Queryable Acyclic Production Systems*

David Tanzer and Dennis Shasha
Computer Science Department
New York University
{tanzer,shasha}@cs.nyu.edu

Abstract

We pose a query problem about the behavior of a consultation system S: given a constraint formula q and a potential conclusion c for S, determine if there is a user input binding that satisfies q and causes S to conclude c. Existing rule-based expert systems, both forward and backward chaining[3], implement a consultation mechanism S, but are not designed for these queries *about* S. For general production systems, the queries are undecidable. Here we solve the problem for useful sublanguages of acyclic production systems.

We implement a query tool in a Datalog + constraints framework, and optimize for "embedded decision trees" in the rule system. Our data complexity is $\Theta(n \cdot f(n))$ in the size of the embedded trees, versus $\Theta(n \cdot f(n) + n^2)$ for existing datalog evaluation algorithms, where $f(n)$ is the cost of destructively conjoining a constraint of unit size into a conjunction of n constraints.

1 Introduction

We define *expert systems* as machines programmed to emulate the reasoning functions of a human expert, and classify them either as *consultation systems*, which interactively question a user in order to reach conclusions, or as *non-consultative expert systems*, which use knowledge-bases as their exclusive form of input. Production systems[3] are a well-known language for writing consultation systems. Thinksheet[7, 8] is a sublanguage of acyclic production systems that has been used to organize complex documents.

*This work was supported in part by the U.S. National Science Foundation under grant number IRI-9224601.

CIKM '99 11/99 Kansas City, MO, USA

A *queryable consultation system* should be able to (1) reach conclusions by posing a context-dependent set of relevant questions to a user, and (2) report which conclusions are reachable under a constraint q ("query formula") on the set of valid inputs. The system queries the user in the first case, and the user queries the system in the second.

The significance of our system is that for many useful applications it provides good support for *both* these modes of operation. In contrast, existing knowledge-based systems are divided into those that support one mode or the other. Existing consultation systems[3] query a user, but are not designed for user queries about the reachability of conclusions. Databases[11, 13, 2, 6] can be queried by the user, but they are not consultation systems since they do not offer a user-tailored dialogue. The root cause of this dichotomy is that the goals of (1) consultative dialogue with a user and (2) efficient reachability queries place differing demands on the underlying data language: declarativeness for user queries to the system, imperativeness for system queries to the user. At their extremes these goals become mutually exclusive, and so a careful balancing is needed to support both. Our acyclic production systems have both a procedural and a declarative interpretation, and this forms the basis for their dual modes of operation.

The sections of this paper are organized as follows. First, we present a model for a generic class of consultation systems, which abstracts from their internal mechanisms and regards them as black-boxes that input answer values and output conclusions. Then we define the reachability queries over this generic class of systems. These queries would be useful for any kind consultation system, but in general they are undecidable. Second, we describe the thinksheet production systems ("thinksheets"). Third, we analyze the query problem for thinksheets, and present a logic-based solution. Fourth, we give an algorithm that implements this solution by transforming thinksheet queries into datalog with constraints. Finally, we optimize the evaluation of the resultant datalog systems for thinksheets with em-

bedded decision trees, giving a performance improvement as asserted in the abstract. For a demo of our query system, see www.cs.nyu.edu/tanzer/querythink.

2 A Query Problem for Consultation Systems

We now present a simple model for a class of consultation systems. Let S_p be a consultation system whose input-output behavior implements the expertise encoded in the "program" p. Associated with p is a finite set of variables V_p, each of which represents a potential question to the user, and a finite set C_p of conclusions C_p that may be reached by S_p. Let D_p be a universe of discourse comprising the values that can be assigned to the variables in V_p.

A *binding* is a mapping b: $V_p \rightarrow D \cup \{\perp\}$. The *attainable* bindings A_p for S_p are the ones which can actually occur in a single run of the system S_p. For instance, if S_p first asks "What is your Age?", and then "What is your Pulse?", then $\{Age = 50, Pulse = 12\}$ is attainable, but $\{Pulse = 12\}$ is not. We formalize the behavior of S_p by the functions say_p: $A_p \rightarrow 2^{C_p}$, ask_p: $A_p \rightarrow 2^{V_p}$, where, for an attainable binding b representing the user inputs, $say_p(b)$ is the set of conclusions asserted by S_p, and $ask_p(b)$ is the set of questions asked by S_p.

We now define the query problem.

Definition 1 (q-Reachability, q-Inevitability). *For a conclusion $c \in C_p$ of the consultation system S_p, and a logic formula q:*

- *if $\exists b \in A_p$ such that $ask_p(b) = \{\}$, q is true in b, and $c \in say_p(b)$, then c is said to be* reachable under q, *and b is a* witness *to this reachability;*
- *if $\forall b \in A_p$ s.t. $ask_p(b) = \{\}$, q is true in b implies that $c \in say_p(b)$, then c is said to be* inevitable under q.[1]

Given a query formula q, we want a query tool to partition the conclusions into three categories: unreachable under q, reachable but not inevitable under q, inevitable under q. Also, for each reachable conclusion we want a witness binding.

The problem of finding witnesses is one of abductive reasoning, which reasons from effects to possible explanations[5, 1]. Whereas in abductive logic programming one seeks witnesses to the reachability of conclusions in a non-consultative kind of expert system—logic programs—we apply the abductive problem to consultative expert systems.

[1]Here we do not assume that the consultation systems are monotonic, i.e., that $b_1 \subseteq b_2 \Rightarrow say(b_1) \subseteq say(b_2)$; the condition $ask(b) = \{\}$ indicates only that we are interested in the permanent conclusions of the system. In the next section we apply this generic query problem to a kind of monotonic consultation system.

3 Thinksheet Production Systems

3.1 Overview

The reachability queries are uncomputable for general production systems, because the undecidable problem of whether a program halts can be framed as a query about whether a conclusion 'halt' is reachable by a production system that simulates the program.[2] Hence, in order to compute the queries we must restrict the production systems.

Thinksheet production systems T have two kinds of rules: *conclusion rules* of the form "if precondition then print a conclusion", and *question rules* "if precondition then allow the user to assign a value to a variable". For each variable there is one rule that assigns to it. Moreover, we assume there are no cycles of dependencies, where a dependency from rule R_1 to rule R_2 means: R_1 is a question rule that assigns to v, and v is referenced in the precondition of R_2.

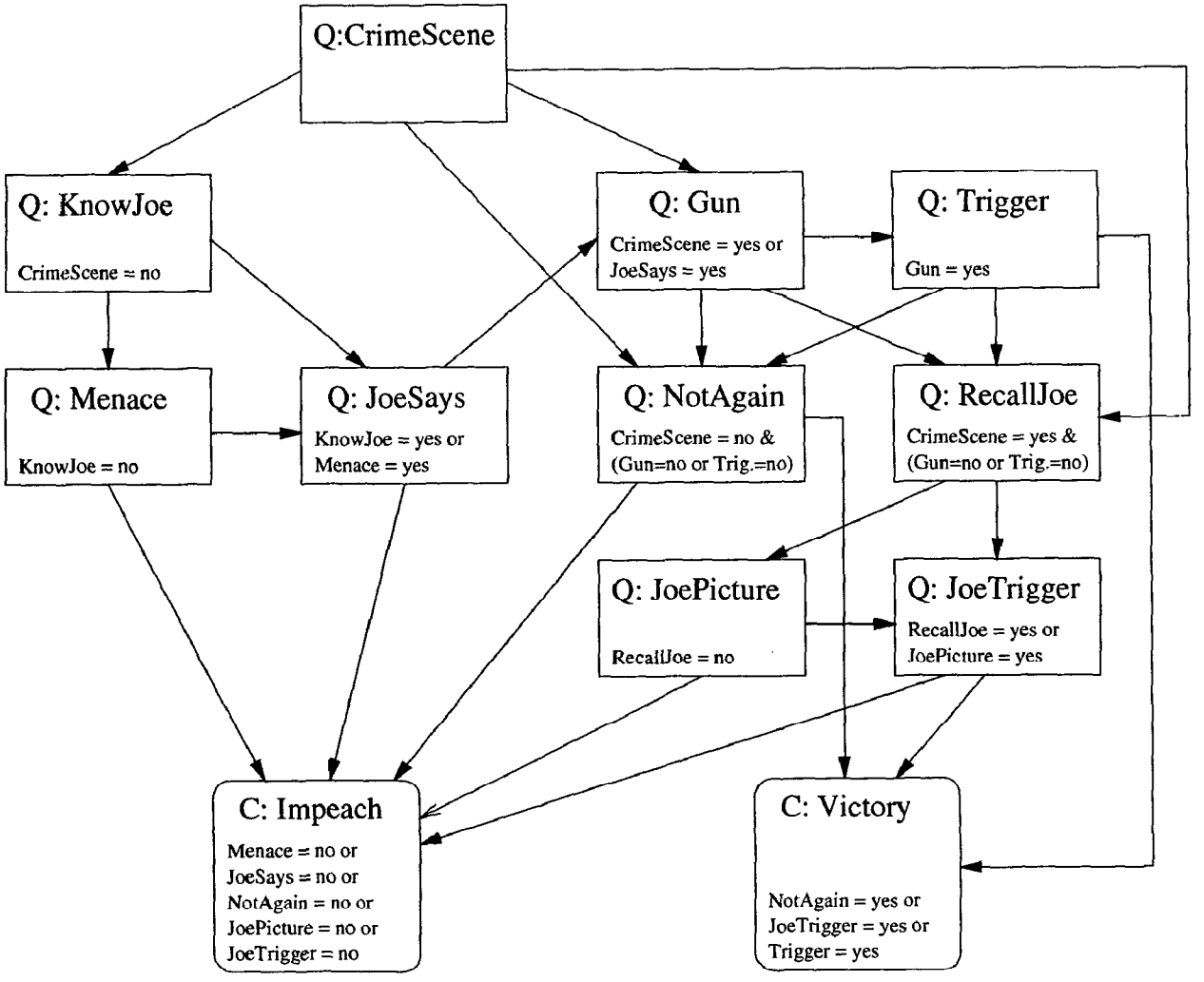
The rule system works with a current binding b which contains the assignments made by the user; initially, b is $\{\}$. The rules that fire are those with precondition formulas that evaluate to true in b. Hence at the outset the only questions answerable are those with preconditions that are constant 'true'. Thus, the user starts with questions at the roots of the dependency graph, and, as preconditions become true, questions further into the DAG become answerable. The conclusion set $say_T(b)$ consists of the conclusions whose preconditions are true in b, and $ask_T(b)$ is the set of variables assigned by question rules with preconditions that are true in b. We do not use a conflict resolution strategy, because we want all of the questions with true preconditions to be simultaneously answerable by the user.

Example 1 (Courtroom Strategy Thinksheet). *Figure 1 shows a strategy for a lawyer who questions a witness and reports conclusions to a jury. Here is an example of a useful query: If the witness denies having a Gun, then is the conclusion Victory reachable? In other words, is Victory q-reachable, where q = 'Gun = no'?*

This example shows that thinksheet generalize the notion of decision trees: when the preconditions of the non-root variables are atomic formulas comparing variables with constants, then thinksheets specialize to decision trees.

Even though our query problem is, in a broad sense, one of backwards reasoning (because it seeks prior conditions that lead to conclusions) the specific technique of backward chaining through the production rules[3, 10] doesn't solve the problem. The reason is that the consequents of the rules do not set the variables to specific literal values, but rather, they input unspecified

[2]The production system has a variable for the program counter, and one rule per program statement.



Q: CrimeScene Were you at the corner of Warner and Tampa on the night of April 11, 1991?

Q: Gun Did you have a gun?

Q: JoePicture Here are pictures of you with Joe just before and after the crime occurred. Now do you remember being with Joe?

Q: JoeSays Joe says you were there. Now do you remember?

Q: JoeTrigger Joe remembers that you pulled the trigger. Now, do you remember as well?

Q: KnowJoe Here is a picture of the key witness Joe. Do you know this man?

Q: Menace Here is a picture of you and Joe together. Now do you remember who Joe is?

Q: NotAgain Mr. Smith, you've already lied when you denied being at the crime scene. Joe said that you pulled the trigger. Did you?

Q: RecallJoe Do you recall that Joe was with you?

Q: Trigger Did you pull the trigger?

C: Impeach Ladies and Gentlemen of the jury, the witness should have lost all credibility.

C: Victory Ladies and Gentlemen of the jury, the witness has confessed to the crime.

The graph depicts the question and conclusion rules comprising the sample production system. There is one rule per node, the precondition of which is shown in smaller text. An edge $v \rightarrow w$ means the precondition of w refers to v, e.g., the precondition of Gun refers to CrimeScene. The complete rule for Gun is: *if CrimeScene=yes or Menace=yes then print "Did you have a gun?" and accept a yes/no answer for the variable Gun.*

Figure 1: Thinksheet Production System for a Lawyer's Courtroom Strategy

values from the user. Consider, for example, the search for a witness to the reachability of a conclusion c appearing as the consequent of the rule: *if* $x > 1.3$ *then assert(c).* Backward chaining takes us from c to $x > 1.3$, but then we face the subgoal of instantiating x in order to build up the witness. The one rule that assigns to x, however, provides no help, because it reads an unspecified value for x. If the domain of x is finite, then we are forced to pick a value for x and make a choicepoint, and if it is infinite we are stuck. Evidently, the problem calls not for the "local" technique of backward chaining, but rather, for a global, constraint-based algorithm.

3.2 Technical Definition of Thinksheets

Domains of Constraints We use a particular kind of constraint domain; see [4] for a more general treatment.

Let $\Theta = (\Sigma, D, I, K)$ be a "constraint domain" comprising: a signature Σ of constants and function and predicate symbols of fixed arities; a universe of discourse D; an assignment I from the constants in Σ to members of D, from the function symbols in Σ to functions over D, and from the predicate symbols in Σ to relations over D; and, a set K of formulas called *constraints* constructed from variables, symbols in Σ, and the conjunction and disjunction operators $\wedge, \vee$. We assume that K is closed under conjunctions, and that satisfiability of formulas in K is decidable. Finally, we assume that for each predicate p in Σ, there is an *opposite* predicate $\overline{p}$ of the same arity, and that I interprets p,$\overline{p}$ so that $p(t_1, ..., t_n)$ is true iff $\overline{p}(t_1, ..., t_n)$ is false.[3]

A formula is *primitive* in K if it cannot be expressed by a conjunction of formulas in K. We therefore may characterize the constraint set K by specifying the subset of primitive constraints.

Example 2 (Θ_{rlo}: Rational Linear Order). *D = the set of rational numbers; $\Sigma = \{=, \neq, <, \geq, >, \leq, +, -, *\} \cup$ {constants for individual rational numbers}; the interpretation of these symbols by I is standard; the primitive constraints in K are equalities and inequalities between linear polynomials with rational coefficients.*

Example 3 (Θ_{uof}: Univariate Order Formulas). *D = an ordered set; $\Sigma = \{=, \neq, <, \geq, >, \leq\}$ and constants; I gives the ordering of D; the primitives in K are formulas built from $\Sigma \cup \{\wedge, \vee\}$ and* one *variable. For instance, if $D = \{january, \ldots, december\}$, then $(Month < may) \vee (Month = nov)$ is primitive, and $(Month < may) \wedge (Year \neq 1900)$ is a conjunction of two primitives.*

We have implemented Θ_{uof} in our prototype, because it can be used to implement consultation systems that are both useful and efficiently queryable. It offers a a generalization from the constraint domain implicit in classical relational databases, where formulas of the form $(attr_1 = val_1) \wedge \ldots \wedge (attr_k = val_k)$ can represent records (c.f.[6]). The Θ_{uof} constraints can be processed efficiently, by representing the solutions to a primitive constraint by an ordered list of disjoint intervals, and the solutions to a conjunction of primitives by a family of these lists. These lists can be conjoined and disjoined by simple linear-time algorithms.

We now formalize the concept of thinksheet production systems.

Definition 2 (Thinksheet).
A thinksheet *T is a tuple $(C_T, V_T, indom, pre)$:*

- *C_T is a finite set of conclusions, each with an associated text;*
- *V_T is a finite set of variables, each with an associated question text;*
- *$\forall v \in V_T$, $indom_v$ is a constraint or a disjunction of constraints with one free variable v;*
- *$\forall w \in V_T \cup C_T$, pre_w is the* precondition *of w, a formula with variables in V_T, built from conjunctions and disjunctions of constraints.*

We stipulate that a thinksheet has an acyclic dependency graph *G_T, which we define as the directed graph with nodes $V_T \cup C_T$ and edges $v \rightarrow w$ whenever v occurs in pre_w.*

We illustrate this notation with Figure 1:
C_T = {Impeach,Victory}
V_T = {CrimeScene, ... ,Trigger}
$indom_{Gun}$ = '$Gun = yes \vee Gun = no$', ...
pre_{Gun} = '$CrimeScene = yes \vee JoeSays = yes$', ...

The production system for T has one rule for each conclusion $c \in C_T$, viz., "if pre_c then print the text for c", and one rule for each variable $v \in V_T$, viz., "if pre_v then print the question for v and allow the user to assign a value to v". In the rule for v, the user's answer choices are limited to dom_v, the domain of v, which comprises the set of values x such that the binding $\{v = x\}$ satisfies $indom_v$. Typically, $indom_v$ will assume simple forms, e.g., '$v = 1 \vee v = 2$', '$v > 0$', '$v = v$', which represent, respectively, the domains $\{1, 2\}$, $\{x \in D \mid x > 0\}$, and D.

Table 1 shows a simple thinksheet T_{xyz} that will be use as the running example for our query analysis.

4 Solution to the Thinksheet Query Problem

Theorem 1 will solve the query problem in the abstract, by reducing the thinksheet queries to formula satisfaction problems. The statement of this theorem involves

[3] The assumption that the formulas in K are negation-free does not lose generality, since the negation of a constraint in K is equivalent to a disjunction of constraints in K, e.g., replace $not(x > 2 \wedge y = 4)$ by $(x \leq 2) \vee (y \neq 4)$.

var v	dom_v	$indom_v$	pre_v
x	$\{0,1\}$	$x=0 \vee x=1$	*true*
y	$\{0,1\}$	$y=0 \vee y=1$	$x=0$
z	$\{0,1\}$	$z=0 \vee z=1$	$y=0$

$G_{T_{xyz}}$: $x \to y \to z$

Table 1: Thinksheet T_{xyz}

formulas that we will define called *pos*,*neg*. The semantics of these formulas will be explored in the proof of the theorem. As a preliminary step, we now define the satisfaction of formula by a binding that may contain $\perp$ values.

Definition 3 ($\models$). *For bindings b and formulas f, the relation $b \models f$, called b* satisfies *f, is defined as follows:*

- *$(b \models f_1 \vee f_2) \iff (b \models f_1$ or $b \models f_2)$*
- *$(b \models f_1 \wedge f_2) \iff (b \models f_1$ and $b \models f_2)$*
- *$(b \models p(t_1,t_2)) \iff$*
  $\forall v$ in $vars(t_1) \cup vars(t_2)$, $b(v) \neq \perp$, and, $(val(t_1,b), val(t_2,b))$ is in the relation for predicate p, where $val(t,b)$ is the value of term t in binding b.[4]

For examples: $\{x=1, y=2\} \models x \leq y$, $not(\{x=1\} \models x \leq y)$, $\{x=1\} \models x \geq 0 \vee y = 5$. We next define a kind of binding that conforms to the logical structure of a thinksheet.

Definition 4 (Solution Binding for T).
A solution *for T is a binding b such that $\forall v \in V_T$:*

$$\text{if } b \models pre_v \text{ then } b(v) \in dom_v \text{ else } b(v) = \perp$$

Proposition 1. *For bindings b and thinksheets T, b is a solution for T iff b is attainable and $ask_T(b) = \{\}$.*

Definition 5 ($pos(e), neg(e)$).
For an expression e, define the formulas $pos(e), neg(e)$ by Figure 3, using the syntax in Figure 2. Figure 4 illustrates pos, neg for thinksheet T_{xyz} (Table 1).

Term	t	::=	$v \mid c \mid f(t_1,\ldots,t_n)$
Formula	ϕ	::=	$true \mid false \mid p(t_1,\ldots,t_n)$ $\mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$
Expr	e	::=	$t \mid \phi$

Legend: t=term, ϕ=formula, e=expression, v=var, c=const, f=function, p=predicate, $\overline{p}$=opposite predicate.

Figure 2: Syntax Used in Fig. 3

[4] E.g., $val(y+1, \{x=3, y=5\}) = 6$.

pos	:	*Expr* $\to$ *Formula*
$pos(v)$	=	$pos(pre_v) \wedge indom_v$
$pos(c)$	=	*true*
$pos(true)$	=	*true*
$pos(false)$	=	*false*
$pos(f(t_1,\ldots,t_n))$	=	$pos(t_1) \wedge \ldots \wedge pos(t_n)$
$pos(p(t_1,\ldots,t_n))$	=	$pos(t_1) \wedge \ldots \wedge pos(t_n)$ $\wedge p(t_1,\ldots,t_n)$
$pos(\phi_1 \wedge \phi_2)$	=	$pos(\phi_1) \wedge pos(\phi_2)$
$pos(\phi_1 \vee \phi_2)$	=	$pos(\phi_1) \vee pos(\phi_2)$
neg	:	*Expr* $\to$ *Formula*
$neg(v)$	=	$neg(pre_v)$
$neg(c)$	=	*false*
$neg(true)$	=	*false*
$neg(false)$	=	*true*
$neg(f(t_1,\ldots,t_n))$	=	$neg(t_1) \vee \ldots \vee neg(t_n)$
$neg(p(t_1,\ldots,t_n))$	=	$neg(t_1) \vee \ldots \vee neg(t_n) \vee$ $(pos(t_1) \wedge \ldots \wedge pos(t_n)$ $\wedge \overline{p}(t_1,\ldots,t_n))$
$neg(\phi_1 \wedge \phi_2)$	=	$neg(\phi_1) \vee neg(\phi_2)$
$neg(\phi_1 \vee \phi_2)$	=	$neg(\phi_1) \wedge neg(\phi_2)$

Figure 3: $pos(e)$ and $neg(e)$

$pos(x)$	=	$(x=0 \vee x=1)$
$pos(y)$	=	$pos(x) \wedge (x=0) \wedge (y=0 \vee y=1)$
	=	$(x=0) \wedge (y=0 \vee y=1)$
$pos(z)$	=	$pos(y) \wedge (y=0) \wedge (z=0 \vee z=1)$
	=	$(x=0) \wedge (y=0) \wedge (z=0 \vee z=1)$
$neg(x)$	=	*false*
$neg(y)$	=	$neg(x) \vee (pos(x) \wedge x \neq 0)$
	=	$(x=1)$
$neg(z)$	=	$neg(x) \vee (pos(y) \wedge y \neq 0)$
	=	$(x=1) \vee (x=0 \wedge y=1)$

Figure 4: pos, neg for variables in T_{xyz}

Theorem 1. *Let T be a thinksheet, $c \in C_T$ a conclusion with precondition p, q a formula with variables in V_T, and S the solution bindings for T. Then:*

$$c \text{ is } q\text{-reachable} \iff \exists b \in S: (b \models q) \text{ and } (b \models p) \iff pos(q) \wedge pos(p) \text{ is satisfiable}$$

$$c \text{ is } q\text{-inevitable} \iff \forall b \in S: (b \models q) \Rightarrow (b \models p) \iff pos(q) \wedge neg(p) \text{ is unsatisfiable}$$

Proof. See www.cs.nyu.edu/tanzer/querythink. □

Example 4. *We solve three queries to T_{xyz}. For $i = 1,2,3$, q_i is a query formula, and p_i is the precondition of a conclusion:*
q_1 = '$x=0$', q_2 = '$z=1$', q_3 = '$x=1 \wedge y=1$';
p_1 = '$y=0$', p_2 = '$x=0$', p_3 = '$y=0$'. Then:
$pos(q_1) \wedge pos(p_1)$ is satisfiable, so p_1 is q_1-reachable;
$pos(q_1) \wedge neg(p_1)$ is satisfiable, so p_1 not q_1-inevitable;
$pos(q_2) \wedge pos(p_2)$ is satisfiable, so p_2 is q_2-reachable;
$pos(q_2) \wedge neg(p_2)$ is unsatisfiable, so p_2 is q_2-inevitable;
$pos(q_3) \wedge pos(p_3)$ is unsatisfiable, so p_3 not q_3-reachable.

Now we state how to compute the witnesses.

Theorem 2. *Suppose that conclusion c is q-reachable, and, as per Theorem 1, let b be a binding satisfying $pos(p) \wedge pos(q)$, where p is the precondition of c. Let S be the set of sub-bindings of b that satisfy $pos(q) \wedge pos(p)$, and let b_0 be minimal in S, where S is ordered by the inclusion relation. Then b_0 is a witness to the q-reachability of c.*

Example 5. *In Example 4, $pos(q_1) \wedge pos(f_1)$ is satisfied by the one minimal binding $\{x = 0, y = 0\}$, which is clearly a witness to the q_1-reachability of f_1.*

Example 6 (Solving the Query in Example 1). *By Thm. 1, Victory is reachable under query 'Gun=no', because $pos(Gun = no) \wedge pos(pre_{victory})$ is satisfied by the binding {CrimeScene=yes, Gun=no, RecallJoe=yes, JoeTrigger=yes}. Since it is minimal among bindings satisfying that formula, it is a witness to the (Gun=no)-reachability of Victory.*

5 Query Algorithm for Thinksheet

Our algorithm has the following aim: to *query an entire thinksheet T with the formula q*, by which we mean to determine the q-reachability and q-inevitability of every precondition in T. The first part of the algorithm implements the definitions of *pos* and *neg* (Figure 3) in order to transform thinksheet queries into datalog systems. Theorem 1 proves the correctness of this transformation. The second part of the algorithm is an optimization for evaluating the resultant datalog systems.

5.1 Transformation of Thinksheet Queries into Datalog with Constraints

An easy consequence of Thms. 1, 2, is that the query results follow from the solutions to the formulas $\{pos(v) \wedge pos(q), neg(v) \wedge pos(q) \mid v \in V_T\}$.

Observe that these formulas share common subformulas, because whenever the precondition of y references x, then $pos(y)$, $neg(y)$ will involve $pos(x), neg(x)$ as subformulas. For instance, in T_{xyz}, $x \rightarrow y \rightarrow z$, and, as shown in Figure 4, $pos(y)$ contains $pos(x)$ as a subformula and $pos(z)$ contains $pos(y)$.

A more compact representation for this set of overlapping formulas F is the DAG in which the common subformulas are shared, which we call the *logic graph* LG for F. For instance, the logic graph for T_{xyz} is represented by the six partial expansions shown in Figure 4: $pos(x) = (x = 0 \vee x = 1)$, $pos(y) = pos(x) \wedge (x = 0) \wedge (y = 0 \vee y = 1)$, ... The roots of LG are the constraints appearing in these partial expansions, and the other nodes are the *AND* and *OR* "gates" that correspond to the conjunction and disjunction operators introduced in the definitions of pos, neg (Figure 3).[5] For example, from the partial expansion for $pos(y)$ we see that the node in LG for $pos(y)$ is an *OR* node which has three children: (1) the output of the node for $pos(x)$, (2) a root node labeled with constraint $(x = 0)$, and (3) a root node labeled with $(y = 0 \vee y = 1)$. Every node n in LG therefore represents an *AND–OR* combination of constraints at the roots that are ancestors of n. A logic graph can be expressed by a nonrecursive system of datalog rules, as illustrated in Figure 5.

$pos_x(X, Y, Z)$:- $(X = 0 \vee X = 1)$.
$pos_y(X, Y, Z)$:- $pos_x(X, Y, Z)$, $X = 0$, $Y \in \{0, 1\}$.
$pos_z(X, Y, Z)$:- $pos_y(X, Y, Z)$, $Y = 0$, $Z \in \{0, 1\}$.
...

Figure 5: Some of the Datalog Rules for Figure 4

We base our evaluation on a standard forward chaining ("bottom-up") approach: the solutions to the node n in LG are represented by an attribute *n.solutions* which holds a *constraint store* comprising a set of conjunctions of constraints, and which means the disjunction of these conjunctions; the node $OR(p_1, p_2)$ is evaluated as $p_1.solutions \cup p_2.solutions$; $AND(p_1, p_2)$ is evaluated as $\{t_1 \wedge t_2 \mid t_1 \in p_1.solutions, t_2 \in p_2.solutions\}$.

The general complexity is exponential in the worst case,[6] which is a normal result since the query problem is NP-hard: we can encode a conjunctive normal formula as the precondition of a conclusion, and then determine its satisfiability by the reachability of the conclusion. Nevertheless, there is an important special case which is tractable: when the preconditions are conjunctions of constraints. The resultant logic graphs will have disjunction nodes (arising from the *neg* formulas), but their outputs are not conjoined, and so there is no combinatorial explosion.

5.2 Optimization for Embedded Decision Trees

We now optimize the evaluation of those datalog rules which arise from embedded decision trees in the thinksheet dependency graph. The optimization does does not apply to inevitability queries, and hence by Theorem 1 it needs to solve only the *pos* nodes.

Definition 6 (Embedded Decision Tree). *Define $v \in V_T$ to be a* (decision) tree variable *if pre_v is a single constraint with one free variable, called* parent(v). *An* embedded tree *is a set of variables $\tau \subseteq V_T$ containing a "pseudo-root" r, such that $\forall v \in \tau - \{r\}$, v is a tree variable, and $parent(v) \in \tau$.*

[5] No negation operators are introduced in the definitions of *pos* and *neg*.

[6] Here we abstract from the domain-specific complexity of conjoining constraints, by viewing the algorithm as a reduction from queries to conjunctions of constraints.

Proposition 2. *For a tree variable v,*
$pos(v) = pos(parent(v)) \wedge pre_v \wedge indom_v$.

We make the assumption that for all tree variables v, $pre_v \wedge indom_{parent(v)}$ is satisfiable.[7]

The algorithm first computes the solutions for $pos(q)$ using forward propagation. If $pos(q)$ is unsatisfiable, then it terminates and reports that all conclusions are q-unreachable. Otherwise, the other *pos* nodes are evaluated by forward propagation, except that: whenever r is the root of an embedded tree τ in G_T, then once forward propagation has computed the solutions to $pos(q) \wedge pos(r)$, the routine *query_tree*(r), shown in Figure 6, is called to query the variables in τ. The routine makes temporary updates—incremental conjunctions—to a global constraint store which is initialized to the solutions to $pos(q) \wedge pos(r)$. It calls a recursive subroutine that destructively conjoins the precondition for the current node in the tree. After the call to *query_tree*(τ) has completed, then for leaves z of τ that are not leaves of G_T, the solutions to $pos(z)$ are computed by conjoining to $pos(r)$.*solutions* the precondition and domain constraints for the variables on the path from r to z. Then, forward propagation resumes from these leaves z.

Our routine queries the trees more efficiently than existing datalog algorithms—top-down[4] and bottom-up (see e.g., [12, 9])—for constraint domains Θ where the cost of destructively conjoining one constraint into a conjunction of n constraints is smaller than n. The general datalog algorithms must of course do $\Omega(w)$ work per node, where w is the number of constraints representing the solutions to that node. Moreover, in the worst case w is $\Theta(n)$, and so the worst-case complexity for querying all the nodes is $\Omega(n^2)$. For inexpensive constraint domains, we are able to avoid this quadratic cost, by taking advantage of our knowledge of the special forms assumed by the datalog systems arising from embedded trees in the original thinksheet. In particular, we don't need all the solutions to the intermediate nodes in the rule system, but rather, we just need to know the satisfiabilities of some of the predicates. Hence we are able to use efficient update operations that lose information we don't need.

Lemma 3 (Used by Algorithm). *Let q be a formula, and $x_0, \ldots, x_k$ be a sequence of variables such that for $i = 1, \ldots, k$, x_i is a tree variable with parent x_{i-1}, and $pre_{x_i} \wedge indom_{x_{i-1}}$ is satisfiable. Let ϕ be the formula $pos(q) \wedge pos(x_0) \wedge pre_{x_1} \wedge \ldots \wedge pre_{x_k}$, and let ϕ' be $\phi \wedge indom_{x_1} \wedge \ldots \wedge indom_{x_k}$. Then:*
ϕ is satisfiable $\iff$ ϕ' is satisfiable.

For the following complexity result, we make two plausible assumptions about the the algorithm for conjoining constraints. First, when the solver is asked to conjoin $(c_1 \wedge c_2) \wedge c_1$, then the result will not be larger than the equivalent $c_1 \wedge c_2$. Second, the cost of conjoining $(c_1 \wedge \ldots c_k) \wedge c$ will not exceed the cost of conjoining $(c_1 \wedge (c_2 \wedge \ldots (c_k \wedge c)))$. These assumptions hold for our constraint domain Θ_{uof}.

[7] This is a reasonable assumption, because if this formula were unsatisfiable, then pre_v would be unreachable under all queries, and so v would be useless.

```
global cstore
query_tree(variable r)
    // r is root of embedded tree τ
    let n be the node for pos(q) ∧ pos(r);
    cstore := n.solutions;
    for each child v of r do
        if v is a tree variable then
            query_subtree(v);
endproc
query_subtree(variable v)
    cstore.conjoin(pre_v);
    // Lemma 3: indom_v needn't be conjoined
    v.reachable := cstore.satisfiable();
    for each child w of v do
        if w is a tree variable then
            query_subtree(w);
    cstore.pop(); // undo conjoin(pre_v)
endproc
```

Figure 6: Subroutine *query_tree*

Theorem 3 (Complexity w.r.t Trees). *Let C be a class of thinksheets such that $\forall T_1, T_2 \in C$, $trim(T_1) = trim(T_2)$, where $trim(T_i)$ is the part of T_i obtained by removing all tree variables. Let q be a fixed formula such that $\forall T \in C$, $vars(q) \subseteq V_T$. Let $f(n)$ be the cost of destructively conjoining a constraint of unit size into a conjunction of n constraints. Let n be the size of the T. Then our data complexity for q-reachability queries as T ranges over C is $\Theta(n \cdot f(n))$, whereas the complexity for existing algorithms is $\Theta(n \cdot f(n) + n^2)$.*

The unit size constraints are the preconditions in the trees, which are branch tests of the form $(v\ Relop\ c)$ between variables v and constants c. For the Θ_{uof} constraint domain, the incremental conjunction operator can be performed with cost $f(n) = O(log(n))$ (because a conjunction of these constraints is representable by a family of ordered lists of intervals, and each list can be maintained by a balanced-tree data structure). Then our data complexity w.r.t. trees is $\Theta(n \cdot log(n))$, compared to $\Theta(n^2)$ for existing algorithms. A more restrictive domain has *atomic* univariate order formulas as primitives. Then $f(n) = O(1)$, and our complexity is $\Theta(n)$, compared to $\Theta(n^2)$.

Example 7. *We apply the query $q =$ 'z = 1' to T_{xyz}, using the optimized algorithm. The variables y and z are queried with two constant time update operations. Existing datalog algorithms must compute the conjunc-*

tion for pos(y), and the conjunction for pos(z), and so five operations are needed.

6 Conclusion

There is a split among existing computer-based information systems. On the one hand, databases offer no guidance help a naive user navigate through the mass of information that they contain. Existing consultation systems, on the other hand, do offer a user-tailored dialogue, but the user is reduced to the passive condition of answering questions and being told things. Ideally, the user should be able to actively ask a consultation system about the knowledge that it contains.

There is a conflict between the goals of the system being consultative and being efficiently queryable: as the computational abilities of the consultation system increase, the querying problem becomes more difficult. For general production systems, it is undecidable, and so we focussed on a sublanguage of acyclic production systems that are efficiently queryable. Although restricted, this sublanguage still can express a wide range of useful consultation systems. In addition, these restrictions simplify the user model so that it can be used with a spreadsheet-like interface familiar to a wide range of computer users.

These considerations raise some interesting theoretical questions. What are the maximally expressive consultation system languages that can be queried within a given complexity class? For a given class of consultation system applications, which languages have minimal query complexity?

A prototype of our queryable consultation system is available on the web.[8] Its general significance is that, if we liken a typical consultation system to a doctor who questions patients and makes diagnoses, we can now ask the doctor some questions in order to learn something about medical reasoning.

Acknowledgements We thank the anonymous reviewers for their insightful comments, and also Gediminas Adomovicas, Len Bloch, Deepak Goyal, Peter Piatko, Archisman Rudra, Sarah Talbot, Ken Tanzer, and Zhe Yang for excellent technical discussions and editorial assistance.

References

[1] L. Console, D.T. Dupre, and P. Torasso. On the relationship between abduction and deduction. *J. Logic Computat.*, 1(5):661–690, 1991.

[2] V. Gaede, A. Brodsky, O. Gunther, D. Srivastava, V. Vianu, and M. Wallace, editors. *Constraint Databases and Applications*, volume 1191, 1997.

[3] P. Jackson. *Introduction to Expert Systems.* Addison-Wesley, second edition, 1990.

[4] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[5] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *J. Logic Computat.*, 2(6):719–770, 1992.

[6] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.

[7] P. Piatko. *Thinksheet: A Tool for Information Navigation.* PhD thesis, New York University, 1998.

[8] P. Piatko, R. Yangarber, D. Lin, and D. Shasha. Thinksheet: A tool for tailoring complex documents. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, page 546, 4–6 June 1996.

[9] P. Revesz. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116:117–149, 1993.

[10] Teknowledge. *S.1 Reference Manual.* Palo Alto CA: Teknowledge, 1985.

[11] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[12] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[13] C.T. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications.* Morgan Kaufmann, 1998.

[8] See http://www.cs.nyu.edu/tanzer/querythink.