ones, as well). In the event that the abstractions chosen are too general, datatyping implicit in the program code will often prevent overgeneration of results.

3.3 Generating a Specialised Meta-Interpreter

Once such a tree has been produced (manually or semi-automatically), we can use an almost trivially simple mechanical procedure to generate the new program from it. By representing each node in the tree (*viz* each state in the execution) by a distinct predicate, whose arguments are the subgoals to be proved at that state, we can represent state changes by Prolog predicates mapping from a state (the clause head) to an immediate successor (the clause body) in the usual Prolog way. We can subsequently improve on this in two ways. Firstly, if there is a sequence of more than two states which includes no branches, we can collapse it into a single state change (in a sort of weak partial evaluation), which causes more efficient execution of the new program because fewer

[2] I will use the term "executable" to mean "giving at least one solution, if run to completion".

[3] Execution of the new program is only non-deterministic where branches guided by data content (*eg* a choice between empty and non-empty lists) or undecidable branches (*eg* where two clauses have the same head) occur in the symbolic trace tree; such branches are obviously unavoidable in general. This will become clear in the forthcoming example.

resolution/unification steps are involved. Secondly, if any part of the new program represents an execution equivalent to that derived under the standard computation rule, that part can be replaced by a direct call to the (original) predicate(s) involved, giving a "full execution", which removes the interpretation overhead for that section of code.

Currently, Leuven's incremental trace tree construction algorithm requires that these "full execution" sections are labelled as such by the user during the initial construction phase – therefore, the user must exercise intelligent choice in determining the correct option *before* the complete trace tree has been seen.

3.4 An Example – Permutation Sort

Let us now clarify all of this with an example: the "slowsort" or "permutation sort" algorithm. The example is taken directly from [Bruynooghe *et al* 89].

```
sort( X, Y )                :-  perm( X, Y ), ord( Y ).
perm( [], [] ).
perm( [H|T] [U|Q] )         :-  del( U, [H|T], W ), perm( W, Q ).
del( X, [X|Y], Y ).
del( X, [Y|U], [Y|Q] )      :-  del( X, U, Q ).
ord( [] ).
ord( [X] ).
ord( [X,Y|Z] )              :-  X ≤ Y, ord( [Y|Z] ).
```

Figure 1: The Standard Slowsort Program

Figure 1 shows the Prolog Slowsort program, which runs inefficiently under the standard computation rule. Figure 2 is the tree representing in abstract terms (the abstraction being simply to the domain {ground, variable} with some extra structure for representing lists) the most efficient execution tree for the Slowsort program called with a ground first argument and a variable second. Ground terms and functors are represented by lower case letters, and variables by upper case. At each state, the next is produced by unfolding an underlined subgoal or fully executing a boxed subgoal. Assignments taking place as part of a state change are attached to the arc connecting the states.

As part of the process of creating the tree, we have somehow to notice that the final state is an alphabetic renaming, with the same instantiation pattern, of the state pattern connected to it in the figure by the broken-line arrow. We therefore assume that we can view the computation as looping and that we need not expand the trace tree any further. We also need to know that we can allow full execution of the del/3 calls.

Once we have produced the tree, we can apply the mechanical procedure outlined above to produce the set of meta-program clauses representing the state changes. After applying the optimisations suggested above, we are left with the efficient specialised meta-interpreter shown in figure 3. Note that the s1 clause exactly represents the transition between the first and second states, and that the fully executed predicate, del/3, is included as normal Prolog calls and not meta-interpreted. The call to $\leq/2$ is also performed at the system level, because its semantics are inexpressible in terms of unification and unfolding alone.

The small number of clauses has been achieved by collapsing together sequences of states containing only one choice point – this is the most complete collapsing possible.

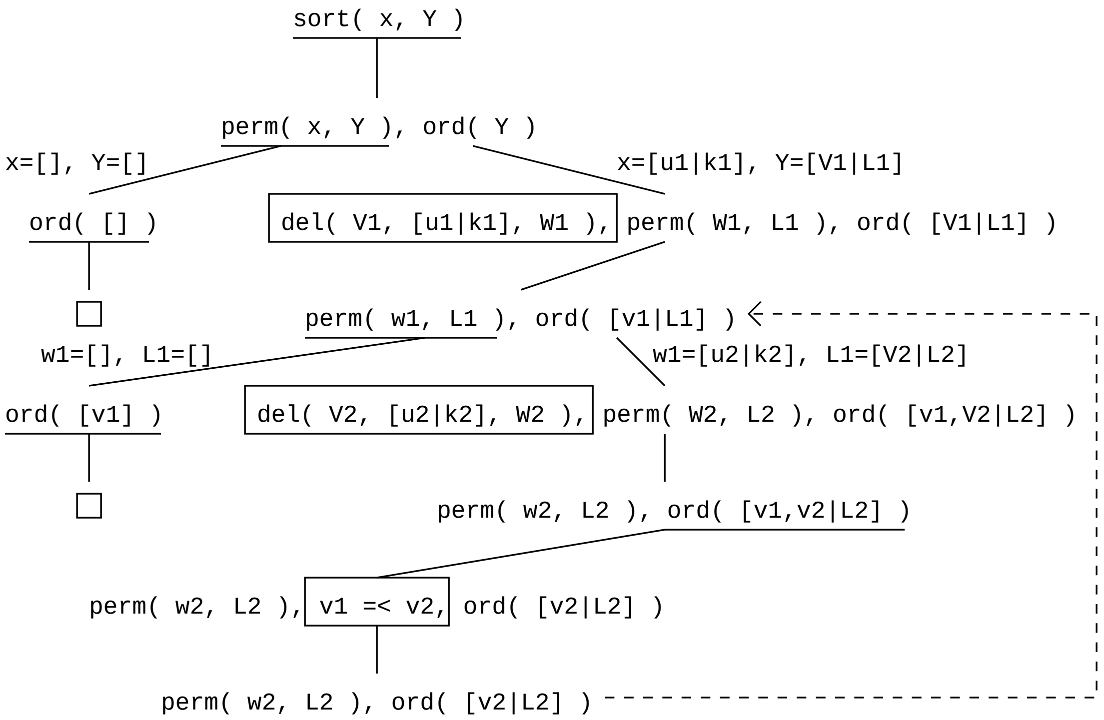

Figure 2: The Symbolic Trace Tree for Slowsort

4 Oyster/CLAM and Proof Planning

Proof Planning has been developed in Edinburgh University's Department of Artificial Intelligence by the DREAM Group [Bundy *et al* 90]. The technique is a means of applying high-level strategies to the elaboration of mathematical proofs. A proof planner, CLAM, has been designed to work in conjunction with a proof development system, Oyster [Horn 88], which in turn was based on the NuPrl proof development system implemented by Constable et al [Constable *et al* 86]. Oyster uses Martin-Löf Type Theory to express theorems and the steps in their proofs.

Proof steps are applied (manually or automatically) in Oyster by starting with a goal theorem, and applying *rules* and *tactics* to perform logical transformations on it. (Note that this is the reverse of Gentzen Sequent Calculus notation: the Oyster proof tree grows *backwards* from the goal, reducing the theorem to "true". Therefore rules such as introduction work in the opposite way from what might be expected.)

Standard proof rules such as ∀-introduction and ∀-elimination may be applied at any stage in a proof, and are mostly initiated with one word commands, Oyster usually determining the meaning of the command from the particular context.

A tactic is a more complex rule, defined in terms of the basic rules built into Oyster, but with the potential inclusion of arbitrarily complex Prolog code. Tactics are often used as procedures or macroes to encapsulate complex manipulations which happen frequently, such as the application of particular induction schemata.

Tactics and rules may be manually applied to nodes in the (partial) proof tree individually or combined together by *tacticals*, or by means of the application of a *proof plan*. Proof plans are currently automatically generated by the proof planner, CLAM [van Harmelen 89]. To enable CLAM to operate, *methods* are used to specify tactics in meta-logic; they include pre- and post-

```
s1( sort( K, L ) )                                  :-  s2( perm( K, L ), ord( L ) ).

s2( perm( [], [] ), ord( [] ) ).
s2( perm( [U1|K1], [V1|L1] ) )                      :-  del( V1, [U1|K1], W1 ),
                                                        s5( perm( W1, L1 ), ord( [V1|L1] ).

s5( perm( [], [] ), ord( [V1] ) ).
s5( perm( [U2|K2], [V2|L2] ), ord( [V1,V2|L2] ) )   :-  del( V2, [U2|K2], W2 ), V1 ≤ V2,
                                                        s5( perm( W2, L2 ), ord( [V2|L2] ) ).
```

Figure 3: The Specialised Interpreter for Slowsort(Ground, Variable)

condition information. CLAM is able to use this information to form plans (via the user's choice from a number of planning strategies) about how best to perform a proof. The search space for the planner has usually proved to be considerably smaller than that for the object-level language of a given proof; moreover, plan steps in the mathematical domain are generally considerably cheaper to execute than object-level steps.

In particular, current work concentrates on *Middle-Out Proof Planning*, guided by the process of *rippling out*. In this approach, proof plans are formed by the proposition of a general form of induction containing meta-variables. The movement of the symbols in the proof around these variables (characterised in general as *rippling out* and described by a small number of *wave rules* at the meta-level) can suggest the particular form of the induction – see [Bundy 88] for more detail.

A part of the on-going work concentrates on program synthesis using these techniques. In Martin-Löf Type Theory, each application of a rule (proof step) corresponds exactly with a step in the construction of a functional term, representing in some constructive sense the force of the proof. In particular, in an existential proof this *extract term* is a program which constructs the item proved to exist. Therefore, if we prove a theorem of the general form

$$\mathit{hypotheses} \vdash \forall \mathit{input}.\exists \mathit{output}.\mathit{specification}(\ \mathit{input},\ \mathit{output}\ )$$

(where *specification* logically defines the program, and *hypotheses* are the logical precepts required for proof – *eg* knowledge about properties of types) we simultaneously produce a program that generates the output specified for all (well-typed) input. See [Madden 88, Madden 89] for more detail.

5 Prolog as a Typed Constructive Logic

Since the essential goal of this research is to apply proof planning techniques to the compiling control domain, let us now consider how to do so in more detail. In particular, we must consider the issue of datatype *narrowing* which typically takes place in the execution of a Prolog program, because of datatyping implicit therein. I will outline two possible solutions to this problem.

In the Compiling Control work, a Prolog program is viewed as a sort of specification, and then reasoning is used to derive the appropriate solution set from that specification, given a set of queries from which can be inferred the general form of a query. In a proof system using resolution ($\sim$ unfolding) as its proof step, there is a fairly obvious (but naïve) analogy between the existential form (shown above) used in Oyster program synthesis (where the *specification* includes all of the information about the logic of the program we wish to synthesise) and the combination of the program specification and general query (where the hypotheses constituting the program clauses are used to expand the top level goal):

$$\vdash \forall \mathit{input}.\exists \mathit{output}.\mathit{specification}(\ \mathit{input},\ \mathit{output}\ )$$

$$\mathit{Program} \vdash P(\ \mathit{ground},\ \mathit{variable}\ )$$

The proof of or reasoning about these entities proceeds in both *Oyster* work and Compiling Control by stepwise reduction to some other entity(ies) which are in some sense generally provable: basic sequents in Martin-Löf, and an empty subgoal set (or subsumed goal, as we will see later) in Prolog. Note that the query here is viewed as a positive logical expression, and not as the usual denial. This is the intuitive view in a proof-theoretic framework, and is logically correct: this language is a subset of λProlog, for which the positive expression of queries is acceptable – see [Miller & Nadathur 88].

One approach to this problem, then, might be to cast the program specification in Martin-Löf Type Theory, specify the mode in terms of universal (∼ ground) and existential (∼ variable) quantifiers, and synthesise an efficient program (which would, presumably, reflect the structure of the Compiling Control solution) with the technique described above.

There are, however, two serious problems with this. The first is the functional nature of the extract term in Martin-Löf Type Theory. The properties of such an extract term mean that external non-determinism in a Prolog program is not expressible in the synthesised program. Thus, the success set of the original Prolog program is not in general preserved, a drawback which does not arise in Compiling Control. This problem is reflected in the looseness of the variable-existential analogy; strictly speaking the existential proof only has part of the force of the Prolog goal, which implies universal quantification over the solutions.

Secondly, (and in some cases more problematically), specification of such a query in Martin-Löf Type Theory normally requires that the exact datatype of the input be known in advance. Now, it is not in general true that the datatypes of a program input are easily visible from inspection of the program, and in general, they may not be so at all – consider, for example, the N-queens problem, which has no solution for $N \in \{2,3\}$, so the (sub)type is $\{1\} \cup \{x | \mathrm{integer}(x)\ \&\ x > 3\}$. Because of the abstraction involved in Compiling Control, this problem does not arise: a correctly chosen abstraction will normally cover a superset of the necessary datatype, and execution of the improved program will simply fail for input data for which no solution exists. In proof theoretic terms, this solution is clearly unacceptable. Thus, the analogy between groundness and universal quantification breaks down.

There are two possible solutions for this latter problem. The first is the use of explicit specification of subtypes as implications in the top-level goal, like this:

$$\vdash \mathit{InType}(\ \mathit{input}\ ) \rightarrow P(\ \mathit{input},\ \mathit{output}\ ).$$

where InType is well-typedness predicates on the input which may include arbitrarily complex logical expressions. respectively. At first sight, this solution seems not to address the issue, because the type predicates are defined in advance. In principle, however, this is a classic application of middle-out proof planning – the two predicate names can be viewed as meta-terms and instantiated as necessary as the planning proceeds.

Give this solution, the type hierarchy shown in figure 5 is adequate, since any type inferences made during the proof planning may be included in the type predicate at proof time.

Alternatively we might use a proof system which is capable of type-inference, or, rather, is able to construct and justify suitable sub-types (*eg* all integers other than 3) for a given proof on the fly. This is in principle possible in Martin-Löf Type Theory, and will be the subject of future work in this project; the implications, however, in terms of complexity of the well-typedness checks occurring in the proof, are formidable. Nevertheless, unless some equivalent type checks are made, a proof including type inference becomes less than a proof: given any step involving the narrowing of a type without some account for the subtype excluded by the narrowing, any universal quantification in the initial goal may become partly uncovered by the proof, which is therefore invalid.

Another problem, which is rather closer to an implementation detail, is that it may often be desirable, when working with "real" programs, to specify extra information about built-in predicates – in particular, the instantiation states and input datatypes required for error-free success. This is not possible in Martin-Löf Type Theory.

As a first step towards a working system which will deal with these problems, I have replaced Martin-Löf Type Theory in Oyster/CLAM with a language equivalent to pure[4] Prolog (*viz* specification of a program in Horn Clauses, with execution viewed as proof by unfolding and symbolic evaluation). Given the specification of appropriate methods, CLAM can plan over the proof search space in exactly the same way as it can over mathematical proofs in ordinary Oyster. The "theorem" to be "proved" is specified as a set of Horn clause hypotheses, and a Prolog-style goal expressed in abstract, typed terms. This will be made clearer by the example below, which is output by this new system, Prolog-Oyster/CLAM.

6 Specifying Programs in Prolog-Oyster/CLAM

To begin with, we must specify the program we wish to improve, and the query for which we wish to optimise it. In Prolog-Oyster, the slowsort program and query from the Leuven example given before look like figure 4 (except that some minor syntax has been changed to improve readability).

```
sort( var x, var y)                           ←   perm(var x,var y),ord(var y).
perm( [], []).
perm([var x| var y],[var u| var q])           ←   del(var u,[var x| var y],var w),
                                                  perm(var w,var q).
del(var x,[var x| var y],var y).
del(var x,[var y| var u],[var y| var q])]     ←   del(var x,var u,var q).
ord([]).
ord([var x]).
ord([var x,var y| var z])                     ←   var x ≤ var y,
                                                  ord([var y| var z]).
var x ≤ var y                                 ←   sys( gnd a:number struct ≤ gnd b:number struct,
                                                   gnd a:number struct ≤ gnd b:number struct,
                                                   predicate )

⊢ sort(gnd x:number list, var ans:term).
```

Figure 4: The Slowsort Specification in Prolog-Oyster

Most of this specification is self-explanatory to those familiar with Prolog. The two parts requiring explanation are the specification of the behaviour of $\leq/2$ and the mode and type information in the goal.

The $\leq/2$ specification means the following. The appearance of the sys functor in the body denotes a built-in predicate. The first of its arguments specifies the call-time instantiation state required for error-free execution of the predicate. The second indicates the output instantiation

[4] With the addition of knowledge about object-level built-in predicates.

state (which is the same as at call time, in this case). The third argument, "predicate", indicates that this call makes no change to the data involved – it merely succeeds or fails, and thus is easy to deal with in a proof.

The mode abstraction used here is simply {gnd, var} for {ground, variable}. The system is capable of reasoning about constants, too, but this facility is not required by this example. In performing an inference step, we are required to ensure that no term becomes *less* ground than before – if any one did, the inference would imply an unmotivated generalisation.

Datatypes in the current implementation are organised in a partial lattice as shown in figure 5. (This would need to be extended considerably, were we to take the sub-type approach described above.) Type inference from the program corresponds with narrowing, that is, moving down the tree. X is a variable over types. The type and mode inferences together correspond with "one-way unification" in [Verschaetse *et al* 88].

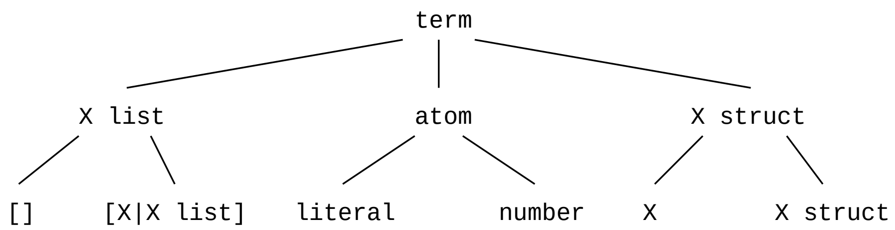

Figure 5: Type Hierarchy in Prolog-Oyster

7 Compiling Control by Proof Planning

7.1 Inference Rules

Now, in order to analyse the behaviour of the program, we need some inference rules based on resolution. The fundamental one of these will be the conventional *unfold*, for the obvious reasons. However, because we are working with abstractions (which can often be viewed as very like datatypes), we need to be able to express coverage of a given abstraction by a set of data items. In other words, if we call a predicate with separate definitions for input of (say) empty and non-empty lists (*ie* a *disjunctive* branch in a concrete Prolog execution) we need to perform a "proof" for both possibilities. Thus, the branch we require in the Oyster tree is *conjunctive*.

Given such a branching rule, we have all the mechanism we need manually to reproduce the trace tree shown in figure 2, except for the recognition of the loop indicated there by the broken arrow and the full execution of the del/3 predicate. This will be dealt with by application of the rippling out idea mentioned before, and will be detailed later, in the example.

Assuming for the moment that we can deal with loops, we now need to be able to perform planning for the execution, which can be thought of informally as a sort of quasi-proof (it is not a real proof because of the unproven type inference involved). For this purpose, we have methods, corresponding one-to-one with the rules (*unfold*, *branch* and *loop* – these will be explained during the example), which describe their behaviour, in an entirely uncontroversial way.

7.2 Planning Strategy

For efficiency's sake (at the planning level) we generally want to find the shortest correct "proof" for a given specification. For this purpose CLAM's iterative deepening planner is ideal: it produces the same result as an exhaustive breadth first search, but is rather faster, because the implementation currently includes an optimisation described below. In the final analysis, goal-directed planning, including heuristics about predicates appearing in the subgoals to be proven, will be an ideal solution. For the moment, and certainly for this example, iterative deepening is adequate.

Planning follows the following procedure. At each node in the (growing) execution tree, the methods are tried, one by one, on each unproven subgoal, from left to right. The first to be tried is the *loop* method, so that any well-formed recursion (see below) will be found before other expansions are performed. The other methods are mutually exclusive with respect to applicability, so their order is theoretically unimportant.

7.3 The Slowsort Example

Let us take a few steps through the planning of the execution for slowsort. Remember that we are aiming for the same tree of inference steps that is produced by the Compiling Control approach (except for the full execution of del/3).

7.3.1 Unfolding a Linear Tree Segment

At the top of the tree, with goal

$$\vdash \quad \mathrm{sort}(\ \mathit{gnd}\ x : \mathit{number\ list},\ \mathit{var}\ y : \mathit{number\ list}\ ). \tag{1}$$

we first look for applicable methods. The *loop* method is inapplicable, because there is no suitable hypothesis on which to base a loop (see below). The *branch* method is inapplicable because only one program clause will unify with the subgoal. The only method we can apply here, then, is the *unfold*ing of the single subgoal with the first clause of the program. This application then leaves us with the subgoals

$$\vdash \quad \mathrm{perm}(\ \mathit{gnd}\ x : \mathit{number\ list},\ \mathit{var}\ y : \mathit{number\ list}\ ),\ \mathrm{ord}(\ \mathit{var}\ y : \mathit{number\ list}\ ). \tag{2}$$

7.3.2 Introducing a Branch

By applying the same sort of reasoning about applicability as before, we find that the first successful possibility is a *branch* between the perm/2 subgoal, and the two clauses of the perm/2 program (*branch* is selected rather than *unfold*, here, because there is more than one unifiable program clause – recall that a branch in the abstract tree is formed by unification of a subgoal with clauses which are *alternatives* at the concrete level). Now, then, we must consider in detail what this *branch* option really does.

7.3.3 Dealing with Recursion by Induction

As was mentioned before, the point of this work is to apply existing techniques in Proof Planning to the domain of Prolog execution. In particular, expertise exists in dealing with inductive proof. The point at which the inductive ideas become most obviously useful is when we attempt to deal with recursive predicates whose definition provides a base case and a step case for structural induction. One easily detectable such predicate is perm/2. When we unfold the two clauses of perm with our subgoal in the *branch* above, we are left with two new subgoals (which must both be "proven" by our execution plan), thus:

$$\vdash \quad \mathrm{ord}(\ [\,]\ ). \tag{3}$$

$$\begin{aligned}\vdash \quad & \mathrm{del}(\ \mathit{var}\ v1 : \mathit{number},\ [\mathit{gnd}\ u1 : \mathit{number} \mid \mathit{gnd}\ k1 : \mathit{number\ list}],\ \mathit{var}\ w1 : \mathit{number\ list}\ ),\\ & \mathrm{perm}(\ \mathit{var}\ w1 : \mathit{number\ list},\ \mathit{var}\ l1 : \mathit{number\ list}\ ),\\ & \mathrm{ord}(\ [\mathit{var}\ v1 : \mathit{number} \mid \mathit{var}\ l1 : \mathit{number\ list}]\ ).\end{aligned} \tag{4}$$

The first sub-tree, executing the isolated ord/1 call in (3), is a trivial *unfold*, leaving us with no subgoals left to prove – which means success. The more interesting part is dealing with the second branch, (4). In the trace tree reproduced in figure 2, this first call, of the del/3 predicate, is "fully executed", and, as mentioned before, there is no clear motivation for this. However, in the proof theoretic framework, we can explain the situation in a more motivated way. First, though, let us consider the full extent of the main program loop, given by the recursion on the perm/2 predicate.

It happens that this *branch* on perm/2 mentioned above is insufficient (as the foundation for a terminating loop) to describe the full behaviour of the program: the same branch cannot deal with the input of both empty lists and singleton lists. This fact is discovered during planning (because plans involving the contrary assumption are unsuccessful), and need not be known in advance. The induction hypothesis suggested by this branch (see next section) is not the correct one to achieve the inductive proof required. Therefore, after the complete del/3 execution, mentioned above, we need to perform another *branch* on perm/2. This takes us from (5) to (6).

$$\begin{aligned} \vdash \quad & \mathit{perm}(\ \mathit{var}\ w1 : \mathit{number\ list},\ \mathit{var}\ l1 : \mathit{number\ list}\ ), \\ & \mathit{ord}(\ [\mathit{gnd}\ v1 : \mathit{number} \mid \mathit{var}\ l1 : \mathit{number\ list}]\ ). \end{aligned} \tag{5}$$

$$\vdash \quad \mathit{ord}(\ [\mathit{gnd}\ v1 : \mathit{number}]\ ). \tag{6}$$

$$\begin{aligned} \vdash \quad & \mathit{del}(\ \mathit{var}\ v2 : \mathit{number},\ [\mathit{gnd}\ u2 : \mathit{number} \mid \mathit{gnd}\ k2 : \mathit{number\ list}],\ \mathit{var}\ w2 : \mathit{number\ list}\ ), \\ & \mathit{perm}(\ \mathit{var}\ w2 : \mathit{number\ list},\ \mathit{var}\ l2 : \mathit{number\ list}\ ), \\ & \mathit{ord}(\ [\mathit{gnd}\ v1 : \mathit{number} \mid \mathit{var}\ v2 : \mathit{number} \mid \mathit{var}\ l2 : \mathit{number\ list}]\ ). \end{aligned} \tag{7}$$

Now, since we are trying to find an inductive "proof" for this execution, we want to find a justified means for doing so. We have, here, a branch which will enable us to cover the type of lists (w2 being the induction term, and being instantiated to empty and non-empty lists respectively in the two branches), so one possible approach might be induction on that type. To perform induction, we need an induction hypothesis (added to our existing hypothesis list) – one such can be generated by applying the list destructor function to w1, to give [_|var w2:number list], like this:

$$\begin{aligned} & \mathit{perm}(\ \mathit{var}\ w2 : \mathit{number\ list},\ \mathit{var}\ l1 : \mathit{number\ list}\ ), \\ & \mathit{ord}(\ [\mathit{gnd}\ v1 : \mathit{number} \mid \mathit{var}\ l1 : \mathit{number\ list}]\ ). \end{aligned} \tag{8}$$

We can proceed from here with our execution, and, when we reach the final node of figure 2, we will find that it matches exactly, except for renaming of variables, with the induction hypothesis. The renaming is explained neatly as part of the induction, by the definition of perm/2: whenever the list destructor is applied to perm/2's first argument, it is also applied to the second. We can easily see, then, that the recursion in perm/2 is fully captured here. So now we have an inductive "proof", subject to the considerations about well-typedness mentioned before.

7.3.4 Dealing with Other Kinds of Recursion

Let us return to the del/3 execution, above, and consider why, as I suggested above, it is different from the simple inductive form explained in the last section. The applicable method leading to the correct proof at this point is the *branch*ing of the del/3 subgoal, (9), by unfolding with the two clauses of its definition.

$$\begin{aligned} \vdash \quad & \mathit{del}(\ \mathit{var}\ v1 : \mathit{number},\ [\mathit{gnd}\ u1 : \mathit{number} \mid \mathit{gnd}\ k1 : \mathit{number\ list}],\ \mathit{var}\ w1 : \mathit{number\ list}\ ), \\ & \mathit{perm}(\ \mathit{var}\ w1 : \mathit{number\ list},\ \mathit{var}\ l1 : \mathit{number\ list}\ ), \\ & \mathit{ord}(\ [\mathit{var}\ v1 : \mathit{number} \mid \mathit{var}\ l1 : \mathit{number\ list}]\ ). \end{aligned} \tag{9}$$

This yields the two subgoals, (10) and (11):

$$\vdash \quad \mathrm{perm}(\ \mathrm{gnd\,k1} : \mathrm{number\ list},\ \mathit{var}\ \mathrm{l1} : \mathrm{number\ list}\ ),$$
$$\mathrm{ord}(\ [\mathrm{gnd\,u1} : \mathrm{number} \mid \mathit{var}\ \mathrm{l1} : \mathrm{number\ list}]\ ). \tag{10}$$
$$\vdash \quad \mathrm{del}(\ \mathit{var}\ \mathrm{v1} : \mathrm{number},\ \mathrm{gnd\,k1} : \mathrm{number\ list},\ \mathit{var}\ \mathrm{y1} : \mathrm{number\ list}\ ),$$
$$\mathrm{perm}(\ [\mathrm{gnd\,u1} : \mathrm{number} \mid \mathit{var}\ \mathrm{w1} : \mathrm{number\ list}],\ \mathit{var}\ \mathrm{l1} : \mathrm{number\ list}\ ),$$
$$\mathrm{ord}(\ [\mathit{var}\ \mathrm{v1} : \mathrm{number\ list} \mid \mathit{var}\ \mathrm{l1} : \mathrm{number\ list}]\ ). \tag{11}$$

Now, observe that the relationship between the second new node and the parent node, is similar to that in the perm/2 case: a destructor function has been applied to the second argument, changing it from a list of the form [_|T] to simply T. The same destructor has been less obviously applied to the third argument. Now, because we have two clauses in the definition of del/3, one of which contains recursion, and one of which does not, we can say that we have something similar to a base case and a step case in induction.

However, there is a difference, here, which is that the two cases of the branch are not mutually exclusive. Therefore, we have something which looks, maybe, like an induction over non-empty lists, but with an infinite set of base cases. Nevertheless, the structural behaviour thus described can (easily, in this case) be shown to terminate the induction for any given member of that datatype. Therefore, the initial call of del/3, above, (after the application of the list destructor) may be viewed as something like an induction hypothesis on the assumption (which must be justified by further "proof") that the first option generated by this application of *branch* is executable. Therefore, we require that application of the *branch* method and rule causes introduction of a new hypothesis in our "proof", corresponding with this induction hypothesis.

The difference between the del/3 analysis and the straightforward induction in the perm/2 analysis is mirrored by the non-determinism occurring in the actual execution of the del/3 predicate. This is one focus for interesting further work in this project.

The analysis presented above brushes under the carpet the fact that an implicit type inference has happened here: by using the first clause of del/3 as our base case with the second argument as the induction term, we *assume* that the datatype over which we are inducing is that of non-empty lists, and not just lists. As was mentioned before, in a genuinely proof-theoretic framework we need to justify this assumption – which will be a goal of future work. For now, though, let us just note the requirement, and assume (not unreasonably) that we can fulfill it – at least for well-behaved cases.

7.4 Finding and Applying Inductive Executions

Now, armed with the kind of inductive view of recursive loop checking outlined above, we are able to proceed in the same way throughout our execution tree. Using the iterative deepening planner, we arrive first[5] at the same tree as that given by the Compiling Control approach. The final node (marked as a loop by the broken arrow in figure 2) is no longer viewed as a *renaming* of the state at which the loop begins, but as a goal which is proved from the existence of a matching induction hypothesis.

The output of the CLAM planner is of the form shown in figure 6 – uninteresting details are omitted. The planner takes 6.4 cpu minutes on a Sun 3/60 to reach this plan by brute force search with the simple pruning optimisation described above, including showing that there is no shorter proof plan for the same theorem. The meanings of the method specifications are as follows. The arguments to *unfold* are a program clause number and a subgoal number, the latter being unfolded in the usual way according to the definition given by the former. *Branch* takes

[5] Though there are clearly equivalent executions, involving more steps in the main inductive loop.

```
Planning to depth of 8 with optimisation

  unfold(1,1) then
    branch([2],[3],1,v416,F1) then
      [unfold(6,1)
       branch([4],[5],1,v436,F2) then
         [branch([2],[3],1,v448,F3) then
            [unfold(7,1)
             branch([4],[5],1,v468,F4) then
               [unfold(8,2) then
                  unfold(9,2) then
                    loop(v448,wave(...))
                loop(v468,wave(...))
              ]
            ]
          loop(v436,wave(...))
         ]
      ]
```

Figure 6: The Prolog-CLAM Plan for Slowsort

a list of (possible) base case clause numbers, a list of (possible) recursive case clause numbers[6], a subgoal number, a label, which is used to mark the "induction hypothesis" introduced by the branch, and finally an "induction form" which describes the induction scheme used in the execution. When the *branch* method is called, this induction form is an uninstantiated variable: the application of the corresponding *loop* method fills in the form later, in a degenerate version of middle out proof planning, detailed below. *Loop* takes as argument a label – that of the induction hypothesis justifying the loop – and an inductive form specifying the scheme to be used. *Succeed* merely terminates a successful (sub-)tree.

The optimisation referred to in the heading of the output relies on the independence of the sub-trees at a branch node. This independence means that if a first sub-tree executes to success, but a subsequent one fails, there is no point in retrying the first sub-tree, because the subsequent one is not affected by such a resatisfaction. The proof search space can be considerably reduced by this optimisation, even when using a planner (such as our chosen iterative deepening planner) which uses depth-first search as its sub-strategy.

Finally, the *wave(...)* arguments to the loop methods in the figure describe transformations taking place between the arguments of the induction hypothesis, in the form of ordered pairs of "before" and "after" arguments. These transformations can guide us to the correct inductive form for the proof, and, in the process of planning the execution, will become bound to the "inductive form" variables in the corresponding *branch* methods. This is the "middle-out" aspect of the planning. (Note that the association of this information with the *branch* methods is important because of the stepwise proof-time construction of the extract term. All the information necessary must be available to Oyster when it constructs the branch.)

[6] Currently naïvely predicted by search for occurrences of calls to a predicate in the body of one of its own clauses.

7.5 Generalising Inductive Behaviour: Wave Rules

The outline above has a distinct flavour of the *ad hoc*. In order to generalise the idea to more forms of recursion control, we need to apply the ideas of *rippling out* and the associated *wave rules* (as introduced in Section 4). Let us now look closely at the behaviour involved, and try to generalise it.

Consider again the two sets of subgoals at the "beginning" and "end" of the execution loop of del/3. The initial state is:

$$\begin{aligned} \vdash \quad & del(\ var\, v1 : number,\ [gnd\, u1 : number \mid gnd\, k1 : number\ list],\ var\, w1 : number\ list\ ), \\ & perm(\ var\, w1 : number\ list,\ var\, l1 : number\ list\ ), \\ & ord(\ [var\, v1 : number \mid var\, l1 : number\ list]\ ). \end{aligned} \tag{12}$$

and the final state is:

$$\begin{aligned} \vdash \quad & del(\ var\, v1 : number,\ gnd\, k1 : number\ list,\ var\, y1 : number\ list\ ), \\ & perm(\ [gnd\, u1 : number \mid var\, y1 : number\ list],\ var\, l1 : number\ list\ ), \\ & ord(\ [var\, v1 : number\ list \mid var\, l1 : number\ list]\ ). \end{aligned} \tag{13}$$

Now, examine the difference between the del/3 subgoals in (12) and (13). We have one argument (the first) invariant, one (the second) clearly showing the effects of a list destructor application, and one (the third) transformed by alphabetic renaming, which we can see on closer inspection to be another the result of a less obvious list destruction. As it happens, we have an obvious corresponding *con*struction in the first argument of perm/2. Because of the nature of the unfolding transformation in this simple recursive case, only the instantiations in the two goals will have changed, and functors must be invariant. The transformation has the form

$$D(X,S(Y),S(Z)),P(S(Z),A),O([B|A]) \rightarrow D(X,Y,Z),P(S(Z),A),O([B|A])$$

where S is (informally) a constructor function on lists.

Now, let us generalise this transformation in the following way. We separate the predicate(s) in which the recursion-limiting transformation occurs from the others, and cast the two (Main and Aux, below, respectively) as single predicates of higher arity. Then, we rewrite these two predicates as predicates of two arguments, these being lists of input and output variables, respectively. Then, we have

$$Main([S(Y)],[X,S(W)]),Aux([S(W),[B|A]],[A]) \rightarrow Main([Y],[X,W]),Aux([W,[B|A]],[A])$$

Generalising this further, we can remove the specific arguments, and build S into four functions, S_1, S_2, S_3, S_4, over lists of arguments:

$$Main(S_1(P),S_2(Q)),Aux(R,T) \rightarrow Main(P,Q),Aux(S_3(R),S_4(T))$$

Now, as might be expected, the generalisation describes not only this particular application of the recursive del/3 clause, but also that clause itself, called as in slowsort with the first and third arguments being output. The concrete clause is this:

$$del(\ A,\ [B|C],\ [B|D]\ ) \leftarrow del(\ A,\ C,\ D\ ).$$

so if we replace Main (in the generalisation) with "del", Aux with "true", Q with [A,D], P with [C], S_1 with $(\lambda x.[B|x])$, and S_2 with $(\lambda[x,y].[x,[B|y]])$, we confirm the generalisation for a trivial case – that of an isolated call to del/3.

The point of all this is that this generalisation constitutes a *wave rule*. Wave rules are used to rewrite an induction conclusion so that it matches an induction hypothesis. They do so by moving out of the way those subexpressions which would prevent the match succeeding. In general (by inspection), we see that the form of the wave rule is as given above, where S_i are functions related to S, but not in general all the same, and reading $\rightarrow$ as "becomes". Some of S_i are usually identity.

We are now in a position to specify generalisations about the behaviour of inductive "proofs" of recursive predicates for each of our datatypes. Since we are still within the planning part of the process, we are able (trivially) to instantiate the hitherto free "induction form" argument in each *branch* method to the form produced by the corresponding application of *loop*. Thus, when we complete our plan, as we apply each step in it, we have all the information encoded in our tree, and therefore all we need to generate an extract term representing the behaviour of the program under the (improved) execution rule.

7.6 Producing a New Execution Rule

It is not yet clear what is the "correct" way of specifying the extract term in the Prolog-Oyster system. (Recall that an extract term is a step-wise constructed algorithm, the steps being in 1–1 correspondence with the application of the rules in the plan to the Prolog-Oyster proof-development system.)

One likelihood is that we would want to represent the unifiers at each stage – and certainly, this, coupled with information about state changes (*viz* which nodes this step is from and to), is exactly equivalent to the information produced by the Leuven Compiling Control method, before the application of the state-collapsing optimisation mentioned before. Such an output is, of course, amenable to the same optimisation as the Compiling Control output, and also to ideas like the removal of redundant variables (ie entities whose value is at all times equivalent) which can make great savings in the time spent on unification.

Indeed, given the execution plan, the production of the appropriate (Leuven-style) specialised meta-interpreter is almost trivial – all the information required is contained explicitly in the plan.

Also, running under the standard computation rule, such a meta-interpreter admits the back-tracking behaviour which we would lose in the Martin-Löf functional style of extract term.

What is more, because of our proof-theoretic approach, we are guaranteed a successful execution for any query covered by the specification of the abstract goal (because of the completeness of inductive methods over types), which is not the case with the Compiling Control approach.

8 Conclusion: Advantages and Disadvantages of the Proof Theoretic Approach

In summary then, the Prolog-Oyster/CLAM system constitutes in general terms a rational reconstruction of the Compiling Control ideas. The main theoretical differences are the replacement (or, rather, implementation) of abstraction and loop checking from renaming by strict typing and the inductive view of recursive loops, respectively. The result of applying a proof development system like Oyster-CLAM to such an application is the complete automation of the development of improved execution rules; and the use of a constructive logic means that the production of an equivalent program is (therefore) automatic and fairly straightforward. However, for this gain in automation and formality, there is a price to be paid.

Even in simple examples, like slowsort, we run into the need to restrict the datatypes over which (sub-)programs work, as we perform our analysis. This is in principle easy for some cases (eg when the sub-type misses, say, the bottom two members of the type), but can be arbitrarily difficult for others – for example, a program working over lists of length less than N, where N is determined by user input. There is a precedent for this kind of mechanism, built into Oyster by Colin Phillips and reported in a forthcoming DREAM group publication, in the form of the

Acc type; this could in principle be introduced into Prolog-Oyster. The implications, though, in terms of proof complexity, are formidable.

Even so, I suggest that the inductive view is preferable to the less formal Compiling Control view (especially with the addition of such an Acc type), particularly given that Compiling Control's solution to this type-coverage problem is depth-bound abstraction, which merely defers the problem to run-time in a particularly arbitrary way.

In order to make Prolog-Oyster/CLAM a fully useful and formal system, the current type-inference system must be augmented with automatic proof of type completeness, and the whole must be cast into a proven proof theory. This will shed some light on the correct form of the correct form of extract term. In the current system, the quasi-extract term mechanism is not implemented, but it has been shown that such an implementation is trivial Prolog programming.

References

[Bruynooghe *et al* 89] M. Bruynooghe, D. de Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 135–162, 1989.

[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 Research Paper No. 349.

[Bundy *et al* 90] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 1990. In press. Earlier version available from Edinburgh as Research Paper No 413.

[Constable *et al* 86] R.L. Constable, S.F. Allen, H.M. Bromley, *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[De Schreye & Bruynooghe 88] D. De Schreye and M. Bruynooghe. The compilation of forward checking regimes through meta-interpretation and transformation. In J. Lloyd, editor, *Proceedings of the Meta88 Workshop*, pages 169–184, 1988.

[De Schreye & Bruynooghe 89] D. De Schreye and M. Bruynooghe. On the tranformation of logic programs with instantiation based computation rules. *Journal of Symbolic Computation*, (7):125–154, 1989.

[Horn 88] C. Horn. *The Nurprl Proof Development System.* Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[Madden 88] P. Madden. Automatic program optimization via the transformation of Nuprl synthesis proofs. In L. Clarke, editor, *Proceedings of the 1988 Alvey Technical Conference*, The Alvey Directorate, 1988.

[Madden 89] P. Madden. *The Specialization and Transformation of Constructive Existence Proofs.* Research paper 416, Dept. of Artificial Intelligence, Edinburgh, 1989. To appear in Proceedings of IJCAI-89.

[Miller & Nadathur 88] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*, MIT Press, 1988.

[Richardson 89] J.D. Richardson. *The Application of Proof Plans to Prolog Program Transformation.* Unpublished M.Sc. thesis, Dept of Artificial Intelligence, University of Edinburgh, 1989.

[van Harmelen 89] F. van Harmelen. *The CLAM Proof Planner, User Manual and Programmer Manual.* Technical Paper TP-4, Dept. of Artificial Intelligence, Edinburgh, 1989.

[Verschaetse *et al* 88] K. Verschaetse, D. de Schreye, and M. Bruynooghe. *Automatic Control Generation in Five Steps.* Technical Report CW 79, Department of Computer Science, Katholieke Universiteit Leuven, October 1988.