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, 32]. 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, 30, 33, 34], computer security [8, 31] formal methods [9], programming languages [5, 10, 17–19, 26], databases [7, 14], artificial intelligence [20, 22, 23], and configuration management [1, 27, 28, 32]. 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, 32] allow errors and utilize omni-directional versions of paraconsistent implication; however, [32] 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.

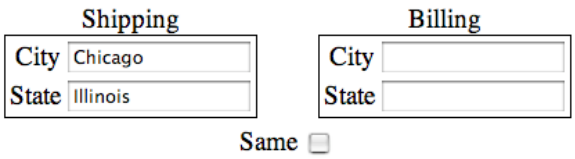


Fig. 1. Example web form

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.

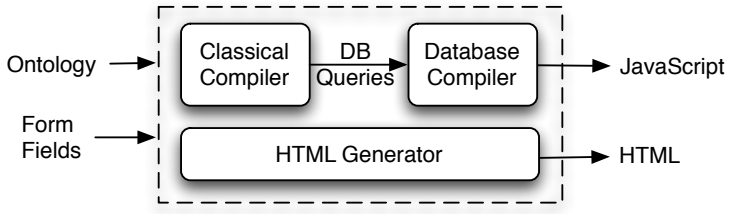


Fig. 2. PLATO architecture.

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* $\mathrm{P} = \mathrm{NP}$.

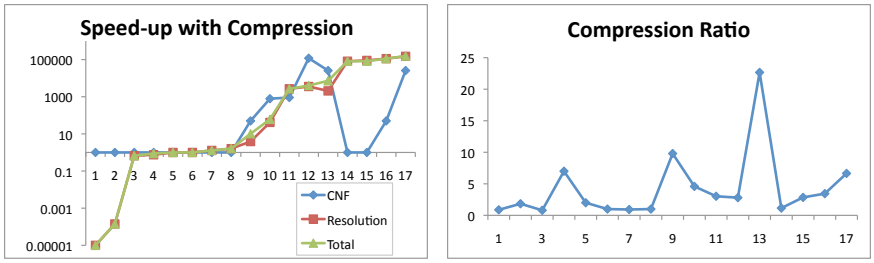
6.2 Empirical

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, 29]), 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.

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. 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)
27. 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)
28. 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)
29. 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)
30. 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)
31. 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)
32. 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)
33. 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)
34. 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)