Plato: A Compiler for Interactive Web Forms

Timothy L. Hinrichs

University of Chicago

Abstract. Modern web forms interact with the user in real-time by detecting errors and filling-in implied values, which in terms of automated reasoning amounts to SAT solving and theorem proving. This paper presents PLATO, a compiler that automatically generates web forms that detect errors and fill-in implied values from declarative web form descriptions. Instead of writing HTML and JavaScript directly, web developers write an ontology in classical logic that describes the relationships between web form fields, and PLATO automatically generates HTML to display the form and browser scripts to implement the requisite SAT solving and theorem proving. We discuss PLATO's design and implementation and evaluate PLATO's performance both analytically and empirically.

1 Introduction

Modern web forms, implemented using a combination of HTML and browser scripts (*e.g.*, JavaScript, Flash, Silverlight), solicit information from users on the World Wide Web. While many web forms are simple to build and maintain, the trend toward interactive web forms has significantly complicated web form development. For example, web forms are now routinely used as a platform for configuration management applications, which help users explore the permissible combinations of components for complex systems, *e.g.*, for a personal computer the processor, hard drive, and memory.

The two types of web-form interactions studied in this paper both occur each time the user changes the web form data: identifying errors and computing implied values. An error arises when the user data conflicts with the intended semantics of the web form, *e.g.*, a credit card's expiration date must be in the future, but the user entered a date in the past. Errors are often highlighted for the user in red. An implied value arises when all possible error-free completions of the form assign a specific value to a specific form field. Implied values are usually filled-in for the user automatically.

Browser scripts that detect errors and compute implied values are difficult to write because, in general, error-detection amounts to SAT solving (SAT), and implied value computation amounts to theorem proving (TP), *e.g.*, [22, 33]. Of course, not all error-detection/implied-value scripts implement the full machinery of SAT/TP; rather, the scripts for each form embody the fragment of SAT/TP necessary to address the form at hand. Conceptually, error-detection and implied-value scripts specialize SAT and TP algorithms to the web form's semantics. The specialization process, however, is error-prone and the resulting scripts can be difficult to maintain.

To complicate matters further, traditional TP is inadequate for computing implied values because web form errors amount to inconsistencies. Recall that in traditional TP an inconsistent premise set implies everything; hence, with traditional TP, all values

would be implied for all web form fields anytime a single error was present. Instead, implied values are computed using paraconsistent TP: where an inconsistent premise set does not necessarily imply every possible conclusion. Thus, in addition to specializing SAT/TP algorithms to implement error-detection/implied-value scripts, the web developer must choose an appropriate version of paraconsistent TP.

Techniques that can be applied to simplify web form construction and maintenance have been investigated by researchers in web engineering [6, 31, 34, 35], computer security [8, 32] formal methods [9], programming languages [5, 10, 17–19, 27], databases [7, 14], artificial intelligence [20, 22, 23], and configuration management [1, 28, 29, 33]. Most of the related work either prohibits web form users from causing errors or forces web form developers to define a paraconsistent version of implication by dictating which direction implied values can propagate (*e.g.*, through the syntax for form descriptions, through priorities, or by requiring web form fields to be structured hierarchically). Techniques that disallow errors are obviously inadequate for forms that allow errors, and forcing developers to dictate the direction implied values propagate results in forms where, for reasons unknown to the user, values only propagate in certain ways. Three notable exceptions [20, 22, 33] allow errors and utilize omni-directional versions of paraconsistent implication; however, [33] advocates approximate SAT/TP algorithms whose accuracy is unknown and [22] details only semantic definitions (for the special case where all form fields have a single value) without algorithmic results. The algorithms in [20] are the starting point for the work reported here; we have applied and tailored them to the web form domain and qualitatively improved their performance.

In this paper we describe PLATO, a tool that automatically constructs web forms from declarative descriptions provided by the web developer. Instead of writing HTML and browser scripts directly, the developer writes an ontology in classical logic that captures the constraints the web form data must satisfy. PLATO then compiles the ontology to (i) a SAT implementation customized to the ontology, (ii) a paraconsistent TP also customized to the ontology, and (iii) an HTML page that displays the form, highlights errors, and automatically fills-in implied values omni-directionally. The compilation process centers around the well-known resolution algorithm and utilizes an ontology-compression pre-processor to produce speed-ups of several orders of magnitude.

This paper is organized as follows. We begin with an example and our approach (Section 2). Our technical contributions, summarized below, follow.

- We report novel computational complexity results for a paraconsistent version of implication over a particular logical ontology language: the quantifier-free, function-free monadic fragment of first-order logic. (Section 3)
- We introduce the first compiler for web forms that generates error-detection and implied-value code specialized to the web form's ontology, outline its architecture, and discuss the challenges it addresses. (Section 4)
- We tailor and enhance existing compilation algorithms [20] to the web form problem. In particular, we introduce a compression algorithm that produces speed-ups of 10^5. (Section 5)
- We report the complexity for our algorithms, identify a special case for which our algorithms are optimal, and empirically evaluate our approach. (Section 6)

Subsequently, we report on related work (Section 7) and conclude (Section 8). Proofs have been omitted for lack of space but are available in the associated technical report.

2 Overview

Example. Figure 1 depicts a web form soliciting (a portion of) shipping and billing addresses for an e-commerce website. Notice that the shipping address is set to ⟨Chicago, Illinois⟩ and that the form includes a checkbox that indicates the shipping and billing addresses should be the same. If the user checks the checkbox, the form automatically copies ⟨Chicago, Illinois⟩ to the billing address. Had the user set the billing address instead of the shipping address, checking the checkbox would have propagated values in the opposite direction, exemplifying omni-directional implied value propagation.

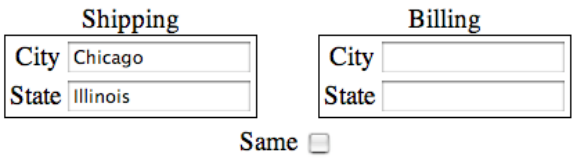


Fig. 1. Example web form

Starting with ⟨Chicago, Illinois⟩ for the shipping address and an empty billing address, suppose the user checks the checkbox, causing the form to fill-in ⟨Chicago, Illinois⟩ for the billing address. Now, suppose the user enters San Francisco for the billing city, thereby overriding the Chicago that was automatically filled-in. An error occurs because the two cities, Chicago and San Francisco, are supposed to be the same but are not. Without deleting one of three pieces of user-supplied data (Chicago, San Francisco, or the checkmark), there is no way to repair the error; hence, the form simply highlights the error for the user.

Approach. Traditionally, developers build such web forms by writing HTML to display the widgets and browser scripts to detect/highlight errors and compute/fill-in implied values. With PLATO, the developer provides a logical ontology describing the constraints the user-supplied data must satisfy (in addition to information about the display and range of permissible values for each field), and PLATO generates the corresponding HTML and browser scripts automatically.

For the example above, the developer provides PLATO with the following sentence that encodes the semantics of the checkbox. Below $Scity$ denotes the shipping address city, $Sstate$ the shipping state, $Bcity$ and $Bstate$ the billing city and state, and $same$ the checkbox. PLATO then generates the form described above.

$$same \Rightarrow \left(\bigwedge \begin{array}{l} Scity(x) \Leftrightarrow Bcity(x) \\ Sstate(x) \Leftrightarrow Bstate(x) \end{array} \right)$$

3 Logical Foundations of Web Forms

Here we give the logical foundations of web forms, errors, and implied values, and analyze the computational complexity of paraconsistent implication.

Fundamentally, the information web forms solicit from users is a set of key-value pairs[1]. Keys (form field names) are drawn from some predefined set F, and values are strings from some character set Σ (*e.g.*, Latin-1 or UTF-8). A web form submission, which we call a *payload*, is represented mathematically as a finite subset of $F \times \Sigma^*$. Logically, a payload is a set of sentences of the form $f(v)$ where $f \in F$ and $v \in \Sigma^*$. For example, the payload shown in Figure 1 is represented logically as $\{Scity(Chicago), Sstate(Illinois)\}$.

A server receiving a web form payload only accepts certain kinds of payloads, *e.g.*, those where the credit card's expiration date is in the future; all others are rejected. Mathematically, the set of *acceptable* payloads is simply a specific set of payloads, *i.e.*, a set of finite subsets of $F \times \Sigma^*$. Logically, the acceptable payloads correspond to the models satisfying a logical ontology.

In this paper we study a simple, first-order ontology language: monadic, quantifier-free, equality-free first-order logic. More precisely, the terms in our language are variables and object constants. Atoms take the form $p(t)$ where p is a predicate and t is a (single) term. Sentences are either atoms or $\{\wedge, \vee, \neg, \Rightarrow, \Leftarrow, \Leftrightarrow\}$ applied to sentences in the usual way. All variables are implicitly universally quantified. MON denotes all such sentences, and an ontology is a consistent subset of MON. The semantics are standard.

Web forms detect errors each time the user enters or edits data. A web form error arises whenever the current payload cannot be extended to an acceptable payload. Mathematically, a payload is *consistent* if it is a subset of some acceptable payload. All other payloads are *inconsistent*. A payload is *minimally inconsistent* if it is inconsistent and none of its subsets are inconsistent. There is one *error* in a payload for every minimally inconsistent subset contained within it. Logically, Λ is a consistent payload if it is logically consistent with the ontology Δ.

Web forms also fill-in implied values for the user automatically. For consistent payloads, implication is defined as usual. Suppose Δ is the web form ontology, and Λ is a consistent payload. The key-value pair $f(v)$ is positively implied by Λ with respect to Δ, written $\Lambda \models^{\Delta} f(v)$, if $f(v)$ belongs to every consistent superset of Λ. The key-value pair $f(v)$ is negatively implied, written $\Lambda \models^{\Delta} \neg f(v)$ if $f(v)$ belongs to no consistent superset of Λ.

Note that the above definition for implication is restricted to consistent payloads. When applied to an inconsistent payload (*i.e.*, a payload with errors), the definition results in all key-value pairs being both positively and negatively implied. Thus, for inconsistent payloads we say that a key-value pair is implied whenever there is a consistent fragment of the payload that implies the pair. Formally, an inconsistent Λ implies $f(v)$, written $\Lambda \models_E^{\Delta} f(v)$, if there is a consistent $\Lambda_0 \subseteq \Lambda$ such that $\Lambda_0 \models^{\Delta} f(v)$; likewise, $\Lambda \models_E^{\Delta} \neg f(v)$ if there is a consistent $\Lambda_0 \subseteq \Lambda$ such that $\Lambda_0 \models^{\Delta} \neg f(v)$.

Though strict implication was studied previously [20, 22], its computational complexity was unknown. Below we show that as long as P $\neq$ NP, the optimal algorithm is

[1] HTML 4.01 form specification: http://www.w3.org/TR/html401/interact/forms.html.

singly exponential, even with a number of strong restrictions. More positively, we show that if the size of the ontology is constant, strict impliciation is in included in P because it is included in LOGSPACE and AC^0.

Theorem 1 (Strict Implication Complexity). *Suppose Δ is in* MON*, and Λ is a finite set of ground atoms. $\Lambda \models_E^{\Delta} p(a)$ is Π_2^{p}-hard and included in Σ_3^{p}. If the number of variables appearing in Δ is bounded by a constant, strict implication is both Σ_1^{p}- (*NP*-) and Π_1^{p}- (*coNP*-) hard and is included in Σ_2^{p}. If Δ is in clausal form, contains a single variable, and includes no object constants, strict implication is Π_1^{p}- (*coNP*-) hard. If Δ is of constant size, $\Lambda \models_E^{\Delta} p(a)$ is included in AC^0.*

Proof. (Sketch) For the polynomial hierarchy results, the inclusion proofs are straightforward: guess a subset of Λ_0 (contributing an existential quantifier) and check if all models (contributing a universal quantifier) satisfy $\forall^* \Delta \Rightarrow p(a)$. (Since the language is monadic, each model is polynomial in the size of the signature.) If the number of variables is bounded by a constant, the check for satisfaction does not contribute a quantifier; otherwise, checking the satisfaction of $\exists^* \neg \Delta \vee p(a)$ (which is equivalent to the implication above) requires an additional existential quantifier.

For the polynomial hierarchy hardness proofs, we first show that strict implication is at least as hard as the well-known existential entailment. Then we embed the satisfiability of $\forall^* \exists^* . \phi$ into existential entailment over MON, where ϕ is monadic, quantifier-free, equality-free, function-free (which is Π_2^{p}-hard). For the restrictions, we embed both satisfiability and unsatisfiability of propositional logic.

For the AC^0 result, suppose Δ is of constant size. Slight modifications to the algorithms presented in this paper construct database queries by analyzing just Δ (which are therefore of constant size) that when evaluated over Λ compute strict implication. Since database query evaluation is included in AC^0 when the size of the queries is a constant, strict implication is included in AC^0. □

4 PLATO

PLATO is a tool that generates fully-functional web forms that provide real-time user feedback about errors (minimal inconsistencies) and implied values (strict implication). Below we discuss the high-level opportunities, challenges, and design decisions that lead to PLATO and follow up with PLATO's architecture. We describe PLATO's algorithms in Section 5.

4.1 Opportunities, Challenges, and Design Decisions

PLATO's design was dictated by two desires: (i) to provide users with a fast, powerful interface for entering web form data and (ii) to provide web developers with simple tools for constructing and maintaining such web forms. We begin by discussing the problem of programming the web browser to compute implied values.

Theorem proving versus knowledge compilation. Conceptually, the simplest way for the web browser to compute implied values is with a paraconsistent theorem prover written in JavaScript that takes as input the ontology, the current web form data, and a

query. This approach fails to leverage a powerful property of the web form domain: the ontology is fixed for the lifetime of the form. Hundreds or thousands of users might all use the same form and in so doing pose millions of queries, all over the same ontology. An implementation that analyzes the ontology anew for each query will repeat the same work over and over. Moreover, the computational complexity of implication when the ontology is fixed is strictly less than the complexity when it is not (see Theorem 1).

To leverage the static nature of the ontology, PLATO employs knowledge compilation [11] to construct JavaScript code that implements a paraconsistent theorem prover specialized to the given ontology. Intuitively, the manipulation of the ontology, which would normally happen at run time, happens at compile time, and the resulting code avoids performing that work for each query.

Compiling ontologies to JavaScript. Without errors, strict implication coincides with traditional implication; hence, constructing a theorem prover for paraconsistent implication specialized to a given ontology implicitly involves constructing a specialized theorem prover for traditional implication. Specializing a theorem prover for traditional implication requires generating JavaScript code that answers implication queries about that ontology. Despite the fact that an ontology can be interpreted as a set of boolean conditionals, this task is difficult because JavaScript and classical logic use disjunction differently. In JavaScript, once p and q are assigned values, `p || q` is a query asking if either p or q (or both) is true; in contrast, in classical logic, $p \vee q$ is akin to an assignment that makes p or q (or both) true without specifying which.

To address this challenge, PLATO decomposes the compilation of an ontology to JavaScript into two steps: compiling the ontology to database queries and compiling those database queries into JavaScript. Database queries are a useful intermediary because the database and JavaScript meanings of disjunction are the same, and techniques for translating database evaluation to imperative code are well-known [25].

Traditional implication to paraconsistent implication. A compiler for traditional implication that generates database queries can easily be adapted to strict implication: augment each database query with an auxiliary consistency-checking query that ensures the data used to answer the original query is consistent with the ontology. The problem is that if the queries are evaluated top-down, the same consistency checks may be executed repeatedly; similarly, if the consistency checks are evaluated bottom-up, many irrelevant consistency checks might be computed.

While standard techniques such as memoization and magic sets are applicable, PLATO utilizes the fact that the web forms it generates always maintain a list of errors, *i.e.*, a list of minimally inconsistent data sets. Consistency can then be checked with special-purpose code that detects whether a given data set contains no errors.

4.2 Architecture

PLATO's architecture is shown in Figure 2. The web developer provides an ontology and the set of web form field predicates (along with display and typing information about those predicates). The Classical Compiler constructs database queries that compute minimal inconsistencies and strict implication when evaluated over a web form payload. The Database Compiler then translates those queries into JavaScript code, which is then embedded in the HTML produced by the HTML Generator.

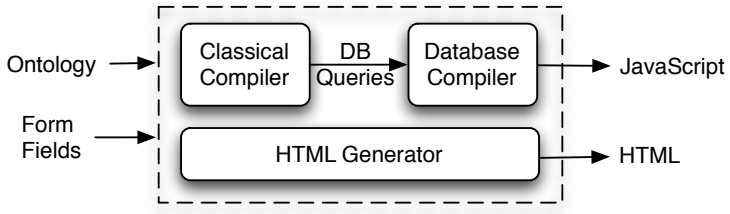


Fig. 2. PLATO architecture.

The novel component of PLATO, the Classical Compiler, solves two conceptually distinct problems: database-query generation for error-detection and database-query generation for implied-values. However, our solutions to the two problems are almost identical; hence, we focus on the more complex of the two: implied-values.

Formally, the implied-value problem the compiler addresses closely resembles the notation we use for strict implication: $\Lambda \models_E^{\Delta} f(v)$. Given an ontology Δ, the compiler must compute database queries that implement $\models_E^{\Delta}$, *i.e.*, when given a web form payload (the database), the queries must answer strict implication questions with respect to the ontology Δ. To simplify the exposition and proofs, we utilize the well-known equivalence of evaluating database queries on a database and evaluating first-order formulae on an interpretation.

Definition 1 (Web Form Constraint Compiler). *A web form constraint compiler is a function α that maps an ontology and a set of predicates to a set of first-order formulae. α is a compiler if for any ontology Δ, predicate set F, and predicate $f \in F$, there are sentences $\phi_f^+(x)$ and $\phi_f^-(x)$ in $\alpha[\Delta, F]$ such that for any payload Λ and any $v \in \Sigma^*$,*

$$\Lambda \models_E^{\Delta} f(v) \text{ if and only if } \models_{\Lambda} \phi_f^+(v) \text{ and}$$
$$\Lambda \models_E^{\Delta} \neg f(v) \text{ if and only if } \models_{\Lambda} \phi_f^-(v)$$

5 Algorithms

Here we explain the difficulty of compiling classical logic to database queries for error-detection and then discuss algorithms for strict implication and minimal inconsistency.

The naïve conversion of a classical ontology to database queries for detecting errors is straightforward: convert the ontology to conjunctive normal form, and treat each of the resulting clauses as a database query. For example, below is a simple ontology and the corresponding database (or logic programming) queries.

Ontology	Database Queries
$p(x) \Rightarrow q(x)$	$error :- p(x) \wedge \neg q(x)$
$q(x) \Rightarrow \neg r(x)$	$error :- q(x) \wedge r(x)$

This conversion fails to preserve the semantics of the ontology because classical logic uses the open world assumption, but databases use the closed world assumption

(CWA); thus, classical logic allows form fields to be unknown, but databases require every form field value to be either true or false.

The queries above are unsound for the payload $\{p(a)\}$, *i.e.*, where p is assigned a and both q and r unknown. To see this, notice that the first database query evaluates to true, thereby signaling an error, because the CWA deems $\neg q(a)$ true; however, the payload is consistent with the ontology. Such errors can be eliminated by only evaluating queries whose form fields are all known.

Assuming the only evaluated queries are those with known form fields, the queries above are incomplete for the payload $\{p(a), r(a)\}$ (where q is unknown). Neither of the database queries above can be evaluated because both rely on q, an unknown value, yet the payload is inconsistent with the ontology. Such incompleteness can be eliminated by accounting for the interaction of the constraints.

PLATO's compilation algorithms expand the ontology to take constraint interaction into account. Whenever an error or implied value arises, the expanded ontology includes a single constraint that detects it without any unknown form fields. For the example above, PLATO generates an additional query: $error :- p(x) \vee r(x)$.

This example also illustrates an inadequacy of today's HTML forms: without using additional fields or special values, there is no way to differentiate selecting zero values for a field and leaving that field unknown. Both are communicated to the server in the same way. Currently, PLATO treats a field with zero values as unknown.

5.1 Strict Implication

PLATO's basic algorithm for constructing database queries implementing strict implication is a five-step process: compute the resolution closure of the web form ontology, compute the contrapositives of each clause in the closure, eliminate all rules with negation in the body, augment each contrapositive with a consistency check, and invoke predicate completion.

We illustrate with the ontology from above: $(\neg p(x) \vee q(x)) \wedge (\neg q(x) \vee \neg r(x))$. The resolution closure adds a single clause: $p(x) \vee \neg r(x)$. Computing the contrapositives, eliminating rules with negation in the body, and appending consistency checks is straightforward and produces the following rules.

$$\begin{aligned} q(x) &\Leftarrow p(x) \wedge \mathit{consistent}_{p(x)}(x) \\ \neg q(x) &\Leftarrow r(x) \wedge \mathit{consistent}_{r(x)}(x) \\ \neg r(x) &\Leftarrow q(x) \wedge \mathit{consistent}_{q(x)}(x) \\ \neg p(x) &\Leftarrow r(x) \wedge \mathit{consistent}_{r(x)}(x) \\ \neg r(x) &\Leftarrow p(x) \wedge \mathit{consistent}_{p(x)}(x) \end{aligned}$$

The consistency checks ensure that witnesses for implication are consistent with the entire ontology.

Definition 2 (*$consistent_{\phi(\bar{x})}$* **[20]).** *For the ontology Δ and sentence $\phi(\bar{x})$, $consistent_{\phi(\bar{x})}(\bar{t})$ is true if and only if $\{\phi(\bar{t})\} \cup \Delta$ is consistent.*

Predicate completion then constructs the first-order formula defining strict implication for each signed predicate ρ: the disjunction of all the rules with ρ in the head.

$$\phi_q^+(x) \equiv p(x) \wedge \mathit{consistent}_{p(x)}(x)$$

$$\phi_q^-(x) \equiv r(x) \wedge consistent_{r(x)}(x)$$
$$\phi_r^-(x) \equiv (q(x) \wedge consistent_{q(x)}(x)) \vee (p(x) \wedge consistent_{p(x)}(x))$$
$$\phi_p^-(x) \equiv r(x) \wedge consistent_{r(x)}(x)$$
$$\bot \equiv \phi_p^+(x) \equiv \phi_r^+(x)$$

This basic algorithm is easy to implement, though there are obvious efficiency problems with computing the resolution closure. To mitigate the expense of resolution, PLATO first compresses the ontology. Consider the following example.

Ontology	Compression
$p(a) \wedge q(b) \wedge r(c)$	$p(x) \wedge q(y) \wedge r(z) \Rightarrow t(x,y,z)$
$\bigvee\ p(b) \wedge q(d) \wedge r(e)$	$t(a,b,c)$
$p(d) \wedge q(c) \wedge r(a)$	$t(b,d,e)$
	$t(d,c,a)$

The ontology on the left lists the possible combinations of p, q, and r in disjunctive normal form. The compression on the right represents the ontology as a single constraint over p, q, and r together with a new predicate t and a database table defining t's semantics as the permitted combinations of p, q, and r. Importantly, the database table t is not included when the resolution closure is computed; rather, it is treated as part of the database representing the web form data. Instead of computing the closure of 28 clauses, PLATO computes the closure of 1 clause; the drawback is that the 1 clause is not monadic because of $t(x,y,z)$.

Algorithm 1, named IMPLCOMPILE, formalizes the algorithm outlined here.

Algorithm 1 IMPLCOMPILE [Δ, F]

Outputs: A set of first-order equivalences.

1: Δ := RES[COMPRESS[Δ]]
2: $\Gamma_p^s := \emptyset$ for all predicates $p \in F$ and all $s \in \{+,-\}$
3: **for all** contrapositives d in $\bigcup_{p \in F}\{p(x) \vee \neg p(x)\} \cup \Delta$ **do**
4: write d as $\pm p(x) \Leftarrow \phi(x, \bar{y})$
5: **if** $p \in F$ and $\neg$ does not occur in $\phi(x, \bar{y})$ **then**
6: $\Gamma_p^{\pm} := \{\exists \bar{y}.\phi(x,\bar{y}) \wedge consistent^{\Delta}_{\phi(x,\bar{y})}\} \cup \Gamma_p^{\pm}$
7: **end if**
8: **end for**
9: **print** $\phi_p^s \equiv \bigvee \Gamma_p^s$ for all predicates $p \in F$ and all $s \in \{+,-\}$

Theorem 2 (Soundness and Completeness). *Without compression, algorithm* IMPLCOMPILE *is a web form constraint compiler for* MON *ontologies.*

5.2 Minimal Inconsistencies

Computing minimal inconsistencies is useful for two reasons: to identify errors and to implement the consistency checks described above. PLATO's algorithm identifies the

minimal inconsistent subsets by computing an over-approximation and then throwing out non-minimal subsets.

More precisely, the algorithm computes an *update* to the set of minimally inconsistent subsets as opposed to computing the entire set from scratch. The web form paradigm supports such updates naturally. Each time a user changes a form field, it is only the minimally inconsistent sets involving that field that need to be changed.

The algorithm, called CONSCOMPILE, is identical to IMPLCOMPILE except it adds no consistency check to the database queries that are generated and eliminates all rules with positive heads. It consists of five steps: compress the ontology, compute the resolution closure, compute the contrapositives of each clause in the closure, eliminate all rules with a positive head or with negation in the body, and perform predicate completion. For lack of space, we omit the formal definition.

The only difference between the queries generated by CONSCOMPILE and the error queries in the example at the start of the section is that instead of having a collection of statements of the form $error :- q(x) \wedge r(x)$, each form field is associated with a set of queries, *e.g.*, field q is associated with $\neg q(x) :- r(x)$ and field r is associated with $\neg r(x) :- q(x)$. If the user makes $q(a)$ true, the form evaluates $\neg q(a)$ using the queries associated with q, records all the form data subsets responsible for making $\neg q(a)$ true, adds $q(a)$ to each subset, and eliminates any subsets that are non-minimal.

6 Evaluation

Our evaluation of plato includes an analytical component, where we focus on resolution, and an empirical component, where we focus on ontology compression.

6.1 Analytical

PLATO's performance has two components: the performance of the compiler and the performance of the code the compiler produces. The performance of the compiler is polynomial in the performance of the resolution theorem prover; the performance of the code the compiler produces is directly related to the *size* of the resolution closure. Since the performance of the theorem prover is bounded from below by the size of the closure, the closure size (*i.e.*, output complexity of resolution) is of paramount interest.

The main reason PLATO's ontology language is no more expressive than MON is that the resolution closure of MON is finite. In particular, resolution's output complexity is either singly or doubly exponential in the size of the input.

Proposition 1 (Resolution Complexity). *The output complexity of resolution for* MON *is* EXPSPACE*-hard and included in* 2EXPSPACE*. When the premises are in clausal form, contain one variable, and include no object constants, the output complexity is* EXPSPACE*-complete.*

Proof. (Sketch) For inclusion in 2EXPSPACE, count the number of monadic clauses. Because a monadic clause may include multiple variables, *e.g.*, $p(x) \vee q(x) \vee \neg r(y)$, each such clause corresponds to a set of propositional clauses, *e.g.*, $\{p \vee q, \neg r\}$. The number of distinct sets of propositional clauses is 2^{3^n}, where n is the number of propositions

(corresponding to the number of monadic predicates). Object constants only introduce a single exponential factor. For hardness, we embed propositional logic, where resolution proofs and therefore resolution closures are well-known to be exponential. For the special case, the closure is the same size as the closure of propositional logic. □

This result has two consequences. The first is that the run-time of the compiler is exponential, which means it will not scale to large ontologies; however, that does not mean PLATO fails to scale to large web *forms*. Large web forms often have relatively small ontologies or have large ontologies that consist of many small, almost independent ontologies. Large web forms with large, complex ontologies are rare simply because people have trouble filling them out; those that exist (*e.g.*, TurboTax) are usually professionally designed to help people navigate them successfully.

Second, the set of database queries the compiler outputs is exponentially larger than the ontology. Fortunately, it turns out that evaluating one MON database query is singly exponential (in combined complexity), ensuring that the implementations of strict implication and inconsistency detection run in a singly exponential factor of the size of the resolution closure. Because strict implication and inconsistency detection are NP-hard, for any class of ontologies for which resolution's output complexity is EXPSPACE, PLATO produces singly exponential implementations of strict implication and inconsistency detection, which is optimal with respect to time if $\mathrm{P} \neq \mathrm{NP}$. Furthermore, ontologies written in clausal form with a single variable and no object constants enjoy the EXPSPACE result.

Proposition 2. *For any class of* MON *for which resolution's output complexity is in* EXPSPACE*, without compression* PLATO *produces time-optimal implementations of strict implication and inconsistency detection, unless* $\mathrm{P} = \mathrm{NP}$.

Corollary 1 (Optimality). *For ontologies written in clausal form with a single variable and no object constants, without compression* PLATO *generates time-optimal implementations of strict implication and inconsistency detection, unless* $\mathrm{P} = \mathrm{NP}$.

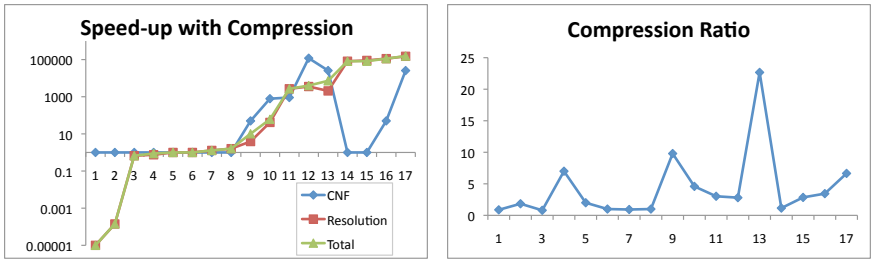
6.2 Empirical

To test the effectiveness of the pre-resolution compression step of IMPLCOMPILE and CONSCOMPILE, we compared the performance of resolution both with and without compression on ontologies from CLib[2], a library of configuration management problems. We chose to test configuration management problems because they represent some of the most complex ontologies PLATO could be expected to handle. We analyzed all 5 of the problems in the Configit format that were supported by our Configit parser at the time of writing. Some Configit problems are decomposed into several components, each of which contains its own ontology. Moreover, for each ontology, we tested two versions: one requiring each form field to have a single value and one that does not. All told, the 5 Configit problems produced 26 distinct ontologies.

We tested both a compressed and an uncompressed version of each ontology. We timed both the conversion to clausal form (CNF), with a max of 900 seconds, and the

[2] http://www.itu.dk/research/cla/externals/clib/

computation of the resolution closure, again with a 900 second max. For 17 ontologies, either the compressed version, the uncompressed version, or both finished before timing out on either step; we report results for those 17 ontologies.



(a) Ratio of uncompressed performance to compressed performance

(b) Ratio of uncompressed size to compressed size

Fig. 3. Impact of compression

Figure 3(a) shows three ratios of uncompressed performance to compressed performance, where high numbers mean compression is beneficial: the time for computing clausal form, the time for computing the resolution closure, and the total time. The 17 test cases are ordered from low to high in terms of total-time performance improvement. (Note there is no relationship between ontologies; however, we find the graphs easier to read when points are ordered and connected with lines.) The resolution and total-time results are virtually identical, indicating that the performance change in CNF conversion due to compression is negligible. The total-time results are mixed. For 9 ontologies, compression improves performance with speed-ups between 10x and 150,000x. For 6 ontologies, compression has little impact. For 2 ontologies, compression is harmful, with slow-downs of 7,000x and 100,000x. Slow downs arise because, despite the fact that the ontology is smaller, it contains predicates with more than one argument.

Compression is therefore sometimes quite useful, but it can also be harmful. Figure 3(b) shows the size ratio of the uncompressed to compressed ontologies for each of the 17 test cases, where size is measured as sentence complexity, *i.e.*, number of boolean connectives and atomic sentences. We conjectured that a high compression ratio would indicate high performance benefits, but some of the instances that benefited most from compression have ratios similar to those for the instances most harmed by compression.

Because it is unclear how to predict when compression will be beneficial, PLATO compresses every ontology and then attempts to compute the closure for some small period of time, *e.g.*, one minute. If the closure of the compressed ontology has not been computed before time expires, it computes the closure of the uncompressed ontology.

The current compression algorithm runs in time linear in the size of the ontology, and for all examples, compression time was negligible. To generate the resolution closure, we used the SNARK automated reasoning kit. All tests were run on a MacBook Pro with a 2.66 GHz Intel Core i7 and 8 GB of memory.

7 Related Work

Related work touches on three topics: web application development, inconsistency tolerance for classical logic, and knowledge compilation. See Section 1 for a discussion of work related to web application development.

Inconsistency tolerance for classical logic has received significant attention over the last decade (see [4] for a recent overview). Because this paper focuses on detecting and tolerating inconsistencies instead of repairing them (*e.g.*, [2, 13, 30]), the closest related work centers around definitions and implementations for paraconsistent implication. Perhaps the closest definition to our strict implication is the well-known existential entailment. Strict implication is better suited for the web-form setting because it differentiates the ontology from the data, whereas existential entailment does not; moreover, our implementation utilizes specific properties of the MON ontology language, which is better suited to the web form domain than propositional logic (the ubiquitous choice for studying existential entailment) but is more implementable than full first-order logic [3, 12]. Another related topic is argumentation theory. Whereas that work is usually concerned with establishing the relationships between all possible arguments with an argument tree, *e.g.*, [4, 12], PLATO constructs two arguments for each atomic conclusion: one supporting and one undermining.

In the context of knowledge compilation, our work is differentiated from most because it addresses inconsistency tolerance. The other efforts we are aware of that address both inconsistency and compilation [15, 16, 20, 21] fail to address the web form's real-time performance demands or fail to capitalize on the properties of the web form domain. Ignoring inconsistency tolerance, the most relevant knowledge compilation work transforms description logic ontologies into relational databases to efficiently reason about large data sets. In their terminology, the web form's constraints correspond to a TBox, the web form data corresponds to an ABox, and the web form domain only requires (positive and negative) instance queries. Our algorithms infuse the TBox into all possible instance queries at compilation-time but leave the database untouched; thus, according to [24], it is a form of *FO-rewriting*, as opposed to *combined FO-rewriting*.

8 Conclusion and Future Work

This paper introduced PLATO, a compiler for constructing web forms that detect errors and compute implied values. In essence, PLATO specializes an inconsistency-tolerant (*i.e.*, paraconsistent) theorem prover to a given ontology to capitalize on the fact that hundreds or thousands of users might combine to ask millions of queries all about the same ontology. We materialized this intuition in formal terms by showing that the parameterized complexity of the web form problem is polynomial when the size of the ontology is fixed. We introduced easy-to-implement compilation algorithms and analyzed how they scale under non-parameterized complexity assumptions. We identified a class of ontologies for which PLATO constructs time-optimal code and demonstrated compression algorithms yielding speed-ups of 10^5. PLATO is available online at `http://tlh.cs.uchicago.edu:5000/plato/`.

Our long-term goal is to provide web developers with a practical tool for building and maintaining web forms. In the future we plan to investigate ontology languages that

are more expressive than the monadic, first-order logic studied here but that retain some of the same computational properties. We hope to guide that work by investigating a version of PLATO that simplifies the construction of a common class of web forms: those that solicit data for a back-end database. The improved PLATO would accept a declarative description of the database view the user is intended to augment and would automatically extract the appropriate ontology from the database integrity constraints.

Acknowledgements. Thanks to Stuart Kurtz, Matthias Baaz, Eyal Amir, and Alexander Razborov for discussions about the complexity results in this paper. Thanks to Jui Yi Kao, Mike Genesereth, and the anonymous referees for comments on the paper.

References

1. Axling, T., Haridi, S.: A tool for developing interactive configuration applications. In: Proceedings of the Journal of Logic Programming. pp. 147–168 (1996)
2. Benferhat, S., Lagrue, S., Rossit, J.: An egalitarist fusion of incommensurable ranked belief bases under constraints. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 367–372 (2007)
3. Besnard, P., Hunter, A.: Practical first-order argumentation. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 590–595 (2005)
4. Besnard, P., Hunter, A.: Elements of Argumentation. MIT Press (2008)
5. Brabel, B., Hanus, M., Muller, M.: High-level database programming in curry. In: Proceedings of the Symposium on Practical Aspects of Declarative Languages. pp. 316–332 (2008)
6. Brabrand, C., Moller, A., Ricky, M., Schwartzbach, M.: Powerforms: Declarative client-side form field validation. World Wide Web pp. 205–214 (2000)
7. Brambilla, M., Ceri, S., Comai, S., Dario, M., Fraternali, P., Manolescu, I.: Declarative specification of web applications exploiting web services and workflows. In: Proceedings of the ACM SIG for the Management of Data. pp. 909–910 (2004)
8. Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure web applications via automatic partitioning. In: Proceedings of the ACM Symposium on Operating Systems Principles. pp. 31–44 (2007)
9. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: Proceedings of the International Symposium on Formal Methods for Components and Objects (2006)
10. Cox, P.T., Nicholson, P.: Unification of arrays in spreadsheets with logic programming. In: Proceedings of the Symposium on Practical Aspects of Declarative Languages. pp. 110–115 (2007)
11. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research 17, 229–264 (2002)
12. Efstathiou, V., Hunter, A.: Algorithms for effective argumentation of classical propositional logic. In: Proceedings of the Symposium on Foundations of Information and Knowledge Systems (2008)
13. Everaere, P., Konieczny, S., Marquis, P.: Conflict-based merging operators. In: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning. pp. 348–357 (2008)
14. Fernandez, M., Florescu, D., Levy, A., Suciu, D.: Declarative specification of web sites with strudel. The VLDB Journal pp. 38–55 (2000)
15. Flouris, G., Huang, Z., Pan, J.Z., Plexousakis, D., Wache, H.: Inconsistencies, negations and changes in ontologies. In: Proceedings of the AAAI Conference on Artificial Intelligence. pp. 1295–1300 (2006)

16. Gomez, S.A., Chesnevar, C.I., Simari, G.R.: An argumentative approach to reasoning with inconsistent ontologies. In: Proceedings of the KR Workshop on Knowledge Representation and Ontologies. pp. 11–20 (2008)
17. Gupta, G., Akhter, S.F.: Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In: Proceedings of the Symposium on Practical Aspects of Declarative Languages. pp. 308–323 (2000)
18. Hanus, M., Klub, C.: Declarative programming of user interfaces. In: Proceedings of the Symposium on Practical Aspects of Declarative Languages. pp. 16–30 (2009)
19. Hanus, M., Koschnicke, S.: An ER-based framework for declarative web programming. In: Proceedings of the Symposium on Practical Aspects of Declarative Languages. pp. 201–216 (2010)
20. Hinrichs, T.L., Kao, J.Y., Genesereth, M.R.: Inconsistency-tolerant reasoning with classical logic and large databases. In: Proceedings of the Symposium of Abstraction, Reformulation, and Approximation (2009)
21. Huang, Z., van Harmelen, F., ten Teije, A.: Reasoning with inconsistent ontologies. In: Proceedings of the International Joint Conference on Artificial Intelligence (2005)
22. Kassoff, M., Genesereth, M.R.: PrediCalc: A logical spreadsheet management system. Knowledge Engineering Review 22(3), 281–295 (2007)
23. Kassoff, M., Valente, A.: An introduction to logical spreadsheets. Knowledge Engineering Review 22(3), 213–219 (2007)
24. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: Combined FO rewritability for conjunctive query answering in DL-Lite. In: Proceedings of the International Workshop on Description Logic (2009)
25. Levy, M.R., Horspool, R.N.: Translating Prolog to C: a WAM-based approach. In: Proceedings of the Compulog Network Area Meeting on Programming Languages (1993)
26. Libkin, L.: Elements of Finite Model Theory. Springer-Verlag (2004)
27. Serrano, M., Gallesio, E., Loitsch, F.: Hop, a language for programming the web 2.0. In: Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications. pp. 975–985 (2006)
28. Soininen, T., Niemela, I.: Developing a declarative rule language for applications in product configuration. In: Proceedings of the Symposium on Practical Aspects of Declarative Languages. pp. 305–319 (1999)
29. Subbarayan, S., Jensen, R., Hadzic, T., Andersen, H., Hulgaard, H., Moller, J.: Comparing two implementations of a complete and backtrack-free interactive configurator. In: Proceedings of the CP Workshop on CSP Techniques with Immediate Application. pp. 97–111 (2004)
30. Subrahmanian, V.S., Amgoud, L.: A general framework for reasoning about inconsistency. In: Proceedings of the International Joint Conference on Artificial Intelligence. pp. 599–604 (2007)
31. Suzuki, T., Tokuda, T.: Automatic generation of intelligent javascript programs for handling input forms in html documents. In: Proceedings of the International Conference on Web Engineering (2005)
32. Vikram, K., Prateek, A., Livshits, B.: Ripley: Automatically securing distributed web applications through replicated execution. In: Proceedings of the ACM Conference on Computer and Communications Security. pp. 173–186 (2009)
33. Vlaeminck, H., Vennekens, J., Denecker, M.: A logical framework for configuration software. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. pp. 141–148 (2009)
34. Yang, F., Gupta, N., Gerner, N., Qi, X., Demers, A., Gehrke, J., Shanmugasundaram, J.: A unified platform for data driven web applications with automatic client-server partitioning. In: Proceedings of the International World Wide Web Conference. pp. 341–350 (2007)

35. Yang, F., Shanmugasundaram, J., Riedewald, M., Gehrke, J.: Hilda: A high-level language for data-driven web applications. In: Proceedings of the International Conference on Data Engineering (2006)

A Appendix

A.1 Paraconsistent Entailment Complexity

Here we give complexity results for two notions of paraconsistent entailment for quantifier-free, function-free, monadic first-order logic, denoted MON. The first is the well-known existential entailment: $\Delta \models_E \phi$ if there is a satisfiable $\Delta_0 \subseteq \Delta$ such that $\Delta_0 \models \phi$. The second is strict entailment, where the premises Γ are restricted to be a set of atoms. $\Gamma \models_E^{\Delta} \phi$ if there is a $\Gamma_0 \subseteq \Gamma$ such that $\Gamma_0 \cup \Delta$ is consistent and entails ϕ.

For each paraconsistent entailment relation we will consider several variations of MON that differ in two dimensions: variables and equality. We consider an arbitrary number of variables (denoted MON_*) and a constant number of variables (MON_k). Orthogonally, we consider MON with and without equality. When equality is allowed, we superscript the language with = to produce either $\text{MON}_k^=$ or $\text{MON}_*^=$. Finally, we consider the case where the MON ontology is of constant size.

We show complexity results for atomic queries. None of our results are tight, but because they place the entailment relations in the lower levels of the polynomial hierarchy (with the exception of a constant-size ontology being in P), our results are sufficient to guarantee that as long as the ontology is not a constant size, the optimal algorithm is singly-exponential in time (barring a collapse of the lower levels of the polynomial hierarchy).

Existential Entailment

Proposition 3 ($\text{MON}_* \models_E$ hardness). *Suppose Δ is a finite set of quantifier-free, function-free, equality-free monadic first-order sentences with two relation constants. $\Delta \models_E p(a)$ is Π_2^{p}-hard.*

Proof. The following problem is Σ_2^{p}-hard. Given a first-order formula $\exists^*\forall^*\phi$ where ϕ is quantifier-free with at most one unary relation constant, no functions, no equality, determine if $\exists^*\forall^*\phi$ is satisfiable. First we claim that $\exists^*\forall^*\phi \models p(a)$ is Π_2^{p}-hard even if $\exists^*\forall^*\phi$ is known to be satisfiable. Second we claim that for satisfiable $\exists^*\forall^*\phi$, $\exists^*\forall^*\phi \models p(a)$ if and only if $\exists^*\forall^*\phi \models_E p(a)$. Third we claim that $\exists^*\forall^*\phi \models_E p(a)$ if and only if the skolemization $\forall^*\phi' \models_E p(a)$. Thus, we can embed a Π_2^{p}-hard problem in polynomial time within existential entailment where the premises are in quantifier-free (equivalently $\forall^*$), function-free, equality-free monadic first-order logic with two relation constants: the one appearing in ϕ and p. Now we prove the three claims.

Claim 1: $\exists^*\forall^*\phi \models p(a)$ for a fresh $p(a)$ is Π_2^{p}-hard even if $\exists^*\forall^*\phi$ is known to be satisfiable. Since satisfiability of $\exists^*\forall^*\phi$ is Σ_2^{p}-hard, unsatisfiability is Π_2^{p}-hard. We show that $\exists^*\forall^*\phi$ is unsatisfiable if and only if $\neg\exists^*\forall^*\phi \Rightarrow p(a) \models p(a)$. ($\Rightarrow$) If $\exists^*\forall^*\phi$ is unsatisfiable, $\neg\exists^*\forall^*\phi$ is true in all models, and every model that satisfies $\neg\exists^*\forall^*\phi \Rightarrow p(a)$ satisfies $p(a)$ and hence entailment holds. ($\Leftarrow$) Suppose entailment holds, then

there is a resolution proof of the empty clause from (the clausal form of) $\exists^*\forall^*\phi \vee p(a)$ and $\neg p(a)$. If $p(a)$ does not appear in the proof, we have a proof of the unsatisfiability of $\exists^*\forall^*\phi$; otherwise, we can reorder the proof so that $p(a)$ is eliminated first (using pure literal elimination). The resulting proof includes within it a proof of the unsatisfiability of $\exists^*\forall^*\phi$ since the elimination of $p(a)$ makes no variable bindings.

Claim 2: for satisfiable $\exists^*\forall^*\phi$, $\exists^*\forall^*\phi \models p(a)$ if and only if $\exists^*\forall^*\phi \models_E p(a)$. ($\Rightarrow$) Suppose entailment holds. Then since the premise is satisfiable, it is the witness required for existential entailment. ($\Leftarrow$) Suppose existential entailment holds. Then since $p(a)$ is not a tautology, the only premise set that could entail it is $\{\exists^*\forall^*\phi\}$, and hence entailment must hold.

Claim 3: $\exists^*\forall^*\phi \models_E p(a)$ if and only if $\forall^*\phi' \models_E p(a)$ where $\forall^*\phi'$ is the skolemization of $\exists^*\forall^*\phi$. By skolemization, we know that the equivalence holds under traditional entailment. Moreover, skolemization preserves satisfiability; hence the two premises are equally satisfiable and equally entail $p(a)$; hence, the existential entailment of $p(a)$ for the two premises is the same.

Proposition 4 ($\textbf{MON}_k \models_E$ hardness). *Suppose Δ is a finite set of quantifier-free, function-free, equality-free monadic first-order sentences with at most a constant k number of variables. $\Delta \models_E p(a)$ is Σ_1^{p}-hard and Π_1^{p}-hard.*

Proof. Here we embed the satisfiability and unsatisfiability of propositional logic within existential entailment to arrive at the Σ_1^{p}-hardness and Π_1^{p}-hardness bounds, respectively. Propositional logic can be encoded easily in monadic logic using no variables, one object constant b, and one monadic predicate for each propositional constant. For any Herbrand interpretation, a monadic predicate p is assigned the object b if and only if the original proposition were true in the corresponding propositional interpretation. Below, we ignore such details.

We know that the satisfiability of a sentence ϕ in propositional logic is NP-hard (Σ_1^{p}-hard). We claim that ϕ is satisfiable if and only if $\{\phi, \phi \Rightarrow p\} \models_E p$, where p is fresh. ($\Rightarrow$) Suppose ϕ is satisfiable. Then both the premises are also satisfiable, and by a single application of modus ponens, the premises entail p. This ensures the premises existentially entail p. ($\Leftarrow$) Suppose existential entailment holds. Since p is fresh and not a tautology, the only premise subset that could entail it is $\{\phi, \phi \Rightarrow p\}$. Since that premise set is satisfiable, so are its members, and hence ϕ is satisfiable. Since both directions hold, we can embed propositional satisfiability within existential entailment (with a constant number of variables) and hence we have Σ_1^{p}-hardness.

We also know that the unsatisfiability of a propositional sentence ϕ is coNP-hard (Π_1^{p}-hard). Consequently, entailment $\phi \models p$ is coNP-hard, and so is entailment when the premise ϕ is known to be satisfiable. We claim that for satisfiable ϕ, $\phi \models p$ if and only if $\phi \models_E p$. ($\Rightarrow$) Suppose $\phi \models p$. Then since ϕ is satisfiable, it is the witness ensuring that $\phi \models_E p$. ($\Leftarrow$) Suppose $\phi \models_E p$. Then since p is not a tautology, the only subset that can entail it is $\{\phi\}$, and hence $\phi \models p$.

Now we prove that $\phi \models p$ is coNP-hard even when ϕ is satisfiable. We show that ϕ is unsatisfiable if and only if $\neg\phi \Rightarrow p \models p$ for a fresh p. ($\Rightarrow$) Suppose ϕ is unsatisfiable. Then in every model $\neg\phi$ is true, and in every model satisfying $\neg\phi \Rightarrow p$, p must be true since $\neg\phi$ is true; thus, p is entailed. ($\Leftarrow$) Suppose entailment holds. Then there

is a resolution proof of the empty clause from $\phi \vee p$ and $\neg p$. For any such proof, if p does not occur, that proof demonstrates that ϕ is unsatisfiable; otherwise, place the removal of the p from clauses of ϕ first, and the result is a proof that demonstrates the unsatisfiability of ϕ.

Proposition 5 ($\textbf{MON}_k \models_E$ fragment hardness). *Suppose Δ is a finite set of quantifier-free, function-free, equality-free monadic first-order sentences in clausal form with one variable and no object constants. $\Delta \models_E p(a)$ is Π_1^{p}-hard.*

Proof. We know that the unsatisfiability of a propositional sentence set Γ is coNP-hard, even when Γ is in clausal form. We claim that Γ is unsatisfiable if and only if $\neg\Gamma \Rightarrow p \models p$ for a fresh p. First, notice that $\neg\Gamma \Rightarrow p$ is satisfied by the model where p is true. Second, $\neg\Gamma \Rightarrow p$ is equivalent to $\Gamma \vee p$, and since Γ is already in clausal form, the conversion of $\Gamma \vee p$ to clausal form requires only adding p to every element of Γ. Assuming the claim, this ensures that $\Gamma \models p$ is coNP-hard even if Γ is satisfiable and in clausal form.

Now we prove the claim. ($\Rightarrow$) Suppose Γ is unsatisfiable. Then $\neg\Gamma$ is true in all models, and every model satisfying the premise satisfies p; hence, p is entailed. ($\Leftarrow$) Suppose $\neg\Gamma \Rightarrow p \models p$. Then the resolution proof of the empty clause can be arranged so that p is resolved away first, leaving a proof of the empty clause from Γ.

We know $\Gamma \models p$ in propositional logic is coNP-hard, even if Γ is in clausal form and known to be satisfiable; hence, the unsatisfiability of $\Gamma \cup \{\neg p\}$ is coNP-hard. Let $\Gamma(x)$ be the clause set Γ where every propositional literal has been changed into a monadic literal using the variable x. We claim that $\Gamma(x) \cup \{\neg p(x)\}$, where $\Gamma(x)$ is satisfiable is unsatisfiable if and only if $\Gamma \cup \{\neg p\}$ is unsatisfiable. To see this, consider a resolution proof in the monadic version. Note that every resolution binds x to x, and hence we can ignore all variables during resolution. A proof of the empty clause in either version allows us to reconstruct a proof of the empty clause in the other version; moreover, the proofs are structurally identical. This suffices to conclude that the unsatisfiability of $\Gamma(x) \cup \{\neg p(x)\}$, where $\Gamma(x)$ is satisfiable and in clausal form is coNP-hard.

The clause set $\Gamma(x) \cup \{\neg p(x)\}$ is unsatisfiable if and only if $\Gamma(x) \cup \{\neg p(a)\}$ is unsatisfiable, by Herbrand's theorem. (Each set is unsatisfiable if and only if its grounding is unsatisfiable, and in the first case since there is no object constant, we are free to choose a constant, say a. Then both sets have exactly the same grounding.) Additionally, $\Gamma(x) \cup \{\neg p(a)\}$ is unsatisfiable if and only if $\Gamma(x) \models p(a)$. Putting these two facts together allows us to embed a coNP-hard problem within our class of existential entailment queries and hence guarantees coNP-hardness (Π_1^{p}-hardness).

Corollary 2 ($\textbf{MON}_*^{=}$ and $\textbf{MON}_k^{=} \models_E$ hardness). *Suppose Δ is a finite set of quantifier-free, function-free, monadic first-order sentences obeying the unique-names assumption. $\Delta \models_E p(a)$ is Σ_2^{p}-hard. If the number of variables is bounded by a constant k then $\Delta \models_E p(a)$ is Σ_1^{p}-hard and Π_1^{p}-hard, and if additionally Δ is restricted to clausal form, $\Delta \models_E p(a)$ is Π_1^{p}-hard.*

Proof. Trivially $\textsc{mon}_*$ can be embedded in $\textsc{mon}_*^{=}$; likewise, $\textsc{mon}_k$ can be embedded in $\textsc{mon}_k^{=}$. Since $\textsc{mon}_*$ is Σ_2^{p}-hard, so is $\textsc{mon}_*^{=}$. Likewise, since $\textsc{mon}_k$ is Σ_1^{p}-hard and

Π_1^{p}-hard, so is $\text{MON}_k^=$, and since the restriction to clausal form for a fragment of MON_k is Π_1^{p}-hard, so is the restriction of $\text{MON}_k^=$ to clausal form Π_1^{p}-hard.

Proposition 6 ($\text{MON}_*^= \models_E$ inclusion). *Suppose Δ is a finite set of quantifier-free, function-free first-order sentences obeying the unique names assumption where every relation constant has arity at most some constant k. $\Delta \models_E p(a)$ is included in Σ_3^{p}.*

Proof. Consider the following alternating time algorithm.

1. (Existential) Guess a subset Δ_0 of Δ.
   // Check satisfiability
   2. (Existential) Guess a Herbrand interpretation I
      3. (Universal) For all variable assignments v
         4. Validate that $\models_I \Delta_0[v]$
   // Check entailment
   4. (Universal) For all Herbrand interpretations I
      5. (Existential) Guess a variable assignment v.
         6. Validate that $\models_I \neg\Delta_0[v] \vee p(a)$

Here we use the fact that $\models \forall\bar{x}.\phi(\bar{x}) \Rightarrow p(a)$ if and only if every interpretation satisfies $\exists\bar{x}.\neg\phi(\bar{x}) \vee p(a)$ or equivalently that for every interpretation there is a variable assignment v such that $\neg\phi(v) \vee p(a)$ is satisfied.

Because Δ is quantifier-free, obeys the unique names assumption, and the query $p(a)$ is ground, Herbrand's theorem ensures that $\Delta \models p(a)$ if and only if every Herbrand model of Δ satisfies $p(a)$. Thus, the algorithm above is sound and complete. As for complexity, assuming steps (4) and (6) run in polynomial time, we have an algorithm in the polynomial hierarchy with quantifier prefix $\exists\exists\forall\exists$. That gives us Σ_3^{p} inclusion.

All that remains is showing that steps (4) and (6) run in polynomial time in the size of Δ, which requires showing that the evaluation of ground subsets of Δ over an interpretation takes polynomial time. First, note that the size of every Herbrand model is at most the number of predicates occurring in Δ times the number of object constants occurring in Δ to the kth power, which is clearly polynomial in Δ since k is a constant. Every ground instance $\Delta'[v]$ is linear in Δ, so it too is polynomial in Δ. Satisfaction of a ground instance $\Delta'[v]$ in I (including $=$ checks, which are especially trivial due to the UNA) is polynomial (at worst $|I| * |\Delta'[v]|$).

Proposition 7 ($\text{MON}_k^= \models_E$ inclusion). *Suppose Δ is a finite set of quantifier-free, function-free first-order sentences obeying the unique names assumption where every relation constant has arity at most some constant j, and Δ has at most some constant k variables. $\Delta \models_E p(a)$ is included in Σ_2^{p}.*

Proof. Consider the following alternating time algorithm.

1. (Existential) Guess a subset Δ_0 of Δ.
   // Check satisfiability
   2. (Existential) Guess a Herbrand interpretation I
      3. Validate that $\models_I \Delta_0[v]$
         for every variable assignment v
   // Check entailment
   4. (Universal) For all Herbrand interpretations I
      5. Validate that $\models_I \neg\Delta_0[v] \vee p(a)$
         for some variable assignment v

This algorithm differs from the previous one in that the search through variable assignments can be folded into the polynomial-time validation steps. To see why, notice that because there are at most k variables in Δ, the number of variable assignments is $|objs[\Delta]|^k$, which is polynomial in Δ. Thus, both validation steps (3) and (5) run in polynomial time. The remainder of the proof is the same as the previous one.

Corollary 3 ($\textbf{MON}_*$ and $\textbf{MON}_k$ $\models_E$ inclusion). *Suppose Δ is a finite set of quantifier-free, function-free, equality-free monadic first-order sentences obeying the unique-names assumption. $\Delta \models_E p(a)$ is included in Σ_3^{p}, and if the number of variables is bounded by a constant k then $\Delta \models_E p(a)$ is included in Σ_2^{p}.*

Proof. Trivially MON_* can be embedded in $\text{MON}_*^=$; likewise, MON_k can be embedded in $\text{MON}_k^=$. Since $\text{MON}_*^=$ is included in Σ_3^{p}, so is MON_*; likewise, since $\text{MON}_k^=$ is included in Σ_2^{p}, so is MON_k.

Theorem 3 ($\models_E$ complexity). *The complexity bounds for existential entailment for MON_*, MON_k, clausal MON_1, $\text{MON}_*^=$, $\text{MON}_k^=$, and clausal $\text{MON}_1^=$ are given below.*

	MON_*	MON_k	*clausal* MON_1	$\text{MON}_*^=$	$\text{MON}_k^=$	*clausal* $\text{MON}_1^=$
Hardness	Π_2^{p}	Σ_1^{p},Π_1^{p}	Π_1^{p}	Π_2^{p}	Σ_1^{p},Π_1^{p}	Π_1^{p}
Inclusion	Σ_3^{p}	Σ_2^{p}	Σ_2^{p}	Σ_3^{p}	Σ_2^{p}	Σ_2^{p}

Proof. The previous propositions and corollaries explicitly prove each of the bounds except for the inclusion results for clausal $\text{MON}_1^{[=]}$. Since clausal $\text{MON}_1^{[=]}$ is a special case of $\text{MON}_k^{[=]}$, the inclusion bounds for $\text{MON}_k^{[=]}$ are also inclusion bounds for clausal $\text{MON}_1^{[=]}$.

Strict Entailment The results for existential entailment are useful because they help us with similar bounds for strict entailment. In particular, existential entailment can be simulated by strict entailment, ensuring that strict entailment is at least as hard as existential entailment.

Proposition 8 ($\textbf{MON}_*^=$ $\models_E^\Omega$ inclusion). *Suppose Δ is a finite set of quantifier-free, function-free first-order sentences obeying the unique names assumption where every relation constant has arity at most some constant k. Further suppose Γ is a set of ground sentences. $\Gamma \models_E^\Delta p(a)$ is included in Σ_3^{p}.*

Proof. Consider the following simple variation on the algorithm for existential entailment.

1. (Existential) Guess a subset Γ_0 of Γ.
    // Check satisfiability
    2. (Existential) Guess a Herbrand interpretation I
        3. (Universal) For all variable assignments v
            4. Validate that $\models_I \Gamma_0 \cup \Delta[v]$
    // Check entailment
    4. (Universal) For all Herbrand interpretations I
        5. (Existential) Guess a variable assignment v.
            6. Validate that $\models_I \neg(\Gamma_0 \cup \Delta)[v] \vee p(a)$

The soundness, completeness, and complexity arguments hold just as before, ensuring the same Σ_3^{p} upper bound.

Proposition 9 ($\mathbf{MON}_k^{=}$ $\models_E^{\Omega}$ inclusion). *Suppose Δ is a finite set of quantifier-free, function-free first-order sentences obeying the unique names assumption where every relation constant has arity at most some constant j, and Δ has at most some constant k number of variables. Further suppose Γ is a set of ground sentences. $\Gamma \models_E^{\Delta} p(a)$ is included in Σ_2^{p}.*

Proof. The following simple variation on the previous algorithm leverages the fact that there are only a polynomial number of variable assignments and therefore only a polynomial number of instances of Δ; thus, the search through variable assignments can be folded into the validation routines, eliminating one level of alternating time splitting. See proof for $\mathrm{MON}_k^{=}$ and existential entailment.

1. (Existential) Guess a subset Γ_0 of Γ.
    // Check satisfiability
    2. (Existential) Guess a Herbrand interpretation I
        3. Validate that $\models_I \Gamma_0 \cup \Delta[v]$
            for every variable assignment v.
    // Check entailment
    4. (Universal) For all Herbrand interpretations I
        5. Validate that $\models_I \neg(\Gamma_0 \cup \Delta)[v] \vee p(a)$
            for some variable assignment v.

Corollary 4 ($\mathbf{MON}_*$ and $\mathbf{MON}_k$ $\models_E^{\Omega}$ inclusion). *Suppose Δ is a finite set of quantifier-free, function-free, equality-free monadic first-order sentences obeying the unique-names assumption. Further suppose Γ is a set of ground sentences. $\Gamma \models_E^{\Delta} p(a)$ is included in Σ_3^{p}, and if the number of variables is bounded by a constant k then $\Gamma \models_E^{\Delta} p(a)$ is included in Σ_2^{p}.*

Proof. Trivially MON_* can be embedded in $\mathrm{MON}_*^{=}$; likewise, MON_k can be embedded in $\mathrm{MON}_k^{=}$. Since $\mathrm{MON}_*^{=}$ is included in Σ_3^{p}, so is MON_*; likewise, since $\mathrm{MON}_k^{=}$ is included in Σ_2^{p}, so is MON_k.

Proposition 10 ($\models_E^{\Omega}$ is $\models_E$-hard). *Suppose Δ is a finite set of first-order sentences. Further suppose Γ is a set of ground atoms. $\Gamma \models_E^{\Delta} p(a)$ is as hard as $\Delta \models_E p(a)$.*

Proof. Consider an existential entailment problem $\Sigma \models_E p(a)$, where Σ is a finite set of first-order sentences. We show how to construct a Δ and a Γ such that $\Sigma \models_E p(a)$ if and only if $\Gamma \models_E^{\Delta} p(a)$.

Δ is constructed as follows. For each sentence σ in Σ, create the sentence $q(a) \Rightarrow \sigma$, where q is a new predicate. (Notice that any universal quantifiers implicitly applied to σ are still applied in the same way to $q(a) \Rightarrow \sigma$.) Γ is then the set of all $q(a)$ where q was introduced during the construction of Δ. Now we must show that the strict entailment query $\Gamma \models_E^{\Delta} p(a)$ exactly when $\Sigma \models_E p(a)$.

($\Leftarrow$) If $\Sigma \models_E p(a)$, then there is a satisfiable $\Sigma_0 \subseteq \Sigma$ that entails $p(a)$. Consider the sentences Σ_0' constructed from elements of Σ_0 and placed into Δ. Let Q denote the set of all $q(a)$ in the antecedents of the Σ_0' implications. Since Σ_0 is satisfiable and all the q are fresh, we can extend any model satisfying Σ_0 to additionally satisfy all of Q by forcing $q(a)$ true for exactly those fresh q occuring in Q. Since that model satisfies all of Σ_0, it satisfies all of Σ_0' because the consequent of every rule evaluates to true; moreover, since the model also satisfies Q, it satisfies $Q \cup \Sigma_0'$. Since the model fails to satisfy the additional fresh q's in Δ, it also satisfies $Q \cup \Delta$. By construction, Q is a subset of Γ, and after $|\Sigma_0|$ applications of modus ponens, $Q \cup \Delta$ produces each sentence in Σ_0, which is known to entail $p(a)$. $Q \cup \Delta$ is satisfiable and entails $p(a)$; thus, $\Gamma \models_E^{\Delta} p(a)$.

($\Rightarrow$) Suppose $\Gamma \models_E^{\Delta} p(a)$. Then there is a $Q \subseteq \Gamma$ that is consistent with Δ such that $Q \cup \Delta$ entails $p(a)$. Since all sentences in Δ are of the form $q(a) \Rightarrow \phi$, and all the q are fresh, Δ alone cannot entail $p(a)$. (The countermodel makes $p(a)$ and all the $q(a)$ false, satisfying Δ but not $p(a)$.) So Q must be nonempty. Consider just those sentences S from Δ whose antecedents belong to Q. We claim that necessarily $Q \cup S$ is both satisfiable (since $Q \cup \Delta$ was) and entails $p(a)$.

For the purpose of contradiction, suppose some of those remaining sentences R in Δ were necessary for entailing $p(a)$—that there were a model M of $Q \cup S$ that did not satisfy $p(a)$ but that all models of $Q \cup S \cup R$ satisfied $p(a)$. For M to be a model of $Q \cup S$ but not $Q \cup S \cup R$, M must not satisfy R; thus, M must satisfy all the $q(a)$ in the antecedents of R. But since all the qs are fresh, there is another model M' that is the same as M (especially in that it satisfies $p(a)$) except it fails to satisfy the $q(a)$ in R, and hence satisfies all of R. M' therefore satisfies $Q \cup S \cup R$ but fails to satisfy $p(a)$, by construction, contradicting the original premise. Hence, there can be no such R.

$Q \cup S$ is therefore satisfiable and entails $p(a)$. The models of $Q \cup S$ must all (a) satisfy $p(a)$ and (b) satisfy all the consequents S' of S (since they satisfy all of Q). Because all the q occurring in Q are fresh, the reducts of the models satisfying $Q \cup S$ to the original language is axiomatized by S'; hence, S' is satisfiable and entails $p(a)$. Since S' is a subset of Σ, $\Sigma \models_E p(a)$.

Corollary 5. *Suppose $\mathcal{M}$ is one of* MON_*, MON_k, $\text{MON}_*^=$, *or* $\text{MON}_k^=$. *If $\mathcal{M}$ is* C*-hard for existential entailment, $\mathcal{M}$ is* C*-hard for strict entailment.*

Proof. The proof above introduces no additional variables and is ambivalent toward $=$; thus, the embedding of existential entailment for MON into strict entailment produces axioms in MON. Thus, if existential entailment is C-hard, then the proof above when instantiated with MON demonstrates that strict entailment is also C-hard.

Theorem 4 ($\models_E^{\Omega}$ complexity). *The complexity bounds for strict existential entailment for* MON$_*$, MON$_k$, *clausal* MON$_1$, MON$_*^{=}$, MON$_k^{=}$, *and clausal* MON$_1^{=}$ *are given below. (They are the same as for $\models_E$.)*

	MON$_*$	MON$_k$	*clausal* MON$_1$	MON$_*^{=}$	MON$_k^{=}$	*clausal* MON$_1^{=}$
Hardness	Π_2^{p}	Σ_1^{p},Π_1^{p}	Π_1^{p}	Π_2^{p}	Σ_1^{p},Π_1^{p}	Π_1^{p}
Inclusion	Σ_3^{p}	Σ_2^{p}	Σ_2^{p}	Σ_3^{p}	Σ_2^{p}	Σ_2^{p}

Proof. The previous propositions and corollaries explicitly prove each of the bounds except for the inclusion results for clausal MON$_1^{[=]}$. Since clausal MON$_1^{[=]}$ is a special case of MON$_k^{[=]}$, the inclusion bounds for MON$_k^{[=]}$ are also inclusion bounds for clausal MON$_1^{[=]}$.

Finally, we show that if the sentences have fixed size, strict entailment for languages without = is in P.

Proposition 11 (Constant-size MON$_*$ $\models_E^{\Delta}$ inclusion). *Suppose Δ is a fixed-size set of quantifier-free, function-free, equality-free first-order sentences. Further suppose Γ is a set of ground atoms. $\Gamma \models_E^{\Delta} p(a)$ is included in* AC^0.

Proof. The compilation algorithm for strict entailment introduced in this paper (Algorithm 1) constructs a non-recursive set of database queries that utilize consistency checks. Those consistency checks can be written as non-recursive database queries as shown in [20]. Both sets of database queries are dependent on Δ (but not Γ), and since Δ is of fixed size, so too are the database queries that are generated.

Because Γ is a set of ground atoms, it can be interpreted as a database or as a set of statements in classical logic, the difference being whether or not the closed-world assumption is applied. The database queries discussed above soundly and completely compute strict entailment when evaluated over Γ interpreted as a database. (See proofs in the next section.) Since database evaluation is well-known to be AC^0 [26] when the size of queries are constant, so too is strict entailment in AC^0.

A.2 Algorithms

Theorem 5 (Soundness and Completeness). *The algorithm* IMPLCOMPILE *is a web form constraint compiler for* MON *ontologies.*

Proof. (Adapted and simplified from [20].) Suppose Δ is an ontology and Λ is a payload. (Soundness) Suppose $\models_\Lambda \phi_f^{\pm}(a)$. Then there is some disjunct $\psi(x) \wedge \kappa$ of $\phi_f^{\pm}(a)$ (where κ is the consistency check) such that $\models_\Lambda \psi(a) \wedge \kappa$ is true. By construction, there is therefore an implication of the form $\pm f(x) \Leftarrow \psi(x)$ entailed by Δ. Because $\phi_f^{\pm}$ and therefore $\psi(x)$ contains no negation, its satisfaction is witnessed by a set of positive atoms $\Lambda_0 \subseteq \Lambda$. Since $\models_{\Lambda_0} \psi(a)$, and ψ contains no negation, $\Lambda_0 \models \psi(a)$ and therefore $\Lambda_0 \cup \Delta$ entails $\pm f(a)$. Since Λ_0 also satisfies the consistency check κ, Λ_0 is consistent with Δ; ergo the algorithm is sound: $\Lambda \models_E^{\Delta} \pm f(a)$.

(Completeness) Suppose $\Lambda \models_E^{\Delta} \pm f(a)$. Then there is a $\Lambda_0 \subseteq \Lambda$ that together with Δ entails $\pm f(a)$ and is consistent with Δ. Since Δ is quantifier-free, resolution's completeness ensures there is a single clause c in the resolution closure of Δ that together

with Λ_0 entails $\pm f(a)$; moreover, the proof must consist of only unit resolutions between c and Λ_0. (To construct such a clause, take any constructive resolution proof of $\pm f(a)$ from the clausal form of $\Delta \cup \Lambda_0$ and re-order the resolutions so that all the unit resolutions involving Λ_0 occur last. Choose the final clause in the proof not belonging to Λ_0 for c.) Since Λ and therefore Λ_0 consists of only positive literals, c must take the form $\pm f(t) \vee \neg a_1(t_1) \vee \ldots \vee \neg a_n(t_1)$, *i.e.*, all literals except for $\pm f(t)$ are negative, and t is either a variable or a. When written as an implication with $\pm f(x)$ in the head, the body is $x = t \wedge a_1(t_1) \wedge \ldots \wedge a_n(t_n)$. By construction each such conjunction is a disjunct of $\phi_f^{\pm}$. We claim that the resolution proof of $\pm f(a)$ from $\pm f(t) \vee \neg a_1(t_1) \vee \ldots \vee \neg a_n(t_n)$ and Λ_0 guarantees that

$$\models_{\Lambda_0} x = a \wedge a_1(t_1) \wedge \ldots \wedge a_n(t_n) \wedge \kappa,$$

where κ is the consistency check. Since one of its disjuncts is satisfied by Λ_0 so is $\phi_f^{\pm}(a)$ (*i.e.*, $\models_{\Lambda_0} \phi_f^{\pm}(a)$); moreover, since $\Lambda \supseteq \Lambda_0$ and $\phi_f^{\pm}(a)$ is positive (*i.e.*, contains no negation), $\models_{\Lambda} \phi_f^{\pm}(a)$.

For the claim, recall that resolution is generatively complete up to subsumption, which ensures one of the unit clauses $\pm f(a)$ or $\pm f(x)$ belongs to the closure of $\pm f(t) \vee \neg a_1(t_1) \vee \ldots \vee \neg a_n(t_1)$ and Λ_0. Suppose σ is the unifier resolution employs to resolve away all the a_i literals. Certainly then $\Lambda_0 \cup \{\neg a_1(t_1) \vee \ldots \vee \neg a_n(t_n)\}\sigma$ is inconsistent, and hence $\Lambda_0 \models a_1(t_1)\sigma \wedge \ldots \wedge a_n(t_n)\sigma$. (We need not introduce quantifiers when negating because $a_i(t_i)\sigma$ is necessarily ground—σ was constructed by unifying $a_i(t_i)$ with a ground atom from Λ_0 for all i.) Certainly since all models of Λ_0 satisfy $a_1(t_1)\sigma \wedge \ldots \wedge a_n(t_n)\sigma$ so does the minimal one; thus, $\models_{\Lambda_0} a_1(t_1)\sigma \wedge \ldots \wedge a_n(t_n)\sigma$.

Now notice that if $\pm f(x)$ is the unit clause appearing in the resolution closure then x does not appear in any of the $a_i(t_i)$ (or else it would have been assigned an object constant in σ); thus, we can extend σ to include x/a without changing the resolution proof. If, on the other hand, $\pm f(a)$ is the final unit clause, $f(t)$ from the original clause is either $f(a)$ or $f(x)$ where x is bound to a by σ. In any case, including t/a in σ does not change the resolution proof, and hence $\models_{\Lambda_0} t = a \wedge a_1(t_1)\sigma \wedge \ldots \wedge a_n(t_n)\sigma$.

Finally, consider the consistency check κ:

$$consistent^{\Delta}_{a_1(t_1) \wedge \ldots \wedge a_n(t_n)}(\bar{x}).$$

Because Λ_0 is by assumption consistent with Δ, and all $a_i(t_i)\sigma$ belong to Λ_0, by definition $consistent^{\Delta}_{a_1(t_1) \wedge \ldots \wedge a_n(t_n)}(\bar{x})\sigma$ is true. This completes the proof of completeness. □

Proposition 12 (Resolution Complexity). *The output complexity of resolution for* MON *is* EXPSPACE*-hard and included in* 2EXPSPACE*. When the premises are in clausal form, contain one variable, and include no object constants, the output complexity is* EXPSPACE*-complete.*

Proof. For inclusion in 2EXPSPACE, simply count the number of monadic clauses. Consider first the case without object constants. For the purpose of counting, each clause of n variables is comprised of n distinct propositional clauses. For example, $p(x) \vee q(x) \vee p(y) \vee \neg q(y)$ corresponds to two propositional clauses: $p \vee q$ and $p \vee \neg q$.

The number of propositional clauses with n propositions is 3^n; thus, the number of monadic clauses without object constants is 2^{3^n}, each of which is at most $n * 3^n$ in size. When we add object constants, each one allows for another propositional clause attached to each monadic clause; thus, there is an additional factor of $(3^n)^o = 3^{no}$ with o object constants and an additional size of no. The total number of monadic clauses is therefore $2^{3^n} * 3^{no}$, each of which is no larger than $n * 3^n + no$.

The EXPSPACE hardness result comes from the fact that in propositional clausal logic, proofs can be exponentially long. All steps in such proofs belong to the resolution closure and hence the size of the resolution closure in propositional logic (and hence clausal monadic logic) is singly exponential, *i.e.*, EXPSPACE-hard.

For the restriction to clausal form with one variable and no object constants, it is easy to see that all clauses in the closure have a single variable., The number of such clauses is at most 2^n, each of which has size no larger than n; hence, the output size is at most $n2^n$, ensuring inclusion in EXPSPACE.

□

Proposition 13 (Strict Implication and Inconsistency Implementations). *For any class of* MON *for which resolution's output complexity is* EXPSPACE*, without compression* PLATO *produces a time-optimal implementation of strict implication and inconsistency detection unless* $\mathrm{P} = \mathrm{NP}$.

Proof. Because of the coNP-hardness results, unless $\mathrm{P} = \mathrm{coNP}$, the optimal algorithm for strict entailment/inconsistency detection runs in EXPTIME. If resolution's output complexity is EXPSPACE, there are at most single exponentially many clauses; hence, PLATO's output consists of (disjunctions over) at most single exponentially many conjunctions of positive atoms. We claim that evaluating each conjunction (each of which includes a consistency check) requires at most single exponential time in the size of the payload Λ and the ontology; since the evaluation of each conjunction is independent of the evaluation of every other conjunction, evaluating all conjunctions and therefore all of PLATO's output requires single exponential time, giving the optimality result for time.

To see that evaluating an arbitrary monadic conjunction of atoms requires single exponential time, we show that evaluating the portion of the conjunction whose arguments are variables, the portion whose arguments are object constants, and the portion corresponding to the consistency check each require single exponential time. Then, because the evaluation of each portion is independent of the other portions, we conclude that the total evaluation time is single exponential.

We begin with the portion of the conjunction where all arguments are variables:

$$p_1^1(x_1) \wedge \ldots \wedge p_{n_1}^1(x_1) \wedge \ldots \wedge p_1^m(x_m) \wedge \ldots \wedge p_{n_m}^m(x_m).$$

Because all of the predicates are monadic, the evaluation of the conjuncts with x_i is independent of the evaluation of the conjuncts with x_j for $i \neq j$. Evaluating the portion dependent on x_i requires enumerating all values in p_1^i and then looking up each of those values in $p_2^i, \ldots, p_n^i$. The number of values in p_1^i is proportional to the payload Λ, and (with no indexing) the cost of lookup is $|\Lambda|$. Thus, the cost of evaluation for a single variable x_i is $O(|\Lambda|^{n_i})$, which is $O(|\Lambda|^p)$ for p predicates. There are at most

3^p variables in each conjunction (since there are at most 3^p propositional clauses over p propositions–see previous proof), but evaluation of each variable is independent of all other variables. Thus, the cost of evaluating the portion of each conjunction with variables for arguments is $O(3^p|\Lambda|^p)$. Now we turn to the evaluation cost of the portion of the conjunction where arguments are object constants. Evaluating a ground conjunct without indexing costs $|\Lambda|$, and since there are at most po such conjuncts, the total cost is $O(po|\Lambda|)$.

Finally, we show that the consistency check requires single exponential time in $|\Lambda|$. Recall that our implementation caches the minimal inconsistent subsets of Λ. Evaluating a consistency check then amounts to checking if a given subset includes any of those minimal inconsistent subsets. The maximal number of inconsistent subsets is bounded above by $2^{|\Lambda|}$, and checking if one set includes another is the product of the set sizes. The largest minimal inconsistent subset is $|\Lambda|$; the largest consistency check is the maximal number of conjuncts in any clause: $p3^p + po$ from the previous proof. Checking consistency therefore costs no more than $2^{|\Lambda|}|\Lambda|(p3^p+po)$, which is singly exponential.

All told, the cost for evaluating a monadic conjunction is $O(3^p|\Lambda|^p + po|\Lambda| + 2^{|\Lambda|}|\Lambda|(p3^p + po))$, which is single exponential in Λ and the ontology. This completes the proof.

□