The design of provably good, distribution-sensitive speedup-learning algorithms

Devika Subramanian*
Computer Science Department
Cornell University
Ithaca, NY 14853

February 18, 2002

Abstract

We propose an approach to the design and analysis of provably good, distribution-sensitive, speedup learning algorithms. First, we formulate a class of speedup learners that improve efficiency by eliminating irrelevant computation at the base, as well as at the meta-level. A special representation called proof grammars is introduced, which allows for the efficient detection and elimination of a special form of meta-level irrelevance: repeated rule-choice computation in declarative theories. Proof grammars are used to reconstruct new theories from the original theory using repair procedures built from the operations of rule deletion and rule ordering, antecedent deletion and ordering, addition of rule compositions, and addition of substitution instances of rules. For this class of speedup learners, we define policies for generating new theories that are, with very high probability, more efficient than the original theory with respect to a stationary but unknown query distribution. We experimentally verify our results in the domains of combinatorial circuit synthesis and planning.

Keywords: Explanation-Based Learning, Utility Analysis, Recursive Domain Theories
Contact: Devika Subramanian, 607-255-9189, devika@cs.cornell.edu

*This research is supported by NSF grant IRI-8902721.

1 Introduction

Our objective is to build algorithms that improve performance of problem solvers by learning from experience. In this paper, we formulate speedup learning as a class of on-line optimization problems that eliminate wasted computation by exploiting regularities in the environment. We introduce REBL, a family of algorithms that optimize problem solving in theories formulated as definite clauses. This class of solvers encompasses standard formulations of planning and scheduling problems. An example is the on-line improvement of the performance of an elevator scheduler by learning properties of call patterns over time in its operating environment.

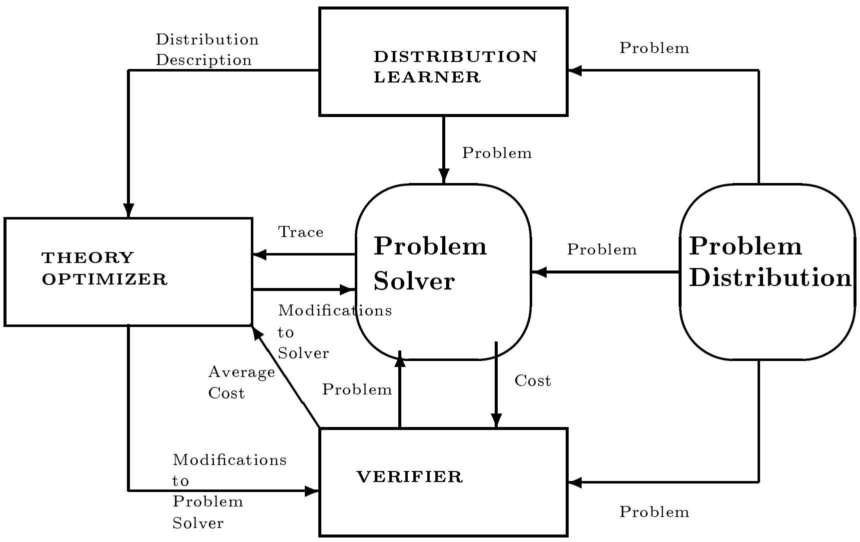


Figure 1: Architecture of Speedup Learners

The architecture of a speedup learner is sketched in Figure 1. The dimensions that characterize REBL algorithms are:

1. What properties of the problem distribution are learned?

   Learning distribution properties relevant to efficiency improvement is a domain-specific inductive task. In this paper, we use a simple multinomial estimation technique to learn equivalence classes of problems and map it to a set of special-purpose solvers.

2. What range of modifications to the solver are considered by the algorithm?

   The key principle behind REBL is to modify a solver to eliminate specific forms of wasted effort. This is in contrast with the traditional generate-and-test method of using a given set of modifications of a solver as generators and then testing the modifications for efficiency gains relative to a problem distribution. While our goal-directed approach requires more detailed analysis of the computation trace, the increased analysis cost is offset by the reduction in the set of compilation candidates considered.

3. What schemes are used to evaluate changes made by REBL?

   We sample an unknown but stationary problem distribution using statistical techniques developed in [17, 37, **?**] to calculate average cost to within user-specified accuracy (ϵ) and confidence (δ) bounds in time polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

Our main contributions are: (1) a new formulation of the speedup learning problem and special representations of computation that support this formulation, (2) a new speedup learning algorithm scheme called REBL with distribution learning, theory generation and evaluation components with provable properties (3) the speedup theorem characterizing REBL with experimental support in the context of two general classes of domain theories: structural non-linear recursive theories without function symbols as exemplified by the circuit synthesis problem, and non-structural recursive theories with functions symbols like the situation calculus planning domain.

Our paper is structured as follows. Section 2 characterizes speedup learning algorithms and outlines the compilation options for declarative theories. We identify our compilation target as a decision list of special-purpose solvers for the problem distribution. In Section 3, we introduce the path grammar representation of computation and derive all the associated algorithms. We also present a multinomial estimation method for learning equivalence classes of problems in the problem distribution. In Section 4, we analyze the performance improvement characteristics of REBL. In Section 5 we present a complex example from the situation calculus planning world to illustrate the sources of inefficiency and their removal by analysis of path grammars. In Section 6, we describe experiments conducted in the planning domain that validate our algorithms and provide evidence for our theoretical analysis. Section 7 casts previous work on this problem in our framework. In Section 8, we conclude with the main points of this paper and suggest directions for future work in the design and analysis of robust speedup learning algorithms.

2 Characterizing Speedup Learning Algorithms

The study of speedup learning algorithms requires careful specification of the problem solver architecture. We focus on declarative problem solvers here: these are solvers that treat computation as controlled logical inference [18]. The focus on declarative problem solving is for three reasons.

1. Speed up learning has been mostly explored in the context of improving declarative problem solving systems. Many important problems in planning and scheduling can be easily formulated declaratively.

2. There is a lot of scope for improving declarative solvers, unlike the area of optimization of functional and imperative solvers, which has been mined out by compiler designers.

3. Declarative solvers are used not only by the AI planning and learning community, but also the database and logic programming community. Our results can synergistically be combined with optimization schemes developed in these communities. With the proliferation of constraint programming in the general computer science community, we will have the unique opportunity of providing them with dynamic optimization tools with certified properties.

We characterize a problem-solver as a pair: (T, C) where T is a definite clause theory and C a procedure or a meta-level theory for interpreting T. C is declaratively specified in a two-level solver and procedurally encoded in a one-level solver. Improvements in a one-level solver are made by reformulating the theory T, whereas improvements in two-level solvers can involve reformulations of the base theory T, as well as the meta-level interpretation policy C. Typically, EBL research in the two-level framework leaves the base theory T untouched and modifies the meta-level control policy C. Within the class of modifications that have been explored in these two frameworks, it can be shown that any modification of C within a fixed vocabulary for C can be simulated by a suitable reformulation of T. Similarly a modification of T, say conjunct reordering, can be simulated by a change in C, the addition of a conjunct preference rule to C. A REBL algorithm improves theories on-line by inductively deriving properties of a problem distribution and using it for optimization by removal of irrelevant computation in (T, C).

2.1 A formal presentation of the problem

The speedup learning problem *Speedup* can be formally characterized as follows:
Given:

- (T_o, C_o): the original (T, C) pair. T_o is the domain theory and C_o is a procedure or a declarative specification for interpreting T_o.
- A stationary but *unknown* probability distribution P over a set of problems $\{GC_i \mid 1 \leq i \leq k\}$. Each GC_i is a distinct problem class. k is assumed to be finite. We start with knowledge of an upper bound for k and devise procedures for learning it by sampling the problem distribution as in [**?**].
- A cost function c for a set S of (T, C) pairs organized as a decision list. This is our learning target, and has the form $((A_1, (T_1, C_1)), \ldots, (A_n, (T_n, C_n)))$. The A_i's are conditions on the problem instance and are acquired by learning. The cost function $c : S \times GC_i \rightarrow \mathcal{R}^+$ provides the average cost of solving problems of type GC_i using S. The cost of selecting the appropriate solver in S is included in $c(S, GC_i)$. Then, the expected cost of using S to solve problems in the distribution is simply

  $$E(S,P) = \sum_{1 \leq i \leq k} P(GC_i)c(S, GC_i)$$

  We have the following guarantee on the estimation process:

  $$Pr\left(\frac{|E(S,P) - \hat{E}(S,P)|}{E(S,P)} \geq \epsilon_{cost}\right) \leq 1 - \delta_{cost}$$

  where ϵ_{cost} and $\delta_{cost} \in (0, 1]$ are user-specified error parameters. Such estimation procedures can be generated using methods in [36, 16].
- The coverage $Cov : S \times P \rightarrow [0, 1]$ of a set S of solvers is defined in the following manner.

  $$Cov(S,P) = \sum_{GC_i \in Solvable(S)} P(GC_i)$$

  where $Solvable(S) \equiv \{GC_i \mid S \text{ solves } GC_i\}$. We learn non-overlapping classes in the problem distribution space, so that we can sum the coverages of the individual solvers.

  We have a guarantee on the process that estimates coverage:

  $$Pr\left(\frac{|Cov(S,P) - \hat{C}(S,P)|}{C(S,P)} \geq \epsilon_{cov}\right) \leq 1 - \delta_{cov}$$

  where real numbers ϵ_{cov} and $\delta_{cov} \in (0, 1]$, where ϵ_{cov} is an error parameter and δ_{cov} is a confidence parameter on coverage.

- A utility function, specified by the user, that allows the learner to calculate the tradeoff between coverage over the distribution and average cost of solving problems drawn from the distribution. The trade-off for a risk-averse learner can be captured by a parametric family of functions of the Cobb-Douglas form defined below. $0 < a < 1$ is a real number provided by the user. For a set S of solvers, the utility metric is:

$$U(S,P) = Cov(S,P)^a \left(\frac{1}{NormE(S,P)}\right)^{1-a}$$

where $NormE(S,P) = \frac{E(S,P)-E(\{T_o,C_o\},P)}{E(\{T_o,C_o\},P)}$ which is the fractional change in average cost over that of (T_o, C_o). A risk-neutral agent can use the simpler, linear combination of coverage and cost.

Find: A finite set S of (T, C) pairs that maximizes $U(S, \hat{P})$ such that $U(S, \hat{P})$ is within ϵ of its true value $U(S, P)$ with probability greater than $1 - \delta$, where $\hat{P}$ is the learned estimate of P.

In the absence of a specific utility function which determines how coverage and cost are to be traded off, the problem is that of maximizing $Cov(S, \hat{P})$ subject to a cost bound $E(S, \hat{P})$ specified by the user, such that both coverage and cost estimates are within ϵ's of their true values with probability greater than $1-\delta$. This is a natural formulation of the speedup learning problem which directly exposes its on-line character since all its inputs are not known to begin with, and the fact that solutions to it are fundamentally trade-offs between distributional coverage and average cost of solution [2].

The offline version of *Speedup*, where P is completely known and where the space of (T, C) pairs is finite and has associated coverages and costs can be simply mapped to the knapsack problem. So a restricted version of *Speedup* itself is NP-complete. For recursive theories, the set of potential (T, C) pairs is countably infinite and there is a cost associated with estimating coverages and average costs.

2.2 Goal-directed generation of new (T, C) pairs

Our formulation of the speedup learning problem has left open the question of how the (T, C) pairs with the desired properties are generated. Our scheme is the following: we use the generalized computation trace for goals drawn from GC to reveal instances of wasted computational effort. We then provide methods for "hacking" the trace to eliminate unnecessary work and calculate the new (T, C) pair that generates the modified trace on the goal scheme. The effectiveness of the new (T, C) pairs derived by this method is a function of the types and extent of the eliminated inefficiencies. To quantify the advantages of this goal directed approach to dynamic optimization of (T, C) pairs, we need to rigorously formalize computations, and establish categories of wasted effort.

2.3 Formalizing computation: C-ordered SLD-trees

We now proceed to formally describe computation traces. Since we work with definite clause theories T, we use SLD-resolution to prove goals drawn from the problem distribution. The goals considered are also definite; they are a conjunction of atomic literals, from the domain of T.

Definition 1 (Lloyd) *We will say that G' is derived from G and the definite clause $H : -A_1, \ldots A_n$ by SLD-resolution if G is the goal $G_1, \ldots, G_m, \ldots G_k$, σ is the most general unifier of H and G_m,*

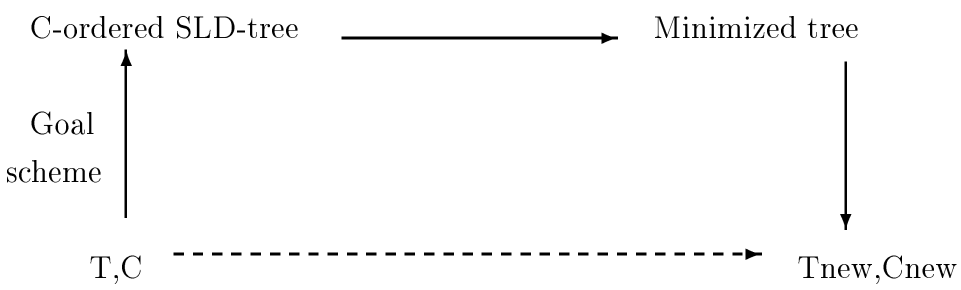


Figure 2: Generating a new (T, C) pair

and G' *is* $(G_1, \ldots G_{m-1}, A_1, \ldots, A_n, G_{m+1}, G_k) \cdot \sigma$, *where* $\cdot$ *is the operator that applies a substitution to a formula.*

Consider the simple definite clause theory about kinship defined below. This is a non-linear recursive theory because there is more than one occurrence of the recursive predicate in the antecedent of the second rule. We will use this theory to illustrate our definitions.

1. `ancestor(X,Y) :- father(X,Y).`
2. `ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).`

Consider the ground goal `ancestor(a,b)`. We can use SLD-resolution with clause 1 to obtain the new goal `father(a,b)` with most general unifier $\{X/\mathtt{a}, Y/\mathtt{b}\}$. SLD-resolution with clause 2 yields the new conjunction `ancestor(a,Z),ancestor(Z,b)` with most general unifier $\{X/\mathtt{a}, Y/\mathtt{b}\}$. We will now define the search tree for G in (T, C). This is the SLD-tree: the computation trace generated by a (T, C) pair over a definite goal.

Definition 2 *Let* T *be a set of definite clauses and* G *a definite goal. An SLD-tree for* $T \cup \{G\}$ *is the tree defined below.*

1. *Each node of the tree is a possibly empty definite goal.*
2. *The root node is* G.
3. *Let* $A_1, \ldots A_m, \ldots A_k$ $(k \geq 1)$ *be a node in the tree, where none of the literals are underlined. The node has at most* k *children: each is of the form* $\{A_1, \ldots, A_{i-1}, \underline{A_i}, A_{i+1}, \ldots, A_k$ *for* $1 \leq i \leq k$. *These correspond to all possible choices for the first literal to solve from the literal set at the node. At most one literal at a node can be underlined.*[1]
4. *Let* $A_1, \ldots, \underline{A_i}, A_k$ $(k \geq 1)$ *be a node in the tree. For the underlined atom* A_i *at this node, let* $R = \{A\text{:-}B_1, \ldots, B_q \in T \mid A_i$ *and* A *are unifiable with mgu* $\sigma\}$. *The node has at most* $|R|$ *children: each child is a node of the form* $(A_1, \ldots A_{i-1}, B_1, \ldots, B_q, A_{i+1}, \ldots, A_{k+1}) \cdot \sigma$. *These are children that are produced by SLD-resolution on the selected literal at the parent node using the elements of* R.[2] *The edge from the node to each child is labeled by the name of the rule in* R *and the substitution* σ *used in the reduction.*

[1] This represents a non-deterministic choice point.
[2] This is another non-deterministic choice point.

5. *Nodes which are the empty clause have no children. Nodes for which no rule head in T unifies with the selected literal also have no children.*

The above definition defines the search space of possible proofs of G in T. If this definition is interpreted as a procedure for constructing the SLD-tree, there are two non-deterministic choice points: the choice of literal to select at a node, and choice of clause in the set R for reducing the selected literal by SLD-resolution. The computation rule C has two components C_A and C_R, called the *antecedent ordering policy* and the *clause ordering policy*. An example is the policy used by Prolog: solving the literals at a node in order from left to right, and using rules to reduce literals in the order in which they are present in T. C_A and C_R can be declaratively encoded as productions or condition-action rules as in Soar and Prodigy. Thus any non-linear or linear planning strategy can be encoded in C. The specific encoding of C is unimportant for now: we think of C as maps from the solver's history and the set of choices at a node to a permutation over that set indicating the order in which the choices are tried. Not all options at a node need be expanded. SLD-trees can be generated dynamically during problem solving, or generated statically from knowledge of (T, C) and a goal scheme [40]. How C_A and C_R are encoded makes a difference in the effectiveness of the algorithms that learn them from examples, as well as the algorithms that interpret them during problem solving.

Definition 3 *Let T be a set of definite clauses and G a definite goal. A* C-ordered SLD-tree *for $T \cup \{G\}$ is the SLD-tree in Definition 2 with the children of every node ordered by C_A or C_R, where $C = (C_A, C_R)$ is the control policy for interpreting T.*

$C = (C_A, C_R)$ imposes a traversal order on the SLD-tree. Each branch of the C-ordered SLD-tree is an attempt to derive G in T using C. There are three types of branches in a C-ordered SLD-tree: successful branches corresponding to successful derivations of G (these end in the empty clause), infinite branches corresponding to infinite derivations, and failed branches with non-empty leaves [23].

Definition 4 *A proof of G in (T, C) is a path in the C-ordered SLD-tree of G in T that starts at the root and terminates in a leaf node with the empty clause.*

We now describe how goal classes are represented in our framework. Let us assume that all the goals we are concerned with, in the kinship theory, have the form `ancestor(X,Y)` where both `X` and `Y` are bound. The actual constants will be provided at run-time. We notate the goal scheme using the convention followed in the database optimization literature [40], by superscripting variables with b or f standing for whether they are bound or free. The definitions below show how annotations on goals are propagated by an SLD-resolution step.

Definition 5 *An annotated literal $G(X^{a_1})$ unifies with the annotated literal $G(P^{a_2})$ where X and P are annotated variables, when $G(X)$ and $G(P)$ unify by the classical rules, and annotations $a_1, a_2 \in \{b, f\}$ are compatible. The pairs (b, f), (f, b) and (f, f) are compatible. If a variable with annotation b is unified with a variable with annotation f, then both variables are coerced to have annotation b. When two variables with annotations f are unified, they continue to have the annotation f.*

This definition can be naturally extended to cover unification of literals with more than one variable. We require that the annotations of unified variables be pairwise compatible.

Definition 6 *We will say that the annotated set of goal literals G' is derived from the annotated goal literal G and the definite clause $H :- A_1, \ldots A_l$ by SLD-resolution if σ is the most general unifier of H (with every variable in H annotated by f) and G computed according to the rules in Definition 5, and G' is $(B_1, \ldots, B_l) \cdot \sigma$, where $\cdot$ is the operator that applies a substitution to a formula and each B_i is A_i with its variables annotated according to the following rules*[3]*:*

1. *If variable X in A_i occurs in the head H and occurs in no A_j for $j < i$, then X inherits the annotation of X in H.*

2. *If variable X in A_i does not occur in H and occurs in no A_j for $j < i$, then it obtains the annotation f.*

3. *If variable X in A_i occurs in some A_j for $j < i$, it obtains the annotation b.*

SLD-resolution of the goal class `ancestor`(X^b, Y^b) with clause 1 yields the goal class `father`(X^b, Y^b). SLD-resolution of `ancestor`(X^b, Y^b) with clause 2 produces the goal class `ancestor`(X^b, Z^f), `ancestor`(Z^b, Y^b). This is because `ancestor`(X^b, Y^b) unifies with `ancestor`(X^f, Y^f): the head of clause 2 augmented with annotations, to produce $\sigma = \{X/X, Y/Y\}$ where X and Y are both annotated by b. The literal `ancestor(X,Z)` is annotated as `ancestor`(X^b, Z^f) since X inherits the annotation from the head of the clause and Z obtains the annotation f by the definition above. The literal `ancestor(Z,Y)` obtains the annotation `ancestor`(Z^b, Y^b), since Z is bound by the previous literal and Y is bound in the head. Using Definition 6, we can generate schematic C-ordered SLD-trees for a goal class instead of a specific goal. A statically generated C-ordered SLD-tree is shown in Figure 3. Nodes that are alphabetic variants of each other are connected by a dashed line in this figure. C-ordered SLD-trees are infinite structures for goal classes in recursive theories.

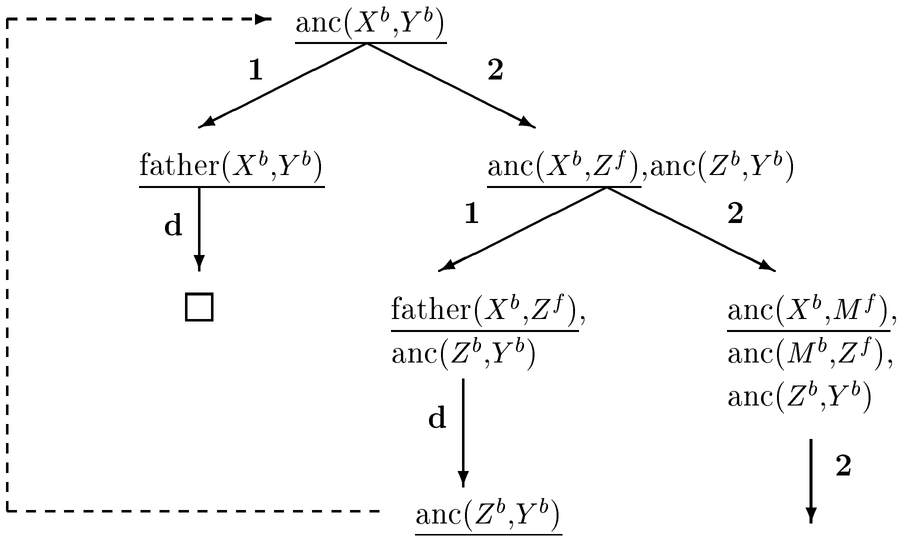


Figure 3: Schematic C-ordered SLD-tree

The cost of computing the C-ordered SLD-tree for an arbitrary policy C can be very high; most of our examples therefore consider fixed, limited-history policies. In principle, the size of the C-ordered SLD-tree for a goal class can be exponential in the size of (T, C).

[3]Note that H acquires new annotations after being unified with G.

Definition 7 *We will call a control policy C* compact *if its description length is smaller than the best encoding for a complete extensional specification of rule and literal order for every node in the C-ordered SLD-tree for a goal class GC.*

Compactness captures the intuition that there are regularities in the choice of literal and rule order for the given theory T and goal class GC. The default Prolog strategy is compact because its description (do the literals in left-to-right order, and rules from top to bottom), is considerably smaller than a pure extensional encoding whose length will be proportional to the size of the C-ordered SLD-tree for GC. One aspect of the speedup learning problem is to acquire compact C's specialized for a given distribution. Compactness can be rigorously characterized using algorithmic information theory and minimal length encoding.

Definition 8 *A control policy C for the goal class GC on the theory T is* Markovian, *if for every $G \in GC$, and for each node in the C-ordered SLD-tree for G, the choice of literal to select, and choice of rule to select to expand the chosen literal, can be made by examining at most that node and its immediate ancestor in the tree.*

The fixed policy of Prolog to solve literals from left to right at each node, and to select rules to solve a literal in the order in which they are entered in the database, is Markovian: in fact, Prolog only needs to examine the current node to implement this policy. A control policy that requires the tracking of the complete history of rule invocations and literal selections is non-Markovian. Non-Markovian policies (even if they are compact) can be very expensive to implement. A compact, Markovian control policy uses local information to make literal and rule choices. This does not ensure that the policy is effective: it could perform a computation that is exponential in the size of its input information, for instance a complete lookahead search from a node. So we add the additional constraint that the policy be effective.

Definition 9 *A control policy C is* effective *if it performs $O(f(I))$ computation where f is a polynomial in I, which is the size its input information.*

In this paper, we restrict our attention to effective, compact, Markovian control policies C. Now that we have identified the class of (T, C) pairs of interest and formulated computation of G as a C-ordered SLD-tree, we are ready to study generic classes of wasted computation.

2.4 Irrelevant computation in C-ordered SLD-trees

We distinguish between base and meta-level computation in a C-ordered SLD-tree. At each node of the C-ordered SLD-tree either a decision about the order in which to solve literals is made, or a decision is made to determine the order in which applicable rules will expand an underlined literal. These are meta-level decisions and there is a cost associated with them, which is a function of how complex C_A and C_R are. Base-level computation consists of the actual expansion of the nodes in a C-ordered SLD-tree. We start with an intuitive definition of irrelevant computation [41, 34].

Definition 10 *We will call a base- or meta-level computation* **irrelevant** *if its results are never used, or if its results are already available from prior effort.*

We can make defeasible inferences about the relevance of a computation or make provably correct ones. In this paper, we limit ourselves to the latter category.

2.5 Irrelevant base-level computation: detection and repair

As an example of an irrelevant base-level computation, consider the repetition of the calculation of $fib(X^b, Y^f)$ for several values of X in a naive formulation of the Fibonacci function. This can be detected by noting the fact that the same underlined *functional* literal appears more than once in the C-ordered SLD-tree. A literal of the form $l(\vec{X}^b, \vec{Y}^f)$ is functional if for every distinct instantiation of $\vec{X}$ there is atmost one instantiation of $\vec{Y}$ which makes $l(\vec{X}, \vec{Y})$ true. Repeated derivation of a functional literal is wasted because the results are available from the previous computation of that literal. This wastage can be eliminated by rewiring the C-ordered SLD-tree to lookup the literal instead of recomputing it. A trace hack that accomplishes this is shown below. This trace hack eliminates repeated computation by caching: remembering (X, Y) pairs that have already been computed.

$\underline{\mathrm{fib}(5,V^f)}$

$\downarrow$ **2**

$\underline{\mathrm{fib}(4,{V_1}^f)}$, $\mathrm{fib}(3,{V_2}^f)$, $V = {V_1}^b + {V_2}^b$

$\downarrow$ **2**

$\underline{\mathrm{fib}(3,{V_3}^f)}$,$\mathrm{fib}(2,{V_4}^f)$, $V_1 = {V_3}^b + {V_4}^b$, $\mathrm{fib}(3,{V_2}^f)$, $V = {V_1}^b + {V_2}^b$

$\downarrow$ **2**

$\underline{\mathrm{fib}(2,{V_5}^f)}$,$\mathrm{fib}(1,{V_6}^f)$, $V_3 = {V_5}^b + {V_6}^b$, $\mathrm{fib}(2,{V_4}^f)$, $V_1 = {V_3}^b + {V_4}^b$, $\mathrm{fib}(3,{V_2}^f)$, $\mathrm{V} = {V_1}^b + {V_2}^b$

$\downarrow$ **2**

$\underline{\mathrm{fib}(1,{V_7}^f),\mathrm{fib}(0,{V_8}^f),V_5 = {V_7}^b + {V_8}^b, \mathrm{fib}(1,{V_6}^f),V_3 = {V_5}^b + {V_6}^b,}$ $\mathrm{fib}(2,{V_4}^f)$, $V_1 = {V_3}^b + {V_4}^b$, $\mathrm{fib}(3,{V_2}^f)$, $V = {V_1}^b + {V_2}^b$

$\downarrow$ **1,0,d,1,d**

$\underline{\mathrm{fib}(2,{V_4}^f)}$, $V_1 = {V_3}^b + {V_4}^b$, $\mathrm{fib}(3,{V_2}^f)$, $V = {V_1}^b + {V_2}^b$

$\downarrow$ **2**

Repeated functional literals

Figure 4: Repeated base-level computation

How hard is it to detect repeated base level computation of this type? Given an instance of the computation, we only need to look for repeated instances of the same underlined literal. We also need to check that the literal is functional. In principle, to check the functionality constraint, we need to determine that there is at most one solution to the unbound variable for every instantiation of the bound variables in the literal. For arbitrary Horn theories, this is decidable, because it reduces to calculating all models of a Horn theory. However the task is #P_complete. We provide functionality declarations to aid the learner.

How can this repetition be eliminated, and how expensive is the repair operation? The repair scheme used for this case involves:

1. Introduction of a new predicate: for storing computed values.

2. Introduction of new rules: for caching computed values and looking them up.

The repair is a mechanical one, and is a cheap (linear in the number of rules in T) operation. We illustrate it in the context of the *fib* computation. We introduce a new predicate *fibcache* of two arguments. We transform each rule in the original domain theory by introducing a step after the computation of a *fib* subgoal in the body of each rule. This step caches the subgoal in *fibcache*. We then introduce a new base case rule before the recursive computation step that checks to see if the value is computed before using the recursive rule for computing it. This transformation has been implemented and predictable experimental results are observed (Figure 5. Each data point on the graph in the figure represents the average solution time over 10 randomly selected *fib* goals. As more *fib* values are cached, subgoaling is replaced by lookups; so very shallow proof trees result.

```
fib(0,1).
fib(1,1).
fib(N,V) :- fibcache(N,V).
fib(N,V) :-
        N > 1, N1 is N-1, fib(N1,V1),
               ck_assert(fibcache(N1,V1)),
               N2 is N-2, fib(N2,V2),
               ck_assert(fibcache(N2,V2)),
               V is V1 + V2,
               ck_assert(fibcache(N,V)).
```

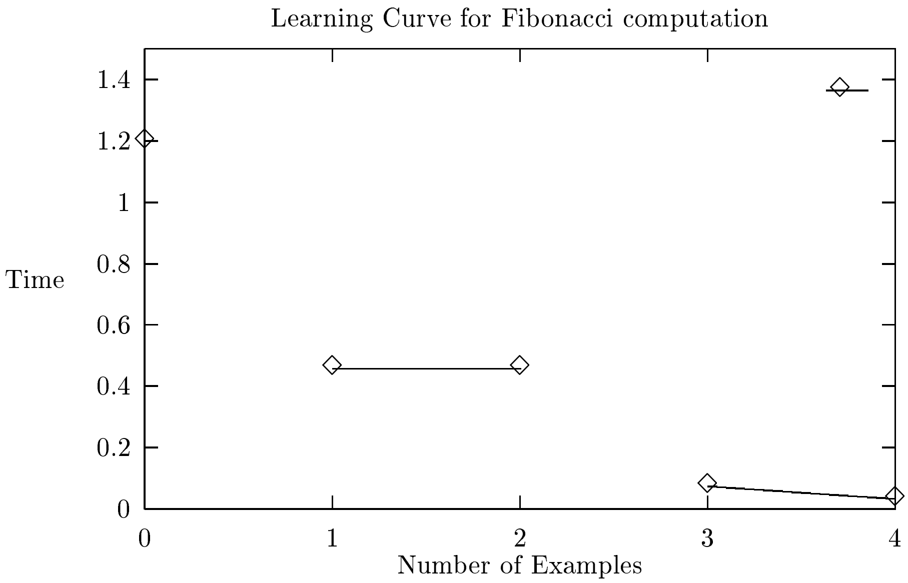


Figure 5: The Fibonacci speedup experiments

Every *fib* subgoal gets repeated and thus they all have to be cached. In theories where every instance of a recursive subgoal is not repeated, we need to calculate which subset to cache. This requires deliberation about the worth of each cached subgoal using information about the distribution of future queries as in [?]. This modification policy does not affect the coverage, but improves efficiency. The extent to which efficiency is improved is a function of the degree of repetition of the goals: the gain is $\sum_{l \in L} frequency(l) * c((T,C),l)$ where L is the set of repeated literals and $frequency(l)$ is the number of times it occurs as an underlined literal in the C-ordered SLD-tree. In the Fibonacci computation, each literal is repeated an exponential number of times so that the performance gain is considerable without loss of coverage. In more general theories, we can expect to get constant factor speedups with caching policies that eliminate repeated derivation of a functional literal. Even though our example eliminated repeated *successful* derivations of a functional literal, the same results can be shown for *repeated derivation of the failure* of a functional literal[8, 31]. It should be emphasized that this not the only possible generic repair scheme for this computation. Another repair scheme for eliminating repeated computation of functional literals converts C into a forward chaining problem solver. This is the repair adopted by the technique of magic sets [40] in the database optimization literature.

2.6 Irrelevant meta-level computation: detection and repair

As an example of irrelevant meta-level computation, consider the repeated derivation of the *decision* to use a specific rule sequence to solve a given underlined literal at different nodes in a C-ordered SLD-tree. We illustrate thus using a non-linear recursive theory for the synthesis of combinational circuits. Let `imp(X,Y)` stand for the fact that `X` is implemented by `Y`. Rule `s1` states that a wire is its own implementation. A typical goal solved by this theory is `imp(¬((¬ c1 ∨ ¬ c2) ∨ ¬ c3),` X^f`)` where the first argument is a boolean expression and the second argument is a free variable that will finally hold the circuit that implements the expression. A generalized C-ordered SLD-tree for this goal scheme `imp`X^bY^f is shown in Figure 6 where lookup steps are omitted for clarity. •'s stand for unification failures. The goal class consists of boolean expressions constructed from the operator set $\{\wedge, \neg, \vee\}$. The only solvable goals are ones that are synthesizable purely using AND-gates.

```
s1.  imp(X,X) :- wire(X).
s2.  imp(¬¬X1,Y) :- imp(X1,Y).
s3.  imp(¬(X1∨X2),Y1∧Y2) :- imp(¬X1,Y1), imp(¬X2,Y2).
```

Note that the underlined literals that match the scheme `imp(¬¬`X^b, Y^f`)` are expanded exactly the same way throughout the proof. The same meta-level decision is rediscovered at multiple points in the tree. This repetition can be eliminated by remembering this decision. It can be achieved by designing a new C or modifying the existing T so that this choice is made exclusively for goals of this form and all others are pruned.

It should be clear that any path from the root to a *non-empty leaf* in a C-ordered SLD-tree is an instance of both irrelevant base-level and meta-level computation. This is because the results of this calculation are not used to solve the goal. Even though we treat them distinctly, there is a strong relationship between repeated computation at the base and meta levels. Repeated meta-level computation need not necessarily result in repeated base level computation. The reduction of `imp(¬¬`X^b, Y^f`)` into `wire(`X^b`)` for distinct values of X through the C-ordered SLD-tree is not an instance of repeated base-level computation. However, repeated base-level computation (as in the fibonacci example) always co-occurs with repeated meta-level computation.

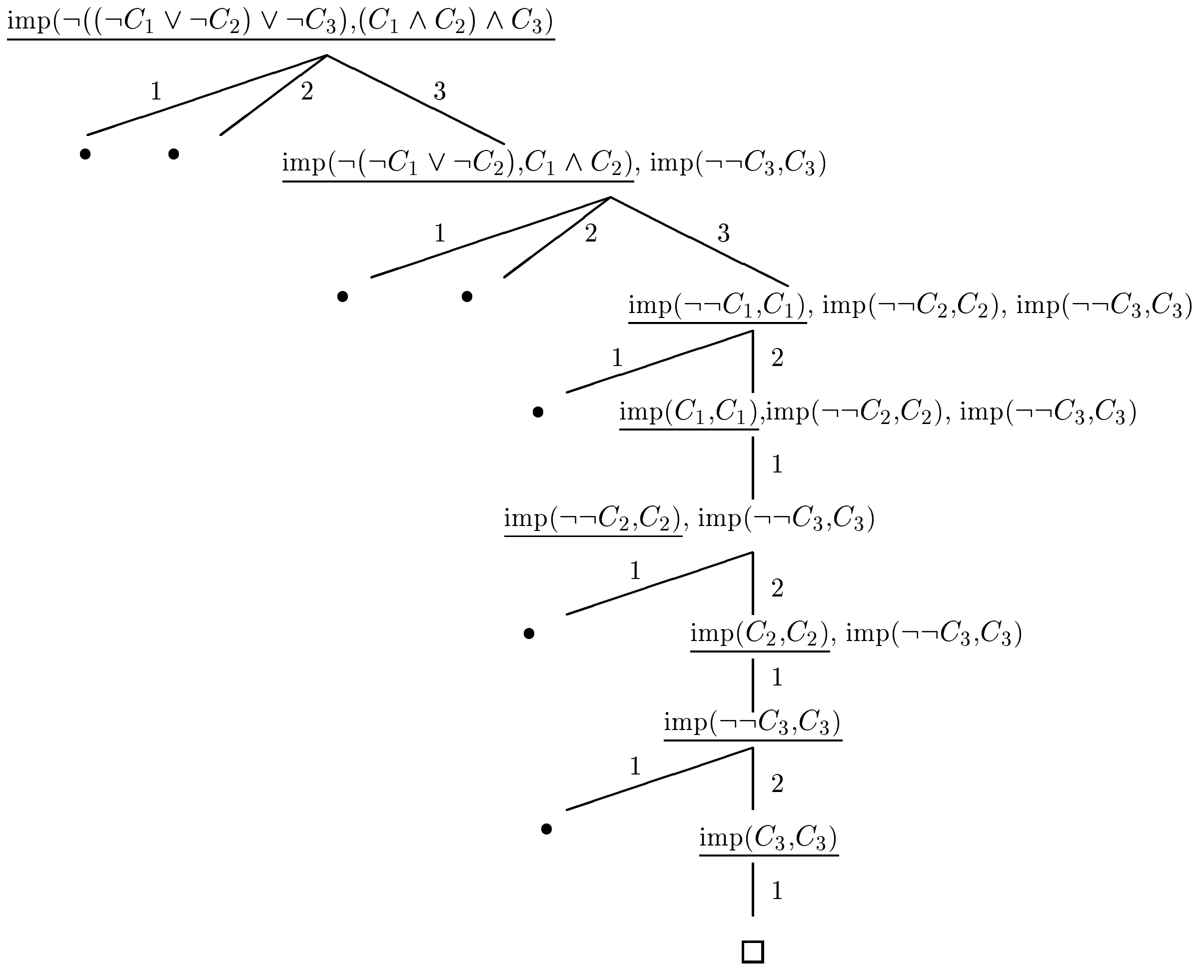


Figure 6: C-ordered SLD-tree for example synthesis

3 Abstractions of C-ordered SLD-trees: Path grammars

Traces of specific computations in recursive theories can be very long and hacking traces directly to detect and repair theories is a computationally intensive task. We therefore consider abstract representations of C-ordered SLD-trees of goal classes; both general as in Figure 3 and specific as in Figure 6 that reveal the presence of specific forms of repeated meta-level computation. In particular, we view a (T, C) pair as a generator of derivations. Each path from the root to a leaf node is an attempt to solve the given goal. We represent paths as a string in a context-free language. A trace is an extensional representation of the paths. A compact intensional representation is developed using a *path grammar*.

A path grammar is a 4-tuple (N, A, P, S), where N is a set of non-terminal symbols, A is a set of terminals composed of rule names and names of fact classes and their associated ranks (the number of conjuncts in the antecedents), P is a set of productions of the form $X \rightarrow rX_1 \ldots X_m; [r_1 \ldots r_l]$ or $X \rightarrow r\ f$, where $r, r_1 \ldots r_l \in A$ and m is the rank of r, $S \in N$ is the start symbol of the grammar. The ordered list of rules $r_1 \ldots r_l$ is tried at the node that X denotes before r was successfully applied. f is a special symbol denoting that the rule failed and is used to capture the failed leaves of the computation tree. Nodes that denote lookups of a data base for ground facts are considered reduced by d_f where f is the retrieved fact class. Recall that these are the labels on the edges in the C-ordered SLD-tree defined before. Paths that end in the empty clause are successful computations. We can generate a context-free grammar that generates all the *successful* paths for a given goal scheme: this special class of path grammars are called *proof grammars*.

Path and proof grammars are regular tree grammars [**?**]. A regular tree-grammar can be viewed as a context-free grammar with a terminal alphabet consisting of rule names and names of fact classes and parantheses (optional) to set off subproofs. If we treat paths in the search tree as words, then the forests generated by regular tree grammars are special context-free languages.

3.1 Extracting proof grammars from a C-ordered SLD-tree

We illustrate the construction of a path and proof grammars from C-ordered SLD-trees of a goal schema in the context of the theory of synthesis of combinational circuits. Consider again the C-ordered SLD-tree in Figure 6 where lookup steps are omitted for clarity. There is one successful path through the C-ordered SLD-tree and several failed paths which end in $\bullet$'s.

Note that in this abstraction of C-ordered SLD-trees, we omit representing a variety of state information available at each node in the tree. These include pending goals at a node, rules that remain to be tried, binding information, among others. We will include them later in more expressive encodings of the C-ordered SLD-tree. The objective here is to record only the information needed to detect a certain class of cases of irrelevant rule-choice computation. Each production represents a local rule choice decision. The productions for our example tree are:

procedure Extract-Proof-Grammar (E, g)

Inputs: generalized proof path E of goal schema g.

Output: minimized proof grammars G_N, corresponding to $\equiv_N, N \geq 1$.

Method:

1. Construct initial grammar G_{init} = (N,A,P,S) from E.
   N is the set $A_i, i \geq 0$ where each A_i denotes a subgoal in E. Each subgoal is labeled by a distinct symbol.
   A is the set of terminals $r_j, j \geq 1$ and d_f for each fact scheme f.
   P is the set of productions of the form $A_i \longrightarrow r\ A_{i_1} \ldots A_{i_m}\ ; L$ for each segment of the path E.
   L is the list of rules that were tried before r.
   The start symbol S is A_0.
2. G = Minimize(G_{init}).

Table 1: The algorithm for constructing proof grammars from a successful path in a C-ordered SLD-tree

$$
\begin{array}{llllll}
A_0 & \longrightarrow & \mathtt{1} & f & & \\
A_0 & \longrightarrow & \mathtt{2} & f & & \\
A_0 & \longrightarrow & \mathtt{3} & A_1 & A_2 & ;[1,2] \\
A_1 & \longrightarrow & \mathtt{1} & f & & \\
A_1 & \longrightarrow & \mathtt{2} & f & & \\
A_1 & \longrightarrow & \mathtt{3} & A_3 & A_4 & ;[1,2] \\
A_3 & \longrightarrow & \mathtt{1} & f & & \\
A_3 & \longrightarrow & \mathtt{2} & A_5 & & ;[1] \\
A_5 & \longrightarrow & \mathtt{1} & & & ;[] \\
A_4 & \longrightarrow & \mathtt{1} & f & & \\
A_4 & \longrightarrow & \mathtt{2} & A_6 & & ;[1] \\
A_6 & \longrightarrow & \mathtt{1} & & & ;[] \\
A_2 & \longrightarrow & \mathtt{1} & f & & \\
A_2 & \longrightarrow & \mathtt{2} & A_7 & & ;[1] \\
A_7 & \longrightarrow & \mathtt{1} & & & ;[]
\end{array}
$$

The initial proof grammar G_c has start symbol A_0, the non-terminal set $\{A_i \mid 0 \leq i \leq 7\}$, the terminal set $\{\mathtt{1}, \mathtt{2}, \mathtt{3}\}$, and those productions in set P shown above that do not contain the symbol f in their right hand sides. $L(G_c)$ consists of only one string: the encoding of the example proof. The generalization task is to induce a *reduced* tree grammar from G_c that generates the example proof and that minimizes repeated rule choice computation not just over this example, but for all examples of this class in the distribution. This is the inductive step.

Since the domain theory is recursive, we have many instances of repeated rule choice computation in this example. For instance, observe that the control decision made for reducing literal A_0 is the same as the one made for literal A_1. In both cases, the underlined literal $\mathtt{imp}X^bY^f$ is reduced by rule 3 after its failure by reductions via 1 and 2. The problem solver derived the control decision made at A_1 again at A_3. We identify and eliminate this by placing the non-terminals A_1 and A_3 in the same equivalence class and by generating a new set of productions as the quotient of the original set with respect to the generated equivalences. In particular, we will say that

$$A_i \equiv_1 A_j \text{ if and only if } \exists r \;\; A_i \longrightarrow r\alpha_1 \ldots \alpha_m; L \in P \text{ and } A_j \longrightarrow r\beta_1 \ldots \beta_m; L \in P$$

A version of this criterion was formulated by [4] in the context of finite-state automata representations of the control choice computation. Since the failures at A_0 and A_1 were unification

procedure Minimize (G)
Inputs: A proof grammar $G = (N, A, P, S)$.
Output: The equivalence classes $\equiv_N$, and the quotient grammars $[G/\equiv_N]$
$= (N/\equiv_N, A, P/\equiv_N, [S]\}$
Method:
1. Construct $\equiv_1$ by putting nodes A_i and A_j that are reduced by the same rule into the same equivalence class.
2. Repeat until successive partitions are identical.
   Construct $\equiv_N$ by placing nodes A_i and A_j in a class, if they are $\equiv_1$, and their children are $\equiv_{N-1}$.
3. Construct Quotient(G,$\equiv_i$), for all i in step 2.

procedure Quotient (G,$\equiv$)
Inputs: A context-free proof grammar $G = (N, A, P, S)$, and an equivalence relation $\equiv$ defined on the non-terminal symbols of the grammar.
Output: The quotient grammar $[G/\equiv] = (N/\equiv, A, P/\equiv, [S]\}$
Method:
1. Let $[A_i]$ stand for the equivalence class of A_i under $\equiv$.
   $[A_i] = \{A_j \mid A_j \equiv A_i\}$. Now construct $N/\equiv = \{[A_i] \mid A_i \in N\}$
2. $P/\equiv = \{q \mid$ substitute every non-terminal A_j in $p \in P$ by $[A_i], A_j \in [A_i]\}$
   Associate with every element in $P/\equiv$ a frequency count of the number of productions in P that it stands for.

Table 2: Constructing $\equiv_N$ by dynamic programming and minimizing proof grammars by $\equiv_N$

failures, we do not expect to gain significantly by the elimination of the repeated decision making. A more serious example of the types of complexity reductions that is made possible by this minimization technique is explained in the next section. We use this example here merely to introduce the concept of a proof grammar and its minimization.

The reduced (or quotient) grammar $G_r = G_c/\equiv_1$ generated by this node equivalence criterion is: $\{\{[A_0], [A_2], [A_5]\}, \{3, 2, 1\}, P_r, [A_0]\}$ The quotient set P_r of productions is shown below. Non-terminal $[A_i]$ is the representative of the equivalence class to which A_i belongs. Note that we have captured all the successful control decisions needed to replay the proof of the example. Each production corresponds to the application of a *single* rule in the domain theory.

$$
\begin{array}{lllllll}
[A_0] & \longrightarrow & 3 & [A_0] & [A_2] & ; [1,2] & (1) \\
[A_0] & \longrightarrow & 3 & [A_2] & [A_2] & ; [1,2] & (1) \\
[A_2] & \longrightarrow & 2 & [A_5] & & ; [1] & (3) \\
[A_5] & \longrightarrow & 1 & & & ; [] & (3)
\end{array}
$$

The $\equiv_1$ notion introduced above is the base case of a more general form of equivalence introduced below. We will say that A_i and A_j, which are two subgoals in a C-ordered SLD-tree are $\equiv_N$ if A_i and A_j are $\equiv_1$ and the subgoals needed to establish A_i and A_j are $\equiv_{N-1}$.

$$
A_i \equiv_N A_j \text{ if and only if } \quad \exists r \;\; A_i \longrightarrow r\alpha_1\alpha_2\ldots\alpha_m; L \in P \text{ and } A_j \longrightarrow r\beta_1\beta_2\ldots\beta_m; L \in P \\
\text{and } \alpha_k \equiv_{N-1} \beta_k, k = 1, \ldots, m
$$

As an example of its use, we construct $\equiv_2$ for our circuit example. A simple dynamic programming method (See Table 2) can be used to calculate $\equiv_N$ from $\equiv_1$ for $N > 1$. The set of rules associated with a class of non-terminals at level 1 is the reducing rule itself. Rules associated with classes of non-terminals at level n are obtained by merging the rules associated with the LHSs of these non-terminals. Thus, for example, $[A_0]$ at level 2 has associated with it the rule tree [3(3)(2)] since its first subgoal is reduced by rule 3 and its second subgoal is reduced by rule 2.

Equivalence class	Equivalence Level	Associated Rules	Frequency
$[A_0, A_1]$	$\equiv_1$	[3]	2
$[A_2, A_3, A_4]$	$\equiv_1$	[2]	3
$[A_5, A_6, A_7]$	$\equiv_1$	[1]	3
$[A_2, A_3, A_4]$	$\equiv_2$	[2(1)]	3
$[A_5, A_6, A_7]$	$\equiv_2$	[1(d)]	3
$[A_0]$	$\equiv_2$	[3 (3) (2)]	1
$[A_1]$	$\equiv_2$	[3 (2) (2)]	1

Note that $\equiv_N$'s are extracting larger and larger subproofs in the tree. The construction of $\equiv_N$ stops, when two successive partitions of the nodes generated are identical. We also extract the frequency of occurrence of these subproofs in the tree. We will determine which of these subproofs are worthwhile keeping over the given distribution of problems.

The reduction of the original grammar with respect to $\equiv_2$ is $G/\equiv_2 = \{\{[A_0], [A_1], [A_2], [A_5]\}, \{3, 2, 1\}, {P^2}_r, [A_0]\}$. ${P^2}_r$ is the following set of productions:

$$
\begin{array}{lllll}
[A_0] & \longrightarrow & 3\ [A_1]\ [A_2] & ;\ [1,2] & (1) \\
[A_1] & \longrightarrow & 3\ [A_2]\ [A_2] & ;\ [1,2] & (1) \\
[A_2] & \longrightarrow & 2\ [A_5] & ;\ [1] & (2) \\
[A_5] & \longrightarrow & 1 & ;\ [] & (3)
\end{array}
$$

Proposition 1 *The construction of the quotient proof grammars from an C-ordered SLD-tree for an example goal takes $O(n^3)$ time where n is the number of nodes in the tree.*

We can use the containment relation between the languages generated by a grammar to define a partial order on proof grammars. We will say that $G_1 \leq G_2$ if and only $L(G_1) \subseteq L(G_2)$. In general, the question of whether $L(G_1) \subseteq L(G_2)$ for arbitrary context-free grammars is undecidable. However, for proof grammars, a $O((p_1 + p_2)^2)$ time algorithm checks containment, where p_1 and p_2 are the number of productions in the two grammars.

procedure check-$\leq$ (G_1, G_2)
Inputs: Proof grammars $G_1 = \{N_1, A_1, P_1, S_1\}$ and $G_2 = \{N_2, A_2, P_2, S_2\}$.
Outputs: true if $G_1 \leq G_2$, false otherwise.
We let $P \mid_{N,A}$ denote those productions from set P that use symbols from the set $N \cup A$.
Method:
if $A_1 \subseteq A_2 \wedge N_1 \subseteq N_2$ then
if every $p \in P_1$ is derivable from $P_2 \mid_{N_1,A_1}$ then return true else return false
else return false.

Table 3: The specialization relation between proof grammars

Proposition 2 *The procedure to check whether $G_1 \leq G_2$ takes $O((p_1 + p_2)^2)$ time where p_1 and p_2 are the number of productions in the proof grammars G_1 and G_2 respectively.*

The proof grammars generated by $\equiv_i$, $i \geq 1$ are ordered by the containment relation defined above. We can easily show that $G/\equiv_{i+1} \leq G/\equiv_i$. This holds because with each level, greater constraints are placed on how subgoals are expanded.

Proposition 3 *The quotient proof grammars generated by Algorithm Minimize from the original proof grammar G are such that $G/\equiv_n \leq G/\equiv_{n-1} \leq \ldots \leq G/\equiv_1$.*

Note the proof grammar generated by $\equiv_2$ describes only one proof: that of the example problem. The grammar corresponding to $\equiv_1$ describes an infinite collection of proofs, where the second subgoal resulting from each $\mathtt{imp}(\neg(X \vee Y, Z))$ goal is reduced by rule 2. Each production in the reduced grammar produced by $\equiv_n$ identifies a sequence of rule choices denoting a subproof of height n in a C-ordered SLD-tree to solve the subgoal at the head of the production. When there are multiple productions with the same non-terminal it represents a disjunctive choice and a learning opportunity to discriminate between the applicability conditions for the alternate solutions to a given subgoal.

If we were to minimize rule choice computation over the given example, we would choose the quotient proof grammar corresponding to the largest n such that the node partition generated by $\equiv_n$ is distinct from that generated by $\equiv_{n-1}$. The proof grammar generated by $\equiv_2$ in our example is such a grammar. It reproduces the exact rule choices made in the solution of the example. Thus at each decision node in the C-ordered SLD-tree, we have exactly one choice of rule to expand the underlined literal. Our challenge, however, is to minimize rule-choice computation not just over this example, but over a population of examples drawn from the problem distribution. The candidates for this minimization are the reduced grammars generated using $\equiv_i$, $1 \leq i < n$. This set of grammars represents the points in the tradeoff between coverage and cost discussed in our formulation of the speedup learning problem. The previous proposition characterized the relationship between the coverages of the elements in the set of reduced grammars generated by $\equiv_i$. The next proposition identifies the relationship between the costs of solving a problem by the various elements in the set of reduced grammars. We use a simple cost model common in the search literature [21] and used by [31, 39]. The cost of building a C-ordered SLD-tree is approximated as b^d where b is its branching factor and d its depth.

Proposition 4 *The rule choices for a given non-terminal in the production set for the quotient grammar corresponding to $\equiv_{i+1}$, is a subset of the choices for the same node in the production set for the quotient grammar generated by $\equiv_i$.*

This proposition is illustrated in the context of our running example. Note that $[A_0]$ can be reduced by the first two productions in the quotient grammar with respect to $\equiv_1$, but there is only way of reducing it in the quotient grammar generated by $\equiv_2$. This is a local reduction in branching factor for subgoals through the tree and thus represents a cost saving.

The $\equiv_i$ quotients reduce the number of rule choices for each distinct subgoal in the C-ordered SLD-tree, but not the total number of decision points in the tree. Efficiency gains can be obtained not just by reducing branching factors but also by reducing the depth of the C-ordered SLD-tree. We can reduce the number of decision points in the tree by a policy of non-terminal elimination in the reduced proof grammar. Here is a specific policy that we found useful in our experimental studies of REBL.

Definition 11 *A non-terminal in a proof grammar is a candidate for elimination, if*

1. *It is non-recursive. That is,* $[A_j] \not\longrightarrow \alpha[A_j]\beta$ *using the production set. The non-terminal* $[A_0]$ *is recursive in the grammar generated from* $\equiv_1$*. The non-terminals* $[A_2], [A_5], [A_7]$ *are non-recursive. What this prevents is the generation of new T's that self-unfold recursive rules.*

2. *It is non-disjunctive. That is,* $[A_j]$ *is eliminable if there is no more than one production with* $[A_j]$ *on the left-hand side. A subgoal that can be reduced in multiple ways is not eliminated, because it multiplies cases throughout the new T. This is similar to the policy adopted in IOE [7].*

The modified production set constructed from the quotient under $\equiv_1$ by eliminating $[A_2]$ and $[A_5]$ is shown below. This grammar is denoted $Elim(G/\equiv_1, \{[A_2], [A_5]\})$.

$$
\begin{array}{lll}
[A_0] & \longrightarrow & \texttt{3(2 (1))(2 (1))} \quad (1) \\
[A_0] & \longrightarrow & \texttt{3 (}[A_0]\texttt{) (2 (1))} \quad (1)
\end{array}
$$

There is still a non-deterministic choice to be made between the two productions for $[A_0]$. We can deductively or inductively acquire characteristics of the inputs to make this production set *separable*. By separable, we mean that the applicability conditions for the two productions are mutually exclusive with respect to the problem distribution. In this case, we can discriminate between the applicability conditions of the two rules by testing `wire` predicates in the antecedents, as we shall see later. If we had chosen to eliminate $[A_5]$ alone we obtain the production set $Elim(G/\equiv_1, \{[A_2]\})$:

$$
\begin{array}{llll}
[A_0] & \longrightarrow & \texttt{3(}[A_2]\texttt{) (}[A_2]\texttt{)} & (1) \\
[A_0] & \longrightarrow & \texttt{3 (}[A_0]\texttt{) (}[A_2]\texttt{)} & (1) \\
[A_2] & \longrightarrow & \texttt{(2(1))} & (3)
\end{array}
$$

Proposition 5 *Consider the quotient grammar* G_i *generated under* $\equiv_i$*, for some* $i \geq 1$*. Let* $S_1 \subseteq S_2$ *be subsets of the set of eliminable non-terminals in* G_i*. Then,* $Elim(G_i, S_2) \leq Elim(G_i, S_1)$*.*

We prove this by setting up a homomorphism from the nodes in the C-ordered SLD-tree corresponding to to $Elim(G_i, S_2)$ to those in the tree corresponding to $Elim(G_i, S_1)$. For each quotient grammar under $\equiv_i$ with l eliminable non-terminals, we can define $Elim(G/\equiv_i, S)$ for the 2^l subsets of these non-terminals.

It is useful to note here that for non-recursive theories, $\equiv_1$ generates the only distinct partition of nodes in the C-ordered SLD-tree. All non-terminal symbols, save for the root, $A_i, i > 0$ are eliminable. This sets up a lattice of $Elim(G/\equiv_1, S)$ where S is a subset of all the eliminable nodes $\{A_i \mid i > 0\}$. In fact, the operationality debate [20] in the early days of research in explanation-based learning can be viewed as debates on the choice of S. The EBG algorithm of [26] chose S to be entire set of eliminable symbols. The papers of [1, 7, 20] proposed using other smaller subsets. In recursive theories, there is repeated rule choice computation, thus there is an $n > 1$ for which $\equiv_n$ generates a distinct partition of the nodes of the C-ordered SLD-tree. We can show that the criterion developed by [4] is $\equiv_1$, and that developed by [32] is $\equiv_2$.

To summarize, the criterion of minimizing redundant rule choice computation has two major components:

1. Eliminate repeated computation: We identify nodes with the "same" control decision and collapse them into an equivalence class. This is captured by $\equiv_i$.

2. Eliminate unneeded intermediate computation: We minimize the number of distinct choice points in the C-ordered SLD-tree by removing eliminable symbols. This is captured by our $Elim(G, S_j)$ construction.

Definition 12 *The set of proof grammars $V(e)$ constructed from an example e using the $\equiv_i$ and the Elim construction is a partially ordered set under the $\leq$ relation. We will call this the version space of proof grammars induced by e.*

The version space $V(e)$ induced by our example in Figure 6 is shown in Figure 7. There are a combinatorial number of elements in $V(e)$, so provable pruning criteria are essential for managing the complexity of acquiring the decision list of solvers for the given distribution of problems. We have no grounds for eliminating any of these candidates on the basis of the one observed example. In Section 3.4 we discuss methods of merging the grammars obtained by solving other problems sampled from the query distribution to limit the number of the grammars considered. We now turn our attention to reconstructing (T, C) pairs from a proof grammar.

3.2 Reconstructing a (T, C) pair from a proof grammar

Given the original pair (T, C), and a reduced proof grammar G, we can reconstruct the new pair (T_s, C_s) such that the C-ordered SLD-tree on (T_s, C_s) can be represented by G. In our first reconstruction procedure, described in Table 4, we consider rule ordering, rule unfolding, and substitution as the only basic theory modification operators. Our reconstruction algorithm is introduced via an example. T_s for the reduced production set $Elim(G/\equiv_1, [A_2, A_5])$ constructed in the previous section is shown below.

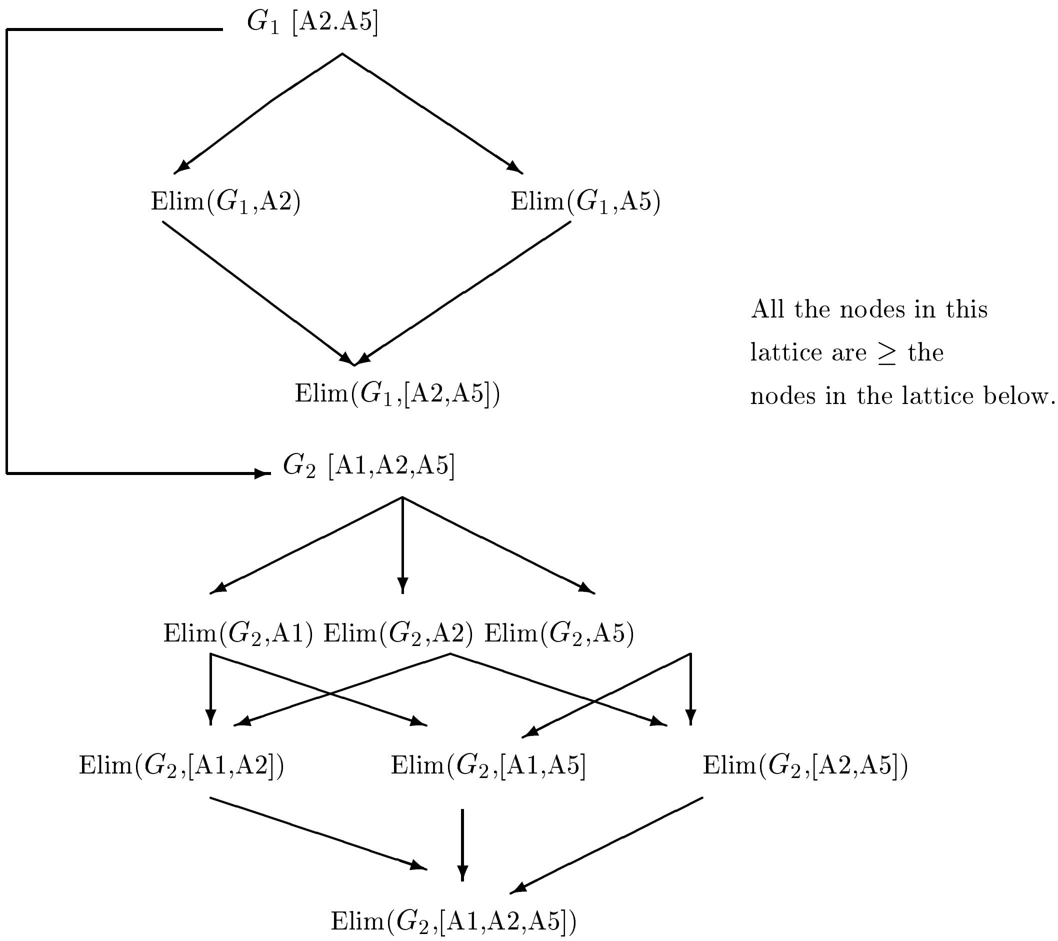
```
4.  imp1(¬(¬ X1 ∨ ¬ X2), and(X1,X2)) :- wire(X1), wire(X2).
5.  imp1(¬(X1 ∨ ¬ X2), and(Y1,X2)) :-  imp1(¬X1,Y1), wire(X2).
```

We have adopted the policy of compiling changes in T while leaving C untouched. An alternate theory reconstruction procedure can adopt the reverse policy.

We introduce a new predicate for the solution of a specialized class of goals. Another choice is to add the specialized rules into the existing domain theory. Since the rules in the specialized solvers can be derived from the original theory, it makes the overall theory redundant. This redundancy causes repetition of base level computation in the presence of failures as noted in [36, 24, 25] . Therefore, we keep the specialized solver separate by renaming the predicate `imp` as `impN`, $\texttt{N} \geq 1$. The C-ordered SLD-tree generated by this new theory is shown in Figure 8. There are two decision points where distinct rule choices are made. This trace is optimal from the perspective of minimization of rule choice computation for this example.

Rules from T that do not appear in the proof of a goal schema are eliminated from consideration. If a rule sequence occurs uniformly, for instance the compound `2 (1)` in the example above, then the individual rules in the compound are eliminated. Rules compositions are formed with substitutions copied from the generalized proof. This fact will become apparent in our next example drawn from constructing towers in the blocks world. Rules generated by Reconstruct are ordered such that recursive rules are placed after the base case rules, for each predicate. If there are multiple base (or recursive) cases, then they are ordered according to frequency of occurrence in the original production set[4]. The reconstruction procedure is very efficient as demonstrated by the following proposition.

[4]We can design an inductive procedure to learn the conditions to make a production set separable. We discuss this in a later section.



G_i is short for $G/\equiv_i$. Each G_i has an associated set of eliminable symbols.

Figure 7: The partially ordered space of proof grammars generated from the example in Figure 8

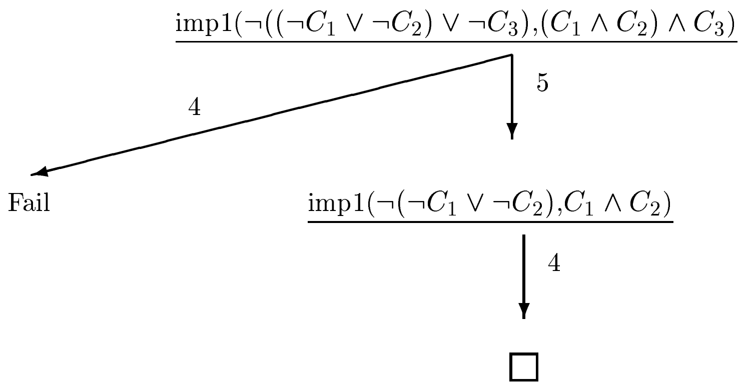


Figure 8: The minimized C-ordered SLD-tree

Proposition 6 *The reconstruction of (T_s, C) from its proof grammar encoding and the original pair (T, C) using the procedure in Table 4, takes $O(pd)$ where p is the number of productions in the proof grammar and d is the number of rules in T.*

procedure Reconstruct(G_N,T,S)
Inputs: a proof grammar G_N reduced with respect to $\equiv_N$, original theory T, and a set S of eliminable symbols.
Output: a specialization T_s of T that generates G_{red} defined below.
Method:
1. *Unfold*: Construct G_{red} by removing all eliminable symbols in G_N.
   Reconstruct-aux(G_{red},T) produces the rules in T_s.
2. *Order*: Order the productions of G_{red} by placing recursive rules after the base cases, for each predicate.
   If there are multiple base (or recursive) cases, order them according to frequency
   of occurrence in the original production set.

procedure Reconstruct-aux (G, T)
Inputs: a proof grammar $G = (N, A, P, S)$ with an ordered production set and the original domain theory T
Output: a specialization T_s of T such that T_s generates G.
Method:
Notation: Let r be a rule name. *head(r)* and *body(r)* denote the head and body literals of r.
$body_i(r)$ is the i^{th} literal of *body(r)*.
$X|_\theta$ applies substitution θ to X. $\theta_1 \cdot \theta_2$ is a substitution composition.
foreach production $X \longrightarrow r\alpha_1 X_1 \ldots \alpha_n X_n \in P$ do
generate $head(r)|_{\theta_1 \cdot \ldots \cdot \theta_n}$:- $lits(\text{compose}(\text{body}_1(r), \alpha_1)), \ldots, lits(\text{compose}(\text{body}_n(r), \alpha_n))$.
where $\alpha_i \in A^*$ and $\theta_i = bindings(\text{compose}(\text{body}_i(r), \alpha_i)$

procedure compose(Literal,RuleList)
Input: a literal and a list of rules.
Outputs: a list of literals and a binding list extracted by *lits* and *bindings*.
Method:
Construct a proof of Literal using rules in RuleList in sequence and
return list of literals established by d_f steps in the proof path, and the bindings generated by unifications.

Table 4: Reconstructing the domain theory from a proof grammar

3.3 The decision list organization of (T, C) pairs

In our running example, we start with the original solver `imp`. The grammar $Elim(G/\equiv_1, [A_2, A_5])$ is obtained by processing the example in Figure 6, and is reconstructed as the (T_s, C) pair with the `imp1` predicate. This derivation was shown in a previous subsection. We similarly learn a new (T, C) pair described by `imp2` that is specialized for solving goals of the form `imp(¬(¬ C1 ∨ (¬ C2 ∨ ¬ C3)),` X^f`)`.

```
6.  imp2(¬(¬ X1 ∨ ¬ X2), and(X1,X2)) :- wire(X1), wire(X2).
7.  imp2(¬(X1 ∨ ¬ X2), and(X1,Y2)) :-  wire(X1), imp2(¬X2,Y2).
```

We organize these solvers as a decision list [29]. For this example, the decision list has the form $((Cond_1, \texttt{imp1})(Cond_2, \texttt{imp2})(Cond_3, \texttt{imp}))$, where (T, C) pairs are abbreviated by their main predicates. Each $Cond_i$ is a conjunction of upto k features on a problem instance. The decision list is processed like a Lisp cond, so that each pair is processed until one of the $Cond_i$ is true. In this example, $Cond_3$ is the catch-all category `true`.

We now report experimental results that support the decision list organization. The experiments were conducted with learned values of $Cond_i$ (e.g., a test to check if the last and the first element of the expression to be implemented is a $\neg X$ where X is a wire). We sampled 100 problems from a distribution of circuits synthesizable using and-gates alone. These circuits were of depth 2 through 4. Each timing (in seconds) reported in the table below (for depth = 2) represents averaging over 10 different runs. Each row in the table stands for a mix of problems suited for `imp`, `imp1` and `imp2`. These results show that it there is a large class of problem mixes for which it is advantageous to use a combination solver instead of a single general purpose solver. Note that here the combination solver has the same coverage as the original solver.

Original	Combination	% `imp`	% `imp1`	% `imp2`
0.58	0.28	0	100	0
0.6	0.35	0	0	100
0.95	1.1	100	0	0
0.7	0.577	33.33	33.33	33.33
0.7	0.6	33.33	0	66.67
0.7	0.567	33.33	66.67	0
$\boxed{0.75}$	$\boxed{0.93}$	0.8	0.1	0.1

We observe one anomalous point (boxed in the table above) where the combination solver is outperformed by the original solver. A simple analytical calculation explains this point. We can derive the exact conditions on the problem mix for which the solvers organized as decision list perform worse than the original solver. Let p_1, p_2 and p_3 ($p_1+p_2+p_3 = 1$) be the percentage mixes of problems solvable by `imp1` alone, `imp2` alone and `imp`. Then, using the first three data points, we can set up the average cost of using the combination solver against the original one.

$$0.58p_1 + 0.6p_2 + 0.95p_3 > 0.28p_1 + 0.35p_2 + 1.1p_3$$

which simplifies to

$$0.45p_1 + 0.4p_2 > 0.15$$

This condition on the p_i's precisely delimits when the combination solver will outperform the original solver.

Now that we have outlined our target compilation architecture, we are ready to discus methods of merging proof grammars that ensure that good subsets of proof grammars are retained by the REBL procedure.

3.4 Merging sets of proof grammars from multiple examples

Given e_1 and e_2, we can construct $V(\{e_1, e_2\})$ in terms of $V(e_1)$ and $V(e_2)$. An obvious definition is the standard one of intersection adopted in inductive learning of conjunctive concepts: $V(\{e_1, e_2\}) = V(e_1) \cap V(e_2)$. This yields a version space of proof grammars that can solve *both* e_1 and e_2. However, this eliminates grammars that are specialized for e_1 and e_2 separately. From the point of view of maximizing $U(S, P)$, learning two distinct (T, C) pairs for e_1 and e_2

might be better than merging their solvers. Thus, we adopt the conservative position of defining $V(\{e_1, e_2\}) = V(e_1) \cup V(e_2)$. Taking arbitrary unions will not yield useful generalizations: a fact that is well-known in inductive concept learning. Therefore we need utility criteria for pruning elements from the union.

An example serves to illustrate the issues involved. Figure 9 shows the version space of proof grammars that results from the example `imp(¬((¬ c1 ∨ ¬ c2) ∨ (¬ c3 ∨ ¬ c4)),` X^f`)`. For this example, two proof grammars H_1 and H_2 are generated using the $\equiv_1$ and $\equiv_2$ relation. The lattice of $Elim(H_i, S)$ is also shown with each H_i. H_1 is shown below; its eliminable symbol set is $\{[A_5]\}$. H_2, shown below H_1, has the set $\{[A_1], [A_3], [A_5]\}$ of eliminable symbols.

$$
\begin{array}{lllll}
[A_0] & \longrightarrow & \mathtt{3} \quad [A_0] \quad [A_0] & ; [1,2] & (3) \\
[A_0] & \longrightarrow & \mathtt{2} \quad [A_5] & ; [1] & (4) \\
[A_5] & \longrightarrow & \mathtt{1} & ; [] & (4)
\end{array}
$$

$$
\begin{array}{lllll}
[A_0] & \longrightarrow & \mathtt{3} \quad [A_1] \quad [A_1] & ; [1,2] & (1) \\
[A_1] & \longrightarrow & \mathtt{3} \quad [A_3] \quad [A_3] & ; [1,2] & (2) \\
[A_3] & \longrightarrow & \mathtt{2} \quad [A_5] & ; [1] & (4) \\
[A_5] & \longrightarrow & \mathtt{1} & ; [] & (4)
\end{array}
$$

We can calculate the following relationships between the grammars in the two examples using the procedure in Table 3.

$$
\begin{array}{c}
G/\equiv_1 \leq H/\equiv_1 \\
G/\equiv_2 \leq G/\equiv_1 \\
H/\equiv_2 \leq H/\equiv_1
\end{array}
$$

That is, H_2 solves a subset of the problems solved by H_1 and G_1 solves a subset of the problems solved by H_1. G_2 and H_2 are incomparable.

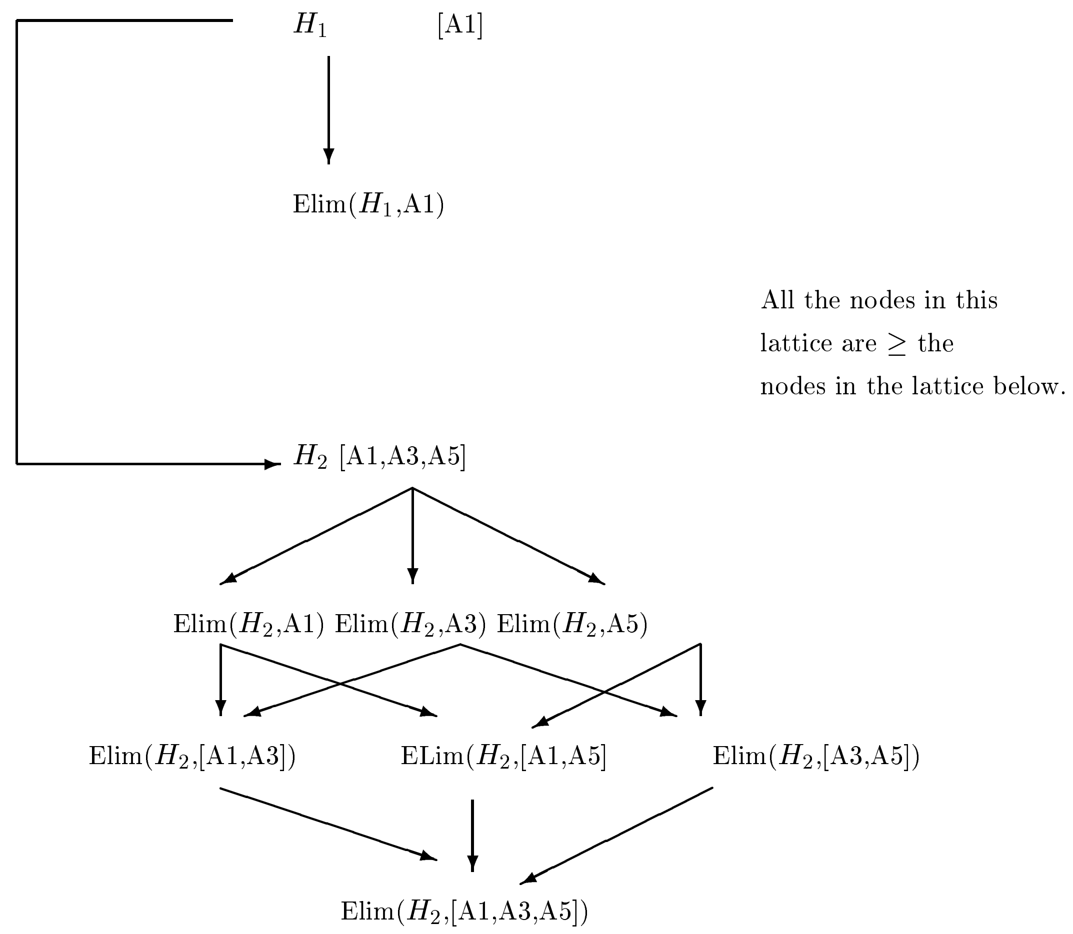
When a new example e is processed, the grammars generated by it are added to the set of grammars from previous examples E using,

$$V(E \cup \{e\}) = V(E) \cup V(e)$$

and the relationship between the newly added grammars with the existing ones is determined as shown in Figure 10. This merging procedure yields a new partially ordered set of proof grammars.

The combinatorics of the $V(e)$ generation lie solely in the consideration of $Elim(G_i, S_j)$ pairs. With the $\equiv_i$ criterion, the number of reduced grammars is at most equal to the depth of the proof tree. So we will follow a two-phase strategy for acquiring a set of proof grammars for covering the problem distribution efficiently. First, we find the reduced-branching-factor grammars (from among the ones generated by $\equiv_i$), and then we find the reduced-depth-grammars among them using the *Elim* construction.

In the first phase of generation each new example adds upto d linearly-ordered proof grammars (a chain), where d is the depth of the proof. An element i below element j in a chain solves a strict subset of the problems solved by j. We can associate an average success cost, an average failure cost, and a probability of success with each element in a set of proof grammars. The selection cost of picking an element is folded in with the success and failure cost associated with that node. In the next subsection we discuss methods of estimating these quantities reliably. Now we describe a pruning method for reducing the set of candidate proof grammars.



H_i is short for $H/\equiv_i$. Each H_i has an associated set of eliminable symbols.

Figure 9: The partially ordered space of proof grammars generated from a new example

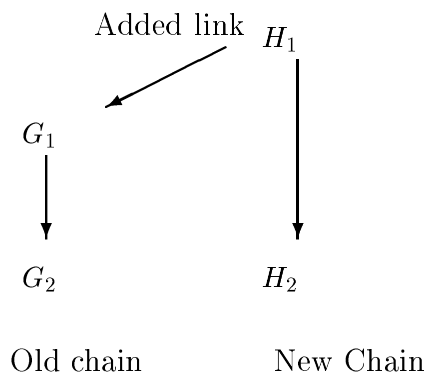


Figure 10: Merging two proof grammar sets

We can eliminate those elements from a chain for which loss in coverage is not compensated by an appropriate difference in average solution cost. We derive the pruning condition for two consecutive elements in a chain. Consider grammar G_1 with coverage p_1, and success and failure costs s_1 and f_1 respectively. Consider the subsumed grammar G_2 with coverage p_2, and costs s_2 and f_2 respectively. The question is: is it worthwhile to keep G_2? The average cost of the combination organized as a decision list G_2, G_1 should outweigh the average cost using G_1 alone. Note that the coverage of the combination solver is the same as that of the single solver G_1. The tradeoff on a chain is made purely on the grounds of efficiency. We set up the relevant cost inequality $E((G_2, G_1), P) > E(G_1, P)$ where P is the problem distribution.

$$\begin{aligned} E((G_2, G_1), P) &= p_2 s_2 + (p_1 - p_2)[f_2 + s_1] + (1 - p_1)[f_1 + f_2] \\ E(G_1, P) &= p_1 s_1 + (1 - p_1) f_1 \end{aligned}$$

We let $C_1 = p_1 s_1 + (1 - p_1) f_1$ and $C_2 = p_2 s_2 + (1 - p_2) f_2$ denote the average costs associated with G_1 and G_2 respectively. Then the inequality above reduces to:

$$\begin{aligned} C_2 + C_1 - p_2 s_1 &> C_1 \\ C_2 &> p_2 s_1 \\ \frac{p_2}{C_2} &< \frac{1}{s_1} \end{aligned}$$

We can generalize this condition and use it all the way up a chain starting at the bottom (the most specific proof grammar). This generalized pruning condition is:

$$\frac{p_j}{C_j} < \frac{1}{s_i} \text{ for all } G_j \leq G_i \tag{1}$$

Our implementation uses this condition to eliminate subsumed solvers in a chain. When chains are merged as shown in Figure 10, we prune elements along each path. At the end of this pruning step, the elements in a chain have the following property:

$$\frac{p_j}{C_j} \geq \frac{1}{s_i} \text{ for all } G_j \leq G_i$$

We can show that $\frac{1}{s_i} \geq \frac{p_i}{C_i}$ as follows:

$$\begin{aligned} \frac{1}{s_i} &\geq \frac{p_i}{p_i s_i + (1 - p_i) f_i} \\ p_i s_i + (1 - p_i) f_i &\geq p_i s_i \\ (1 - p_i) f_i &\geq 0 \end{aligned}$$

The last inequality holds because cost $f_i \geq 0$ and $(1 - p_i) \geq 0$ since p_i is a probability.

Proposition 7 *After the pruning of subsumed elements, the elements in a chain are in decreasing order of* $\phi(i) = \frac{p_i}{p_i s_i + (1 - p_i) f_i}$. *That is,* $\phi(j) \geq \phi(i)$ *for all* $G_j \leq G_i$

After merging and pruning, we have a partially ordered set S of proof grammars that needs to be converted into a total order. This total ordering can be used as a combination solver when the applicability conditions of the elements have not yet been acquired to construct the decision list organization discussed in a previous section. Here we develop a combination solver guided only the probabilities and costs of the individual solvers in S. We use the methods developed in [19] for this construction. This is because we impose no ordering on the quality of different solutions. All we seek is a solution because all solutions are considered to be equally good. We start with the simplest configuration for the set S and work our way up to more complex configurations. Suppose S consists of a set of chains all of length 1. Each element solves a disjoint class of problems in the distribution. Then the optimal order of the elements in S is in decreasing order of $\phi(i)$.

Proposition 8 *The optimal total order of elements in the partially ordered set S of proof grammars where $\forall s, t \in S,\ \ s \not\leq t \wedge t \not\leq s$ is in decreasing order of $\phi(i)$.*

The elements of S are incomparable to one another and have an associated probability of success p_i and success and failure cost s_i and f_i for problems in the given distribution P. Let a, b be elements of S. Then the expected costs of the order ab and ba are given by the equations below.

$$\begin{aligned} E(ab, P) - E(ba, P) \quad = \quad & p_a s_a + (1 - p_a) p_b [f_a + s_b] + (1 - p_a)(1 - p_b)[f_a + f_b] \\ & - p_b s_b - (1 - p_b) p_a [f_b + s_a] + (1 - p_a)(1 - p_b)[f_a + f_b] \end{aligned}$$

Then a precedes b in an optimal order if:

$$E(ab, P) < E(ba, P) \text{ iff } \frac{p_a}{p_a s_a + (1 - p_a) f_a} > \frac{p_b}{p_b s_b + (1 - p_b) f_b}$$

So we locally swap a and b to decrease the expected cost of a sequence of which ab is a part. When no more swaps are possible, the elements are ordered by the ratio derived and the expected cost of the sequence is minimized.

A somewhat more complex organization of S has it consist of a set of disjoint chains of length greater than or equal to 1. An *admissible* total ordering of elements in S respects the chain constraint. That is, if $s \leq t$ then t does not precede s in an admissible ordering, for all $s, t \in S$. We assume that each chain has been pruned according to Condition 1. This condition ensures that every element in the chain is useful. The pruning condition 1 guarantees that $\phi(i)$ decreases up a chain as demonstrated in Proposition 7. An optimal total ordering (one that minimizes expected cost over P) is described by the following proposition.

Proposition 9 *An optimal total ordering of S, where S is a set of disjoint chains, consists of an admissible ordering with elements sorted by decreasing values of ϕ.*

The same result holds when S is a directed acyclic graph. The cost of a total ordering can still be decreased by swapping elements that are not in decreasing order of ϕ as shown before.

Proposition 10 *An optimal ordering on S consists of an admissible ordering with elements sorted in decreasing order of ϕ.*

Randomized strategies There are many possible problem-solving regimes that we could have used. The total order among solvers discussed before is simply one among them. Another interesting organization uses a randomized strategy for picking an element from the partially ordered set of proof grammars. An example serves to illustrate this. Consider again grammar G_1 with coverage p_1, and success and failure costs s_1 and f_1 respectively, and the subsumed grammar G_2 with coverage p_2, and costs s_2 and f_2 respectively. We compare using the solvers in the order G_1G_2 which is chosen by the total order regime above versus the randomized procedure that picks G_1 or G_2 ensuring on the average that G_2 gets picked with probability p_2 and G_1 gets picked with probability $p_1 - p_2$. The cost comparison between the randomized solver and the total order solver is:

$$\begin{aligned}
E(Random(G_1, G_2), P) &= p_2 s_2 + (p_1 - p_2)s_1 + (1 - p_1)f_1 \\
E((G_2, G_1), P) &= p_2 s_2 + (p_1 - p_2)[f_2 + s_1] + (1 - p_1)[f_1 + f_2] \\
Cov((G_2, G_1), P) &= p_1 \\
Cov(Random(G_1, G_2), P) &= p_2 p_2 + (p_1 - p_2)p_1
\end{aligned}$$

It is clear that $E(Random(G_1, G_2), P) < E((G_2, G_1), P)$ since all costs are positive. However, there is a tradeoff in coverage. The randomized solver may not return a solution every time the decision list solver returns one. Explicit utility criteria (e.g., the Cobb-Douglas form discussed in the formulation of the speedup learning problem) can be used for trading off coverage and utility to calculate the best combination solver for a given distribution P.

3.5 Learning problem distributions: multinomial estimation

There are two learning problems: one is to acquire the probability of success p_i, and the success and failure cost s_i and f_i of each element in S. Each element in S corresponds to a partition of the problem distribution. We need to ensure that at least one instance of all existing partitions in the distribution have been observed. This constraint, together with the constraint of estimating p_i, s_i and f_i to within the error parameters ϵ and δ specified by the user, imposes lower bounds on the number of samples to be obtained from the problem distribution. The other learning problem is to acquire $Cond_i$ which are predicates on the problem instance that determine the applicability condition of each element in S. We only discuss the first learning problems in this paper.

Our algorithm treats the problem distribution as a multinomial distribution with a finite number k of problem types with a probability $p_i, 1 \leq i \leq k$ of occurrence. The number of classes k needs to be estimated along with the occurrence probabilities. An upper bound on k is provided by the user. Thus only k of the solvers in S will survive at the end.

Before we characterize sample complexity of multinomial estimation, we provide a simple example. Consider an urn with 20 red balls, 30 white balls and 50 blue balls. Suppose 10 balls are selected at random one at a time from the urn, with replacement. What is the probability that at least one colour will be missing from the 10 selected balls? The probability that no red ball occurs in the sample is 0.8^{10}. Similarly, the probability that no white ball occurs is 0.7^{10} and the probability that no blue ball occurs is 0.5^{10}. The union of these three event spaces counts the event that all balls are red twice (once because they are non-white, and again because they are non-blue). So the probability that at least one colour will be missing from the 10 selected balls is: $[0.8^{10} + 0.7^{10} + 0.5^{10}] - [0.2^{10} + 0.3^{10} + 0.5^{10}]$, which is about .135. The probability that no colour will be missing is therefore 0.865. With 20 samples, the probability that no colour is missing is 0.99.

If we have k classes and if the probabilities of the individual classes are $p_i, 1 \leq i \leq k$, then the sample size N needed to ensure that we observe at least one element from each of the k classes is such that $1 - \sum_{i=1}^{k}[(1-p_i)^N - p_i^N] \leq \epsilon_{est}$. ϵ_{est} is the probability that no class is missing in the sample. Equivalently, we need

$$\sum_{i=1}^{k}[(1-p_i)^N - p_i^N] \leq 1 - \epsilon_{est} \tag{2}$$

We make the assumption that a class with probability less than 1% is pruned from S. Thus the largest value that the sum can take is $(0.99)^N$. Then we see that for $\epsilon_{est} = 0.95$, we need $N = 300$. In general, we need $N \geq \frac{ln(1-\epsilon_{est})}{ln .99}$. For $\epsilon_{est} = 0.99$, the sample size is 458. So, the sample size necessitated by the constraint of observing at least one instance of a class with probability greater than 1% is fairly low.

We now calculate the sample size needed to estimate the parameter p_i to within ϵ_{cov} of its true value with probability greater than $1 - \delta_{cov}$, and costs s_i and f_i to within ϵ_{cost} of their true values with probability greater than $1 - \delta_{cost}$. First we discuss the estimation of p_i's. We use a proposition of Bollabas cited in [16]. This is derived from using the method of Chernoff and Hoeffding to bound the tail probabilities of the sum of independent and identically distributed random variables.

Proposition 11 (Bollabas) *If N independent experiments are performed to obtain an estimate $\hat{p}$ of a probability p, then*

$$Pr(\mid \hat{p} - p \mid > \epsilon_{cov}) \leq 2e^{-2N{\epsilon_{cov}}^2}$$

where $0 < \epsilon_{cov} < 1$.

This proposition yields the corrollary that the sample size needed to estimate the probability p_i that an element of S solves a problem in the distribution is:

$$N \geq \frac{1}{2\epsilon_{cov}^2} ln \frac{2}{\delta_{cov}}$$

Note the sample size is polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$. Now we turn to estimating costs. There are two approaches to cost estimation in the literature. The first makes no assumptions about the underlying distribution of the quantities. The value is estimated as an average over a number N of independently and randomly drawn samples of the quantity. Greiner introduced this method for cost estimation [16] in the context of inference in non-recursive theories and [36] extended it to cover inference in recursive domain theories. The relevant sample complexity result is stated here without proof; the reader is invited to look at the above references for the details of their derivation.

Proposition 12 *To estimate the success and failure cost s and f of a (T, C) pair to within ϵ_{cost} with probability greater than $1 - \delta_{cost}$, we need a sample size N such that*

$$N \geq \frac{U^2}{2\epsilon_{cost}^2} ln \frac{2}{\delta_{cost}}$$

where U is an upper bound on the cost.

4 REBL and its analysis

Now we are ready to put together the algorithm REBL with all the components described in the previous sections. We know that each example generates a chain of new solvers ordered by $\equiv_i$. These can be pruned by condition 1 once we estimate the quantities p_i, s_i and f_i. The algorithm checks that improvements in the utility metric $U(S, P)$ are made with each addition, and stops when it achieves a local maximum. Upto k of these solvers survive at the end corresponding to a maximum of k partitions of the problem distribution. The final solver is a total ordering on the partially ordered set of solvers ordered by values of ϕ.

procedure REBL
    S = (T_0, C_o)
    Phase 1: generate S
        repeat
          1. Get e from distribution.
          2. Generate V(e) using $\equiv_i$, $i \geq 1$.
          3. NewS = merge(S,V(e)).
          4. Estimate p_i, s_i and f_i for every element in NewS.
          5. Prune NewS using condition 1.
          6. if better(NewS,S) then S = NewS.
        until stopping criterion.
    Phase 2: generate totally ordered list from S
        Sort elements s in S on $\phi(s)$ in decreasing order according to Proposition 9.

We explain each of the component steps in the algorithm in more detail. The *merge* step computes the union of S and $V(e)$ and establishes the ordering relationships between elements in S and the newly added elements in $V(e)$. The estimation of costs and probabilities in step 4 is governed by the sample complexity result in Proposition 11 and 12 and ensures that the quantities are estimated within user-specified error and confidence bounds.[5] Useless elements in NewS are pruned using condition 1. This step eliminates subsumed solvers for which the loss of coverage is not compensated for by an adequate reduction of average cost. Before we commit to NewS as the new set S we check that it does improve the utility metric over the previous candidate S. The stopping criterion (condition 2) ensures that we observe enough samples to observe all the partitions in the problem distribution and that we are at a local optimum with respect to U.

All that remains to be discussed is the criterion used for checking whether NewS is better than S with respect to the problem distribution. Theoretically, we can calculate $U(S, P)$ and $U(NewS, P)$ using the probability and cost estimates computed in step 4. This will necessitate making independence assumptions that may or may not hold between the elements in S. So, we conduct an empirical test of the hypothesis that $U(NewS, P) - U(S, P) > 0$ using the signed-rank test of Wilcoxson, as suggested by [11] and [5]. First, we sample N problems from the distribution and get pairs of observations (u_1, u_2) corresponding to the utility of solving each problem by NewS and S. We then compute the differences between the u's for each problem and rank the N elements according to the absolute value of the differences sorted from least to the most. We then associate a sign with each rank r_i based on whether the difference in utility was positive or negative. The statistic S_N is calculated as the sum of the signed ranks over the N samples. If NewS is better

[5]It should be noted here that in the implementation of REBL these sample sizes are used as upper bounds. When the difference between successive values of the quantities is within a threshold we stop sampling. In our experiments we have never needed more than about 200 samples of estimating cost and probabilities.

than S we will expect $S_N > 0$. For sufficiently large values of N, $Z_N = \frac{S_N - E(S_N)}{\sigma(S_N)}$ is a standard normal distribution with mean 0 and variance 1. We can estimate the sample size for which we can treat Z_N as a normal distribution using Chebyshev's inequality. This turns out to be quadratic in $\frac{1}{\epsilon}$ where ϵ is the relative error we are willing to tolerate. The test for our null hypothesis $(S_N > 0)$ at significance level δ requires that we reject it if $Z_N > a$ where $\Phi(a) = \delta$, where Φ is the standard normal distribution. δ is the significance level associated with the test.

We now analyze procedure REBL to understand the types of theoretical guarantees we can provide on its performance. The construction of the specialized solvers (step 2) is polynomial in the size of the C-ordered SLD-trees. Each specialized solver provably eliminates repeated meta-level computation with respect to the original (T, C) pair on a subclass of goals in the distribution. The merging step is polynomial in the size of S and $V(e)$. It is also polynomial in the size of each proof grammar in $S \cup V(e)$ as shown in Proposition 2. The sample complexity of the computation of coverage and cost, in step 4, for each specialized solver is polynomial in the product of $\frac{1}{\epsilon_{cost}}$ and $\frac{1}{\delta_{cost}}$ and $\frac{1}{\epsilon_{cov}}$ and $\frac{1}{\delta_{cov}}$. The test for checking that NewS is better than S also requires sample size polynomial in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$, the error and significance level in the Wilcoxson signed-rank test. Problem mix information is gathered by REBL dynamically. The sample size required to identify the partitions of the problem distribution is given by condition 2 and the test for local optimum only requires that $U(NewS, P) < U(S, P)$, which is established in the test on step for 6. Phase 2 of REBL is linear in the size of S. The standard decision-list learning method (e.g. Rivest) associates problem properties with specialized solvers in time polynomial in given error and confidence bounds. So all the individual steps are polynomial in the sizes of error and confidence bounds, the C-ordered SLD-trees and the proof grammars.

Non-Theorem 1 *With probability $1-\delta$, REBL finds a subset of solvers organized as a decision list such that $|U(S,P) - U(S_{opt}, P)| \leq \epsilon$, using $O(\frac{1}{\epsilon})$ and $O(\frac{1}{\delta})$ examples drawn from the distribution P, for user-specified error and confidence parameters $\epsilon \geq 0$ and $0 \leq \delta < 1$. S_{opt} is the best decision-list solver for (T_o, C_o) and P.*

This theorem is hard to establish under very general conditions. This is because identification of S_{opt} itself is impossible since the problem of deciding that two solvers have the same coverage is undecidable. This restricts us to considering spaces where the equivalence condition is decidable. The proof grammar representation was chosen to explore the decidable subset of this space. The problem of finding globally optimal S's has already been established to be NP-complete for the case that the space of candidate S's is finite. The non-theorem can be interpreted as saying that the speedup learning problem has a fully polynomial approximation scheme. When the set of candidate S's is finite, this problem reduces to the knapsack problem. Using methods developed by Ibarra and Kim, we can efficiently find solutions that are within a factor of two off the optimal solution. No fully-polynomial time approximable solutions can is known even this restricted subset of the problem.

With REBL, all we can show is that the method finds local optima in the utility space. Since we start with (T_o, C_o) and each S is no worse than the previous, we are guaranteed to find a final S that is no worse than the starting element. We thus settle for a much weaker local improvement guarantee in the style of [3, 17].

Theorem 1 *With probability $1 - \delta$, REBL finds a subset S of solvers organized as a decision list such that $U(S, P) \geq U((T_o, C_o), P)$, after observing a number of examples polynomial in $\frac{1}{\epsilon_{cov}}$, $\frac{1}{\epsilon_{cost}}$, $\frac{1}{\epsilon_{est}}$, $\frac{1}{\delta_{cov}}$ and $\frac{1}{\delta_{cost}}$, $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$, drawn from an unknown, stationary problem distribution P.*

REBL is not distribution-specific, but it is sensitive to the distribution because it learns solvers adapted for a given distribution. Note that each step in REBL involves polynomial computation except for steps requiring solution of problems drawn from P. We now proceed to describe more classes of elimination of irrelevant meta-level computation before turning to an experimental evaluation of REBL.

5 More types of irrelevant computation in C-ordered SLD-trees

Consider a logical encoding of a blocks world problem in the situation calculus. It is a theory with function symbols which raises interesting issues in analysis. This theory constructs towers of a height within given bounds from an existing set of blocks originally all on the table. A restriction of this problem is equivalent to the subset sum problem, known to NP-complete. `tower(Top,Hmin,Hmax,S)` checks if the height of the block `Top` is between the bounds `Hmin` and `Hmax` in a situation `S` reachable from the initial state `s0` by the operator `transfer`.

```
t1. tower(Top,Hmin,Hmax,S,Moves) :-
       reachable(S,s0,Moves),clear(Top,S),
       towerheight(Top,S,H), H >= Hmin, H =< Hmax.
```

The predicate `reachable` is the generator of states, and `clear` and `height` are the test predicates. We now describe the generator. The state `s0` is reachable from itself by 0 moves. The state `do(transfer(X,Y),S)` is reachable from state `S1` with `Moves` number of transfer actions, provided `S` is reachable from `S1` by `Moves - 1` actions and that `X` and `Y` are clear blocks in the situation `S`.

```
r1. reachable(s0,s0,0).
r2. reachable(do(transfer(X,Y),S),S1,Moves) :-
       Moves > 0, Moves1 is Moves-1,
       reachable(S,S1,Moves1),
       block(X), block(Y), X\== Y,
       clear(X,S), clear(Y,S).
```

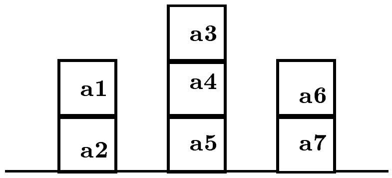
In the initial state all blocks are on the table, for our problem distribution. The states generated by `reachable` consist of a set of towers. A typical state is the following:

$$\texttt{(do (transfer(}X_1\texttt{,}X_2\texttt{), do(transfer(}X_3\texttt{,}X_4\texttt{), } \ldots\texttt{, do(transfer(}X_{n-1}\texttt{,}X_n\texttt{),s0)))),}$$

$X_i \neq X_{i+1}$ for $1 \leq i \leq n$, $n > 0$. Since X_{2i} can be distinct from X_{2i+1}, where $1 \leq i \leq n/2$, the state described corresponds to a set of towers. There are more constraints between the X_i's which are determined by the `clear` predicate – which we defer for now. For a world with N blocks constructed with d successive transfer actions there are $O(N^d)$ situations generated by the predicate `reachable`. An example state over 7 blocks is shown in Figure 11. It is important to note that the state descriptions are not *canonical* – that is, there is a many-to-one mapping between state descriptions and states. This is because a given set of towers can be constructed by a several distinct sequences of `transfer` actions. For instance, the two states below are treated as distinct by the generator.

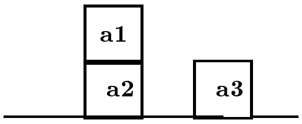
```
do(transfer(a1,a2),do(transfer(a1,a3),s0))
          do(transfer(a1,a2),s0)
```

However, they both represent the situation shown in Figure 12. Eliminating the generation of syntactically different state descriptions that denote identical states improves the efficiency of the generator. We will present the `clear` and `towerheight` predicates before we discuss methods for eliminating this source of inefficiency.



do(transfer(a1,a2),do(transfer(a3,a4),do(transfer(a4,a5),do(transfer(a6,a7),s0))))

Figure 11: A state generated by the predicate reachable



do(transfer(a1,a2),s0)
do(transfer(a1,a2),do(transfer(a1,a3),s0))

Figure 12: Non-canonicality of state descriptions

Once a state is generated, we test whether a named block is the top of a tower and check if its height is within the given bounds. To locate the tower that the named block is top of, we use `clear(Top,S)` with both arguments instantiated. A block X is clear if (1) there is no transfer action in the state description sequence that moves a block atop X (2) if there is a transfer action that moves Y atop X, there is a later transfer action in the sequence that moves Y atop some other block and thus away from X.

The predicate `clear_aux` reads the state description sequence from left to right in reverse order of execution[6]. The first two `clear_aux` rules are base case rules. All blocks are clear in the situation `s0`. A block moved atop another is clear by the constraints of the `transfer` action. As a transfer action `transfer(X,Y)` is processed, the block X is stored on a list of moved blocks: this is what the last `clear_aux` rule accomplishes. The third `clear_aux` rule implements (2) above by declaring that the block `X` is clear, if the block atop it has been moved away from it. The four rules are mutually exclusive, and there is no backtracking in this computation, when both arguments of `clear` are bound. The time taken to check that a block is clear is thus linear in the length of the state description sequence. So the `clear` computation, unlike the `reachable` computation whose time complexity is exponential in the length of the state description, is efficient.

```
c1. clear(X,S) :- clear_aux(X,S,[]).
c2. clear_aux(X,s0,_).
c3. clear_aux(X,do(transfer(X,Y),S),M).
c4. clear_aux(X,do(transfer(A,X),S),MovedBlocks) :-
```

[6] Thus transfer actions that occur later in time are processed before transfer actions that occur earlier.

```
    member(A,MovedBlocks).
c5. clear_aux(A,do(transfer(X,Y),S),MovedBlocks) :-
    A \== X, A\== Y, clear_aux(A,S,[X|MovedBlocks]).
```

The height of the tower whose top block is `X` is determined by the predicate `towerheight` defined below. The height of a block on the table is simply its intrinsic height. A block of intrinsic height h placed atop block `Y` is at a height h above the height of the tower whose top block is `Y`. The last `towerheight` rule is a frame axiom. The height computation is specified by three mutually exclusive rules, and the time taken to obtain the height of a block is linear in the length of the state description sequence. A speedup learner needs to be able to perform quick-and-dirty complexity analyses such as this one to determine where to focus its efforts to improve the computation.

```
towerheight(X,s0,H) :-
    height(X,H).
towerheight(X,do(transfer(X,Y),S),H) :-
    height(X,H1),
    towerheight(Y,S,H2),
    H is H1+H2.
towerheight(A,do(transfer(X,Y),S),H) :-
    A \== X, A \== Y,
    towerheight(A,S,H).
```

We now examine a sample proof of a `tower` goal in this theory. Figure 13 shows the proof of `tower(a1,10,15,S,2)`. `S` is instantiated to be `do(transfer(a1,a4),do(transfer(a4,a2),s0)`.

While `reachable` produces a set of towers, the tester predicates `clear` and `height`, only test the height of the tower whose top block is a named block. Thus only one tower is relevant. Conceptually, we need to modify the generator so that it returns a single tower with the named block as its top. How can we modify the theory to achieve this reduction? We need to maintain the top block of the tower under construction, and ensure that the next transfer action moves a clear block atop this top block. We show how this can be generalized from a successful tower computation using the proof grammar minimization scheme introduced in the previous section. From Figure 14, we (and the grammar constructor) note that in all successful paths in the C-ordered SLD-tree, the first `clear` subgoal in the `reachable` computation is reduced by the sequence `c1` whereas the second `clear` subgoal is reduced by `c1c5`*`c2`. Now we need to compile this sequence into the theory. There are several choices. We can modify T or C to achieve this. We have chosen to modify T in this case. A subset of the modified theory generated by procedure Reconstruct is shown below. This is the theory produced by reduction via $\equiv_1$.

```
tower1(Top,Hmin,Hmax,do(transfer(Top,X),S),Moves) :-
     reachable1(do(transfer(Top,X),S),s0,Moves),
     towerheight1(Top,do(transfer(Top,X),S),H),
     H >= Hmin, H <= Hmax.

reachable1(do(transfer(X,Y),s0),s0,1) :-
     block(X), block(Y), X\==Y, clearx1(X,s0), cleary1(Y,s0).
reachable1(do(transfer(X,Y),S),S1,Moves):-
     Moves>0, Moves1 is Moves-1,
     reachable2(S,S1,Moves),block(X),block(Y),X\==Y,
```

tower(a1,10,15,S,2)

$\downarrow$ **t1**

reachable(S,s0,2), clear(a1,S), towerheight(a1,S,H), H≥10,H≤15

$\downarrow$ **r2**

reachable(S1,s0,1), clear(a1,S1), clear(a4,S1), a1≠a4,
clear(a1,S), towerheight(a1,S,H), H≥10,H≤15

$\downarrow$ **r2**

reachable(S2,s0,0), clear(a4,S2), clear(a2,S2), a4≠a2,
clear(a1,S1), clear(a4,S1), a1≠a4,
clear(a1,S), towerheight(a1,S,H), H≥10,H≤15

$\downarrow$ **r1,d,d,d**

clear(a1,S1), clear(a4,S1), a1≠a4,
clear(a1,S), towerheight(a1,S,H), H≥10,H≤15

$\downarrow$ **clp-1**

clear(a4,S1), a1≠a4,
clear(a1,S), towerheight(a1,S,H), H≥10,H≤15

$\downarrow$ **clp-2 , d**

clear(a1,S), towerheight(a1,S,H), H≥10,H≤15

$\downarrow$ **clp-3**

towerheight(a1,S,H), H≥10,H≤15

$\downarrow$ **th, d, d**

□

S = do(transfer(a1,a4),do(transfer(a4,a2),s0))
S1 = do(transfer(a4,a2),s0)
S2 = s0
clp-1, **clp-2**, **clp-3** and **th** are rule sequences

Figure 13: C-ordered SLD-tree for a `tower` computation

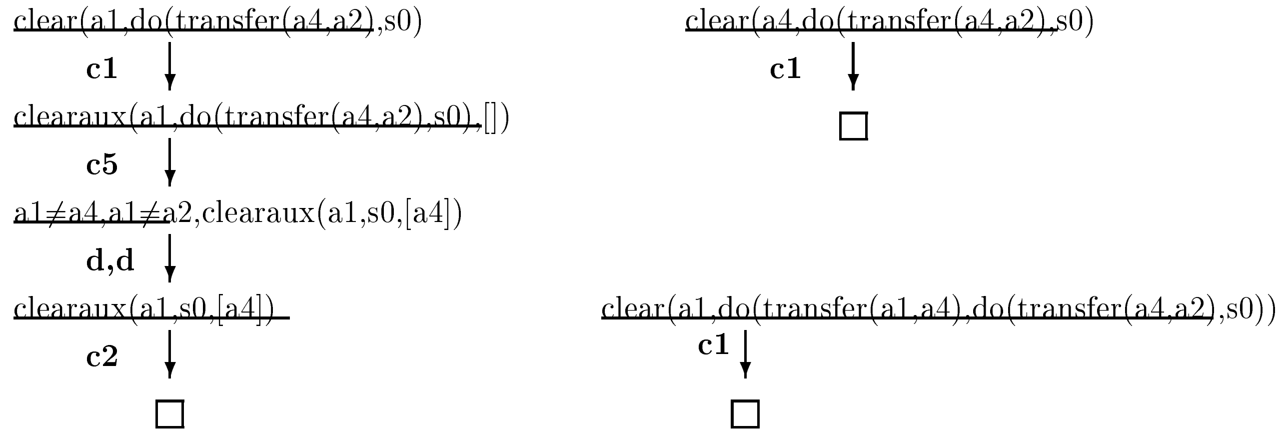


Figure 14: C-ordered SLD-tree for various `clear` computations

```
    clearx1(X,S,[]),cleary1(Y,S,[]).

clearx1(X,s0).
clearx1(X,S) :- clearaux_x1(X,S,[]).

clearaux_x1(A,do(transfer(X,Y),s0),M) :- A\== X, A\==Y.
clearaux_x1(A,do(transfer(X,Y),S),M):- A\==X, A\==Y, clearaux_x1(A,S,[X|M]).

cleary1(X,s0).
cleary1(X,do(transfer(X,Y),S)).

towerheight1(X,do(transfer(X,Y),s0),H) :-
    height(X,H1), height(Y,H2), H is H1+H2.
towerheight1(X,do(transfer(X,Y),S),H) :-
    height(X,H1), towerheight1(Y,H2), H is H1+H2.
```

There are several interesting things to notice about this theory.

1. The specialization of `clearx1` and `cleary1` embodies the sequence of rule firings in a successful tower computation. This rule sequence ensures that the generator `reachable` produces single towers. This solves the problem of non-canonicality of state descriptions. The gain obtained from this modification is considerable. Since there are $N_{P_1} + N_{P_3} + \ldots + N_{P_N} = O(N!)$ such states with a single tower, we have reduced the generator's complexity from $O(N^d)$ with d potentially unbounded, to $O(N!)$.

2. Note that the theory is also tuned to producing a single tower with the named block on top. This is a constant factor improvement, leading to a generator that has complexity $O((N-1)!)$.

3. Note that the `clear` test in the `tower` computation has been identified as redundant and has been eliminated.

4. This theory can generate towers of height greater than 1. If the distribution contains tower construction queries of height greater than 1, then this theory has 100% coverage and is more efficient than the original theory.

5. This theory has minimized rule choice computation, but not all base-level irrelevancies have been eliminated. For instance, in the first `reachable1` rule, the `clearx1` and `cleary1` tests can be eliminated.

Reduction via $\equiv_2$ generates the same theory, except for the predicate `reachable` produced below. Note that the irrelevant `clearx1` and `cleary1` tests on the state `s0` have been compiled away. Additionally, because the lookahead depth was increased to 2, we generate the sequence of transfer actions of `X` through `Y` to `Z`. This provides another constant-factor improvement, which allows us to solve larger problems within the specified time limits of a 1000 cpu seconds. Reductions via $\equiv_N, N > 2$ cause further unfoldings to be suggested. Note that the binding patterns have been preserved. Merely unfolding `reachable` twice will not generate the second rule below. $\equiv_1$ never unfolds a recursive rule. $\equiv_N$ can unfold it upto $N-1$ steps.

```
reachable2(do(transfer(X,Y),s0),s0,1):-
     block(X),block(Y),X\==Y.

reachable2(do(transfer(X,Y),do(transfer(Y,Z),S)),S1,Moves):-
Moves>0, Moves1 is Moves-1,
reachable2(do(transfer(Y,Z),S),S1,Moves1),
block(X),block(Y),X\==Y,clearx2(X,do(transfer(Y,Z),S)).

reachable2(do(transfer(X,Y),S),S1,Moves):-
     Moves>0, Moves1 is Moves-1,
     reachable2(S,S1,Moves),block(X),block(Y),X\==Y,
     clearx1(X,S,[]),cleary1(Y,S,[]).
```

The syntactic changes made to the code consist of performing unfoldings, deletions and substitutions in both cases. We now have a method of generating new theories using the Reconstruct procedure with well-defined unfolding, ordering and substitution policies.

A point to note now is that a tower consisting of blocks `a1`, `a3`, and `a5` in that order has the same height as a tower consisting of `a1`, `a5` and `a3`. Thus, if the first state description *failed* to satisfy the tower height restriction, the second will not meet it either. The two descriptions below are equivalent from the standpoint of `towerheight`. We need to restore canonicality in the generator with respect to this new equivalence relation on states.

```
do(transfer(a1,a3),do(transfer(a3,a5),s0)).
do(transfer(a1,a5),do(transfer(a5,a3),s0)).
```

Since there are $N_{C_1} + N_{C_3} + \ldots + N_{C_N} = 2^N - 1$ states with a single tower where block order is irrelevant, this step reduces the generator's complexity from $O((N-1)!)$ to $O(2^N)$. This wasted computation is a result of *order irrelevance.* This is a special but commonly-occurring form of base-level irrelevance in a C-ordered SLD-tree and can be detected by gathering failed paths as shown in Figure 15. Observe that the two failed paths, fail at the same subgoal and are identical at the level of rule sequencing. At the level of rules alone, it appears to be a case of base level repetition of computation as in the fibonacci computation. However, the `tower` predicate is not functional, as fibonacci was. For non-functional predicates, we examine binding patterns and determine that the two failed `tower` computations examine the state sequences `do(transfer(a1,a2),do(transfer(a2,a3),s0))` and `do(transfer(a1,a3),do(transfer(a3,a2),s0))`.

Definition 13 *Any variable annotated by f in any term in a Horn clause is a* generate-and-test *variable, if it is non-functional (has several possible legal bindings).*

The variable S in the `tower` predicate is a generate-and-test variable. All other variables in that predicate are bound at the start of the computation.

Definition 14 *We will say that two instantiations of a generate-and-test variable are* permutations *if one of them can be generated from the other by re-ordering binding decisions along a path in a C-ordered SLD-tree.*

The two instantiations of S `do(transfer(a1,a2),do(transfer(a2,a3),s0))` and `do(transfer(a1,a3),do(transfer(a3,a2),s0))` are permutations by this definition. Now we are ready to define order irrelevance. The two instantiations are order irrelevant because they are equivalent with respect to the `towerheight` predicate. They yield the same value for the unbound arguments of the functional `towerheight` computation. We can detect this from the C-ordered SLD-tree, because both paths fail by evaluation of $11 \geq 15$. To automatically determine the predicate with respect to which two instantiations are order irrelevant, requires solving a complex credit-assignment problem. Our implementation is guided by the user in its choice of this predicate.

Definition 15 *There is order irrelevance in a C-ordered SLD-tree if there are two paths from the root to a leaf, such that the paths instantiate the same generate-and-test variable and these instantiations are permutations.*

How robust is this definition of order irrelevance? Strictly speaking, the definition makes an inductive leap from the occurrence of two permutations of the bindings that all other permutations must behave identically. We can construct examples where order irrelevance pertains to a subset of the permutations and not to the entire set. The repair procedure below extends the theory inductively and may cause it provide incorrect answers on future problems.

We now demonstrate the canonical repair procedure for order irrelevance in the context of the tower computation. To enforce order irrelevance, we have the generator pick blocks in a fixed order. This requires sorting the blocks by their names. We then introduce a predicate `before` that tests the precedence of blocks. We modify the `reachable` predicate to choose blocks according to this ordering. The only change is the addition of a `before` test right after the bindings of the blocks are generated. By this scheme, the first `transfer` sequence above will be generated (assuming sorting in increasing alphanumeric name order), but not the second.

```
blocks([a1,a2,a3,a4,a5,a6,a7,a8,a9,a10]).   % a sorted list of
                                            % blocks in the universe.

before(X,Y) :- blocks(B), before1(X,Y,B).
before1(X,Y,[X|S]) :- member(Y,S).
before1(X,Y,[Z|S]) :- X \== Z, before1(X,Y,S).
```

tower(a1,S1,15,20,2)

↓

~~reachable(S1,s0,2),~~clear(a1,S1),towerheight(a1,S1,H),
H ≥ 15, H ≤ 20

↓ **r2**

~~reachable(S2,s0,1),~~block(a1),block(a2),a1≡a2,clear(a1,S2),
clear(a2,S2), clear(a1,S1),towerheight(a1,S1,H),
H≥15, H≤20

↓ **r2**

~~reachable(s0,s0,0),block(a2),block(a3),a1≡a3,clear(a2,s0),~~
~~clear(a1,s0)~~,clear(a1,S2),clear(a2,S2),clear(a1,S1),
towerheight(a1,S1,H),H≥15,H≤20

↓ **r1,d,d,d,d,d**

~~clear(a1,S2),clear(a2,S2),clear(a1,S1),~~
towerheight(a1,S1,H), H≥15, H≤20

↓ **clp1,clp2,clp1**

~~towerheight(a1,S1,H),~~H≥15,H≤20

↓ **th2**

~~height(a1,4),~~towerheight(a2,S2,H1),H=4 + H1,
H≥15,H≤20

↓ **d**

~~towerheight(a2,S2,H1),~~H=4+H1,H≥15,H≤20

↓ **th2**

~~height(a2,5),towerheight(a3,s0,H2),H1=5+H2,H=4+H1,~~
H≥15,H≤20

↓ **d,th1,d,d**

~~11≥15,~~, 11≤20

↓ **d**

FAIL

tower(a1,S3,15,20,2)

↓

~~reachable(s3,s0,2),~~clear(a1,S3),towerheight(a1,S3,H),
H ≥ 15, H ≤ 20

↓ **r2**

~~reachable(S4,s0,1),~~block(a1),block(a3),a1≡a3,clear(a1,S4),
clear(a3,S4),clear(a1,S3),towerheight(A1,S3,H),
H≥15, H≤20

↓ **r2**

~~reachable(s0,s0,0),block(a3),block(a2),a3≡a2,clear(a2,s0),~~
~~clear(a3,s0),~~clear(a1,S4),clear(a3,S4),clear(a1,S3),
towerheight(a1,S3,H),H≥15,H≤20

↓ **r1,d,d,d,d,d**

~~clear(a1,S4),clear(a3,S4),clear(a1,S3),~~
towerheight(a1,S3,H),H≥15,H≤20

↓ **clp1,clp2,clp1**

~~towerheight(a1,S3,H),~~H≥15,H≤20

↓ **th2**

~~height(a1,4),~~towerheight(a3,S4,H1),H=4+H1,
H≥15,H≤20

↓ **d**

~~towerheight(a3,S4,H1),~~H=4+H1,H≥15,H≤20

↓ **th2**

~~height(a3,2),towerheight(a2,s0,H2),H1=2+H2,H=4+H1,~~
H≥15,H≤20

↓ **d,th1,d,d**

~~11≥15,~~11≤20

↓ **d**

FAIL

Figure 15: Two failed paths in the C-ordered SLD-tree for a `tower` computation

```
reachable1(do(transfer(X,Y),s0),s0,1) :-
     block(X), block(Y), before(X,Y), X\==Y, clearx1(X,s0), cleary1(Y,s0).
reachable1(do(transfer(X,Y),S),S1,Moves):-
     Moves>0, Moves1 is Moves-1,
     reachable2(S,S1,Moves),block(X),block(Y), before(X,Y),X\==Y,
     clearx1(X,S,[]),cleary1(Y,S,[]).
```

In general, the repair procedure for fixing order irrelevance, is to gather the objects in the domain of the bindings for the generate-and-test argument that is the source of the inefficiency. As we discussed earlier, `reachable` generates bindings for the situation `S`, which is the only generate-and-test argument. Next, we pick an order on that domain. Our implemented repair procedure only handles total orders on a domain as shown above, but it is fairly straightforward to extend it to partial orders. We then modify the generator predicate by introducing a new test, right after the bindings for the generate-and-test predicate are produced.

The only source of inefficiency that remains after this transformation, is due to the height bounds being ignored during generation. How can we incorporate the `towerheight` test into the `reachable2` generator? We observe that the problem of building a tower whose height is within bounds H_{min} and H_{max} is one of picking a subset of the set of blocks, where the sum of the sizes of elements in a subset is within the two given bounds. Should $H_{min} = H_{max}$, then this problem is equivalent to the NP-complete subset-sum problem. If the distribution of problems predominantly contains instances with $H_{min} = H_{max}$, we should expect the average time complexity to be $O(2^N)$. What is the complexity of the problem when H_{min} and H_{max} are distinct?

Formally, the problem is: Given a finite set A of elements $\{a_1, \ldots, a_n\}$, with sizes $s(i) \in R^+, 1 \leq i \leq n$, and two bounds $B_1 \in R^+$ and $B_2 \in R^+$, where $B_1 \leq B_2$, find a subset A' of A such that $B_1 \leq \sum_{a \in A'} s(a) \leq B_2$.

We note that if $\sum_{a \in A} s(a) < B_1$, then no solution to the problem above exists. If $B_1 \leq \sum_{a \in A} s(a) \leq B_2$, then there is one solution to the problem: the entire set A. The most interesting case occurs when $\sum_{a \in A} s(a) > B_2$. This is the case that we will study: it requires construction of a proper subset of A whose sum is between the given bounds. An obvious dynamic programming formulation of the problem is as follows. We let $S(i, P)$ denote the fact that a subset of the first i elements of A sum *exactly* to P. Then we set up the following recurrence on S.

$$S(i, P) \equiv \exists j < i S(j, P) \vee \exists j S(j, P - s(i))$$

We have $S(1, s(1))$ to be true and all other $S(1, x)$ to be false, where $1 \leq x \leq \sum_{a \in A} s(a)$. If there exists a row k in the matrix S for which $S(k, P)$ is true, with $B_1 \leq P \leq B_2$, then the problem is soluble and we can construct the subset from the matrix S. The time complexity of this algorithm is $O(n^2 L)$ where L is the sum of the sizes of elements in A. This is therefore a psuedo-polynomial method.

What this analysis tell us is that the tower construction problem is intrinsically difficult, and that no speed-up method can reformulate the problem to make it soluble in polynomial time. The speed-ups we showed involved reducing exponents from $O(N^d)$ to $O(N!)$ to $O(2^N)$, which are all non-trivial reductions. Each reduction involved identifying redundancies in the generator which was the locus of inefficiencies in this problem. Redundancies resulted from the fact that the generator was non-canonical with respect to equivalence relations on state description sequences that were imposed by the tester predicates. Each step in our reasoning identified the source of redundancy using failed or successful computations, and modified the generator to make it canonical. The steps are summarized in Figure 16. We started with the original theory and learned Theory 1, Theory

2 and Theory 3 using the principle of minimization of rule choice computation. We then used order irrelevance to generate Theory 4 from Theory 1. The same composition of canonical repair procedures was applied to generate Theories 5 and 6 respectively.

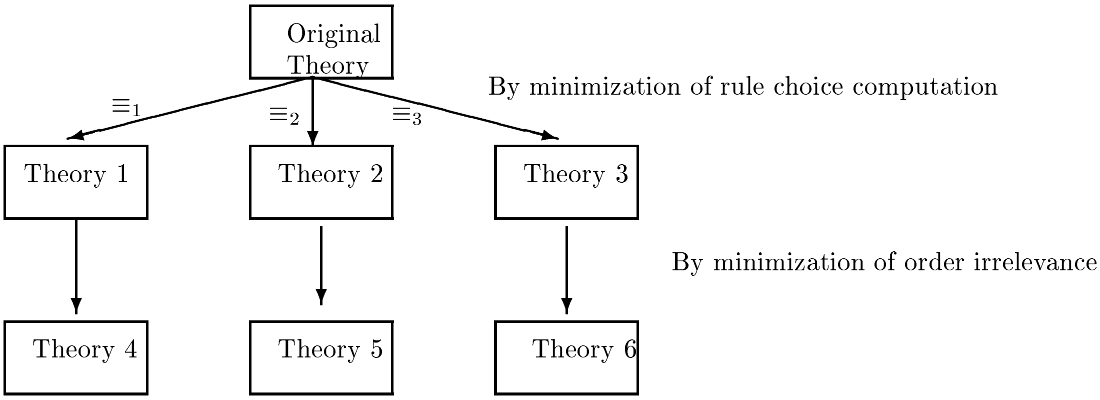


Figure 16: Goal-directed optimization of the `tower` computation

6 Experimental results

We present experimental results in the context of an iterative-deepening problem solver that works with a situation calculus formulation of the blocks world. The domain theory is adapted from [32]. One of the goal classes is building towers with heights within specified bounds from a given configuration of the world. This problem is isomorphic to the subset sum problem, known to be NP-complete. Figure 17 shows how the average cost of solution of a problem varies with increasing height bounds for the original (T, C) pair. The second goal class consists of sorting the blocks in an existing tower in increasing order of height. The sort is formulated as an exponential time generate and test method. The variation of the average cost of solution with increasing length of the list of blocks to be sorted in shown in Figure 18.

Experimental methodology The problem mix was varied continuously from 0 to 100 in increments of 10 reflecting the percentage of problems of the two types. We randomly generated 35 blocks-world scenarios each with 10 blocks of heights uniformly distributed from 1 to 10. We trained on 10 scenarios, and reserved 25 for testing. For each scenario characterized by specific block heights, we trained on 10 problems. The sample size of 100 is substantially smaller than the recommended sample size[7] for acquiring a multinomial distribution with k=2. The procedure REBL learned two specialized solvers. All solvers corresponding to $\equiv_i, i \geq 2$ were pruned by the utility criterion. The two solvers obtained a coverage of a 100 percent over the different distributions. The learned decision list uses a simple test of the goal as conditions for invoking the appropriate solver. Thus the selection cost for the solvers is negligible.

We tested the combination solver organized as a decision list on 25 randomly generated blocks-world scenarios. For each problem mix, and for each world we generated 15 tower stacking and 15 sorting problems. A time limit of 100 cpu seconds was imposed for the solution of each problem.

[7] In fact, we terminated sampling the distribution when the probabilities of occurrence for the two classes did not change by more than 1% over a run of 10 samples.

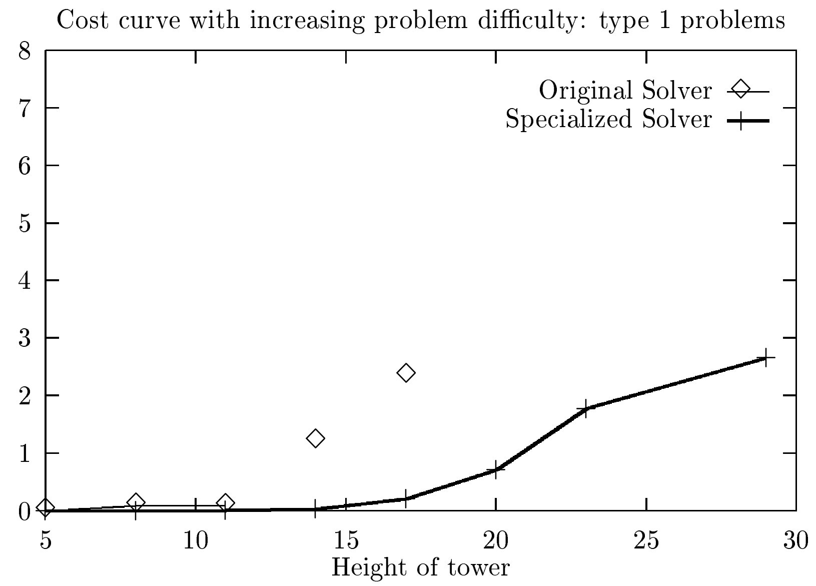


Figure 17: Original Solver vs Special Solver 1 for Blocksworld

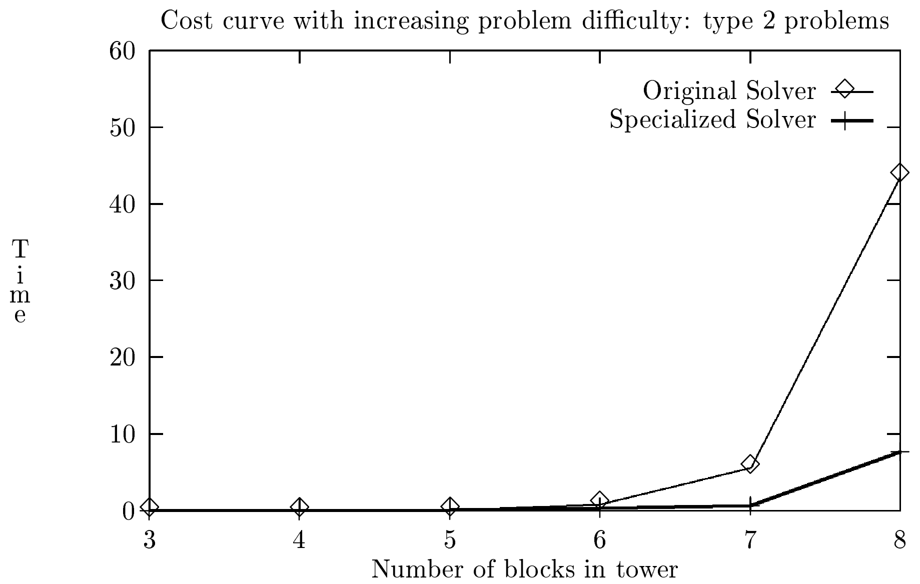


Figure 18: Original Solver vs Special Solver 2 for Blocksworld

The height range for towers in our experiments was uniformly distributed from 5 to 35, and the length of the block sequence to be sorted was uniformly distributed between 3 to 8. The experiments were performed using Quintus Prolog on a Sparc 1+.

The improvements result in the transformation of a problem solver with a high exponent in time complexity, to one which has a much lower exponent. This is because the domain theory generated from the proof grammar acquires the fact that the current block should be transferred to the top of the previous one in the stacking problem, as explained in the previous section. In the sort problem, special cases for sorting lists of length less than or equal to two provide constant factor improvement as shown in Figure 18. Figure 19 shows the performance improvement of the combined solver over the original solver over the blockworld problems. Note that when the percentage of problems of type 1 exceeds 20 percent, the original solver runs out of time.

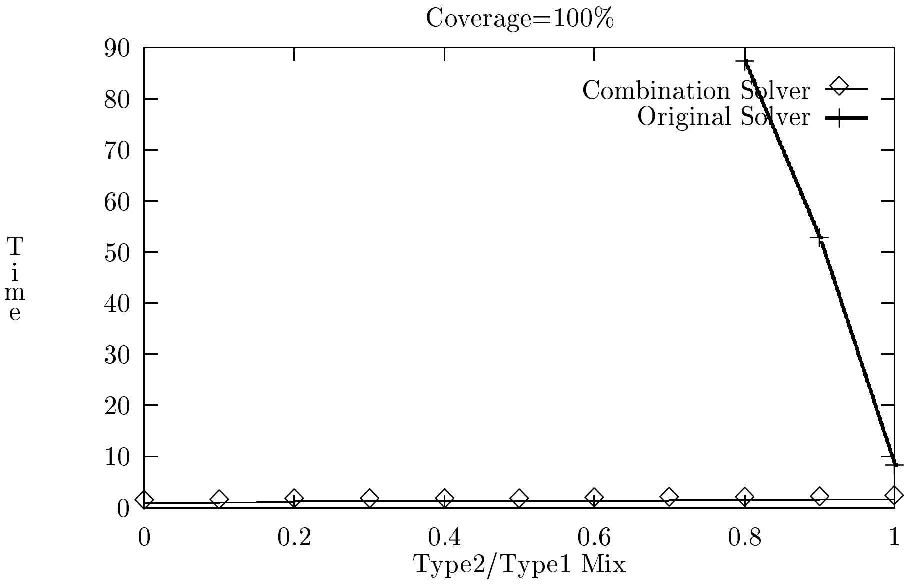


Figure 19: Combination Solver vs Original Solver for Blocksworld

We ran another series of experiments in Stripsworld. We formulated the domain theory in such a way that literal orderings could also be learned. This requires the addition of two explicit rules in the domain theory to allow for solving conjunctive goals in all possible orders. The first goal class in this domain is a set of `holding(`X^b`)` goals for blocks X at the bottom of a tower. The second goal class consisted of constructing a fully instantiated tower starting with all blocks on the table. The control information acquired by REBL is encoded in two special-purpose solvers, one for each goal class. The search space for the first goal class is exponential in the number of blocks.

Experimental Methodology The solvers for the tower building and unstacking problems were acquired with 10 test problems each consisting of n blocks, where n was drawn from a uniform distribution from 3 to 10. To generate a goal of the first type, we assemble a start state with a tower of the n blocks and a goal state with the robot holding the bottom block of this tower. To generate a goal of the second type, the goal state is the start state of the previous goal class, and the start state is a sequence of `ontable(X)`, for every block X in the world and a `handempty` assertion. The goal test to discriminate between the two solvers is very simple: the first solver solves goals of the form `holding(X)`, the second solves conjunctions of `on(X,Y)`'s. The solver is a linear planner.

The solver was given a bound of 1500 seconds for each problem. Each timing measurement from which the average times were computed was repeated 10 times. The tests of the combined solver against the original solver were performed over 100 problems for each problem mix. For instance, for the mix (0.1,0.9), 10 problems of type 1 and 90 problems of type 2 were generated. The number of blocks in each problem was uniformly drawn from the range 3 to 10 just as the training set was.

Figure 20 shows the average cost vs. coverage curves for both specialized solvers that were learned in this domain. This graph shows that each solver fails very quickly on problems solvable by the other solver. This is why Solver 1 (resp. Solver 2) has the highest average cost when the problem mix consists only of problems of type 1 (resp. type 2). Each special solver encodes the correct order in which to try operators, and the correct order in which to establish antecedents. Figure 21 shows the performance improvement of the combined solver consisting of the two specialized solvers over that of the original domain theory. The original domain theory could not solve goals involving problems of type 1 beyond 20 percent within the 1500 second limit. Large constant factor gains are obtained by the use of the combination solver.

7 Related Work

We now give a thumbnail sketch of the early developments in EBL which inspired our method for the analysis and design of EBL algorithms. We will describe them in terms of the three defining characteristics of EBL methods: the space of modifications considered, the methods for evaluating proposed changes to a domain theory and problem solver pair, and the distribution learning scheme.

Space of Modifications : In 1986, when the notion of explanation-based learning was proposed, the focus in the community was on devising the space of modifications to the existing domain theory, problem solver pair. EBG [26] and EGGS [6] worked with non-recursive theories and proposed the addition of the single rule m corresponding to $Elim(G, \{A_i \mid i > 0\})$. That is, all intermediate nodes $A_i, i > 0$ in the C-ordered SLD-tree G generated from a single successfully solved example were eliminated. The space of new rules that could be added by these techniques was the closure of the original domain theory under the operations of substitution and unfolding. The clause-selection policy of the solver was changed to consider the newly added rule ahead of all the original rules in the domain theory. These methods did not change the coverage of the new domain theory, solver pair as compared to the original pair. The inductive inference that was made on the basis of one example was that $Cost((T \cup \{m\}, C \text{ with selection rule}), P) \leq Cost((T, C), P)$. It was realized later that this inference was justified only for some special distributions P [35]. The algorithms therefore did not have distribution learning schemes.

It was quickly realized that the policy of augmenting the domain theory by a redundant rule and altering the clause selection method of the underlying solver caused slowdowns rather than speedups as the learner was trained on an increasing number of problems. This effect, dubbed the *utility problem*, was also observed in two-level systems like Prodigy and Soar that made only changes to the control policy and left the domain theory intact.

Two types of solutions were proposed to handle the utility problem: one was to make generation policies for T more sophisticated, the other to design control policies C that mitigated the ill-effects of a simple generation policy for T. Early examples of the first strategy were couched in terms of the *operationality debate*. Braverman and Russell [1] proposed generating partial unfoldings of rules in the domain theory. In our framework we can understand them as attempts to define a policy for elimination of non-terminal symbols in the production set of the context-free grammar generated by the example proof. A more recent example of the use of controlled generation is [39] who restricted the expressiveness of the learned rules in Soar. This arose from the desire to not change Soar's control policy of uniform chunking. Examples of the second approach of generating more sophisticated control policies include [25] and [27]. Mooney advocated not chaining on the antecedents of redundant rules in a domain theory. [25] added an evaluation component to the learner to explicitly estimate the costs and benefit of learned rules and devised simple policies to determine which rules to retain.

Further work on EBL revealed that when the underlying domain theory is recursive, the space of rules that can be learned is infinite. For the fixed-length unfoldings of a recursive domain rule that are generated by EBG-like schemes, the cost generally outweighs the benefit obtained from their use. The problem of compiling effective rules from a recursive domain theory was called the *generalization-to-n* problem because of the need to generalize rule invocation patterns that worked for any number of repetitions, not just the number observed in the training example. [33] produced the first strategy to generate rules in recursive theories: they involved compressing all distinct subproofs of height 2 in the generalized proof. [4, 38, 35] introduced more sophisticated techniques for generalizing number. [35, 9] demonstrated that unless strong assumptions about future query distribution are made, unfolding recursive rules as advocated by [32] was not an effective compilation strategy. [10] proposes a theory of problem space structure to guide speedup learning: he advocates not learning from recursive proofs. He also demonstrates that static analysis of a (T, C) pair can yield the performance improvements obtained with example-intensive methods. Our results in this paper are consistent with this observation: many of the inefficiencies revealed by a specific example are endemic to T itself and can be obtained just as easily using a context-free grammar generated from the given (T, C) pair.

Other generation policies have been advanced in the EBL literature in the context of other problem solver architectures. [38] showed that EBG-style macros were useful to form in propositional state-space problem solvers with a sparse solution space. This work built on the insight advanced by the work of Korf [21] in using block decomposability as a sufficient condition for determining useful macro sequences in a propositional state space problem. Many of the early EBG methods in state-space frameworks can be viewed as methods of converting slow state space searches into lookup over a set of partial solution caches. Chunking [28] can be viewed in this light. The technical questions are: at what granularity should solutions be stored and how is the utility of cached solutions to be determined. [3] compacted a cache of entire solutions to a class of state space problem by merging common prefixes in solutions. [39] identified the existence of a generality/usefulness tradeoff in the creation of these caches and proposed a technique of syntactically restricting what could be stored in the cache to guarantee that the items would not slow down the solver. EBL has also been studied in the context of forward chaining problem solvers [42] and in partial order planners.

Next we discuss work on designing effective control policies C for the T's generated by EBL algorithms. The possibility of designing optimal control strategies for a domain theory with redundancies was eliminated by [16]. He showed that the problem was was NP-complete. [17] presented a theoretical hill-climbing scheme for designing approximately locally optimal control policies for special classes of redundant theories. [12] studied the problem of finding a practically usable variant of this scheme and demonstrated significant speedups with his technique [13] on a non-trivial scheduling problem.

Many of the generation schemes for EBL discussed above produce new theories T' from the original domain theory T that do not sacrifice on coverage. [4] produced an elegant generation scheme using rule ordering alone that cast EBL as a multi-objective optimization problem: maximizing coverage while minimizing average problem solving cost. The formulation of the speedup learning problem advanced in this paper is a generalization of Cohen's formulation.

Evaluation: The use of a single example as a predictor of future problems was questioned in [7, 4, 15, 35, 14]. [7, 4] developed a method for combining information in multiple examples to generate more robust rules. [20] raised general issues in evaluation of performance improvement of theories. [25] pioneered the development of heuristic estimates of individual control rules added by learning. Refinements to his methods for estimating rule utility were advanced by [24]. [16, 14, 36] developed statistically sound sampling techniques for estimating the utility of candidate rules in a theory. [13] made the important observation that the utility of a rule set was not the sum of the utilities of its constituent rules. This provided evidence for the proposition that the utility estimation itself is a very expensive process if exact answers are required. [15] developed fully-polynomial time approximations to the utility estimation problem for non-recursive theories and [36] extended the method to special classes of recursive theories.

The need for care in judging performance improvement especially in the face of solvers with resource limits was raised in [30]. In an editorial in *Machine Learning*, Langley explained the importance of using methods from statistical hypothesis testing for determining whether performance improvement had occurred as a result of learning. A number of tests have been proposed: the one most relevant for evaluating EBL system is the use of signed rank test suggested by [11] and [5].

Distribution Learning: One of the more recent developments in EBL is the recognition of the importance of the problem distribution in the learning process. In [35], macro rules added in a recursive domain theory were shown to degrade performance in a domain as the problem mix was uniformly varied from one that favoured the macro to one that did not. [15] was one of the first attempts to adapt a theory to a stationary distribution by dynamically learning the optimal order of rules for solving a given goal class in a non-recursive T. [22] proposed the use of the term dynamic optimization to refer to the process of acquiring problem characteristics of a distribution before attempting optimization of a theory. We are guided by the very same philosophy in our work reported here.

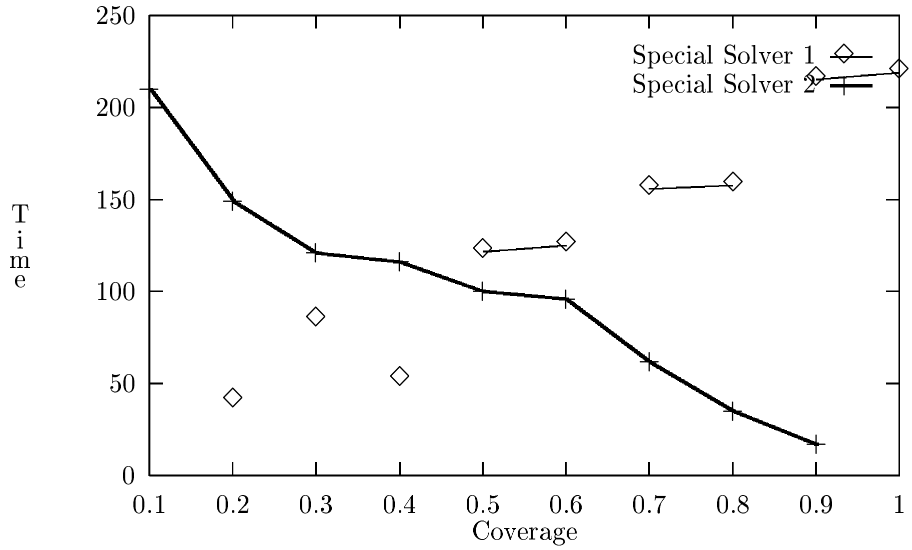


Figure 20: Coverage/Utility curves for two special solvers in Stripsworld

8 Conclusions and future work

This paper demonstrates the importance of examining the sources of inefficiency in domain theories and the need to analyze how they can be detected and eliminated. It proposes the use of compact representations of computation traces (or C-ordered SLD-trees), called path and proof grammars, for recording unsuccessful and successful attempts to solve a goal. Path and proof grammars can be reduced efficiently to eliminate repeated work at the base and at the meta-level. Instead of picking a single solver that improves upon the original (T, C) pair, we employ a set of specialized solvers organized as a decision list. The decision list is a representation of the mapping between equivalence classes in the problem distribution and the specialized solvers. Preliminary experimental results in support of this architecture is provided in the context of hard blocksworld planning problems.

Plenty remains to be done. The development of sophisticated methods of organizing specialized solvers is open: the decision list method used in this paper is quite simplistic. Better approximations to the NP-hard problem of constrained optimization of coverage subject to selection cost and average solution costs constraints remain to be analyzed and experimentally studied. My hope is that this formulation of the speedup learning problem will revitalize the business of generating new algorithms which are relevant to practical on-line learning problems in the real world.

Acknowledgements

We would like to thank Saumya Debray, Gerry deJong, Oren Etzioni, David Gries, Jane Hsu, Dexter Kozen, Keshav Pingali, Alex Russell, Stuart Russell, Alberto Segre, K. Sivaramakrishnan, Milind Tambe, and John Woodfill for valuable discussions and comments on previous drafts of this paper. We also thank the anonymous reviewers for constructive suggestions that substantially improved both the form and content of this paper.

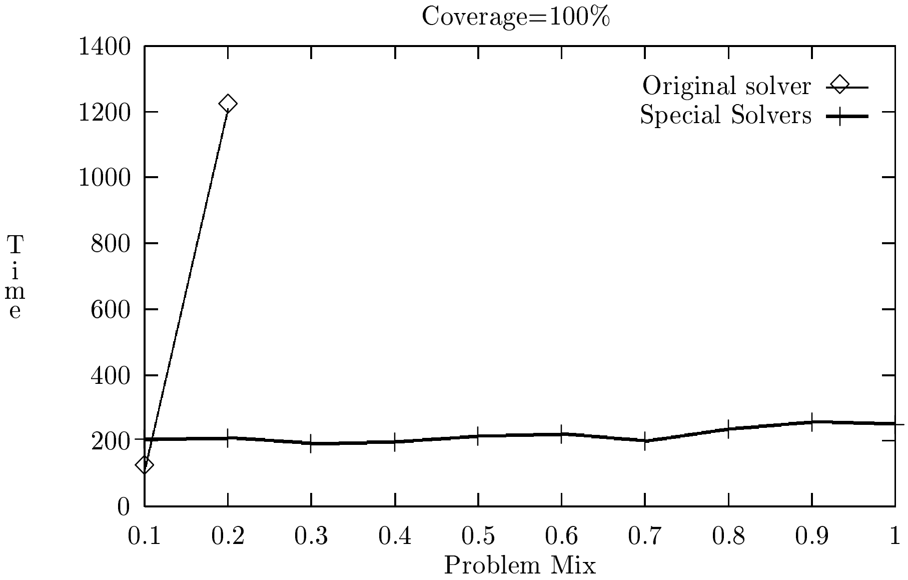


Figure 21: Original Solver vs Special Solver Combination for Stripsworld

References

References

[1] M. Braverman and S. Russell. Boundaries of operationality. In *Proceedings of the 5th International Conference on Machine Learning*, 1989.

[2] W. Cohen. *Concept learning using EBL as an abstraction mechanism.* PhD thesis, Rutgers University, October 1990.

[3] W. Cohen. Solution path caching mechanisms which provably improve performance. *Computational Intelligence*, 2(8), 1992.

[4] W.W. Cohen. Generalizing number and learning from multiple examples in ebl. In *Proceedings of the Conference on Machine Learning.* Morgan Kaufmann, 1988.

[5] M. deGroot. *Probability and statistics, second ed.* Addison-Wesley, 1989.

[6] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[7] T. Dietterich and N. Flann. Inductive methods in explanation-based learning. *Machine Learning*, 1989.

[8] C. Elkan. Conspiracy numbers and caching for searching and-or trees and theorem proving. In *Eleventh International Joint Conference on Artificial Intelligence.* Morgan Kaufman, 1989.

[9] O. Etzioni. Why prodigy/ebl works. In *Proceedings of the Eighth National Conference on Artificial Intelligence.* Morgan Kaufmann, 1990.

[10] O. Etzioni. *A structural theory of explanation-based learning.* PhD thesis, Carnegie-Mellon University, 1991.

[11] O. Etzioni and R. Etzioni. Statistical methods for analyzing speed-up learning experiments. Technical report, University of Washington, 1992.

[12] J. Gratch, S. Chien, and G. DeJong. Learning search control knowledge for deep space network scheduling. In *International Conference on Machine Learning.* Morgan Kaufman, 1993.

[13] J. Gratch and G. DeJong. An analysis of learning to plan as a search problem. In *International Conference on Machine Learning.* Morgan Kaufman, 1992.

[14] J. Gratch and G. DeJong. Rational learning: finding a balance between utility and efficiency. Technical report, University of Illinois, Urbana-Champaign, 1992. UIUCDCS-R-92-1756.

[15] R. Greiner. Finding the optimal derivation strategy in a redundant knowledge base. In *Proceedings of the Sixth International Workshop on Machine Learning.* Morgan Kaufmann, 1989.

[16] R. Greiner. Finding the optimal derivation strategy in a redundant knowledge base. In *Proceedings of the Second International Conference on Knoweldge Representation and Reasoning.* Morgan Kaufmann, 1991.

[17] R. Greiner and P. Orponen. Probably approximately optimal satisficing strategies. Technical report, Siemens Corporate Research, 1992.

[18] Patrick Hayes. Computation as deduction. Technical report, Essex University, 1977.

[19] J.B. Kadane and H.A. Simon. Optimal problem-solving search: all-or-none solutions. *Artificial Intelligence*, 6, 1975.

[20] R. Keller. Defining operationality for explanation-based learning. In *Proceedings of AAAI 1987*, 1987.

[21] R. Korf. Planning as search: a quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[22] P. Laird. Dynamic optimization. In *International Conference on Machine Learning.* Morgan Kaufman, 1992.

[23] J.W. Lloyd. *Foundations of logic programming, second edition.* Springer Verlag, 1987.

[24] S. Markovitch and P. Scott. Utilization filtering: a method for reducing the harmfulness of deductively learned knowledge. Technical report, Center for Machine Intelligence, 1989.

[25] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569. Morgan Kaufmann, 1988.

[26] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based learning: A unified view. *Machine Learning*, 1(1):47–80, 1986.

[27] R. Mooney. The effect of rule use on the utility of explanation-based learning. In *Proceedings of the Eleventh International Conference on Artificial Intelligence.* Morgan Kaufmann, 1989.

[28] J.E. Laird A. Newell and P.S. Rosenbloom. Chunking in soar: the anatomy of a general learning mechanism. *Machine Learning*, 1(1), 1986.

[29] R. Rivest. Learning decision lists. *Machine Learning*, 2, 1987.

[30] A. Segre, C. Elkan, and A. Russell. Technical note: a critical look at experimental evaluations of ebl. *Machine Learning*, 6(2), 1991.

[31] A.M. Segre. Bounded overhead caching for definite-clause theorem proving. *Journal of Logic Programming*, 1993. to appear.

[32] J. Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 1990.

[33] J. Shavlik and G. DeJong. Bagger: An ebl system that extends and generalizes explanations. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 516–520. Morgan Kaufmann, 1987.

[34] D. Subramanian. *A Theory of Justified Reformulations.* PhD thesis, Stanford University, March 1989.

[35] D. Subramanian and R. Feldman. The utility of ebl in recursive domain theories. In *Proceedings of AAAI-90*, 1990.

[36] D. Subramanian and S. Hunter. Measuring utility and the design of provably good ebl algorithms. In *International Conference on Machine Learning.* Morgan Kaufman, 1992.

[37] D. Subramanian and S. Hunter. Measuring utility: Estimating the cost of solving query distributions. Technical report, Computer Science Department, Cornell University, February 1992.

[38] P. Tadepalli. A formalization of explanation-based learning. In *Proceedings of IJCAI.* Morgan Kaufmann, 1989.

[39] M. Tambe and P. Rosenbloom. Eliminating expensive chunks by restricting expressiveness. In *Proceedings of the Eleventh International Conference on Artificial Intelligence*, pages 731–737. Morgan Kaufmann, 1989.

[40] J.D. Ullman. *Principles of database and knowledge-based systems, Vol II.* Computer Science Press, 1989.

[41] A. Webber. *Optimizing functional programs by minimizing irrelevant computation.* PhD thesis, Cornell University, 1993. Cornell Computer Science Technical Report.

[42] K. Yoshida and H. Motoda. Automatic knowledge reformulation. In *Proceedings of the Japanese Workshop on Knowledge Acquisition*, 1992.