Injecting the How into the What: Investigating a Finite Classical Logic

Timothy L. Hinrichs
University of Chicago
tlh@uchicago.edu

Michael R. Genesereth
Stanford University
genesereth@stanford.edu

Abstract

Computer scientists routinely design algorithms to efficiently solve problems found in nature. The resulting algorithms encapsulate the original problem as well as extra information about how to solve that problem; thus, nature's original description that says what the problem is has been augmented with information about how to solve it. In this paper, we investigate the automation of this process by concentrating on declarative languages, arguing that certain classes of declarative languages encode more information about how to solve a problem than others, and demonstrating techniques for automatically translating between two languages separated in the what-to-how spectrum.

Introduction

Feigenbaum posits that all computer systems and formal languages can be arranged in a what-to-how spectrum (Feigenbaum 1996). The fewer details a user must provide about how to accomplish a task, the closer the system or language lies to the "what" end of the spectrum. This intuition can be grounded by examining formal languages and, more generally, classes of formal languages.

The major language classes, according to the programming language community, are well known: declarative, functional, and imperative. Declarative languages are commonly believed to lie closer to the "what" end of the spectrum than functional languages, which in turn lie closer to the "what" end than imperative languages. That intuition can be explained by placing language classes on the what-to-how spectrum based on order-relevance: to what extent is ordering important? The less order matters, the closer to the "what" end of the spectrum a language class lies. Ordering is less important in declarative languages than in functional languages, and ordering is less important in functional languages than in imperative languages.

In an imperative language like C++, the order in which statements occur is very important; in addition, the order within statements is important. An assignment $x = y$ in C++ places the contents of the variable y into the variable x and not the other way around. In functional languages, the order in which statements occur is relevant if there are side effects, and even if there are no side effects, order is important within each function definition. The definition $f(x, y) := g(x, y)$ says that in order to compute $f(a, b)$, it is necessary and sufficient to compute $g(a, b)$; however, the expression $g(a, b)$ will never be evaluated by computing $f(a, b)$. In a declarative language, order is even less relevant than in a functional language. The statement $h(x, y) = h(y, x)$ in first-order logic says little about how to compute $h(a, b)$; it simply guarantees that $h(a, b)$ is the same as $h(b, a)$.

Order-relevance can also be used to arrange languages in the what-to-how spectrum within a class. In declarative languages, the order of statements is irrelevant, but for some languages, order within each statement is more important than others. Consider classical logic, where order has little importance, and logic programming, where order has significantly more importance. An implication $a \Leftarrow \neg b$ in propositional logic is equivalent to $b \Leftarrow \neg a$. In classical logic, the impact of both implications must be considered, which is done explicitly in certain reasoning procedures like model elimination (Loveland 1978). In Prolog, however, the user decides which of these statements to include, often to the exclusion of the other. We conclude that classical logic is closer to the "what" end of the spectrum than logic programming languages because order is less relevant in the former than in the latter.

In light of the fact that languages closer to the "how" end of the spectrum say more about how to solve a given problem, it is not surprising that imperative languages are considered more efficient than functional languages, which are in turn considered more efficient than declarative languages. Likewise, we conclude that logic programming languages are more efficient than classical languages, a claim substantiated by the fact that the most industrially successful declarative language to date, SQL, lies far closer to logic programming than classical logic.

In this paper, we report on algorithms for automatically transforming problem descriptions stated in a classical logic into a logic programming language, effectively investigating the automatic injection of "how" into "what". Such transformation algorithms have the potential to solve problems more efficiently than traditional techniques. The logic programming language we have chosen to study seems to be one of the most imperative in the class and can be evaluated by

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

industrial-strength tools: nonrecursive DATALOG with negation. The classical logic we have chosen is function-free first-order logic where quantifiers range over a finite and known set of objects, guaranteeing decidability without sacrificing arbitrary (syntactic) quantification. These languages were chosen to ensure the translation is decidable and to enable off-the-shelf database systems to answer queries about a classical logic.

Translating between these languages is exemplified throughout this paper using the graph coloring problem: given a graph and a set of colors, paint each node in the graph so that no two adjacent nodes are colored the same. Such a problem can be axiomatized in classical logic:

$$\begin{array}{l} color(x,y) \Rightarrow node(x) \wedge hue(y) \\ color(x,y) \wedge adj(x,z) \Rightarrow \neg color(z,y) \\ hue(x) \Leftrightarrow (x = red \vee x = blue) \\ node(x) \Leftrightarrow (x = n_1 \vee x = n_2 \vee x = n_3) \\ adj(x,y) \Leftrightarrow ((x = n_1 \wedge y = n_2) \vee \\ \qquad\qquad (x = n_2 \wedge y = n_3)) \end{array} \tag{1}$$

The problem instance represented here consists of three nodes (n_1, n_2, n_3), connected so that only n_2 is adjacent to n_1 or n_3, and two colors (red and $blue$). Coloring the graph corresponds to finding assignments for the variables x, y, and z so that the following sentence is satisfied.

$$color(n_1,x) \wedge color(n_2,y) \wedge color(n_3,z)$$

The algorithms in this paper transform the problem above into a logic program that enumerates the graph colorings.

$$\begin{array}{l} ans(x,y,z) :- hue(x) \wedge hue(y) \wedge hue(z) \wedge \\ \qquad\qquad x \neq y \wedge y \neq z \\ hue(red) \\ hue(blue) \end{array}$$

First-order model builders solve the graph coloring problem using the first formulation. Preliminary tests showed that for the graph coloring instance on page three of (McCarthy 1982), a simple DATALOG implementation was at least 40 times faster than several well-known first-order model builders: Paradox (Claessen & Sorensson 2003), Mace4 (McCune 2003), and FMDarwin (Baumgartner *et al.* 2007).[1]

In the remainder of the paper, the first section defines the syntax and semantics of the languages studied. The next section formally defines the problem. The next two sections consider two different cases for the transformation: where the original theory is complete and where the original theory is incomplete. The next section discusses some experimental results, and the rest of the paper consists of related work, conclusions, and future work. Details not presented for lack of space can be found in (Hinrichs 2007).

Preliminaries

The two languages studied in this paper are well-known; we use common conventions for their syntax and semantics.

[1]In these tests, the equality checks in the DATALOG version were placed as early in the rule as possible, a simple and easy to implement conjunct-ordering scheme. Without reordering, i.e. placing all the hues first, DATALOG was only four times faster.

The source language for the translation is first-order logic with equality and the following three restrictions: no function constants, a domain closure assumption (DCA), and a unique names assumption (UNA). In this logic, the objects in the universe of every model are exactly the object constants in the language. We call this logic Finite Herbrand Logic (FHL) because the only models of interest are the finite Herbrand models.

A vocabulary is a countable set of variables together with a finite collection of object constants and predicates with associated arities. A term is a variable, denoted by a letter from the end of the alphabet, or an object constant, denoted by a letter from the beginning of the alphabet. An atom is a predicate of arity n applied to n terms, and a literal is an atom or $\neg$ applied to an atom. Logical sentences are defined inductively. Atoms are sentences; if ϕ and ψ are sentences and x is a variable then all of the following are sentences: $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$, $(\phi \Leftarrow \psi)$, $(\phi \Leftrightarrow \psi)$, $(\forall x.\phi)$, and $(\exists x.\phi)$. Sentences without variables are ground. Variables not captured by a quantifier are free. Sentences with free variables are open sentences, and sentences without free variables are closed sentences. The set of all sentences for a given vocabulary is the language for that vocabulary.

We use standard metalevel notation. A lower-case Greek letter denotes a single sentence, and an upper-case Greek letter denotes a finite set of sentences. Instead of writing out n terms as $t_1, \ldots, t_n$, we may write $\bar{t}$.

The definitions for a model and for satisfaction are standard but are simplified to take advantage of the UNA and DCA. A model in FHL is a set of ground atoms from the language. A model satisfies all the ground atoms included in the set. Satisfaction is defined as usual for the Boolean connectives applied to ground sentences. Satisfaction of non-ground sentences reduces to satisfaction of ground sentences. An open sentence $\phi(\bar{x})$ with free variables $\bar{x}$ is equivalent to $\forall\bar{x}.\phi(\bar{x})$. $\forall x.\phi(x)$ is satisfied exactly when $\phi(a)$ is satisfied for every object constant a. $\exists x.\phi(x)$ is satisfied exactly when $\phi(a)$ is satisfied for some object a.

A set of sentences is satisfiable (or consistent) when there is at least one model that satisfies it. In this paper, all sentence sets are satisfiable unless stated otherwise. A set of sentences is incomplete whenever there is more than one model that satisfies it; otherwise, the sentences are complete. Logical entailment is defined as usual: $\Delta \models \phi$ if and only if every model that satisfies Δ also satisfies ϕ. A set of sentences closed under logical entailment is a theory. For complete, satisfiable theories, the notion of consistency and entailment coincide. In this case, we may write that a sentence is true in a theory to mean that the theory is complete, and the sentence is consistent with it (entailed by it).

This paper introduces algorithms that translate FHL queries into queries about nonrecursive DATALOG with negation, denoted NR-DATALOG$^\neg$. Every sentence in NR-DATALOG$^\neg$ is an implication, where $:-$ replaces $\Leftarrow$.

$$h :- b_1 \wedge \ldots \wedge b_n$$

h, the head of the rule, is an atom. Each b_i is a literal, and the conjunction of b_is is the body of the rule. The body may be empty, but the head may not. Every rule must be

safe: every variable that occurs in the head or in a negative literal must occur in a positive literal in the body. Finally, a rule set (NR-DATALOG$^\neg$ program) must be nonrecursive. If predicate p appears in the head of a rule and q occurs in the body, p depends upon q. A nonrecursive rule set guarantees that if p transitively depends on q then q never transitively depends on p.

A model for NR-DATALOG$^\neg$ is the same as that for FHL: a set of ground atoms; however, in contrast to FHL where sentences may be satisfied by more than one model, a set of NR-DATALOG$^\neg$ sentences is always satisfied by exactly one model. Without negation, this model is the smallest one (ordered by subset) that satisfies the sentences under the FHL definition of satisfaction. With negation, the stratified semantics (Ullman 1989) chooses one model out of all those that satisfy the sentences under the FHL definition. If the model assigned to the NR-DATALOG$^\neg$ sentences Δ is M then $\Delta \models \phi$ if and only if M satisfies ϕ in FHL.

It is noteworthy that satisfiability for FHL is NEXPTIME-complete (Hinrichs 2007), and satisfiability/entailment for NR-DATALOG$^\neg$ is PSPACE-complete, e.g. (Vorobyov & Voronkov 1998).

Problem Statement

When translating FHL to NR-DATALOG$^\neg$, the most important thing to understand is the difference in the semantics of the two languages. FHL sentences can be satisfied by more than one model, but NR-DATALOG$^\neg$ sentence sets are always satisfied by exactly one model. Because the translation involves turning a theory with more than one model (an incomplete theory) into a theory with exactly one model (a complete theory), the translation is a form of theory-completion.

Unlike other forms of theory-completion, e.g. the closed-world assumption (Reiter 1978), predicate completion (Lloyd 1984), and circumscription (McCarthy 1988), where the purpose is to minimize the theory, the theory-completion investigated here is performed solely for the purpose of efficiency. The completion procedure must preserve some or all of the logical consequences of the theory. In this paper, we introduce a theory-completion algorithm that preserves whether or not a given sentence is logically entailed. Equivalently, the algorithm preserves the satisfiability of the given sentence. We call such a theory-completion algorithm satisfaction-preserving.

Definition 1 (Satisfaction-Preserving Theory Completion). *Given a finite set of sentences Δ and a sentence $\phi(\bar{x})$, construct a complete theory Δ' and a query $\phi'(\bar{x})$ so that for every tuple of object constants $\bar{a}$,*

$$\Delta \cup \{\phi(\bar{a})\} \text{ is satisfiable} \\ \text{if and only if} \\ \Delta' \cup \{\phi'(\bar{a})\} \text{ is satisfiable.}$$

Satisfaction-preserving theory-completion makes no commitment as to the source or target languages. The completed theory can be written in the same language as the original or not. The transformation from FHL to NR-DATALOG$^\neg$ clearly requires the original and completed theories to be written in different languages; however, viewing this rewrite procedure as a single operation conflates two separable transformations: the completion of the theory and the translation of that complete theory into NR-DATALOG$^\neg$. In this paper, we perform the transformation in two steps.

1. Perform theory-completion entirely within FHL.
2. Translate the complete FHL theory into NR-DATALOG$^\neg$.

For example, consider the graph coloring problem described earlier with three nodes (n_1, n_2, and n_3), connected so that only n_2 is adjacent to n_1 or n_3, and two colors (red and $blue$). A solution consists of a variable assignment that satisfies the following sentence together with Sentence Set 1.

$$color(n_1, x) \wedge color(n_2, y) \wedge color(n_3, z). \tag{2}$$

The first step in the translation to NR-DATALOG$^\neg$ completes Sentence Set 1 with respect to the sentence above, producing the following FHL sentences and the query $ans(x, y, z)$.

$$\begin{aligned} ans(x, y, z) &\Leftrightarrow (hue(x) \wedge hue(y) \wedge hue(z) \wedge \\ &\qquad x \neq y \wedge y \neq z) \\ hue(x) &\Leftrightarrow (x = red \vee x = blue) \end{aligned}$$

Notice that $ans(a, b, c)$ is satisfied (entailed) by these sentences if and only if $\{x/a, y/b, z/c\}$ is a satisfying assignment to Sentence 2, ensuring the resulting theory and query is a satisfaction-preserving completion

The second step transforms the resulting, complete theory into NR-DATALOG$^\neg$.

$$\begin{aligned} & ans(x, y, z) :- hue(x) \wedge hue(y) \wedge hue(z) \wedge \\ & \qquad x \neq y \wedge y \neq z \\ & hue(red) \\ & hue(blue) \end{aligned} \tag{3}$$

An important observation is that the theory-completion step can be skipped when the FHL theory is already complete; thus complete FHL theories are handled differently than incomplete FHL theories.

The Complete Case

Complete, satisfiable FHL theories correspond naturally to NR-DATALOG$^\neg$: every such theory is logically equivalent to some NR-DATALOG$^\neg$ program. The difficult part of the translation from a complete FHL theory to NR-DATALOG$^\neg$ is efficiently constructing the right program.

It turns out that the well-known Lloyd-Topor algorithm (Lloyd 1984; Lloyd & Topor 1984) partially solves this problem. It transforms complete theories of a special form into equivalent NR-DATALOG$^\neg$ programs in polynomial time. All that remains is finding an algorithm that translates an arbitrary complete theory into an equivalent theory written in that special form.

The special form required by Lloyd-Topor is itself very similar to the syntax of NR-DATALOG$^\neg$. Lloyd-Topor expects as input a nonrecursive set of biconditionals. A biconditional is a sentence of the form $p(\bar{x}) \Leftrightarrow \phi(\bar{x})$. We say that the definition for predicate p depends upon all of the

predicates that occur in the sentence $\phi(\bar{x})$. A set of biconditionals is nonrecursive if whenever p transitively depends on q, q does not transitively depend on p.

Definition 2 (Nonrecursive biconditional definitions). *A biconditional definition for predicate p is a sentence of the form $p(\bar{x}) \Leftrightarrow \phi(\bar{x})$, where $\bar{x}$ is a nonrepeating sequence of variables, and $\phi(\bar{x})$ is a sentence with free variables $\bar{x}$. We refer to $p(\bar{x})$ as the head and $\phi(\bar{x})$ as the body. A set of biconditional definitions Δ is nonrecursive if and only if the graph $\langle V, E\rangle$ is acyclic.*

V : the set of predicates that occur in Δ

E : $\langle q, r\rangle$ is a directed edge if and only if there is a biconditional in Δ with q in the head and r in the body.

For example, the set of biconditionals seen earlier is nonrecursive and constitutes the input to Lloyd-Topor for the map coloring problem.

$$\begin{aligned} ans(x,y,z) &\Leftrightarrow (hue(x) \wedge hue(y) \wedge hue(z) \wedge \\ &\qquad x \neq y \wedge y \neq z) \\ hue(x) &\Leftrightarrow (x = red \vee x = blue) \end{aligned} \tag{4}$$

It is easy to construct a sound and complete algorithm that translates a complete FHL theory into a nonrecursive set of biconditionals, but it is difficult to make such an algorithm efficient. Here we discuss an incomplete but low-order polynomial algorithm for performing the translation. (Incompleteness in this case means that some complete theories are not transformed into nonrecursive biconditionals.)

Conceptually the algorithm can be broken down into two operations: selection and confirmation. The selection operation examines the given FHL sentences to find a subset that is likely to entail a biconditional definition, and the confirmation operation determines whether or not the selected sentences actually entail such a definition. These two operations are interleaved both to make the selection operation simpler and to guarantee that if definitions are found for every predicate then the resulting set of definitions is nonrecursive.

The selection operation is based on the following question. Suppose the FHL sentence set were originally written as nonrecursive, biconditional definitions. Further suppose that each definition were then rewritten independently of all the others without introducing any additional predicates while preserving logical equivalence. For each predicate p, how do we find all those sentences that were produced from the biconditional definition for p?

For example, the clauses in Figure 1 are logically equivalent to Sentence Set 4 and were generated under the conditions above. How does one determine which clause came from which definition?

Because the original biconditionals were nonrecursive, there must be at least one of them where the only predicate that occurs in the body is =. Since no predicates were added when that biconditional was rewritten, all of the sentences that were derived from it must mention exactly one predicate besides equality. Thus, the first selection step finds all those sentences ψ such that the only predicates that occur in ψ are = and some predicate p.

$ans(x,y,z) \vee \neg hue(x) \vee \neg hue(y) \vee \neg hue(z) \vee$
$\quad x = y \vee y = z$
$\neg ans(x,y,z) \vee hue(x)$
$\neg ans(x,y,z) \vee hue(y)$
$\neg ans(x,y,z) \vee hue(z)$
$\neg ans(x,y,z) \vee x \neq y$
$\neg ans(x,y,z) \vee y \neq z$
$hue(red)$
$hue(blue)$
$\neg hue(x) \vee x = red \vee x = blue$

Figure 1: Clauses equivalent to Sentence Set 4

After a sentence set has been selected, the confirmation operation checks whether or not those sentences entail a biconditional definition. This is a metalevel theorem-proving operation. Given a set of sentences Γ and a predicate p, find a sentence $\phi(\bar{x})$ such that

$$\Gamma \models p(\bar{x}) \Leftrightarrow \phi(\bar{x}).$$

We will refer to such an algorithm as CONFIRM. Our implementation is sound but incomplete and runs in time polynomial in the size of the given sentences by performing some very simple normalization and syntactic equality checks.

If the confirmation operation succeeds, the algorithm recurses on the sentences not selected, this time looking for sentences that mention a single predicate besides $=$ and the predicate just found to have a biconditional definition. If the selected sentences do not entail a biconditional, the algorithm selects a different predicate q and sentences mentioning only q and =, and the process repeats. At recursion depth i, definitions for predicates $p_1, \ldots, p_i$ have been found and the sentences that entail those definitions have been removed. The algorithm selects sentences with a single predicate besides $\{p_1, \ldots, p_i, =\}$ and checks whether or not they entail a biconditional. The recursion stops when either the algorithm has found a biconditional for every predicate, or there is no selection of the sentences remaining that entail a biconditional.

This algorithm guarantees that the resulting biconditionals are nonrecursive. Intuitively, each time predicate p is found to have a definition, any of the predicates without definitions are allowed to depend upon p because (inductively) the definition of p does not depend upon any of the predicates without definitions. This algorithm is a variation on the context-free grammar (CFG) marking algorithm for determining whether or not a grammar is empty: when a biconditional for p is found, all occurrences of p in the remaining sentences are marked, and when a sentence has all but one of its predicates marked, it is a candidate for selection.

For the sentences in Figure 1, the first selection step might choose the sentences in which the predicates $\{ans, =\}$ occur or the sentences in which $\{hue, =\}$ occur. In the former case, the confirmation step fails, so suppose the algorithm chooses the sentences in the latter case.

$hue(red)$
$hue(blue)$
$\neg hue(x) \vee x = red \vee x = blue$

Assuming CONFIRM finds the definition for hue entailed by these sentences,

$$hue(x) \Leftrightarrow (x = red \vee x = blue),$$

the algorithm recurses on the sentences that were not selected, this time attempting to find a definition for the remaining predicate ans. The recursive call knows that the definition for ans can depend upon both $=$ and hue because the definition for hue has already been found. Thus, as one would expect, all of the remaining sentences are selected because each mentions only ans and a subset of $\{hue, =\}$. Assuming CONFIRM finds the definition for ans entailed by the selected sentences, the overall result of the algorithm is Sentence Set 4.

Algorithm 1, PREP-FOR-LLOYDTOPOR, embodies the procedure outlined here and is invoked with a sentence set Δ and a singleton set of predicates: $\{=\}$. It runs in time

$$O(|\text{PREDS}(\Delta)|^2(|\Delta| + f_{\text{CONFIRM}}(\Delta))),$$

where $|\text{PREDS}(\Delta)|$ is the number of predicates in Δ, $|\Delta|$ is the size of Δ, and $f_{\text{CONFIRM}}(\Delta)$ is the maximum cost of CONFIRM run on any selected subset of Δ.

Algorithm 1 PREP-FOR-LLOYDTOPOR(Δ, $basepreds$)

1: $sents$:= $\{\langle p, d\rangle | d \in \Delta$ and $p \notin basepreds$ and PREDS(d) contains just p and members of $basepreds\}$
2: $preds$:= $\{p | \langle p, d\rangle \in sents\}$
3: $bicond$:= false
4: **for all** $p \in preds$ **do**
5: $partition$:= $\{d | \langle p, d\rangle \in sents\}$
6: $bicond$:= CONFIRM($partition$, p)
7: $pred$:= p
8: **when** $bicond \neq$ false **then** exit **for all**
9: **end for**
10: **when** $bicond =$ false **then return** false
11: $remaining$:= $\Delta - partition$
12: **when** $remaining = \{\}$ **then return** $\{bicond\}$
13: $rest$:= PREP-FOR-LLOYDTOPOR($remaining$, $\{pred\} \cup basepreds$)
14: **when** $rest \neq$ false **then return** $\{bicond\} \cup rest$
15: **return** false

The polynomial complexity is achieved by forgoing completeness, though soundness is guaranteed. Moreover, that incompleteness cannot be blamed on CONFIRM, for even if CONFIRM were complete, PREP-FOR-LLOYDTOPOR would still be incomplete. The nature of that incompleteness is characterized by the theorem below. Proofs can be found in (Hinrichs 2007).

Theorem 1 (PREP-FOR-LLOYDTOPOR Soundness). *Under the conditions listed below, if* PREP-FOR-LLOYDTOPOR(Δ, $\{=\}$) *returns a nonempty* Γ*, then* Γ *is a nonrecursive set of biconditionals with one definition per predicate in* Δ *besides equality and* Δ *is logically equivalent to* Γ.

- Δ *is a satisfiable sentence set in* FHL.
- CONFIRM *is sound, i.e. if* CONFIRM(Σ, p) *returns* $p(\bar{x}) \Leftrightarrow \phi(\bar{x})$ *then the result is a nonrecursive biconditional definition for* p *entailed by* Σ.

Theorem 2 (PREP-FOR-LLOYDTOPOR Completeness). *Under the conditions listed below, if* Δ *is a complete theory in* FHL *then* PREP-FOR-LLOYDTOPOR($\Delta, \{=\}$) *does not return false.*

- Δ *is a satisfiable, nonempty sentence set.*
- Δ *is logically equivalent to a nonrecursive set of biconditionals with one definition per predicate besides equality:* $\{\beta_1, \ldots, \beta_n\}$.
- *There is a partitioning of* Δ *into* $S_1, \ldots, S_n$ *such that* β_i *is logically equivalent to* S_i *and* PREDS(β_i) = PREDS(S_i)*, for every* i.
- CONFIRM *is complete, i.e. if* Σ *entails some nonrecursive biconditional definition* $p(\bar{x}) \Leftrightarrow \phi(\bar{x})$ *then* CONFIRM(Σ, p) *returns an equivalent biconditional definition for* p*; otherwise, it returns false.*

Recall that the purpose of PREP-FOR-LLOYDTOPOR is to make a complete FHL theory amenable to the Lloyd-Topor algorithm, which translates FHL sentences into NR-DATALOG$^\neg$. The example given earlier conveys the basic idea of Lloyd-Topor: turn each biconditional defining predicate p into a series of NR-DATALOG$^\neg$ rules, each of which has p in the head. We use the name LLOYDTOPOR to refer to the full algorithm, which deals with quantifiers, negation, disjunction, etc.

The algorithm for translating a complete FHL theory into a logically equivalent NR-DATALOG$^\neg$ program is given in Algorithm 2 (as if PREP-FOR-LLOYDTOPOR were complete). After converting the theory into a nonrecursive set of biconditionals using PREP-FOR-LLOYDTOPOR, the algorithm applies LLOYDTOPOR. The result is a logically equivalent NR-DATALOG$^\neg$ program. (In addition to translating an FHL theory into NR-DATALOG$^\neg$, Algorithm 2 handles a satisfaction query in the usual way—by defining a fresh predicate $ans(\bar{x})$ so that it is equivalent to the query.)

Algorithm 2 COMPLETE-TO-DATALOG(Δ, $\phi(\bar{x})$)

Assumes: Δ is a finite set of FHL sentences and $\phi(\bar{x})$ is an FHL sentence with free variables $\bar{x}$.
1: $biconds$:= PREP-FOR-LLOYDTOPOR(Δ, $\{=\}$)
2: Δ' := LLOYDTOPOR($biconds \cup \{ans(\bar{x}) \Leftrightarrow \phi(\bar{x})\}$)
3: **return** Δ' and $ans(\bar{x})$

Algorithm 2 assumes that the given theory is known to be complete, which is a valid assumption in the upcoming section, but sometimes the completeness of a theory is unknown. To use Algorithm 2, we may need to determine whether or not a given sentence set is complete. With the limitations listed in Theorem 2, the implementation of PREP-FOR-LLOYDTOPOR discussed here can detect complete theories. If the algorithm rewrites the theory as a nonrecursive set of biconditionals, the theory is complete. If the algorithm fails to find a rewriting, the theory may or may not be complete. As it turns out, treating a complete theory as if

it were incomplete may affect efficiency; however, it will not affect the overall soundness or completeness of the method.

The Incomplete Case

Incomplete FHL theories are satisfied by more than one model. Every NR-DATALOG$^\neg$ program is satisfied by exactly one model. Thus, unlike the case of complete FHL theories, translating an incomplete FHL theory into NR-DATALOG$^\neg$ does not preserve logical equivalence. But for the translation to be useful, something must be preserved. In this paper we have chosen to preserve the entailment/satisfiability of a single, given sentence: the satisfaction query. This section introduces an algorithm for the task. It is a form of satisfaction-preserving theory-completion operating entirely within FHL: a given FHL theory Δ and a sentence $\phi(\bar{x})$ are transformed into a complete FHL theory Δ' and a sentence $\phi'(\bar{x})$ so that for every object constant tuple $\bar{a}$,

$$\Delta \cup \{\phi(\bar{a})\} \text{ is satisfiable}$$
$$\text{if and only if}$$
$$\Delta' \cup \{\phi'(\bar{a})\} \text{ is satisfiable.}$$

After performing this transformation, the results of the previous section can be applied to transform Δ' and $\phi'(\bar{x})$ into NR-DATALOG$^\neg$.

Recall that satisfaction-preserving theory-completion is a process by which answers that were originally satisfiable become entailed (because satisfaction and entailment coincide in complete, satisfiable theories). Of the main forms of inference (deduction, abduction, and induction), it turns out that abduction is the procedure most closely related to a satisfaction-to-entailment rewriting.

Abduction is the process that finds explanations for why a conclusion could possibly hold. Given a set of premises Δ and a conclusion ϕ, abduction finds another sentence set Γ so that when added to Δ, ϕ is entailed: $\Delta \cup \Gamma \models \phi$. Intuitively, ϕ is only satisfiable to start, and abduction finds a way to make it entailed; thus, abduction is a satisfaction-to-entailment procedure.

One technical detail about abduction must be mentioned as it bears on the upcoming discussion. Abduction places certain constraints on Γ to avoid uninteresting explanations. Of interest here is the constraint that Γ must be written using a prescribed set of predicates P. This allows us to ensure that Γ does not include ϕ itself.

The satisfaction-preserving theory-completion algorithm presented here is inspired by a well-known and easy-to-implement abductive variant of model elimination. (McIlraith 1998) calls the variant proof-tree completion, which we explain by example.

Consider the following propositional rules.

$$r \Leftarrow q \wedge p$$
$$p \Leftarrow s$$
$$p \Leftarrow t$$

Suppose we wanted to construct an explanation for why r might be entailed using the predicates $\{q, s, t\}$. There are two such explanations: $q \wedge s$ and $q \wedge t$. Proof-tree completion is an abduction algorithm that will compute such explanations. Very similar to model elimination (which is itself very similar to Prolog), the trace for computing the condition $q \wedge s$ is shown in Figure 2.

Call: r	r	()
Call: q	[r] q	()
Save: q	[r] [q]	(q)
Call: p	[r] p	(q)
Call: s	[r] [p] s	(q)
Save: s	[r] [p] [s]	(q, s)
Exit: p	[r] [p]	(q, s)
Exit: r	[r]	(q, s)

Figure 2: Proof-tree completion example, showing the trace, the stack, and the set of saved literals.

One can also view proof-tree completion as a rewriting procedure. In the example above, proof-tree completion rewrote r in terms of $\{q, s, t\}$: $(q \wedge s) \vee (q \wedge t)$. There is a proof for r if and only if there is a proof for the disjunction. Such an entailment-preserving rewriting forms the basis for our satisfaction-preserving theory-completion algorithm.

Definition 3 (Entailment-Preserving Rewriting). *Let Δ be a finite sentence set, P a set of predicates, and $\phi(\bar{x})$ a sentence with free variables $\bar{x}$. An entailment-preserving rewriting of $\phi(\bar{x})$ in terms of P using Δ is a sentence $\psi(\bar{x})$ such that the following hold.*

- *The only predicates that occur in $\psi(\bar{x})$ are in P.*
- *For every tuple of object constants $\bar{a}$,*

$$\Delta \models \phi(\bar{a}) \text{ if and only if } \Delta \models \psi(\bar{a}).$$

We will denote a generic entailment-preserving rewriting procedure as EPR.

Notice that entailment-preserving rewriting requires three inputs: the sentence to be rewritten, a theory, and a set of predicates P. Our procedure for satisfaction-preserving theory-completion is a simple application of EPR. Two of the inputs to EPR come from the inputs for satisfaction-preserving theory-completion: the satisfaction query and the theory; all that remains is choosing the predicate set P. The output of EPR corresponds to the new query about the completed theory.

Intuitively, if EPR produces a query about a complete theory, then the predicates used to express that query must have complete definitions. An important observation is that every theory in FHL can be decomposed into the complete portion (corresponding to the set of predicates with complete definitions) and the incomplete portion (the remaining predicates). For a theory broken into the complete portion C, with definitions for predicates Q, and the remainder of the theory I, EPR can be used to perform satisfaction-preserving theory-completion. The completed theory is simply C, and the new query is an application of EPR using the theory $C \cup I$, the predicates Q, and the satisfaction query $\phi(\bar{x})$.

Sometimes theories decomposed as $C \cup I$ occur in nature, but sometimes the decomposition requires some work. A small change to the PREP-FOR-LLOYDTOPOR algorithm discussed in the previous section produces an algorithm that can decompose a theory into $C \cup I$. Instead of failing when

the remaining sentences entail no biconditional, the altered version of PREP-FOR-LLOYDTOPOR returns all of the confirmed biconditionals as C and all of the remaining sentences as I. Like the original, the resulting algorithm is sound but incomplete, meaning that everything it puts into C is in fact complete, but it may fail to make C as large as possible. Incompleteness is tolerable because the decomposition is an optimization. The EPR algorithm works correctly as long as C is complete, but as a rule of thumb, it performs better when C is larger.

A theory partitioned into $C \cup I$ is amenable to satisfaction-preserving theory-completion via EPR. To illustrate how to apply EPR, we first consider the naïve approach. Consider the graph coloring example, where the decomposition into $C \cup I$ has already been performed. The incomplete portion of the theory is shown below. The complete portion consists of the definitions for $node$, hue, and adj, found in Sentence Set 1.

$$color(x,y) \Rightarrow node(x) \wedge hue(y)$$
$$color(x,y) \wedge adj(x,z) \Rightarrow \neg color(z,y)$$

Recall that the graph coloring problem corresponds to finding variable assignments that satisfy the sentence

$$color(n_1,x) \wedge color(n_2,y) \wedge color(n_3,z).$$

In terms of types, we can apply EPR to rewrite the sentence above using Sentence Set 1 in terms of the predicates $\{node, hue, adj, =\}$. The result turns out to be empty because there are no colorings of the graph that are entailed. Even though multiple colorings are consistent, not one is entailed.

All is not lost. Instead of applying EPR to the satisfaction query itself, we can apply it to the negation of the query, producing an expression that captures exactly those answers that are necessarily inconsistent with the theory.

For example, applying proof-tree completion to rewrite the query $\neg(color(n_1,x) \wedge color(n_2,y) \wedge color(n_3,z))$, in terms of $\{hue, node, adj, =\}$ using Sentence Set 1 produces a disjunction with nine disjuncts.

1. $\neg node(n_1)$	4. $\neg hue(x)$	7. $adj(n_1,n_2) \wedge x = y$
2. $\neg node(n_2)$	5. $\neg hue(y)$	8. $adj(n_2,n_3) \wedge y = z$
3. $\neg node(n_3)$	6. $\neg hue(z)$	9. $adj(n_1,n_3) \wedge x = z$

Such a disjunction enumerates the conditions under which the original query is inconsistent with the theory. The conditions under which the original query is consistent with the theory is then the negation of that disjunction, and it is this expression that constitutes the new query required for satisfaction-preserving theory-completion.

The negation of the disjunction of the above nine expressions is shown below as the definition for $ans(x,y,z)$.

$$ans(x,y,z) \\ \Leftrightarrow \\ \begin{pmatrix} node(n_1) \wedge hue(x) \wedge (adj(n_1,n_2) \Rightarrow x \neq y) \wedge \\ node(n_2) \wedge hue(y) \wedge (adj(n_2,n_3) \Rightarrow y \neq z) \wedge \\ node(n_3) \wedge hue(z) \wedge (adj(n_1,n_3) \Rightarrow x \neq z) \end{pmatrix}$$

It is noteworthy that this definition is qualitatively different than the one in Sentence Set 4. It assumes nothing about adj nor that n_1, n_2, and n_3 are actually nodes. Producing the other definitions using proof-tree completion would require reducing the predicates for expressing the rewriting from $\{hue, node, adj, =\}$ to $\{hue, =\}$. However, because we are interested in theory-completion and each of those predicates already has a complete definition in C, there is no reason to force proof-tree completion to absorb those definitions during rewriting.

This example illustrates our satisfaction-preserving theory-completion algorithm when applied to a theory Δ partitioned into $C \cup I$ and a sentence $\phi(\bar{x})$, shown formally in Algorithm 3. Run EPR on $\neg\phi(\bar{x})$ giving it the sentences $C \cup I$ and the predicates defined in C (extracted with PREDS). The result is a sentence $\psi(\bar{x})$. Add the following definition to C to produce the completed theory. The new query is $ans(\bar{x})$.

$$ans(\bar{x}) \Leftrightarrow \neg\psi(\bar{x})$$

Intuitively, this algorithm boils away the incompleteness of a decomposed theory $C \cup I$ to rewrite the sentence $\phi(\bar{x})$ entirely in terms of predicates with definitions in C.

Algorithm 3 INCOMPLETE-TO-COMPLETE($C \cup I$, $\phi(\bar{x})$)

Assumes: $C \cup I$ is a finite, satisfiable set of FHL sentences, where C is complete, and $\phi(\bar{x})$ is an FHL sentence with free variables $\bar{x}$.

1: $\psi(\bar{x})$:= EPR($\neg\phi(\bar{x})$, $C \cup I$, PREDS(C) $\cup \{=\}$)
2: Δ' := $C \cup \{ans(\bar{x}) \Leftrightarrow \neg\psi(\bar{x})\}$
3: **return** Δ' and $ans(\bar{x})$

The following theorem guarantees that Algorithm 3 performs satisfaction-preserving theory-completion.

Theorem 3 (Soundness and Completeness). *Suppose $C \cup I$ is a satisfiable* FHL *theory where C consists of complete definitions for predicates P, and only predicates P occur in C. Given $C \cup I$ and $\phi(\bar{x})$, Algorithm 3 implements satisfaction-preserving theory-completion.*

Proof. $\psi(\bar{x})$ is the result of an entailment-preserving rewriting, which ensures that for every $\bar{a}$

$$C \cup I \models \neg\phi(\bar{a}) \text{ if and only if } C \cup I \models \psi(\bar{a}).$$

Because only the predicates $P \cup \{=\}$ occur in $\psi(\bar{x})$, $C \cup I$ is satisfiable, and C is complete, I does not contribute to the proof of $\psi(\bar{a})$:

$$C \cup I \models \neg\phi(\bar{a}) \text{ if and only if } C \models \psi(\bar{a}).$$

Negating both sides of the if and only if gives

$$C \cup I \not\models \neg\phi(\bar{a}) \text{ if and only if } C \not\models \psi(\bar{a}).$$

On the left, $C \cup I$ does not entail $\neg\phi(\bar{a})$ exactly when there is some model that satisfies $\phi(\bar{a})$. On the right, because C is complete, it entails either $\psi(\bar{a})$ or $\neg\psi(\bar{a})$ for all $\bar{a}$.

$$C \cup I \cup \{\phi(\bar{a})\} \text{ is satisfiable if and only if } C \models \neg\psi(\bar{a}).$$

Assuming ans is a fresh predicate, the body of the definition for $ans(\bar{x})$ is $\neg\psi(\bar{x})$, which ensures

$$C \cup I \cup \{\phi(\bar{a})\} \text{ is satisfiable} \\ \text{if and only if} \\ C \cup \{ans(\bar{x}) \Leftrightarrow \neg\psi(\bar{x})\} \models ans(\bar{a}).$$

Finally, because the lower theory is complete, the equivalence of satisfaction and entailment guarantees we have an instance of satisfaction-preserving theory-completion. □

Experiments

The purpose of translating a "what" language into a "how" language is to gain efficiency. This section discusses experiments investigating the efficiency of FHL and NR-DATALOG$^\neg$, respectively the "what" and "how" languages studied in this paper. More precisely, the experiments compare two particular systems for processing FHL and NR-DATALOG$^\neg$. Paradox (Claessen & Sorensson 2003), the system for processing FHL, attempts to build a finite model that satisfies a given set of first-order sentences. It was chosen for two reasons: it won the finite satisfiability division of CASC (Suttner & Sutcliffe 1998) the last five consecutive years, and it outperformed Mace4 (McCune 2003) and FMDarwin (Baumgartner *et al.* 2007) on our initial experiments. Epilog, the system for processing NR-DATALOG$^\neg$, performs top-down, left-to-right evaluation without tabling/caching. XSB (`http://xsb.sourceforge.net/`), the famous Prolog implementation, imposes rule-length limitations that made it unsuitable for our experiments.

The experiments consisted of a series of graph coloring problems, where the FHL and NR-DATALOG$^\neg$ formulations were almost identical to those shown in the Introduction. The FHL formulation differed only in that the UNA and DCA were included; the NR-DATALOG$^\neg$ formulation differed only because we employed a simple and easy to implement conjunct-ordering scheme: all negative literals were moved as far to the left as possible in each rule.

The graphs ranged from 25 to 60 nodes in increments of five and contained edges that were chosen uniformly at random. Each graph size was tested on 20 different edge sets. For each graph, we tested three different sets of colors and guaranteed that at least one of the color sets yielded a viable coloring. If we let k denote the maximum number of edges incident on any node, the three sets contained $k+1$, $\frac{2}{3}k$, and $\frac{1}{3}k$ colors. It can easily be shown that $k+1$ colors is sufficient to guarantee the existence of a coloring[2].

All experiments were run on a Linux box with 500MB of RAM and a 1.3GHz Athlon processor. The results reported below ignore the cost of reformulating FHL into NR-DATALOG$^\neg$, under the assumption that the transformation is amortized over many different queries (different color sets). All experiments were limited to 300 seconds and were reported as 300 if they exceeded that limit.

Table 1 summarizes the results. The average performance of Epilog was better than Paradox by about 25%, but the performance of Epilog was less predictable than Paradox by about 20%. Epilog outperformed Paradox 70% of the time, with an average speedup (ratio of Paradox's time to Epilog's time) of almost 100x (especially significant when the largest possible speedup is 300x). In those tests where Paradox outperformed Epilog, the average speedup was 10x.

	Paradox			Epilog		
Overall						
Average	107s			84s		
Stdev	109			131		
% wins	26%			73%		
Speedup	10x			96x		
By Hues	$\frac{1}{3}k$	$\frac{2}{3}k$	$k+1$	$\frac{1}{3}k$	$\frac{2}{3}k$	$k+1$
Average	107s	93s	120s	250s	1s	1s
Stdev	110	101	115	100	1	1
% wins	90%	0%	0%	9%	100%	100%
Speedup	10x	N/A	N/A	24x	86x	112x
By Sat	Sat	Unsat		Sat	Unsat	
Average	103s	26s		52s	209s	
Stdev	102	41s		111	121	
% wins	18%	90%		82%	8%	
Speedup	5x	17x		98x	6x	

Table 1: Summary of graph coloring experiments for Paradox (FHL) and Epilog (NR-DATALOG$^\neg$). The results are broken down by the number of colors and whether or not the graph could be colored. Average time (in seconds) and standard deviation are reported. Additionally, the percentage of problems for which one system runs in less time than the other and the average speedup when one system outperforms the other are reported.

As expected, the number of available colors had a significant impact on the results. Epilog dominated in the case of $\frac{2}{3}k$ and $k+1$ colors: it outperformed Paradox in every single test, with an average speedup of about 100x. However, in the case of $\frac{1}{3}k$ colors, Paradox outperformed Epilog in 90% of the tests with an average speedup of 10x. The $\frac{1}{3}k$ case differed from the other two because it contained all of the graph coloring instances without solutions.

One might conjecture that Epilog outperformed Paradox on every satisfiable problem, and Paradox outperformed Epilog on every unsatisfiable problem. The data refutes this conjecture but supports the general sentiment. Epilog outperformed Paradox on 82% of satisfiable problems, and Paradox outperformed Epilog on 90% of unsatisfiable problems.

The scalability of the two systems was measured by examining graphs of different sizes. Figure 3 shows a log-scale plot of the data arranged by graph size for the two larger color sets ($\frac{2}{3}k$ and $k+1$), where Epilog dominated Paradox. Epilog's run-time is virtually unchanged as the graph size increases, but Paradox's run-time increases, as shown by the power regression lines. Figure 4 plots the remaining color set ($\frac{1}{3}k$) using the standard scale. Epilog's performance is unpredictable, and Paradox's performance gradually grows

[2]Inductive step: suppose a graph with n nodes of maximum degree k can be colored with $k+1$ colors. Consider a graph of size $n+1$ with maximum degree k. Remove an arbitrary node. The remaining graph has n nodes, and retains the property that the maximum degree is k. By the inductive hypothesis, color that graph with $k+1$ colors. Add the removed node back into the graph, and extend the coloring by painting the new node whichever color none of its neighbors (of which there are at most k) have been painted.

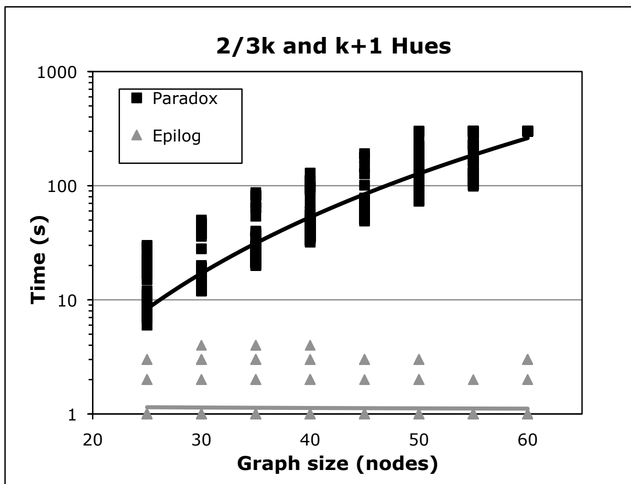

Figure 3: A log plot comparing Paradox and Epilog on the two larger sets of colors.

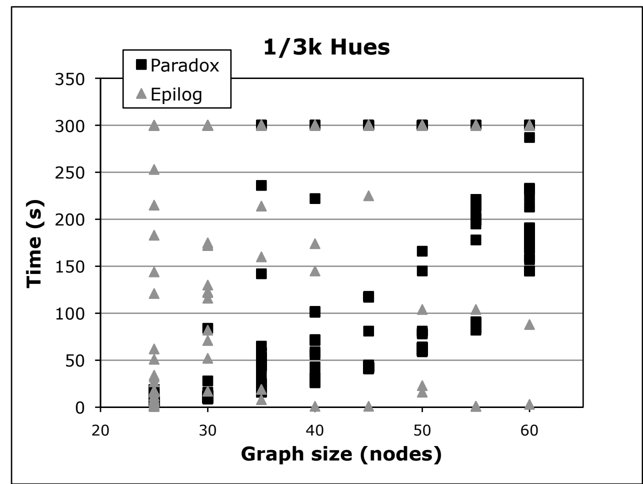

Figure 4: A standard plot comparing Paradox and Epilog on the smallest set of colors.

with increasing graph size. Trend lines have been omitted in Figure 4 due to their questionable signficance. Breaking down the data depicted in Figure 4 based on whether or not the problem was satisfiable is no more illuminating than the plot shown here.

The conclusion we draw from these results is that Epilog is less predictable than Paradox, but it has more potential for efficiency. For the graph coloring problems tested, Epilog outperformed Paradox on 70% of the problems and did so by 100x; Paradox outperformed Epilog on 30% of the problems and did so by 10x. Epilog's performance was marred by poor results on unsatisfiable and what we believe were the hardest satisfiable problems. For such problems, an intelligently crafted search space is more important than for easy problems. Our experiments paid little attention to optimizing Epilog's randomly-generated search space, and Epilog itself does nothing to prune the space it has been given; it simply explores that search space very rapidly. A plethora of relatively easy, satisfiable problems enabled Epilog to outperform Paradox overall, but these experiments make clear that techniques for automatically optimizing NR-DATALOG$^\neg$ are very important. Fortunately, such techniques have been studied by the database and logic programming communities, e.g. (Chirkova 2002), and in future experiments we hope to include NR-DATALOG$^\neg$ optimizations.

Related Work

The related work spans several areas including translation algorithms, Knowledge Compilation, answer set programming (ASP), constraint satisfaction, language interoperability, and nonmonotonic reasoning.

In the long tradition of algorithms for translating between formal languages, the relevant work we are aware of starts with first-order logic and converts to propositional logic using a finite universe (Ramachandran & Amir 2005; Gammer & Amir 2007; Claessen & Sorensson 2003; Schulz 2002) or starts with propositional logic and converts to answer set programming (Niemela 1999). Targeting propositional logic does not appear to illuminate the what-vs-how tradeoff as order (the criterion used to separate languages on the what-to-how spectrum) is equally irrelevant in both the source and target. Translating propositional logic to ASP bridges the classical to logic programming gap; however, because ASP is a hybrid of classical logic and logic programming, translating to ASP is less informative than NR-DATALOG$^\neg$ in terms of what-vs-how.

Knowledge Compilation endeavors to reduce the complexity of query-answering by building a structure that is either logically equivalent but in a more vivid form (Levesque 1986; Etherington *et al.* 1989; Davis 1994) or not logically equivalent but an efficient approximation, e.g. (Kautz & Selman 1991; Selman & Kautz 1991; Kautz & Selman 1992; Hammer & Kogan 1993; Kautz, Kearns, & Selman 1995; del Val 1996). Much of the work has focused on propositional logic and considers various target languages for compilation (Darwiche & Marquis 2002). Compilation for propositional logic (as opposed to FHL) is limited in that only a finite number of distinct queries can be asked; in the FHL graph coloring example, the set of hues can be made arbitrarily large without having to recompile. Knowledge Compilation is mature enough to have developed a theoretical framework defining complexity classes and reduction techniques for problems that lend themselves to compilation (Cadoli *et al.* 2000; Chen 2003). In particular, the framework focuses on classifying problems based on whether or not compilation routines exist for decreasing a problem's computational complexity, which necessitates exponential compilation algorithms. In contrast, in our work we would prefer polynomial compilation algorithms where the result simply runs faster by some polynomial (or even constant) factor, in analogy to traditional compilers today. The fact that FHL belongs to NEXPTIME and NR-DATALOG$^\neg$ belongs to PSPACE is unfortunate, from our perspective, because it virtually guarantees that the compilation algorithm

must be exponential. In the future we plan on examining the translation from FHL to DATALOG with stratified negation and recursion, narrowing the complexity gap.

Constraint satisfaction is likewise intimately connected to this work. The satisfaction queries we consider, however, are not expressed in a form that would be appropriate for the yearly CSP competition (`http://cpai.ucc.ie/08/`). Our queries are written using sentences in classical logic, but the queries in the CSP competition are expressed using permissible and conflicts tables[3], i.e. tables that explicitly state which combinations of values can and cannot be assigned to which combinations of variables, respectively. This tabular input format can be seen as a special case of NR-DATALOG$^\neg$, which means that the algorithms presented here make strides toward algorithms for reformulating a constraint satisfaction problem expressed using classical logic into the form CSP solvers expect. It is noteworthy that the language specification for the competition implies that this restriction may soon be lifted, effectively allowing queries written in NR-DATALOG$^\neg$.

Language interoperability, e.g. (Motik & Rosati 2007; Eiter *et al*. 2004) is an important problem in the semantic web community, and formally the work presented here belongs in this category; however, from a practical standpoint, there are better solutions to building a reasoner that handles both FHL and NR-DATALOG$^\neg$ than the conversion discussed here, e.g. converting NR-DATALOG$^\neg$ into FHL. One could also procedurally attach a (deductive) database to a theorem prover, obviating the need to perform any translation at all.

Because theory-completion is so central to our work, results on nonmonotonic reasoning are applicable. Recall that our form of theory-completion requires that the satisfaction of the query of interest be preserved. This differs from what is usually studied in the nonmonotonic reasoning literature, e.g. the closed-world assumption (Reiter 1978), predicate completion (Lloyd 1984), and circumscription (McCarthy 1988), because in those settings it is desirable to minimize the theory in some way—to change its logical consequences. In contrast, our theory-completion is performed only so that a different suite of automated reasoning tools can be used to answer the query at hand; satisfiability and unsatisfiability are both preserved for a given query.

Conclusions and Future Work

To investigate injecting information about how to solve a problem into a description of the problem itself, we designed algorithms that automatically translate a decidable, classical logic (FHL) into one of the more procedural logic programming languages (NR-DATALOG$^\neg$). Those reformulation algorithms implement a form of theory-completion that can preserve either a satisfaction or a logical entailment query. These algorithms enable a standard (deductive) database system to answer queries about a classical logic.

Much work remains. The proof-tree completion algorithm that inspired the theory-completion algorithm does not terminate in all cases. The typical example is ancestry. Suppose we want to rewrite the sentence $anc(x, y)$ in terms of the predicate $parent$.

$$anc(x, y) \Leftarrow parent(x, y)$$
$$anc(x, y) \Leftarrow parent(x, z) \wedge anc(z, y)$$

Without knowing the size of the universe, proof-tree completion will attempt to unroll the transitivity, resulting in a never-ending procedure. Consequently, it is unclear whether or not proof-tree completion constitutes a complete entailment-preserving rewriting procedure.

Second, proof-tree completion only applies to quantifier-free sentences. Addressing quantifiers is a good next step. Because quantifiers in FHL range over a finite and known set of objects, any quantifed sentence can be rewritten as a ground, quantifier-free sentence, enabling the algorithms reported here to perform the translation. However, this form of quantifier-elimination is exponential in time and space.

Third, in some situations it is desirable to translate an FHL theory into NR-DATALOG$^\neg$ so that more than one query can be answered by the result. It appears that a generatively-complete version of resolution can be used instead of the query-sensitive proof-tree completion algorithm as the basis for such a reformulation algorithm. Again, termination is an issue, as the resolution closure may be infinite without taking the finite universe into account.

Fourth, the translation of FHL to NR-DATALOG$^\neg$ must absorb a complexity theoretic jump from coNEXPTIME to PSPACE for entailment queries, which probably guarantees an exponential lower bound. Translating FHL to DATALOG with recursion and negation would change the jump to coNEXPTIME to NEXPTIME, and would allow the translation algorithm to leverage recursion, an important feature in functional and imperative languages.

Fifth, one could investigate translating NR-DATALOG$^\neg$ into a functional language, the key difference being that declarative languages use the same statements to both validate answers (given a ground query, check if it is entailed) and generate answers (given a query with variables, find bindings for the variables so the query is entailed). Functional languages do not natively support the latter functionality, and doing so efficiently can be challenging.

Sixth, one could consider translating classical logic into other automated reasoning paradigms and empirically comparing the results. FHL could be reasoned about with SAT solvers, CSP solvers, ASP solvers, and first-order theorem provers. Each tool requires a different translation algorithm. For all but ASP, proof of concept tests have been performed on complete theories, showing that NR-DATALOG$^\neg$ is sometimes superior to the tools mentioned above (Hinrichs 2007).

Seventh, one could investigate translating full first-order logic (FOL) into a logic programming language with similar expressiveness, e.g. Prolog with stratified negation. The approach used in this paper is unlikely to yield a decision algorithm because it relies on constructing definitions that capture satisfiability, which in first-order logic is not recursively enumerable. At the very least, it would be informative to translate a decidable fragment of FOL with function constants into Prolog, forcing the translator to tolerate complex terms. The proof of Theorem 3 guarantees that for a broad class of logics, an entailment-preserving rewriting is suffi-

[3] The only exceptions are arithmetic expressions over integers.

cient to perform satisfaction-preserving theory-completion. The proof appears to work for answer set programming and first-order logic, though a precise characterization of the applicable class of logics is the subject of future work.

Whether or not these issues have positive resolutions, it seems clear that people are sometimes able to perform the transformations discussed here and do so for good reason. Investigating algorithms that automatically perform those transformations forces us to understand more deeply the tradeoffs of those languages.

References

Baumgartner, P.; Fuchs, A.; Nivelle, H.; and Tinelli, C. 2007. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*.

Cadoli, M.; Donini, F. M.; Liberatore, P.; and Schaerf, M. 2000. Preprocessing of intractable problems. *Information and Computation* 176(2):89–120.

Chen, H. 2003. A theory of average-case compilability in knowledge representation. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Chirkova, R. 2002. *Automated Database Restructuring*. Ph.D. Dissertation, Stanford University.

Claessen, K., and Sorensson, N. 2003. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE Workshop on Model Computation*.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17:229–264.

Davis, E. 1994. Lucid representations. Technical report, New York University.

del Val, A. 1996. Approximate knowledge compilation: The first order case. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 498–503.

Eiter, T.; Lukasiewicz, T.; Schindlauer, R.; and Tompits, H. 2004. Combining answer set programming with description logics for the semantic web. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*.

Etherington, D.; Borgida, A.; Brachman, R. J.; and Kautz, H. 1989. Vivid knowledge and tractable reasoning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1146–1152.

Feigenbaum, E. A. 1996. How the "what" becomes the "how". *Communications of the ACM* 39(5):97–104.

Gammer, I., and Amir, E. 2007. Solving satisfiability in ground logic with equality by efficient conversion to propositional logic. In *Proceedings of the Symposium of Abstraction, Reformulation, and Approximation*, 169–183.

Hammer, P., and Kogan, A. 1993. Optimal compression of propositional Horn knowledge bases: Complexity and approximation. *Artificial Intelligence* 64(1):131–145.

Hinrichs, T. L. 2007. *Extensional Reasoning*. Ph.D. Dissertation, Stanford University.

Kautz, H., and Selman, B. 1991. A general framework for knowledge compilation. In *Proceedings of the International Workshop on Processing Declarative Knowledge*, 287–300.

Kautz, H., and Selman, B. 1992. Forming concepts for fast inference. In *Proceedings of the ECAI Workshop on Knowledge Representation*, 200–215.

Kautz, H.; Kearns, M.; and Selman, B. 1995. Horn approximations of empirical data. *Artificial Intelligence* 74(1):129–145.

Levesque, H. J. 1986. Making believers out of computers. *Artificial Intelligence* 30(1):81–108.

Lloyd, J., and Topor, R. 1984. Making Prolog more expressive. *Journal of Logic Programming* 1(3):225–240.

Lloyd, J. 1984. *Foundations of Logic Programming*. Springer-Verlag.

Loveland, D. W. 1978. *Automated Theorem Proving: A Logical Basis*. North-Holland Publishing.

McCarthy, J. 1982. Coloring maps and the Kowalski doctrine. Technical report, Stanford University.

McCarthy, J. 1988. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence* 13:27–39.

McCune, W. 2003. Mace4 reference manual and guide. Technical report, Argonne National Laboratory.

McIlraith, S. 1998. Logic-based abductive inference. Technical report, Stanford University.

Motik, B., and Rosati, R. 2007. A faithful integration of description logics with logic programming. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 477–482.

Niemela, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3-4):241–273.

Ramachandran, D., and Amir, E. 2005. Compact propositional encodings of first-order theories. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 340–345.

Reiter, R. 1978. On closed world databases. In *Proceedings of the International Conference on Management of Data*.

Schulz, S. 2002. A comparison of different techniques for grounding near-propositional CNF formulae. In *FLAIRS Conference*, 72–76.

Selman, B., and Kautz, H. 1991. Knowledge compilation using Horn approximations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 904–909.

Suttner, C., and Sutcliffe, G. 1998. The CADE-14 ATP system competition. *Journal of Automated Reasoning* 21(1):99–134.

Ullman, J. 1989. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.

Vorobyov, S., and Voronkov, A. 1998. Complexity of non-recursive logic programs with complex values. In *ACM SIG for the Management of Data*, 244–253.