Basic primal infon logic

CARLOS COTRINI, *Swiss Federal Institute of Technology*

YURI GUREVICH, *Microsoft Research*

Abstract

Primal infon logic (PIL) was introduced in 2009 in the framework of policy and trust management. In the meantime, some generalizations appeared, and there have been some changes in the syntax of the basic PIL. This article is on the basic PIL, and one of our purposes is to 'institutionalize' the changes. We prove a small-model theorem for the propositional fragment of basic primal infon logic (PPIL), give a simple proof of the PPIL locality theorem and present a linear-time decision algorithm (announced earlier) for PPIL in a form convenient for generalizations. For the sake of completeness, we cover the universal fragment of basic PIL. We wish that this article becomes a standard reference on basic PIL.

Keywords: Authorization language, basic primal logic, DKAL, infon logic, Kripke semantics, linear time algorithm.

1 Introduction

Primal infon logic (PIL) was discovered in the framework of research on policy and trust management [15]. It was generalized in [4, 10], and more generalizations are in preparation. This article is on the basic version of PIL.

Infons. Principals of a distributed system (people, organizations and computers) communicate by sending each other items of information called infons in [14]. Infons often look like formulas in first-order logic, but it may make no sense to ask whether an infon is true or false. The meaningful question is whether the infon is known to a principal.

Infon algebra. Infons are naturally preordered by the relation 'at least as informative as.' The preorder contains a known-to-all infon $\top$ on one extreme and an unknowable infon $\bot$ on the other extreme. In accordance with the venerable logic tradition to have $\top$ at the top, we write $y \leq x$ to express that y is at least as informative as x.

The conjunction of infons is their union, so that it is at least as informative as either of the conjuncts, and it is a least informative infon with this property. The preorder of infons gives rise to a partial order if one identifies equally informative infons. Conjunction turns that partial order into a semilattice. On the semilattice live unary operations p `said` where p ranges over the principals. If infons x and y are equally informative, then so are p `said` x and p `said` y.

From infon algebra to propositional PIL. Implication $a \to b$ can be introduced as a solution of inequalities $(a \wedge x) \leq b$ and $b \leq x$. Similarly, disjunction $a \vee b$ can be introduced as a solution of inequalities $a \leq x$ and $b \leq x$. This leads to basic propositional primal infon logic (PPIL) studied in [13, 15, 16]. PPIL has a remarkable combination of expressivity and feasibility. PPIL expresses typical policy scenarios that do not involve substitutions, and the multi-derivability problem for PPIL (decide which of given queries follow from given hypotheses) is solvable in linear-time [13].

Published online July 22, 2013 doi:10.1093/logcom/ext021

Requiring that there is a greatest (that is least informative) solution of the inequalities $(a \wedge x) \leq b$ and $b \leq x$ leads to intuitionistic implication, and requiring that there is a smallest (that is most informative) solution of the inequalities $a \leq x$ and $b \leq x$ leads to intuitionistic disjunction [3] (L. Beklemishev *et al.*, in preparation). In contrast to PPIL, the derivability problem (decide whether a given query follows from given hypotheses) for the $\rightarrow$ fragment of intuitionistic logic is polynomial-space complete [19]. The derivability problem for the combination of conjunction, primal implication and intuitionistic disjunction is NP-complete [4]. The contrast makes PPIL and its feasible extensions attractive in applications.

Basic PIL. Of course, policy scenarios may and often do involve object variables, e.g. 'Every password contains digits and letters,' or 'Any employee has a manager.' It is typically presumed that the object variables are universally quantified. That leads to the universal fragment of PIL [6].

One modification of basic PIL. Originally, in addition to unary connectives p `said`, PIL had unary connectives p `implied`, and it was postulated that p `said` x entails p `implied` x. The intention was to increase the expressivity of Distributed Knowledge Authorization Language DKAL [7, 14, 15]. But DKAL customers found the `said/implied` interplay confusing. Further development of DKAL made the use of connectives p `implied` unnecessary, and the `implied` construct was removed from PIL [13].

This article. This article is self-contained. We intend to make it a standard reference on basic PIL, especially on its propositional part PPIL. The main distinguishing features of our version of PPIL and this article in general, comparative with their predecessors, in particular [16], are as follows.

- *Connectives* The propositional connectives of the original PPIL were $\wedge$, $\rightarrow$, $\top$, p `said` and p `implied` [16]. Here the connectives are $\wedge, \vee, \rightarrow, \top, \bot$ and p `said`. We already explained the reason for dropping `implied`. In the meantime, we realized the necessity of connectives $\bot$ and $\vee$. $\bot$ is necessary (in addition to $\top$) because of the need to translate Boolean values, `true` and `false` to infons. Disjunction is necessary in particular because policy rules often have the form $(x_1 \wedge \cdots \wedge x_k) \rightarrow y$, where y is atomic and every x_i is atomic or a disjunction of atomic formulas. For example, 'An employee can read File 13 if he/she works on project 7 or project 11 and is a US or Canadian citizen.' It is easy to see that any such rule can be faithfully rewritten as several disjunction-free rules, but this may be prohibitively expensive.
- *Locality theorem* The locality theorem for PPIL [16, Theorem 5.11] asserts this: if hypotheses Γ entail a formula φ, then there is a derivation of φ from Γ composed from formulas local to $\Gamma \cup \{\varphi\}$ in some precise sense. Our proof of the locality theorem is much simplified.
- *Complexity* It is proven in [16] that the multi-derivability problem (decide which of given queries follow from given hypotheses) is linear-time for any fragment of the original PPIL with bounded quotation depth. The presence of `implied` and the resulting opportunity to alternate `said` and `implied` make the restriction on the quotation depth necessary. Here we prove the result, announced in [13], that the multi-derivability problem is linear-time for the whole of our PPIL. (It is claimed in [2] (A. Baskar *et al.*, in preparation) that the derivability problem, and thus the multi-derivability problem, for the whole original PPIL is polynomial time.)
- *Decision algorithm* Our linear-time decision algorithm for (the multi-derivability problem for) PPIL is a simplification of that in [16]. One change is related to the need to decide in constant-time (after some preprocessing) whether two nodes of a parse tree represent identical formulas. Instead of the Cai–Paige algorithm [8], used in [16], here we use suffix arrays that are better known and have efficient implementations available. Also, we use compact parse trees of [5]. Following [6], we extend (in §6) the decision algorithm for PPIL to that for the universal fragment of PIL. The extended algorithm has been implemented and is available at [11].

- *Witness extraction* We extend the decision algorithm to extract in linear-time a witness that the queries deemed to be derivable (resp. not derivable) from the given hypotheses are indeed derivable (resp. not derivable).

Related work. We refer the reader to the related-work subsection 1.1 in a recent article [4] on propositional primal logic with disjunction. Here are additional remarks, mostly summarizing references above. For brevity, we say that a logic L is linear-time, polynomial time, etc. if the derivation problem for L (decide whether a given query follows from given hypotheses) is so.

This article builds on its precursors [16] and [6]. The linear-time decision algorithm for the version of the original PPIL without the `implied` construct was announced in [13]. It is claimed in [2] (A. Baskar *et al.*, in preparation) that the original PPIL is polynomial time.

The extension of the original PPIL with full-fledged disjunction (with disjunction introduction and disjunction elimination rules) was studied in [4] and proven NP-complete there. The extension of the original PPIL with primal disjunction (with disjunction introduction rules but no disjunction elimination rules) is addressed in [4, Remark 4.7]; it is shown there that the quotation-free fragment of that extension is linear-time.

2 Hilbertian calculus for PPIL

PPIL formulas are built from propositional variables and the propositional constants $\top, \bot$ by means of the binary connectives $\wedge$, $\vee$ and $\rightarrow$ and unary connectives q `said`, where q ranges over principal constants. We presume that there are countably infinite lists of propositional variables and principal constants. Thus every formula x has the form

$$q_1 \texttt{ said } q_2 \texttt{ said } \ldots q_k \texttt{ said } y$$

such that y is atomic or is a binary combination (conjunction, disjunction or implication) of two subformulas. Here y is the *body* of x. If $k > 0$, then x is a *quote formula* or simply a *quote*; otherwise it is its own body (but of course it may have quote subformulas). Every string q_1 `said` $\ldots$ q_j `said` with $0 \leq j \leq k$ is a *quotation prefix* of x. In case $j = 0$, the prefix is empty. In case $j = k$, the prefix is the maximal quotation prefix of x.

Let x, y range over formulas and `pref` range over quotation prefixes.

PPIL calculus

Axioms

($\top$) $\quad \texttt{pref}\,\top$

Rules of inference

($\wedge$i) $\quad \dfrac{\texttt{pref}\,x \qquad \texttt{pref}\,y}{\texttt{pref}\,(x \wedge y)}$

($\wedge$e) $\quad \dfrac{\texttt{pref}\,(x \wedge y)}{\texttt{pref}\,x} \qquad \dfrac{\texttt{pref}\,(x \wedge y)}{\texttt{pref}\,y}$

($\rightarrow$i) $\quad \dfrac{\texttt{pref}\,y}{\texttt{pref}\,(x \rightarrow y)}$

$$(\rightarrow\text{e})\quad \frac{\texttt{pref}\,x \qquad \texttt{pref}(x\rightarrow y)}{\texttt{pref}\,y}$$

$$(\vee\text{i})\quad \frac{\texttt{pref}\,x}{\texttt{pref}(x\vee y)} \qquad \frac{\texttt{pref}\,y}{\texttt{pref}(x\vee y)}$$

As usual, a *derivation* D of a formula x from a set Γ of hypotheses is a finite tree (or, more generally, a finite dag—directed acyclic graph—with a single source node, the root) where each node u is labelled with a formula $D(u)$. The root is labelled with x. The leaves are labelled with axioms or hypotheses. If $v_1, v_2, \ldots, v_n$ are the children of node u, then

$$\frac{D(v_1) \qquad \ldots \qquad D(v_n)}{D(u)}$$

is an instance of an inference rule. Of course in our case, n is 1 or 2. The *length* of the proof is the number of its nodes.

A set Γ of formulas *entails* a formula x if x is derivable from Γ, i.e., if there is a derivation of x from Γ. The following two definitions will be used throughout the article.

DEFINITION 2.1 (Local formulas)
We define inductively the set of formulas *local* to a given formula x:

- Formula x is local to x.
- For any binary connective $*$, if $\texttt{pref}(y*z)$ is local to x, then $\texttt{pref}\,y$ and $\texttt{pref}\,z$ are local to x.

A formula is *local* to a set Γ of formulas if it is local to some formula in Γ.

DEFINITION 2.2 (Local prefixes)
A prefix $\texttt{pref}$ is *local* to a formula x if some formula $\texttt{pref}\,z$ is local to x. A prefix is local to a set of formulas Δ if it is local to some formula $x\in\Delta$. (Note that, if a prefix π is an initial segment of a prefix local to Δ, then π is also local to Δ.)

For example, let Δ be

$$\{p\ \texttt{said}(q\ \texttt{said}\,y\wedge r\ \texttt{said}\,s\ \texttt{said}\,x),\ p\ \texttt{said}(x\rightarrow((q\ \texttt{said}\,x)\rightarrow x))\}$$

where x, y are propositional variables. Then Δ-local formulas are the two formulas in Δ plus

$$p\ \texttt{said}\,q\ \texttt{said}\,y,\ p\ \texttt{said}\,r\ \texttt{said}\,s\ \texttt{said}\,x,$$
$$p\ \texttt{said}\,x,\ p\ \texttt{said}((q\ \texttt{said}\,x)\rightarrow x),\ p\ \texttt{said}\,q\ \texttt{said}\,x.$$

The Δ-local prefixes are

$$p\ \texttt{said},\ p\ \texttt{said}\,q\ \texttt{said},\ p\ \texttt{said}\,r\ \texttt{said},\ p\ \texttt{said}\,r\ \texttt{said}\,s\ \texttt{said}.$$

3 Soundness and completeness

DEFINITION 3.1
A *Kripke model* is any structure M whose vocabulary comprises (i) binary relations S_q where q ranges over the principal constants and (ii) unary relations V_x where x ranges over the formulas. The elements of (the universe of) M are called *worlds*.

DEFINITION 3.2
Given a Kripke model M, we define when a world w *satisfies* a formula x, symbolically $w \vDash x$. We do that by induction on x. Every world satisfies $\top$. Further:

1. w satisfies a propositional variable x if $w \in V_x$.
2. $w \vDash x_1 \wedge x_2$ if $w \vDash x_1$ and $w \vDash x_2$.
3. $w \vDash x_1 \vee x_2$ if $w \vDash x_1$ or $w \vDash x_2$ or $w \in V_{x_1 \vee x_2}$. In other words:
   (a) If w satisfies some x_i, then it satisfies $x_1 \vee x_2$.
   (b) Otherwise w satisfies $x_1 \vee x_2$ if and only if it belongs to $V_{x_1 \vee x_2}$.
4. $w \vDash x_1 \to x_2$ if $w \vDash x_2$ or ($w \nvDash x_1$ and $w \in V_{x_1 \to x_2}$). In other words:
   (a) If w satisfies x_2, then it satisfies $x_1 \to x_2$.
   (b) If w satisfies x_1 but not x_2, then it does not satisfy $x_1 \to x_2$.
   (c) If w satisfies neither x_1 nor x_2, then it satisfies $x_1 \to x_2$ if and only if it belongs to $V_{x_1 \to x_2}$.
5. $w \vDash x = q\ \texttt{said}\ x_1$ if $w \vDash x_1$ for all w' with $wS_q w'$.

A world w *satisfies* a set Γ of formulas if it satisfies every formula in Γ.

Although Kripke models have unary relations V_x for all formulas x, these relations are only relevant for variables, implications and disjunctions.

THEOREM 3.3 (Soundness and completeness)
A set Γ of formulas entails a formula y if and only if, for every Kripke model, y holds in every world where Γ holds.

PROOF. ($\Rightarrow$). To establish the soundness, we suppose that a world w of a given Kripke model satisfies Γ and prove that $w \vDash y$ as well. That is done by induction on the size of the given derivation of y from Γ. The case when y is a hypothesis is trivial. If y is an axiom $\texttt{pref}\,\top$, induct on the length of $\texttt{pref}$.

Now, suppose that y is obtained by means of an inference rule R. Several cases arise depending on what R is. All these cases are straightforward. We consider here only the case where R is the rule

$$(\to\text{e})\ \frac{\texttt{pref}\,x \qquad \texttt{pref}(x \to y)}{\texttt{pref}\,y}$$

By the induction hypothesis, $w \vDash \texttt{pref}x$ and $w \vDash \texttt{pref}(x \to y)$. We prove $w \vDash \texttt{pref}\,y$ by an auxiliary induction on the length of $\texttt{pref}$. For the empty quotation prefix, by the definition of primal implication, we have either $w \vDash y$ or else ($w \nvDash x$ and $w \in V_{x \to y}$). As $w \vDash x$, we conclude that $w \vDash y$. Now suppose $\texttt{pref} = q\ \texttt{said}\ \texttt{pref}'$ and let w' range over the worlds such that $wS_q w'$. As $w \vDash \texttt{pref}x$ and $w \vDash \texttt{pref}(x \to y)$, we have $w' \vDash \texttt{pref}'x$ and $w' \vDash \texttt{pref}'(x \to y)$. By the induction hypothesis, $w' \vDash \texttt{pref}'y$. Hence, $w \vDash q\ \texttt{said}\ \texttt{pref}'y$.

($\Leftarrow$). To establish completeness, we assume that Γ does not entail y and construct a Kripke model M with a world where Γ holds but y fails. Call a quotation prefix *local* if it is local to $\Gamma \cup \{y\}$. Let $\texttt{pref}$ range over the local prefixes. Now we are ready to define M.

- The worlds are the local prefixes.
- $V_x = \{\texttt{pref} : \Gamma \vdash \texttt{pref}x\}$.
- $S_q = \{(\texttt{pref}, \texttt{pref}') : \texttt{pref}' = \texttt{pref}\ q\ \texttt{said}\}$.

LEMMA 3.4
$\texttt{pref} \vDash x$ iff $\Gamma \vdash \texttt{pref}x$, for any local prefix $\texttt{pref}$.

PROOF OF LEMMA. Induction on x.

- If x is a variable, the claim follows from the definition of M.
- Suppose that $x = x_1 \wedge x_2$. We have

$$\begin{aligned} \mathtt{pref} \vDash x_1 \wedge x_2 &\Leftrightarrow \mathtt{pref} \vDash x_1 \text{ and } \mathtt{pref} \vDash x_2 \\ &\Leftrightarrow \Gamma \vdash \mathtt{pref}\ x_1 \text{ and } \Gamma \vdash \mathtt{pref}\ x_2 && \text{by induction hypothesis} \\ &\Leftrightarrow \Gamma \vdash \mathtt{pref}(x_1 \wedge x_2) && \text{by rules } (\wedge\text{i}) \text{ and } (\wedge\text{e}) \end{aligned}$$

- Let $x = x_1 \vee x_2$. First suppose that $\mathtt{pref} \vDash x_i$ for some i. By the definition of disjunction satisfaction, $\mathtt{pref} \vDash x$. By the induction hypothesis, $\Gamma \vdash \mathtt{pref} x_i$ so that, by rule $(\vee\text{i})$, $\Gamma \vdash \mathtt{pref} x$.
  Next suppose $\mathtt{pref} \nvDash x_i$ for both $i = 1$ and $i = 2$. By the induction hypothesis, $\Gamma \nvdash x_i$ for both $i = 1$ and $i = 2$. We have

$$\mathtt{pref} \vDash x \Leftrightarrow \mathtt{pref} \in V_x \Leftrightarrow \Gamma \vdash \mathtt{pref} x.$$

  The first equivalence is based on the definition of disjunction in Kripke models, and the second is based on the definition of model M.
- Let $x = x_1 \to x_2$. Consider the three cases of item 4 in definition 3.2 and invoke the induction hypothesis. In case 1, we have $\mathtt{pref} \vDash x$ and $\Gamma \vdash \mathtt{pref} x$. In case 2, we have $\mathtt{pref} \nvDash x$ and $\Gamma \nvdash \mathtt{pref} x$. In case 3, we have that $\mathtt{pref} \vDash x$ if and only if $\mathtt{pref}$ belongs to V_x if and only if (by the definition of M) $\Gamma \vdash \mathtt{pref} x$.
- Let $x = q\ \mathtt{said}\ x'$. We have

$$\begin{aligned} &\mathtt{pref} \vDash q\ \mathtt{said}\ x' \\ \Leftrightarrow\ &\mathtt{pref}' \vDash x' && \text{for all } \mathtt{pref}' \text{ such that } \mathtt{pref}\ S_q\ \mathtt{pref}' \\ \Leftrightarrow\ &\mathtt{pref}\ q\ \mathtt{said} \vDash x' && \text{because } \mathtt{pref}\ S_q\ \mathtt{pref}' \text{ iff } \mathtt{pref}' = \mathtt{pref}\ q\ \mathtt{said} \\ \Leftrightarrow\ &\Gamma \vdash \mathtt{pref}\ q\ \mathtt{said}\ x' && \text{by the induction hypothesis.} \end{aligned}$$

■

To finish the proof of the theorem, let ϵ be the empty prefix. As $\Gamma \nvdash y$, Γ holds in ϵ but y does not. ■

One could expect that $\bot$ fails in every world of any Kripke model, but this is not necessarily so. If it were so then $\bot$ would entail every formula, but our calculus does not have any axioms or rules specifically for $\bot$. It is possible of course to extend the calculus with a rule that $\bot$ entails every formula and then require that $\bot$ fails in every world of any Kripke model. Theorem 3.3 would remain valid.

By induction on a formula x, we define its *width* $|x|$. If x is a propositional variable or constant, then $|x| = 1$; if $x = x_1 * x_2$, where $*$ is any binary connective, then $|x| = |x_1| + |x_2| + 1$, and if $x = q\ \mathtt{said}\ x_1$, then $|x| = |x_1| + 2$. For a set Δ of formulas, the *width* $|\Delta| = \sum\{|x| : x \in \Delta\}$.

THEOREM 3.5 (Small model)
If Γ does not entail y, then there is a counterexample model of size $< 1 + |\Gamma \cup \{y\}|/2$.

PROOF. We start with an auxiliary lemma.

LEMMA 3.6
The number of non-empty prefixes local to a formula x is less than $|x|/2$.

PROOF. Let $LP(x)$ be the number of non-empty prefixes local to x. This lemma is easily proved by induction on the number of connectives of x. For the case $x = q\ \texttt{said}\ x'$, note that the non-empty local prefixes of x are $q\ \texttt{said}$ plus all the prefixes of the form $q\ \texttt{said}\ \texttt{pref}$, where $\texttt{pref}$ is non-empty and local to x'. Hence,

$$\begin{aligned} LP(q\ \texttt{said}\ x') = 1 + LP(x') < 1 + |x'|/2 &= (2 + |x'|)/2 \\ &= |q\ \texttt{said}\ x'|/2. \end{aligned}$$

■

Now consider the model M built in the proof of Theorem 3.3. Recall that the worlds of the underlying Kripke structure of M are all prefixes local to $\Gamma \cup \{y\}$. The previous lemma implies that the number of non-empty prefixes local to a set of formulas Δ is less than $|\Delta|/2$; so the number of prefixes local to Δ is less than $1 + |\Delta|/2$. Hence, the size of M is less than $1 + |\Gamma \cup \{y\}|/2$. ■

REMARK 3.7 (Possible worlds)
The original definition of Kripke models for primal logic in [16] contained a partial order on the worlds. We simplified the definition because the partial order is not needed for soundness and completeness theorem. But, in the intended applications of primal logic in policy and trust management, a partial order on the worlds makes good sense. It reflects possible developments. In a world w_1, Bob is proposing Alice to be a Facebook friend of his but she isn't his friend in w_1. However, she is a Facebook friend of his in some world $w_2 > w_1$. In the presence of partial order $\leq$ on the worlds, the following constraints are necessary to maintain soundness (in particular to ensure that satisfaction of formulas is preserved upward in the partial order):

- If $u \leq v$ and $vS_q w$, then $uS_q w$.
- If $u \leq v$ and $u \in V_x$, then $v \in V_x$.

4 Local derivations

LEMMA 4.1
Let D be a shortest derivation of a formula from a set of formulas Γ. Whenever an elimination rule is used in D, its premises are local to Γ. More explicitly:

1. If D has an instance

$$\dfrac{\dfrac{\vdots}{\texttt{pref}(x_1 \wedge x_2)}}{\texttt{pref}\ x_i}$$

of rule ($\wedge$e), then the premise $\texttt{pref}(x_1 \wedge x_2)$ is local to Γ. Consequently, $\texttt{pref}\ x_1$ and $\texttt{pref}\ x_2$ are local to Γ.

2. If D has an instance

$$\dfrac{\dfrac{\vdots}{\mathtt{pref}\,x} \qquad \dfrac{\vdots}{\mathtt{pref}\,(x \to y)}}{\mathtt{pref}\,y}$$

of rule ($\to$e), then the premises $\mathtt{pref}\,x$ and $\mathtt{pref}\,(x \to y)$ are local to Γ. Consequently, $\mathtt{pref}\,y$ is local to Γ.

PROOF. We prove claims 1 and 2 simultaneously, by induction on the derivation of the premise z where z is the only premise $\mathtt{pref}\,(x_1 \wedge x_2)$ in case 1 and z is the major premise $\mathtt{pref}\,(x \to y)$ in case 2. Note that, in case 2, the minor premise is local to the major premies, so it will also be local to Γ.

Base case. In either case, due to its form, the premise z cannot be an axiom, so it is a hypothesis.

Induction step. The premise z is the conclusion of an instance of some inference rule R. We consider our two cases separately.

1. $z = \mathtt{pref}(x_1 \wedge x_2)$, so R cannot be ($\to$i) or ($\vee$i). It cannot be ($\wedge$i) either; otherwise D can be shortened as follows:

$$(\wedge\text{i})\ \dfrac{\dfrac{\dfrac{\vdots}{\mathtt{pref}\,x_1} \qquad \dfrac{\vdots}{\mathtt{pref}\,x_2}}{(\wedge\text{e})\ \dfrac{\mathtt{pref}\,(x_1 \wedge x_2)}{\mathtt{pref}\,x_i}}}{\vdots} \qquad \Rightarrow \qquad \dfrac{\dfrac{\vdots}{\mathtt{pref}\,x_i}}{\vdots}$$

So, R must be either ($\wedge$e) or ($\to$e). In either case, use the induction hypothesis. It follows that $\mathtt{pref}\,(x_1 \wedge x_2)$ is local to Γ.

2. $z = \mathtt{pref}\,(x \to y)$, so R cannot be ($\wedge$i) or ($\vee$i). It cannot be ($\to$i) either; otherwise D could be shortened. Hence, R must be ($\wedge$e) or ($\to$e). In either case, use the induction hypothesis. It follows that $\mathtt{pref}\,(x \to y)$ is local to Γ.

■

DEFINITION 4.2 (Local derivations)
A derivation of a formula y from hypotheses Γ is *local* if all node formulas of the derivation (including the axioms if any) are local to $\Gamma \cup \{y\}$.

THEOREM 4.3 (Local derivations)
Any shortest derivation of y from Γ is local.

PROOF. Clearly y is local to $\Gamma \cup \{y\}$. Now, suppose that we have proved that a formula x' in the shortest derivation is local to $\Gamma \cup \{y\}$. Formula x' is obtained by means of some derivation rule R. Let x be a premise in that application of R. It suffices to prove that x is local. If R is an elimination rule, use the previous lemma. The cases when R is an introduction rule are all obvious. ■

Axioms may be used in a shortest derivation of y from Γ but they are local. For example, a shortest derivation of y from $\{\top \to y\}$ uses axiom $\top$ local to the hypothesis. A shortest derivation of $x \wedge \top$ from x uses axiom $\top$ local to the conclusion.

THEOREM 4.4 (Interpolation)
If Γ entails y, then there is a set Δ of formulas such that

1. every Δ formula is simultaneously local to Γ and local to y,
2. there is a derivation of every Δ formula from Γ using only formulas local to Γ and
3. there is a derivation of y from Δ using only introduction rules and axioms local to y, and thus using only formulas local to y.

PROOF. Suppose $\Gamma \vdash y$. Note that claim 1 follows from the other two claims. Let D be a shortest derivation of y from Γ. We say that a node u of D is an *elimination node* if the rule used to obtain $D(u)$ is an elimination rule.

Let U be the set of nodes u in D such that no node between u and the root of the derivation tree (or dag) is an elimination node, but u itself is either an elimination node or else a leaf such that $D(u)$ is local to Γ. The desired $\Delta = \{D(u) \colon u \in U\}$. Claim 3 follows from the definition of Δ.

Consider any $D(u)$ in Δ. If u is a leaf, then $D(u)$ is local to Γ by the definition of U, and if u is an elimination node, then $D(u)$ is local to Γ by Lemma 4.1. Consider a part D' of D rooted at u. As D is a shortest derivation of y, D' is a shortest derivation of $D(u)$. Claim 2 follows from Theorem 4.3. ■

5 Decision algorithm

THEOREM 5.1 (Decision algorithm)
There is a linear-time algorithm that, given two finite sequences of formulas, H (hypotheses) and Q (queries), decides which formulas in Q are derivable from H.

The rest of this oversized section is devoted to proving Theorem 5.1. After some preliminaries in § 5.1, we construct the desired decision algorithm and explain how it works.

5.1 Preliminaries

High-level idea. Call a formula or quotation prefix *local* if it is local to the union $H \cup Q$ of the given H and Q. Call a formula y *locally derivable* if it is local and derivable from H. By Theorem 4.3, if y is locally derivable, then there is a derivation of y from H that consists of formulas local to $H \cup \{y\}$.

The high-level idea of the algorithm is to derive all locally derivable formulas and output the locally derivable queries. For the sake of clarity, we forgo some possible optimization to simplify the exposition.

EXAMPLE 5.2
Let

$$H = \{a \wedge b, c, e, (a \wedge c) \to (d \to e)\},$$
$$Q = \{(a \wedge (d \to e)) \to d, b \to (d \to e)\}.$$

All hypotheses are trivially locally derivable. From $a \wedge b$ we can derive a and b by rule ($\wedge$e). From a and c we can derive $a \wedge c$ by rule ($\wedge$i). From $(a \wedge c)$ and $(a \wedge c) \to (d \to e)$ we can derive $d \to e$ by

($\rightarrow$e). From this formula, we can derive $b \rightarrow (d \rightarrow e)$ by ($\rightarrow$i). We can also derive $(a \wedge (d \rightarrow e))$ and $(b \wedge (d \rightarrow e))$, but the latter is irrelevant because we are only interested in local formulas. We cannot derive any more local formulas, so the locally derivable formulas are:

$$a \wedge b, c, e, (a \wedge c) \rightarrow (d \rightarrow e), a, b, a \wedge c, (d \rightarrow e), b \rightarrow (d \rightarrow e), (a \wedge (d \rightarrow e)).$$

Accordingly, $b \rightarrow (d \rightarrow e)$ is the only query derivable from H.

The stages of the decision algorithm. Our algorithm works in five stages.

1. Parse the input and bind the nodes of the resulting parse tree to appropriate input positions [§5.2].
2. Construct a convenient data structure of the local quotation prefixes [§5.3].
3. Bind local formulas to nodes of the input parse tree [§5.4].
4. Construct additional data structures needed for fast derivation of local formulas [§5.5].
5. Derive the locally derivable formulas and output the derivable queries [§5.6].

In §5.7 we prove that the algorithm is indeed linear-time, establish its correctness and remark on computing—in linear-time—not only the derivable queries but also their derivations.

Computation model. We use the standard computation model of the analysis of algorithms [9]. It is the random access machine such that (i) the registers are of size $O(\log n)$ where n is the size of the input and (ii) the basic register operations are constant-time. The main justification for that computation model is that the traditional uniprocessor computer can be viewed as a unit-cost random access machine. 'In algorithms you use the unit-cost RAM model where basic register operations over $O(\log n)$ bit registers count as a single computation step. There are some good arguments for this: As technology improves for us to handle larger input sizes, the size of the registers tend to increase as well. For example, registers have grown from 8 to 64 bits on microprocessors over the past few decades,' [12].

Syntax assumptions. We assume that the formal syntax of our formulas satisfies the following rather usual requirements.

1. Formulas are strings in a fixed finite alphabet.
2. Any occurrence of any subformula of a formula x is a contiguous segment of the string x.
3. No two subformula occurrences in a formula x start at the same position of the string x. We will use the starting position of a subformula occurrence o as a key to identify o.
4. There is a deterministic pushdown automaton with an output tape (a deterministic pushdown transducer) that, given a formula, detects the initial position Key(o) of every subformula occurrence o, computes the length of the subformula in question and associates it with Key(o).

The standard syntax of formulas with all binary operators in prefix position satisfies the requirements; no parentheses are required. The infix position for the binary operators is no problem; just put parenthesis around every non-atomic subformula including the whole formula. We have been allowing ourselves to skip the outermost parentheses because they are not needed for human comprehension. But, formally, they are required.

Notation and terminology.

- The term 'input' will be used for a current input for the desired decision algorithm. The input thus is a sequence of the given hypotheses and then the given queries. More precisely (and pedantically), it is the sequence of letters obtained by concatenating the given hypotheses, then

some separator and then the given queries. We presume that the input went through a lexical analyzer and so the names of the variables and constants are of length $O(\log n)$.
- H and Q are the sets of the hypotheses and queries, respectively. A formula or quotation prefix is local if it is local to $H \cup Q$.
- n is the length of the input.
- If J is a contiguous segment of the input, then the initial position p of J (in the form of a binary number) is its key, symbolically $\mathrm{Key}(J) = p$. Note that every formula occurrence in the input is uniquely identified by its key.

5.2 *Parsing*

Contrary to our derivation trees, which grow up in accordance with logic tradition, our parse trees grow down in accordance with computer science tradition. In particular, the root of a parse tree is at the top of the tree. Following [16], we define compact parse trees of formulas where the edge labels may carry substantial information.

DEFINITION 5.3 (Formula parse tree)
By induction on formula x, we define the parse tree $\mathrm{PT}(x)$ of x.

- If x is atomic, $\mathrm{PT}(x)$ consists of one node labelled with x.
- If x is a quote $\texttt{pref}z$ with body z, then the root r of $\mathrm{PT}(x)$ has a unique child r', the r'-rooted subtree is isomorphic to $\mathrm{PT}(z)$ and the edge (r, r') is labelled with $\texttt{pref}$.
- Let x be a binary combination $x_1 \star x_2$. Then the root r of $\mathrm{PT}(x)$ is labelled with the binary connective $*$ and has two children, a left child r_1 and right child r_2. Let T_i be the r_i-rooted subtree of $\mathrm{PT}(x)$, and let T_i' be the extension of T_i with r and the edge (r, r_i). Three cases arise.

  1. If neither x_i is a quote, then each T_i is isomorphic to $\mathrm{PT}(x_i)$, and the edges (r, r_i) are unlabelled.
  2. If x_i is a quote but x_j is not, then T_j is isomorphic to $\mathrm{PT}(x_j)$, the edge (r, r_j) is unlabelled, and T_i' is isomorphic to $\mathrm{PT}(x_i)$.
  3. If both x_i are quotes, then each T_i' is isomorphic to $\mathrm{PT}(x_i)$.

We present some examples, see Figures 1–4, to clarify this definition. In the examples, x and y are propositional variables.

If u is a parse-tree node labelled with a binary connective, then we use u_l and u_r to denote the left and the right child of u, respectively.

REMARK 5.4
The parse tree $\mathrm{PT}(x)$ of a formula x is compact in the following sense. Normally, every subformula occurrence x would be represented by a separate node u of the parse tree for x (with the u-rooted subtree isomorphic to the parse tree of the subformula). In our case, however, this only holds if the

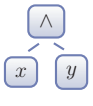

FIGURE 1. Parse tree for $x \wedge y$.

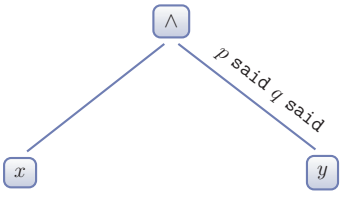

FIGURE 2. Parse tree for $x \wedge (p\,\texttt{said}\,q\,\texttt{said}\,y)$.

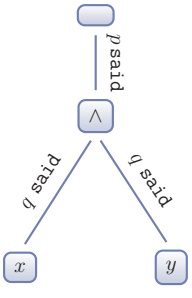

FIGURE 3. Parse tree for $p\,\texttt{said}((q\,\texttt{said}\,x) \wedge (q\,\texttt{said}\,y))$.

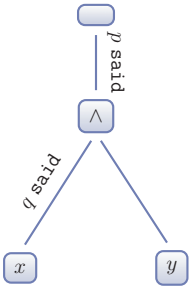

FIGURE 4. Parse tree for $p\ \texttt{said}((q\ \texttt{said}\ x) \wedge y)$.

subformula is a non-quote or x itself. Such more compact parse trees are more convenient for our purposes.

DEFINITION 5.5 (Input parse tree)
Recall that the input for the desired decision algorithm is a sequence of hypothesis followed by a sequence of queries. The top node of *the input parse tree* is labelled `input`. It has two children labelled `hypothesis` and `query`. The parse trees of the hypotheses hang under the `hypothesis` node in the order they occur in H. If the root node of a hypothesis x is unlabelled it is merged with the `hypothesis` node; otherwise the edge from the `hypothesis` node to the root of x is unlabelled. In a similar way, the parse trees of the queries hang under the `query` node.

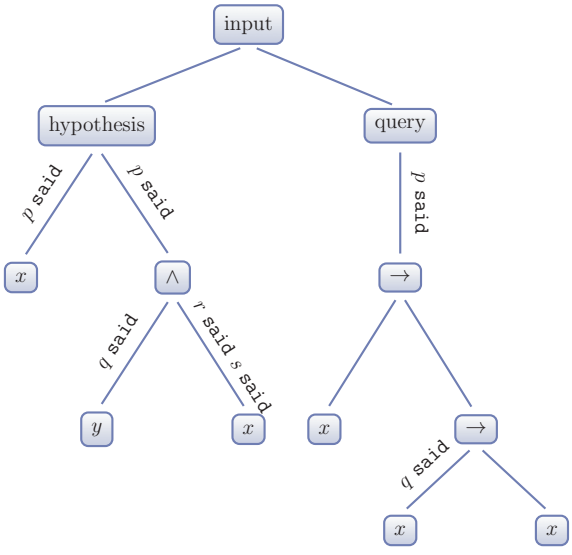

FIGURE 5. Parse tree for an instance of the multi-derivability problem.

For example, Figure 5 shows the parse tree for input

$$H = \{p \texttt{ said } x, p \texttt{ said } (q \texttt{ said } y \wedge r \texttt{ said } s \texttt{ said } x)\},$$
$$Q = \{p \texttt{ said } (x \to (q \texttt{ said } x \to x))\}.$$

DEFINITION 5.6 (Regular nodes and their body formulas)
A node u of the input parse tree is *regular* if u is labelled with an atomic formula or a binary connective; otherwise it is irregular. The *body formula* BF(u) of a regular node u is the formula x such that the u-rooted subtree is isomorphic to the parse tree of BF(u).

Thus only the three top nodes of the input parse tree, labelled `input`, `hypothesis` and `query`, are irregular.

COROLLARY 5.7
- A subformula x of the input is the body formula of some regular node if and only if x is not a quote subformula.
- If x is a non-quote subformula with n occurrences in the input, then there are exactly n nodes with x as their body formula.

In accordance with Remark 5.4, every occurrence o of a non-quote formula x in the input gives rise to a separate node u, with BF(u) $= x$, of the input parse tree. And every regular node u is obtained this way. We say that u represents o and that the key of u is that of o. Here is a more careful version of that definition.

DEFINITION 5.8
Let x be a non-quote subformula of $H \cup Q$; let $o_1, \ldots, o_m$ be the occurrences of x in the input, listed left to right; and let $u_1, \ldots, u_m$ be the m nodes of the input parse tree, in the depth-first order, with x as their body formula. Then each u_i *represents* the occurrence o_i of x, and the *key* of u_i is that of o_i.

COROLLARY 5.9
There is a linear-time algorithm—*Algorithm 1*—that, given an input in the form of a list of hypotheses and queries, builds the following:

- A parse tree for the input where every node u is decorated with the following additional fields:
  - Key(u), a number that uniquely identifies node u, namely, the initial position of the particular occurrence of formula BF(u) in the input represented by u.
  - $H(u)$, a pointer of type node, to be used in stage 3 of the desired decision algorithm and set to `nil` for the time being.
  - Length(u), the length of BF(u).
  - Vertex(u), to be used in stage 2 of the decision algorithm and set to `nil` for the time being.
  - $T(u)$, a record, to be used in stages 4 and 5 of the decision algorithm and set to `nil` for the time being.
- An array Node indexed by input positions. If p is the initial position of a subformula occurrence, then Node[p] is the node u with Key(u)$=p$. Otherwise Node[p]$=$`nil`.

PROOF. The desired algorithm uses standard parsing techniques to compute the required structure and fields [1]. ■

STAGE 1 Algorithm 1 is stage 1 of the desired decision algorithm.

5.3 Constructing a trie of local prefixes

A *trie* (also known as a *keyword tree* in the pattern-matching community) is a well-known data structure for quick storage and retrieval of strings [17, 18]. A trie for a list $\langle s_1, s_2, \ldots, s_k \rangle$ of strings in some finite alphabet Σ is a tree with the following properties. It will be convenient for us to use the term vertex for trie's nodes.

- Every edge is labelled with a letter in the alphabet Σ. As a result, every vertex v is associated with the string on the route from the root vertex to v.
- Distinct vertices v_1, v_2 are associated with distinct strings.
- For every string s_i there is a vertex v_i associated with s_i; the vertex v_i is labelled with the string s_i. There are no other labelled vertices.
- Every leaf vertex is labelled. Some internal vertices may be labelled as well.

EXAMPLE 5.10
We sketch a construction of a trie (see Figure 6 for a list

$$\langle \text{ BE, SO, BAT, BEE, SIN, BELL, BEST, SING, SINK } \rangle$$

of strings in capital Latin letters. Start by creating a root vertex. Then read the given strings one after another, letter by letter, and put the current string on the trie as follows:

The case of BE. As the root does not have an outgoing B-labelled edge, create such an edge as well as a subsequent E-labelled edge. Label with BE the vertex associated with that string. The case of SO is similar.

The case of BAT. As the root has already a B-labelled edge, go down that edge. This brings you to a vertex without an outgoing A-labelled edge. Create such an edge as well as a subsequent T-labelled edge. Label with BAT the vertex associated with that string. And so on.

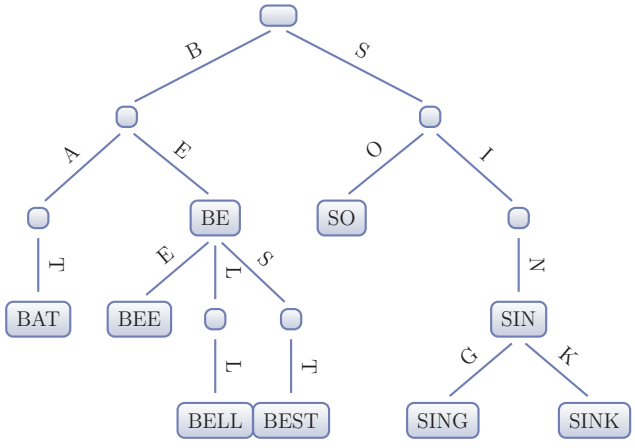

FIGURE 6. A trie for ⟨be, so, bat, bee, sin, bell, best, sing, sink⟩.

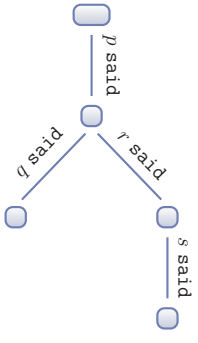

FIGURE 7. Trie of prefixes for the parse tree of Figure 5.

It is easy to see that tries over a fixed finite alphabet are built in linear-time.

For the purpose of the next definition, but also below, unlabelled edges of the input parse tree are viewed being labelled with the empty prefix ϵ.

DEFINITION 5.11 (Node prefixes)
For every node u of the input parse tree, $\mathrm{Pref}(u)$ is the concatenation of the edge labels from the root to u. In other words, if u is the root, then $\mathrm{Pref}(u)=\epsilon$; otherwise $\mathrm{Pref}(u)=\mathrm{Pref}(v)\mathrm{Label}(v,u)$, where v is the parent of u.

THEOREM 5.12
There is a linear-time algorithm—*Algorithm 2*—that, given the output of *Algorithm 1*, builds an auxiliary datastructure of the local prefixes in such a way that questions whether $\mathrm{Pref}(u)=\mathrm{Pref}(w)$ is decidable in constant-time.

PROOF. The auxiliary datastructure is the trie of node prefixes, the *prefix trie*. Figure 7 shows a simplified prefix trie for the parse tree in Figure 5 where each unary quotation prefix is treated as one letter. In the honest trie, principal constants should be spelled in full, so that each edge of Figure 7 may have internal nodes, and the two edges of Figure 7 that come from the same vertex may have a common initial segment. The occurrences of `said` may be omitted. However, the blank separating a principal constant from `said` should stay or be replaced by another character that does not occur in principal constants. Otherwise, different prefixes may give rise to the same string, e.g. `andrea said sandy said` and `andreas said andy said` give rise to the same string `andreasandy`.

The desired algorithm traverses the input parse tree in the depth-first way. The vertices of the desired prefix trie are records. In particular, there are these two numerical fields: the identifier and the SA-Position field. By default, when a vertex v is created, SA-Position(v) is set to -1; the SA-Position field will be used on a later stage. Also, for every regular node u, the desired algorithm sets Vertex(u) to the unique trie vertex representing Pref(u), so that Pref(u) = Pref(Vertex(u)).

The algorithm starts with creating the root vertex of the desired trie and setting the Vertex field of the root node of the input parse tree to the root vertex. The rest of the work is done whenever the algorithm goes down an edge (u,u') of the input parse tree. Let $v=$Vertex(u) and $\pi=$Label(u,u'). If π is the empty prefix, then Vertex(u') is set to v. Otherwise let π_0 be the maximal prefix of π such that the current trie has a branch b_0 with a final vertex v_0 such that b_0 starts from v and the letters on b_0 spell π_0. If $\pi_0=\pi$, the algorithm sets Vertex(u') $=v_0$. Otherwise π has the form $\pi_0\pi_1$, and the algorithm constructs a branch b_1 from v_0 down to a leaf v' such that the letters on b_1 spell π_1. Then the algorithm sets Vertex(u') to v'.

Now, for any two nodes u and w of the input parse tree, it can be decided in constant-time whether Pref(u) = Pref(w); just check whether Vertex(u) = Vertex(w). ■

STAGE 2 Algorithm 2 is stage 2 of the desired decision algorithm.

5.4 A node representation of local formulas

Recall that for every contiguous segment J of the input, Key(J) is the start position of J. Let $J_0,\ldots,J_{n-1}$ be the nonempty suffixes of the input in the lexicographic order. (If J is a prefix of another suffix J', then J lexicographically precedes J'.) The *suffix array* for the input is the array

$$\big[\langle \text{Key}(J_0), \text{LCP}(0)\rangle \ldots \langle \text{Key}(J_{n-1}), \text{LCP}(n-1)\rangle\big]$$

where LCP(i) is the length of the longest common prefix of J_i and J_{i-1}, unless $i=0$, in which case $LCP(i)=0$. The suffix array of a string is computable in linear-time [17].

EXAMPLE 5.13
Consider a formula

(	(	(	x	$\to$	y	)	$\to$	x	)	$\to$	x	)
0	1	2	3	4	5	6	7	8	9	10	11	12

and suppose that the lexicographic order of the symbols is

$$x, y, \to, (,).$$

Table 1 shows the suffix array for the formula.

TABLE 1. Suffix array for $(((x \to y) \to x) \to x)$. The extra column J_i is included for clarity; it is not a part of the suffix array

i	Key(J_i)	LCP(i)	J_i
0	3	0	$x \to y) \to x) \to x)$
1	11	1	$x)$
2	8	2	$x) \to x)$
3	5	0	$y) \to x) \to x)$
4	10	0	$\to x)$
5	7	3	$\to x) \to x)$
6	4	1	$\to y) \to x) \to x)$
7	2	0	$(x \to y) \to x) \to x)$
8	1	1	$((x \to y) \to x) \to x)$
9	0	2	$(((x \to y) \to x) \to x)$
10	12	0	$)$
11	9	1	$) \to x)$
12	6	4	$) \to x) \to x)$

DEFINITION 5.14 (Complete node formulas CF(u))
For every regular node u, CF(u) = Pref(u)BF(u). In other words, if u is a leaf, then CF(u) = Pref(u)Label(u), and otherwise

$$\mathrm{CF}(u) = \mathrm{Pref}(u)\big[\big(\mathrm{Label}(u, u_l)(\mathrm{BF}(u_l)\big) * \big(\mathrm{Label}(u, u_r)\mathrm{BF}(u_r)\big)\big].$$

DEFINITION 5.15 (Homonymy)
Let u, w range over the regular nodes of the input parse tree. If CF(u) = CF(w), then u, w are *homonyms*.

COROLLARY 5.16
Every CF(u) is a local formula, and every local formula is the complete node formula CF(u) for some node u.

PROOF. To prove the first claim, induct on the number of edges from the root to u. As for the second claim, let x be an arbitrary member of $H \cup Q$. It suffices to prove that every formula local to x is the complete formula CF(u) for some node u. Induct on the formulas local to x. ■

THEOREM 5.17
There is a linear-time algorithm—*Algorithm 3*—that, given the input sequence of hypotheses and queries and given the outputs of *Algorithm 1* and *Algorithm 2*,

- computes a particular node, the *homonymy leader*, in every homonymy class of regular nodes, and
- sets the pointer $H(u)$ to the homonymy leader of u, for every regular node u.

PROOF. Let Suff(u) be the suffix J of the input with Key(J) = Key(u). We will choose a regular node u as a *homonymy leader* if Suff(u) lexicographically precedes Suff(w) for every other homonym w of u.

Observe that, if x is a non-quote subformula of the input, and if $u_1, \ldots, u_m$ are the regular nodes with BF(u_i) = x ordered according to the lexicographic ordering of the associated suffixes Suff(u_l), then

1. (Suff(u_1), …, Suff(u_m)) is a contiguous segment in the lexicographic ordering of the input suffixes.

2. If p_l is the position of Suff(u_l) in the first column of the suffix array, then LCP(p_1) < Length(x), LCP(p_i) ≥ Length(x) for $1 < i \leq m$ and LCP($1+p_m$) < Length(x).
3. u_l is a homonymy leader if and only if Pref(u_l) differs from every Pref(u_i) with $i < l$.

The desired algorithm works as follows. First it constructs the suffix array for the input. Then it traverses the suffix array. The work is done when it traverses stretches Suff(u_1), …, Suff(u_m) of the kind described in the observations above. Let's consider the traversal of one of such stretches. We use the notation of the observations; in addition let v_i = Vertex(u_i). Note that formula x = BF(Node[p_1]); let ℓ = Length(x). The suffix-array position p_1 is recognized because Node[p_1] ≠ `nil` and LCP(p_1) < ℓ. The position $1+p_m$ is recognized because it is the first position p after p_1 such that LCP(p) < ℓ.

At any position p_l, the algorithm checks whether SA-Position(v_l) < p_1. If yes, then it sets $H(u_l) = u_l$, thus making u_l a homonymy leader, and it sets SA-Position(v_l) = p_l. Otherwise the algorithm sets $H(u_l)$ = Node[SA-Position(v_l)].

We illustrate the algorithm on an example where $m = 4$, $v_1 = v_3$, $v_2 = v_4$ but $v_1 \neq v_3$. All four nodes u_i have the same body formula x, and there are no other nodes with that body formula. However, the complete node formulas CF(u_i) are not all the same. We have CF(u_1) = CF(u_3) and CF(u_2) = CF(u_4) but CF(u_1) ≠ CF(u_2). Thus the four nodes u_i give rise to two distinct homonymy classes $\{u_1, u_3\}$ and $\{u_2, u_4\}$. The algorithm starts the traversal of the stretch Suff(u_1), Suff(u_2), Suff(u_3), Suff(u_4) by checking whether SA-Position(v_1) < p_1. The answer is positive. Indeed, if the algorithm has never touched SA-Position(v_1), then SA-Position(v_1) = $-1 < p_1$. Otherwise the algorithm touched SA-Position(v_1) for the last time as it traversed some earlier stretch Suff(u'_1), Suff(u'_2), … of the suffix array, and so SA-Position(v_1) is some position p'_i = Key(u'_i) with Suff(u'_i) in that earlier stretch. That earlier suffix Suff(u'_i) starts at position p'_i, and so $p'_i < p_1$. Accordingly, the algorithm sets $H(u_1) = u_1$, making u_1 the leader in the homonymy class $\{u_1, u_3\}$. Similarly, the algorithm will discover that SA-Position(v_2) < p_1 and will set $H(u_2) = u_2$, making u_2 the leader in the homonymy class $\{u_2, u_4\}$. The algorithm will discover that SA-Position(v_3) = p_1 and will set $H(u_3)$ = Node[SA-Position(v_3)] = Node[p_1] = u_1. The algorithm will discover that SA-Position(v_4) = $p_2 > p_1$ and will set $H(u_4)$ = Node[SA-Position(v_4)] = Node[p_2] = u_2. ■

STAGE 3 Algorithm 3 is the stage 3 of the decision algorithm.

DEFINITION 5.18
A regular node u is the *node representation* of the local formula CF(u) if u is a homonymy leader.

5.5 *Preprocessing*

Recall that every node u of the input parse tree is decorated with a record $T(u)$. View the record $T(u)$ as an entry of a table T. Initially every entry is `nil`. At its fourth stage, the decision algorithm fills in some content into the table T, to be used on the final fifth stage. We ask the reader to bear with us. The intended meaning of the table will become clear in the next subsection.

Description of table T. A record $T(u)$ has the following fields where $*$ ranges over the binary connectives. Let CF(u) = `pref`x where x may be a quote. We are interested only in records $T(u)$ where u is a homonymy leader.

- ($*$, left): A list of all homonymy leaders w such that CF(w) = `pref`($x*y$) for some y.
- ($*$, right): A list of all homonymy leaders w such that CF(w) = `pref`($y*x$) for some y.

- A numeric field Status(u) takes values 1, 2 and 3. Contrary to the previous fields that stay unchanged during Stage 5, the status of a node may change at that stage.

The intended meaning of the status. Intentionally the status in question is that of the formula CF(u). Besides, it is the status of $H(u)$ that matters.

- Status($H(u)$) = 1 indicates that CF(u) has not been derived yet. We say that $H(u)$ is *raw*.
- Status($H(u)$) = 2 indicates that CF(u) has been derived but not processed in the sense explained in the next section. We say that $H(u)$ is *pending*.
- Status($H(u)$) = 3 indicates that CF(u) has been derived and processed. We say that $H(u)$ is *processed*.

In Stage 4, the status of every homonymy leader u is initialized to 1, unless u represents an axiom or hypothesis, in which case the status of u is initialized to 2.

STAGE 4 The algorithm traverses the input parse tree in the depth-first order and constructs the table T. On the same occasion, it constructs a queue, called the *pending queue*, and initializes it with the axioms and hypotheses.

- If the label of u is a binary connective $*$ and if $H(u) = u$, then append $H(u)$ to the ($*$, left) field of $T(H(u_l))$ and to the ($*$, right) field of $T(H(u_r))$.
- If Label(u) = $\top$ or u is a child of the `hypothesis` node, and if Status($H(u)$) = 1, then append $H(u)$ to the pending queue and set Status(u) = 2.

The one-node computation is constant-time. Therefore, Stage 4 takes linear-time.

5.6 Deriving local formulas

Finally we reach the stage of the decision algorithm where it derives all the locally derivable formulas. The idea is simple. Pick the first pending node u and apply all derivation rules to CF(u), which may cause some raw formulas to become pending, and then remove u from the pending queue and set Status(u) = 3. Repeat this procedure until there are no pending formulas. The following invariant is maintained: if a homonymy leader u is pending or processed, then CF(u) is derivable.

But what does it mean to apply a derivation rule to the formula CF(u)? We explain that. So let u be a homonymy leader. For brevity, to make a node w pending means to append w to the pending queue and to set Status(w) = 2.

Applying (∧e) to u.

Do this only if u is labelled with ∧. If $H(u_l)$ is raw, make it pending. Similarly, if $H(u_r)$ is raw, make it pending.

Justification If u is labelled with ∧, then CF(u) = `pref`($x \wedge y$), CF($H(u_l)$) = `pref`x and CF($H(u_r)$) = `pref`y. As u is pending, `pref`($x \wedge y$) is derivable. By rule (∧e), `pref`x and `pref`y are derivable.

Applying (∧i) to u.

1. Walk through the nodes w in the (∧,left) field of $T(u)$. If w is raw and if $H(w_r)$ is pending or processed, then make w pending.

   Justification Suppose CF(u) = `pref`x. The (∧,left) field of $T(u)$ comprises homonymy leaders w such that CF(w) = `pref`($x \wedge y$) for some y. It follows that CF($H(w_r)$) = CF(w_r) = `pref`y. As

u is pending, $\texttt{pref}x$ is derivable. If $H(w_r)$ is pending or processed, then $\texttt{pref}y$ is derivable as well, and then—by the rule($\wedge$i)—$\texttt{pref}(x \wedge y)$ is derivable.

2. Similarly, walk through the nodes w in the ($\wedge$,right) field of $T(u)$. If w is raw and if $H(w_l)$ is pending or processed, then make w pending.

Applying ($\vee$i) to u.

Walk through the ($\vee$,left) and ($\vee$,right) fields of $T(u)$ and make pending each raw node w there.
Justification. Let $\text{CF}(u) = \texttt{pref}x$. The ($\vee$,left) list comprises the homonymy leaders w with $\text{CF}(w) = \texttt{pref}(x \vee y)$ for some y. Similarly, the ($\vee$,right) list comprises the homonymy leaders w with $\text{CF}(w) = \texttt{pref}(y \vee x)$ for some y. As u is pending, $\texttt{pref}\ x$ is derivable. By rule ($\vee$i), $\texttt{pref}(x \vee y)$ and $\texttt{pref}(y \vee x)$ are derivable.

Applying ($\rightarrow$i) to u.

Walk through the ($\rightarrow$, right) field of $T(u)$ and make pending each raw node w there.
Justification. Let $\text{CF}(u) = \texttt{pref}\ x$. The ($\rightarrow$, right) list comprises homonymy leaders w such that $\text{CF}(w) = \texttt{pref}(y \rightarrow x)$ for some y. As $\texttt{pref}x$ is derivable, so is $\texttt{pref}(y \rightarrow x)$.

Applying ($\rightarrow$e) to u. $\text{CF}(u)$ can be used as (1) the left of the ($\rightarrow$e) rule or (2) the right premise of the rule. Accordingly, we have two substeps.

1. Walk through the ($\rightarrow$,left) list of $T(u)$. For each node w there, if w is pending or processed but $H(w_r)$ is raw, then make $H(w_r)$ pending.
Justification. Let $\text{CF}(u) = \texttt{pref}\ x$. The ($\rightarrow$,left) field of $T(u)$ comprises homonymy leaders w such that $\text{CF}(w) = \texttt{pref}(x \rightarrow y)$ for some y. Then $\text{CF}(w_r) = \texttt{pref}\ y$. As u is pending, $\texttt{pref}\ x$ is derivable. If w is pending or processed, then $\texttt{pref}(x \rightarrow y)$ is also derivable, and then—by the rule ($\rightarrow$e)—$\texttt{pref}\ y$ is derivable.
2. If u is labelled with $\rightarrow$ and if $H(u_l)$ is pending or processed but $H(u_r)$ is raw, then make $H(u_r)$ pending. Otherwise do nothing.
Justification. Suppose that $\text{CF}(u) = \texttt{pref}(x \rightarrow y)$, so that $\text{CF}(u_l) = \texttt{pref}\ x$ and $\text{CF}(u_r) = \texttt{pref}\ y$. As u is pending, $\texttt{pref}(x \rightarrow y)$ is derivable. If $\texttt{pref}\ x$ is also derivable, then, by the rule ($\rightarrow$ e), $\texttt{pref}\ y$ is derivable.

Now we are ready to summarize the derivation stage of the decision algorithm.

STAGE 5 The algorithm repeats the following procedure until the pending queue is empty.

Node-processing procedure. The algorithm picks the first pending node u and applies every derivation rule to $\text{CF}(u)$ in the way described above. Then it removes the node u from the pending queue and sets $\text{Status}(u) = 3$.

When the pending queue is empty, the algorithm compiles a list of the numbers of derivable queries. To this end, it walks through the children $u_1, u_2, \ldots, u_k$ of the `query` node. If $H(u_i)$ is processed, then the algorithm appends the number i to the list.

That concludes the construction of the decision algorithm.

REMARK 5.19
If necessary, we can print the derivable queries. Indeed, walk through the children $u_1, \ldots, u_m$ of the `query` node. The i^{th} query is $\text{CF}(u_i)$, which is $\texttt{pref}_i BF(u_i)$ where $\texttt{pref}_i$ is the label from the `query` node to u_i. So, if $\text{Status}(u_i) = 3$, then print $\texttt{pref}_i$ and then print the letters in input positions

Key(u_i) to Key(u_i)+Length(u). As separate queries are separate segments of the input, the printing process takes linear-time. If you worry that the i^{th} query may be identical to a later query and you don't want to print it again, mark $H(u_i)$ printed, e.g. by setting Status($H(u_i)$) to 4. Ironically, we may be unable to print the numbers of the derivable queries in linear-time. Consider e.g. the case with no hypotheses and n queries $\top,\ldots,\top$. The length of the list $1,\ldots,n$, in binary or decimal, is of the order $n\times\log n$.

5.7 Analysis

THEOREM 5.20
The decision algorithm works in linear-time.

PROOF. We have already checked that Stages 1–4 take linear-time. As compiling the list of derivable queries is linear-time, it remains to show the derivation process of Stage 5 takes linear-time. This is not completely obvious because the processing of one pending node is not necessarily constant-time.

It suffices to check that, for every derivation rule R, the total time of all applications of R is linear-time. The case of applying rule ($\wedge$e) is obvious, and so is the second case of applying rule ($\rightarrow$e). We restrict attention to rule ($\wedge$i) because the remaining cases are similar to it.

The application of rule ($\wedge$,i) to a pending node u is proportional to the number of nodes in the ($\wedge$,left) and ($\wedge$,right) lists of u. The key point is that different homonymy leaders have disjoint ($\wedge$,left) lists, and they have disjoint ($\wedge$,right) lists. So the number of nodes in all ($\wedge$,left) lists is $\leq n$, and the number of nodes in all ($\wedge$,right) lists is $\leq n$. Hence the total time to apply the ($\wedge$i) rule to pending nodes is linear. ■

THEOREM 5.21
The decision algorithm is sound (so that every query deemed derived by the algorithm from the given hypotheses is indeed derived) and complete (so that all given queries that are derivable are derived by the algorithm).

PROOF. The soundness is obvious at this point; we have already provided sufficient justifications. It remains to establish the completeness. To this end, it suffices to prove that, for every homonymy leader u, if CF(u) is derivable from the hypotheses, then u becomes pending at some point. We prove that by induction on the length of the given local derivation of CF(u). If CF(u) is an axiom or hypothesis, then u becomes pending at Stage 4. Otherwise CF(u) is the conclusion of some inference rule R. Several cases arise. Three of these cases are easy, so we consider the other two.

• R is ($\wedge$i). The last part of the derivation of CF(u) looks like this:

$$(\wedge\text{i})\ \dfrac{\dfrac{\begin{array}{c}H\\ \vdots\\ \texttt{pref}\,x\end{array}}{\qquad}\qquad\dfrac{\begin{array}{c}H\\ \vdots\\ \texttt{pref}\,y\end{array}}{\qquad}}{\texttt{pref}(x\wedge y)}$$

Thus CF(u) has the form $\texttt{pref}(x\wedge y)$, CF(u_l)=$\texttt{pref}x$ and CF(u_r)=$\texttt{pref}y$. The derivations of $\texttt{pref}x$ and $\texttt{pref}y$ from H are shorter, so by the induction hypothesis $H(u_l)$ and $H(u_r)$ become pending during the execution of the algorithm. By symmetry, we may assume without loss of generality that $H(u_r)$ is processed earlier than $H(u_l)$. When we apply ($\wedge$i) to $H(u_l)$, we walk through

the nodes in the ($\wedge$,left) list of u_l and find u there. As $H(u_r)$ had become pending earlier, we check whether u is raw and, if yes, make it pending.

- R is ($\rightarrow$e). The last part of the derivation of CF(u) looks like this:

$$(\rightarrow\text{e})\ \dfrac{\dfrac{\overset{H}{\overset{\vdots}{\texttt{pref}\,x}}}{\texttt{pref}\,y}\qquad\dfrac{\overset{H}{\overset{\vdots}{\texttt{pref}(x\rightarrow y)}}}{\texttt{pref}\,y}}{\texttt{pref}\,y}$$

By assumption, this derivation uses only local formulas, so there must be a homonymy leader w such that CF(w)=$\texttt{pref}(x\rightarrow y)$. Then CF($w_l$)=$\texttt{pref}x$ and $u=H(w_r)$. The derivations of $\texttt{pref}x$ and $\texttt{pref}(x\rightarrow y)$ from H are shorter, so by the induction hypothesis $H(w_l)$ and w become pending during the execution of the algorithm. Suppose w is processed earlier than $H(w_l)$ (the alternative, where $H(w_l)$ is processed earlier, is easier). When we apply ($\rightarrow$e) to $H(w_l)$, we walk through the nodes of the ($\rightarrow$,left) list of $H(w_l)$ and find w there. As w had been processed earlier, we check whether $H(w_r)$ is raw and, if yes, make it pending. But $H(w_r)=u$. ■

COROLLARY 5.22 (Witness extraction)
The decision algorithm can be extended to extract in linear-time a witness that the given queries claimed to be derivable are indeed derivable from the given hypotheses and that the remaining given queries are not derivable from the given hypotheses.

PROOF. The desired witness is a derivation dag D (directed acyclic graph) on the homonymy leaders that can be constructed as a byproduct of the decision algorithm. First put down the nodes representing axioms and hypotheses. As a new node u becomes pending, put it on the dag with pointers to the nodes representing the premises of the derivation rule used to make u pending. For every query y claimed to be derived, D includes a derivation of y from the hypotheses; just consider the sub-derivation of D rooted at the node representing y.

By virtue of By Theorem 4.3, the derivation D can be also seen as a 'negative witness,' albeit indirect, that any remaining query z is underivable from the hypotheses: if the node representing z is not in D, then z is underivable. Indeed, the collection of the formulas labelling D nodes is closed under local derivations and thus contains every locally derivable formula. By Theorem 4.3, it contains every local formula derivable from the hypotheses. ■

The use of D as a negative witness requires checking that the set of formula labels is closed under local derivations. This may be not as satisfying as one would desire. Still D is a negative witness. It can be used to construct an alternative negative witness in the form of a Kripke model. See also the Normal Form theorem, Theorem 5.8 in [16], in this connection.

6 PIL with variables

PIL was developed in the context of policy and trust management. Its propositional version PPIL is insufficient for the intended applications. Knowledge assertions often involve object variables, e.g. 'Every password contains digits and letters,' or 'Any employee should have a manager.'

Our presentation of PIL follows that of [6], where PIL is called PIV, an allusion to Primal Infon logic with Variables.

6.1 Syntax and semantics

We restrict attention to the universal fragment of PIL. Atomic formulas are those of multi-sorted first-order logic without equality or function symbols of positive arity. In particular, there is a sort *principal*. Other formulas are built from atomic formulas and propositional constants $\top$, $\bot$ by means of conjunction, disjunction, implication and unary connectives p `said`, where p is a variable or constant of sort *principal*.

Formulas may have nullary function symbols, also known as constants; these are object constants not to be confused with propositional constants $\top$ and $\bot$. As there are no function symbols of positive arity, a term is either a variable or a constant.

The intended meaning of a formula $\varphi(v_1, \ldots, v_k)$ is the universal closure $\forall v_1 \ldots \forall v_k \, \varphi(v_1, \ldots, v_k)$. In other words, the variables are implicitly quantified. A Hilbertian calculus for (the universal fragment of) PIL is the extension of the Hilbertian calculus for PPIL in §2, where x and y range over (quantifier-free) PIL formulas and `pref` ranges over quotation prefixes, with one additional rule of inference:

$$\text{(substitution)} \ \frac{\varphi}{\xi\varphi}$$

where ξ is any substitution of variables with terms. In the rest of this section, we work in the Hilbertian calculus for PIL, and denote PIL formulas with Greek letters.

DEFINITION 6.1
A substitution ξ is *native* to a set Δ of formulas if every term $\xi(v)$ occurs in Δ.

6.2 Derivation problem for PIL

Recall that the derivation problem for a logic L is the problem whether given hypotheses H entail a given formula γ. Call a substitution *native* if it is native to the given $H \cup \{\gamma\}$. Call a derivation *propositional* if it does not use the substitution rule.

THEOREM 6.2
[6, Theorem 18 and Corollary 22] If H entails γ, then there is a set H' of native-substitution instances of the hypotheses such that H' propositionally entails γ. Furthermore, suppose that at least one term occurs in γ or at least one constant occurs in H. Then every variable of H' also occurs in γ. In particular, if γ is ground (without variables), then so is H'.

By virtue of Theorems 4.3 and 4.4, we may impose additional restrictions on the propositional part of the deduction.

COROLLARY 6.3
Theorem 6.2 remains true if we require that the deduction of γ from H' satisfies the following requirements.

- The derivation of γ from H' is local to $H' \cup \{\gamma\}$.
- If γ is atomic, then it is local to H', and the deduction of γ from H' is local to H'. Otherwise the deduction of γ from H' splits into, first, a part consisting of formulas local to H', and, second, a part building up γ from these formulas local to H' by means of introduction rules.

A decision algorithm. For the purpose of Theorem 6.2, the variables of γ are treated as constants. The theorem gives rise to an algorithm deciding whether given hypotheses H entail a given query γ. Indeed we may assume without loss of generality that the set C of constants in $H \cup \{\gamma\}$ is not empty. The desired H' can be the set of all substitution instances $\xi(\alpha)$ where $\alpha \in H$ and $\mathrm{Range}(\xi) \subseteq C$. This gives rise to a decision algorithm for PIL.

Of course there are in general exponentially many such substitution instances. However, the following happens in many policy scenarios. Even though there may be many object variables in the policy rules, almost all variables have been replaced with constants when the need arises to check entailment. The decision algorithm in question was implemented by Artem Melentyev in 2012 and is available at [11].

Other decision algorithms. A Prolog-like decision algorithm for (the universal fragment of) PIL was written by Michał Moskal in 2010.

In §6 of [6], the decision problem for (the universal fragment of) PIL is reduced to that of Datalog. A more practical version of that reduction is constructed in [5].

Acknowledgments

We are grateful to Andreas Blass for many useful discussions; to Artem Melentyev for implementing the linear-time decision algorithm for PPIL as well as its extension to the universal fragment of PIL; to Ori Lahav and Yoni Zohar for a correction and—the last but not the least—to our very vigilant referees.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman, 1986.
[2] A. Baskar, P. Naldurg and K. R. Raghavendra. DKAL — update. In preparation.
[3] L. Beklemishev, A. Blass and Y. Gurevich. What is the logic of information? In preparation.
[4] L. Beklemishev and Y. Gurevich. Propositional primal logic with disjunction. *Journal of Logic and Computation*, **22**, 26, 2012.
[5] N. Bjørner, G. de Caso and Y. Gurevich. From primal infon logic with individual variables to datalog. In *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, E. Erdem *et al*. eds., Vol. 7265, Lecture Notes in Computer Science, pp. 72–86. Springer, 2012.
[6] A. Blass and Y. Gurevich. Hilbertian deductive systems and datalog. *Bulletin of the EATCS*, **102**, 122–150, 2010.
[7] A. Blass, Y. Gurevich, M. Moskal and I. Neeman. Evidential authorization. In *The Future of Software Engineering*, S. Nanz, ed., pp. 77–99. Springer, 2011.
[8] J. Cai and R. Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, **145**, 189–228, 1995.
[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest and Clifford Stein. *Introduction to Algorithms*. MIT Press, 1990.
[10] C. Cotrini and Y. Gurevich. Transitive primal infon logic: the propositional case. *The Review of Symbolic Logic*, **6**, 281–304, 2013.
[11] DKAL at CodePlex. http://dkal.codeplex.com/, viewed August 10, 2012.
[12] L. Fortnow. Shaving logs with unit cost. http://blog.computationalcomplexity.org/2009/05/shaving-logs-with-unit-cost.html, seen July 22, 2012.

[13] Y. Gurevich. Two notes on propositional primal logic. *Microsoft Research Technical Report MSR-TR-2011-70*, Microsoft, Redmond, WA, USA. May 2011. http://research.microsoft.com/pubs/149526/207.pdf.
[14] Y. Gurevich and I. Neeman. DKAL: distributed-knowledge authorization language. In *Proceedings of 2008 IEEE Computer Security Foundations Symposium (CSF 2008)*, pp. 149–162. IEEE Computer Society, 2008.
[15] Y. Gurevich and I. Neeman. DKAL 2 — a simplified and improved authorization language. *Microsoft Research Technical Report MSR-TR-2009-11*. Microsoft, Redmond, WA, USA. February 2009. http://research.microsoft.com/apps/pubs/default.aspx?id=79528
[16] Y. Gurevich and I. Neeman. Logic of infons: the propositional case. *ACM Transactions on Computational Logic*, 12, article 9, 2011. *Microsoft Research Technical Report MSR-TR-2011-90*. July 2011, corrects an oversight in the published version.
[17] D. Gusfield. *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. Cambridge University Press, 1997 (Reprinted 1999, with corrections).
[18] R. Sedgewick and K. Wayne. Algorithms, 4th edn. Addison-Wesley Professional, 2011.
[19] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, **9**, 67–72, 1979.

Received 19 August 2012