Reduced Implicate Tries with Updates

Neil V. Murray\(^1\) and Erik Rosenthal\(^2\)

\(^1\) Department of Computer Science, State University of New York, Albany, NY 12222, USA, nvm@cs.albany.edu
\(^2\) Department of Mathematics, University of New Haven, West Haven, CT 06516, USA, erosenthal@newhaven.edu

Abstract. The reduced implicate trie, introduced in [11], is a data structure that may be used as a target language for knowledge compilation. It has the property that, even when large, it guarantees fast response to queries. Specifically, a query can be processed in time \textit{linear in the size of the query} regardless of the size of the compiled knowledge base.

The knowledge compilation paradigm typically assumes that the “intractable part” of the processing be done once, during compilation. This assumption could render updating the knowledge base infeasible if recompilation is required. The ability to install updates without recompilation may therefore considerably widen applicability. In this paper, several update operations not requiring recompilation are developed. These include disjunction, substitution of truth constants, conjunction with unit clauses, reordering of variables, and conjunction with arbitrary clauses.

1 Introduction

Several investigators have represented knowledge bases as propositional theories, typically as sets of clauses. However, since the question, Does \(\text{NP} = \text{P}\) remains open — i.e., there are no known polynomial algorithms for problems in the class \(\text{NP}\) — the time to answer queries is (in the worst case) exponential. The reduced implicate trie was developed [11–13] as a solution to a problem posed by Kautz and Selman [7] in 1991. Their idea, known as knowledge compilation, was to pay the exponential penalty once by compiling the knowledge base into a target language that would guarantee fast response to queries. They specified that the size of the target language be polynomial in the size of the original theory, and that query response time be polynomial in the size of the compiled theory. The result would then be polynomial response time to all queries.

Several authors achieved the second goal but not the first. But answering queries in time polynomial (indeed, often linear) in the size of the compiled theory is not very fast if the compiled theory is exponential in the size of the underlying theory. As a result, many investigators have focused on minimizing the size of the compiled theory, possibly by restricting or approximating the original theory. But no target language that provides fast response to queries and is guaranteed to be polynomial in the size of the original knowledge base has been discovered.

\* This research was supported in part by the National Science Foundation under grants IIS-0712849 and IIS-0712752.
The reduced implicate trie (ri-trie) takes a different approach: Admit large compiled theories (typically, stored off-line) on which queries can be answered quickly. In [11] ri tries are shown to guarantee response time linear in the size of the query.

In the knowledge compilation paradigm, the emphasis is typically on performing the “intractable part” of the processing once, during compilation. In the absence of an efficient updating technology, this favors knowledge bases that are stable; i.e., a single compilation is expected to provide a repository that remains useful over a large number of queries. But several authors have considered updating operations (referred to as transformations in [3]) for various target languages. In this paper, a number of updating operations for ri tries are developed.

Basic notions involving logical formulas, clauses, and implicants are covered in Section 2.1, in order that the paper be self-contained. The definition of reduced implicate trie and many of their properties are refined and described in the remainder of Section 2. In Section 3, several update techniques are introduced; none require recompilation.

2 Implicate Tries

The goal of knowledge compilation is to enable fast query response. Prior approaches had the goal of a small (i.e., polynomial in the size of the initial knowledge base) compiled knowledge base. Typically, query-response time is linear, so that the efficiency of querying the compiled knowledge base depends on its size. The approach used here, which builds upon [11], is to admit target languages that may be large as long as they enable fast query response time: Specifically, queries are answered in time linear in the size of the query. In particular, if the compiled knowledge base is exponentially larger than the initial knowledge base, the query must be processed in time logarithmic in the size of the compiled knowledge base. The ri trie is one data structure that admits such fast queries.

2.1 Propositional Logic

As usual, an atom is a propositional variable, a literal is an atom or the negation of an atom, and a clause is a disjunction of literals. Clauses are often referred to as sets of literals. Many authors restrict the theory to be compiled to conjunctive normal form (CNF) — a conjunction of clauses — but no such restriction is required in this paper.

Consequences expressed as minimal clauses that are implied by a formula are its prime implicants; (and minimal conjunctions of literals that imply a formula are its prime implicants). Implicants are useful in certain approaches to non-monotonic reasoning [9, 15, 17], where all consequences of a formula — for example, the support set for a proposed common-sense conclusion — are required. The implicants are useful in situations where satisfying models are desired, as in error analysis during hardware

---

3 The term off-line is used in two ways: first, for off-line memory, such as hard drives, as opposed to on-line storage, such as RAM, and secondly, for “batch preparation” of a knowledge base for on-line usage.

4 The term clause is sometimes used for a conjunction of literals, especially if disjunctive normal form is being used.
verification. Many algorithms have been proposed to compute the prime implicates (or implicants) of a propositional boolean formula [1, 4–6, 8, 14, 16, 18, 19].

A typical query of a propositional theory has the form, \textit{Is a clause logically entailed by the theory?} An \textit{implicate} of a logical formula is a clause entailed by the formula, i.e., a clause that contains a prime implicate. Thus a clause \( C \) is an implicate of a logical formula \( F \) if and only if \( C \) is satisfied by every interpretation that satisfies \( F \). As a result, asking whether a given clause is entailed by a formula is equivalent to the question, \textit{Is the clause an implicate of the formula?} In this paper, the term \textit{query} will always be interpreted as this latter question.

### 2.2 Complete Implicate Tries

The trie is a well-known data structure introduced by Morrison in 1968 [10]; it is a tree in which each branch represents the sequence of symbols labeling the nodes\(^5\) on that branch, in descending order. A prefix of such a sequence may be represented along the same branch by defining a special \textit{end symbol} that is attached to the last node in the prefix. (The term prefix is used in the usual way; a formal definition for the case of a clause in which the variables are ordered is given below.) One common application of tries is dictionaries. Tries work well because each word in the dictionary is present precisely as one (partial) branch in the trie. Checking a string for membership in the dictionary merely requires tracing a corresponding branch in the trie. This will either fail or be done in time linear in the size of the string.

Tries have also been used to represent logical formulas, including sets of prime implicates [17]. The nodes along each branch represent the literals of a clause, and the conjunction of all such clauses is a CNF equivalent of the formula represented by the trie. But observe that this CNF formula — see Figure 1 — is significantly larger than the corresponding trie, which turns out to be the \textit{complete implicate trie} for the example in Section 2.3 below. The literal that labels an interior node of the trie appears once in each of the clauses represented by branches that contain the node. Note that the root is labeled zero in Figure 1; this assures a single trie (rather than a forest).

Tries that represent logical formulas can be interpreted directly as formulas in negation normal form (NNF): A trie consisting of a single node represents the label of that node. Otherwise, the trie represents the disjunction of the label of the root with the conjunction of the formulas represented by the tries rooted at its children.

A trie that stores all (non-tautological) implicates of a formula is called a \textit{complete implicate trie}. To define it formally, first select an ordering \( \{p_1, p_2, \ldots, p_n\} \) of the variables that appear in the (propositional) formula \( F \), let \( q_i \) be either the literal \( p_i \) or the literal \( \neg p_i \), and order the literals by \( q_i \prec q_j \) if \( i < j \). (This can be extended to a total order by defining \( \neg p_i \prec p_i \), \( 1 \leq i \leq n \). But neither queries nor branches in the trie will contain such complementary pairs.) A \textit{prefix} of a clause \( \{q_{i_1}, q_{i_2}, \ldots, q_{i_m}\} \), is defined to be a clause of the form \( \{q_{i_1}, q_{i_2}, \ldots, q_{i_m}\} \), where \( 0 \leq m \leq k \). A prefix is \textit{proper} if \( m < k \). Note that if \( m = 0 \), then the prefix is the empty clause; i.e., the empty clause is a prefix of all clauses.

\(^5\) Many variations have been proposed in which arcs rather than nodes are labeled, and the labels are sometimes strings rather than single symbols.
The \textit{complete implicate trie} for $F$, with respect to a given variable ordering, is a tree defined as follows: If $F$ is truth constant, the tree consists of a root labeled with that constant. Otherwise, the complete implicate trie is a tree in which the root is labeled 0, all other nodes are labeled with literals, and the tree satisfies these properties:

1. No node has distinct children with the same label.
2. The set of labels of any prefix of any branch is a prefix of an implicate of $F$. (Note that the root, labeled 0, represents the empty clause, which is a prefix of any clause.)
3. If a prefix of the labels of a branch represent an implicate, then the last node in that prefix is marked with the end symbol. In particular, all leaves are marked with the end symbol.
4. Every implicate is the (not necessarily proper) prefix of some branch.

\textbf{Observations}

1. Every leaf is labeled with $q_n$ (i.e., with $p_n$ or with $\neg p_n$).
2. Whether a clause is an implicate of $F$ can be determined in time linear in the size of the clause simply by traversing the corresponding branch.
3. Since any superset of an implicate is an implicate, if a node labeled $q_k$ is marked with the end symbol, and if $k < n$, then the node will have as children nodes labeled $q_{k+1}$ and $\neg q_{k+1}$, and these children will be marked with the end symbol.
4. Part 4 of the definition slightly abused the term \textit{branch} by using it to mean the clause represented by its labels. Branch will be used in this way throughout this paper, typically assuming implicitly that any constants along the branch have been simplified away. (Later in this paper, a ternary representation of \textit{ri}-tries in which branches may have multiple constants will be described.)

\subsection{Reduced Implicate Tries}

Recall that for any logical formulas $F$ and $\alpha$ and subformula $G$ of $F$, $F[\alpha/G]$ denotes the formula produced by substituting $\alpha$ for every occurrence of $G$ in $F$. If $\alpha$ is a truth functional constant 0 or 1 (\textit{false} or \textit{true}), and if $p$ is a negative literal, we will slightly
abuse this notation by interpreting the substitution \([0/p]\) to mean that 1 is substituted for the atom that \(p\) negates.

The following simplification rules are useful (even if trivial).

**SR1.** \(\mathcal{F} \rightarrow \mathcal{F}[G / G \lor 0]\) \(\mathcal{F} \rightarrow \mathcal{F}[G / G \land 1]\)

**SR2.** \(\mathcal{F} \rightarrow \mathcal{F}[0 / G \land 0]\) \(\mathcal{F} \rightarrow \mathcal{F}[1 / G \lor 1]\)

**SR3.** \(\mathcal{F} \rightarrow \mathcal{F}[0 / p \land \neg p]\) \(\mathcal{F} \rightarrow \mathcal{F}[1 / p \lor \neg p]\)

These rules are sound for arbitrary logical formulas and, in particular, since the tries in this paper can be interpreted as NNF formulas, the rules can be applied to these tries. Since it is convenient to label the root 0, **SR1** will never be applied to the root. In fact, applications of **SR1** and **SR2** will be restricted to leaves. Any trie that is produced by a sequence of these rules and that has no leaves labeled with a constant (other than the root) is called an *implicate trie*. Observe that some applications of the simplification rules will produce a trie that has nodes other than the root — necessarily leaves — labeled with constants. These are not implicate tries. Of course, such a trie can be simplified to an implicate trie. An implicate trie that cannot be simplified with these rules is a *reduced implicate trie* (ri-trie).

The complete implicate trie for \(p \land q\), with \(p\) the first variable, is shown on the left in Figure 2. Applying rule **SR3** (conjunctive case) to the nodes labeled \(q\) and \(\neg q\) produces the trie on the right. This is not an implicate trie but becomes one after application of **SR1** removes the leaf labeled 0. No further application of the three rules is possible, so this last trie is a reduced implicate trie.

![Fig. 2. Application of SR3 to the Complete Implicate Trie for \(p \land q\)](image)

**Observations**

1. Suppose \(q\) is (the label of) a node with two leaf children, \(p\) and \(\neg p\). Then **SR3** replaces the two children with a single leaf labeled 0, and **SR1** deletes the new leaf.
2. If a node of an implicate trie is marked with the end symbol, then, since any superset of an implicate is an implicate, all extensions of that branch are implicates. As a result, if a node labeled \(q_k\) is marked with the end symbol, \(k < n\), it will have children labeled \(q_j\) and \(\neg q_j\), \(k < j \leq n\), and they will be also be marked with the end symbol. Thus, repeated applications of the simplification rules will delete the entire subtree below \(q_k\).
3. If the formula $F$ is a tautology, then repeated applications of the rules will produce a trie in which the root has a single child labeled 1; $\textbf{SR1}$ then produces the $ri$-trie consisting of a root labeled 1.

4. The only nodes in a reduced implicate trie with the end marker are leaves. In particular, no proper prefix of a branch in an $ri$-trie represents an implicate of the trie.

5. Any implicate for which no proper prefix is also an implicate is a branch in the $ri$-trie.

The last two observations lead naturally to a definition: If $F$ is a logical formula, and if the variables of $F$ are ordered, then a relatively prime implicate is an implicate for which no proper prefix is also an implicate. The next theorem is now immediate.

**Theorem 1.** Given a logical formula $F$ and an ordering of the variables of $F$, then the branches of the corresponding $ri$-trie represent precisely the relatively prime implicates. In particular, the prime implicates are relatively prime, and each is represented by a branch in the trie. ⊓ ⊔

**Example.** Suppose that the knowledge base $F$ contains the variables $p, q, r, s$, in that order, and suppose that $F$ consists of the following clauses: $\{p, q, \neg s\}$, $\{p, q, r\}$, $\{p, r, s\}$, and $\{p, q\}$. Both the complete implicate trie and the $ri$-trie are represented in Figure 3. The latter is simply the subtrie obtained by making the circled node a leaf. In the implicate trie, there are eight branches and eleven end markers, representing the eleven implicates — nine in the subtree rooted at $q$, and one each at the two rightmost occurrences of $s$.

![This node is a leaf in the reduced implicate trie](image)

Fig. 3. Implicate Trie and $ri$-Trie

The NNF equivalent of the $ri$-trie is $p \lor (q \land (\neg q \lor r \lor s) \land (r \lor s))$. A procedure for producing $ri$-tries from logical formulas is developed in the next section.

### 2.4 Computing Reduced Implicate Tries

Let $F$ be a logical formula, and let the variables of $F$ be $V = \{p_1, p_2, \ldots, p_n\}$. Then the $ri$-trie of $F$ can be obtained by applying the recursively defined RIT operator (introduced in [11]):
RIT(\mathcal{F}, V) = \begin{cases} \mathcal{F} & V = \emptyset \\ p_i \lor \text{RIT}\left(\mathcal{F}[0/p_i], V - \{p_i\}\right) \\ \neg p_i \lor \text{RIT}\left(\mathcal{F}[1/p_i], V - \{p_i\}\right) \\ \text{RIT}\left(\left(\mathcal{F}[0/p_i] \lor \mathcal{F}[1/p_i]\right), V - \{p_i\}\right) \\ \end{cases} \quad p_i \in V

where \(p_i\) is the variable of lowest index in \(V\). Implicit in this definition is the use of simplification rules \(\text{SR1}, \text{SR2}, \text{and SR3.}\)

There is a slight technical flaw in the original presentation because the RIT operator produces a forest; i.e., it does not include the root labeled 0. The fix is simple: Define \(\text{rin}(\mathcal{F}, V) = 0 \lor \text{RIT}(\mathcal{F}, V)\).\(^6\) When there is no possibility of confusion, we will often write \(\text{rin}(\mathcal{F})\) for \(\text{rin}(\mathcal{F}, V)\).

The distinction between \(\text{rin}(\mathcal{F}, V)\) and \(\text{RIT}(\mathcal{F}, V)\) turns out to be quite useful. It is occasionally necessary to refer to the forest of tries rooted at the children of the \(\text{rin}\)-trie of a formula \(\mathcal{F}\), and this is precisely \(\text{RIT}(\mathcal{F}, V)\).

Theorems 2 and 3 and the related lemmas were originally proved in [11]; proofs are presented here for the sake of completeness. Those results are stated for \(\text{RIT}(\mathcal{F}, V)\), but they also apply to \(\text{rin}(\mathcal{F})\). They assure us that \(\text{rin}(\mathcal{F})\) is the \(\text{rin}\)-trie of \(\mathcal{F}\) (with respect to the variable ordering \(V\).) In other words, the resulting trie is logically equivalent to the original formula, its branches are all relatively prime implicates of the formula, and any implicate has a unique prefix that corresponds precisely to a unique branch.

Note that a logical formula containing only one variable \(p\) must be logically equivalent to one of the following four formulas: 0, 1, \(p\), \(\neg p\). The next lemma, which is a trivial consequence of the definition of the RIT operator, says that in that case, \(\text{RIT}(\mathcal{F}, \{p\})\) is precisely the simplified logical equivalent.

**Lemma 1.** Suppose that the logical formula \(\mathcal{F}\) contains only one variable \(p\). Then \(\text{RIT}(\mathcal{F}, \{p\})\) is logically equivalent to \(\mathcal{F}\) and is one of the formulas 0, 1, \(p\), \(\neg p\). \(\square\)

**Theorem 2.** If \(\mathcal{F}\) is a logical formula with variable set \(V\), then \(\text{RIT}(\mathcal{F}, V)\) is logically equivalent to \(\mathcal{F}\).

**Proof.** Proceed by induction on the number \(n\) of variables in \(\mathcal{F}\). The last lemma takes care of the base case \(n = 1\) (and \(n = 0\)), so assume the theorem holds for all formulas with at most \(n\) variables, and suppose that \(\mathcal{F}\) has \(n + 1\) variables. Let \(\mathcal{F}_0 = \mathcal{F}[0/p_1]\), let \(\mathcal{F}_1 = \mathcal{F}[1/p_1]\), and let \(V_1 = V - \{p_1\}\); we must show that

\[
\mathcal{F} \equiv \text{RIT}(\mathcal{F}, V) = p_1 \lor \text{RIT}(\mathcal{F}_0, V_1) \land \neg p_1 \lor \text{RIT}(\mathcal{F}_1, V_1) \land \text{RIT}(\left(\mathcal{F}_0 \lor \mathcal{F}_1\right), V_1).
\]

\(^6\) These tries are \(n\)-ary. A ternary representation is introduced in Section 2.5; the superscript \(n\) is used to distinguish between the two.
By the induction hypothesis, \( \mathcal{F}_0 \equiv \text{RIT}(\mathcal{F}_0, V_1) \), \( \mathcal{F}_1 \equiv \text{RIT}(\mathcal{F}_1, V_1) \), and 
(\( \mathcal{F}_0 \lor \mathcal{F}_1 \)) \( \equiv \text{RIT}((\mathcal{F}_0 \lor \mathcal{F}_1), V_1) \). Let \( \tilde{I} \) be any interpretation that satisfies \( \mathcal{F} \), and suppose first that \( \tilde{I}(p_1) = 1 \). Then \( \tilde{I} \) satisfies \( p_1, \mathcal{F}_1 \), and \((\mathcal{F}_0 \lor \mathcal{F}_1) \), so \( \tilde{I} \) satisfies each of the three conjuncts of \( \text{RIT}(\mathcal{F}, V) \); i.e., \( \tilde{I} \) satisfies \( \text{RIT}(\mathcal{F}, V) \). The case when \( \tilde{I}(p) = 0 \) and the proof that any satisfying interpretation of \( \text{RIT}(\mathcal{F}, V) \) satisfies \( \mathcal{F} \) are similar.

Let \( \text{Imp}(\mathcal{F}) \) denote the set of all implicates of \( \mathcal{F} \).

**Lemma 2.** Suppose that the clause \( C \) is an implicate of the logical formula \( \mathcal{F} \), and that the variable \( p \) occurs in \( \mathcal{F} \) but not in \( C \). Then \( C \in \text{Imp}(\mathcal{F}[0/p]) \cap \text{Imp}(\mathcal{F}[1/p]) \).

**Proof.** Let \( \tilde{I} \) be an interpretation that satisfies \( \mathcal{F}[1/p] \); we must show that \( \tilde{I} \) satisfies \( C \). Extend \( \tilde{I} \) to \( \tilde{I}' \) by setting \( \tilde{I}'(p) = 1 \). Clearly, \( \tilde{I}' \) satisfies \( \mathcal{F} \), so \( \tilde{I}' \) satisfies \( C \). But then, since \( p \) does not occur in \( C \), \( I \) satisfies \( C \). The proof for \( \mathcal{F}[0/p] \) is identical, except that \( \tilde{I}(p) \) must be set to 0.

**Lemma 3.** Let \( \mathcal{F} \) and \( \mathcal{G} \) be logical formulas. Then \( \text{Imp}(\mathcal{F}) \cap \text{Imp}(\mathcal{G}) = \text{Imp}(\mathcal{F} \lor \mathcal{G}) \).

**Proof.** Suppose first that \( C \) is an implicate of both \( \mathcal{F} \) and \( \mathcal{G} \). We must show that \( C \) is an implicate of \( \mathcal{F} \lor \mathcal{G} \), so let \( \tilde{I} \) be an interpretation that satisfies \( \mathcal{F} \lor \mathcal{G} \). Then \( \tilde{I} \) satisfies \( \mathcal{F} \) or \( \tilde{I} \) satisfies \( \mathcal{G} \), say \( \mathcal{F} \). Then, since \( C \) is an implicate of \( \mathcal{F} \), \( \tilde{I} \) satisfies \( C \).

Suppose now that \( C \in \text{Imp}(\mathcal{F} \lor \mathcal{G}) \). We must show that \( C \in \text{Imp}(\mathcal{F}) \) and that \( C \in \text{Imp}(\mathcal{G}) \). To see that \( C \in \text{Imp}(\mathcal{F}) \), let \( \tilde{I} \) be a satisfying interpretation of \( \mathcal{F} \). Then \( \tilde{I} \) satisfies \( \mathcal{F} \lor \mathcal{G} \), so \( \tilde{I} \) satisfies \( C \). The proof that \( C \in \text{Imp}(\mathcal{G}) \) is entirely similar.

**Corollary.** Let \( C \) be a clause not containing \( p \) or \( \neg p \), and let \( \mathcal{F} \) be any logical formula. Then \( C \) is an implicate of \( \mathcal{F} \) iff \( C \) is an implicate of \( \mathcal{F}[0/p] \lor \mathcal{F}[1/p] \).

The last lemma means that the implicates being computed in the third conjunct of the RIT operator are precisely those that occur in both of the first two (ignoring, of course, the root labels \( p_1 \) and \( \neg p_1 \)). This third conjunct can thus be computed from the first two, and the direct recursive call on \( (\mathcal{F}[0/p_1] \lor \mathcal{F}[1/p_1]) \) can be avoided. This is significant because in this call, the size of the argument essentially doubles. Further use of this lemma is made in Section 3.

**Lemma 4.** Let \( C \) be a clause containing the literal \( p \). Then \( C \) is an implicate of \( \mathcal{F} \) iff \( C \setminus \{p\} \) is an implicate of \( \mathcal{F}[0/p] \).

**Proof.** Suppose first that \( C \in \text{Imp}(\mathcal{F}) \), and let \( \tilde{I} \) be an interpretation that satisfies \( \mathcal{F}[0/p] \). Extend \( \tilde{I} \) by defining \( \tilde{I}(p) = 0 \). Then \( \tilde{I} \) satisfies \( \mathcal{F} \), so \( \tilde{I} \) satisfies \( C \). Since \( I \) assigns 0 to \( p \), \( I \) must satisfy a literal in \( C \) other than \( p \); i.e., \( I \) satisfies \( C \setminus \{p\} \). Now suppose that \( C \setminus \{p\} \in \text{Imp}(\mathcal{F}[0/p]) \), and let \( \tilde{I} \) be an interpretation that satisfies \( \mathcal{F} \). If \( \tilde{I}(p) = 0 \), then \( \tilde{I} \) satisfies \( \mathcal{F}[0/p] \), so \( \tilde{I} \) also satisfies \( C \setminus \{p\} \). But then \( \tilde{I} \) assigns 1 to some literal in \( C \) other than \( p \), so \( \tilde{I} \) satisfies \( C \). If \( \tilde{I}(p) = 1 \), then \( \tilde{I} \) certainly satisfies \( C \), and the proof is complete.

\( \square \)

It is possible that there are variables other than \( p \) that occur in \( \mathcal{F} \) but not in \( \mathcal{F}[1/p] \). But such variables must “simplify away” when 1 is substituted for \( p \), so \( \mathcal{F} \) can be extended to an interpretation of \( \mathcal{F} \) with any truth assignment to such variables.
Theorem 3. Let $\mathcal{F}$ be a logical formula with variable set $V$, and let $C$ be an implicate of $\mathcal{F}$. Then there is a unique branch of $\text{RIT}(\mathcal{F}, V)$ that is a prefix of $C$, and every branch is a relatively prime implicate.

Proof. Every branch is an implicate of $\mathcal{F}$ since, by the distributive laws, $\text{RIT}(\mathcal{F}, V)$ is logically equivalent to the conjunction of its branches. It suffices to show that each implicate has a unique branch that is a prefix since showing that also proves that each branch is a relatively prime implicate. Proceed by induction on $n$, the size of $V$; Lemma 1 takes care of the base case $n = 1$.

Assume now that the theorem holds for all formulas with at most $n$ variables, and suppose that $\mathcal{F}$ has $n + 1$. Let $C$ be an implicate of $\mathcal{F}$, say $C = \{q_{i_1}, q_{i_2}, ..., q_{i_k}\}$, where $q_{i_j}$ is either $p_{i_j}$ or $\neg p_{i_j}$, and $i_1 < i_2 < ... < i_j$. We must show that a prefix of $C$ is a branch in $\text{RIT}(\mathcal{F}, V)$, which is the formula

$$p_1 \lor \text{RIT}(\mathcal{F}_0, V_1)$$

$$\land$$

$$\text{RIT}(\mathcal{F}, V) = \neg p_1 \lor \text{RIT}(\mathcal{F}_1, V_1)$$

$$\land$$

$$\text{RIT}((\mathcal{F}_0 \lor \mathcal{F}_1), V_1).$$

Observe that the induction hypothesis applies to the third conjunct since $p_1$ does not appear there. Thus, if $i_1 > 1$, there is nothing to prove, so suppose that $i_1 = 1$. Then $q_1$ is either $p_1$ or $\neg p_1$. Consider the case $q_1 = p_1$; the proof when $q_1 = \neg p_1$ is entirely similar. By Lemma 4, $C - \{p_1\}$ is an implicate of $\mathcal{F}_0$, and by the induction hypothesis, there is a unique prefix $B$ of $C - \{p_1\}$ that is a branch of $\text{RIT}(\mathcal{F}_0, V_1)$. But then $A = \{p_1\} \cup B$ is a prefix of $C$ and a branch of $\text{RIT}(\mathcal{F}, V)$.

To complete the proof, we must show that $A$ is the only such prefix of $C$. Suppose to the contrary that $D$ is another prefix of $C$ that is a branch of $\text{RIT}(\mathcal{F}, V)$. Then either $D$ is a prefix of $A$ or $A$ is a prefix of $D$; say that $D$ is a prefix of $A$. Let $D = \{p_1\} \cup E$. Then $E$ is a prefix of $B$ in $\text{RIT}(\mathcal{F}_0, V_1)$, which in turn means that $E$ is a prefix of $C - \{p_1\}$. But we know from the inductive hypothesis that $C - \{p_1\}$ has a unique prefix in $\text{RIT}(\mathcal{F}_0, V_1)$, so $E = B$, so $D = A$. If $A$ is a prefix of $D$, then it is immediate that $E$ is a prefix of $C - \{p_1\}$ in $\text{RIT}(\mathcal{F}_0, V_1)$, and, as before, $E = B$, and $D = A$. □

2.5 Ternary Representation

Observe that the RIT operator essentially produces a conjunction of three tries. It is therefore natural to represent an $ri$-trie as a ternary trie; this is illustrated in Figure 4.

In the ternary representation, the root of the third (right-most) subtrie is labeled 0. One advantage of this representation is that the $i$th variable appears only at level $i$. Another is that any subtrie (including the entire trie) is easily expressed as a four-tuple consisting of its root and the three subtries. For example, for a subtrie $T$ we might write

$$\langle r, T^+, T^-, T^0 \rangle$$

where $r$ is the root label of $T$, and $T^+$, $T^-$, and $T^0$ are the three subtries. These three subtries will often be referred to as, respectively, the first, second, and third subtries.
A trivial technical difficulty arises with the ternary representation: The zeroes along branches interfere with the prefix property of Theorem 3. But this is easily dealt with by interpreting the statement, *A branch B is a prefix of a clause C*, to mean *The clause represented by B with zeroes simplified away is a prefix of C*. The zeroes cause no difficulty when traversing branches in the trie.

Obtaining the ternary representation with the RIT operator requires only a minor change: disjoining 0 to the third conjunct.

\[
\text{RIT}(F, V) = \begin{cases} 
F & V = \emptyset \\
\left( p_i \lor \text{RIT}(F[0/p_i], V - \{p_i\}) \right) \land \left( \neg p_i \lor \text{RIT}(F[1/p_i], V - \{p_i\}) \right) \lor \left( 0 \lor \text{RIT}((F[0/p_i] \lor F[1/p_i]), V - \{p_i\}) \right) & p_i \in V
\end{cases}
\]

The notation \( ri(F, V) = 0 \lor \text{RIT}(F, V) \) will be used for the ternary \( ri \)-trie of \( F \) with variable ordering \( V \), and, if there is no possibility of confusion, we will often write \( ri(F) \). For the remainder of this paper, we will generally assume this ternary representation. As a result, the forest denoted by \( \text{RIT}(F, V) \) will contain three tries whose roots are labeled by a variable, its complement, and zero.

### 2.6 Uniqueness of \( ri \)-Tries

Reduced implicate tries can be represented as \( n \)-ary or as ternary trees. Otherwise, they are unique; that is, if \( F \) and \( G \) are logically equivalent, then \( ri(F, V) \) is isomorphic to \( ri(G, V) \), and \( ri^n(F, V) \) is isomorphic to \( ri^n(G, V) \). The theorems in this section are stated and proved for \( ri(F, V) \) but are easily adapted and proved for \( ri^n(F, V) \).

**Definition.** Let \( D_1 \) and \( D_2 \) be directed acyclic graphs (dags). Then \( D_1 \) and \( D_2 \) are said to be isomorphic if there is a bijection \( f \) such that if \( (A, B) \) is an edge in \( D_1 \),
then \((f(A), f(B))\) is an edge in \(D_2\). If the nodes of the dags are labeled, then the isomorphism is label-preserving if for every node \(A\), \(\text{Label}(A) = \text{Label}(f(A))\). In this paper, all dag isomorphisms are assumed to be label-preserving.

**Theorem 4.** Let \(\mathcal{R}\) be an \(ri\)-trie with the ternary representation, and let \(f\) be an isomorphism from \(\mathcal{R}\) to \(\mathcal{R}\). Then \(f\) is the identity map on \(\mathcal{R}\).

**Proof.** We proceed by induction on the number of variables in \(\mathcal{R}\). The result is trivial if there is only one variable, so assume true for any \(ri\)-trie with at most \(n\) variables, and suppose \(\mathcal{R}\) has \(n + 1\) variables. Since \(f\) preserves edges, the root must be mapped to itself, and its children must be mapped to children of the root. Since \(f\) preserves labels, and since the root has at most three children, labeled \(p_1, \neg p_1, 0\), \(f\) must map each of these children to itself. Note that the edge preservation property of \(f\) ensures that \(f\) maps each subtrie to itself. The induction hypothesis thus applies to each subtrie, and the proof is complete.

The theorem, while straightforward, is not immediate. The dag in Figure 5 has a label-preserving isomorphism — defined by the dotted arrows — that is not the identity. Of course this dag is not an \(ri\)-trie.

![Fig. 5. Non-Identity Label-Preserving Isomorphism](image)

The induction of the last theorem can easily be adapted to prove

**Theorem 5.** Let \(\mathcal{F}\) and \(\mathcal{G}\) be logically equivalent formulas. Then, with respect to a fixed variable ordering, \(ri(\mathcal{F})\) is isomorphic to \(ri(\mathcal{G})\). □

Note that \(\mathcal{F}\) and \(\mathcal{G}\) may have different variable sets. In that case, we assume that the fixed ordering in the theorem refers to the union of these sets. As a result, comparing the \(ri\)-tries of two formulas amounts to a (not necessarily practical) test for logical equivalence. On the other hand, if the formulas are known to be equivalent, attention can be restricted to variables in the intersection of their variable sets; all others are necessarily redundant.

It is interesting to observe that a labeled trie that “looks like” an \(ri\)-trie need not be one. For example, the left trie in Figure 6 represents the CNF formula \((p \lor q \lor r) \land (p \lor \neg r)\). Since \((p \lor q)\) is a resolvent, it is an implicate. In fact, it is a prime implicate and must therefore be a branch in the \(ri\)-trie. One problem is the branch \(\{p, q, r\}\), which contains
two prefixes of the implicate \( \{p, q, r\} \), namely \( \{p, q\} \) and itself. What is required is that the trie have the unique prefix property, which is defined in the next paragraph. The trie to the right, which does have that property, is the \( ri \)-trie of (the formula represented by) the trie to the left.

![Fig. 6. A non-\( ri \)-trie and its \( ri \)-trie](image)

Define an ordered propositional trie to be a tree consisting of a single node labeled with a truth constant, or a tree whose root is labeled 0, and whose remaining nodes are labeled with literals from an ordered set \( V \) of variables in such a way that if \( p \) labels a node above (i.e., closer to the root than) a node labeled \( q \), then \( p < q \). Moreover, no node may have distinct children with the same label. If \( T \) is an ordered propositional trie, let \( F \) be the CNF formula whose clauses consist of the (labels of the) branches in \( T \). Then \( T \) is said to have the unique prefix property if each implicate of \( F \) has a prefix that is a branch in \( T \), and if no proper prefix of any branch is an implicate of \( F \). Observe that in a trie that does have the unique prefix property, each implicate has a unique prefix that is a branch. Theorem 3 states that \( ri \)-tries have the unique prefix property; Theorem 6 is a kind of converse.

**Theorem 6.** Let \( T \) be an ordered propositional trie, and let \( F \) be the corresponding CNF formula. If \( T \) has the unique prefix property, then \( T \) is the \( ri \)-trie for \( F \).

**Proof.** Let \( R \) be the \( ri \)-trie for \( F \); it suffices to show that \( T \) and \( R \) have the same branches. The key to the proof is that both \( T \) and \( R \) have the unique prefix property. Suppose first that \( B \) is a branch in \( T \). Then \( B \) is a clause in \( F \) and thus an implicate of \( F \), so there is a unique branch \( C \) in \( R \) that is a prefix of \( B \). Since \( C \) is a branch in the \( ri \)-trie, it is also an implicate, so there is a unique branch \( D \) in \( T \) that is a prefix of \( C \). Thus, \( B \) and \( D \) are both branches in \( T \) that are prefixes of \( B \), so they must be the same. Since \( D \) is a prefix of \( C \), which is a prefix of \( B \), \( C = B \); i.e., \( C \) is the desired branch in \( R \). The proof that each branch in \( R \) is a branch in \( T \) is entirely similar.

**Corollary.** If \( r \) is a node in an \( ri \)-trie, and if \( T \) is the forest of all branches below \( r \), then \( 0 \lor T \) is the \( ri \)-trie of the branches of \( T \).

### 3 Updating \( ri \)-Tries

It is typical in the knowledge compilation paradigm to assume that the intractable part of the processing is done only once (or at least not very often). In the absence of an
efficient updating technology, this favors knowledge bases that are stable; i.e., a single compilation is expected to provide a repository that remains useful over a large number of queries. The original knowledge base can always be modified and then recompiled, but in general this is expensive. As a result, updates that can be installed into the compiled knowledge base without recompiling have the potential to widen applicability considerably.

Four update operations for $ri$-tries are developed in this section: Intersection, substitution of a truth constant, variable reordering, and conjunction of a clause. Intersecting two $ri$-tries yields the $ri$-trie of the disjunction of their input formulas: Given $F$ and $G$ and their respective $ri$-tries $T_F$ and $T_G$, the $ri$-trie for $F \lor G$ is precisely the intersection of $T_F$ and $T_G$. The process of substituting truth values for variables in an $ri$-trie requires some care: Preserving logical equivalence is trivial, but preserving the unique prefix property is not.

Conjoining a clause is a bit tricky. The technique presented here depends on conjoining a unit clause and reordering the variables so that the one in the unit is first. Conjoining an arbitrary clause is then accomplished by employing conjunction with units, variable reordering, and intersection.

Several of the operations described in this section operate on two $ri$-tries and require that both have the same variable ordering. As a result, unless otherwise stated, all tries under consideration will have the same variable ordering.

### 3.1 Intersecting $ri$-tries

Given two formulas $F$ and $G$, fix an ordering of the union of their variable sets, and let $T_F$ and $T_G$ be the corresponding $ri$-tries. The intersection of $T_F$ and $T_G$ is defined to be the $ri$-trie (with respect to the given variable ordering) that represents the intersection of the implicate sets. By Theorems 2 and 5 and Lemma 3, this is the $ri$-trie for $F \lor G$.\(^8\)

It is convenient to assume the ternary representation, and to represent a trie $T$ rooted at $p_i$ as the 4-tuple $\langle p_i, T^+, T^-, T^0 \rangle$. Observe that this trie represents the formula $p_i \lor (T^+ \land T^- \land T^0)$.

The intersection of two tries (with the same variable ordering) is produced by the INT operator, which depends on the boolean function $\text{diverge}$ and the binary operator $\bigoplus$, both of which are formally defined within the definition of INT. The $\bigoplus$ operator is the recursive call; it invokes INT on three corresponding pairs of children of two nodes. The boolean functions $\text{diverge}$ and $\text{leaf}$ test for the base cases that terminate the recursion. Further explanation of the INT operator can be found in the observations following its definition.

\(^8\) This definition captures the computation of the function $\text{buildzero}$ in [11], which is part of an algorithm that realizes the RIT operator.
\[
\begin{align*}
\text{INT}(\mathcal{T}_F, \mathcal{T}_G) &= \begin{cases} 
\emptyset & \mathcal{T}_F = \emptyset \lor \mathcal{T}_G = \emptyset \lor \text{diverge}(\mathcal{T}_F, \mathcal{T}_G) \\
\mathcal{T}_F & \text{leaf}(\mathcal{T}_G) \\
\mathcal{T}_G & \text{leaf}(\mathcal{T}_F) \\
\mathcal{T}_F \oplus \mathcal{T}_G & \text{otherwise}
\end{cases}
\end{align*}
\]

where \( \text{diverge}(\mathcal{T}_F, \mathcal{T}_G) = \)
\[
[(\mathcal{T}_F^+ = \emptyset \neq \mathcal{T}_G^+) \lor (\mathcal{T}_F^+ \neq \emptyset = \mathcal{T}_G^+)] \land [(\mathcal{T}_F^- = \emptyset \neq \mathcal{T}_G^-) \lor (\mathcal{T}_F^- \neq \emptyset = \mathcal{T}_G^-)];
\]

and, with \( r \) as the root label of both \( \mathcal{T}_F \) and \( \mathcal{T}_G \),
\[
\mathcal{T}_F \oplus \mathcal{T}_G = (r, \text{INT}(\mathcal{T}_F^+, \mathcal{T}_G^+), \text{INT}(\mathcal{T}_F^-, \mathcal{T}_G^-), \text{INT}(\mathcal{T}_F^0, \mathcal{T}_G^0)).
\]

**Observations.**

1. The first case is trivial when one of the two tries is empty: An intersection with the empty set is empty. The \( \text{diverge} \) test simply checks that in each of the first two pairs of corresponding subtrees, one is empty and the other is not, also implying an empty intersection. Please note that empty does not mean a leaf node; quite the contrary, it means no trie whatsoever.
2. The next two cases reflect the fact that if one trie is a leaf, then all descendant branches are implicitly present, so the intersection is the entire other trie.
3. The last case is the recursive call.
4. Zero labels are irrelevant to the INT computation.
5. The INT operator produces a trie that looks like an \( ri \)-trie; Lemma 5 and Theorem 7 demonstrate that this trie is in fact the intersection of its arguments.

The \( ri \)-trie for \( \mathcal{F} = \neg(p \leftrightarrow q) \land \neg r \) is shown in Figure 7.

**Fig. 7. Intersection Examples in the \( ri \)-Trie for \( \neg(p \leftrightarrow q) \land \neg r \)**
The rightmost subtrie below the node labeled \( p \) is the intersection of the two siblings on the left (treated as though each had a zero root – see the definition of RIT below which employs INT). The same is true for the rightmost subtrie below \( \neg p \). In both cases, the intersection can make use of subtrie structure that is present in one of the two tries being intersected. Such situations correspond to the second or third case in INT; the returning value is one of the two arguments, and the implementor can choose to return a copy of the argument or a pointer to it. The latter is assumed in the figure.

The rightmost subtrie below the zero root of the entire subtrie is of course the intersection of the leftmost two, and effective structure sharing will be available to the implementor.

**Lemma 5.** Let \( T_F \) and \( T_G \) be \( ri \)-tries (with the same variable ordering), and let the clause \( C_F \) be a prefix of \( C_G \), where \( C_F \) is a branch in \( T_F \) and \( C_G \) is a branch in \( T_G \). Then \( C_G \) is a branch in \( \text{INT}(T_F, T_G) \).

**Proof.** Proceed by induction on the number \( n \) of literals in \( C_F \). If \( n = 1 \), then \( C_F = \{ p_i \} \) is a singleton. Since it is also a branch in \( T_F \), \( p_i \) must be the label of a leaf. The third case in the definition of INT thus applies, and the intersection will be \( T_G \). Since \( C_G \) is a branch in \( T_G \), it is in the intersection.

Assume now that the lemma holds for at most \( n \) literals in \( C_F \), and suppose \( C_F \) has \( n+1 \) literals; let \( p_i \) be the first. Since both \( C_F \) and \( C_G \) correspond to branches representing clauses of length greater than one, Case 5 must apply. Thus, in the 4-tuple representation, \( r = p_i \), and \( T_F \bigoplus T_G = \langle p_i, \text{INT}(T_F^+, T_G^-), \text{INT}(T_F^-, T_G^+), \text{INT}(T_F^0, T_G^0) \rangle \). Each of \( C_F - \{ p_i \} \) and \( C_G - \{ p_i \} \) is non-empty, and the former is a prefix of the latter. As a result, \( C_F - \{ p_i \} \) is a branch in \( T_F^+, T_F^- \), or \( T_F^0 \), say \( T_F^+ \) — the proof is similar in the other cases. Then \( C_G - \{ p_i \} \) is a branch in \( T_G^- \). Since \( C_F - \{ p_i \} \) contains at most \( n \) literals, and since \( T_F^+ \) is an \( ri \)-trie by Theorem 2, the induction hypothesis applies. Hence \( C_G - \{ p_i \} \) is a branch in \( \text{INT}(T_F^+, T_G^-) \), which implies that \( C_G \) is a branch in \( \text{INT}(T_F, T_G) \).

Essentially, the converse of Lemma 5 also holds.

**Lemma 6.** Let \( T_F \) and \( T_G \) be \( ri \)-tries, and let \( C \) be a branch in \( \text{INT}(T_F, T_G) \). Then \( C \) is a branch in one of \( T_F \) or \( T_G \), and a prefix of \( C \) is a branch in the other.

**Proof.** All non-empty branches of \( \text{INT}(T_F, T_G) \) are constructed through some number of recursive calls, terminating in Case 2 or Case 3 — say Case 2 — in the definition of INT. When Case 2 arises, a path \( C' \) in \( \text{INT}(T_F, T_G) \) has been constructed. That path is a branch in \( T_G \) and is a prefix of every branch built by Case 2, including \( C \).

**Theorem 7.** Let \( T_F \) and \( T_G \) be the respective \( ri \)-tries of \( F \) and \( G \) (with the same variable ordering). Then \( \text{INT}(T_F, T_G) \) is the intersection of \( T_F \) and \( T_G \); in particular, \( \text{INT}(T_F, T_G) \) is the \( ri \)-trie of \( F \lor G \) (with respect to the given variable ordering).

**Proof.** Let \( C \) be an implicate of \( F \lor G \). By Lemma 3, \( C \) is an implicate of both \( F \) and \( G \). Then by Theorem 3, there is a unique branch in \( T_F \) that is a prefix of \( C \); call it \( C_F \). Similarly, there is a prefix \( C_G \) of \( C \) that is a branch in \( T_G \). We must show that some prefix of \( C \) is a unique branch in the intersection.
If \( C \) is the empty clause, then \( \mathcal{F} \lor \mathcal{G} \) is (logically equivalent to) the empty clause, and both \( T_F \) and \( T_G \) are singleton roots labeled 0. Thus, the intersection is empty, and there is nothing to prove. If either \( C_F \) or \( C_G \) is empty, then one of the tries is a singleton root, the intersection is the other trie, and the result is immediate. So assume that none of \( C, C_F \), and \( C_G \) are empty. One or both of \( C_F \) and \( C_G \) is a prefix of the other; without loss of generality, assume \( C_F \) to be a prefix of \( C_G \). In particular, the branch labeled \( C_F \) in \( T_F \) is a prefix of the branch in \( T_G \) labeled \( C_G \). By Lemma 5, \( C_G \) is a branch in \( \text{INT}(T_F, T_G) \).

To complete the proof, we must show that \( \text{INT}(T_F, T_G) \) has the unique prefix property, i.e., that no branch in \( \text{INT}(T_F, T_G) \) other than \( C_G \) is a prefix of \( C_G \). Suppose \( C' \) were such a branch. By Lemma 6, \( C' \) or a prefix of \( C' \) would be a branch in \( T_G \). But this would violate the unique prefix property in \( T_G \).

\[ \square \]

Theorem 7 provides a formal basis for a definition of the RIT operator that produces \( ri \)-tries using intersection and structure sharing. The definition of RIT below\(^9\) is the ternary case; the \( n \)-ary case can be similarly handled. Naturally, \( ri(\mathcal{F}) = 0 \lor \text{RIT}(\mathcal{F}) \) is unchanged.

\[
\text{RIT}(\mathcal{F}, V) = \begin{cases} 
\mathcal{F} & V = \emptyset \\
(p_i \lor B_1) \land (\neg p_i \lor B_2) \land B_3 & p_i \in V
\end{cases}
\]

where \( p_i \) is the variable of lowest index in \( V \), and

\[
B_1 = \text{RIT}(\mathcal{F}[0/p_i], V - \{p_i\}) \\
B_2 = \text{RIT}(\mathcal{F}[1/p_i], V - \{p_i\}) \\
and \quad B_3 = \text{INT}(0 \lor B_1, 0 \lor B_2)
\]

In the call to \( \text{INT} \) from \( \text{RIT} \), \( B_1 \) and \( B_2 \) are disjoined with zero; i.e., \( ri(B_1) \) and \( ri(B_2) \) are the arguments of \( \text{INT} \). The careful reader will recall that \( B_1 \) and \( B_2 \) are forests, not tries. The \( \text{INT} \) operator not only expects tries, but tries with identically labeled roots. So a zero root is provided for each argument in the initial call, and the intersection trie is constructed with the required zero root. Recursive calls that stem from this initial call are invoked on subtries, i.e., on tries, not on forests resulting from the \( \text{RIT} \) operator.

### 3.2 Truth Functional Simplification

Suppose \( p \) is a variable in a formula \( \mathcal{F} \) whose \( ri \)-trie is \( T_F \), and suppose we want to find the \( ri \)-trie for \( \mathcal{F}[1/p] \). As we shall see shortly, computing \( T_F[1/p] \) can be accomplished by removing \( \neg p \) from all branches on which it occurs, cutting off any branch that contains \( p \) at \( p \), and simplifying. This produces a trie whose branches correspond to a CNF equivalent of \( \mathcal{F}[1/p] \). The resulting trie, however, is not in general an \( ri \)-trie: The unique prefix property may not hold. The \( ri \)-trie on the left in Figure 8, for example, represents the eleven implicates from Figure 3. Read as an NNF formula, it represents

\[ \text{This definition is implicit in the algorithm introduced in [11] that computes \( ri \)-tries.} \]

---

\[ \]
\((p \lor (q \land (\neg q \lor r \lor s) \land (r \lor s)))\). The trie on the right is the result of substituting 1 (true) for \(q\) and simplifying. This trie is not an \(ri\)-trie since the unique prefix property does not hold: The subtrie rooted at 0 is subsumed by the subtrie originally rooted at \(\neg q\).

![Fig. 8. Substituting a truth constant and the prefix property](image)

This redundancy follows a predictable pattern that can be avoided. We begin the analysis with two lemmas.

**Lemma 7.** Let \(F\) be a set of clauses containing the unit clause \(\{p\}\), and let \(\tilde{F}\) be the formula obtained from \(F\) by deleting all clauses containing \(p\) and removing all occurrences of \(\neg p\). Then \(F \equiv \{p\} \land \tilde{F}\), and \(\tilde{F} \equiv F[1/p]\). In particular, \(F \equiv \{p\} \land F[1/p]\).

**Proof.** Since \(F\) contains \(\{p\}\), all other clauses containing \(p\) are subsumed by this one. Hence, they may be removed, preserving logical equivalence. Resolving this unit with all clauses containing \(\{\neg p\}\) in effect eliminates the literal \(\neg p\) from \(F\). Thus, \(F \equiv \{p\} \land \tilde{F}\). Substituting 1 for every occurrence of \(p\) in \(F\) and simplifying also produces \(\tilde{F}\), and the proof is complete. \(\square\)

**Remark.** Lemma 7 is stated for CNF formulas, but that restriction is unnecessary. For any logical formula \(F\), if \(\tilde{F} = F[1/p]\), then \(F \land \{p\} \equiv \tilde{F} \land \{p\}\). Only minor modification of the proof is required for formulas not in CNF.

**Lemma 8.** Let \(F\) be a logical formula, and let \(\tilde{F} = F[1/p]\). If \(C\) is an implicate of \(\tilde{F}\) but not an implicate of \(F\), then \(D = C \cup \{\neg p\}\) is an implicate of \(F\), and the prefix of \(D\) ending in \(\neg p\) is a relatively prime implicate of \(F\).

**Proof.** Let \(I\) be any interpretation that satisfies \(F\); we must show that \(I\) satisfies \(D\). If \(I(p) = 0\), then \(I\) satisfies \(D\) since \(\neg p \in D\). If \(I(p) = 1\), then \(I\) satisfies \(\tilde{F}\), so \(I\) satisfies \(C\), so \(I\) satisfies \(D\). This completes the proof because it is now immediate that the prefix of \(D\) ending in \(\neg p\) is a relatively prime implicate of \(F\). \(\square\)

To understand how to compute the \(ri\)-trie of \(F[1/p]\) from the \(ri\)-trie of a formula \(F\), let \(V = \{p_1, \ldots, p_n\}\) be the variable set of \(F\), let \(T = ri(F, V)\), and select \(p_m \in V\). We have seen that substituting 1 for every occurrence of \(p_m\) in \(T\) and simplifying
T
−
T
0
T
+
T
−
T
0
T
+

Fig. 9. Occurrence of \( p_m \) in \( T \)

does not produce the \( ri \)-trie of \( F[1/p_m] \). To obtain the desired \( ri \)-trie, observe that any occurrence of the variable \( p_m \) in \( T \) has the form shown in Figure 9.

The root \( r \) can be \( p_{m-1} \), \( \neg p_{m-1} \), or 0; in the latter case, if \( m = 1 \), then \( r \) is the root of the entire trie. Consider first substituting 1 for \( p_m \).

Case 1. The second branch \( T^- \) is empty.\(^{10}\) Since \( T^0 \) is the intersection of the first two branches, it too does not exist. Thus, \( p_m \) is the only child of \( r \). If \( p_m \) is set to 1, then we have \( r \lor 1 \lor T^+ \), which simplifies to 1, and further simplifications are necessary but straightforward.

Case 2. The first branch \( T^+ \) is empty. Thus, \( \neg p_m \) is the only child of \( r \), and setting \( p_m = 1 \) makes \( \neg p_m \) 0. Since it is disjoined to its children, simplifying replaces the one child of \( r \) with the three children of \( \neg p_m \).

Case 3. Neither of the first two branches is empty. Substituting 1 for \( p_m \) makes the first branch 1, as in Case 1, but now the 1 is conjoined to the other children of \( r \), and simplifying deletes that first child. Since \( \neg p_m = 0 \), the second branch simplifies as in Case 2. Since \( T^0 \) is the intersection of \( T^+ \) and \( T^- \), it is subsumed by (the simplified) \( T^- \) and thus is unnecessary. As a result, the third branch may be deleted, and the three branches of \( r \) become the three children in \( T^- \), as in Case 2.

Cases 1’, 2’, 3’. If 0 is substituted for \( p_m \), the effect is similar except that it is the second branch that vanishes.

Remark. This process can be repeated for all occurrences of \( p_m \), every one of which is at level \( m \) in the ternary representation. The result is denoted \( T[1/p_m]^s \) or \( T[0/p_m]^s \), as appropriate. The superscript \( s \) is there to remind us that we must do both constant and subsumption simplifications. Theorem 8 states that \( T[1/p_m]^s \) is the \( ri \)-trie for \( F[1/p_m] \).

Theorem 8. Let \( F \) be a logical formula with variable set \( V = \{p_1, \ldots, p_n\} \), and let \( T \) be the \( ri \)-trie of \( F \) with respect to \( V \). Then \( ri(F[1/p_m], V') = T[1/p_m]^s \), and \( ri(F[0/p_m], V') = T[0/p_m]^s \), where \( V' = V - \{p_m\}, 1 \leq m \leq n \).

Proof. We will prove the theorem for the case in which 1 is substituted for \( p_m \); the proof when 0 is substituted for \( p_m \) is the exact dual. Let \( \tilde{F} = F[1/p_m] \), and let \( \tilde{T} = T[1/p_m]^s \).

\(^{10}\) Note that this means that the second branch does not exist; it does not mean that \( \neg p_m \) is a leaf.
Note that, since \( \tilde{T} \) is produced by simplifying \( T \), each branch in \( \tilde{T} \) is produced by reducing a branch in \( T \) and is a subset of that branch. In particular, \( \tilde{T} \) is an ordered propositional trie. By Theorem 6 it suffices to prove that \( \tilde{T} \) has the unique prefix property; i.e., to prove that every implicate of \( \tilde{F} \) has a prefix that is a branch in \( \tilde{T} \), and that no proper prefix of a branch in \( \tilde{T} \) is an implicate of \( \tilde{F} \).

Suppose first that \( C \) is an implicate of \( \tilde{F} \). Then by Lemma 8, \( C' = C \cup \{ \neg p_m \} \) is an implicate of \( F \). Thus some branch in \( \tilde{T} \) is a prefix of \( C' \); the branch in \( \tilde{T} \) to which this branch reduces is a prefix of \( C \).

To complete the proof, we must show that no branch in \( \tilde{T} \) has a proper prefix that is an implicate of \( \tilde{F} \). Suppose to the contrary that some branch \( B \) has a proper prefix \( C \) that is an implicate. Then, by Lemma 8, \( C' = C \cup \{ \neg p_m \} \) is an implicate of \( F \). Let \( p_k \) be the last variable in \( C \); there are two cases to consider.

**Case 1.** \( k > m \). Then the branch \( B \) was produced by Case 2 or 3 of the definition of \( T[1/p_m]^* \), i.e., by removing \( \neg p_m \) from a branch in \( \tilde{T} \), but with no further simplifications. The reason there could not have been further simplifications is that subsumption simplifications do not apply, and constant simplifications that remove branches necessarily occur above level \( m \); i.e., there are no constant simplifications below level \( m \). Thus \( B' = B \cup \{ \neg p_m \} \) is a branch in \( T \). But \( C' \) is an implicate of \( F \) and a proper prefix of \( B' \), violating the unique prefix property in \( T \).

**Case 2.** \( k < m \). Note that \( k = m \) is impossible since \( p_m \) does not occur in \( \tilde{T} \).

Now, since \( C \) is a proper prefix of a branch in \( \tilde{T} \), \( C \) is a proper prefix of a branch in \( T \). Thus, since \( T \) satisfies the unique prefix property, \( C \) is not an implicate of \( F \), so, by Lemma 8, \( C' \) is a relatively prime implicate of \( F \). In particular, \( C' \) is a branch in \( T \), and the leaf of that branch is labeled \( \neg p_m \). Also, each node on that branch below the node labeled \( p_k \) and above the leaf is labeled 0. When 1 is substituted for \( p_m \), 0 is substituted for the leaf \( \neg p_m \). That leaf is conjoined with its siblings. The result is 0, which is disjoined with the parent. If the parent is not \( p_k \), this disjunction is 0, and that process continues up the trie, terminating when \( p_k \) becomes a leaf. But then \( C \) is itself a branch in \( T \) and thus cannot be a proper prefix of a branch. This contradiction completes the proof.

### 3.3 Conjoining Clauses and Reordering Variables

In this section, methods for adding a clause to an \( ri \)-trie and for reordering the variables of an \( ri \)-trie are presented. Consider first a unit clause. Specifically, how can the \( ri \)-trie of \( F \land \{ p \} \) be found from the \( ri \)-trie of \( F \)? By Lemma 7, \( F \land \{ p \} \equiv F[1/p] \land \{ p \} \). Since the \( ri \)-trie of \( F[1/p] \) can be determined using Theorem 8, and since \( F[1/p] \) does not contain \( p \) (nor \( \neg p \)), what is required is finding the \( ri \)-trie of \( F \land \{ p \} \) when \( p \) does not occur in \( F \). We will treat \( p \) as a positive literal — i.e. as a variable — the dual case when it is negative is straightforward.

Let \( V = \{ p_1, \ldots, p_n \} \) be the variables of \( F \), let \( V' = \{ p \} \cup V \), where \( p \) is the first variable in \( V' \), and let \( T = ri(F, V) \); our goal is to compute \( ri(F \land \{ p \}, V') \) from \( T \). Observe that \( (F \land p)[0/p] = 0 \), and \( (F \land p)[1/p] = F \). Thus, from the definition of the RIT operator, we have

\[ ri(F \land \{ p \}, V') = ri(F \land \{ p \}, V') \]

\footnote{We cannot say prefix because \( p_m \) may be deleted from the “middle” of a branch.}
\[
\text{RIT}((\mathcal{F} \land p), V') = \left( \begin{array}{c}
p \lor \text{RIT}((\mathcal{F} \land p)[0/p], V) \\
\neg p \lor \text{RIT}((\mathcal{F} \land p)[1/p], V) \\
0 \lor \text{INT}(0, \text{RIT}((\mathcal{F} \land p)[1/p], V))
\end{array} \right) = \left( \begin{array}{c}
p \\
\neg p \lor \text{RIT}(\mathcal{F}, V) \\
0 \lor \text{RIT}(\mathcal{F}, V)
\end{array} \right).
\]

The result is pictured in Figure 10.

![Figure 10. Conjoining the Unit Clause \{p\}](image)

Theorem 9 is now immediate. Note that if \( p \not\in V \), then \( \mathcal{F}[1/p] = \mathcal{F} \), so, in that case, \( T[1/p]^* = T \). As a result, Theorem 9 is valid whether or not \( p \in V \). Also note that while the duplicate \( ri \)-tries of Figure 10 and Theorem 9 are required, structure sharing can be employed by an implementation to avoid storage of both.

**Theorem 9.** Let \( T = ri(\mathcal{F}, V) \), and let \( V' = \{p\} \cup V \), where \( p \) is the first variable in \( V' \). If \( p \in V \), \( V' \) is a reordering of \( V \) in which the only change is that \( p \) is made the first variable. Then

\[
ri(\mathcal{F} \land \{p\}, V') = (0, p, \neg p \lor T[1/p]^*, 0 \lor T[1/p]^*)
\]

\( \square \)

**Observation.** The conclusion of this theorem is very nice. This is not surprising since unit clauses are very useful in most settings. It does tell us that when unit clauses are present, the variables in them should almost certainly be first in the variable list.

Reordering variables may generally be useful and can be accomplished without adding a unit clause; indeed, without any modification of the knowledge base represented. Let \( T = ri(\mathcal{F}, V) \), where \( V = \{p_1, \ldots, p_n\} \), and let \( T' = ri(\mathcal{F}, V') \), where \( V' = \{p_i, p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n\} \). To determine \( T' \) from \( T \), apply the RIT operator to \( T \), using the new variable order:
\[
\text{RIT}(T, V') = \begin{align*}
p_i \lor \text{RIT}(T[0/p_i], V' - \{p_i\}) \\
\neg p_i \lor \text{RIT}(T[1/p_i], V' - \{p_i\}) \\
0 \lor \text{INT} \left( \text{RIT}(T[0/p_i], V' - \{p_i\}), \text{RIT}(T[1/p_i], V' - \{p_i\}) \right)
\end{align*}
\]

Notice that Theorem 8 applies to all four invocations of RIT on the right side; the result is Theorem 10.

**Theorem 10.** Let \( T = \text{RIT}(F, V) \), where \( V = \{p_1, \ldots, p_n\} \). Let \( T^+ = T[0/p_i]^* \), \( T^- = T[1/p_i]^* \), and \( T^0 = \text{INT}(T^+, T^-) \). If \( V' = \{p_i, p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n\} \), then

\[
\begin{align*}
p_i \lor T^+ \\
\neg p_i \lor T^- \\
0 \lor T^0
\end{align*}
\]

Consider now updating an ri-trie when a new clause is added to the knowledge base. Assume \( F \) has been compiled into ri-trie \( T \), and let \( C \) be the clause \( \{l_1, \ldots, l_m\} \). Our goal now is to compute the ri-trie \( T^C \) for \( F \land C \) without recompiling \( T \).

First observe that \( F \land C = \bigvee_{i=1}^m (F \land \{l_i\}) \). As a result, the ri-trie for \( F \land C \) is the intersection of the ri-tries for \( (F \land \{l_i\}) \), \( 1 \leq i \leq m \). The process for computing this intersection is easiest to describe as an iterative algorithm using pseudo-code. Each unit conjunction is computed using Theorem 9. The first unit conjunction initializes the ri-trie \( T \), and successive unit conjunctions are then intersected with \( T \). Since each unit conjunction reorders the variables, the variables of \( T \) must be reordered during each iteration using Theorem 10.

**function ADDCLAUSE**\( (T_F, \{l_1, \ldots, l_m\}) \)

\[
T \leftarrow (T_F \land \{l_1\}) \\
i \leftarrow 1 \\
\text{while } (i < m) \\
T' \leftarrow (T_F \land \{l_{i+1}\}) \\
\text{Recompute } T \text{ making the variable of } l_{i+1} \text{ first.} \\
T \leftarrow \text{INT}(T, T') \\
i \leftarrow i + 1 \\
\text{end while} \\
\text{return(T)}
\]

end ADDCLAUSE
Observe that the $m$ variables of $C$ become the first variables in the ordered variable set of $T$, but their order in $C$ is reversed. It will therefore be convenient to use the following notation for the final ordered variable set: Let $q_i$ be the variable of literal $l_i$; i.e., $l_i = q_i$ or $l_i = \neg q_i$. Let $V_C = \{q_1, \ldots, q_m\}$, and let $V$ be the original ordered variable set of $F$. Then $V^C = \{l_m, l_{m-1}, \ldots, l_1, (V - V_C)\}$ is the ordered variable set produced by the function ADDCLAUSE.

The correctness of the function ADDCLAUSE is an immediate consequence of Theorems 7, 8, 9, and 10.

**Theorem 11.** Let $T$ be the $ri$-trie for $F$ with variable ordering $V = \{p_1, \ldots, p_n\}$, and let $C = \{l_1, \ldots, l_m\}$ be a clause. Then the output of ADDCLAUSE($T, V^C$) is $ri(\mathcal{F} \land C, V^C)$.

\[\square\]

4 Conclusion

The results of the last section provide a variety of update operations that can be performed on or between $ri$-tries without recompiling. These include disjunction (Theorem 7), substituting a truth constant for a variable (Theorem 8), conjoining a unit clause (Theorem 9), adjusting the variable order (Theorem 10), and conjoining a new clause (Theorem 11). Because disjunction is accomplished with intersection, it is done between $ri$-tries, though both must be compiled with respect to the same variable ordering. Conjoining a clause to an $ri$-trie is accomplished whether or not some or all of the variables in the clause are among those that appear in the $ri$-trie.

A detailed analysis of the efficiency of the operations developed here is beyond the scope of this paper. Nevertheless, it is easy to see that other than conjoining a clause, all operations are no worse than linear in the size of the $ri$-trie. The operations could require visiting every node, but at each visited node the computation is $O(1)$. Conjoining a clause requires executing a loop whose duration is proportional to the size of the clause. The practicality of these operations is likely to be determined through experimentation, but updating is a desirable alternative to recompilation. An implementation of reduced implicate tries is currently under development; it is anticipated that these update operations will ultimately be incorporated.

5 Acknowledgments

The authors are grateful to the referees, each of whom provided useful suggestions and constructive criticism.
References