Primal Infon Logic with Conjunctions as Sets

Carlos Cotrini[1], Yuri Gurevich[2], Ori Lahav[3], and Artem Melentyev[4]

[1] Swiss Federal Institute of Technology, Switzerland
[2] Microsoft Research, Redmond, WA, USA
[3] Tel Aviv University, Israel
[4] Ural Federal University, Russia

Abstract. Primal infon logic was proposed by Gurevich and Neeman as an efficient yet expressive logic for policy and trust management. It is a propositional multimodal subintuitionistic logic decidable in linear time. However in that logic the principle of the replacement of equivalents fails. For example, $(x \wedge y) \to z$ does not entail $(y \wedge x) \to z$, and similarly $w \to ((x \wedge y) \wedge z)$ does not entail $w \to (x \wedge (y \wedge z))$. Imposing the full principle of the replacement of equivalents leads to an NP-hard logic according to a recent result of Beklemishev and Prokhorov. In this paper we suggest a way to regain the part of this principle restricted to conjunction: We introduce a version of propositional primal logic that treats conjunctions as sets, and show that the derivation problem for this logic can be decided in linear expected time and quadratic worst-case time.

1 Introduction

Propositional infon logic is a version of propositional multimodal intuitionistic logic [7]. It is applicable for policy and trust management but the derivability problem for propositional infon logic is PSpace-complete. Nevertheless, an expressive fragment of this logic, called *propositional primal infon logic* (**PIL**, in short), is decidable in linear time [7]. **PIL** is far below propositional infon logic in the time-complexity hierarchy. A natural problem arises how to extend the expressive power (and usefulness) of **PIL** keeping the logic feasible. In this paper, we present substantial progress toward this goal.

One of the main limitations of **PIL** is that it does not satisfy the principle of replacement of equivalents, that allows us to substitute a formula with an equivalent one in any context. For example, the formulas $x \wedge y$ and $y \wedge x$ are equivalent in **PIL** (i.e., each one is derivable from the other). However, $(x \wedge y) \to z$ and $(y \wedge x) \to z$ are not. In general, replacing a variable occurring in some formula with $x \wedge y$ is not the same as replacing it with $y \wedge x$. A similar situation occurs, e.g., with formulas of the form $(x \wedge y) \wedge w$ and $x \wedge (y \wedge w)$.

Imposing the full principle of replacement of equivalents on **PIL** makes it NP-hard [2]. Nevertheless, in this paper, we present an extension of **PIL**, called **SPIL**, that overcomes this limitation for conjunction. The idea behind **SPIL** is to treat conjunctions as sets of conjuncts (the 'S' in **SPIL** alludes to the word "set"). In other words any two conjunctions are viewed equivalent if the sets (not multisets!) of their conjuncts are the same, and the reasoning is done modulo this equivalence. For example, this equivalence relation identifies formulas $(x \wedge y) \to z$ and $(y \wedge x) \to z$.

This paper is organized as follows. First, we recall the syntax of **PIL** (Section 2). Then we define **SPIL** (Section 3), and we prove the local formula property for its Hilbertian calculus: any derivation of a formula X from a set Ω of formulas can be modified so that it uses only a small set of "local formulas" computable from $\Omega \cup \{X\}$. In Section 3.1 we present a Kripke-style semantics for **SPIL**. Finally, in Section 4, we present an efficient algorithm for the multi-derivability problem for **SPIL**. An implementation of the algorithm is available at `http://dkal.codeplex.com/` (in the context of Distributed Knowledge Authorization Language [3]).

Related work. We refer the reader to detailed related work sections: subsection 1.1 in the article [1] on propositional primal logic with disjunction, and section 6 in the article [5] on extensions of **PIL** with transitivity of implication. In addition, we note that proof systems in which derivations are performed modulo an equational theory between propositions were studied earlier in different contexts (see, e.g., [6]).

2 Preliminaries

We start with describing **PIL** (propositional primal infon logic), originally presented in [7]. We presume a set of propositional variables $\{v_1, v_2, \ldots\}$, a set of principal constants $\{p_1, p_2, \ldots\}$ and a constant $\top$ (used to denote an item of information that is known to all principals). The formulas of **PIL** are built from the propositional variables and $\top$ using the binary connectives $\wedge, \vee, \to$, and unary connectives of the form "q:" (called: *quotations*) where q ranges over principal constants. The intended meaning of a formula $q{:}x$ is that: the principal q said x. The size $sz(x)$ of a formula x is taken to be the number of connectives occurring in x. For any sequence of principal constants $q_1, q_2, \ldots, q_k$, we call the string $\boldsymbol{q} = q_1 : q_2 : \ldots q_k :$ a *quotation prefix* (where ϵ, the empty sequence, is a quotation prefix as well).

$$(\top)\ \frac{}{\boldsymbol{q}\,\top} \qquad (\wedge\mathrm{i})\ \frac{\boldsymbol{q}\,x \qquad \boldsymbol{q}\,y}{\boldsymbol{q}\ (x \wedge y)} \qquad (\wedge\mathrm{e})\ \frac{\boldsymbol{q}\ (x \wedge y)}{\boldsymbol{q}\,x} \qquad \frac{\boldsymbol{q}\ (x \wedge y)}{\boldsymbol{q}\,y}$$

$$(\vee\mathrm{i})\ \frac{\boldsymbol{q}\,x}{\boldsymbol{q}\ (x \vee y)} \qquad \frac{\boldsymbol{q}\,y}{\boldsymbol{q}\ (x \vee y)} \qquad (\to\mathrm{i})\ \frac{\boldsymbol{q}\,y}{\boldsymbol{q}\ (x \to y)} \qquad (\to\mathrm{e})\ \frac{\boldsymbol{q}\,x \qquad \boldsymbol{q}\ (x \to y)}{\boldsymbol{q}\,y}$$

Fig. 1. Calculus for **PIL**. $\boldsymbol{q}$ ranges over quotation prefixes and x, y over formulas.

Figure 1 provides a Hilbertian calculus defining **PIL**. For a set of formulas Γ (called *hypotheses*), a *derivation* D of a formula x from Γ in **PIL** is a finite tree such that each node u is labeled with a formula $D(u)$. The root is labeled with x and leaves are labeled with (instances of) axioms or formulas in Γ. If a node u has children $u_1, u_2, \ldots, u_n$, then $D(u_1), \ldots, D(u_n)$ / $D(u)$ is an instance of an inference rule. The *size* of the derivation is the number of nodes in this tree.

Given two sets of formulas Γ and Δ, the problem of deciding which formulas in Δ are derivable from Γ in **PIL** is called *the multi-derivability problem for* **PIL**. This problem is decidable in linear time [4,7].

3 The Logic SPIL

We present an extension of **PIL** that we call **SPIL**. The letter 'S' alludes to the word "set" and reflects our intention to treat conjunctions as sets of conjuncts. To define **SPIL** we use an auxiliary notion of *abstract formulas*.

Definition 1. An equivalence relation $\sim$ between formulas is defined as follows: $x \sim y$ if x and y are related according to the reflexive transitive symmetric closure of the rewriting relation induced by the following term rewriting system:[1]

- $(x_1 \wedge x_2) \longrightarrow (x_2 \wedge x_1)$
- $((x_1 \wedge x_2) \wedge x_3) \longrightarrow (x_1 \wedge (x_2 \wedge x_3))$
- $(x_1 \wedge x_1) \longrightarrow x_1$
- $(x_1 \wedge \top) \longrightarrow x_1$
- $q{:}(x_1 \wedge x_2) \longrightarrow (q{:}x_1) \wedge (q{:}x_2)$
- $q{:}\top \longrightarrow \top$

Roughly speaking, we have $x \sim y$ if x and y are the same formulas modulo the following properties of $\wedge$: commutativity, associativity, idempotence, contraction of the identity element $\top$, as well as the distributivity of quotations over $\wedge$.

Example 1. The formula $(v_1 \rightarrow p_1{:}((p_1{:}v_1) \wedge v_2))$ is equivalent to $(((v_1 \wedge v_1) \wedge \top) \rightarrow (p_1{:}v_2 \wedge (p_1{:}p_1{:}v_1)))$.

Definition 2. *Abstract formulas* are equivalence classes of formulas under $\sim$. The size $sz(X)$ of an abstract formula X is defined as $\min\{sz(x) \mid x \in X\}$.

We use $X, Y, \ldots$ as metavariables for abstract formulas. The equivalence class of a formula x under $\sim$ is denoted by $[x]$. Since abstract formulas play a dominant role in **SPIL**, we will refer to them simply as *formulas*, where true (non-abstract) formulas will be called *concrete formulas*. We define several operations on formulas.

Definition 3. For two formulas X, Y and connective $* \in \{\rightarrow, \vee\}$, $X * Y := [x * y]$ for some $x \in X$ and $y \in Y$. Similarly, for a formula X and a quotation prefix $\boldsymbol{q}$, $\boldsymbol{q}\,X := [\boldsymbol{q}\,x]$ for some $x \in X$.

Definition 4.

- A formula X is called *conjunctive* if $X = [\top]$ or $X = [x \wedge y]$ for concrete formulas x, y satisfying $x \not\sim y$, $x \not\sim \top$ and $y \not\sim \top$.

[1] Recall that rewriting rules of a term rewriting system may be applied under any context, and not necessarily on the topmost level.

- A finite set of non-conjunctive formulas with at least two elements is called a *conjunction set.*
- For a conjunction set S, $\bigwedge S := [(\cdots((x_1 \wedge x_2) \wedge x_3)\dots) \wedge x_n]$ for some concrete formulas $x_1,\dots,x_n$ such that $S = \{[x_1],\dots,[x_n]\}$.

It is easy to see that these operations are well-defined. In particular, the choices of concrete formulas is immaterial. Note that we use conjunction *sets* rather than *multisets*, and that, by definition, conjunction sets contain at least two members.

Proposition 1. *$\bigwedge S$ is conjunctive for every conjunction set S.*

The next proposition allows us to use inductive definitions and prove claims by induction on size of formulas.

Proposition 2. *Every formula X is either non-conjunctive and exactly one of the following holds:*

- *$sz(X) = 0$ and $X = [v]$ for a unique propositional variable v.*
- *$X = Y * Z$ for unique formulas Y and Z and $* \in \{\to, \vee\}$. In this case $sz(X) = sz(Y) + sz(Z) + 1$.*
- *$X = q{:}Y$ for unique principal constant q and formula Y. In this case $sz(X) = sz(Y) + 1$, and Y is non-conjunctive.*

or else X is conjunctive and either $sz(X) = 0$ and $X = [\top]$, or $X = \bigwedge S$ for a unique conjunction set S. In the latter case, $sz(Y) < sz(X)$ for every $Y \in S$.

$$(\tilde{\top})\ \frac{}{[\top]} \qquad (\tilde{\wedge}\mathrm{i})\ \frac{X_1 \quad X_2 \quad \dots \quad X_n}{\bigwedge S}\ \text{where } S = \{X_1,\dots,X_n\} \text{ and } n \geq 2$$

$$(\tilde{\wedge}\mathrm{e})\ \frac{\bigwedge S}{X}\ \text{where } X \in S \qquad (\tilde{\vee}\mathrm{i})\ \frac{\boldsymbol{q}\,X}{\boldsymbol{q}\ (X \vee Y)} \qquad \frac{\boldsymbol{q}\,Y}{\boldsymbol{q}\ (X \vee Y)}$$

$$(\tilde{\to}\mathrm{i})\ \frac{\boldsymbol{q}\,Y}{\boldsymbol{q}\ (X \to Y)} \qquad (\tilde{\to}\mathrm{e})\ \frac{\boldsymbol{q}\,X \qquad \boldsymbol{q}\ (X \to Y)}{\boldsymbol{q}\,Y}$$

Fig. 2. Calculus for **SPIL**. $\boldsymbol{q}$ ranges over quotation prefixes, X, Y over formulas, and S over conjunction sets.

SPIL is defined via the Hilbertian calculus given in Figure 2. The definitions of a derivation and its size are naturally adopted to this Hilbertian calculus. Note that derivations now consist of *abstract* formulas. We write $\Omega \vdash X$ to denote that the abstract formula X has a derivation from the set Ω of abstract formulas in **SPIL**.

Definition 5. The consequence relation $\vdash$ between concrete formulas in **SPIL** is given by: $\Gamma \vdash x$ if $\{[y] \mid y \in \Gamma\} \vdash [x]$.

Note that the language of the concrete formulas is that of **PIL**. Abstract formulas are used only for defining this consequence relation.

Theorem 1. *If Γ entails x in* **PIL***, then it does so in* **SPIL** *as well.*

Next, we show that **SPIL** enjoys a locality property similar to that of **PIL**, which allows one to confine derivations of X from Ω to those built from a certain small set of formulas computable from X and Ω. This property is essential for the correctness of the decision algorithm for **SPIL**.

Definition 6. The set of formulas that are *local* to a formula X is inductively defined by: (a) X is local to X; (b) If $\boldsymbol{q}\,(Y * Z)$ is local to X (for $* \in \{\rightarrow, \vee\}$ and quotation prefix $\boldsymbol{q}$) then so are $\boldsymbol{q}\,Y$ and $\boldsymbol{q}\,Z$; and (c) If $\bigwedge S$ is local to X (for conjunction set S) then so is every $Y \in S$. A formula is local *to a set* Ω of formulas if it is local to some $X \in \Omega$.

Definition 7. A derivation of a formula X from a set Ω of formulas is called *local* if all node formulas of the derivation are local to $\Omega \cup \{X\}$.

Theorem 2. *Any shortest derivation of X from Ω in* **SPIL** *is local.*

The following definition will be useful in the sequel.

Definition 8. A quotation prefix $\boldsymbol{q}$ is *local* to a formula X if some formula of the form $\boldsymbol{q}\,Y$ is local to X. A quotation prefix is local *to a set* Ω of formulas if it is local to some $X \in \Omega$.

3.1 Semantics

We adapt the semantics for **PIL** presented in [4,7] to **SPIL**.

Definition 9. A *Kripke model* is any structure M whose vocabulary comprises of (i) binary relations S_q where q ranges over the principal constants and (ii) unary relations V_X where X ranges over non-conjunctive formulas. The elements of (the universe of) M are called *worlds*.

Definition 10. Given a Kripke model M, we define when a world w *satisfies* a formula X, symbolically $w \vDash X$, by induction on $sz(X)$, distinguishing the cases according to Proposition 2:

1. $X = [\top]$: $w \vDash X$ for every w.
2. $X = [v]$ (where v is a propositional variable): $w \vDash X$ if $w \in V_{[v]}$.
3. $X = Y \rightarrow Z$: $w \vDash X$ if $w \vDash Z$ or ($w \nvDash Y$ and $w \in V_X$).
4. $X = Y \vee Z$: $w \vDash X$ if $w \vDash Y$ or $w \vDash Z$ or $w \in V_X$.
5. $X = q{:}Y$ (for non-conjunctive formula Y): $w \vDash X$ if $w' \vDash Y$ for all w' with wS_qw'.
6. $X = \bigwedge S$: $w \vDash X$ if $w \vDash Y$ for all $Y \in S$.

A world w satisfies *a set* Ω of formulas if it satisfies every $X \in \Omega$.

Theorem 3 (Soundness and Completeness). *Let Γ be a set of concrete formulas and x a concrete formula. $\Gamma \vdash x$ if and only if, for every Kripke model and world w, $w \vDash [x]$ whenever w satisfies $\{[y] \mid y \in \Gamma\}$.*

Remark 1. One of our referees wondered whether the full principle of replacement of equivalents holds in **SPIL**. It does not. Intuitively the reason is that, while **SPIL** generously enriches the algebra of conjunction, it imposes only mild restrictions on implication. Here is a example showing that the full principle of replacement of equivalents fails: $(x \wedge y) \to z \nvdash (x \wedge (x \to y)) \to z$ while $x \wedge y$ and $x \wedge (x \to y)$ are interderivable. This can be easily verified using our Kripke semantics.

4 A Decision Algorithm

In this section we present an efficient decision algorithm for the the multi-derivability problem for **SPIL**.

Definition 11. The *multi-derivability problem for* **SPIL** is defined as follows. Given two sequences of concrete formulas, called *concrete hypotheses* and *concrete queries* respectively, decide which concrete queries are derivable from the concrete hypotheses in **SPIL**, and print them.

Theorem 4. *There is a randomized algorithm that solves the multi-derivability problem for* **SPIL** *in expected linear time and worst-case quadratic time.*

Note that in "expected linear time" the average is taken for internal random choices during the execution, while assuming any input. We employ the same standard computation model of analysis of algorithms used in [4], according to which the registers are of size $O(\log n)$ where n is the size of the input, and the basic register operations are constant time. We also presume a function Random that generates $\lceil \log(n) \rceil$ random bits in constant time.

The rest of this paper is devoted to prove Theorem 4. The algorithm has two main stages. First, we construct a data structure that succinctly represents the input (Sections 4.1 and 4.2). Then, we use this data structure to compute the derivable concrete queries (Section 4.3).

4.1 Input Parse Dag and Local Prefixes Dictionary

We refer to the abstract formulas that correspond to the concrete hypotheses simply as *hypotheses*, and similarly to these of the concrete queries as *queries*. A formula (quotation prefix) is called *a local input formula (local prefix)* if it is local to the set of hypotheses or the set of queries (see Definitions 6 and 8). The input is represented in a directed acyclic graph (dag, for short) data structure.[2] We assume that each node u is uniquely identified by a constant-size key, denoted by Key(u) (e.g., its memory address), stores the keys of its children in a list Ch(u), and of its parents in a corresponding list Pa(u). To handle quotation prefixes, we will use of the following auxiliary data structure:

[2] By graph we actually mean multigraph, where parallel edges are allowed.

Definition 12. A *local prefixes dictionary* for a given input is a data structure that assigns a unique constant-size key $\mathrm{Key}(\boldsymbol{q})$ to every local input quotation prefix $\boldsymbol{q}$. Given such a key k, we will denote by $\mathrm{Prf}(k)$ the quotation prefix $\boldsymbol{q}$ such that $\mathrm{Key}(\boldsymbol{q}) = k$.

Note that the *trie of local prefixes* as defined in [4] is a particular implementation of a local prefixes dictionary, where $\mathrm{Key}(\boldsymbol{q})$ is taken to be the memory address of the trie node that corresponds to $\boldsymbol{q}$. Given a local prefixes dictionary, our dag data structures are defined as follows.

Definition 13. A *parse dag* is a rooted dag in which every node u is decorated with two additional (constant-size) fields: $\mathrm{Label}(u)$ and $\mathrm{PrfKey}(u)$. Its root r has two children denoted by r_h and r_q, where $\mathrm{Label}(r) = \mathrm{Label}(r_h) = \mathrm{Label}(r_q) =$ `nil` and $\mathrm{PrfKey}(r) = \mathrm{PrfKey}(r_h) = \mathrm{PrfKey}(r_q) = \mathrm{Key}(\epsilon)$. All other nodes are called *regular nodes*. For each regular node u, $\mathrm{Label}(u)$ is $\top$,$\rightarrow$,$\vee$,$\wedge$ or a propositional variable, and $\mathrm{PrfKey}(u)$ holds a key of some local input quotation prefix, such that:

1. If $\mathrm{Label}(u)$ is $\top$ or a propositional variable, then $\mathrm{Ch}(u)$ is empty.
2. If $\mathrm{Label}(u)$ is $\rightarrow$ or $\vee$, then $\mathrm{Ch}(u)$ contains exactly two keys.
3. If $\mathrm{Label}(u)$ is $\wedge$, then $\mathrm{Ch}(u)$ contains at least one key.
4. If u is a child of v, then $\mathrm{Prf}(\mathrm{PrfKey}(v))$ is a prefix of $\mathrm{Prf}(\mathrm{PrfKey}(u))$.

Each node in a parse dag naturally represents a (concrete and abstract) formula. Formally, this relation is defined as follows.

Notation 5. *For a regular node u, we denote $\mathit{Prf}(\mathit{PrfKey}(u))$ by $\mathit{Prf}(u)$.*

Notation 6. *Given two quotation prefixes $\boldsymbol{q}$ and $\boldsymbol{p}$, we denote by $\boldsymbol{p}\backslash\boldsymbol{q}$ the quotation prefix $\boldsymbol{r}$, such that $\boldsymbol{pr} = \boldsymbol{q}$, or ϵ if such $\boldsymbol{r}$ does not exist.*

Definition 14. The *complete concrete formula of a regular node u with respect to a quotation prefix $\boldsymbol{q}$* is denoted by $\mathrm{F}(u, \boldsymbol{q})$, and defined by:

1. If u has no children, then $\mathrm{F}(u, \boldsymbol{q}) = (\boldsymbol{q}\backslash\mathrm{Prf}(u))\mathrm{Label}(u)$.
2. If $\mathrm{Label}(u) = *$ for $* \in \{\rightarrow, \vee\}$, then $\mathrm{F}(u, \boldsymbol{q})$ is $(\boldsymbol{q}\backslash\mathrm{Prf}(u))$ $(\mathrm{F}(u_1, \mathrm{Prf}(u)) * \mathrm{F}(u_2, \mathrm{Prf}(u)))$ where u_1 and u_2 are the first and second children of u (respectively).
3. If $\mathrm{Label}(u) = \wedge$, then $\mathrm{F}(u, \boldsymbol{q})$ is $(\boldsymbol{q}\backslash\mathrm{Prf}(u))((\cdots(\mathrm{F}(u_1, \mathrm{Prf}(u)) \wedge \mathrm{F}(u_2, \mathrm{Prf}(u)))\ldots) \wedge \mathrm{F}(u_k, \mathrm{Prf}(u)))$ where $u_l, \ldots, u_k$ are u's children in the order they occur in $\mathrm{Ch}(u)$.

The *complete concrete formula of a regular node u* is denoted by $\mathrm{F}(u)$ and defined to be $\mathrm{F}(u, \epsilon)$. The *complete (abstract) formula of a regular node u* is denoted by $\tilde{\mathrm{F}}(u)$ and defined to be $[\mathrm{F}(u)]$.

Definition 15. A parse dag *for input* $x_1, \ldots, x_k \vdash y_1, \ldots, y_m$ is any parse dag that satisfies the following conditions: (1) $\{\tilde{\mathrm{F}}(u) \mid \mathrm{Key}(u) \in \mathrm{Ch}(r_h)\} = \{[x_1], \ldots, [x_k]\}$; (2) $\{\tilde{\mathrm{F}}(u) \mid \mathrm{Key}(u) \in \mathrm{Ch}(r_q)\} = \{[y_1], \ldots, [y_m]\}$; and (3) Every child u of r_q is decorated with a list $\mathrm{Inputs}(u)$ of all y_i's that satisfy $y_i \in \tilde{\mathrm{F}}(u)$.

Note that the input parse tree as defined in [4] (ignoring the edge labels) is also an input parse dag. For the next stage, we should ensure that there are no two different nodes that represent the same formula. Thus we are interested in a *compressed* input parse dag, as defined next.

Definition 16. A node u in a parse dag D is *unique* if $\tilde{\mathrm{F}}(u') \neq \tilde{\mathrm{F}}(u)$ for any $u' \neq u$. D is called *compressed* if its nodes are all unique, and $\mathrm{Label}(u)$ is not $\wedge$ or $\top$ whenever u is a child of a node labeled with $\wedge$.

Proposition 3. *Consider a compressed input parse dag. For every local input formula X, there is exactly one regular node u such that $\tilde{F}(u) = X$.*

Theorem 7. *There is a randomized algorithm with expected linear time and worst-case quadratic time complexities, that constructs a local prefixes dictionary and a compressed input parse dag for a given input.*

4.2 Construction of a Compressed Input Parse Dag

This section is devoted to prove Theorem 7. To facilitate the exposition and the analysis, we will present this algorithm as a composition of sub-algorithms. Initially, we construct (in linear time) a local prefixes dictionary and an initial (uncompressed) input parse dag (in the form of a trie of local prefixes and an input parse tree) exactly as done in [4]. It remains to modify the parse tree into a compressed parse dag. First, we reformat the tree as detailed in Algorithm 4.1. Roughly speaking, this step accounts for the associativity of conjunction. Its time complexity is $O(N)$, where N denotes the number of leaves in the initial parse tree.

Algorithm 4.1. Initial reformatting

```
1: Traverse the initial parse dag in depth-first manner. Suppose that $u$, the node
   currently visited, is labeled with $\wedge$ or $\top$, and that its parent $v$ is labeled with $\wedge$.
   In that case, for each child $w$ of $u$, make $w$ a child of $v$, and delete $u$ from the dag.
   If $v$ is left with no children, set its label to $\top$.
```

Time complexity: $O(N)$

Next, we "compress" the resulting tree into a dag. This process requires several additional data structures and fields:

1. A work list C of length N, initialized with (the keys of) all leaf nodes.
2. Two auxiliary arrays A and HT (Hash Table) of length $M = 2^{\lceil \log_2(N) \rceil}$. The entries of A are node keys, while those of HT are lists of keys. Initially $A[i] = -1$ and $HT[i] = \emptyset$ for all $i < M$.
3. Numeric fields $\mathrm{Counter}(u)$ and $\mathrm{Hash}(u)$ for each node u, initialized with 0 and a random number $< M$ (respectively).

The compression works iteratively using the work list C. In each iteration, the nodes in C are made unique. Initially, C includes all leaf nodes. When a node u is removed from C, it increments the counter in its parent v. If $\mathrm{Counter}(v)$ reaches $|\mathrm{Ch}(v)|$ (the length of the list $\mathrm{Ch}(v)$), v is added to C for the next iteration. Thus, the following invariant is preserved:

Invariant 8. *Children of nodes in C are all unique, and each node in C or ancestor of a node in C has a unique parent node.*

Compression of the Leaves. Before the first iteration, the work list C includes all leaf nodes. They are compressed using a *plagiarism checker*:

Definition 17. An element a_j in an array $L = (a_0, \ldots, a_{k-1})$ is *original* if $a_i \neq a_j$ for any $i < j$. If a_i is original, $i \leq j$ and $a_i = a_j$ then a_i is the *original of a_j*. A *plagiarism checker* for the array L is an array B of length k such that every $a_{B[j]}$ is the original of a_j.

Theorem 9. *There is an algorithm that, given an array L of d-tuples of natural numbers $< M$ and an array A of length M initialized with (-1)'s, computes the plagiarism checker B for L and re-initializes the array A with (-1)'s (so it can be reused to compute future plagiarism checkers). This algorithm takes $O(|L|d)$ time.*

The plagiarism checker is computed on an array L that includes the *extended labels* of the leaf nodes. For each node u in C, the extended label of u, denoted by $\mathrm{EL}(u)$, is a constant-size tuple that satisfies the following property: for every two nodes u_1 and u_2 in C, $\mathrm{EL}(u_1) = \mathrm{EL}(u_2)$ iff $\tilde{\mathrm{F}}(u_1) = \tilde{\mathrm{F}}(u_2)$. For a leaf node u, the extended label $\mathrm{EL}(u)$ is taken to be the ordered pair $(\mathrm{PrfKey}(u), \mathrm{Label}(u))$ if $\mathrm{Label}(u)$ is a propositional variable, or just $\top$ if $\mathrm{Label}(u) = \top$. It is easy to see that this definition of $\mathrm{EL}(u)$ guarantees the required property for the (leaf) nodes in C. From this observation, Algorithm 4.2 for compression of the leaves follows. Note that computing the extended label of a leaf takes constant time. Thus computing L and U takes $O(N)$ time. Since extended labels have constant length, the plagiarism checker can be computed in $O(N)$ time as well.

Compression of Internal Nodes. After applying Algorithm 4.2 in the first iteration, all leaf nodes are unique. In addition, Algorithm 4.2 prepares it for the next iteration, so it includes all nodes whose children are all unique. In fact, the next iteration also applies Algorithm 4.2, with a different definition of the extended labels. We refer to the nodes of C whose label is $\rightarrow$ or $\vee$ as *binary nodes*, and to these whose label is $\wedge$ as *set nodes*. The extended label for the binary nodes is simple, as we can take $\mathrm{EL}(u)$ of a binary node u in C with first child v and second child w to be the ordered tuple $(\mathrm{PrfKey}(u), \mathrm{Label}(u), \mathrm{Key}(v), \mathrm{Key}(w))$.

Proposition 4. *For two binary nodes u_1 and u_2 in C, $\mathrm{EL}(u_1) = \mathrm{EL}(u_2)$ iff $\tilde{F}(u_1) = \tilde{F}(u_2)$.*

Algorithm 4.2. Compression of the nodes in the work list

```
1: Copy the work list $C$ to an array $U$, and set $C \leftarrow \emptyset$.
2: Compute an array $L$ with corresponding extended labels, i.e. $L[i] = \mathrm{EL}(U[i])$.
3: Compute the plagiarism checker $B$ of $L$.                          ▷ Theorem 9
4: For $i$ from 0 to $|U| - 1$
5:     Let $u$ and $w$ be the nodes with keys $U[i]$ and $U[B[i]]$ (respectively).
6:     Let $v$ be the parent of $u$.
7:     If $i \neq B[i]$ then
8:         Remove $u$ from the parse dag.
9:         If $v = r_q$, append $Inputs(u)$ to $Inputs(w)$.
10:        Replace $U[i]$ in $\mathrm{Ch}(v)$ by $U[B[i]]$.
11:    Increment Counter($v$).
12:    If $v$ is regular and $\mathrm{Counter}(v) = |\mathrm{Ch}(v)|$ then add $v$ to the work list $C$.
```

Time complexity: $O(|C|)$

The compression of the set nodes, however, is more involved, and requires some preprocessing to account for the idempotence of $\wedge$, and to compute the extended labels of the set nodes. Several additional notations are used in this preprocessing stage:

- N_C denotes the sum $\sum |\mathrm{Ch}(u)|$ for all set nodes in C.
- For each set node u in C, $CH(u) = \{w \mid w \text{ is a child of } u\}$.

The preprocessing for the set nodes consists of two steps. First, we reformat the parse dag, by removing duplicate children of set nodes, as well as contracting set nodes that are left with only one child (this may add new binary nodes to C). Algorithm 4.3 provides the technical details. Intuitively, this step accounts for the idempotence of $\wedge$.

Algorithm 4.3. Reformatting of set nodes in C

```
1: For each set node $u$ of in $C$
2:     Copy $\mathrm{Ch}(u)$ to an array $U$.
3:     Compute the plagiarism checker $B$ of $\mathrm{Ch}(u)$.           ▷ Theorem 9
4:     $\mathrm{Ch}(u) \leftarrow \emptyset$.
5:     For $i$ from 0 to $|B|$
6:             If $B[i] = i$, then append $U[i]$ to $\mathrm{Ch}(u)$.
7:     If $|\mathrm{Ch}(u)| = 1$ then
8:         Let $v$ be $u$'s parent and $w$ be $u$'s child.
9:         Remove $u$ from parse dag and replace it with $w$ in $\mathrm{Ch}(v)$.
10:        If $v = r_q$, append Inputs($u$) to Inputs($w$)
11:        Increment Counter($v$).
12:        If $v$ is regular and $\mathrm{Counter}(v) = |\mathrm{Ch}(v)|$, add $v$ to the work list $C$.
```

Time complexity: $O(N_C)$.

Next, we compute the extended labels for the set nodes. This step involves a hash table, where the hash function assigns to each node u the initially chosen random number Hash(u). Note that $(\wedge, \mathrm{Ch}(u))$ cannot serve as an extended label (since two set nodes with different permutations of the same list of children would have diffrent extended labels).

Compute Extended Labels. We assume that each set node u is decorated with an additional field called *set label* and denoted by SL(u). For each set node u in C, SL(u) is initialized with $\mathrm{Hash}(u_1) \oplus \cdots \oplus \mathrm{Hash}(u_k)$, where $u_1, \ldots, u_k$ are the children of u whose keys are listed in Ch(u), and $\oplus$ is the bitwise XOR operation (between binary representations of numbers). The computation of SL(u) takes $O(|\mathrm{Ch}(u)|)$ time for each node u. Hence the computation for all set nodes in C takes $O(N_C)$ time. Note that for two set nodes u and v with $CH(u) = CH(v)$, we have $\mathrm{SL}(u) = \mathrm{SL}(v)$; the converse, however, is "almost always" true as the following lemma shows.

Lemma 1. *For every two set nodes,* $P(\mathrm{SL}(u) = \mathrm{SL}(v))$ *is* 1 *if* $CH(u) = CH(v)$, *and* $1/M$ *otherwise.*

It follows that SL cannot serve as an extended label for the set nodes. To generate the extended labels EL, we use a hash table for detecting and fixing collisions in SL. This is described in Algorithm 4.4. We explain the time complexity for this computation. Let $u_1, u_2, \ldots, u_{|C|}$ be the set nodes in C and suppose that the loop in line 1 process them in that order. For $i < j \leq |C|$, let X_{ij} be a random variable which takes value 1, if $\mathrm{EL}\,(u_i) = (\wedge, \mathrm{Key}(u_i))$ and $\mathrm{SL}(u_i) = \mathrm{SL}(u_j)$; and 0, otherwise. Let T be the random variable that gives the time complexity for this computation. T is the sum over j of the time needed to compute $\mathrm{EL}(u_j)$. To compute $\mathrm{EL}(u_j)$ we check the nodes in the entry $HT\,[\mathrm{SL}(u_j)]$. The length of $HT\,[\mathrm{SL}(u_j)]$ is $\sum_{i=1}^{j-1} X_{ij}$ and each comparison (the check if $CH(u) = CH(v)$) takes $O(|\mathrm{Ch}(u_j)|)$ time. Therefore, $T = \sum_{j=1}^{|C|} \left(O(|\mathrm{Ch}(u_j)|) \cdot \sum_{i=1}^{j-1} X_{ij} \right)$. In the worst-case, for any two set nodes u and v we have $\mathrm{SL}(u) = \mathrm{SL}(v)$ but $CH(u) \neq CH(v)$. This yields an execution time of $O\,(|C| \cdot N_C)$. To see that $E(T) = O\,(N_C)$, it suffices to show that $\sum_{i=1}^{j-1} E(X_{ij})$ is constant. Now, $E\,(X_{ij}) = P\,(X_{ij} = 1) \leq P\,(\mathrm{SL}\,(u_i) = \mathrm{SL}\,(u_j))$, and by Lemma 1 we obtain that $E\,(X_{ij}) \leq 1$ if $CH(u_i) = CH(u_j)$, and $E\,(X_{ij}) \leq 1/M$ otherwise. Algorithm 4.4 stores at most one set node u_i with $CH(u_i) = CH(u_j)$ in $HT\,[\mathrm{SL}(u_j)]$. Hence, $\sum_{i=1}^{j-1} E(X_{ij}) \leq 1 + \frac{j-2}{M} \leq 1 + \frac{N}{M} \leq 2$.

Compression of Internal Nodes. Equipped with extended labels for all nodes in the work list C, we apply the compression for these nodes. Since each two nodes u_1 and u_2 in C have $\tilde{\mathrm{F}}(u_1) = \tilde{\mathrm{F}}(u_2)$ iff $\mathrm{EL}(u_1) = \mathrm{EL}(u_2)$, we can compress the nodes in C exactly as we did for the leaves using Algorithm 4.2. This algorithm also prepares C for the next iteration.

This concludes the computation of a compressed parse dag from the parse tree. Algorithm 4.5 gives a summary of this construction. To see the time complexity of Algorithm 4.5, note that the inner step of the loop takes expected time proportional to the number of nodes in C plus the number of their children. Since every node is added to C exactly one time, summing this over all iterations we

Algorithm 4.4. Computing extended labels for set nodes

```
1: For each set node $u$ in $C$
2:     If $CH(u) = CH(v)$ for some $v \in HT[\mathrm{SL}(u)]$ then
3:         $\mathrm{EL}(u) \leftarrow (\wedge, \mathrm{Key}(v))$.
4:     Else
5:         $\mathrm{EL}(u) \leftarrow (\wedge, \mathrm{Key}(u))$.
6:         Append $u$ to $HT[\mathrm{SL}(u)]$.
7: For each set node $u$ in $C$, set $HT[\mathrm{SL}(u)] \leftarrow \emptyset$.
```

Expected time complexity: $O(N_C)$
Time complexity (worst-case): $O(|C| \cdot N_C)$

get expected time proportional to the number of nodes. In a similar way, we get $O(N^2)$ time in the worst-case for the inner loop. The complexities of all other steps were explained above. Finally, note that N (the number of leaves in the initial input tree) is clearly less than the length of the input.

Algorithm 4.5. Construction of a compressed parse dag

```
1: Construct a parse tree and a local prefixes dictionary
2: Perform initial reformatting of the parse tree.                 ▷ Algorithm 4.1
3: Construct a work list $C$ initialized with a list of the (keys of) leaf nodes, field
   Counter($u$) for each node $u$ initialized with 0, arrays $A = (-1, -1, \dots, -1)$ and
   $HT = (\emptyset, \emptyset, \dots, \emptyset)$ of length $M$, and a field Hash($u$) for each node $u$ initialized
   with a random number $< M$.
4: Compress the nodes in $C$.                                      ▷ Algorithm 4.2
5: While $C$ is not empty
6:   Reformat the set nodes.                                       ▷ Algorithm 4.3
7:   Compute a set label $\mathrm{SL}(u)$ for every set node $u$.
8:   Compute an extended label $\mathrm{EL}(u)$ for every set node $u$.    ▷ Algorithm 4.4
9:   Compress the nodes in $C$.                                    ▷ Algorithm 4.2
```

Expected time complexity: $O(N)$
Worst time complexity: $O(N^2)$

4.3 Deriving Local Formulas

The second stage of algorithm computes all derivable queries. This is done similarly to the corresponding stage for **PIL** [4]. First, we traverse the parse dag and decorate each regular node u with a boolean flag Der(u). It is initialized to 0, unless Label(u) = $\top$ or u represents a hypothesis (u is a child of r_h) in which case Der(u) is initialized to 1. Der(u) = 0 indicates that $\tilde{\mathrm{F}}(u)$ has not been derived from the hypotheses yet, and in this case we say that u is *raw*. Der(u) = 1 indicates that $\tilde{\mathrm{F}}(u)$ has been derived from the hypotheses, and u is called *pending*. We also construct a *pending queue*, that contains all pending nodes. To *make pending* a node u means to insert u to the pending queue and set Der(u) = 1. The following invariant holds throughout the execution of the algorithm.

Invariant 10. *Whenever a node u becomes pending, the formula $\tilde{F}(u)$ is derivable from the hypotheses in* **SPIL**.

To *apply a rule R to u* means to make pending every raw node w for which there are pending nodes $v_1, v_2, \ldots, v_k$, such that $u \in \{v_1, v_2, \ldots, v_k\}$ and $\tilde{\mathrm{F}}(v_1), \ldots, \tilde{\mathrm{F}}(v_k)$ / $\tilde{\mathrm{F}}(w)$ is an instance of the rule R. The algorithm repeatedly takes a node u from the pending queue, applies as many rules to it as possible and then removes u from the pending queue. The algorithm terminates when the pending queue is empty. We explain how to apply each rule R to a node u, and show (for several cases) that these applications preserve Invariant 10. Note that an additional numeric field Counter(u) (initialized to 0) is used for each node u labeled with $\wedge$.

($\tilde{\wedge}$e) If Label(u) $= \wedge$, then make pending every raw child of u.
Justification: Let $u_1, \ldots, u_k$ be the children of u in the order they appear in Ch(u). Then $\tilde{\mathrm{F}}(u) = [\mathrm{Prf}(u)((\cdots(\mathrm{F}(u_1, \mathrm{Prf}(u)) \wedge \mathrm{F}(u_2, \mathrm{Prf}(u)) \ldots) \wedge \mathrm{F}(u_k, \mathrm{Prf}(u)))]$. Since u is pending, $\tilde{\mathrm{F}}(u)$ is derivable. This entails that $\tilde{\mathrm{F}}(u_i) = [\mathrm{Prf}(u)(\mathrm{F}(u_i, \mathrm{Prf}(u))]$ is derivable as well. To see this, note that $\boldsymbol{q}\, x_i$ is derivable from $\boldsymbol{q}\,((\cdots(x_1 \wedge x_2) \ldots) \wedge x_k)$ in **PIL** for every concrete formulas $x_1, \ldots, x_k$, quotation prefix $\boldsymbol{q}$, and $1 \leq i \leq k$. By Theorem 1, we have that $\tilde{\mathrm{F}}(u) \vdash \tilde{\mathrm{F}}(u_i)$.

($\tilde{\wedge}$i) For every raw parent w of u labeled with $\wedge$, increment Counter(w) and make w pending if it exceeds the number of children of w.
Justification: Let $u_1, \ldots, u_k$ be the children of w in the order they appear in Ch(w). Then $\tilde{\mathrm{F}}(w) = [\mathrm{Prf}(w)((\cdots(\mathrm{F}(u_1, \mathrm{Prf}(w)) \wedge \mathrm{F}(u_2, \mathrm{Prf}(w)) \ldots) \wedge \mathrm{F}(u_k, \mathrm{Prf}(w)))]$. If Counter($w$) was incremented k times, then each $\tilde{\mathrm{F}}(u_i) = [\mathrm{Prf}(w)(\mathrm{F}(u_i, \mathrm{Prf}(w))]$ is derivable. This entails that $\tilde{\mathrm{F}}(w)$ is derivable as well (again using Theorem 1, since $\boldsymbol{q}\,((\cdots(x_1 \wedge x_2) \ldots) \wedge x_k)$ is derivable from $\boldsymbol{q}\, x_1, \ldots, \boldsymbol{q}\, x_k$ in **PIL** for every concrete formulas $x_1, \ldots, x_k$ and quotation prefix $\boldsymbol{q}$).

($\tilde{\vee}$i) Make pending every raw parent w of u labeled with $\vee$.

($\tilde{\rightarrow}$i) For every raw parent w of u such that Label(w) is $\rightarrow$ and u is the second child of w, make w pending.

($\tilde{\rightarrow}$e) $\tilde{\mathrm{F}}(u)$ can be used as the left or the right premise of ($\tilde{\rightarrow}$e). Accordingly, we have two substeps: (1) For every pending parent w of u, such that Label(w) is $\rightarrow$ and u is the first child of w, make the second child of w pending if it is raw; (2) If Label(u) is $\rightarrow$ and the first child u_1 of u is pending, then make pending the second child u_2 of u if it is raw.

When the pending queue is empty, the algorithm prints a list of the derivable concrete queries. To do so, walk through the nodes $u_1, \ldots, u_m$ that represent queries (i.e. the children of the node r_q). If Der(u_i) $= 1$ then print the strings in Inputs(u_i). Since separate concrete queries are separate segments of the input, the printing process takes linear time.

Theorem 11. *The decision algorithm for* **SPIL** *is sound and complete, and it works in expected linear time and quadratic time in the worst-case.*

References

1. Beklemishev, L., Gurevich, Y.: Propositional primal logic with disjunction. Journal of Logic and Computation 24(1), 257–282 (2014)
2. Beklemishev, L., Prokhorov, I.: On computationally efficient subsystems of propositional logic (to appear)
3. Blass, A., De Caso, G., Gurevich, Y.: An Introduction to DKAL. Microsoft Research technical report, MSR-TR-2012-108 (2012)
4. Cotrini, C., Gurevich, Y.: Basic primal infon logic. Journal of Logic and Computation (2013)
5. Cotrini, C., Gurevich, Y.: Transitive primal infon logic. The Review of Symbolic Logic 6, 281–304 (2013)
6. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. J. Autom. Reason. 31(1), 33–72 (2003)
7. Gurevich, Y., Neeman, I.: Logic of infons: The propositional case. ACM Trans. Comput. Logic 12(2), 9:1–9:28 (2011)