Incrementality and Logic Programming

Murray Shanahan
March 1988

Imperial College
Department of Computing,
180 Queen's Gate,
London SW7 2BZ.

Most of the work described in this paper was done at the University of Cambridge Computer Laboratory, whilst working on my PhD, supervised by William Clocksin.

1 Introduction

An RMS offers efficiency gains to an inference mechanism by facilitating *selectivity* of search and *incrementality* of movement in a logical structure. Selective (or "intelligent") backtracking mechanisms for Prolog interpreters have received some attention in the literature (Bruynooghe and Pereira [1], Matwin and Pietrzykowski [11], Shanahan [15]), but there has been less effort to match the incrementality of an RMS within a logic programming framework. This paper attempts to rectify this by describing an incremental Prolog-like theorem prover which records the structure of a search space using data dependencies, and which can incrementally assimilate changes to a database of clauses by propagating those changes through the recorded dependency structures.

2 Data Dependencies

Suppose that we are concerned to reason with with a set of logical formulae T. By *explicitly* representing the formulae in T, we are *implicitly* representing the structure of logical relationships which T defines. What we are doing when we reason with T is recovering components of this structure, and because it is only implicitly represented, this requires some computational effort. To minimise this effort, each component of logical structure thus recovered can be recorded, so that, as reasoning proceeds, more and more of what was previously only implicitly represented becomes explicitly represented. A Reason Maintenance System (Doyle [6], de Kleer [3], Reiter and de Kleer [12]) provides facilities for building and accessing just such a record of logical structure.

In particular, an RMS records a structure of *justifications* and *nogoods*, which are both kinds of *data dependency* (Charniak *et al.* [2]). If one formula is shown to be a logical consequence of a set of others then the RMS can be informed of this fact, which it records as a justification. If a set of formulae are shown to be inconsistent, then the RMS can be informed of this fact too, which it records as a nogood. Now, how can this information be exploited?

One important application is to *search*. Suppose that the problem solving task is to find some set of values which meet a given set of constraints. This can be considered as the problem of finding some consistent set of *assumptions* Δ such that $T + \Delta \vdash G$, where G describes the set of constraints and T describes the set of rules for the domain in question. The problem solver must efficiently search the space of possible Δ's. The information recorded in an RMS facilitates an efficient search in two ways — it offers *incrementality* and *selectivity*.

Recording justifications allows incrementality. The logical structure of some $T + \Delta$ can have much in common with that of some $T + \Delta'$. Rather than redundantly remaking inference steps common to both, recording justifications allows the problem solver immediate access to logical consequences it has already discovered. Suppose that the problem solver examines some Δ where $P \subset \Delta$, and determines that $T + P \vdash Q$ and therefore that $T + \Delta \vdash Q$. There is no need for it to duplicate

the work of making this discovery when it examines any Δ' where $P \subset \Delta'$. Recording that $T + P \vdash Q$ as a justification makes it possible to determine straight away that $T + \Delta' \vdash Q$. So, movement in the space of possible Δ's is incremental in the sense that the work done in examining one Δ is not wasted but is carried over to the examination of others.

Recording nogoods allows selectivity. The problem solver does not have to explicitly examine every possible Δ to completely explore the search space, and clearly the search is most efficient when the fewest Δ's are examined. Suppose that the problem solver examines some Δ where $P \subset \Delta$, and determines that $T + P \vdash \neg G$, and therefore that $T + \Delta \vdash \neg G$. Recording and exploiting a nogood to the effect that P is inconsistent with G prevents the problem solver from exploring any Δ' such that $P \subset \Delta'$. So exploration of the space of possible Δ's is selective, in the sense that the problem solver will not examine a new Δ which it has already shown not to contribute to a solution.

A second important application is to *databases*. Suppose we have a database which evolves gradually over time. By recording data dependencies, each update can be assimilated incrementally, and the logical consequences of the information in the modified database do not need to be recomputed from scratch. Not only does this enable the database to answer subsequent queries more quickly, but if an update renders the database inconsistent, data dependencies allow the source of the inconsistency to be identified.

Data dependencies have other uses too. Another application of incrementality is to the computational aspect of the *frame problem* (see Shanahan [15]). A problem solver is more efficient if it incrementally transforms an already computed representation of a situation into the representation of a temporally nearby situation than if it recomputes it from scratch.

A Reason Maintenance System can also provide a framework for *default reasoning*. With a system like Doyle's ([6]), this is due to the form of the justifications it records. In de Kleer's system ([3]), default inferences are not recorded and the responsibility for default reasoning lies with the problem solver. The next section describes the work of Doyle and de Kleer in more detail.

3 Reason Maintenance

Influenced by the work of Stallman and Sussman ([16]) on dependency directed backtracking, Doyle developed a Reason Maintenance System ([6]), which automatically maintains consistency in a database of formulae. Associated with each formula in the database is a corresponding *node* in the RMS, and these nodes are connected together in a web of data dependencies. Each node has a status of either *in* or *out*, and a *justification*. If its justification is *valid*, then a node is in, and otherwise it is out. In Doyle's system, a justification has the form $(< inlist >< outlist >)$, and is valid if all the nodes in its inlist are in and all the nodes in its outlist are out. The support for an in node must be *well-founded*, that is, circular, self-supporting networks of justifications are not valid.

The form of the justifications permits *non-monotonic* inference. In other words, the coming in of a node that was previously out can result in the going out of a node that was previously in. This facilitates the creation of revisable *assumptions* as well as non-revisable *premises*. For example, an assumption P and a premise Q would be represented as follows

N_1	P	$(() \ (N_2))$
N_2	$\neg P$	
N_3	Q	$(() \ ())$

Note that separate nodes are used to represent P and $\neg P$. Thus, P is assumed unless there is a valid justification for $\neg P$.

The RMS performs two basic operations on the web of dependencies: *reason maintenance* and *dependency directed backtracking*. Reason maintenance is invoked whenever the problem solver adds a new node or justification, and it ascertains which nodes are in and which are out. Dependency directed backtracking is invoked to resolve contradictions by backtracking through the thread of justification for the contradictory node in search of an assumption which it can retract in order to restore consistency. The propositional content of a node is transparent to the RMS, which can only see the structure of dependencies. It is the responsibility of the problem solver to perform inference and to indicate contradictions.

When a node is identified as a contradiction and dependency directed backtracking is invoked, the set of *maximal* assumptions supporting the contradiction is computed, from which one assumption, the *culprit*, is chosen and retracted, thus restoring consistency. An assumption is maximal if it does not itself support another justification which supports the contradiction. Before the culprit is chosen, a new node, called a *nogood* node, is created to record the contradiction. The culprit is then chosen and is retracted by justifying one of the nodes which appears in its outlist. This new justification must record the rest of the inconsistent assumptions, in case the chosen culprit turns out to be the wrong one, and backtracking has to try one of the alternatives.

De Kleer ([3], [4]) introduced the idea of an *Assumption-Based* Reason Maintenance System. In Doyle's RMS, only one labelling of nodes with in or out is considered at any one time. The problem solver can only focus on a single set of assumptions and their consequences. In de Kleer's system, a node is not labelled as either in or out, but is labelled with the set of *environments* in which that node is in. An environment is a set of assumptions. The problem solver can explore many possibilities at once, and can compare solutions and potential solutions to problems. Furthermore, the resulting mechanism obviates the need for backtracking. The discovery of an inconsistency does not result in dependency directed backtracking to restore consistency, but results in a reduction of the sets of environments labelling some of the nodes.

The Assumption-Based RMS offers an improvement over a conventional RMS for search problems where all or many solutions are required. When only one or a few solutions are required, the conventional RMS is more efficient. Acknowledging

this and other deficiencies, de Kleer ([5]) presents a hybrid algorithm which he calls assumption-based dependency directed backtracking.

The next section introduces logic programming and surveys a number of techniques within logic programming which exploit similar ideas to those used in an RMS, and which offer the same functionality.

4 Logic Programming

The Horn clause subset of the predicate calculus is used throughout the rest of this paper. It is defined as follows.

A *term* is either a constant or a variable or has the form $f(T_1...T_n)$, where f is an n-ary function and $T_1...T_n$ are terms. A *literal* or *goal* has the form $p(T_1...T_n)$, where p is an n-ary predicate and $T_1...T_n$ are terms. A *definite clause* has the form $L_0 \leftarrow L_1, L_2, ...L_n$, where $L_0...L_n$ are literals. If $n = 0$ then a definite clause may be called a *fact*. If $n > 0$ then it may be called a *rule*. A *goal clause* has the form $\leftarrow L_1...L_n$, where $L_1...L_n$ are literals. The goal clause $\leftarrow$ is called the *empty clause*. A *Horn clause* is a definite clause or a goal clause.

Variables start with an uppercase letter and may be subscripted. Constants, function names and predicate names start with a lowercase letter and may be subscripted. Meta-variables will be denoted by an uppercase letter and may be subscripted.

A Prolog interpreter is a top-down Horn clause theorem prover which employs SLD resolution (Kowalski and Kuehner [10]). Presented with a goal clause G_0 and a set of definite clauses P, it searches for refutations of G_0 in the form of a sequence of goal clauses $G_0, G_1...G_n$ where G_n is the empty clause and each G_{i+1} is obtained by resolving G_i with some clause in P whose head unifies with the leftmost literal in G_i. Each such refutation generates a corresponding set of variable bindings, called an answer substitution. Since there may be many clauses in P whose heads will unify with the leftmost literal of any given G_i, the theorem prover has to search a space of possible refutations, and for each refutation discovered it outputs the corresponding answer substitution. Most extant Prolog interpreters use backtracking to effect a depth-first search, choosing clauses from P in top to bottom order.

The search performed by a Prolog interpreter can be thought of as a search for Δ's such that $T + \Delta \vdash G$, where T is a set of definite clauses augmented with an equality theory, G is the goal clause and Δ is a set of equalities (*i.e.* answer substitutions). By widening the notion of what can be contained in a Δ to include constraints (Jaffar and Lassez [9]), by incorporating integrity checking (Sadri and Kowalski [13]), and by allowing abduction (Eshghi and Kowalski [7]), a very powerful problem solving paradigm is obtained. But can a mechanism be built for this paradigm which exploits dependency information as effectively as it is exploited in an RMS?

The sets of assumptions manipulated by an RMS are a particular sort of logically structured entity. Their logical structure is represented by the web of data dependencies between assumptions, premises and consequences. An RMS is in essence a system for managing data dependencies. Data dependencies facilitate easy move-

ment in a space of logically structured entities, by allowing the efficient transformation of one such entity into another, exploiting the similarities between the two so as to prevent the redundant re-making of inference steps which are common to them both. This is the sense in which an RMS is incremental.

Data dependencies can also be used to guide search, to permit selectivity in backtracking. Using an RMS, a forward-reasoning problem solver conducts a search by repeatedly adding to the web of justifications until it converges on a set of assumptions Δ which is consistent with the required constraints for a solution, that is, such that $T + \Delta \vdash G$ for some set of rules T and constraints G. Because it records and exploits justifications and nogoods, a problem solver using an RMS exhibits much better backtracking behaviour than chronological backtracking.

Although it is a backward- not forward-reasoning mechanism, the backtracking mechanism of a Prolog interpreter can also be much improved by recording and exploiting the same kind of dependency information (see Bruynooghe and Pereira [1], Matwin and Pietrzykowski [11], Shanahan [15]).

A Prolog interpreter searches a space of possible refutations. A possible refutation is, in a sense, a logically structured entity: it is a sequence of goal clauses, each derived from its predecessor by a resolution step. The search performed by a Prolog interpreter is, in a sense, incremental: it searches a tree of possible refutations, and when it moves from one node to another on the same branch, it does not, of course, have to perform all the resolution steps from the root to the new node all over again.

The need for incrementality does not arise only in search, but also for database applications (and other things; see Section 2). The problem solving paradigm outlined above can be incorporated into a *deductive database* (Gallaire *et al* [8]). But as it stands, the Prolog mechanism provides no facility for incrementally manipulating sets of clauses and their consequences, so that each update to the database demands the recomputation of answers to all queries from scratch. Similarly, each time the database is updated, integrity constraints have to be checked from scratch, and this can also involve redundant recomputation. In the next section I present a Prolog-like theorem prover which is incremental in this sense.

5 An Incremental Theorem Prover

Having fully explored the search space, a non-incremental theorem prover such as Prolog throws away all record of how each substitution was computed; what sequences of resolutions were tried, which were successful and which were not. Additions and deletions of clauses are then straightforward database operations, but every time refutations have to be found for a goal clause, the search space has to be explored from scratch. Now, suppose that the use of the theorem prover is characterised by the repeated presentation of the same set of goal clauses for a slightly changed set of definite clauses. The search spaces explored for each slightly modified set of definite clauses are then likely to overlap considerably. Under these circumstances it is economical to employ an *incremental* theorem prover which maintains *dependency structures* showing how each set of answer substitutions is obtained.

Then, if a small change takes place in the set of definite clauses, the consequences of this change are propagated through the dependency structures to the set of answer substitutions. It is not necessary to regenerate the answer substitutions from scratch. The burden of computation is then shifted to the incremental modification of these dependency structures when the set of definite clauses is modified, reducing the search for answer substitutions to a simple lookup. Clearly, for this to be a viable proposal, the resulting savings must outweigh the overheads of recording the dependency structures and propagating the consequences of change. I will outline the construction of an incremental top-down resolution Horn clause theorem prover, in which the maintenance of dependency structures incurs acceptable overheads.

The mechanism I will describe performs the following operations, given a goal clause G_0, a set of definite clauses P and corresponding dependency structures S: add a clause p to P and update S, delete a clause p from P and update S, and output corresponding substitutions for all refutations of G_0. Of course, many other sets of operations are possible. For instance, facilities might be included for modifying parts of clauses, and this would permit a finer grain of incremental modification of S. The techniques described extend naturally to the incremental modification of the dependency structures for a set of goal clauses. It is also possible to incorporate negation-as-failure, but I will not discuss this problem here, nor will I discuss the problem of dealing with infinite proof trees.

6 Recording Dependencies

Consider the search tree T for a goal clause G_0 and a set of definite clauses P, and suppose that exploring this tree has generated a set of answer substitutions B. Let P' be the same set of clauses as P but with one addition. Then, the search tree T' for G_0 and P' will be the same as T but with a number of extra branches grafted on, and the set of answer substitutions B' will be a superset of B. Alternatively, if P' is the same set of clauses as P but with one deletion, then the search tree T' for G_0 with P' will be the same as T but with a number of branches pruned away, and the set of answer substitutions B' will be a subset of B. This analysis would be more complicated if it took account of negation-as-failure or any form of non-monotonicity, since clause deletions could then add to the search tree and clause additions could subtract from it.

Figure 2 shows the growth which results in the tree of Figure 1 from the addition of a new clause. The predicate $stack(X, Y, Z)$ represents that the blocks X, Y and Z are piled on top of one another and form a stack. In Figure 1 the tallest pile of blocks is only two high, so there are no stacks, but in Figure 2 a new block has been added, and a stack has been formed. Deleting the new clause again causes the tree to shrink back to that of Figure 1.

For each operation, if a record of B and T is maintained, then the search space for G_0 and P' can be explored by propagating the consequences of changing P to P' through T, thus obtaining T', and then propagating the consequences of changing T to T' through B, thus obtaining B'. In addition to preserving B and T, it

Figure 1: An Example Search Tree

Figure 2: An Enlarged Search Tree

is necessary to record which substitutions in B depend on which branches in T, and which branches of T depend on which clauses in P; respectively the *answer dependencies* and the *clause dependencies*. Then, the deletion of a clause from P must bring about the removal of those branches in T which depend on it, giving T', and each deletion of a branch in T must bring about the deletion of those substitutions in B which depend on it, giving B'. Also, it is necessary to record the *predicate dependencies*; for each predicate in P, the set of nodes in T at which backtracking took place because of the exhaustion of clauses for that predicate. Then, the addition of a clause for a predicate must bring about the restoration of the state of computation at each node at which clauses for that predicate were exhausted. For each such restored state, search is resumed, thus generating new branches to be grafted onto T, giving T', and possibly producing new substitutions to be added to B, giving B'.

The three dependency structures mentioned above must be maintained with respect to this tree; the answer dependencies, the clause dependencies and the predicate dependencies. For each leaf in the tree, a record is kept of whether the path from the root to that leaf constitutes a refutation, and if so, the corresponding answer substitution in B is indicated. For each clause in P, a list is kept of those nodes in the tree which point to that clause. Finally, for each predicate in P, a list is maintained of those nodes in the tree whose childrens' root nodes all point to clauses for that predicate. Using the same example as in Figures 1 and 2, Figures 3 and 4 illustrate the three types of dependency. Although not shown, all three dependency structures must, of course, be maintained for both trees.

The cost of building the dependency structures during search is two list insertions, of the kind that do not require search (see below), for each resolution step performed. The time savings are obtained at the expense of a storage overhead which will be directly proportional to the complexity of the search space.

The deletion of a clause C proceeds as follows. For each node N in the clause dependencies for C, the tree from N downwards is removed. The removal of a node requires that all references to that node are deleted from the dependency structures. So that this does not involve search, the clause and predicate dependencies can be threaded through the tree in doubly linked circular lists with dummy first elements, and the removal of a node is then preceded by the deletion of its entries in those lists. Whenever a leaf is reached, if that leaf is at the end of a refutation then the corresponding answer substitution is deleted from B. So, the cost of deleting a clause is directly proportional to the total amount of search subspace whose existence depends on it. If this is a small proportion of the overall search space then the savings resulting from adopting the incremental approach are correspondingly large. If it is a large proportion of the overall search space then the savings will be negligible and the extra cost will be of the same order as the cost of the original search.

The addition of a clause C proceeds like this. For each node N in the predicate dependencies of C, the state of computation at N is restored and the search is resumed, causing the new branches to grow from N, until an area of search space is reached which has already been explored (*i.e.* until the theorem prover backtracks past N).

Figure 3: An Example Search Space and its Dependency Structures

Figure 4: An Enlarged Search Space and its Dependency Structures

There are two ways that the state of computation at a given node in the search tree can be restored. The first incurs a greater time overhead and a smaller storage overhead. The only information recorded about the structure of the search is held in a tree whose shape is isomorphic to that of the search tree, but each of its nodes contains only a record of the clause its parent was resolved with to obtain that node. Then, the state represented by a node is restored by tracing back from that node to the root of the tree, forming a list L of the nodes on that path (in root to node order), and, starting with the goal clause G_0, generating the sequence of goal clauses $G_1 ... G_n$ by resolving each G_i with the $(i+1)^{\text{th}}$ member of L to obtain G_{i+1}.

Instead of reconstructing the state, further time savings can be made, at the expense of further storage overheads, by recording the whole structure of the search tree, including variable bindings. The state of computation at a given node in the tree is restored simply by a few pointer assignments. With this method, it must be possible to recover the pending goals at a given node, along with the corresponding bindings for their variables at that node, independently of any bindings that may have been made subsequently by any of the node's children. In a normal Prolog interpreter, a record is kept of the bindings made by each resolution step, and these bindings are undone on backtracking. But restoring the state of computation here is not the same as backtracking. A complete record of *every* node in the search tree must be maintained, so when a variable is bound by a resolution step, it must be *copied* into the corresponding new node of the search tree.

Whichever method is used, since all the new search subspaces explored have to be explored anyway, the only overhead of adopting the incremental approach to clause addition is the initial cost of storing the dependency structures, which is negligible. As with deletion, the savings obtained will depend on the proportion of newly explored search space to overall search space.

7 Refinements of the Mechanism

The incremental mechanism described so far is capable of making savings when the effects of additions to and deletions from the set of definite clauses are confined to the outermost parts of the tree, near the leaves, or when they are confined to only a few branches and the tree is wide. The mechanism will also prove useful for clause *replacements* (the deletion of a clause followed by the addition of a clause for the same predicate). Replacements are an important kind of modification. For instance, a change in location of a block in the Blocks World of the previous section is represented by the deletion of one *on* clause and the addition of another. It is often the case that the effects of a replacement are confined to a region near the root of the tree, leaving the peripheral foliage untouched. A finer grain of incrementality would be obtained if the mechanism avoided duplicating the work done below the affected area.

A complete solution to this problem is very difficult and is beyond the scope of this paper. The mechanism must remove those parts of the tree that are dependent on the replaced clause whilst saving the branches below. New nodes are grown

to replace the removed sections, using the new clause, and the saved branches are grafted back onto each new section that did not lead to a failure. As a result of growing a new section, some variables may change their bindings, and the consequences of these changes must be propagated through the rest of the tree. This can involve a similar pruning, growing and grafting process to that already described, since some clauses that failed to match before will match now, whilst others that did match before will fail to now. Again, sections of the tree will have to be lifted out and replaced, but this time a new section can have more offshoots than the one it replaces, so that when all the available saved branches have been grafted on, new ones have to be grown for any offshoots that are still incomplete.

Another refinement of the basic mechanism would be to make incrementality *lazy*. Suppose that a cache of search trees is recorded for the last n goal clauses for which the incremental mechanism was asked to find refutations. There is no point in expending effort propagating the consequences of subsequent modifications through all n search trees if only a few of them are ever going to be consulted again. Accordingly, it might be better to queue up the modifications to be propagated through a search tree until that tree is required again. If the queue becomes too long, it will no longer be any more economical to propagate the modifications through the stored tree than to start from scratch, and the tree can be discarded.

Finally, it may be desirable to use Prolog as a meta-level problem solver as well as for an object-level representation formalism. Built-in predicates *add* and *demo* could be provided which, respectively add/delete clauses to a theory and demonstrate the consequences of a theory, both using the incremental mechanism. The following extension to the mechanism may then be necessary. Consider the solution of a goal $add(T1, C, T2)$. It would be inefficient to keep entirely distinct copies of $T1$ and $T2$, since then the expense of copying $T1$ when a clause is added would obviate the advantages of the incremental approach. The same problem arises for deletion. Rather, what is required is a single structure which represents both theories. This could be obtained by labelling some of the nodes in the tree with the *contexts* (sets of axioms) in which those nodes (and their children) are to be considered part of the tree (de Kleer [3]; see Section 1). Detailed investigation of this and the other refinements is a subject for further research.

References

1. Bruynooghe M. and Periera L.M., Deduction Revision by Intelligent Backtracking, in *Implementations of Prolog*, ed Campbell J.A., Ellis Horwood (1984), p194.

2. Charniak E., Riesbeck C.K., McDermott D.V. and Meehan J.R., Artificial Intelligence Programming, Lawrence Erlbaum (1987).

3. de Kleer J., Choices Without Backtracking, *Proceedings American Association for Artificial Intelligence Conference 1984*, p79.

4. de Kleer J., An Assumption-Based TMS, *Artificial Intelligence*, vol 28 (1986), p127.

5. de Kleer J., Back to Backtracking: Controlling the ATMS, *Proceedings American Association for Artificial Intelligence Conference 1986*, p910.

6. Doyle J., A Truth Maintenance System, *Artificial Intelligence*, vol 12 (1979), p231.

7. Eshghi K. and Kowalski R.A., Abduction through Deduction, Technical Report, Imperial College, London (1988).

8. Gallaire H., Minker J. and Nicolas J-M., Logic and Databases: A Deductive Approach, *Computing Surveys*, vol 16 (1984), no 2, p153.

9. Jaffar J. and Lassez J-L., Constraint Logic Programming, *Proceedings 14th ACM POPL Conference 1987*.

10. Kowalski R.A. and Kuehner D., Linear Resolution with Selection Function, *Artificial Intelligence*, vol 2 (1971), p227.

11. Matwin S. and Pietrzykowski T., Intelligent Backtracking in Plan-Based Deduction, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol 7 (1985), no 6, p682.

12. Reiter R. and de Kleer J., Foundations of Assumption-Based Truth Maintenance Systems, *Proceedings American Association for Artificial Intelligence Conference 1987*, p183.

13. Sadri F. and Kowalski R.A., A Theorem-Proving Approach to Database Integrity, in *Foundations of Deductive Databases and Logic Programming*, ed Minker J., Morgan Kaufmann (1988).

14. Shanahan M.P., An Incremental Theorem Prover, *Proceedings International Joint Conference on Artificial Intelligence 1987*, p987.

15. Shanahan M.P., Exploiting Dependencies in Search and Inference Mechanisms, PhD Thesis, University of Cambridge Computer Laboratory (1987).

16. Stallman R.M. and Sussman G.J., Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis, *Artificial Intelligence*, vol 9 (1977), p135.