Queryable Expert Systems

by

David Tanzer

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2001

Dennis Shasha

To Sarah

Acknowledgments

Many thanks go to my advisor Dennis Shasha, who was my unswerving guide. And to my wife Sarah, for her goodness, love and devotion. And to Gediminas Adomovicas, Len Bloch, Deepak Goyal, Peter Piatko, Archisman Rudra, Ken Tanzer, and Zhe Yang for excellent technical discussions and editorial assistance.

Abstract

Interactive rule-based expert systems, which work by "interviewing" their users, have found applications in fields ranging from aerospace to help desks. Although they have been shown to be useful, people find them difficult to query in flexible ways. This limits the reusability of the knowledge they contain. Databases and noninteractive rule systems such as logic programs, on the other hand, are queryable but they do not offer an interview capability. This thesis is the first investigation that we know of into query-processing for interactive expert systems.

In our query paradigm, the user describes a hypothetical condition and then the system reports which of its conclusions are reachable, and which are inevitable, under that condition. For instance, if the input value for bloodSugar exceeds 100 units, is the conclusion diabetes then inevitable? Reachability problems have been studied in other settings, e.g., the halting problem, but not for interactive expert systems.

We first give a theoretical framework for query-processing that covers a wide class of interactive expert systems. Then we present a query algorithm for a specific language of expert systems. This language is a restriction of production systems to an acyclic form that generalizes decision trees and classical spreadsheets. The algorithm effects a reduction from the reachability and inevitability queries into datalog rules with constraints. When preconditions are conjunctive, the data complexity is tractable. Next, we optimize for queries to production systems that contain regions which are decision trees. When general-purpose datalog methods are applied to the

rules that result from our queries, the number of constraints that must be solved is $O(n^2)$, where n is the size of the trees. We lower the complexity to $O(n)$. Finally, we have built a query tool for a useful subset of the acyclic production systems. To our knowledge, these are the first interactive expert systems that can be queried about the reachability and inevitability of their conclusions.

Contents

List of Figures

Chapter 1

A Query Paradigm for Interactive Expert Systems

1.1 The General Query Problem

1.1.1 Motivation and Overview

In a commentary on the reasons why expert systems have not been as widely applied as databases have, Feigenbaum remarked that whereas the knowledge in databases can be utilized by many different applications, the knowledge in expert systems tends to be bound up with specific application programs [35]. This comparative lack of reusability in part is due to the rigid form of dialogue that these systems offer to their users (and to application programs). Unlike databases, which let the user question the system in active and creative ways, existing interactive expert systems *query the user* according to a fixed logic, thereby reducing the user to the passive role of answering questions and being told things. The flexible queryability of

databases means that we can learn by posing *research questions* to the knowledge they contain, but with existing interactive expert systems, this is not possible, because *they* direct the course of the dialogue with the user.[1]

Ideally, we want a knowledge-based information system that can present its knowledge in either of two complementary modes of operation: (1) a query mode, where it can be flexibly queried as a database of expertise, and (2) a consultation mode, where it provides a user-tailored dialogue to guide a non-technical through the information it contains. In other words, we would like to have a *queryable expert system.*

Existing knowledge-based information systems [36, 15] fall short of this two-sided ideal. Databases [42, 46, 14, 21], and, more generally, declarative knowledge bases, e.g., logic programs [37], are queryable but are not intended to query a user; existing interactive expert systems [15] query a user, but are not queryable in flexible ways. Non-interactive expert systems, which autonomously reach conclusions by directly reading a knowledge base (e.g., offline weather prediction) are neither querying nor queryable. Expert-assistant programs, such as the classic Dendral [15], provide a toolbox of expertise-based functions, but the use of these functions is no more a matter of flexible querying than is the use of the functions on a calculator.

Our goal is to explore conditions under which such a two-sided queryable system systems is feasible, and to implement a proof-of-concept prototype. We must begin by specifying what it means for an interactive expert system to be "queryable".

Suppose that we have before us a particular interactive system, which, for the

[1]We use the generic term "expert system" to mean any machine programmed to emulate the reasoning functions of human experts.

sake of illustration, we take to be a machine programmed with some specific rules that enable it to function as a medical consultant on blood abnormalities. This consultant works by prompting a doctor for a series of measurements about a blood sample. Whenever anomalies are found, the system proceeds to pose a chain of data-dependent questions to the doctor, in order to effect a diagnostic process leading to diagnoses of probable illnesses and recommended courses of therapy.

We will formulate two types of queries about this system and the knowledge that it contains: (1) which of the potential conclusions are still reachable by the system if the user inputs are constrained to satisfy some formula? and (2) which of the conclusions will necessarily be reached by the system, if the user inputs are constrained to satisfy a formula? Here is an example of each type of query. First, if the input value supplied for the parameter *bloodsugar* is under 20 units, then is the conclusion *diabetes* still reachable? Second, if the input value for *bloodsugar* exceeds 40 units, then must the conclusion *diabetes* inevitably be reached? Also, if *diabetes* is reachable under the condition '$bloodsugar < 20$', then we may ask for a combination of input values that satisfies the condition and causes *diabetes* to be concluded. We call such an input combination a *witness* to the reachability of the conclusion under that condition.

We envision the query tool working in the following manner. The user (doctor) poses a query in the form a hypothetical condition, as a formula such as '$bloodsugar < 20$'. Implicitly, the question is, given the logic of the rules encoded in that system, what are the ramifications of the query formula? The query tool will reply by grouping the conclusions into one of three categories, which indicate three progressively stronger connections between the query condition and conclu-

sion, namely, conclusions that are (1) unreachable under the query, (2) reachable but not inevitable, or (3) inevitable. In other words, which conclusions are ruled out, which are possible, and which are positively confirmed.[2]

These queries are, unfortunately, hard to solve by algorithmic methods, even when posed to relatively simple systems. They are hard because they are "meta-queries" that ask *about* the behavior of the system, which is regarded as a black-box that maps input values to conclusions. Hence, even though the system may be a simple mechanism, the queries pertain to the *inverse* of the mapping performed by the system, and hence may be undecidable if the number of possible inputs is infinite. To illustrate the difficulty, suppose that the system inputs two values x and y, computes a polynomial function $v = f(x, y)$, and then outputs the conclusion '$output = v$'. Now consider the query: is the conclusion '$output = 0$' reachable if '$x^2 + y^2 = 1$'? This amounts to the question of whether the polynomial f has any roots on the unit circle. Hence, reachability queries to simple systems can be as difficult as determining the mutual satisfiability of algebraic equations.

[2]Of course, the answers to these queries are only as empirically reliable as the rules used to program the system. The answers to the queries reveal the ramifications of the logic of the rules, and hence they can be used to perform "sanity checks" on the reasonableness of the rules. To the extent that the system is sound, the reachability of a conclusion indicates a real fact. If the system is also complete, in the sense that it takes into account all possible causes of the conclusions in its repertoire, then inevitability in the system indicates a real empirical relationship. For an example of a complete expert system, we can envision a troubleshooting system that asks an engineer a series of questions of the form "Is circuit component number N outputting an unexpected value?", and "Is component N physically defective?" in order to find the broken component and so repair the system.

These queries are undecidable for expert system languages that are Turing-complete, or based on precondition formulas whose satisfiability is undecidable. In our pursuit of a queryable expert system, we must search for simple yet useful expert system languages. First, however, we need to define the "interactive expert systems" to which the query problem will apply, and then to give an exact specification of this query problem.

1.1.2 A Generic Formal Model of Interactive Expert Systems

We adopt a *functional view* of expert systems, in which we abstract completely from their internal mechanisms and language constructs, and instead regard them solely from the point of view of their external behavior, i.e., regard them as "black-boxes" that map user inputs into conclusions. We will now examine the form and structure of these mappings.[3]

We will be applying the query problem to various classes of expert systems, where each class contains the systems definable by some language L of programs. Typically, the programs of L will be rule systems, but in the generic formal model that we are about to present, we make few assumptions about the syntax of the programs. L could be, for example, a standard imperative programming language instead of a rule language. In this case, each member of L would be an imperative program that is hard-coded to function as an interactive consultant. We will identify the members of L with the systems s that they define. Here are the assumptions about the systems $s \in L$, which together constitute our formal model for interactive

[3]The following formalism for interactive expert system was presented in [40], and will be summarized in a table at the close of this passage.

systems.

Question Variables V_s

For each system $s \in L$, we assume that there is a finite set of *variables* V_s, and that each question posed by s takes the form of printing some text to the user and then accepting an assignment to a variable in V_s. For instance, print "What is the value for *bloodsugar*?" and then accept a value for *bloodsugar* from the user. Thus, the variables V_s will serve as abstract representatives for the questions in the vocabulary of the system s.

Derived Variables W_s

Let W_s be a finite set of variables, disjoint from V_s, which are called derived because they are assigned values that are calculated by the system.

Conclusions C_s

We denote by C_s the set of conclusions that may be potentially asserted by s. The conclusions C_s might be $\{$*diabetes,hypoglycemia,* $\ldots$ $\}$, in our medical example.[4]

[4]From a minimalist perspective, the constructs of derived variables W_s and conclusions C_s are not both necessary, because: (1) the assertion of e.g. the conclusion *diabetes* can be modeled by the assigning of *true* to the boolean derived variable *diabetes*, and (2) the assignment of z to the derived variable w can be modeled by the assertion of the conclusion $w = z$. The usage patterns that we find in practice, however, make it convenient to have a separate notation for the conclusions. We will typically picture the assertion of conclusions in C_s as the final judgements of the system, whereas the assignments to derived variables will express internal calculations that the system uses to arrive at those judgements.

The Domain of a Variable: dom_x

For each $x \in V_s \cup W_s$, we define dom_x to be the *domain* of x, comprising the defined values that can meaningfully be assigned to x. We let $D_s \equiv \cup_{x \in V_s \cup W_s} dom_x$ be the union of all these domains, called the *universe of discourse* for s. Let $\perp \notin D_s$ be a symbol that stands for "undefined."

Bindings B_s

At any moment in the course of a session with the system, the values assigned to the variables are described by a *binding* b, which is a partial mapping from $V_s \cup W_s$ into D_s, *i.e.*, $b : V_s \cup W_s \rightarrow D_s \cup \{\perp\}$. We use notation of the form $\{pulse = 20, temp = 10\}$ to represent the bindings. We denote the set of bindings for s by B_s. A binding b is *well-typed* if $b(x) \in dom_x \cup \{\perp\}$, for all $x \in V_s \cup W_s$.

Definition 1.1.1 (vars(b))

For a binding $b : V_s \cup W_s \rightarrow D_s \cup \{\perp\}$*, define:*

$vars(b) = \{v \in V_s \cup W_s \mid b(v) \neq \perp\}$.

Attainable Bindings A_s

The course of a consultation session between user and system can be described as the process of building up the "current" binding for the variables. Initially, this binding is empty. Whenever the user assigns a value z to a variable $v \in V_s$, the pair (v, z) gets added to the current binding. Whenever the system assigns z to a variable $w \in W_s$, the pair (w, z) gets added to the current binding.

We say that a binding b is *attainable* for s if it can actually occur as the current binding in some session with s. For example, the empty binding $\{\}$ is attainable

for all s, because $\{\}$ occurs as the current binding at the start of any session. Every attainable binding is well-typed, but not all well-typed bindings are attainable. For instance, if the system first asks the question "What is the pulse?" and then asks "What is the temperature?", then $\{pulse = 20\}$ and $\{pulse = 20,\ temp = 20\}$ would be attainable, but $\{temp = 20\}$ would not be attainable. We denote the set of attainable bindings for s by A_s.

The Conclusion Function say_s

Given the current attainable binding b, define $say_s(b) \subseteq C_s$ to be the set of conclusions that are asserted by s given the data in b. Thus, $say_s : A_s \rightarrow 2^{C_s}$ is the mapping that expresses the conclusion function of the system.[5]

The Questioning Function ask_s

In the current attainable binding b, the variables in $vars(b)$ represent the questions that are already answered by the user. Also, in this binding, the system will pose a set of questions to be answered by the user. Each of these questions is some variable that is outside of $vars(b)$. We denote the set of all these assignable variables by $ask_s(b) \subseteq V_s - vars(b)$. By defining $ask_s(b)$ as a *set* of variables, we can model "user-friendly" systems that allow the user to answer the posed questions in any

[5]At this stage we avoid any assumptions other than those implied by our definitions. In particular, we do not assume anything about the monotonicity of the conclusion mechanism, because this property can be introduced as a secondary distinction: that s is *monotonic* means that $b' \supseteq b \Rightarrow say_s(b') \supseteq say_s(b)$, where $b' \supseteq b$ means that the binding b' is an extension of the binding b. We cast as wide a net as possible in *defining the problem.* Once the problem is defined, we will investigate cases that are solvable.

desired order.

The Calculation Function $calc_s$

Given the current attainable binding b, define $calc_s(b) : W_s \to D_s \cup \{\perp\}$ to be the binding for W_s that contains the assignments to derived variables that are calculated by s from the data in b. We assume that $vars(calc_s(b))$ is disjoint from $vars(b)$—because $calc_s(b)$ is intended to represent the calculations that can be performed but have not yet been performed.[6]

Final Bindings F_s

We define an attainable binding b to be *final* if $ask_s(b) = \{\}$ and $calc_s(b) = \{\}$. When the current binding is final, the system asks no more questions and performs no more calculations, and so the consultation session has ended. We denote the set of final bindings for s by $F_s \equiv \{b \in A_s \mid ask_s(b) = calc_s(b) = \{\}\}$.

We summarize the behavior of the system s by the abstract procedure shown in Figure 1.1. Finally, we use this procedure to give the following precise definition of the attainable bindings.

Definition 1.1.2 (Attainable Bindings A_s) *For a system s, we define A_s to be*

[6]By defining say_s, ask_s, and $calc_s$ as functions on *bindings*, we are tacitly assuming that the mechanisms of s produce the same effects regardless of the order in which the assignments are supplied by the user and by the system. What is actually produced during the course of a session is a *sequence* of assignments. Our treatment could easily be generalized by replacing "binding" with "sequence," but we prefer to work with bindings, because they are declarative, and because we believe that it is undesirable for an expert system to produce different answers from different orderings of the same set of assignments.

the set of bindings that can be assumed by b in the procedure in shown in Figure 1.1.

```
binding b = {}          // the current attainable binding
while ask_s(b) ≠ {} OR calc_s(b) ≠ {}
   if the user supplies value z for some v ∈ ask_s(b) then
      b(v) = z
   endif
   if ∃ (w, z) ∈ calc_s(b) then
      b(w) = z
   endif
   present the conclusions in say_s(b) to the user
end
```

Figure 1.1: Abstract Description of the Interactive System s

Definition 1.1.3 (Terminology for Interactive Expert Systems)

L — *language of systems*

$s \in L$ — *individual system*

V_s — *question variables for* s

W_s — *derived variables for* s

dom_x — *domain for variable* x

D_s — *universe of discourse for* s

A_s — *attainable bindings for* s

F_s — *final attainable bindings for* s

C_s — *potential conclusions for* s

$say_s : A_s \to 2^{C_s}$ — *conclusion function for* s

$ask_s : A_s \to 2^{V_s}$ — *question function for* s

$calc_s : A_s \to (W_s \to D_s)$ — *calculation function for* s

1.1.3 Formal Statement of the General Problem

We now give a formal statement of the query problem, using the terminology that was summarized in Definition 1.1.3. For an illustration of the following definitions, recall the query where the conclusion c is *diabetes*, and the query formula q is '$bloodsugar < 20$'.

For bindings b and formulas q, we write $b \models q$ to mean b satisfies q, *i.e.*, that q evaluates to True in the environment given by b. For instance, $\{pulse = 20,\ temp = 20\} \models$ '$pulse < 40$', because the formula is clearly True in this binding.[7]

Definition 1.1.4 (Reachable, Inevitable and Possible Conclusions)
For a system $s \in L$, recall from Definition 1.1.3 that F_s is the set of final bindings for s. Then for each conclusion $c \in C_s$ and query formula q, we pose these questions:

[7] In defining the relation $b \models f$, some attention needs to be given to the fact that b may contain $\perp$ values. The full definition, to be given in the section called "Groundwork", is based on the intuitive notion of q successfully evaluating to True in b. To illustrate, $\{pulse = 20\} \models$ '$pressure = 99 \vee pulse < 40$', because the second disjunct successfully evaluates to True.

- *Is conclusion c* reachable *under condition q?*

  $\exists\, b \in F_s \ : \ (b \models q) \wedge c \in say_s(b)$

- *Is conclusion c* inevitable *under condition q?*

  $\forall\, b \in F_s \ : \ (b \models q) \Rightarrow c \in say_s(b)$

- *Is conclusion c* possible *under condition q?*

  $(\exists\, b \in F_s \ : \ (b \models q) \wedge c \in say_s(b)) \wedge (\exists\, b \in F_s \ : \ (b \models q) \wedge c \notin say_s(b))$

We abbreviate these relations between formulas q and conclusions c with the following terminology: c is *q-reachable*, *q-inevitable*, *q-possible*. The connections between reachability, inevitability, and possibility are stated in the next two propositions.

Proposition 1.1.1 *For a system s, and formula q, assume that $\exists\, b \in F_s$ such that $b \models q$. Let R, I, P be, respectively, the sets of conclusions that are reachable, inevitable, and possible under q. Then R is the disjoint union of I and P.*

In words, reachable means possible or inevitable, and possible and inevitable are mutually exclusive.[8]

The next proposition, which holds under all conditions, states that q-possibility can be determined from q-reachability and q-inevitability, and so, for the technical development, we may restrict our attention to q-reachability and q-inevitability.

[8]Since our definition of "possible" excludes inevitable, it really means "properly possible," *i.e.*, there are final conditions under which the conclusion arises, and there are final conditions under which it does not arise. When there are no final bindings that satisfy q, then q is inconsistent with the logic of the system, and we have the uninteresting result that no conclusions are reachable, no conclusions are possible, and, vacuously, all conclusions are inevitable.

Proposition 1.1.2

A conclusion is q-possible iff it is q-reachable but not q-inevitable.

Definition 1.1.5 (Witness Binding) *If c is q-reachable, then we call a binding $b \in F_s$ such that $b \models q$ and $c \in say_s(b)$ a* witness *to the q-reachability of c.*

We now give analogous definitions for the reachability, inevitability and possibility of *questions* under a query.

Definition 1.1.6 (Reachable, Inevitable and Possible Questions)
For a system $s \in L$, variable v, and formula q, we say that the question associated with v is:

- q-reachable if $(\exists\, b \in F_s \ : \ (b \models q) \wedge b(v) \neq \bot)$
- q-inevitable if $(\forall\, b \in F_s \ : \ (b \models q) \Rightarrow b(v) \neq \bot)$
- q-possible if *v is q-reachable but not q-inevitable*

Although the querying of questions is important from the point of view of the user and the interface program, for the purposes of analyzing and solving the queries, we are fortunate in that without loss of generality *we may focus our attention exclusively on the querying of conclusions.* We justify this claim by giving a reduction of the problem of querying of questions in a system to the problem of querying of the conclusions in a variant system. For a given system s, let s' be the variant system with the same conclusions as s, plus a set of "dummy" conclusions which will stand for the questions. We construct s' to be the same as s, with the addition that whenever s poses a question, then s' asserts the associated dummy

conclusion. Then the queries about the questions in s are reducible to queries about the dummy conclusions in s'.

Because the definitions of reachable, possible, and inevitable conclusions began with the qualification "for a system $s \in L$," it follows that for each language L there is an associated case of the query problem. As shown in Figure 1.2, we can visualize this case of the problem in the form of an ideal tool for answering the queries associated with L. The tool accepts for inputs a description of system $s \in L$, a conclusion $c \in C_s$, and a formula q, and outputs two bits of information, viz., the reachability and inevitability of c under q.

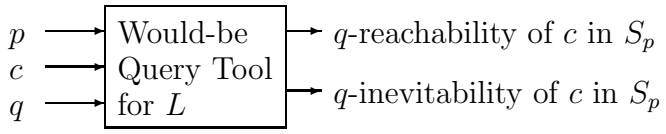


Figure 1.2: Graphical Representation of the Query Problem for Language L

The Bulk Form of the Query Problem

In practice, for a given query formula q, we may want to determine the reachability/inevitability q of *all* the potential conclusions and questions of the system. For example: Which diseases are inevitable under the condition that *bloodsugar* exceeds a certain threshold? We call this the *bulk form* of the query problem, and specify it as follows: for a system s and query formula q, partition the conclusions and questions of s into three categories of conclusions that are q-unreachable, q-possible, and q-inevitable. Such queries can of course be answered by repeatedly calling an algorithm that determines the reachability/inevitability of an individual question

or conclusion, but there may be opportunities for optimization if the questions and conclusions are based upon shared structures.

1.1.4 Features of a Query Tool

We now give a specification of the principal features of a query tool. It is an "ideal" specification in the sense that, depending upon the complexity of the systems being queried, the specified behavior may or may not be achievable.

The main function of a query tool is to accept a query from the user, and then to present the conclusions and questions so as to reflect their status under the query. The query specification has two parts: (1) an explicit user-supplied query formula q, and (2) the assignments that comprise the current attainable binding b. Whenever the user adds an assignment '$v = z$' to the current binding, we will interpret it as a constraint to be conjoined with the "extended query formula." Accordingly, we define the *extended query formula* q' to be the conjunction $q \wedge (\wedge b)$ of the user-specified query formula and the assignment constraints in the current binding.

Thus, the assignments in b constitute an implicit query formula $(\wedge b)$. The formulas $(\wedge b)$ define a special class of conjunctive queries, which we will call the *assignment queries*. A conclusion c is reachable under the query formula $(\wedge b)$ iff there is a final extension of b that causes c, and it is inevitable under $(\wedge b)$ iff all final extensions of b cause c.

The query tool should classify the conclusions and questions according to their status of being unreachable, possible, or inevitable under the extended query formula. Unreachable questions and conclusions should be made invisible. Inevitable

conclusions should be highlighted, and the user should be encouraged to read the texts of these "positive conclusions." Inevitable questions can be indicated as such. The possible nodes may become unreachable, or may become inevitable, depending upon how the user completes the current binding b. Only the existence of these nodes should be indicated to the user, but, because they may turn out to be irrelevant, their contents should not be presented to the user. Their existence could be indicated by listing the titles of the possible nodes, or just by reporting the number of possible questions and the number of possible conclusions.

We also want the query formula q to function as an integrity constraint on the input values supplied by the user. For instance, one of the conjuncts in q might be the constraint '$\mathit{YourAge} \leq \mathit{FathersAge}$'. For another example, suppose that whenever the conclusion *diabetes* were generated, the system assigns *true* to *diabetesVar*. If the query formula is set to '$\mathit{diabetesVar} = \mathit{true}$', then we want it to restrict the user inputs to just those values that lie on a path to the conclusion *diabetes*.

The integrity constraint associated with q is the requirement that any final binding producable by the user must satisfy q. Thus, as an integrity constraint, the query formula filters down the set of final bindings that can be attained. This filtering will be accomplished by using the extended query formula q' to filter down the set of answer choices for each variable v in $\mathit{ask}(b)$. Specifically, the menu of choices for v should be restricted to the values that v can assume in final bindings that satisfy q'. Here is the formal specification of the requirement.

Definition 1.1.7 ($\mathit{choices}_s(v, q')$) *For a system s, binding $b \in A_s$, variable $v \in \mathit{ask}_s(b)$, and formula q, we define the menu of choices for v under the extended*

query $q' \equiv q \wedge (\wedge b)$ *as follows:*

$$choices_s(v, q') \equiv \{b(v) \in dom_v \mid b \in F_s \ \ \& \ \ b \models q'\}$$

If $choices(v, q')$ is empty, then the question v should be dropped from the display.[9]

Our last requirement is that the query tool should provide the following feature for finding bindings that witness the reachability of conclusions. The user selects a potential conclusion c, and then the system searches for a binding that witnesses the reachability of c under the extended query formula q'. If the system finds a witness binding, then the user can have the option of having the system automatically append the witness to the current attainable binding. This will enable the user to explore the further consequences of the witness binding. Finally, the system should provide a mechanism for retracting the assignments in the witness binding.

1.1.5 Difficulty of the Problem

The general problem just presented, because it involves logical quantifications over the generally infinite set of final bindings F_s, is difficult, in fact, generally undecidable. We can trace two main sources of this complexity, each of which suffices

[9]Apart from the motivation that we offered for the requirement that q act as an integrity constraint, we also show that it is a *necessary* requirement. If the user were allowed to select a value z for v outside of $choices(v, q')$, then there would be *no* final bindings that satisfy the new extended query formula $q' \wedge (v = z)$, and so the system would be in the uninteresting "dead end" state where all conclusions and questions are unreachable. By restricting the menu to $choices(v, q')$, we prevent the user from entering the dead-end state.

to make the problem undecidable: complexity in the logic of the systems, and complexity in the language of the query formulas.

Whenever the language L for writing the systems is Turing-complete, then the query problem for L is undecidable. This is shown by the following simple reduction from the Halting problem to the reachability query problem. Let r be a RAM program, and let $s \in L$ be a system that simulates the behavior of r. Let $v_1, \ldots, v_k$ be the variables of s that initially hold the input values. Let s' be the variant of s that asserts the conclusion 'halt' whenever r reaches the halt state. Then it is easy to see that r halts on input $(z_1, \ldots, z_k)$ iff 'halt' is reachable in s' under the query formula '$v_1 = z_1 \wedge \cdots \wedge v_k = z_k$'.

We show that the query formulas are a source of undecidability, by reducing the satisfiability of a formula q to an associated q-reachability problem. Let s be the trivial system which has one variable for each variable in q, and which works by inputting a value for each of the variables in V and then unconditionally asserting the conclusion *done*. Since each of the variables in V is independently answerable, the final bindings that comprise F_s are just the total, well-typed bindings for the variables V. Now, that *done* is unconditionally asserted means precisely that for all $b \in F_s$, $\mathit{done} \in \mathit{say}_s(b)$. Hence, using the definition of q-reachability, we have that:

$$
\begin{array}{ll}
\mathit{done} \text{ is } q\text{-reachable} & \Leftrightarrow \\
\exists b \in F_s \ : \ (b \models q) \wedge \mathit{done} \in \mathit{say}_s(b) & \Leftrightarrow \\
\exists b \in F_s \ : \ (b \models q) & \Leftrightarrow \\
q \text{ is satisfiable} &
\end{array}
$$

Since the satisfiability-testing of constraint formulas is undecidable, e.g., equal-

ity constraints between polynomials over integer variables ("Diophantine equations") [28], it follows that, with unrestricted query formulas, the reachability query problem is undecidable.

Conclusion: We must restrict both the data language and the query formulas.

1.2 Queries to Production Systems

Henceforth, we will be concerned with the application of our query problem to production systems, which are a well-known and widely used framework for writing expert systems (see e.g. [15]). Here we describe the basic anatomy of a production system.

Definition of Production Systems

A *production system* has three major components: (1) a set of production rules of the form "if *precondition* then do *action*", (2) a global *working memory*, which gets tested by the preconditions of the rules and modified by the actions of the rules, and (3) an *interpreter*, which determines the order in which to activate the rules and performs the actions of the rules. In any given state of the working memory, some subset of the rules will have preconditions that evaluate to True, and the interpreter may have a "conflict resolution" mechanism for determining which of these rules to activate. In essence, each rule is a restricted construct of an imperative programming language: the precondition is a Boolean-valued expression, and the action is a command.

Example 1.2.1 (Production Rules)

- *if pressure* > 100 *then input(temperature)*
- *if bloodsugar* > 1000 *then assert-conclusion(diabetes)*

We can model each rule as a parallel process that continually monitors the shared memory. Whenever the precondition of the rule becomes true, then this process performs the action of the rule. The conflict resolution mechanism can then be modeled as a master process that controls the synchronization of the processes for the rules.[10]

Production systems can be broadly classified by the way in which the working memory is organized. The classical form of production systems was a theoretical model with a working memory consisting of a single string of symbols (of unbounded length). The rules were string-rewriting "productions," with preconditions that tested for the presence of specific substrings, and actions that replaced these substrings with new strings [11]. In the widely applied OPS5 system [15], the working memory consists of a set of "object-attribute-value vectors," which are pairs consisting of an object and a property list for that object, e.g., (jim ('age 17 'home NYC)). The preconditions of the OPS5 rules test for the existence of vectors satisfying certain properties, and the actions include creating a new vector, removing a vector, and updating a field within a vector. For our queries, we will use *binding-based* production systems, where the memory consists of a fixed set of

[10]Thus, even though production rules bear a superficial resemblance to the rules that we find in logic programs, since they both have antecedents and consequents, they are essentially different. Production rules are imperative, and communicate through side-effects to a global working memory. The rules of a logic program are declarative sentences, whose meaning has nothing to do with a notion of state. We will discuss logic programs in the section on related work.

variables and the contents of the memory is given by a binding for the variables. In such a system, a logic formula can function as a precondition: it fires when it is satisfied by the current binding for the variables.

Complexity of Queries to Production Systems

Production systems, in all the forms that we have described, are Turing-complete, and thus our query problem is undecidable for them. We now outline the proof of the Turing-completeness of the binding-based systems. The proof will illuminate the ways we must restrict the systems in order to make the queries decidable.[11]

In binding-based production systems, there are two sources of system complexity: the interactions between the rules, and the individual rules themselves. To show the complexity due to the rule interactions, we will use a system of simple rules to simulate a RAM program p. For each variable of p, we introduce a corresponding variable in the working memory of the production system. Also, there is a special variable in the working memory, called PC, that will simulate the program counter of p. For each statement of p, located at address a, we have the following production rule:

if $PC = a$ then perform the action of p and update PC

[11]The Turing-completeness of the classical production systems is shown by using the string in the working memory to simulate the tape of a Turing machine, and introducing one production for each transition rule of the Turing machine [11]. A binding is a special case of the OPS5 memory organization, and so the binding-based systems are a subclass of the OPS5 systems. Hence, our proof of the Turing-completeness of binding-based systems will imply the Turing-completeness of the OPS5 systems.

Thus, the *interactions* between simple rules suffice to make the production rules a Turing-complete language. The complexity stemming from a single rule is shown by the single-rule system:

if precondition then assert-conclusion(satisfiable)

Here, the precondition is satisfiable iff the conclusion is reachable.

Conclusion: We will have to limit the individual rules, the interactions between the rules, and the query formulas.

1.3 Insufficiency of Chaining Methods

The two standard algorithms for evaluating production systems, called forward chaining and backward chaining [15], are methods which, in the way that they are used for interactive expert systems, have the effect of driving the course of individual consultation sessions between user and system.

There may be some confusion arising from the fact that reachability querying is a backwards *reasoning* problem, in the sense that it asks about antecedent conditions under which results are generated. As emphasized in [15], backwards reasoning, which refers to the semantic level of the *problem*, is fully distinguishable from backwards chaining, which is a specific low-level technique for processing the symbols that represent the rules.

Now we can see why production-system backward chaining does not solve our backwards reasoning problem. The reachability queries are backwards reasoning problems in exactly the same way that the inverting of functions is a backwards

reasoning problem. All of these backwards reasoning problems require a mathematical analysis of the particular subjects of the problem—*i.e.*, the production systems, or the functions—and *may not be solvable*. Backward chaining, on the other hand, is just a heuristic which says that given the rule $\alpha \rightarrow \beta$ and the goal β, try to prove α as a subgoal. Its unsuitability for our reachability queries is illustrated by the fact that if the goal to be solved is a formula β, and there are no rules whose actions can establish β, then the backward chaining production systems will *prompt the user* to input specific values for the variables in β. Thus, the backward chaining is merely determining the truth-value of β within the *individual* binding defined by the user's answer values.

Furthermore, there is no simple extension of backward chaining that will solve the queries. To see this, consider a system with some rules of the form "if precondition then input(x)". The reachability of various conclusions will depend upon the preconditions of these rules, yet these rules cannot be used in any effective way by backward chaining. For instance, if the goal we are trying to prove is a formula involving x, then the rule that inputs a value for x is unusable, because it reads an unspecified value from the console. Even harder to imagine is how production-system backward chaining could be extended to cover the queries about the inevitability of conclusions under a query condition.[12]

[12]We are not claiming that chaining algorithms in general cannot be applied to our query problem, but rather, that the chaining algorithms *for production rules* do not solve the problem. There are also chaining algorithms for systems of logic rules. But the logic rules have an entirely different semantics than the production rules, because the former are declarative sentences and the latter are imperative commands. Hence, the chaining algorithms for logic systems are not applicable to the production rules.

1.4 Groundwork in Logic and Constraints

Here we spell out some basic notions of logic and constraints, which we will use in both our general theory of queries to expert systems and in the definition of the particular systems that we intend to query. The central notion that we need is that of a "constraint domain" (domain of constraints), which provides a convenient packaging of a set of logic formulas along with a semantic interpretation for their constituent symbols. (For a fuller treatment of constraint domains, see [17].) We must begin with the following basic definitions from first-order logic.

A *signature* Σ is a set of predicate, function and constant symbols, each of which has a definite degree (argument count).[13] A Σ-*term*, is either a constant in Σ, a variable, or the syntactic form $f(t_1, \ldots, t_k)$, where each t_i is a Σ-term and $f \in \Sigma$ is a function symbol of degree k. An *atomic formula* over Σ is a syntactic form $p(t_1, \ldots, t_k)$, where $p \in \Sigma$ is a predicate symbol of degree k and $t_1, \ldots, t_k$ are Σ-terms. A Σ-formula is either an atomic formula, or $OP(f_1, \ldots, f_k)$, where each f_i is a Σ-formula, and OP is a logical operator, such as conjunction $\wedge$, disjunction $\vee$, negation *not*, universal quantification $\forall_x$, existential quantification $\exists_x$, or implication $\rightarrow$. We call a formula *monotone* if it is built with only conjunction and disjunction operators. A Σ-*expression* is either a Σ-formula or a Σ-term.[14] A

Our query algorithm will work by performing a non-obvious transformation from production system queries to systems of first-order logic formulas. These formulas may in turn be converted to datalog systems. *Then* we will have the option of applying datalog chaining algorithms to the transformed systems.

[13] Constant symbols have degree zero, and are regarded as the function symbols of degree zero.

[14] Thus, terms are expressions that evaluate to domain-values, and formulas evaluate to truth-values. In other contexts, *e.g.*, spreadsheets, "formulas" are extended to cover terms as well, but

Σ-*sentence* is a Σ-formula with no free variables, and a Σ-*theory* is a set of Σ-sentences.

An *interpretation* I for Σ (a Σ-*interpretation*) consists of a *universe of discourse* D_X, and a mapping that sends each constant $c \in \Sigma$ to a value $c^I \in D$, each k-ary function symbol $f \in \Sigma$ to a k-ary operation f^I on D, and each k-ary predicate symbol $p \in \Sigma$ to a k-ary relation p^I on D.

A *constraint domain* X consists of a signature Σ_X, an interpretation I_X for Σ_X, and a set C_X of Σ_X-formulas called *constraints*. Also, there is a designated set $P_X \subseteq C_X$ of *primitive* constraints, and it is assumed that every member of C_X can be expressed as a conjunction of primitives in P_X. Hence C_X is closed under conjunctions. Furthermore, we assume that C_X is closed under existential quantifications, and that quantifier elimination can be effectively performed. This implies that satisfiability of the constraints is decidable.[15]

A constraint domain X has *opposite predicates* if for each predicate symbol $p \in \Sigma_X$, there is an associated *opposite* predicate symbol $\overline{p} \in \Sigma_X$ such that $p(z_1, \ldots, z_k)$ is true in I_X iff $\overline{p}(z_1, \ldots, z_k)$ is false in I_X. The constraint domains for our queryable systems will be required to have opposite predicates.[16]

in logic we do not want to blur the distinction between truth-values and domain values. Hence, we will stick to the restricted meaning of the word "formula." This usage is implied whenever we speak of e.g. $x > y$ as an *atomic* formula.

[15]To determine the satisfiability of a constraint formula f, we apply the quantifier elimination procedure to the existential closure $\exists f$. This procedure will produce a logically equivalent formula f'. Since it is logically equivalent to the variable-free formula $\exists f$, then f' must be variable-free as well as quantifier-free. Hence, f' will either be *true* or *false*. In this way, the satisfiability of f is determined.

[16]The assumption of opposite predicates does not represent a theoretical loss of generality,

Example 1.4.1 (Rational Polynomial Comparisons)

- D_X = *the set of rational numbers*
- $\Sigma_X = \{$*constants for rational numbers;* $+,-,*;\ =, \neq, \geq, >, \leq, <\}$
- I_X = *standard arithmetic interpretation of symbols in* Σ_X
- P_X = *atomic* Σ_X*-formulas*

For instance, $5x + 9y \geq 20z^2$ is a constraint in this domain.[17] This constraint domain has the obvious pairs of opposite predicates, e.g., $=$ and $\neq$ are opposite. In the next example, we restrict this constraint domain to the subdomain of linear constraints.

Example 1.4.2 (Linear Rational Polynomial Comparisons)
Here everything is the same as in the preceding example, except that P_X *is restricted to comparisons between* linear Σ_X*-terms.*

Our final example is a simple constraint domain whose formulas are useful for data records.

Example 1.4.3 (Univariate Monotone Order Constraints (UMO))

because a given constraint domain without opposite predicates can always be extended by introducing, for each predicate symbol p, a new predicate symbol $\overline{p}$ for the opposite. On the other hand, it may represent additional computational complexity, because the constraint-processing methods must be able to operate on arbitrary conjunctions of constraints.

[17] In the setting of arithmetic, the word "term" covers $5x$, but not $5x + 9y$. We shall stick, however, with the first-order logic terminology, where a term is any expression denoting a domain value. Then, $5x + 9y$ is surely a term.

- D = *an ordered set*
- Σ_X = {*constant symbols,* $=, \neq, <, \geq, >, \leq$}
- I_X = *the interpretation of* Σ_X *that expresses the ordering*
- C_X = *the set of monotone formulas whose atomic constituents are comparisons between a variable and a constant*

Here is an example of a UMO constraint: $((Month < may) \vee (Month = july)) \wedge (Year > 1900)$. This example illustrates the fact that the primitive constraints and the conjunctions of primitives are not necessarily disjunction-free.

Given a constraint domain with opposite predicates, we may, without losing generality, simplify our technical development by restricting our attention to monotone formulas. To do this, we use De Morgan's laws to replace each boolean formula by its negation-free equivalent.[18] For instance, replace $not(x = 1 \wedge y = 1)$ by $x \neq 1 \vee y \neq 1$. Here is the full definition.

Definition 1.4.4 (Equivalent Formula f^+ and Opposite Formula f^-) *For a boolean formula f, we define two associated monotone formulas, the negation-*

[18] In the closing chapter, we will describe an alternative semantics for unary negations in the presence of $\perp$ values, and show how our negation-free query algorithm can be extended in a simple way to cover this semantics.

free equivalent f^+, and the negation-free opposite f^-, as follows:

$$\begin{aligned}
(f_1 \wedge f_2)^+ &= (f_1)^+ \wedge (f_2)^+ \\
(f_1 \vee f_2)^+ &= (f_1)^+ \vee (f_2)^+ \\
(p(t_1, ..., t_k))^+ &= p(t_1, ..., t_k) \\
(not(f))^+ &= (f)^-
\end{aligned}$$

$$\begin{aligned}
(f_1 \wedge f_2)^- &= (f_1)^- \vee (f_2)^- \\
(f_1 \vee f_2)^- &= (f_1)^- \wedge (f_2)^- \\
(p(t_1, ..., t_k))^- &= \overline{p}(t_1, ..., t_k) \\
(not(f))^- &= (f)^+
\end{aligned}$$

We now define the satisfaction relation $b \models_I f$ between bindings b and formulas f, in the context of the interpretation I. Informally, this means that f evaluates to True under the interpretation of the symbols provided by I and b. If the binding has $\perp$ values that prevent f from evaluating to True, then the relation $b \models_I f$ does not hold.

Definition 1.4.5 (The Value of a Term in a Binding, $val_I(t, b)$)
For a Σ-term t, Σ-interpretation I, and binding b, define:

$$\begin{aligned}
val_I(const, b) &= const^I \\
val_I(var, b) &= b(var) \\
val_I(f(t_1, \ldots, t_n), b) &= \begin{cases} \perp, \text{ if } val_I(t_j, b) = \perp \text{ for some } j \\ f^I(val_I(t_1, b), \ldots, val_I(t_n, b)) \text{ otherwise} \end{cases}
\end{aligned}$$

Definition 1.4.6 (The Satisfaction Relation, $b \models_I f$)

For a binding b, monotone Σ-formula f, and Σ-interpretation I, define:

$b \models f_1 \wedge f_2 \Leftrightarrow (b \models f_1)$ *and* $(b \models f_2)$

$b \models f_1 \vee f_2 \Leftrightarrow (b \models f_1)$ *or* $(b \models f_2)$

$b \models p(t_1, \ldots, t_k) \Leftrightarrow (val_I(t_1, b), \ldots, val_I(t_k, b)) \in p^I$

1.5 Related Work

Parts of this thesis were published in summary form in [40]. A detailed exposition of portions of the following survey is given in [39]. We will discuss four areas of related work.

Thinksheet Expert Systems: They are based on acyclic production systems. We will restrict them to obtain queryable expert systems.

Constraint Domains: Our algorithm gives a reduction from production-system queries to conjunctive satisfiability problems. By combining it with conjunctive solvers for specific constraint domains, we obtain instances of the algorithm.

Constraint Logic Programming and Datalog+Constraints: These systems can optionally be used for the satisfaction-testing phase of the query processing.

Abductive Logic Programming: The problem of finding bindings that witness the reachability of conclusions in an interactive system falls under the general

heading of abductive reasoning problems. We touch upon abductive logic programming, because it approaches an analogous problem for logic programs.

1.5.1 Constraint Logic Programming

Overview

This material on constraint logic programming is not necessary in order to understand our algorithm. Its relevance is that these techniques can be used as one approach to solving the constraints generated by our algorithm.

Constraint logic programming (CLP) [17, 16, 6, 7, 18] is a generalization of classical logic programming [37, 1, 25], whereby the equality constraints between terms—*e.g.*, *father_of(X)=father_of(henry)*—and their solution by unification algorithms are replaced by generic constraint domains and their associated constraint-processing methods. Van Hentenryck observed that this generalization has simplified the basic theory of logic programming, because the whole group of concepts arising from the constraint domain of term equations, such as unification algorithms and most general unifiers, which were originally presented at the center of logic programming theory, are now ancillary topics that are handled in one sphere of application of constraint logic programming [45].

A constraint logic program is a representation for knowledge that takes the form of a theory of logic. The axioms of a constraint logic program are sentences called "rules," which may contain constraint formulas as constituents. Here is an example of a constraint logic program that defines a collection of spatial relations.

Example 1.5.1 (Constraint Logic Program)

```
square(R,X,Y) :- abs(X) = R, abs(Y) <= R.
square(R,X,Y) :- abs(Y) = R, abs(X) <= R.
%
diagline(R,X,Y) :- X >= 0, Y >= 0, X + Y = R.
%
diamond(R,X,Y) :- diagline(R,X,Y).
diamond(R,X,Y) :- diagline(R,X,Y1), Y1 = -Y.
diamond(R,X,Y) :- diagline(R,X1,Y), X1 = -X.
diamond(R,X,Y) :- diagline(R,X1,Y1), X1 = -X, Y1 = -Y.
%
squareTower(R,X,Y) :- R >= 1, square(R,X,Y).
squareTower(R,X,Y) :- R >= 1, R' = .9 * R, squareTower(R',X,Y).
%
diamondTower(R,X,Y) :- R >= 1, diamond(R,X,Y).
diamondTower(R,X,Y) :- R >= 1, R' = .8 * R, diamondTower(R', X, Y).
```

This example illustrates the fact that the *syntax* of constraint logic programs is identical to the syntax of the classical logic programming language Prolog. The difference is that CLP surpasses the very limited constraint-handling capabilities of Prolog. We illustrate this difference with an example. If Prolog is given the query '$square(1, X, Y) \wedge X = Y$', which asks for a point (X, Y) on the intersection of the unit square and the line through the origin with slope 1, it will fail with the complaint that X and Y are uninstantiated. A CLP interpreter, on the other hand, should be able to succeed and produce the answers '$X = 1 \wedge Y = 1$' and

'$X = -1 \wedge Y = -1$'.

There are two types of formulas which can appear in the rules: *constraints* belonging to the constraint domain, such as '$abs(Y) = R$', and *program atomic formulas*, such as '$squareTower(R*0.9, X, Y)$', which have predicate symbols that are not interpreted by the constraint domain. The meaning of these predicate symbols is instead determined by the system of rules as a whole. The formulas at the heads of the rules must be program atomic formulas. Typically, the predicate symbols for the constraints are written in infix notation, while the program predicates are written in prefix notation.

A rule is *primitive* if all the formulas in its body are constraints. For instance, "$square(R, X, Y) \leftarrow abs(X) = R,\ abs(Y) \leq R$" is primitive. Primitive rules generalize the "ground facts" (individual tuples) of classical logic programs, since the ground facts are equivalent to primitive rules with equality constraints. For example, the ground fact *brother(john,henry)* can be written as the primitive rule $brother(X, Y) \leftarrow X = john,\ Y = henry$.

The meaning of the logic program can be described by a mapping that sends each program predicate of degree k to an associated relation of degree k. The meaning of our example program is simple to understand. Namely, the relation for *square* consists of all triples (R, X, Y) such that (X, Y) lies on the square of width R centered at the origin; the relation for *diamond* is all triples (R, X, Y) such that (X, Y) lies on the rotated square with corners at $(0, R), (R, 0), (0, -R), (-R, 0)$; the meaning of *squareTower* is all triples (R, X, Y) where (X, Y) lies on the union of concentrically nested squares whose outer square has width R, and for which each successive square is 90% of the size of its surrounding square, where the union has

only one square of size greater than 1; and similarly for *diamondTower*, which has a shrinking factor of 80%.

Here is an example query: which points are common to $squareTower(99, X, Y)$ and $diamondTower(101, X, Y)$? This query is given by the conjunctive goal formula '$squareTower(99, X, Y) \wedge diamondTower(101, X, Y)$'. We view this goal formula as a request for a usable representation for the set of (X,Y) pairs in the intersection.

The following construction shows that the problem of solving these queries is subsumed by the general problem of determining the meanings of the program predicates. Suppose that the goal formula is $G_1 \wedge \cdots \wedge G_m$. Then introduce a new predicate called *result*, and add the rule $result(\alpha_1, \ldots, \alpha_k) \leftarrow G_1, \ldots, G_m$, where $\alpha_1, \ldots, \alpha_k$ are the free variables in $G_1, \ldots, G_m$. So, for our example query, the rule would be $result(X, Y) \leftarrow squareTower(99, X, Y),\ diamondTower(101, X, Y)$. Then the relation that gives the meaning for the predicate *result* gives the answer to the query.

In general, the meaning-relations for the predicates will contain an infinite number of points, and so we must look for constraint domains where the meanings of the predicates will have a usable *finite representation*. In the CLP setting, it will be natural to represent relations by sets of primitive rules, where each set of rules will be interpreted to mean the union of all the points that are described by one of the rules.

Example 1.5.2 *The result of the query* diagline(1,X,Y) *can be represented by the primitive rule:*

$$result(X, Y) \leftarrow X \geq 0,\ X \leq 1,\ Y = 1 - X$$

Even though the point-set for this rule is infinite, it is still directly usable, because it expresses a function from the independent variable x ranging over [0,1] to the dependent variable y.

Example 1.5.3 *The answer to the query* diamond(1,X,Y) $\wedge$ Y ≥ 0 *is representable by the two primitive rules:*

$$\mathit{result}(X,Y) \leftarrow X \geq -1,\ X \leq 0,\ Y = X + 1$$
$$\mathit{result}(X,Y) \leftarrow X \geq 0,\ X \leq 1,\ Y = -X + 1$$

For an example where there is no useful constraint representation for the meaning relations, consider the constraint logic programs over the domain of integer polynomial equality constraints. Then an infinite point set can be represented by a fact such as "$\mathit{result}(X,Y) \leftarrow 35*X^3+11*Y^5+X*Y*Z = 0$", but such representations are unusable, because the satisfiability of such constraints is undecidable—and hence we cannot expect to learn *anything* about the points in the solution sets.

Syntax

For a given constraint domain X, we now will define $CLP(X)$, the set of constraint logic programs with constraints in X. Recall from Section 1.4 that X consists of a triple (Σ_X, I_X, C_X) where Σ_X is a signature of predicate, function and constant symbols, I_X is a Σ_X-interpretation, and C_X is a collection of Σ_X-formulas called the constraints.

Let P be a set of "program" predicate symbols, and assume that P is disjoint from Σ_X. In our example, $P = \{\mathit{square}, \mathit{diagline}, \ldots, \mathit{diamondTower}\}$. Define a

program atomic formula to be an atomic formula $p(t_1, \ldots, t_k)$, where $p \in P$, and each t_i is a Σ_X-term, e.g., $squareTower(R * 0.9, X, Y)$.

A *rule* of $CLP(X)$ is a formula of the form $\forall(\alpha_1 \wedge \cdots \wedge \alpha_x \rightarrow \beta)$, where each α_i is either a program atomic formula or a constraint formula, and β is a program atomic formula. A *program* of $CLP(X)$ is a set of $CLP(X)$ rules.

Semantics

Although the meaning of $CLP(X)$ programs is clear for simple examples—e.g., for the program $\{square(X, Y) \leftarrow abs(X) = 1,\ abs(Y) = 1\}$, the meaning of *square* is the relation $\{(x, y) \mid abs(x) = 1, abs(y) = 1\}$—we still need a formal definition of the semantics because (1) the picture is more complex when the rules are recursive, and (2) the correctness proofs for query algorithms will require a formal definition of the meaning of a program.

A logic program is a theory, *i.e.*, a set of declarative sentences, and we define its meaning to be the set of facts that are logical consequences of this theory, where a *fact* is an atomic formula $p(c_1, \ldots, c_k)$, such that each c_i is a constant.[19] I.e., for a program T of $CLP(X)$, and a program predicate p, we define:

$$meaning(p) = \{facts\ f\ s.t.\ T, I_X \models f\}$$

where $T, I_X \models f$ means that f is a logical consequence of the theory T and the interpretation I_X which the constraint domain X provides.[20]

[19] As a technical convenience, we assume that for each member d of the universe of discourse, there is a constant c_d whose interpretation is d. With this assumption, relations over the universe of discourse can be represented by sets of facts.

[20] See e.g. [39] for a complete definition of the relation $T, I \models f$. Here is an informal definition,

Query Algorithms

We will explain the query algorithms in terms of "rule trees," which represent chains of valid deductions. These trees will have constraints at their leaves, and derived facts at their roots. For simplicity, we assume that all program atomic formulas are in the form $p(X_1, \ldots, X_k)$, where $X_1, \ldots, X_k$ is a list of distinct variables. This canonical form is easily obtained by introducing auxiliary equality constraints. For example, $p(3, abs(4))$ can be replaced by the equivalent $p(X_1, X_2) \wedge X_1 = 3 \wedge X_2 = abs(4)$.

For a program T of $CLP(X)$, a *rule tree* over T is a tree whose nodes are formulas, where (1) the leaves are constraints in C_X, (2) the internal nodes are program atomic formulas, and (3) for each internal node I with children $N_1, \ldots, N_k$, the formula '$(N_1 \wedge \cdots \wedge N_K) \rightarrow I$' is a substitution-instance of one of the rules of T. If the tree has program atomic formula α at its root, and constraints $\beta_1, \ldots, \beta_n$ at its leaves, then the primitive rule $\forall(\beta_1 \wedge \cdots \wedge \beta_n \rightarrow \alpha)$ is a logical consequence of the program T and I_X. The significance of such a *derived rule* for T is that we can obtain facts that are consequences of T by calling a conjunctive solver with the arguments $\beta_1, \ldots, \beta_n$, and then, if the conjunction is satisfiable, asking the solver to project the results onto the variables in α. Moreover, it can be shown that every fact which is a logical consequence of T, I_X can be obtained from some rule tree [39].

which should make the idea clear. The interpretation I specifies meanings for some of the symbols appearing in T and f, and leaves others unspecified. That $T, I \models f$ means that whenever I' is an interpretation that extends I and all of the sentences of T are true in I', then f is also true in I'. For example, if A is the standard interpretation over the rationals for the arithmetic symbols, and T is the theory $\{\forall(abs(x) = 1 \wedge abs(y) = 1 \rightarrow square(x, y))\}$, then $T, A \models square(1, 1)$.

Example 1.5.4 (Rule Tree for a CLP Query) *Figure 1.3 shows the rule tree for a query to the constraint logic program of Example 1.5.1. The query,* $R = 10 \wedge squareTower(R, X, Y)$*, asks for points on the tower of squares with radius 10. Every internal node of the tree corresponds to an instance of one of the program rules, e.g., '*$sqTow(R', X, Y) \leftarrow R' \geq 1 \wedge square(R', X, Y)$*' is a variant of one of the rules. The derived primitive rule for this tree is obtained by conjoining all of the leaves:*

$result(X, Y) \leftarrow R = 10 \wedge R \geq 1 \wedge R' = .9R \wedge R' \geq 1 \wedge abs(X) = R' \wedge abs(Y) \leq R.$

Put in simpler form, it is: $result(X, Y) \leftarrow abs(X) = 9 \wedge abs(Y) \leq 9$*. Hence it follows that e.g.* $(-9, 7)$ *belongs to that tower of nested squares.*

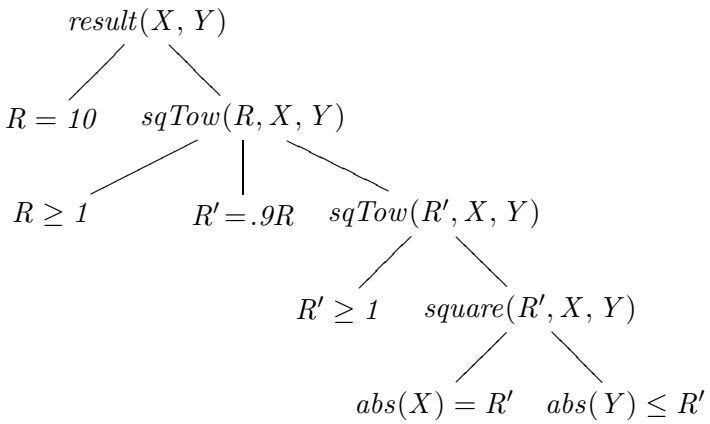


Figure 1.3: Rule Tree for Query $R = 10 \wedge squareTower(R, X, Y)$

The two fundamental CLP evaluation algorithms, viz., so-called top-down and bottom-up evaluation, can be pictured as methods that construct rule trees and then call a constraint solver to extract solutions from the trees. In the bottom-up evaluation, the rule trees are in effect constructed starting from leaves and ending

with the root, and, in top-down evaluation, the construction starts from the root and ends with the leaves.

In fact, what the algorithms construct is not the trees themselves, but the primitive rules that are obtained from the leaves and roots of these trees. Therefore, the complexity of the algorithms is not governed by the sizes of the trees *per se*, but rather by the number of distinct constraints which label the leaves of these trees.[21]

Bottom-Up Evaluation

The bottom-up algorithm maintains a set F of derived primitive rules, which is initialized to the primitive rules in the program T that is being evaluated. The algorithm iterates the following sequence of steps, until no more rules can be added to F:

1. Choose a rule $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k \rightarrow \Gamma$ in T, with program atomic formulas $\alpha_1, \ldots, \alpha_k$, and constraints $\beta_1, \ldots, \beta_k$

2. Choose instances $i_1, \ldots, i_k$ of primitive rules in F, such that the consequent of i_j is equal to α_j, for $j = 1, \ldots, k$

3. Form the rule $r' := body(i_1), \ldots, body(i_k), \beta_1, \ldots, \beta_k \rightarrow \Gamma$

4. If the body of r' is satisfiable, and it cannot be easily determined that r' is already implied by F, then add r' to F. (One simple test is to see whether r'

[21] The rule tree in Figure 1.3 illustrates the fact that if the rules are recursive, then there may be rule trees that are arbitrarily large. If the rules are nonrecursive, then the height of the trees is bounded by the number of rules. The rules that will arise from translating our reachability formulas into datalog will have a very specific, nonrecursive form, which will guarantee that the sizes of the derived rules are linearly bounded by the size of the rule system.

is a variant of a rule in F.)

Top-Down Evaluation

The top-down algorithm takes a goal formula g as its argument, and then attempts to find a rule tree that proves the formula at its root. It is based on a simple modification of the algorithm used by a classical pure prolog interpreter. The change is that the MGU of the Prolog interpreter—*i.e.*, the canonical form for a conjunction of "Herbrand" equality constraints—is replaced by a domain-specific representation for a conjunction of constraints. Specifically, the algorithm maintains two data structures: (1) a set G of program atomic formulas, called "goals", which have yet to be expanded, and (2) a set of constraints C, represented in the form of a conjunctive store.

G is initialized to be the singleton containing the root goal, and C is initialized to hold the empty conjunction of constraints. At the end of the procedure—if it succeeds—G will be empty, and C will be logically equivalent to the conjunction of the leaves of a derived primitive rule with g at its root. The algorithm iterates the following sequence of steps:

1. Remove a goal formula t from G
2. Find a variant $\alpha_1 \wedge \cdots \wedge \alpha_k \wedge \beta_1 \wedge \cdots \wedge \beta_m \rightarrow t$, where the α_i are program atomic formulas and the β_i are constraint formulas
3. Add $\alpha_1, \ldots, \alpha_k$ into G
4. Conjoin $\beta_1, \ldots, \beta_m$ into C
5. Put the new C into canonical form

6. If C is inconsistent, then fail

1.5.2 Constraint Domains

The following material is not necessary in order to understand our algorithm, but rather provides a survey of various constraint representations that can be used for particular spheres of application of our algorithm. In the body of the thesis, when we describe our implementation, we will also give the details of our constraint representation.

We now describe some representations for conjunctions of various types of constraints X. A representation for the constraints of the domain X, when equipped with methods for performing conjunctions, testing satisfiability and projecting onto a set of variables, can serve as a basis for constraint logic programming.

Binding Equality Constraints

Here the constraints are conjunctions of formulas '*var* = *const*', *e.g.*, '*Month* = *jan* $\wedge$ *Year* = 1998'. These formulas are equivalent to the records of finite relational databases, *i.e.*, the single-valued mappings from attributes to values. Hence the record is the natural form of representation for such conjunctive formulas.

Difference Constraints

In this constraint domain, the primitive constraints are of the form $x \leq^k y$, where x and y are variables or constants, and k is a number which is called the "gap" value. The meaning of the constraint is that the gap between x and y is at least k, *i.e.*, that $y - x \geq k$. For instance, $J \leq^4 S$ can express the fact that John is at least 4

years younger than Sam. A conjunction of these constraints can be represented by a directed graph with one node for each variable and constant, and an edge from x to y with label k for each of the conjuncts $x \leq^k y$.

See [9] for an introduction, in a linear programming context, to difference constraints over real variables. See [32] for constraint-processing algorithms for difference constraints with integer variables and non-negative gap values. These algorithms are implemented in a system described in [3].

Linear Arithmetic Constraints

A conjunction of linear equality constraints can be represented by a matrix in canonical form, e.g. row-reduced echelon form, and conjunction and canonicalization can be performed by Gaussian elimination [38]. For linear inequalities, the vast field of linear programming algorithms is available [10, 33]. The constraint logic programming systems CLP(R) [18] and Eclipse [13] use a simplex algorithm for processing linear inequalities.

Term Equality Constraints

The combination of top-down evaluation with this constraint domain gives us a classical logic programming interpreter [37, 1, 25]. The signature Σ_X for the constraint domain is $K \cup \{\text{`='}\}$, where K is a set of function and constant symbols. The universe of discourse is defined as the set of all variable-free K-terms. The primitive constraints are equalities $t_1 = t_2$, where t_1 and t_2 are K-terms that *may* have variables. This constraint domain thus has the peculiarity that the symbols in K are used both in the construction of the members of the universe of discourse

and in the constraint formulas which are interpreted over this universe of discourse.

Each function symbol f in K is interpreted as the operation on ground K-terms which maps the ground terms $g_1, \ldots, g_n$ to the ground term $f(g_1, \ldots, g_n)$. For instance, if $K = \{f, g, h, 2, 3\}$, then '$f(X, g(Y)) = f(2, g(h(Z)))$' is a primitive constraint which is satisfied by the binding $\{(X, 2), (Y, h(h(3))), (Z, h(3))\}$.

A *most general unifier* (MGU) for a conjunction ϕ of term equations is a canonical representation in the form of a logically equivalent conjunction ϕ' of term equations. This form makes it easy to find solutions to ϕ, because it partitions the variables into two categories: independent variables, which appear on the right sides of the equations, and dependent variables, which appear on the left sides. Specifically, the equations in an MGU are all of the form $var = term$, where *var* appears in only one of the conjuncts of the MGU, and *var* does not occur in any of the terms on the right sides.

It can be shown that if ϕ is satisfiable, then it has an MGU. An algorithm that either computes an MGU for ϕ, or fails if ϕ is unsatisfiable, is called a *unification* algorithm.

Finite-Domain Constraint Satisfaction Problems

The work [44] effectively introduced, CLP(Finite-Domains), constraint logic programming over the (broad) domain of constraints in which each of the variables has a finite domain of possible values. There, top-down expansion of the program rules was combined with so-called "consistency techniques" for processing finite-domain constraint satisfaction problems (CSPs) (see e.g. [24]), which are heuristics for reducing the branching factor of the search for solutions to a constraint. Imple-

mentations of CLP(Finite-Domains) include Eclipse [13] and CHIP [12].

The consistency techniques maintain a set of possible values that each variable can assume, and, as new constraints get inferred or added to the constraint store, inconsistent values may be removed from the sets. The sets are used by a backtracking search algorithm, which constructs bindings and tests to see if they satisfy the constraints of the problem. Each instantiation produced by the search algorithm effectively conjoins a new constraint $var = const$ into the store. In this way, the backtracking search, propagation of constraints, and reduction of domains are tightly intertwined.

For instance, [44] shows how the 5-queens problem, which seeks to place 5 queens on a five-by-five chessboard so that none can attack another, can be solved with only two choices and no backtrackings. First, the problem is modeled as a CSP with five variables $X_1, \ldots, X_5$, each taking values in $\{1, \ldots, 5\}$; the value of X_i indicates the row on which the queen in column i is placed. That no queen can attack another is then expressed by a conjunctive constraint formula $\phi =$ '$(X_1 \neq X_2) \wedge (X_1 \neq X_3) \ldots \wedge (X_2 \neq X_1 + 1) \cdots$'. All of the domains are initialized to $\{1, \ldots, 5\}$. Then the first choice is made: place the first queen in the first row, i.e., conjoin '$X_1 = 1$' into ϕ. This conjunction implies that 1 is no longer a valid value for the other variables, and so their domains are reduced to $\{2, 3, 4, 5\}$. Further, because of the diagonal attacks of the first queen, i can be removed from the domain of X_i, for $i = 2, \ldots, 5$. The domain of X_2 is then $\{3, 4, 5\}$. Another choice is made: $X_2 = 3$. This has the effect of reducing the domains of X_3 and X_4 to single values, and so they are each instantiated—without a choicepoint. Constraint propagation then reduces the domain of X_5 to a single value, and a solution is obtained.

This is called a "constrain-and-generate" strategy, because it processes constraints before (and during) the search for solutions. In contrast, generate-and-test is less efficient, because it processes the constraints only after a binding has been fully constructed. Standard chronological backtracking is a weak form of constrain-and-generate, since it checks partially constructed bindings to see if they falsify the constraints—but, unlike the domain-reduction techniques, it does not use the constraints to reduce the choices that will be made later in the search process.

1.5.3 Datalog with Constraints

Again, this material is not necessary to understand our algorithm, but the areas surveyed have provided constraint-processing techniques that may be useful for an implementation of the algorithm.

In this section we discuss an area of constraint databases[22] that emerged out of constraint logic programming: the theory and practice of finitely representable

[22]In the constraint database paradigm, a database is construed as the solutions to a collection of formulas over a domain of constraints, the query as a small formula, and query-processing as constraint-solving. In this *reduction* of database theory to applied, data-intensive logic, we encounter issues which are not matters of pure logic but rather have to do with the *processing* of database formulas, including (1) how to manage massive collections of formulas, (2) how to minimize the number of accesses to the secondary storage on which the formulas reside, (3) how to *index* the relations and objects described by the database formulas in order to accelerate the processing of typical queries, and, more generally, (4) how and under what conditions can we perform efficient logic-based queries? Two basic paradigms for constraint databases have emerged: constraint databases in the form of logic programs ("datalog+constraints"), and constraint databases as systems of objects. See [2, 32, 14] for some reports on early prototypes and applications.

relational databases[21].

Before the advent of constraint logic programming, there was a well-known connection between classical relational databases and logic programming, viz., that the "datalog" sublanguage of logic programs suffices to express both relational databases and relational algebra queries over these databases. In classical datalog, compound terms are disallowed, i.e., the arguments to the predicates are restricted to being variables or constants. With these constructs, each record of a table t can be represented by a primitive rule $t(c_1, \ldots, c_k) \leftarrow \mathit{true}$, for constants $c_1, \ldots, c_k$. Hence, the whole database is representable by a set of such primitives. Also, the relational algebra queries are representable by datalog rules, e.g., '$p(X, Y, Z) \leftarrow q(X, Y)$, $r(Y, Z)$' expresses a join of the relations q,r, and, '$s(X, Z) \leftarrow p(X, Y, Z)$' expresses a projection of p onto two of its attributes.

The extension from classical to constraint logic programs leads to a corresponding extension of datalog, by allowing constraints in the bodies of the rules. Augmented with constraints, datalog becomes indistinguishable from general constraint logic programs. Nevertheless, because datalog+constraints evolved out of a database context, with its emphasis on largeness of scale, indexing, etc., it still connotes a particular perspective on constraint logic programs.

An important and natural database application of CLP—but not a privileged application—is spatial constraint databases [**?**]. The spatial relations generally contain infinite sets of points. For instance, our example CLP program on page 31 can be seen as database that describes a spatial configuration of nested squares and diamonds. For a more realistic example, consider the infinite relation (Latitude,Longitude,Continent). This relation itself is not finitely representable, but for

practical purposes, the boundaries may be approximated by chains of line segments Thus the regions are representable by unions of convex polyhedra. Since each convex polyhedron is an intersection of half-planes, and a half-plane is the solution set to a linear inequality, it follows that the whole region can be represented by a disjunction of conjunctions of linear inequalities.

For finite relational databases, [4] established a theoretical model for the query languages whereby each query expression is some syntactic form that determines a transformation from input databases to output relations, *i.e.*, an operation on relations. All the various query languages, such as relational algebra, relational calculus, and datalog rules, are then cast as languages for defining such relational operations. This model requires the *closure condition* that the output from a query has the same form as its inputs. A query expression which violates the closure condition is called "unsafe." The closure condition ensures that the queries are composable and thus form an algebra. For finite relational databases this condition means that the result of a query must be a *finite* relation. Negation in relational algebra leads to unsafe queries over finite relational databases, because, if the universe of discourse is infinite, the complement of a finite relation is infinite [42].

This algebraic query model was generalized in [20, 21] to relations that are *finitely representable* by disjunctive normal formulas.[23] A transformation of relations is "safe" if for all input relations representable by DNF formulas, the transformed relation is also representable by a DNF formula. If the formula for the output relation is computable from the formulas for the input relations, we can

[23]I.e., relations that are the solution sets for disjunctions of conjunctions of constraints, such as $\{(x, y) \mid (x > 0 \wedge y > 0) \vee (x < 0 \wedge y < y)\}$.

then process the queries by working with these finite representations of the relations.

We now will use the "bottom-up" evaluation of constraint logic programs (c.f. Section 1.5.1) to define a query algebra of disjunctive normal formulas [21]. The DNF formulas will be represented by a set of primitive rules: each rule represents the projection of its body onto its head, and the set of rules represents the disjunction of these projected formulas. Now, recall that bottom-up evaluation produces a set of derived primitive rules representing the facts that are logical consequences of the program. A program P then determines the following query transformation of sets of primitive rules: given a set of primitive rules *Facts*, form the program $P' \equiv P \cup \mathit{Facts}$, and then use bottom-up evaluation to compute the set of primitive rules derivable from P'. In this way, each program of $CLP(X)$ defines a transformation of sets of primitive rules, *i.e.*, a transformation of DNF formulas. Thus, each program defines an operation in the query algebra. This generalizes the classical datalog view of a logic program as defining a transformation of sets of ground facts [42, 43].

By extending the database constraint language to to include disequalities $\mathit{var} \neq \mathit{const}$, relational algebra queries with unrestricted negations then become closed, *i.e.*, the safety problems of classical relational algebra *simply disappear*. On the other hand, there are richer languages of constraints for which the closure condition fails to hold for datalog queries, e.g., polynomial equality constraints.[24]

[24]Here is an example taken from [21]. Consider the program which contains the two rules "$\mathit{transclosure}(X, Y) \leftarrow X = Y$", and "$\mathit{transclosure}(X, Y) \leftarrow r(X, Z)$, $\mathit{transclosure}(Z, Y)$". If the input relation is given by the fact "$r(X, Y) \leftarrow Y = 2 * X$", then the meaning of *transclosure* is the relation $\{\mathit{transclosure}(X, Y) \mid Y = X * 2^i$ for some integer i$\}$—which is not representable by a finite set of primitive rules.

1.5.4 Abductive Logic Programming

Abductive reasoning is a mode of plausible inference that uses a theory to "infer" hypotheses that can explain observed facts [5, 19, 8]. In the words of [5], abduction is the generation of explanations for what we see around us, e.g., given the theory that drunk people cannot walk straight, and that Jack cannot walk straight, we might hypothesize that Jack is drunk. I.e., from $(a \rightarrow b)$, and b, hypothesize a. There may of course be other explanations, for instance, Jack is dizzy after a roller coaster ride, and so abductive reasoning only provides plausible explanations.

As Charles Sanders Peirce observed in the 19th century, abduction is one member of a triad of modes of reasoning that includes deduction and induction as well. To illustrate, consider the following statements:

Fact: These marbles are from the box

Rule: All marbles in the box are white

Consequent: These marbles are white

The three modes of reasoning are:

Deduction: From the Fact and the Rule, conclude the Consequent

Induction: From the Fact and the Consequent, hypothesize the Rule

Abduction: From the Rule and the Consequent, hypothesize the Fact

Abductive logic programming [19] is the application of abductive reasoning to the theories that are constituted by logic programs. Whereas in the ordinary,

deductive use of logic programs, we seek the conclusions that follow from the facts and rules of the program, in the abductive use, given some conclusions, we seek sets of facts which, when combined with the program, imply those conclusions.

The approach described in [19] is as follows. Certain predicates are declared to be "abductable," which means that facts with these predicates are to be collected as hypotheses instead of being proven through the rules. Specifically, if p is an abductable predicate, and the goal $p(\alpha)$ is expanded by the (top-down) evaluation process, then no attempt will be made to expand $p(\alpha)$ by means of the rules, but instead $p(\alpha)$ will get added into the current set of hypotheses. If the computation starting with the top-level goal g succeeds, then the set of facts collected in this way, in combination with the rules of the program, provides a hypothesis that is sufficient to prove g.

Our reachability query problem is an abductive reasoning problem for interactive expert systems, since it seeks inputs that lead to given conclusions. Abductive logic programming, however, cannot be applied to this problem, because it works in a framework of declarative logic rules, whereas the systems that we want to query use imperative production rules.

1.5.5 Thinksheet Expert Systems

The following account of the "thinksheet" systems is directly related work, because we will be developing a query algorithm for a subclass fo these systems.

Thinksheet is a sublanguage of production systems which has been applied to the organization of complex documents [26, 27, 34]. When organized as a thinksheet production system, a complex document is able to "intelligently" query a user about

information needs and then filter itself down to an appropriate subdocument.

Definition

The working memory of a thinksheet production system is organized as a finite set of variables, each of which holds either a defined value or $\perp$. Thus the memory can be modeled as a binding. The actions of the rules are restricted to the forms $display(text)$, and $var := expr$. Furthermore, for each variable v, there is exactly one rule that assigns to v in its action. Thus we can write these rule as "if $precond_v$ then $action_v$", and we speak of the precondition and action *for* v. Specifically, the following three kinds of rules are admitted:

Question Rules: if $precond_v$ then $v := accept\_input(dom_v)$

Calculation Rules: if $precond_v$ then $v := term_v$

Conclusion Rules: if $precond_c$ then $display(text_c)$

where:

- dom_v is the set (menu) of possible answer choices
- $accept\_input$ prompts the user for an input value—but does not block the progress of the rest of the system
- $term_v$ is an expression not containing input statements
- $text_c$ is the text of conclusion c

The *dependency graph* of the rule-set is the directed graph with one node for each variable, and an edge $x \rightarrow y$ whenever x occurs in the precondition or the action of y. Finally, the dependency graph is required to be *acyclic*.

When multiple questions have preconditions that evaluate to *true*, the system allows the user to answer them in any order. This is accomplished by allowing all the rules with true preconditions to fire simultaneously, *i.e.*, by *not* using a conflict resolution strategy, and by having the input statements *accept* values from the user without blocking the rest of the system.

An interpreter for thinksheet production systems has been implemented as a spreadsheet that is augmented with a precondition facility [26]. When all of the preconditions are constant *true*, then the special case of a pure spreadsheet is obtained. The preconditions extend the spreadsheet to include the functionality of an interactive expert system, since, with preconditions, an input variable gets activated only when its precondition is true, and so only the applicable are posed to the user. By this mechanism, the extended spreadsheet provides a user-tailored dialogue, and so it can function as an interactive consultant. Also, [26] describes a form-based web interface to the thinksheet production rule interpreter.

Applications

The thinksheet production systems can express a diverse range of applications, many of which fall under the heading of "decision DAGs."

Example 1.5.5 *Figure 1.4 shows a sketch of a decision DAG that functions as a medical consultant, by asking a doctor about measurements of a blood sample, and, in case of anomalies, leading the doctor through a diagnostic chain.*

Each node of the graph represents a production rule, with question rules shown as boxes and conclusion rules as ovals. The precondition of a node is drawn above

it. For instance, the full rule the node *Ulcer* is: *if Bleeding=yes then print "Ulcer?" and accept a yes/no answer for the variable Ulcer.* An edge $v \rightarrow w$ indicates that v occurs in the precondition of w.

Initially, the only questions with preconditions evaluating to True are the roots, and so these are the ones that are first answerable by the doctor. If the doctor then supplies the value 50 for the variable *Iron*, then the question which prompts "Iron is low. Evidence of bleeding?" becomes activated. If *FolicAcid* is then specified as 10, and the answer to "Evidence of bleeding?" is specified as *No*, then the question "Malabsorption. Intestinal damage?" becomes activated, etc.

Example 1.5.6 *Figure 1.5 shows a decision DAG for a lawyer's strategy for cross-examining a witness in a courtroom.*

Decision DAGs generalize the familiar notion of decision trees. Whereas in a decision tree, each non-root node has one parent, and the entry condition for that node is a boolean function of the value assigned to the parent, in a decision DAG, the nodes may have any number of parents, and the entry condition is a function of all of the values at the parents. Specifically, a decision tree is a decision DAG in which the precondition of each non-root variable v is an atomic formula $(p\ OP\ const)$, where p is the parent variable and OP is a comparison operator.[25]

This generalization from one to multiple parents leads to a qualitative change in the semantics of the graphs. In a decision tree, since each internal node has a single

[25]Im this description of a decision tree, the internal nodes represent variables that are to be assigned by the user. This covers as a special case the decision trees where each node is a boolean test, if we regard each test as a question posed to the user, the results of which will get stored in a boolean variable for that node.

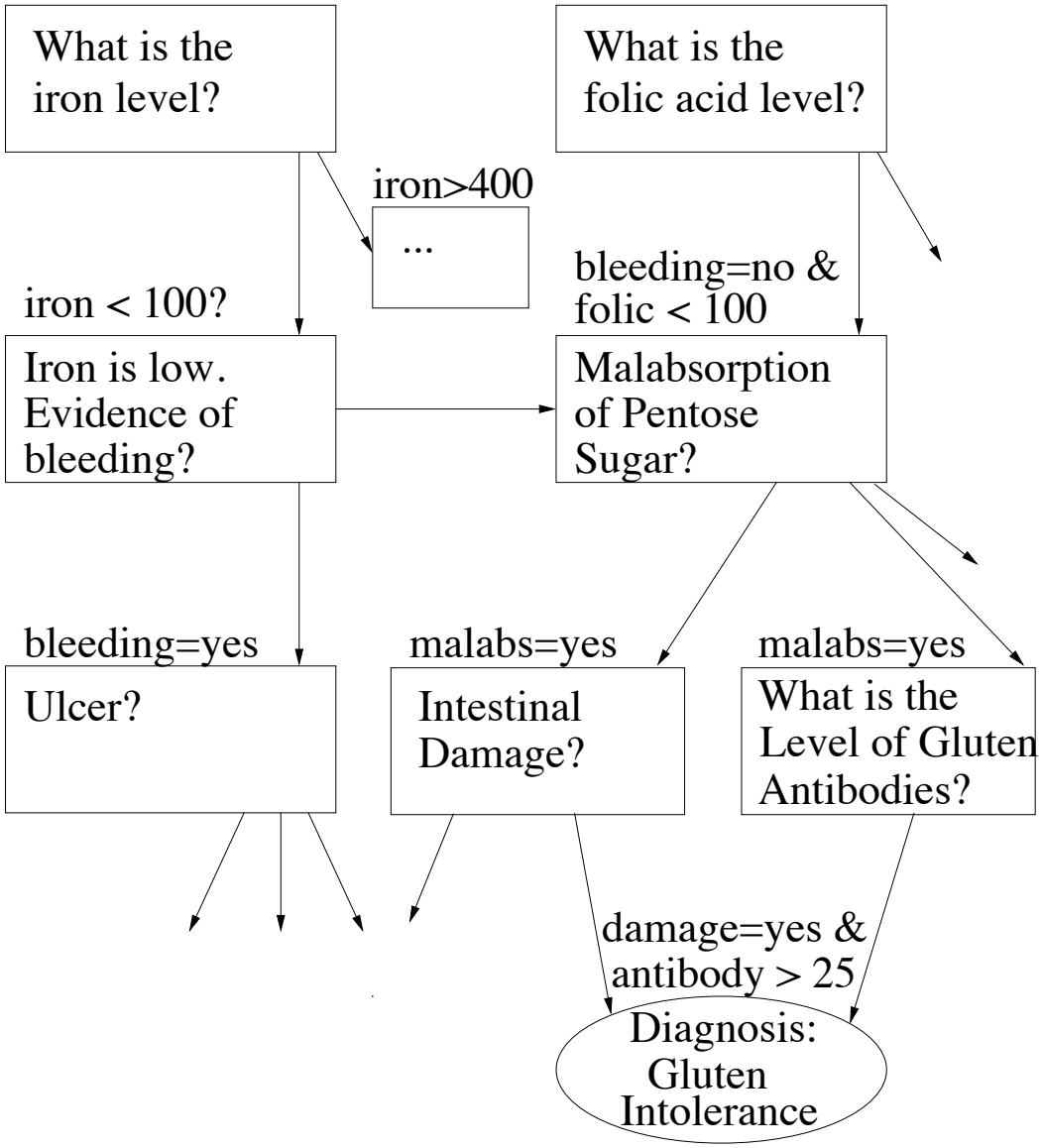
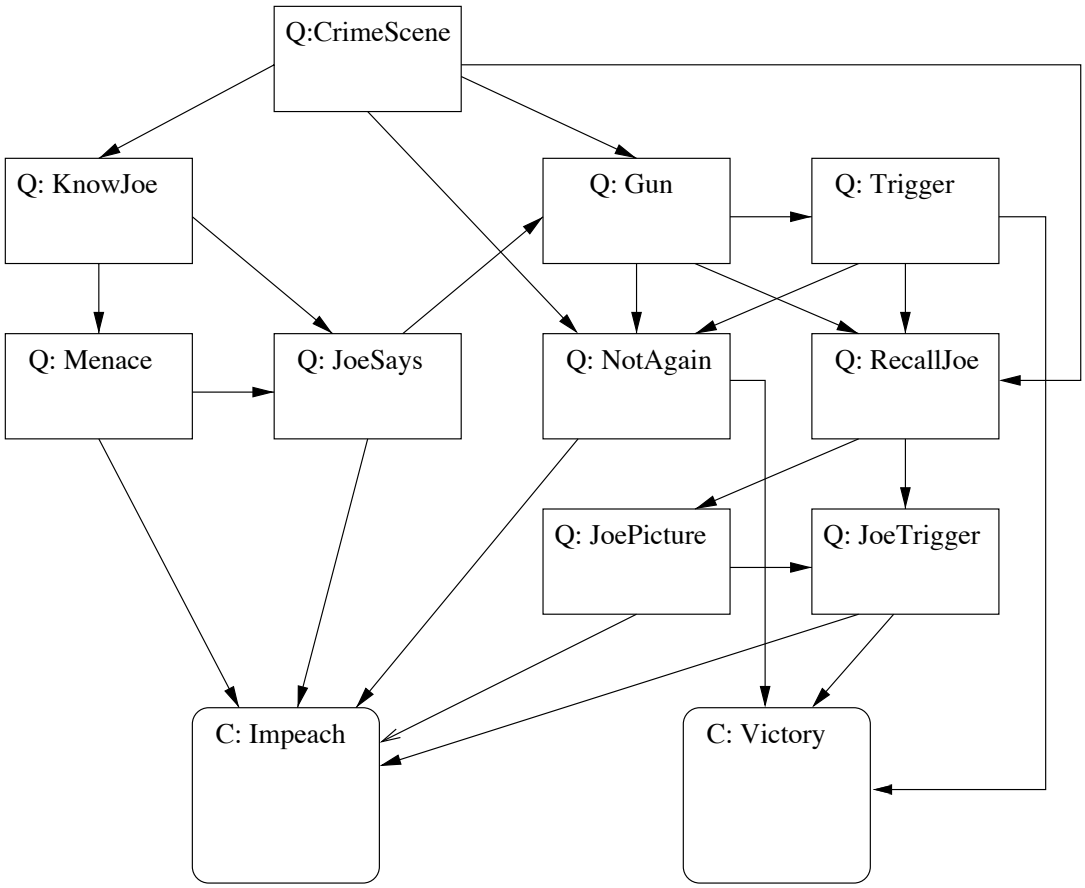


Figure 1.4: Uses of Acyclic Production Systems: Decision DAGs



See the next two figures for the contents and preconditions of these nodes.

Figure 1.5: Graph of the Trial Strategy Production System

parent, it is possible—and typical—to attach the entry condition for the node to the incoming *edge*. This leads to a sequential, control-flow perspective on the tree, where its behavior is visualized in terms of the execution of linear paths from the root towards the leaves. When we generalize to allow multiple parents, it is no longer meaningful to attach the precondition to a single incoming edge, and so the path-based control-flow model breaks down. For instance, if the precondition of z

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.

Figure 1.6: Text of Questions and Conclusions for the Trial

CrimeScence: true

Gun: CrimeScene=yes or JoeSays=yes

JoePicture: RecallJoe=no

JoeSays: KnowJoe=yes or Menace=yes

JoeTrigger: RecallJoe=yes or JoePicture=yes

KnowJoe: CrimeScene=no

Menace: KnowJoe=no

NotAgain: CrimeScene=no and (Gun=no or Trig.=no)

RecallJoe: CrimeScene=yes and (Gun=no or Trig.=no)

Trigger: Gun=yes

Impeach: Menace=no or JoeSays=no or NotAgain=no or JoePicture=no or JoeTrigger=no

Victory: NotAgain=yes or JoeTrigger=yes or Trigger=yes

Figure 1.7: Preconditions for the Nodes in the Trial

is $x < y$, then the edge from x to z in the graph does not indicate that we can "get to z from x," but the weaker statement that the condition for getting to z depends *in part* upon x. The semantics of the graphs are therefore qualitatively richer than that of decision trees.

An important application of decision DAGs is the organization of complex documents [26]. The DAGs can function as "smart documents" which ask the reader a series of relevant questions (e.g., about personal data, or topics of interest) and then filter themselves down to a report containing just the pertinent sections. To see how a decision DAG can function in this way, we can picture the conclusions,

which are presented at the leaves of the DAG, in the form of sections of text attached to the leaves. The totality of these texts, along with the questions and preconditions that connect them, constitute the smart document. As the reader answers the questions, some of the leaves become false (their preconditions become false), and some become true. The filtered report is obtained by composing all the texts at the leaves with preconditions that evaluate to True. Thus, the DAG functions as a complex document along with a semantic index—the connective logic of the questions and preconditions—that helps to guide the reader to the relevant portions of the document.

The general approach to writing such a DAG is to take the existing complex document and (1) break it into sectional units, (2) determine the parameters which will function as questions, (3) attach a precondition to each unit of text, which serves to define the conditions under which the text is applicable, and (4) attach preconditions to the questions, to specify when they are applicable. For instance, in a smart document that presents the text of an insurance policy, a fragment might contain the questions Q1 = "Are you married?", Q2 = "What is your spouse's income?", and a textual section T1 which describes the parts of the policy that apply to holders whose spouses earn over $50,000; then we would have precondition(Q2) = 'Q1=yes', precondition(T1) = 'Q2 > 50,000'.

Such DAGs are particularly useful for large documents where readers are interested in only a small portion of the document, i.e., where they read it in a "non-linear" way, with a specific query in mind. Examples include legislation and complex requirements documents. When I consult the social security law, for instance, rather than reading the whole document, the query that I have in mind is

"Which benefits apply in my case?" Ideally, after posing a handful of questions about my personal circumstances, the DAG would filter itself down to a version of the law as if it were written specifically for me.

In [26], the following applications of complex documents are described: immigration law, Social Security benefits, and telecommunications requirements.

The Effective-Value Algorithm

Although this was not its stated intention, the "effective value" algorithm, based on a 3-valued logic, which was presented in [26, 34], can function as a partial decision procedure for assignment queries to acyclic production systems.[26] The thinksheet interpreter [26] uses the effective value algorithm to determine certain questions and conclusions which are unreachable under the current binding, and others which are inevitable; the former are made to vanish from the display, and the latter are presented to the user. In this way, the system thus behaves in accordance with the features of a query tool that we outlined in Section 1.1.4 for the restricted case where the query formula is of the form $var_1 = val_1 \wedge \cdots \wedge var_k = val_k$.

The algorithm uses a logic of three truth-values that are ordered on a scale of "definedness":

$$\mathsf{False} < \mathsf{Possible} < \mathsf{True}$$

Here is the key definition.

Definition 1.5.7 ($eff_s(e, b)$, Effective Value of an Expr. in a Binding) *For an acyclic production system s with universe of discourse D, an expression e,*

[26] Recall from Section 1.1.4 that the assignment queries ask about the reachability/inevitability of questions and conclusions under the query formula $\wedge b$, where b is the current binding.

and a binding b, define $\mathit{eff}_s(e,b) \in D \cup \{\mathsf{False}, \mathsf{Possible}, \mathsf{True}\}$ *according to Figure 1.8.*

$$
\begin{array}{lcll}
\mathit{eff} & : & \mathit{Expr} \times \mathit{Binding} \rightarrow D \cup \{\mathsf{False}, \mathsf{Possible}, \mathsf{True}\} & \\
\mathit{eff}(v,b) & = & \mathsf{if}\ \mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}\ \mathsf{then} & ;\ \text{question var v} \\
 & & \quad \mathsf{if}\ b(v) \in \mathit{dom}_v\ \mathsf{then}\ b(v)\ \mathsf{else}\ \mathsf{Possible} & \\
 & & \mathsf{else}\ \mathit{eff}(\mathit{precond}_v, b) & \\
\mathit{eff}(w,b) & = & \mathsf{if}\ \mathit{eff}(\mathit{precond}_w, b) = \mathsf{True}\ \mathsf{then}\ \mathit{eff}(\mathit{term}_w, b) & ;\ \text{derived var w} \\
 & & \mathsf{else}\ \mathit{eff}(\mathit{precond}_v, b) & \\
\mathit{eff}(c,b) & = & c^I & ;\ \text{constant c} \\
\mathit{eff}(g(e_1,\ldots,e_n),b) & = & \mathsf{if}\ \exists i\ s.t.\ \mathit{eff}(e_i,b) = \mathsf{False}\ \mathsf{then}\ \mathsf{False} & ;\ \text{function} \\
 & & \mathsf{else\ if}\ \exists i\ s.t.\ \mathit{eff}(e_i,b) = \mathsf{Possible}\ \mathsf{then}\ \mathsf{Possible} & ;\ \text{or} \\
 & & \mathsf{else}\ g^I(\mathit{eff}(e_1,b),\ldots,\mathit{eff}(e_n,b)) & ;\ \text{predicate g} \\
\mathit{eff}(\phi_1 \wedge \phi_2, b) & = & \mathsf{min}(\mathit{eff}(\phi_1,b),\ \mathit{eff}(\phi_2,b)) & \\
\mathit{eff}(\phi_1 \vee \phi_2, b) & = & \mathsf{max}(\mathit{eff}(\phi_1,b),\ \mathit{eff}(\phi_2,b)) & \\
\mathit{eff}(\mathit{not}(\phi), b) & = & \mathsf{neg}(\mathit{eff}(\phi,b)) &
\end{array}
$$

Key:

D = universe of discourse

I = interpretation for constraint domain

$\mathsf{False} < \mathsf{Possible} < \mathsf{True}$

neg exchanges True and False, and leaves Possible unchanged

Figure 1.8: Definition of Effective Value

By memoizing the effective values of the variables, the effective values of all the variables and preconditions can be computed by a single traversal of the production system. Further optimizations are described in [26] which incrementally maintain the computed effective values when the current binding is changed by a single

assignment from the user.

The significance of the effective-value algorithm for the processing of assignment queries, which can be proven with some effort, is that for formulas f and attainable bindings b: (1) $\mathit{eff}_s(f, b) = \mathsf{False} \Rightarrow f$ is $(\wedge b)$-unreachable, and (2) $\mathit{eff}_s(f, b) = \mathsf{True} \Rightarrow f$ is $(\wedge b)$-inevitable.[27] (Since our query algorithm works for a far more general class of query formulas, and it is not based upon the thinksheet effective value algorithm, we omit the proof of this statement.)

It is natural to wonder whether this 3-valued notion of effective value is solely a technical device for a useful algorithm, or whether it has a deeper significance, *i.e.*, does it express semantic principles with objective content? In [26], Piatko makes some informal suggestions about an underlying semantics:

> ... *false* and *possible* represent meta-statements about a particular variable ... [if the effective value is *false*] it means that the value is unnecessary, or not applicable ... [whereas] *possible* can represent a node value that is *undefined* i.e. the node's precondition may be *true* and thus it is applicable, but no value has been assigned to it yet. This undefined state is propagated to the other nodes that depend on this node, so they too must have the [effective] value *possible*.

In this thesis, for our analysis for queries and development of a query algorithm, we will not be using the notions of 3-valued logic and effective values. The reasons

[27] The incompleteness of the effective value algorithm for assignment queries follows from the fact that, at least for special cases, the converses of these two implications are not true. For instance, (1) the formula '$x = x$' is inevitable under the empty binding, but $\mathit{eff}(\text{'}x = x\text{'}, \{\}) = \mathsf{Possible} \neq \mathsf{True}$, and (2) '$x \neq x$' is unreachable, but $\mathit{eff}(\text{'}x \neq x\text{'}, \{\}) = \mathsf{Possible} \neq \mathsf{False}$.

are that (1) these constructions themselves need to be explained, and (2) since they are predicated upon the internal structures of a particular kind of expert system—the precondition formulas—they are not suitable as first concepts for our theory of queries to generic expert systems. Instead, we take the notion of attainable bindings as the point of departure. Nevertheless, as we show in the final chapter, the theory that we develop in this thesis will also provide a partial explanation for the meaning of the effective values.

1.6 The Expert Systems to be Queried

We now specify a restriction of the thinksheet production systems. This provides the case of the query problem that we will solve in this thesis. In the general thinksheet production systems, the preconditions and calculation expressions may be arbitrary program fragments. To make the queries decidable, we limit the preconditions to boolean formulas, and the calculation expressions to logic terms. Furthermore, from this point until chapter 4, we assume that the preconditions are monotone boolean formulas.

This class of production systems is still broad enough to cover the decision DAGs, classical spreadsheets, and precondition-based expert system spreadsheets. For instance, the examples we presented in Figures 1.4 and 1.5 are covered by this language. Here is a complete, formal specification of the systems to be queried.

Definition 1.6.1 (Acyclic Production System) *Let X be a constraint-domain with opposite predicates. An* acyclic production system *s consists of a set of question variables V_s, derived variables W_s, and conclusions C_s, plus a set of rules of*

the following forms:

Question Rules: *if* $precond_v$ *then* $v := input(dom_v)$

Calculation Rules: *if* $precond_w$ *then* $w := term_w$

Conclusion Rules: *if* $precond_c$ *then* $display(text_c)$

where:

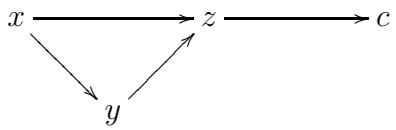
1. *There is exactly one question rule for each* $v \in V_s$*, one calculation rule for each* $w \in W_s$*, and one conclusion rule for each* $c \in C_s$
2. $\forall x \in V_s \cup W_s \cup C_s$*,* $precond_x$ *is a monotone* Σ_X*-formula with variables in* $V_s \cup W_s$
3. *for each* $v \in V_s$*, there is an associated formula* $indom_v$ *that it satisfiable and has* v *as its only free variable*
4. $dom_v \neq \{\}$*, the domain of values assignable to* v*, is defined as the set of values for* v *that satisfy* $indom_v$*. For instance, a domain of* {*1,2*} *is represented by the indom formula '*$v = 1 \vee v = 2$*', and a domain that equals the universe of discourse is represented by '*$v = v$*'*
5. $term_w$ *is a* Σ_X*-term with variables in* $V_s \cup W_s$
6. *for each* $w \in W_s$*, we define* $indom_w$ *to be the constraint* $w = term_w$
7. $G(s)$*, the* dependency graph *for* s*, is defined as the directed graph over the nodes* $V_s \cup W_s \cup C_s$*, with an edge* $x \rightarrow y$ *if* x *occurs in* $precond_y$ *or in* $term_y$
8. $G(s)$ *must be acyclic*

Example 1.6.2 (Simple Acyclic Production System S_{xyz})

Let $V_s = \{x, y\}$, $W_s = \{z\}$, $C_x = \{c\}$, and define the rules by the following table:

node n	dom_n	$indom_n$	$precond_n$	$term_n$	$text_n$
x	$\{-2,-1,0,1,2\}$	$x = -2 \vee \cdots \vee x = 2$	*true*		
y	$\{-2,-1,0,1,2\}$	$y = -2 \vee \cdots \vee y = 2$	$x > 0$		
z		$z = x * y$	$y < 0$	$x * y$	
c			$z = -2$		*'z is -2'*

Here is the dependency graph:



In this system, the conclusion c is unreachable under the query $x + y = 0$, and a conclusion with precondition $x = 2$ would be inevitable under the query $z = -4$.

These systems work by maintaining a current binding b for the variables $V_s \cup W_s$, which is initialized to $\{\}$. Whenever a precondition is satisfied, then the rule fires. If the rule is for question variable v, then the user is permitted to assign to v. If the rule is for derived variable w, the system makes the assignment to w. We give the exact definition as follows.

Definition 1.6.3 (Operational Description of the Acyclic Systems)

For an acyclic production system s, and an attainable binding $b \in A_s$, define:

$$say_s(b) = \{c \in C_s \mid (b \models precond_c)\}$$
$$ask_s(b) = \{v \in V_s \mid (b \models precond_v) \wedge b(v) = \bot\}$$
$$calc_s(b) = \{(w, val(term_w, b)) \in W_s \times D_s \mid (b \models precond_w) \wedge b(w) = \bot\}$$

We now give four examples of queries to the acyclic production systems.

Example 1.6.4 *In the medical decision DAG shown in Figure 1.4, is the diagnosis Gluten-Intolerance reachable under the condition '$Iron = 100 \vee FolicAcid = 100$'?*

Example 1.6.5 *In the trial strategy shown in Figure 1.5, is conclusion Victory reachable if the witness says No to the question Gun?*

Example 1.6.6 *In the trial strategy, let p_v be the precondition of the conclusion Victory, and p_i be the precondition of Impeach. Are there any inputs which lead to the conclusion $p_v \wedge p_i$? This would indicate an error in the design of the system, since Victory and Impeach were intended to be mutually exclusive outcomes.*

Example 1.6.7 *Let p_i be the precondition of Impeach in the trial strategy DAG. Under the query formula p_i, is the conclusion formula 'CrimeScene=no' inevitable? I.e., is the answer CrimeScene=no necessary in order to reach conclusion Impeach?*

1.7 Issues with Negation

We now explain why negations in the preconditions pose some subtle complexities. As we defined it, $say(b)$ consists of all those preconditions p for conclusions such that $b \models p$. But how are we to define the relation $b \models p$, in the case that p may

contain negations? More specifically, how are we to define the relation $b \models not(f)$, for a formula f? It would seem natural enough to define $b \models not(f)$ to mean $not(b \models f)$, but when the bindings b may contain $\perp$ values, this leads to anomalies. For instance, suppose the precondition formula p is $not(x = 0)$, and that the binding b is the initial binding $\{\}$. Then b does not satisfy p, yet it doesn't make sense for $not(x = 0)$ to be *true* when x still remains to be specified.

One approach is to say that, since '$x = 0$' is $\perp$ in the empty binding, then, if we assume that the unary negation operator is "strict," i.e., that $not(\perp) = \perp$, the evaluation of $\{\} \models not(x = 0)$ should itself evaluate to $\perp$, and so $\{\}$ would not satisfy $not(x = 0)$. This approach is equivalent to replacing $not(x = 0)$ with $x \neq 0$, and, more generally, replacing each boolean formula f by its negation-free equivalent f^+, as per Definition 1.4.4. Thus, by viewing $not(f)$ as an abbreviation for the monotone formula f^-, we get the strict negation operator within the simpler framework of production systems with monotone preconditions. This approach also has the advantage of simplifying the query problem to monotone production systems. That is what we will do.

There is, however, no law of thought that prevents us from considering an alternative kind of negation that is not strict. There is no justification for the *a priori* requirement that the logical operators be strict. In fact, as the following example shows, ordinary disjunction behaves in a nonstrict manner. Suppose that Joe is 31 and single, and that he reads the following clause in a contract: if your spouse is employed or you are over 20, then you are eligible for benefit X, i.e., if (*spouseEmployed* or $Age > 20$) then benefit. The benefit clearly applies to Joe. Now, for Joe's case, the rule is tantamount to: if ($\perp$ or *true*) then benefit. Hence, ($\perp$ or *true*)

must be *true*. Thus, since the $\perp$ argument didn't cause the disjunction to be $\perp$, disjunction is not strict. Still, of course, the burden of proof would remain on us to show that a nonstrict negation operator is not an arbitrary formalism, *i.e.*, that it has an objective significance.

In a closing chapter, we will present a semantics for a non-strict negation operator. Its implementation, however, comes at the cost of a more complicated operational semantics. The theory that we will develop for analyzing queries to the monotone systems will provide the basis for extending the theory and query algorithm to cover preconditions with general boolean formulas.

1.8 Thesis Contribution

This thesis, parts of which were presented in [40], is, to our knowledge, the first exploration into the subject of query-processing for interactive expert systems. The paradigm we contribute is the problem of querying the knowledge in the rules that the system uses to question a user. We single out the following useful kind of query: Which conclusions are reachable, and which are inevitable, under a hypothetical constraint on the user input? Although reachability problems have been widely studied in other contexts, *e.g.*, the halting problem, we believe this to be the first investigation of reachability problems for interactive rule-based expert systems. This query paradigm raises the challenge for researchers to find expert system languages that will admit a reachability query algorithm.

Towards this aim, we present a theory of queries to a broad class of expert systems. We define these systems by a language-independent monotonity requirement

on the questioning mechanism: once a question is posed by the system, then, even if the user answers other questions, it should not be retracted until it is answered. For these systems, we develop a logic semantics that is relativized to the content of the knowledge-rules. In this semantics, we define a series of useful concepts, such as the core of applicable assignments within a binding, the applicability and inapplicability of terms under a binding, and the relationships under which a binding justifies the acceptance, or justifies the rejection, of a formula by the system. We then use these concepts to re-express the queries in a form that may be conducive to algorithmic solutions. This whole theory, which includes the semantic concepts, and the analysis of queries, offers the benefit of language-independence. We hope that this generic analysis may offer guidance for the discovery of query algorithms for specific languages.

We have discovered one such algorithm, which we believe to be the first reachability query algorithm for a language of interactive expert systems. The language is a restriction of production systems to a simple acyclic form which has a diverse range of applications. When the preconditions are conjunctive, the data complexity of the algorithm, measured in units of constraints, is $O(n^2)$, where n is the size of the input production system. We also give a query optimization for the frequently occurring pattern of local decision trees in the rule systems. When general datalog methods are used for the final, satisfaction-testing phase of the query algorithm, the number of constraints that must be solved is $O(n^2)$, where n is the size of the local trees. With our optimization, it is $O(n)$. Finally, we have used the algorithm to build a query tool for acyclic production systems. To our knowledge, they are the first interactive expert systems that can be *queried* about their conclusions.

Chapter 2

A Theory of Queries to Expert Systems

In this chapter, we define the "regular" systems by a monotonicity requirement on the question-asking mechanism, and then we proceed to analyze semantic notions of logic, such as applicability and inapplicability, in the context of the rules of an expert system. The regular systems include the acyclic production systems, for which we will later give an effective query algorithm. The keys to understanding the query algorithm will be given by the theorems, which we present at the culmination of this chapter, about the nature of queries to regular systems.

2.1 An Axiomatic Approach to the Theory of Expert Systems

Our theory of expert systems, which will take its basic terms from the generic formal model presented in Section 1.1.2, is *axiomatic* (generic) because its basic terms are fully abstracted from the internal mechanisms of the systems. Since this kind of theory is built from terms that refer only to the external characteristics of the systems, its results are independent of the languages used to program the systems. This can lead to generality of results. Our query problem, for instance, applies with equal validity to interactive expert systems written with production rules as it does to systems written in BASIC. Furthermore, the generic concepts may illuminate the essential characteristics of the systems. For instance, we may take up the analysis of systems that have a monotonic conclusion function say_s.

For a well-known illustration of the benefits of the axiomatic approach, consider the ordinary theory of real arithmetic [23]. This theory, which comprises the "field" axioms that define the relations among the arithmetic operations, plus the axioms for a complete total order, offers genericity, because the operands, *i.e.*, the numbers, are treated as an abstract data type. The axioms determine, up to isomorphism, all of the properties of the real numbers. A constructive approach to real arithmetic, in contrast, can be achieved with infinite models that satisfy the axioms, e.g., the system of infinite decimal expansions. In many instances, the axioms can offer an approach that is conceptually simpler than the model-based approach. For instance, we often make deductions by repeated applications of the transitivity axiom on the ordering, without referring to infinite decimal expansions. In this framework, we

may attain generality of results by the natural method of analyzing subsets of the axioms. For example, if we drop the completeness axiom, then we get a theory whose results cover the rationals as well as the reals.

The axiomatic theory of "regular" interactive systems, which we present in this chapter, is characterized by the axiom that any question posed by the system will not be retracted until it is answered by the user. Our analysis of the regular systems will yield both (1) useful concepts, such as generic definitions of applicability and inapplicability, and (2) a theoretical generality that extends beyond the systems for which this thesis provides a query algorithm. The extent of this generality, *i.e.*, the class of programs which conform to the axioms of regular systems, is a topic for further research.

2.2 Regular Expert Systems

We now introduce a notion of "regular" expert systems, which means, roughly, that the specification of new information cannot cause a questions or computations to be revoked by the system. The assumption of regularity has many ramifications, and it will serve as the cornerstone of our query theory.

Definition 2.2.1 (Regular System)
Recall from the terminology on page 11 that A_s is the set of attainable bindings for the system s. Then a system s is regular *if both of the following conditions hold:*

1. $\forall\, b, b' \in A_s \;:\; b \subseteq b' \Rightarrow (ask_s(b) \subseteq vars(b') \cup ask_s(b'))$

2. $\forall\, b, b' \in A_s \;:\; b \subseteq b' \Rightarrow (calc_s(b) \subseteq b' \cup calc_s(b'))$

Thus, the regularity assumption means that (1) if a question is asked in an attainable binding, then it will either be answered or asked in every attainable extension of that binding, and (2) if a value is calculable in an attainable binding, then it will either be calculated or calculable in every attainable extension. The next proposition gives an alternative, algebraic characterization of the regular systems.

Proposition 2.2.1 *A system s is regular iff the following two functions are monotonic with respect to inclusions of bindings:*

1. $\lambda b \in A_s \;.\; vars(b) \cup ask_s(b)$
2. $\lambda b \in A_s \;.\; b \cup calc_s(b)$

The next proposition shows that the acyclic production systems that we will be querying—including the example decision DAGs that we have seen so far—are regular systems.

Proposition 2.2.2
A production system where precondition formulas are monotone is a regular interactive system.

For purposes of comparison, we exhibit a system that is not regular.

Example 2.2.2 (A Non-Regular Interactive System) *Suppose that:*

1. $V_s = \{x, y\}$
2. *Initially, x and y are answerable*
3. *If x is answered first, then y remains answerable*

4. If y is answered first, then the session terminates, i.e.*, ask becomes empty*

Here is the formal specification for this system:

$$ask_s(b) = \begin{cases} \{x, y\} & \textit{if } vars(b) = \{\} \\ \{y\} & \textit{if } vars(b) = \{x\} \\ \{\} & \textit{if } vars(b) = \{y\} \\ \{\} & \textit{if } vars(b) = \{x, y\} \end{cases}$$

This system fails to be regular because, starting from the initial binding, where both variables are answerable, the answering of y causes the question x to be revoked. Formally, let $b = \{\}$, and b' be any attainable binding such that $vars(b) = \{y\}$. Then $b \subseteq b'$, yet $ask(b) = \{x, y\}$ is not a subset of $ask(b') \cup vars(b') = \{y\}$. Thus, the regularity condition does not hold, because x has been revoked in the transition from b to b'.

This system illustrates the fact that in a non-regular system, a final binding may not be maximal among the attainable bindings. For example, $\{y=1\}$ is a final binding that is properly contained in the attainable binding $\{x = 1, y = 1\}$. On the other hand, in all systems—whether regular or not—the maximal attainable bindings are final. This is because a non-final binding is by definition extendible, *i.e.*, non-maximal.

We now show that the regularity condition ensures that the maximal attainable bindings coincide with the final bindings. The proof uses the following lemma, which implies that in a regular system, if b and b' are attainable bindings such that $b \subseteq b'$, then there is an attainable sequence of assignments that converts b to b'.

Lemma 2.2.3 *For a regular system s,*

$\forall\, b, b' \in A_s \;:\; (b \subseteq b' \wedge b \neq b') \Rightarrow (ask_s(b) \cap vars(b') \neq \{\})$ *or* $(calc_s(b) \cap b' \neq \{\})$

Proposition 2.2.4 *For a regular system s,*

$\forall\, b \in A_s \;:\; b$ *is maximal in* $A_s \Leftrightarrow b \in F_s$

2.3 The Core of a Binding

We now show that in a regular system, every binding contains a "core" sub-binding that constitutes the valid part of the binding. This core is obtained by removing from the binding all of the assignments to inapplicable variables. For instance, in a system s where the preconditions are realistic, we will have that:

$$core_s(\{Married = no,\ SpouseAge = 30\}) = \{Married = no\}$$

We will prove that for regular systems, the core of a binding is well-defined, and equals the maximum attainable sub-binding.

We now give an effective procedure for computing $core_s(b)$, which uses a simulation of s on the data in b in order to produce the maximum attainable sub-binding. The procedure can be described by the following experiment. We start a session with s, and we use the data in b as a pool of assignments that may potentially added to a binding that will be built up to be the core of b. Whenever s asks us for the value for the question variable v, we check the value for v that is provided by b. If $b(v)$ is defined and belongs to the domain of v, then (1) we supply the $b(v)$ as the answer value for v, and (2) we add the pair $(v, b(v))$ to the core of b. Otherwise, we supply no answer value for v, and leave the value of v in $core(b)$ at $\perp$. Whenever s wants to assign the value z to a derived variable w, then we check the value $b(w)$. If

$b(w) = z$ then we add the pair (w, z) into the core of b. Otherwise, we add nothing into the core of b, and "prevent" the system from making that assignment. We now give the formal definition.

Definition 2.3.1 (Core of a Binding, $core_s(b)$)
For a regular system s, and a binding b, define $core_s(b) \subseteq b$ according to the procedure shown in Figure 2.1.

```
binding core_s(binding b)
begin
   binding b' = {}
   loop
      if ∃ v ∈ ask_s(b') s.t. b(v) ∈ dom_v then b'(v) = b(v)
      else if ∃ (w,z) ∈ calc_s(b') s.t. b(w) = z then b'(w) = b(w)
      else break
   end
   return b'
end
```

Figure 2.1: The Core of a Binding in a Regular System

We will show that for regular systems, the core is well-defined. I.e., we show that for regular systems, the non-determinism of the algorithm in Figure 2.1 doesn't affect the binding that it computes. We will need a lemma, which says that the union of two "compatible" attainable bindings is itself an attainable binding.

Definition 2.3.2 (Compatible Bindings) *Bindings b_1, b_2 are* compatible *if:*

$\forall v \in vars(b_1) \ \cap \ vars(b_2) \ : \ b_1(v) = b_2(v)$

Lemma 2.3.1 *For a regular system s,*

$(b_1, b_2$ *compatible and* $b_1, b_2 \in A_s) \Rightarrow b_1 \cup b_2 \in A_s$

I.e., the attainable bindings are closed under compatible unions.[1]

Corollary 2.3.2 *For a regular system s and a binding b, within b there is a unique maximal attainable sub-binding, viz., the union of the attainable sub-bindings of b,* $\cup\{b' \in A_s \mid b' \subseteq b\}$.

Theorem 2.3.3 *For a regular system s,* $core_s(b)$ *is uniquely defined, and it equals the maximum attainable sub-binding* $\cup\{b' \in A_s \mid b' \subseteq b\}$.

We have therefore succeeded in defining the core of applicable assignments within a binding, using the operational logic of a regular system. We now state some algebraic properties of the operator $core_s$. The first proposition shows that the fixpoints of the core operator (the bindings b such that $b = core_s(b)$) are precisely the attainable bindings.

Proposition 2.3.4 *For a regular system s:*

$$A_s = \mathit{fixpoints}(core_s) = \mathit{image}(core_s)$$

[1]On the other hand, the attainable bindings are not necessarily closed under compatible intersections. Take for example the production system with variables $\{x, y, z\}$, where x and y have precondition *true*, and z has precondition '$x = 0 \vee y = 0$'. Consider the bindings $\{x = 0, z = 0\}$ and $\{y = 0, z = 0\}$, which are attainable and compatible. Their intersection, $\{z = 0\}$, is not attainable.

The next proposition shows that $core_s$ is a closure operator.

Proposition 2.3.5 *The function $core_s$ is a downward closure operator,* i.e.*, $core_s$ is monotonic, decreasing and idempotent:*

- $b_1 \subseteq b_2 \Rightarrow core_s(b_1) \subseteq core_s(b_2)$
- $core_s(b) \subseteq b$
- $core_s(core_s(b)) = core_s(b)$

The following definition gives a useful notation for the final extensions of the core of a binding.

Definition 2.3.3 (Final Extensions $F_s(b)$)
For a regular system s and binding b, define:

$$F_s(b) \equiv \{b' \in F_s \mid b' \supseteq core_s(b)\}$$

Proposition 2.3.6 *For a regular system s and binding b,*
$F_s(b) \neq \{\}$

Proposition 2.3.7 *For a regular system s and binding b,*
$b \in F_s \Leftrightarrow F_s(b) = \{b\}$.

2.4 The Affirmation and Disaffirmation of Formulas

We now introduce a pair of logical relations between bindings b and formulas f that is induced by a regular system s: a positive relation $b \models^+_s f$, called b *affirms* f, where

the binding justifies the acceptance of the formula by the system, and a negative relation $b \models\!\!\!=_s f$, called b *disaffirms* f, where the binding justifies the rejection of the formula by the system.

Definition 2.4.1 (Affirmation $b \models\!\!\!\stackrel{+}{=}_s f$ and Disaffirmation $b \models\!\!\!\stackrel{-}{=}_s f$)
Recall from Definition 2.3.3 that $F_s(b)$ is the set of final extensions of the core of b. Then for a regular system s, formula f, and binding b, define:

$$b \models\!\!\!\stackrel{+}{=}_s f \Leftrightarrow (\forall\, b' \in F_s(b) \,:\, b' \models f)$$
$$b \models\!\!\!\stackrel{-}{=}_s f \Leftrightarrow (\forall\, b' \in F_s(b) \,:\, b' \not\models f)$$

Thus, $b \models\!\!\!\stackrel{+}{=}_s f$ says that f is satisfied by all of the final extensions of the core of b, and $b \models\!\!\!\stackrel{-}{=}_s f$ says that f is satisfied by none of these final extensions. Intuitively, $b \models\!\!\!\stackrel{+}{=}_s f$ means that the applicable assignments within b guarantee that f will eventually become satisfied, and $b \models\!\!\!\stackrel{-}{=}_s f$ means that these assignments guarantee that f will never be satisfied in a final outcome. This is why we say that $b \models\!\!\!\stackrel{+}{=}_s f$ means the data in the binding justifies the acceptance of the formula f by the system, and that $b \models\!\!\!\stackrel{-}{=}_s f$ justifies the rejection of f by the system.

It may happen that a binding b neither affirms nor disaffirms a formula f. This occurs when some of the final extensions of $core(b)$ satisfy f and some do not satisfy f. On the other hand, the next proposition states that a binding cannot both affirm and disaffirm a given formula.

Proposition 2.4.1 *For a regular system s and formula f,*
$\nexists b \in B_s \; s.t. \; (b \models\!\!\!\stackrel{+}{=}_s f) \wedge (b \models\!\!\!\stackrel{-}{=}_s f)$

Example 2.4.2 *Consider the system s with the two variables $\{M, SA\}$, where M stands for the question "Are you married?" and SA for the question "What is*

your spouse's age?". Let M be a root question, and let SA have the precondition '$M = yes$'. Suppose that the formula f is '$SA > 30$'. Then we have the following:

	binding b	$core_s(b)$	$b \models^{+}_{s} f$	$b \models^{-}_{s} f$	*status of f*
1.	$\{\}$	$\{\}$	*No*	*No*	*indeterminate*
2.	$\{M = Yes,\ SA = 50\}$	$\{M = Yes,\ SA = 50\}$	*Yes*	*No*	*affirmed*
3.	$\{M = Yes,\ SA = 20\}$	$\{M = Yes,\ SA = 20\}$	*No*	*Yes*	*disaffirmed*
4.	$\{M = No\}$	$\{M = No\}$	*No*	*Yes*	*disaffirmed*
5.	$\{M = No,\ SA = 50\}$	$\{M = No\}$	*No*	*Yes*	*disaffirmed*

Explanation. *Since this system has no derived variables, $calc_s$ maps every binding to itself. Call the bindings in the first column $b_1, \ldots, b_5$. Observe that $b_1, \ldots, b_4$ are attainable, and b_5 is unattainable. Bindings b_2, b_3 and b_4 are final.*

1. *$core(b_1)$ is $\{\}$, and $\{\}$ has final extensions that satisfy f (e.g., b_2) and final extensions that do not satisfy f (e.g., b_3). Hence b_1 is indeterminate.*

2. *$core(b_2)$ has just one final extension, namely, b_2. Since $b_2 \models f$, f is satisfied in every final extension of $core(b_2)$. Hence $b_2 \models^{+}_{s} f$.*

3. *$core(b_3)$ has just one final extension, namely, $core(b_3)$. Since $core(b_3) \not\models f$, f is unsatisfied in every final extension of $core(b_3)$. Hence $b_3 \models^{-}_{s} f$.*

4. *$b_4 \models^{-}_{s} f$, by the same argument as (3)*

5. *$b_5 \models^{-}_{s} f$, by the same argument as (3)*

We may distinguish between two contrasting modes of disaffirmation. In the Example 2.4.2, whereas the formula '$SpouseAge > 30$' is disaffirmed by the binding

$\{Married = Yes,\ SpouseAge = 20\}$ because it is *false*, it is disaffirmed by the binding $\{Married = No\}$ because it is *inapplicable*. The notion of disaffirmed formulas emphasizes the negative aspect common to these formulas: given the logic of the rules, the system has grounds for rejecting all such formulas, because they can never become true.

The next proposition shows that for final bindings, affirmation coincides with satisfaction, and disaffirmation coincides with dissatisfaction.

Proposition 2.4.2 *For a regular system s, binding $b \in F_s$, formula f, and conclusion $c \in C_s$:*

1. $b \models\!\!\!\models^{+}_{s} f \Leftrightarrow b \models f$
2. $b \models\!\!\!\models^{-}_{s} f \Leftrightarrow b \not\models f$
3. $b \models\!\!\!\models^{+}_{s} c \Leftrightarrow c \in say_s(b)$
4. $b \models\!\!\!\models^{-}_{s} f \Leftrightarrow c \notin say_s(b)$

We will now extend the notation of affirmation and disaffirmation to include the forms $b \models\!\!\!\models^{+}_{s} c$ and $b \models\!\!\!\models^{-}_{s} c$, where c is a *conclusion*. The basic idea remains unchanged.

Definition 2.4.3 (Affirmation $b \models\!\!\!\models^{+}_{s} c$ and Disaffirmation $b \models\!\!\!\models^{-}_{s} c$)
For a regular system s, conclusion c, and binding b, define:

$$b \models\!\!\!\models^{+}_{s} c \Leftrightarrow (\forall\, b' \in F_s(b)\ :\ c \in say_s(b')$$
$$b \models\!\!\!\models^{-}_{s} c \Leftrightarrow (\forall\, b' \in F_s(b)\ :\ c \notin say_s(b'))$$

To summarize, we have defined the relations $b \models^{\pm}_s f$ and $b \models^{=}_s f$, which are analogous to the satisfaction and dissatisfaction relations $b \models f$, $b \not\models f$ of first-order logic. The difference is that with the affirmation and disaffirmation relations, the binding and the formula are mediated by the rules of an expert system.

2.5 Applicable and Inapplicable Terms

We now define the concepts of the applicability, inapplicability, and possible applicability of a term in a binding for a regular system. These definitions use a special case of the affirmation and disaffirmation relations $b \models^{\pm}_s f$, $b \models^{=}_s f$.

Definition 2.5.1 (Applicable and Inapplicable Terms)
For a regular system s, binding b, and term t, define:

$$\begin{array}{ll} t \text{ is } \textit{applicable} \text{ in } b & \text{if } b \models^{\pm}_s (t = t) \\ t \text{ is } \textit{inapplicable} \text{ in } b & \text{if } b \models^{=}_s (t = t) \\ t \text{ is } \textit{possibly-applicable} \text{ in } b & \text{otherwise} \end{array}$$

In words, a term is applicable in a binding when it is defined in every final extension of the core of the binding, and it is inapplicable when it is undefined in every final extension of the core. For the case of final bindings, it is easy to show that this definition specializes to the following simple form: a term is applicable in a final binding iff it is defined in that binding, and it is inapplicable iff it is undefined.

Example 2.5.2 *Consider the system s of Example 2.4.2, with variables $\{M, SA\}$*

for the questions Married and SpouseAge. Let t be the term SA + 3.

binding b	$core_s(b)$	*status of t*
$\{\}$	$\{\}$	*possibly applicable*
$\{M = Yes,\ SA = 50\}$	$\{M = Yes,\ SA = 50\}$	*applicable*
$\{M = Yes\}$	$\{M = Yes\}$	*applicable*
$\{M = No\}$	$\{M = No\}$	*inapplicable*
$\{M = No,\ SA = 50\}$	$\{M = No\}$	*inapplicable*

2.6 Analysis of Queries to Regular Systems

We now use the affirmation and disaffirmation relations to analyze the structure of queries. Our first theorem shows that an algorithm capable of determining the affirmability and disaffirmability of formulas and conclusions will provide a sufficient basis for a reachability/inevitability query algorithm.

Theorem 2.6.1 *For a regular system s, conclusion c, formula q and variable v,*

- *c is q-reachable* $\Leftrightarrow (\exists\, b \in B\ :\ b \models^{\pm}_s q$ *and* $b \models^{\pm}_s c)$
- *c is q-inevitable* $\Leftrightarrow (\nexists\, b \in B\ :\ b \models^{\pm}_s q$ *and* $b \models^{-}_s c)$
- $(b \models^{\pm}_s q$ *and* $b \models^{\pm}_s c) \Rightarrow$

  each binding in $F_s(b)$ *witnesses the q-reachability of c*

In words, c is q-reachable iff q and f are affirmable by the same binding, and c is q-inevitable iff there is no binding in which q is affirmed and c is disaffirmed.[2]

[2]These statements are structurally analogous to the definitions of q-reachability and q-inevitability. The significant difference is that whereas the original definitions refer to the ex-

The next theorem strengthens the previous theorem, by a weakening of its premises. Specifically, it defines a class of "sufficient approximations" to the affirmation and disaffirmation relations, and then shows that an algorithm capable of determining affirmability/disaffirmability relative to one of these approximations will still provide a sufficient basis for a query algorithm. It therefore identifies a potentially larger class of algorithms that are capable of solving the queries.

Corollary 2.6.2 *For a regular system s, suppose that P and N are relations between bindings b and formulas (and conclusions) f, such that the following conditions hold:*

- $P(b, f) \Rightarrow b \models^{+}_{s} f$
- $N(b, f) \Rightarrow b \models^{-}_{s} f$
- $(b \in F_s \text{ and } b \models^{+}_{s} f) \Rightarrow P(b, f)$
- $(b \in F_s \text{ and } b \models^{-}_{s} f) \Rightarrow N(b, f)$

Then for conclusions c, formulas q and variables v:

- *c is q-reachable* $\Leftrightarrow (\exists\, b \in B : P(b, q) \text{ and } P(b, c))$
- *c is q-inevitable* $\Leftrightarrow (\nexists\, b \in B : P(b, q) \text{ and } N(b, c))$

istence of *final* bindings that satisfy certain conditions, this theorem reformulates the problem in terms of the existence of *any* bindings that meet the analogous conditions. Thus, we are relieved of the complex requirement that the bindings be final. The complexity of the system, and of the query problem, has been concentrated into the structure of the affirmation and disaffirmation relations.

- $P(b, q)$ *and* $P(b, c) \Rightarrow$

  each binding in $F_s(b)$ *witnesses the q-reachability of* c

That completes our analysis of queries. In the next chapter, we will use Corollary 2.6.2 to prove the correctness of our query algorithm for acyclic production systems.

Chapter 3

The Query Algorithm

We now present a reachability/inevitability query algorithm for acyclic production systems.

3.1 The Query Algorithm

The query algorithm consists of (1) the mutually recursive functions $pos_s(e)$ and $neg_s(e)$, shown in Figure 3.1, which compute, respectively, the positive and negative reachability formulas for an expression e within a system s, and (2) the top-level routine, shown in Figure 3.2, which uses pos_s and neg_s to construct particular reachability formulas whose satisfiability indicates the answers to the queries. Also, the algorithm for $core_s(b)$, given in Figure 2.1 of the previous chapter, is used in the computation of the witness bindings.

The arguments to the top-level routine are the system s, the query formula q, and a domain-specific constraint-solver x. Three methods of x are used: (1) $x.satisfiable(f)$, which tests the satisfiability formula f, (2) $x.solution(f)$, which,

for satisfiable f, produces a satisfying binding, and (3) $x.project(vars, f)$, which projects the constraint f onto the variables *vars*. Each conclusion and question has the following three attributes that are set by the algorithm: a boolean indicating reachability, a boolean indicating inevitability, and an attribute that holds a witness binding. For questions, the algorithm also sets an attribute that holds the answer choices under the query.

The heart of the algorithm consists of the functions *pos* and *neg*.

Definition 3.1.1 (The Reachability Formulas $\mathbf{pos}_s(\mathbf{e})$, $\mathbf{neg}_s(\mathbf{e})$)
For an expression e and acyclic production system s, we name the formulas constructed in Figure 3.1 as follows. The formula $pos_s(e)$ is the positive reachability formula *for e, and $neg_s(e)$ is the* negative reachability formula *for e.*

The maps *posmap* and *negmap*, which send variables to formulas, are used in the functions *pos* and *neg* as an optimization that leads to a more compact representation for the reachability formulas that are returned by *pos* and *neg*. These maps ensure that for each variable x, the formulas $pos(x)$ and $neg(x)$ will get computed at most once, and, when they are computed, pointers to them will be kept in the map entries $posmap(x)$ or $negmap(x)$.[1]

Remark on the Use of Negations. The cases in the definitions that handle the formulas $\neg\phi$, are, at present, parts of the code that will not be executed, since we have restricted the precondition and query formulas to be *monotone*. When we

[1]Later, in the section on complexity, we will prove that the resulting "dagification" of all the formulas produced by the repeated calls to *pos* and *neg*, causes the total number of formula-nodes to be linear in the size of the input production system plus the query formula.

```
global posmap, negmap : Vars → Formulas    // for memoization
```

```
function pos_s(e) : AcyclicSystem s × Expression e → Formula
begin
  if e ∈ Vars and posmap(e) ≠ ⊥ then return posmap(e)
  result := case e of
```

$true$	$\rightarrow$	$true$	
$false$	$\rightarrow$	$false$	
$\phi_1 \wedge \phi_2$	$\rightarrow$	$pos_s(\phi_1) \wedge pos_s(\phi_2)$	
$\phi_1 \vee \phi_2$	$\rightarrow$	$pos_s(\phi_1) \vee pos_s(\phi_2)$	
$\neg\phi$	$\rightarrow$	$neg_s(\phi)$	
$p(t_1, \ldots, t_n)$	$\rightarrow$	$pos_s(t_1) \wedge \cdots \wedge pos_s(t_n) \wedge p(t_1, \ldots, t_n)$	(predicate)
$f(t_1, \ldots, t_n)$	$\rightarrow$	$pos_s(t_1) \wedge \cdots \wedge pos_s(t_n)$	(function)
c	$\rightarrow$	$true$	(constant)
v	$\rightarrow$	$pos_s(precond_{v,s}) \wedge indom_{v,s}$	(question var)
w	$\rightarrow$	$pos_s(precond_{w,s}) \wedge pos_s(term_{w,s}) \wedge w = term_{w,s}$	(derived var)

```
  if e ∈ Vars then posmap(e) := result
  return result
end
```

```
function neg_s(e) : AcyclicSystem s × Expression e → Formula
begin
  if e ∈ Vars and negmap(e) ≠ ⊥ then return negmap(e)
  result := case e of
```

$true$	$\rightarrow$	$false$	
$false$	$\rightarrow$	$true$	
$\phi_1 \wedge \phi_2$	$\rightarrow$	$neg_s(\phi_1) \vee neg_s(\phi_2)$	
$\phi_1 \vee \phi_2$	$\rightarrow$	$neg_s(\phi_1) \wedge neg_s(\phi_2)$	
$\neg\phi$	$\rightarrow$	$pos_s(\phi)$	
$p(t_1, \ldots, t_n)$	$\rightarrow$	$neg_s(t_1) \vee \cdots \vee neg_s(t_n) \vee pos_s(\overline{p}(t_1, \ldots, t_n))$	

```
querySystem(AcyclicSystem s, Formula q, Solver x)
begin
  for each conclusion c ∈ C_s do
    c.reachable := x.satisfiable(pos_s(q) ∧ pos_s(precond_{c,s}))
    c.inevitable := not(x.satisfiable(pos_s(q) ∧ neg_s(precond_{c,s})))
    if c.reachable then
      c.witness := core_s(x.solution(pos_s(q) ∧ pos_s(precond_{c,s})))
  endfor
  for each question v ∈ V_s do
    v.reachable := x.satisfiable(pos_s(q) ∧ pos_s(v))
    v.inevitable := not(x.satisfiable(pos_s(q) ∧ neg_s(v)))
    if v.reachable then
      v.witness := core_s(x.solution(pos_s(q) ∧ pos_s(v)))
    v.choices := x.project({v}, pos_s(q) ∧ pos_s(v))
  endfor
end
```

Figure 3.2: Top-Level of the Query Algorithm — Subroutines in Figure 3.1

later extend the operational semantics to include negations, then the algorithm as we have now written it will also work correctly in the extended framework.

Irrespective of whether a formula f contain negations, the reachability formulas $pos_s(f)$ and $neg_s(f)$ will always be monotone. The reason is simply that no negation operators $\neg$ are introduced in the right-hand-side expressions of the functions *pos*

and neg.[2]

Example 3.1.2 (Reachability Formulas for S_{xyz})

We compute the positive and negative reachability formulas for the expression e ranging over $\{x, y, z\}$, for the system S_{xyz} from Example 1.6.2. Here are the relevant parts of S_{xyz}:

$var\ v$	dom_v	$precond_v$	$term_v$
x	$\{-2,-1,0,1,2\}$	$true$	
y	$\{-2,-1,0,1,2\}$	$x > 0$	
z		$y < 0$	$x * y$

Here are the reachability formulas:

$$
\begin{array}{lcll}
pos(x) & = & indom_x & = & x \in \{-2, \ldots, 2\} \\
pos(y) & = & pos(x) \wedge (x > 0) \wedge indom_y & = & x \in \{1,2\} \wedge y \in \{-2, \ldots, 2\} \\
pos(z) & = & pos(y) \wedge (y < 0) \wedge z = xy & = & x \in \{1,2\} \wedge y \in \{-2,-1\} \wedge z = xy \\
neg(x) & = & false & = & false \\
neg(y) & = & neg(x) \vee (pos(x) \wedge x \leq 0) & = & x \in \{-2,-1,0\} \\
neg(z) & = & neg(y) \vee (pos(y) \wedge y \geq 0) & = & x \in \{-2,-1,0\} \vee \\
 & & & & (x \in \{1,2\} \wedge y \in \{0,1,2\})
\end{array}
$$

The reachability formulas for the variables provide the data for answering all the possible queries to the system. We illustrate with three queries to the system

[2]This was also true in the definition of the monotone equivalent formula f^+ and the monotone opposite formula f^- (c.f. Definition 1.4.4). In fact, the cases in the definitions of $pos_s(f)$ and $neg_s(f)$ for the boolean connectives $\wedge$, $\vee$ and $\neg$ correspond exactly to the cases in the definitions of f^+ and f^-.

S_{xyz}. First, we answer them with informal arguments. Then we show that the *pos* and *neg* formulas give the correct results.

Query 1: Is $z = -2$ reachable if $x = 1$?

Query 2: Is $z = -2$ reachable if $x = -y$?

Query 3: Is $x = 2$ inevitable if $z = -4$?

Answer 1: Yes, because the final binding $\{x = 1,\ y = -2,\ z = -2\}$ satisfies both the query and the conclusion.

Answer 2: No, because z is a product of two integers x, y, and -2 cannot be expressed as a product of two integers with the same absolute value.

Answer 3: Yes, because in order for $z = x * y$ to be defined, x must be either 1 or 2, and y must be either -1 or -2, and so the condition that $z = -4$ implies that $x = 2$.

Example 3.1.3 (Formal Analysis of Query 1)

Here, q = 'x = 1', and $precond_c$ = 'z = −2'. To determine the q-reachability of c, the algorithm constructs the formula:

$(pos(q) \wedge pos(precond_c)) =$

$((pos(x) \wedge x = 1) \wedge (pos(z) \wedge z = -2)) =$

$(x = 1 \wedge (x \in \{1, 2\} \wedge y \in \{-2, -1\} \wedge z = xy \wedge z = -2)) =$

$(x = 1 \wedge z = y = -2)$.

This is satisfiable, by the one binding $b = \{x = 1,\ y = -2,\ z = -2\}$. The algorithm then computes $core(b) = \{x = 1,\ y = -2,\ z = -2\} = b$, and so reports this binding

as a witness to the q-reachability of c. This is correct, because $b \in A_s$, $b \models q$, *and* $c \in say(b)$.

Example 3.1.4 (Formal Analysis of Query 2)

Here, $q =$ *'*$x = -y$*', and* $precond_c =$ *'*$z = -2$*'. To determine the q-reachability of c, the algorithm constructs the formula:*

$(pos(q) \wedge pos(precond_c)) =$

$((pos(x) \wedge pos(y) \wedge x = -y) \wedge (pos(z) \wedge z = -2)) =$

$(pos(z) \wedge x = -y \wedge z = -2) =$

$((x \in \{1, 2\} \wedge y \in \{-2, -1\} \wedge z = x * y) \wedge x = -y \wedge z = -2) =$

$(x \in \{1, 2\} \wedge (z = -2 = -x^2) \wedge \cdots).$

This formula is unsatisfiable, and so the algorithm answers No to the reachability query.

Example 3.1.5 (Formal Analysis of Query 3)

Here, $q =$ *'*$z = -4$*', and* $precond_c =$ *'*$x = 2$*'. To determine the q-inevitability of c, the algorithm constructs the formula:*

$(pos(q) \wedge neg(precond_c)) =$

$(pos(z) \wedge z = -4 \wedge (neg(x) \vee (pos(x) \wedge x \neq 2))) =$

$(pos(z) \wedge z = -4 \wedge pos(x) \wedge x \neq 2) =$

$(x \in \{1, 2\} \wedge y \in \{-2, -1\} \wedge z = x * y \wedge z = -4 \wedge x \in \{-2, -1, 0, 1\}) =$

$(x = 1 \wedge y \in \{-2, -1\} \wedge z = x * y = -4).$

This formula is unsatisfiable, and so the algorithm answers Yes to the inevitability query.

3.2 The Reachability Formulas

Here we study the meanings of the *pos* and *neg* formulas constructed by the query algorithm. We give theorems describing relationships between systems, expressions, and the satisfying bindings for $pos_s(e)$ and $neg_s(e)$. This material will be used in our correctness proof for the algorithm.

The next two theorems give non-procedural descriptions for the attainable and final bindings.

Theorem 3.2.1 *Recall from the terminology on page 11 that V_s is the set of question variables for system s; W_s is the set of calculation variables; A_s is the set of attainable bindings. Then, for an acyclic production system s,*

$$b \in A_s \Leftrightarrow \quad (\forall\, x \in V_s \cup W_s \,:\, b(x) \neq \perp \Rightarrow b \models (indom_x \wedge precond_x))$$

Theorem 3.2.2 *Recall from the terminology on page 11 that F_s is the set of final bindings for system s. Then, for an acyclic production system s,*

$$\begin{aligned} b \in F_s \Leftrightarrow \quad & ((\forall\, v \in V_s \,:\, \ \mathsf{if}\ b \models precond_v\ \mathsf{then}\ b \models indom_v\ \mathsf{else}\ b(v) = \perp)\ \ \& \\ & (\forall\, w \in W_s \,:\, \ \mathsf{if}\ b \models pre_w\ \mathsf{then}\ b(w) = val(term_w, b)\ \mathsf{else}\ b(w) = \perp)) \end{aligned}$$

We will need the following notation $A_s(b)$ for the attainable extensions of the core of a binding.

Definition 3.2.1 (Attainable Extensions $A_s(b)$)
For a regular system s and binding b,

$$A_s(b) \equiv \{b' \in A_s \mid b' \supseteq core_s(b)\}$$

The next theorem gives essential properties of the satisfying bindings for the reachability formulas.

Theorem 3.2.3 *For an acyclic production system s, binding $b \in B_s$, formula f, and term t:*

- $b \models pos_s(f) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \models f)$
- $b \models neg_s(f) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \not\models f)$
- $b \models pos_s(t) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \models (t = t)$
- $b \models neg_s(t) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \not\models (t = t)$

In words, for a formula f and term t, (1) if a binding b satisfies $pos(f)$, then f is true in every attainable extension of the core of b, (2) if b satisfies $neg(f)$, then f is not satisfied in any of these extensions, (3) if b satisfies $pos(t)$, then t is defined in all the extensions, and (4) if b satisfies $neg(t)$, then t is undefined in all of the extensions. Since the final extensions $F_s(b)$ are a subset of the attainable extensions $A_s(b)$, by a weakening of the this theorem, we obtain the following two corollaries.

Corollary 3.2.4 *For an acyclic production system s, binding b, and formula f:*

- $b \models pos_s(f) \Rightarrow b \models^{+}_{s} f$
- $b \models neg_s(f) \Rightarrow b \models^{-}_{s} f$

Corollary 3.2.5 *For an acyclic production system s, binding b, and term t:*

- $b \models pos_s(t) \Rightarrow t$ *is applicable in* b

- $b \models neg_s(t) \Rightarrow t$ *is inapplicable in* b

In words, (1) the satisfying bindings for $pos(f)$ cause f to be affirmed, (2) the satisfying bindings for $neg(f)$ cause f to be disaffirmed, (3) the satisfying bindings for $pos(t)$ cause t to be applicable, and (4) the satisfying bindings for $neg(t)$ cause t to be inapplicable.

The next theorem shows that for final bindings, the meaning of $pos(f)$ coincides with the meaning of f, and the meaning of $neg(f)$ coincides with the meaning of a negation of f.

Theorem 3.2.6 *For an acyclic production system* s*, and final binding* $b \in F_s$*,*

$b \models pos_s(f) \Leftrightarrow b \models f$

$b \models neg_s(f) \Leftrightarrow b \not\models f$

Corollary 3.2.7 *For an acyclic production system* s*, and final binding* $b \in F_s$*,*

$b \models^{+}_{s} f \Rightarrow b \models pos_s(f)$

$b \models^{-}_{s} f \Rightarrow b \models neg_s(f)$

Proof

Suppose $b \in F_s$. Suppose that $b \models^{+}_{s} f$. Then, since b is final, $F_s(b) = \{b\}$. Hence $b \models f$. Hence, by Theorem 3.2.6, $b \models pos_s(f)$. Similarly, $b \models^{-}_{s} f \Rightarrow b \models neg_s(f)$. ■

The last theorem shows how we can use the *pos* formulas to get the menu of choices $choices(v, q)$, as defined on page 17, for the variable v under the query q.

Theorem 3.2.8 *For acyclic production system* s*, variable* $v \in V_s$*, and formula* q*:*

$choices_s(v, q) = \{b(v) \mid b \in B \ \ \& \ \ b \models (pos_s(q) \wedge pos_s(v))\}$

That concludes our analysis of the semantics of the *pos* and *neg* formulas. The usefulness of these semantic results follows from the fact that the *pos* and *neg* formulas are defined from constructive procedures.

3.3 Proof of Correctness

We now prove the correctness of the query algorithm. The proof is an application of Corollary 2.6.2. Recall that this theorem gives conditions for relations $P(b, f)$ and $N(b, f)$ to approximate affirmation $b \models^{\pm}_{s} f$ disaffirmation $b \models_{s} f$, so that the reachability/inevitability queries are reducible to questions of affirmability and disaffirmability relative to P and N.

In the proof, we will use the *pos* and *neg* formulas to define a pair of such relations P_s and N_s for each acyclic production system s. Using the results of the previous subsection, we will show that P_s and N_s conform to the conditions in the premises of Corollary 2.6.2. The correctness of the query algorithm then follows as an immediate consequence.

Theorem 3.3.1 *The query algorithm shown on page 87 is correct.*

Proof

Let s be an acyclic production system. For bindings b and formulas f, define $P_s(b, f)$ to be the relation $b \models pos_s(f)$, and $N_s(b, f)$ be the relation $b \models neg_s(f)$.

Claim now that P_s and N_s satisfy the four premises of Corollary 2.6.2. The first two premises, viz., that $P_s(b, f) \Rightarrow b \models^{\pm}_{s} f$ and that $N_s(b, f) \Rightarrow b \models_{s} f$, are stated in Corollary 3.2.4. The last two premises, viz., that for final bindings b,

$b \models^{+}_{s} f \Rightarrow P_s(b, f)$ and $b \models^{-}_{s} f \Rightarrow N_s(b, f)$, are stated in Corollary 3.2.7. Hence, the consequent Corollary 2.6.2 on page 83 holds.

We now use this consequent to justify the following actions of the query algorithm.

1. The theorem says that c is q-reachable iff $\exists b \in B : P(b, q)$ and $P(b, c)$, *i.e.*, iff $\exists b \in B \; s.t. \; b \models pos(f)$ and $b \models pos(precond_c)$. This result justifies the first step in the algorithm:

$$c.reachable := satisfiable(pos(q) \wedge pos(precond_c))$$

2. The theorem says that c is not q-inevitable iff $\exists b \in B : P(b, q)$ and $N(b, c)$, *i.e.*, iff $\exists b \in B \; s.t. \; b \models pos(f)$ and $b \models neg(precond_c)$. This result justifies the second step in the algorithm:

$$c.inevitable := not(satisfiable(pos(q) \wedge neg(precond_c)))$$

3. The theorem says that if $\exists b \in B : P(b, q)$ and $P(b, c)$, *i.e.*, iff $\exists b \in B \; s.t. \; b \models pos(f)$ and $b \models pos(precond_c)$, then every final extension of $core(b)$ witnesses the q-reachability of c. This result justifies the third step in the algorithm:

$$if \; c.reachable \; then \; c.witness := core(solution(pos(q) \wedge pos(precond_c)))$$

The next three statements, which set the attributes $v.reachable$, $v.inevitable$, and $v.witness$, for a question variable v, have a similar justification. In order to make the same argument work for these three statements, we need only to observe that: (1) a question v is q-reachable/inevitable iff the conclusion $v = v$ is q-reachable/inevitable, and (2) $pos(v = v)$ is the same formula as $pos(v)$, and $neg(v = v)$ is the same formulas as $neg(v)$.

Finally, Theorem 3.2.8 on page 93 states that the menu of choices for v under the query q, aka $choices_s(v, q)$, equals $\{b(v) \mid b \in B \ \ \& \ \ b \models (pos_s(q) \wedge pos_s(v))\}$. This result justifies the last step in the algorithm:

$$v.choices := project(v,\ pos(q) \wedge pos(v)$$

■

3.4 Answers to Selected Queries

Here are the answers to the queries that we presented at the end of Section 1.6.

1. **Query:** In the medical decision DAG on page 53, is the diagnosis *GlutenIntol* reachable if '$Iron = 100 \vee FolicAcid = 100$'?

   Answer: No.

   Reason: $pos(Iron = 100 \vee FolicAcid = 100) \wedge pos(GlutenIntol)$ is unsatisfiable, since the constraints $Iron < 100$ and $FolicAcid < 100$ are conjuncts of $pos(GlutenIntol)$. Here is an informal justification for this answer. First, $Iron = 100 \Rightarrow Bleeding$ is inapplicable $\Rightarrow$ *Malabsorption* is inapplicable $\Rightarrow$ *GlutenIntol* is unreachable. Second, $FolicAcid = 100 \Rightarrow$ *Malabsorption* is inapplicable $\Rightarrow$ *GlutenIntol* is unreachable.

2. **Query:** In the trial strategy DAG on page 54, is conclusion *Victory* reachable if the witness says No to the question *Gun*?

   Answer: Yes.

   Reason: $pos(Gun = no) \wedge pos(precond_{victory})$ is satisfiable, by the binding $\{CrimeScene = yes,\ Gun = no,\ RecallJoe = yes,\ JoeRemembers = yes\}$.

3. **Query:**

   In the trial strategy DAG, is the conclusion $precond_{victory} \wedge precond_{impeach}$ reachable?

   Answer: No.

   Reason: $pos(precond_{victory} \wedge precond_{impeach})$ is unsatisfiable.

4. **Query:** In the trial strategy DAG, is the conclusion '$CrimeScene = no$' inevitable under the query $precond_{impeach}$?

   Answer: No.

   Reason: $pos(precond_{impeach}) \wedge neg(CrimeScene = no)$ is not unsatisfiable. For instance, it is satisfied by $\{CrimeScene = yes,\ Gun = no,\ RecallJoe = yes,\ JoePicture = no\}$. This is an attainable binding that satisfies the precondition of *Impeach* but not the conclusion '$CrimeScene = no$'.

3.5 Complexity

Here we analyze the complexity of the query algorithm. The first question is how to measure the complexity of queries.

Our algorithm performs a reduction from production-system queries into conjunctive satisfiability problems. We will see that the time complexity of this reduction is linear in the number of constraints generated. We therefore measure the *constraint-complexity* of the algorithm, defined as the aggregate size of the formulas that the algorithm transfers to a conjunctive constraint solver. When we combine the constraint-complexity with the complexity function for a specific constraint solver, we will then obtain a specific time-complexity.

The constraint-complexity of an individual query is a function of both the query formula and the production system. As we saw in Section 1.1.5, production-system queries are NP-hard with respect to the query formula. Hence we will follow the standard practice of analyzing the *data complexity* of queries, *i.e.*, the complexity for arbitrary fixed query formulas, measured as a function of the size of the input data object. By combining the notions of data complexity and constraint complexity, we arrive at the following measure for our complexity analysis.

Definition 3.5.1 (Data Constraint-Complexity)
The data constraint-complexity *of an algorithm is the aggregate size of the formulas that it transfers to a conjunctive constraint solver, expressed as a function of the size of the input production system.*

We begin by analyzing the complexity of queries to general monotone production systems, and then we specialize to the important and tractable case where the preconditions are conjunctive.

3.5.1 Complexity of Queries to Monotone Systems

The following proposition, which shows the difficulty of queries to acyclic production systems with monotone formulas, is proven by a direct reduction from SAT.

Proposition 3.5.1 *For monotone acyclic production systems, the determination of the q-reachability of a conclusion is NP-hard in the size of the system.*

We now describe a graph that we will use to analyze the complexity of classes of queries.

The Formula Graph for a Query

Each call to the top-level of the query algorithm leads to the construction of a collection of *pos* and *neg* formulas that are to be tested for satisfiability. It would be inefficient, however, to construct these formulas in isolation from each other, because they will generally have many subformulas in common. These replications arise from the recursive structure of the definitions of $pos(e)$ and $neg(e)$; whenever the variable x appears in the expressions for y, then the reachability formulas for x will occur as subformulas of the reachability formulas for y. For example, in the reachability formulas for the system S_{xyz}, which we computed in Example 3.1.2 on page 88, we see that $pos(x)$ occurs in $pos(y)$, and $pos(y)$ occurs in $pos(z)$, and similarly for the *neg* formulas.

Because of the memoing statements placed at the beginning of the functions *pos* and *neg* on page 86, the algorithm does not construct multiple copies of the shared subformulas. These statements ensure that for each expression e, the formulas $pos(e)$ and $neg(e)$ are constructed at most once. In particular, for each variable x, the formulas $pos(x)$ and $neg(x)$ are each constructed once. For an expression e with free variables V, the formulas $pos(e)$ and $neg(e)$ are constructed "on top" of the formulas $\{pos(x), neg(x) \mid x \in V\}$. Then the incremental cost of constructing $pos(e)$ and $neg(e)$ is $\Theta(size(e))$. Each top-level call to the query algorithm thus results in the construction of a unified *system* of overlapping positive and negative reachability formulas. We represent this system by the following DAG.

Definition 3.5.2 (The Formula Graph fg(s,q))

For each acyclic production system s and formula q, we use the execution of the

top-level call to the query algorithm in Figure 3.2 to define a DAG with three kinds of nodes: AND and OR nodes, which are analogous to logic gates, and constraint nodes. The constraint nodes are the parentless roots of the graph, and the other nodes each have two parents. The AND nodes are created by the execution of the $\wedge$ *operators in the right-hand-side expressions of the functions pos or neg, and the OR nodes are created by the execution of the* $\vee$ *operators. The constraint nodes are created by the evaluation of constraints, such as* $indom_v$ *and* $precond_v$*, in the right-hand-side expressions. As we explained in the previous paragraph, the memoing statements in the algorithm ensure that for any subexpression e, there will be just one node in* $fg(s, e)$ *for* $pos_s(e)$*, and just one node for* $neg_s(e)$*.*

Example 3.5.3 *The figures on page 101 show a production system with variables* $\{x, y\}$*, the positive and negative reachability formulas for the variables, and the formula graph for this system of formulas.*

The next proposition states that the size of the formula graph is linear in the size of the production system plus the query.

Proposition 3.5.2

$$size(f(s, q)) = O(s + q)$$

Corollary 3.5.3 *Since, for each system s and query formula q, the cost of constructing* $fg(s, q)$ *is* $O(s + q)$*,* i.e.*, is linear, then the theoretical complexity of the algorithm will be determined by the complexity of evaluating this formula graph. We will analyze this complexity in the remainder of this section.*

A Simple Production System

var v	$precond_v$	$indom_v$	$pos(v)$	$neg(v)$
x	*true*	$x = 1 \vee x = 2$	$indom_x$	*false*
y	$x = 1$	$y = 1 \vee y = 2$	$pos(x) \wedge (x = 1) \wedge indom_y$	$neg(x)$
				$(pos(x) \wedge (x \neq 1))$

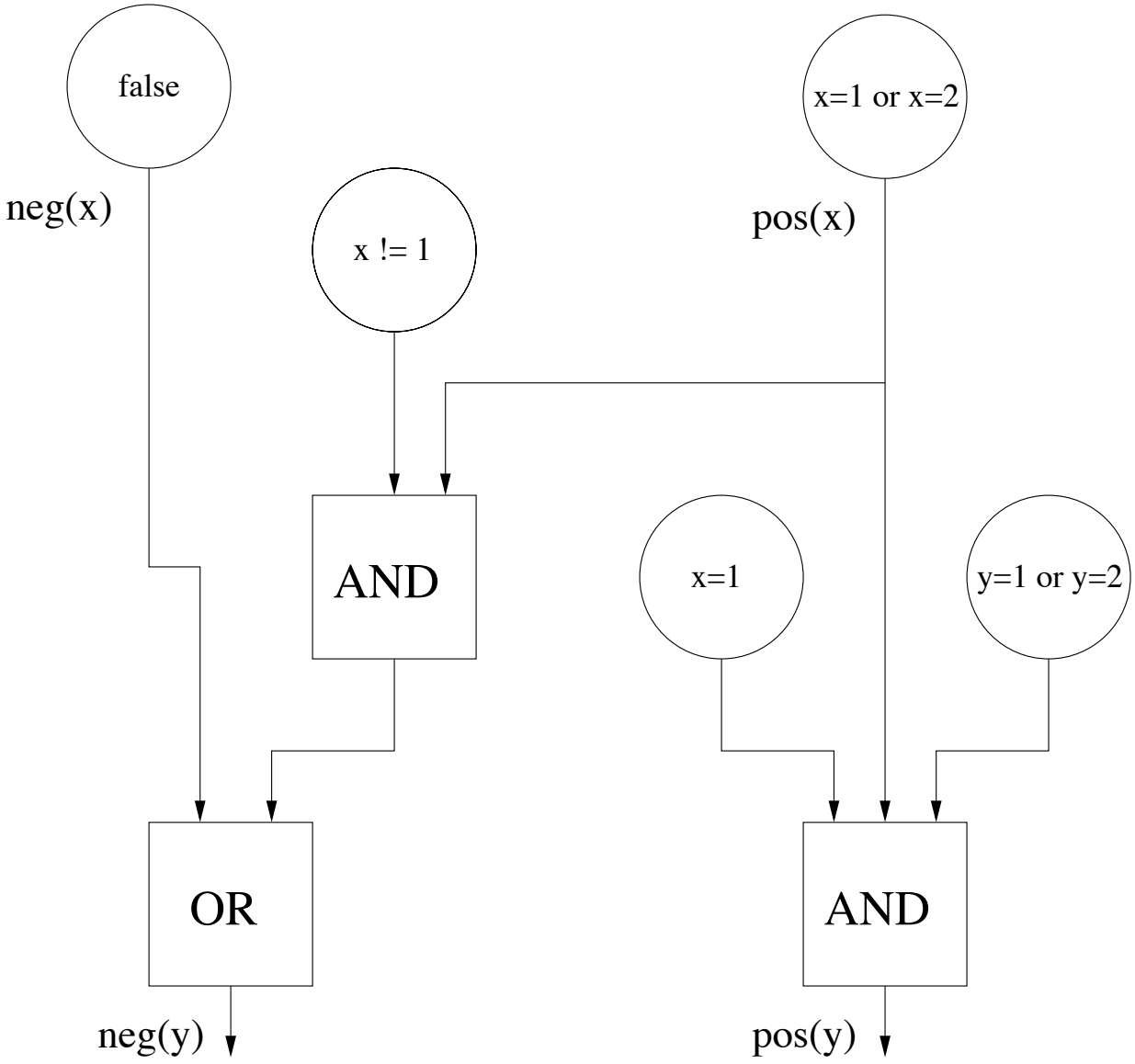


Figure 3.3: The Formula Graph

Every node of the formula graph represents the formula that is obtained by the recursive traversal which "forgets" the DAG structure: formula(AND(x,y)) = formula(x) $\wedge$ formula(y), formula(OR(x,y)) = formula(x) $\vee$ formula(y), and formula(root(c)) = c. We may therefore apply any formula-related notion to the nodes of the formula graph. For instance, we may speak of the satisfiability of a node, and of its satisfying bindings.

DNF-Based Analysis of the Formula Graph

We now use the disjunctive normal expansions of the nodes in the formula graph in order to bound the complexities of individual queries. We will represent the DNF of each node f in the graph by an associated set of sets of constraints $dnf(f)$, which is taken to mean the disjunction of the conjunctions of the constraints.

Definition 3.5.4 (dnf(f)) *For monotone formulas, define:*

$$\begin{aligned} dnf(constraint) &= \{\{constraint\}\} \\ dnf(f_1 \wedge f_2) &= \{t_1 \cup t_2 \,:\, t_1 \in dnf(f_1),\, t_2 \in dnf(f_2)\} \\ dnf(f_1 \vee f_2) &= dnf(f_1) \cup dnf(f_2) \\ dnf(false) &= \{\} \\ dnf(true) &= \{\{\}\} \end{aligned}$$

Example 3.5.5

$$dnf(x = 1 \wedge (y = 2 \vee z = 3))) = \{\{x = 1, y = 2\}, \{x = 1, z = 3\}\}$$

In the framework of [21], a conjunction of constraints is viewed as a "generalized tuple." In this spirit, we define the *tuples* of f to be the members of $dnf(f)$, where

each such tuple is taken to represent the conjunction of its members. The following easy proposition says that a formula is equivalent to the disjunction of its tuples.

Proposition 3.5.4 *For a monotone formula f:*

$$f \cong \bigvee_{t \in dnf(f)} \bigwedge_{c \in t} c$$

The DNF is useful because (1) a conjunctive satisfiability tester can be applied to each of the tuples in $dnf(f)$ in order to determine the satisfiability of f, and (2) conjunctive satisfiability testers exist for many useful constraint domains.[3] The size of $dnf(f)$ therefore puts a bound on the constraint-complexity of testing the satisfiability of f. The exponential blowup in the size of $dnf(f)$ which occurs when unrestricted combinations of disjunctions and conjunctions are allowed, e.g., CNF, therefore expresses the NP-hardness of the satisfiability testing.

We now introduce a measure of the "disjunctive complexity" of a formula f, which will serve to put an upper bound on the number of tuples in the DNF expansion of f.

Definition 3.5.6 ($dc(f)$, the disjunctive complexity of f)

$dc(constraint) = 1$

$dc(f_1 \vee f_2) = dc(f_1) + dc(f_2)$

$dc(f_1 \wedge f_2) = dc(f_1) \times dc(f_2)$

Proposition 3.5.5 *For a formula f:*

$$|dnf(f)| \leq dc(f)$$

[3]See our survey section on Constraint Domains.

We now define a measure of the "weight" of a node x in the formula graph, which is the size of the constraints in the "dagified" formula for x.

Definition 3.5.7 ($weight(x)$)
For a node x in a formula graph g, define:

$$weight(x) = size(roots(g) \cap ancestors_g(x))$$

The next proposition puts a bound on the sizes of the tuples.

Proposition 3.5.6 *For a node x in the formula graph $f(s, q)$ and a tuple $t \in dnf(x)$:*

$$|t| \leq weight(x) \leq size(s) + size(q)$$

3.5.2 Conjunctive Systems

We now analyze the complexity of queries to an important case of acyclic production systems, viz., the *conjunctive systems* in which the preconditions are conjunctions of primitive constraints. For example, our medical decision DAG on page 53 is conjunctive.

Significance of Conjunctive Systems

Although not capable of expressing all the boolean relations, the conjunctions have a special objective status, which gives a basis for a diverse range of applications

of conjunctive production systems. One indication of this special status is the fact that conjunction appears to be the default logical connective in natural language. For example, when we speak of the cold, rainy night, this is understood to mean cold *and* rainy, and when we say that the plane travels from Chicago to Dallas, we mean that it starts in Chicago *and* it lands in Dallas. The root of the special character of conjunctions is indicated by the derivation of the word itself: whereas "dis-junction" means joined apart, "con-junction" means joined together.

Whenever we join things *together*, the constraint that describes the resultant whole will be the conjunction of the constraints that describe the parts. For example, in a circuit built from logic gates, each of the gates is described by a constraint that relates the values at the input and output terminals, and the input-output behavior of the whole circuit is described by the conjunction of these local constraints. Or, put into relational terms, the global relation between the values at the connection points is the *join* of the relations for each of the gates.[4]

The most basic description of an object is a conjunction of its attributions. That is the reason why the basic unit of relational databases, viz., the record $\{attr_1 = val_1, \ldots, attr_k = val_k\}$, is implicitly taken to mean the conjunction of these attributions.

[4]Disjunctions arise from sources that are qualitatively different than the sources of conjunctions. For instance, they can arise from abstractions, e.g., a thing is either an animal, a vegetable or a mineral, and from states of uncertainty, e.g., my missing keys are either at home or at work.

Complexity of Queries to Conjunctive Systems

The tractability of queries to conjunctive systems arises from the forms assumed by the *pos* and *neg* formulas in these systems. The following proposition describes the structure of the *pos* formulas for the variables in a conjunctive system.[5]

Proposition 3.5.7 *For a conjunctive system s and a variable x:*

$$pos_s(x) = \bigwedge_{y \in ancestors(x)} precond_y \wedge indom_y$$

Hence, as stated in the next corollary, in a conjunctive system, there is just one tuple in the *pos* formula for any disjunction-free expression.

Corollary 3.5.8 *For a conjunctive system s and disjunction-free expression e,*

$$|dnf(pos_s(e))| \leq 1$$

The following proposition and corollary show that, in a conjunctive system, the structure of the query formula q places a bound—which is independent of the system s—on the number of tuples in $pos_s(q)$.

Proposition 3.5.9 *For a conjunctive system s and a monotone formula q:*

$$dc(pos_s(q)) = dc(q)$$

[5]We can prove an analogous but more complicated proposition for the *neg* formulas, but this will not be needed in the proof of the complexity result.

Corollary 3.5.10 *For a conjunctive system s and a monotone formula q:*

$$|dnf(pos_s(q))| \leq dc(q)$$

The next theorem gives the complexity of queries to conjunctive systems.

Theorem 3.5.11 *For a conjunctive system s and formula q, the data complexity in units of constraints of querying the q-reachability and q-inevitability of all the conclusions in s is $O(n \times w)$, where $n = size(s)$ and w is the maximum weight of a node in the formula graph. Since w is $O(n)$, this complexity is $O(n^2)$.*

3.6 Implementation

3.6.1 Methods for the Constraints

We now give constraint-processing methods for the low-cost but useful domain of univariate monotone order (UMO) constraints.[6] Recall that the primitive constraints of this domain are comparison-based monotone formulas with one free variable, such as '$Month = jan \vee Month > june$', and that these constraints suffice for our medical and courtroom decision DAGs. In combination with our query algorithm on page 87, and the DNF-based formula graph described in the previous subsection, these constraint-processing methods provide the basis for a complete query algorithm for acyclic production systems over the UMO constraint domain.

The methods needed for the DNF-based evaluation are: (1) a routine for conjoining two conjunctions of primitive constraints, (2) a satisfiability test for con-

[6] See page 26 in the groundwork section.

junctions of primitives, and (3) a routine to project a conjunction of primitives onto a set of variables. Disjunctions, on the other hand, are not handled by the methods for the constraint domain, but instead are handled by the constraint-independent method of building sets of tuples.

We represent each UMO primitive by a variable and a sorted list of disjoint intervals. For example, the constraint '$Month = Jan \vee Month > June$' is represented by the variable *Month* and the interval-list $[[Jan, Jan], (June, Dec]]$. We then represent a conjunction of primitives by a mapping that sends each of the variables to its associated list of intervals. The meaning of the conjunction is the Cartesian product of the respective sets of intervals.

We use the function shown in Figure 3.4 to conjoin two UMO primitives, *i.e.*, to "merge" two sorted interval-lists into an interval-list that represents their conjunction. The conjunction of two UMO tuples is then accomplished by componentwise application of the conjunction function for primitives. The satisfiability-test for a tuple is just the check for whether any of the interval-lists are empty. Finally, the projection of a tuple onto a set of variables is accomplished by restricting the map for the tuple to those variables.

Example 3.6.1 *Suppose that* $x_1 = [[1,3],[4,6]]$*, and* $x_2 = [[2,5]]$*. Then the loop begins by initializing* $i_1 = [1,3]$ *and* $i_2 = [2,5]$*. Out of the six cases, the third is true:* $i_2.lower \in i_1$, i.e., $2 \in [1,3]$*. Hence, result gets set to* $[i_1 \cap i_2]$, i.e., *result* $= [[2,3]]$*. Also,* x_1 *gets popped, so that now* $x_1 = [[4,6]]$*.*

On the second iteration, $i_1 = [4,6]$*, and* $i_2 = [2,5]$*. Then the sixth case is true:* $i_1.lower \in i_2$*. So, result gets set to* $[[2,3],[4,5]]$*. Also,* x_2 *gets popped, so that now* $x_2 = []$*. Since* x_2 *is empty, result gets returned.*

```
conjoin(IntervalList x1, IntervalList x2)
   result = []
   loop
      if x1 = [] OR x2 = [] then return result
      Interval i1 = first(x1)
      Interval i2 = first(x2)
      if (i1 disjoint from and left of i2)
         x1 = rest(x1)
      else if (i1 ⊆ i2)
         result = result + [i1]
         x1 = rest(x1)
      else if (i2.lower ∈ i1)
         result = result + [i1 ∩ i2]
         x1 = rest(x1) // the three symmetric cases:
      else if (i2 disjoint from and left of i1)
         x2 = rest(x2) ...
   end
```

Figure 3.4: Function for Conjoining Primitive UMO Constraints

It is easily verified that all of these methods run in *linear time*. Therefore, for conjunctive systems with UMO constraints, the $O(n \times w)$ constraint-complexity that we proved in Theorem 3.5.11 is in fact an $O(n \times w)$ *time* complexity. We stress that this conjunctive complexity result *will* apply to systems with preconditions like '$(Month = jan \vee Month > june) \wedge Year = 2000$', because these are conjunctions of

primitive constraints.[7]

3.6.2 Conversion to Datalog

We now describe an optional modification to the output syntax of the query algorithm, to make it generate the *pos* and *neg* formulas in the syntax of datalog rules. This modification does not restrict the generality of the algorithm; is not restricted to conjunctive systems. If it were implemented, then existing datalog systems could be used for the satisfaction-testing phase of the query-processing.[8] The key to understanding this "modification" is the fact that the formula graph constructed by the query algorithm is *already* an implicit form of nonrecursive datalog system. To show this, we now give the construction of the datalog system corresponding to a formula graph. Let g be a formula graph over the variables $\{X_1, \ldots, X_k\}$. For each node n in g, let p_n be an associated predicate symbol. In the datalog system, the program atomic formulas appearing in the heads and the bodies of the rules will *all*

[7] A simple optimization to the definitions of *pos* and *neg* is needed in order to get the stated complexity result. As the definitions stand now, when *pos* is applied to the primitive $(Month = jan \vee Month > june)$, the result is $(pos(Month) \wedge Month = jan) \vee (pos(Month) \wedge Month > june)$. We don't want *pos* and *neg* to "descend" into the primitive constraints, because *then* the disjunctions can lead to a combinatorial blowup in the structure of the resultant formulas. The solution is simple: use the equivalent formula $pos(Month) \wedge (Month = jan \vee Month > june)$.

The general principle is to use the following optimization. For a monotone formula f where all of atomic subformulas of f are comparisons between a fixed variable v and constants, use the following identities: (1) $pos(f) \cong (pos(v) \wedge f)$, and (2) $neg(f) \cong neg(v) \vee (pos(v) \wedge f^-)$.

[8] In order to obtain an improvement in constant factors, rather than using an off-the-shelf datalog system, in our implementation we have built a customized evaluator that works directly on the logic graph.

take the form $p_n(X_1, \ldots, X_k)$. The program atomic formulas will therefore differ *only* by their predicate symbols p_n. Hence a global meaning will be imparted to the variables $X_1, ..., X_k$, since X_i will always unify with itself.

For each node n in the formula graph, we now give the evident construction of the set of datalog rules associated with n. First, if n is a root node labeled with constraint c, then introduce the primitive rule:

$$p_n(X_1, \ldots, X_k) \leftarrow c$$

Second, if n is an *AND* node with parents n_1, n_2, then introduce the rule:

$$p_n(X_1, \ldots, X_k) \leftarrow p_{n_1}(X_1, \ldots, X_k),\, p_{n_2}(X_1, \ldots, X_k)$$

Third, if n is an *OR* node with parents n_1, n_2, then introduce the rules:

$$p_n(X_1, \ldots, X_k) \leftarrow p_{n_1}(X_1, \ldots, X_k)$$
$$p_n(X_1, \ldots, X_k) \leftarrow p_{n_2}(X_1, \ldots, X_k)$$

3.6.3 The Query Tool

Our query tool for acyclic production systems implements the specification for query tools that we gave in Section 1.1.4 on page 15. It is implemented as a system of objects written in about 7500 lines of C++, and compiled under g++. The implementation platform is Solaris 5.6.

Overall Architecture

The tool consists of a query module, which implements the query algorithm presented in Section 3.1, plus the user-interface component of the Thinksheet system.

As we described in Section 1.5.5, the Thinksheet system [26] consists of an interpreter for acyclic productions systems, along with various user-interfaces (spreadsheet, CGI, and TCL). The thinksheet rule interpreter provides an API through which the interface programs are informed of which questions and conclusions to present to the user.

Our query tool is constructed by keeping the thinksheet interface program and replacing the rule interpreter with a query module that implements the API provided by the rule interpreter. Through this connection, the query module is able to direct the behavior of the user interface.

The input to the query tool consists of (1) a user-specified query formula q, and (2) the current attainable binding b whose assignments were supplied at the interface. As we specified in Section 1.1.4, the role of the query tool is to determine and present the reachability/inevitability status of the conclusions and questions in relation to the extended query formula $q' \equiv q \wedge (\wedge b)$. We recall from that specification that (1) unreachable nodes should be hidden, (2) inevitable nodes should be highlighted, (3) any question v which is asked should have for its menu of choices the set $choices_s(v, q')$. The totality of these requirements determines the public methods that the query module must provide.

In the interactive version of the query tool, input and output take place through a spreadsheet user interface (provided by the thinksheet system). When the spreadsheet is controlled by the query module, its behavior conforms to the specifications for a query tool that we presented in Section 1.1.4. In particular, each node of the production system is presented at a cell of the spreadsheet. When the user issues a query formula, the unreachable nodes are made to vanish from the display.

Figure 3.5 shows a screendump of the spreadsheet interface, for the trial stragegy production system that we presented on page 54. The picture shows the state of the system just after the user has entered the query formula q shown in the bottom window, viz., $E2 = 2$, which stands for the formula '$Gun = yes$'. Under this query, both of the conclusions *Victory* and *Impeach* are shown as *Possible*—since they are q-reachable but not q-inevitable. Because there is only a single value for the variable *Gun* that is consistent with this query, viz., 2, the system has automatically instantiated it to 2. The questions *CrimeScene* and *Trigger* are shown in a different color, in order to indicates that they are inevitable under this query. [9]

The Query Module

The query module is implemented as a system of objects written in C++. The design of the classes directly reflects the algorithms and data structures which we have already presented: (1) the query algorithm, which constructs a formula graph with distinguished *pos* and *neg* nodes, (2) the DNF-based evaluation of this graph, and (3) the methods for processing the constraints the UMO domain. In general, there is one software class for each kind of object referred to by the algorithms, *e.g.*, a class for the formula graph, for the nodes of the formula graph, for the tuples and for the constraint stores.

The query module is presented to the user-interface program as a "large object" that encapsulates the formula graph. We now describe the methods of this object, according to the following three phases of the query-processing: (1) the construction

[9]Recall that a question is inevitable under query q iff the question must be answered in any final binding that satisfies q.

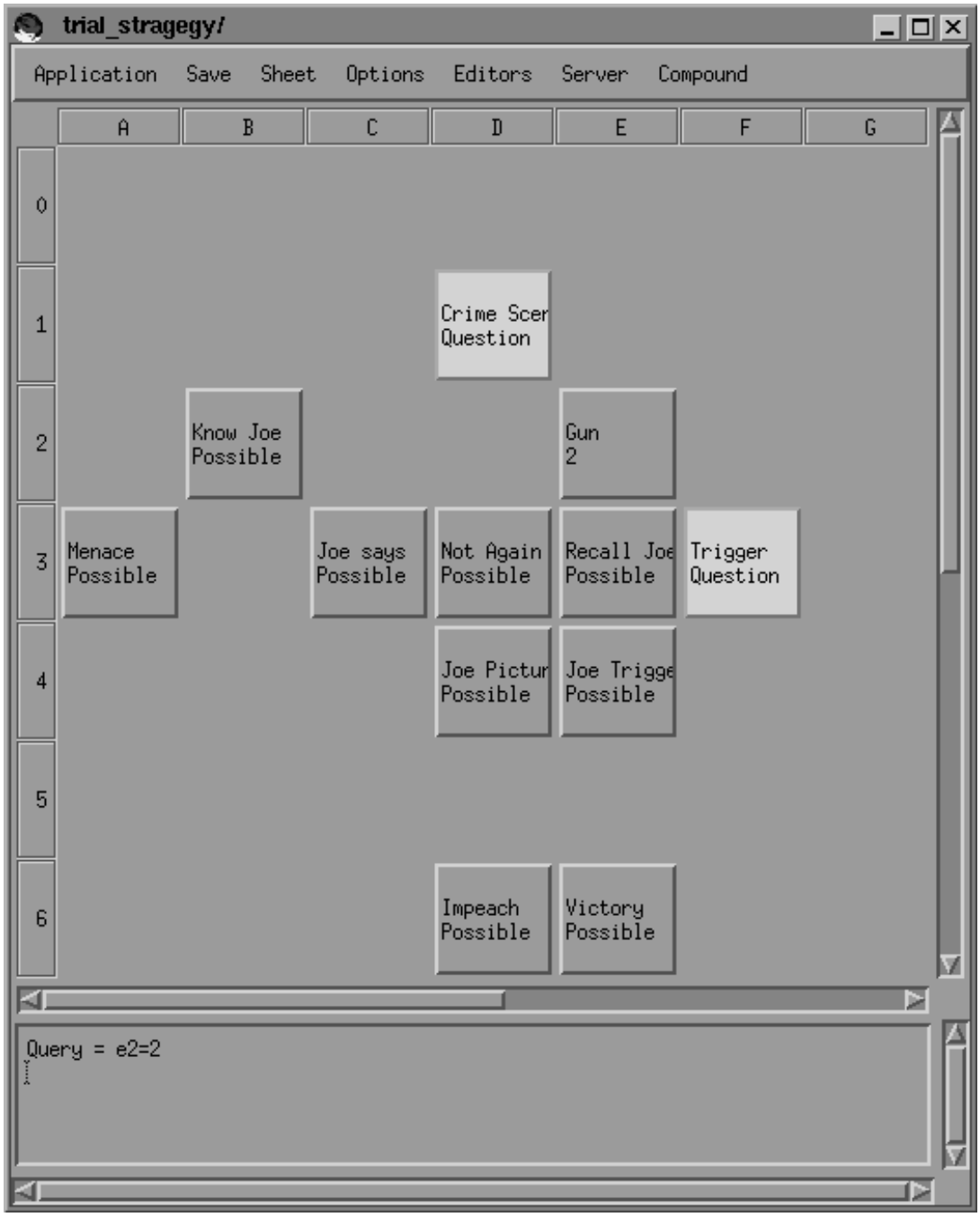


Figure 3.5: The Spreadsheet Interface to the Query Tool

of the formula graph, (2) the setting of the query formula and the processing of assignments from the user, and (3) the reporting of the reachability/inevitability status of the questions and conclusions.

Construction and Evaluation of the Formula Graph

The method *build(System s)*, which takes a description of the acyclic production system s as its argument, builds a formula graph by making repeated called to the routines in Figure 3.1 on page 3.1: for each variable x, it constructs the formulas $pos_s(x)$ and $neg_s(x)$, and for each conclusion c, it constructs $pos_s(precond_c)$ and $neg_s(precond_c)$. In this process, the query module also builds up the maps *posmap* and *negmap*, which serve as indicies to the distinguished nodes of the formula graph: for each variable or conclusion y, the value of $posmap[y]$ is the node in the formula graph for $pos(y)$, and $negmap[y]$ is the node for $neg(y)$.

The formula graph consists of a set of *AND*, *OR* and constraint nodes. These *formula-nodes* are created by the evaluation of the conjunction and disjunction operators in the right-hand-side expressions of the *pos* and *neg* routines. Each *AND* and *OR* node contains a pointer to each of its parents. Attached to each formula-node is a *Store* object, which holds a constraint-based representation for its satisfying bindings. These Store objects are created by the DNF-based evaluation which we described in Section 3.5.1.

Each Store is organized as a set of *Tuples*. Each Tuple represents a conjunction of primitive constraints. The Tuple class is a virtual base class that provides the following methods: (1) conjoin two Tuples, (2) test for satisfiability, (3) project onto a set of variables, and (4) if one exists, find a satisfying binding. The system

also provides an *UmoTuple* class that implements the Tuple specification for the application domain of UMO constraints. The methods for this class were presented in Section 3.6.1. The system maintains the invariant that all Tuple objects held in the Stores are satisfiable. To enforce this invariant, whenever a new Tuple is created, it is immediately tested for satisfiability, and, if unsatisfiable, is discarded.

The Stores provide three methods: disjunction, conjunction, and test for satisfiability. Two Stores are disjoined by forming the union of their respective sets of Tuples. Two Stores are conjoined by forming the set of all conjunctions of pairs of Tuples from the respective Stores (and discarding all of the unsatisfiable conjunctions). The satisfiability test for a Store is therefore just the test for whether it contains any Tuples.

The Store representing the solutions to a node in the formula graph is computed by a lazy, memoized, bottom-up DNF-based evaluation. The store for a root node in the formula graph can be computed once and for all, using the fixed constraint that labels the node. When the Store of an *OR* node is requested, then if it has already been computed, a pointer to the memoized Store is returned; otherwise, the disjunction of the Stores at the parents is computed, memoized, and returned. Requests for the Store of an *AND* node are handled similarly.

The satisfiability of a node in the graph can be tested by using the DNF-based evaluation to obtain its Store, and then checking to see whether the store has any Tuples. Nevertheless, since the satisfiability-test only requires one bit of information about the Store—but not the whole Store itself—we should use the following optimized procedure. Like the Store itself, we will memoize the satisfiability bit for each of the nodes. When the satisfiability bit is requested, but has not yet been

computed and memoized, then we do the following. First, if the Store is already memoized, then the method uses the Store to report the satisfiability of the node. Otherwise, if the node is an *OR* node, then the method calls the satisfiability-testing methods of the parents, and returns True if either one is satisfiable. Otherwise, if the node is an *AND* node, then the Store for the node can be partially constructed; if ever one Tuple enters into the Store, then the routine can return immediately and report that the node is satisfiable.[10]

Processing the Query The query module is given the user-specified query formula q and the current attainable binding b. Recall that its function is to report on the status of the questions and conclusions under the extended query formula $q' \equiv q \wedge (\wedge b)$. The query module is "lazy" in that it does not compute the status of all the questions and conclusions when q' is specified, but rather, it computes the status of a node when requested to do so by the interface program.

In order to compute the status information for node n, the query module must determine the satisfiabilities of the formulas $pos(q') \wedge pos(n)$ and $pos(q') \wedge neg(n)$. Furthermore, if n is a question node and the menu of choices is requested for n, then it must return the projection of $pos(q') \wedge pos(n)$ onto n.

The first step in processing the query, therefore, is to compute the solutions

[10]This optimized satisfiability-test is needed for the $O(n \times w)$ complexity result for inevitability queries to conjunctive systems which we proved in Section 3.5.2. The inevitability queries to a conjunctive production system give rise to disjunctive *neg* formulas. We are fortunate, however, in that, for queries to conjunctive production systems, the only use of the *OR* nodes in the formula graph is the satisfiability-test. Hence, using the optimization that we have just described, the Stores for the *OR* nodes will never need to be constructed.

to $pos(q')$, *i.e.*, the solutions to $pos(q) \wedge pos(\wedge b)$. The next proposition leads to a useful optimization.

Proposition 3.6.1 *For an acyclic production system s with monotone precondition, and an attainable binding $b \in A_s$, the formulas $(\wedge b)$ and $pos_s(\wedge b)$ are logically equivalent.*

Corollary 3.6.2 *For an acyclic production system s with monotone preconditions, and an attainable binding $b \in A_s$, the formulas $pos_s(q')$ and $pos_s(q) \wedge (\wedge b)$ are logically equivalent.*

Proof

Because $pos(q') = pos(q) \wedge pos(\wedge b) = pos(q) \wedge (\wedge b)$. ■

This corollary shows that we can optimize the computation of $pos(q')$ by omitting all the conjuncts $pos(x)$ for variables $x \in vars(b)$. These conjuncts are formally part of the definition of $pos(q')$, but, since b is attainable, the preceding proposition shows that they are already implied by the conjunct $(\wedge b)$.

When the user enters a query formula q, the query module performs the call $pos_s(q)$. This leads to the construction of new nodes in the formula graph. The query module keeps a pointer to the node for $pos(q)$, because its solutions will be needed to answer all subsequent requests from the interface program.

Reporting Back to the Interface Program When the query module is asked about the status of a production-system node n, it proceeds as follows. The nodes for $pos(n)$, $neg(n)$ and $pos(q)$ are already present in the formula graph. So, in order

to compute the satisfiability of $pos(q') \wedge pos(n)$, the query module temporarily builds a conjunction node for $pos(q) \wedge (\wedge b) \wedge pos(n)$, and similarly, it builds a conjunction node for $pos(q) \wedge (\wedge b) \wedge neg(n)$. It then calls the satisfaction-testing methods of these nodes, and uses the resulting two bits of information to report the status, as per the code on page 87. If the menu of choices for a variable v is requested, then the projection method of the temporary *pos* node is called. If a witness binding is required, then the *findSolution* method of the temporary *pos* node is called, and then the core of the solution binding is computed and returned.[11]

Optimizations to Reduce the Amount of Constraint-Processing

We now give two propositions that identify circumstances under which the query module can compute the status of a question or conclusion without having to evaluate the associated *pos* and *neg* nodes.

Definition 3.6.2 (Status of a Node Under a Query) *For an acyclic production system s, formula q, and question $v \in V_s$, define the* status *of n under q to be the triple (r, i, c), where*

- *r is a bit indicating the q-reachability of v*
- *i is a bit indicating the q-inevitability of v*

[11] We can show that, in a monotone system, for any formula f, if b is a binding that satisfies $pos(f)$, then any minimal satisfying sub-binding of b is attainable. Hence, a minimal satisfying binding for $pos(f)$ is a witness to the reachability of f. Furthermore, for our UMO implementation, we can show that all bindings produced by the *findSolution* method are minimal. Therefore, for our UMO implementation, we do not need to call the core algorithm.

- $c = choices_s(v, q)$ *is the menu of choices for* v *under* q

(Recall that $choices_s(v, q)$ *was defined on page 17.) For a conclusion* $c \in C_s$*, define the status of* c *under* q *to be the pair of bits* (r, i)*.*

Proposition 3.6.3 *For a monotone acyclic production system* s*, let* $b \in A_s$ *be an attainable binding. Let* q *be a formula. Let* q' *be the extended query formula* $q \wedge (\wedge b)$*, and suppose that* $pos(q')$ *is satisfiable. Let* n *be a node in* s *that is not a descendent of any of the variables in* $ancestors(FV(q)) \cup vars(b)$*. Then:* $status(n, q') = status(n, true)$*.*

Thus, if n is not a descendant of any of the ancestors of q, then q has no effect on the status of n. Hence, if we compute the solutions to $pos(n)$ and $neg(n)$ once and for all, then to determine the status of n, we need only conjoin the current binding into these pre-computed solutions and then test for satisfiability. Also, for such n, the propagation-based effective value algorithm described in Section 1.5.5 can be used to estimate the status of the node.

The next proposition gives an optimization for the circumstance where the user makes a new answer choice.

Proposition 3.6.4 *For an acyclic production system* s*, let* $b \in A_s$ *be an attainable binding. Let* q *be a formula. Let* v *be a variable in* $ask(b)$*, and assume that* v *is not an ancestor of any variable in* q*. Let* x *be a value in* dom_v*, and let* b' *be the binding* $b \cup \{(v, x)\}$*. Then:*

$$n \notin descendents(v) \Rightarrow status(n, q \wedge (\wedge b')) = status(n, q \wedge (\wedge b))$$

In other words, the status of these nodes n does not need to be recomputed when the user makes an answer choice for v.

3.7 Optimization for Local Decision Trees

Here we give an optimization for reachability queries that exploits a natural and frequent pattern in the input data: production system that contain local regions that are decision trees. By means of a simple mechanism, we get a lower tree-processing complexity than would be obtained using general-purpose datalog algorithms for the satisfiability-testing phase of the query-processing.

The crux of this improvement is not that we are "beating" datalog algorithms *per se*, but rather that: (1) the translation that we presented from production-system queries into datalog rules gives rise to a very specific form of datalog systems, and (2) for the purposes of solving the reachability queries, we need only the solutions to a specific subset of the predicates. Generic datalog algorithms, which are not aware of these special forms, will compute more information than we need to solve our problem. Hence we are able to develop a tailored algorithm that is asymptotically faster.

Local Trees

A local tree in a directed acyclic graph G is a tree-shaped subgraph, which we define as follows. A *tree node* is a node that has one parent. A *local tree* $T = \{r\} \cup T'$ is a set of nodes where (1) $r \in G$ is called the "local root" of T, (2) T' is a set of tree-nodes, and (3) for each $x \in T'$, there is exactly one path in G from r to x. For

an acyclic production system s, define a *local decision tree* to be a set of nodes that is a local tree in the dependency graph for s, such that all of the tree nodes have conjunctive preconditions.

Example 3.7.1 *In the medical decision DAG on page 53, there are three maximal local subtrees:*

1. *The root FolicAcid, by itself*
2. *The root IronLevel, its two children, and the node Ulcer*
3. *Malabsorption, IntestinalDamage, and AntibodyLevel*

Local decision trees are a natural and frequent pattern. In one style of production system, for instance, the system consists of a decision tree along with a set of conclusions, with preconditions that are disjunctions of constraints on the leaves of the tree, that function to summarize the outcome of a session with the tree.

A Measure for the Complexity of Tree-Processing

For purposes of comparing algorithms, we now define a measure for the tree-processing complexity of a query algorithm. The idea is to fix a query formula and the parts of a production system that are not tree nodes, and then to measure, in units of constraints, the complexity of queries as a function of the aggregate size of the local trees. Formally, define the *skeleton* of an acyclic production system to be the set of nodes that have multiple parents, and define two systems to be *tree-equivalent* if their skeletons are identical. For a given query algorithm, a class of tree-equivalent systems, and a query formula, there is a complexity function that

relates the size of the input systems to the maximum number of primitive constraints that will be passed to the constraint solver. If all of these functions are $O(f)$, then we say that the *tree-complexity* of the algorithm is $O(f)$; if any of them are $\Omega(f)$, then we say that the tree-complexity is $\Omega(f)$.

Tree-Complexity of the Queries using Generic Datalog Methods

Here will put a lower bound on the tree-processing complexity of our reachability query algorithm when generic datalog evaluation methods[22, 29, 30, 31, 41, 21, 32, 17]—top-down, bottom-up and hybrid—are used for the satisfiability-testing. In the worst case, for systems with large unbalanced local trees, we now will give easy arguments to show that the worst-case tree-processing complexity is quadratic.

We base our explanation upon an extreme case, which will put the general principal into relief. Once we understand the quadratic behavior in this case, then the quadratic behavior for all systems with large unbalanced local trees trees will become immediately clear.

Consider the degenerate case of a tree that consists of a linear chain of variables $v_1, \ldots, v_n$, along with the corresponding preconditions $p_1, \ldots, p_n$. Let V be shorthand for the vector $(v_1, \ldots, v_n)$ of all the variables. Let Q be all of the constraints in $pos(q)$. Here are the datalog rules for the q-reachability queries:

$r_1(V)$:- p_1, Q.

$r_2(V)$:- p_2, $r_1(V)$.

...

$r_n(V)$:- p_n, $r_{n-1}(V)$.

Then, the variable v_i is q-reachable iff the datalog query $r_i(V)$ succeeds.

When a bottom-up evaluator is called to solve these rules, the evaluator will annotate each predicate with the canonical-form conjunctive tuples that represent its solution set. The purpose of the canonical-form is to support subsequent operations, such as conjunctions and projections, as quickly as possible. In particular, inconsistent tuples—which contribute nothing to the solution sets—are detected and discarded immediately. General result: after bottom-up evaluation is completed, each predicate r_i is decorated with the canonical-form representation for the conjunction $Q \wedge p_1 \wedge \ldots \wedge p_i$. Conclusion: the size of the *output* of bottom-up evaluation is quadratic.[12]

To solve our problem using a top-down evaluator, we would make repeated calls to the evaluator, for each of the goals $r_i(V)$. Each time that we call $r_i(V)$, then the recursive mechanism of top-down evaluation causes all of the constraints $Q, p_1, \ldots, p_i$ to be collected and conjoined. Hence, the total running-time is $\Omega(n^2)$, in the worst-case.[13]

[12]One might wonder about the proposed canonical form in which we just store the pair (p_i, x), where x is a pointer to this same canonical form for $Q \wedge p_1 \wedge \cdots \wedge p_{i-1}$. This form amounts to using the syntaxtic formulas themselves as the canonical form. But this form would not be used in practice, because the conjunctive formulas are unbounded in size, even when there are a small number of variables and the individual constraints are small. Furthermore, the running time would still be $\Omega(n^2)$, because the satisfaction-test performed by the evaluator would then require examining a chain of constraints.

[13]What about a scheme where the interpreter saves some information from the call to $r_i(V)$, in order to accelerate the running time of subsequent calls? In general, this approach is called top-down evaluation with memoing—a hybrid scheme that combines aspects of top-down and bottom-up evaluation. For our specific datalog systems, the thing that could be be saved up from the call to $r_i(V)$ is the conjunctive tuple that gives the solution to r_i. Then, using this

It is easy to see, now, that we have in fact proven the following, more general, result: using generic datalog evaluation methods, the cost of evaluating a node in a tree is $\Omega(w)$, where w is the weight of the node in the graph, *i.e.*, the sum of the sizes of the preconditions of its ancestors. Hence, for these reachability queries, their overall complexity is $\Theta(n \times w)$, where w is the mean weight of the tree nodes. Therefore, their worst-case behavior, $\Omega(n^2)$, is achieved on classes of production systems that contain fully unbalanced local decision trees. We summarize in the following proposition.

Proposition 3.7.1 *Using generic datalog methods for satisfiability testing, the worst-case tree-complexity of reachability-query testing is* $\Omega(n^2)$.

The Optimization

Our algorithm is a modified bottom-up evaluation, that constructs the canonical-form tuples only for the datalog predicates that arise from production-system nodes which are *outside* of the local decision trees. For the predicates arising from the tree nodes, we don't need the canonical-form tuples in the final result, but just the results of their satisfiability test. Hence, for these predicates, our algorithm can use incremental operations that update a global constraint store.

Specifically, we optimize the DNF-based, bottom-up evaluation of the formula graph in the following manner. Whenever the bottom-up evaluation reaches the

memoization, we could make the top-down calls in the order $r_1(V)$, $r_2(V)$, etc. But notice now, that with this modification, the evaluation has become indistinguishable from basic bottom-up evaluation! (This identity only holds because the rules for the trees are conjunctive.) Hence, the argument, given above, as to why bottom-up evaluation is $\Omega(n^2)$, comes into force.

root of a local tree, it is temporarily suspended, and the routine *query_tree* shown in Figure 3.6 is called to process the nodes in the local tree. When this routine returns, the bottom-up evaluation is resumed.

The arguments to *query_tree* are (1) the variable r at the root of the local tree and (2) a constraint store $posQ$ holding the tuples for $pos(q)$. The routine starts by initializing a global, mutable constraint store to hold the tuples for $pos(q) \wedge pos(r)$. Then, for each child n of r with one parent, it calls the recursive subroutine *query_subtree(n)*. The subroutine performs the following steps:

1. Conjoin the constraints for n into the global store
2. Determine the satisfiability of n and mark n accordingly
3. Recursively call the children of n that have one parent
4. Undo the effects of the conjunction operation

The correctness of the routine follows from the next proposition.

Proposition 3.7.2 *For an acyclic production system s, let x be a variable in $V_s \cup W_s$, and c a conclusion in C_s. Suppose that $precond_v$ and $precond_c$ are conjunctive formulas. Then:*

1. *If x has only one parent p, then $pos_s(x) = pos_s(p) \wedge precond_x \wedge indom_x$*
2. *If c has only one parent p, then $pos_s(c) = pos_s(p) \wedge precond_c$*

(An important detail is how to compute the solutions to $pos(q)$, which are required *prior* to the modified bottom-up evaluation that we have just described.

```
global Store cstore

query_tree(Variable r, Store posQ)   ; r is root of local tree
begin
   cstore := posQ ∧ pos(r).store()
   for each child n of r that has one parent do
      query_subtree(n)
end

query_subtree(Node n)
begin
   if n is a conclusion then
      cstore.conjoin(precond_n)
   else
      cstore.conjoin(precond_n ∧ indom_n)
   endif
   n.reachable := cstore.satisfiable()
   for each child m of n that has one parent do
      query_subtree(m)
   end
   cstore.pop()        ; undo the cstore.conjoin
end
```

Figure 3.6: Subroutine for Querying a Local Tree

In order to obtain these solutions, we first marking all of the nodes that are ancestors of $pos(q)$, and then perform, on these nodes, the modified bottom-up evaluation described above, using *true* as the value for the *posQ* argument.)

Proposition 3.7.3 *The tree-complexity, measured in units of constraints, of the optimized algorithm is* $O(n)$.

Proposition 3.7.4 *We can provide methods for the UMO constraint domain, so that, for classes of tree-equivalent systems where the individual rules are of bounded size, the time-complexity of reachability queries is* $O(n)$.

Performance Experiments

Appendix B gives an experimental confirmation of the linear-time result that was proven in Proposition 3.7.4.

Synposis of the Optimization

Whence our ability to optimize? In order to answer this concretely, suppose that for each datalog rule $r_i(V)$ which is used to express the reachability of a decision tree node v_i, that we introduce the following auxiliary datalog rule:

$s_i()$:- $r_i(V)$.

The solution to the predicate s_i provides the one bit of information indicating the satisfiability of r_i, *i.e.*, the reachability of node v_i. (Note that we cannot simply eliminate the rule for $r_i(V)$, and replace it with $s_i()$, because $r_i(V)$ is called by the rules for the descendents of v_i.)

To solve the reachability query problem, we need just the solutions to the predicates s_i. Because we don't require the solutions to $r_i(V)$ in the final output, we are able to use efficient destructive operations when creating these solutions—the tuples for $r_i(V)$ will indeed exist in canonical form, but only momentarily, just long enough for the satisfaction bit to be computed. Generic datalog evaluation algorithms, on the other hand, which by definition must compute the solutions to the r_i predicates, are overkill for this application. Compared to these algorithms, our algorithm produces less but sufficient information, and this is why it can run more efficiently.

This datalog optimization is applicable because of a combination of special conditions: (1) all predicates have the same arguments, (2) the rules in the tree consist of a constraint and a call to a single parent, and (3) we need the results of evaluating just some of the rules. Furthermore, in order for it to yield an asymptotic complexity improvement, the constraint domain must be simple enough that incremental conjunctions can be performed in sublinear time with respect to the size of the constraint stores.

Because of these conditions, the general significance of the optimization is not as a contribution of general-interest to datalog methods. Rather, it is that in a new context that we have created—queryable expert systems—the important problem of reachability queries to local decision trees leads to a special datalog problem that meets all of the conditions for our datalog optimization. Thus, for the satisfiability-testing problem that arises from our theoretical solution to the reachability query problem, it pays to use our tailored datalog algorithm in place of the normal, general-purpose datalog algorithms.

Chapter 4

Queries with Negation

In this chapter, we specify an operational behavior for acyclic production systems with general boolean preconditions. Then we show that a query algorithm for these systems can be obtained by a slight extension of the query algorithm for monotone systems, which we presented in Chapter 3.

4.1 Systems with General Boolean Preconditions

We will define our operational semantics so that the precondition $\neg f$ will become activated whenever the formula f is rejected by the system, either on account of it being false, or inapplicable. For example, the precondition $\neg(\mathit{SpouseIncome} \leq 50000)$ should fire either if *SpouseIncome* is defined and exceeds 50000, or if the user has no spouse, in which case *SpouseIncome* is inapplicable. To repeat, the precondition $\neg f$ should become active when f is rejectable—either false or inapplicable—in the current attainable binding b. Here is an example application of negations in the preconditions.

Example 4.1.1 *Recall the production system on page 54, which contains the strategy that a prosecuting attorney plans to use when cross-examining the defendant in a murder trial. Given that this attorney intends to win the case, there are only two possible outcomes: the conclusion Impeach, indicating that the defendant has given testimony which contradicts the evidence, and Victory, which indicates that the defendant has confessed to the crime.*

The attorney might want to verify that there are no loopholes in this strategy, viz., that there is no complete set of answers that the defendant could provide such that neither the conclusion Impeach nor Victory would be reached. To test for this, an auxiliary conclusion called Loophole could be created, with precondition $\neg precond_{victory} \wedge \neg precond_{impeach}$. *This precondition will become activated whenever the conclusions Victory and Impeach are both rejected. Hence, the Loophole conclusion should be unreachable. This could be verified with a reachability query tool. Furthermore, if there were a loophole, then a query tool should provide a binding that produces the loophole. This binding will serve as a counterexample for use in debugging the strategy.*

Earlier, in Section 2.4, we gave a technical definition for the relationship $b \models\!\!\!=_s f$ under which, assuming the rules of the system s, the formula f is rejectable on the binding b. Recall that, for attainable bindings b, the meaning of $b \models\!\!\!=_s f$ is that all final extensions b' of b fail to satisfy f, *i.e.*, b guarantees the unsatisfiability of f. We would like to use this relation $b \models\!\!\!=_s f$ as the firing rule for the precondition f. In order to do this, however, we must first extend the definition of satisfaction $b' \models f$ to cover general boolean formulas f.

Definition 4.1.2 ($b \models f$)

For a final binding b and a boolean formula f, define $b \models f$ by taking the cases for conjunction, disjunction and atomic formulas from Definition 1.4.6 on page 29, and by defining:

$$b \models \neg f \Leftrightarrow b \not\models f$$

In other words, we extend the definition $b \models f$ by adding the most natural way of handling the case of a negated formula.[1] Now, the relations $b \models^{\pm}_{s} f$ and $b \models^{-}_{s} f$ are well-defined for general boolean formulas f. The following useful proposition is immediate.

Proposition 4.1.1 *For a regular system s, binding b, and boolean formula f,*

$b \models^{-}_{s} f \Leftrightarrow b \models^{\pm}_{s} \neg f$

Hence, the principle that $\neg f$ should fire under b just when $b \models^{-}_{s} f$ is equivalent to the condition that $\neg f$ should fire whenever $b \models^{\pm}_{s} \neg f$. Generalizing from this, we

[1] We restrict this extension of $b \models f$ to just the final bindings, for the following reason. In general, the $\perp$ values in a binding indicate places where the binding is unspecified. For final bindings b, however, since $ask(b) = \{\}$, the $\perp$ values have a stronger meaning: they indicate places where the binding *cannot* be specified. Assuming that the system allows the user to specify all variables that are applicable, it follows that the $\perp$ values in a final binding are inapplicable values. Hence, the failure of a final binding to satisfy a formula indicates that the formula is either false or depends upon inapplicable values, *i.e.*, that it is rejectable.

For a simple illustration of the need for this restriction to final bindings, consider the empty binding $b_0 = \{\}$. Here, the $\perp$ values do not indicate inapplicability, but merely that the values are unspecified. Now, for any non-trivial formula f, we will have that b_0 fails to satisfy f. But we almost certainly do not want $\neg f$ to fire in the empty binding, since the further specification of values may make f become true!

arrive at the following specification of our desired operational mechanics.

Definition 4.1.3 (Firing Rule for Systems with Boolean Preconditions) *In the current attainable binding b, the precondition f should ideally fire whenever $b \models^{\pm}_s f$. I.e., f should fire whenever its satisfaction is guaranteed by the current binding.*

We now turn our attention the the structure of the final bindings for these idealized systems. The next definition, which expresses a necessary and sufficient condition for a binding to be final in a monotone acyclic production system—see Theorem 3.2.2—will provide a key concept for the analysis of general boolean production systems.

Definition 4.1.4 (Solution Bindings S_s) *For an acyclic production system s, define the* solution bindings *for s, denoted by S_s, to be those bindings which satisfy the following conditions:*

1. $\forall\, v \in V_s$: $\mathsf{if}\ b \models \mathit{precond}_v\ \mathsf{then}\ b \models \mathit{indom}_v\ \mathsf{else}\ b(v) = \bot$
2. $\forall\, w \in W_s$: $\mathsf{if}\ b \models \mathit{precond}_w\ \mathsf{then}\ b(w) = \mathit{val}(\mathit{term}_w, b)\ \mathsf{else}\ b(w) = \bot$

The next proposition shows that, for boolean production systems with the idealized firing rule, the final bindings coincide completely with the solution bindings. The proof is based upon the fact that, for final bindings $b \in F_s$, we have that $b \models^{\pm}_s f \Leftrightarrow b \models f$, and $b \models^{\mp}_s f \Leftrightarrow b \not\models f$.

Proposition 4.1.2 *For a boolean system s conforming to Definition 4.1.3:*

$$F_s = S_s$$

Since the definitions of q-reachability and q-inevitability in s depend only upon the final bindings F_s, the query problem for these ideal systems may be re-visualized in terms of the solution bindings S_s. The conceptual advantage of this perspective is that, whereas the final bindings are defined in a procedural way, the equivalent solution bindings are defined in a purely declarative way, whereby each rule is seen as a constraint on the solution bindings.

We now define the notion of a firing rule for boolean production systems that approximates the ideal behavior in a way which is sufficient for the purposes of querying.

Definition 4.1.5 (Final-Exact Systems)

An operational interpretation of a boolean production system s is said to be final-exact*, or exact for final bindings, if:*

$$F_s = S_s$$

For a given syntactic specification s of a boolean production system, all of the final-exact interpretations of s—including the ideal interpretation—are indistinguishable from the standpoint of the reachability/inevitability queries.

Although it is difficult to perfectly implement the idealized semantics—because, for attainable bindings b, the determination of the relation $b \models^{\pm}_s f$ makes a statement about all of the final extensions of b—the firing rule used by the Thinksheet systems [26] gives an operational interpretation that is final-exact. That firing rule is the following: the precondition $precond_x$ will fire in binding b whenever

$\mathit{eff}_s(\mathit{precond}_x, b) = \mathsf{True}$, where the effective value function is defined in Figure 1.8 on page 59. Here is the formal statement of the claim.[2]

Theorem 4.1.3 *For a general boolean system s with the effective-value-based firing rule:*

$$F_s = S_s$$

Also, as the next proposition shows, the thinksheet systems are regular.

Proposition 4.1.4 *For a general boolean system s with the effective-value-based firing rule: s is a regular interactive system.*

4.2 The Extended Query Algorithm

In anticipation of this part of the thesis, the cases for negations were already written into the query algorithm that we gave on pages 87 and 86. Specifically, these new cases are: (1) $\mathit{pos}_s(\neg f) \equiv \mathit{neg}_s(f)$, and (2) $\mathit{neg}_s(\neg f) \equiv \mathit{pos}_s(f)$. We now claim that with these additions, the query algorithm is also correct for systems with boolean preconditions.

[2]In the introduction, we mentioned that our theory could help to clarify the meaning of the effective-values. With some effort, the following relationships can be proven for thinksheet systems s: for a binding b, formula f, and term t: (1) $\mathit{eff}_s(f, b) = \mathsf{True} \Rightarrow b \models^{+}_s f$, (2) $\mathit{eff}_s(f, b) = \mathsf{False} \Rightarrow b \models^{-}_s f$, (3) $\mathit{eff}_s(t, b) = \mathsf{True} \Rightarrow b \models^{+}_s (t = t)$, (4) $\mathit{eff}_s(t, b) = \mathsf{False} \Rightarrow b \models^{-}_s (t = t)$. We omit the proof, because the next theorem is all that is required to show that our query algorithm is correct for thinksheet systems.

Theorem 4.2.1 *The query algorithm shown on pages 87 and 86 is correct for acyclic production systems with general boolean preconditions.*

Example 4.2.1 *We show how this extended algorithm solves the Loophole query given in the example at the beginning of this chapter. Recall that the precondition of Loophole is* $\neg precond_{victory} \wedge \neg precond_{impeach}$*, and that we want to test for the reachability of this conclusion. For this, we compute the pos formula, as follows:*
$pos(precond_{loophole}) = pos(\neg precond_{victory}) \wedge pos(\neg precond_{impeach}) =$
$neg(precond_{victory}) \wedge neg(precond_{impeach})$.
By a routine calculation, it can be verified that this formula is unsatisfiable, and hence the Loophole conclusion is unreachable.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

There is a dichotomy among existing computer-based information systems. On the one hand, databases are open-ended in their ability to answer a potentially unlimited number of questions from the user, but offer no guidance help a naive user to navigate through the mass of information that they contain. The existing interactive expert systems, on the other hand, do enter into a dialogue with the user that is tailored to the user's specific requirements, but the user is reduced to the passive condition of answering questions and being told things. Ideally, we would also like the user to be to assume an active role and ask the system about the knowledge that it contains.

In this thesis, we began by defining a paradigm for queries to an interactive expert system: Which conclusions are reachable, and which are inevitable, under a query condition? For instance, is conclusion *diabetes* inevitable if the input for *bloodSugar* exceeds 100? For general interactive systems, this problem is undecid-

able.

Then we undertook a theoretical investigation of a large class of interactive systems—the "regular" systems, which never retract a question once it has been posed to the user—with an eye towards developing query algorithms for cases of these systems. This led us to introduce a series of useful concepts, such as the core of applicable assignments within a binding, the applicability of terms under a binding, and the relationship $b \models^{\pm}_{s} f$ under which a binding justifies the acceptance of a formula by the system.

Then we chose a simple and useful language of acyclic production systems, which are regular, and used these theoretical results to develop a working query algorithm. We have implemented this algorithm in a working system. It is the first interactive system we know of that can be queried about its conclusions. The general significance of this *queryable expert system* is that, if we liken a typical consultation system to a doctor who interrogates patients and makes diagnoses, we can now ask the doctor some questions in order to learn something about medical reasoning.

5.2 Future Work

New Languages and Algorithms: Our query paradigm raises the challenge for researchers to find other languages and algorithms for queryable expert systems.

Typology of Acyclic Production Systems: It would be useful to develop a typology of acyclic production systems and their applications. What kind of structures in reality are well-modeled by the preconditions of an acyclic production system?

How can we generalize from the structures in trial strategy system that we we showed on page 54? Such systems may also, for example, be used for strategies for convincing an opponent of the correctness of a theory, and handling anticipated counterarguments. Note that the trial strategy system uses a very restricted language of preconditions, involving just conjunctions and disjunctions of comparisons of the form $var = yes$ and $var = no$. Furthermore, it is structured in such a way that whenever a node x is reached, there is a linear chain of answers that leads to the precondition of x being satisfied.[1] This suggests that "path-structured" systems of preconditions are useful for certain classes of strategy applications.

By way of comparison, for relational databases, there is the well-developed entity-relationship model, along with normalization theory, etc., that serves as a guide for applications of relational databases. Systems with preconditions are more subtle and complex than flat relational systems, so more effort is needed to analyze the structures to which they apply.

Connections with Modal Logic? The operators $b \models^{+}_{s} f$, $b \models^{-}_{s} f$, and the formulas $pos_s(f)$, $neg_s(f)$ which we introduced have the flavor of modal operators that assert that a formula f is "justified" and "dis-justified" by the data in a binding. Interestingly, the formulas $pos_s(f)$ and $neg_s(f)$ are expansions—relative to the rules of s—that express the justification and dis-justification of f as formulas *within first order logic*. Is there any meaningful connection here to modal logic? It can be shown, for instance, that the formulas $pos(pos(f))$ and $pos(f)$ are logically

[1] This wouldn't be the case, for example, if the node x had the precondition $y = yes \wedge z = yes$, where y and z are roots. In this case, the binding $\{y = yes,\ z = yes\}$ causes x to be reached, yet the nodes $\{y, z\}$ do not form a linearly ordered subgraph of the dependency graph.

equivalent.

Queryable Relational Expert Systems: An interesting and useful class of acyclic production systems are those in which the domains of the variables may be sets and relations. In this case, the preconditions and answer terms involve membership tests and relational algebra operations such as projection and selection. Such a system, called Attman (attribute management system), is now under development by Dennis Shasha, Eric Simon, et. al.

For an example application, consider an interactive catalog for a large deparment store. The menu for the root node is presented as a table, the rows of which represent the entries in the main catalog. For instance, a row might be: (majorcategory=sportswear, minorcategory=summer). The user browses through this table, and clicks on the rows of interest. The selected subtable then serves as the answer value for the root node. The root node, therefore, is an input node, whose domain is the powerset of the full relation. Subsidiary input nodes will, in a similar way, present the tables for the various departments, and will have precondition such as: *sportwear* is contained in the projection of the root node onto the attribute *majorcategory*.

Suppose that in this application, there are several product tables, each of which have an attribute *color*. It may be useful to get an overview of all of the colors available overall. To do this, we can introduce a derived node, whose answer term is the relational algebra operation that projects the respective tables onto *color*, and then takes the union of all these projections. Then, if the user constrains *color* to be blue or green, all of the rows for products that are neither blue nor green

should disappear from the various menus throughout the system.

It seems to make sense to treat a node with a false precondition as representing the empty table. In this way, for example, a product table with a false precondition would contribute no rows to the derived color table. This represents a divergence from the semantics that we have assumed in this thesis, because now the value $\perp$—which we associate with a node whose precondition is false—is merged with a specific domain value, viz., the empty table $\{\}$. As a consequence, the union operator, which can appear in the answer terms, is nonstrict.

A research project is to develop a reachability/inevitability query algorithm for such systems. Because the union operator is nonstrict, our semantic theory will have to be reworked. It may make sense, for example, to define $pos(x \cup y)$ to be $pos(x) \vee pos(y)$.

Theoretical Questions about Query Complexity There is a conflict between the goals of the system being a powerful questioning agent and of it being efficiently queryable: as the computational abilities of the rule language increase, the querying problem become more difficult. These consideration leads to 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, what are the languages that have minimal query complexity?

Appendix A

Proofs

Proposition 1.1.1 *For a system s, and formula q, assume that $\exists\, b \in F_s$ such that $b \models q$. Let R, I, P be, respectively, the sets of conclusions that are reachable, inevitable, and possible under q. Then R is the disjoint union of I and P.*

Proof

First, claim that $I \cap P = \{\}$. We prove this by showing that $c \in P \Rightarrow c \notin I$. Suppose $c \in P$. This means that $\exists\, b_1, b_2 \in F_s \ s.t.\ b_1, b_2 \models q,\ c \in say(b_1),\ c \notin say(b_2)$. Hence $\exists\, b \in F_s \ s.t.\ b \models q$ and $c \notin say(b)$. Thus, it is not true that $(\forall b \in F_s \ s.t.\ b \models q \ : \ c \in say(b))$, *i.e.*, it is not true that c is q-inevitable, *i.e.*, $c \notin I$.

Now claim that $I \cup P \subseteq R$. First subclaim: $I \subseteq R$. Suppose $c \in I$. Then $\forall b \in F_s \ s.t.\ b \models q \ : \ c \in say(b)$. Since, by assumption, $\exists b \in F_s \ s.t.\ b \models q$, it follows that $(\exists b \in F_s \ s.t.\ b \models q$ and $c \in say(b))$, *i.e.*, that $c \in R$. Second subclaim: $P \subseteq R$. Suppose $c \in P$. Then $\exists b_1, b_2 \in F_s, b_1, b_2 \models q$, $c \in say(b_1), c \notin say(b_2)$. Hence $c \in R$.

Now claim that $R \subseteq I \cup P$. Suppose $c \in R$, *i.e.*, $\exists b \in F_s$ *s.t.* $b \models q$ and $c \in say(b)$. We show that $c \in I \cup P$ by showing that $c \notin I \Rightarrow c \in P$. Suppose that $c \notin I$, *i.e.*, that $\exists b' \in F_s$ *s.t.* $b' \models q$ and $c \notin say(b)$. The combination of these two existential statements says that $c \in P$. ■

Proposition 1.1.2

A conclusion is q-possible iff it is q-reachable but not q-inevitable.

Proof

c is q-possible $\Leftrightarrow$

$\exists b_1, b_2 \in F_s, b_1, b_2 \models q,\ c \in say(b_1),\ c \notin say(b_2) \Leftrightarrow$

$(\exists b_1 \in F_s$ *s.t.* $b_1 \models q$ and $c \in say(b_1))$ & $(\exists b_2 \in F_s$ *s.t.* $b_2 \models q$ and $c \notin say(b_2)) \Leftrightarrow$

c is q-reachable and c is not q-inevitable. ■

Proposition 2.2.1 *A system s is regular iff the following two functions are monotonic with respect to inclusions of bindings:*

1. $\lambda b \in A_s\ .\ vars(b) \cup ask_s(b)$
2. $\lambda b \in A_s\ .\ b \cup calc_s(b)$

Proof

Define:

$f(b) = vars(b) \cup ask_s(b)$

$g(b) = b \cup calc_s(b)$.

Then by definition, s is regular means:

- $\forall b, b' \in A_s \ : \ b \subseteq b' \Rightarrow (ask_s(b) \subseteq vars(b') \cup ask_s(b'))$
- $\forall b, b' \in A_s \ : \ b \subseteq b' \Rightarrow (calc_s(b) \subseteq b' \cup calc_s(b'))$

Also, f, g monotonic means:

- $\forall b, b' \in A_s \ : \ b \subseteq b' \Rightarrow (vars(b) \cup ask_s(b) \subseteq vars(b') \cup ask_s(b'))$
- $\forall b, b' \in A_s \ : \ b \subseteq b' \Rightarrow (b \cup calc_s(b) \subseteq b' \cup calc_s(b'))$

Hence, f, g monotonic $\Rightarrow$ s is regular. Now suppose s is regular. Suppose $b, b' \in A_s$, $b \subseteq b'$. We must show that $vars(b) \subseteq vars(b') \cup ask(b')$, and that $b \subseteq b' \cup calc_s(b')$. These propositions follow immediately from the fact that $b \subseteq b'$. ■

Proposition 2.2.2

A production system where precondition formulas are monotone is a regular interactive system.

Proof

Suppose $b, b' \in A_s$, and $b \subseteq b'$. We must show that (1) $ask(b) \subseteq vars(b') \cup ask(b')$, and (2) $calc(b) \subseteq b' \cup calc(b')$.

Part 1. Suppose that $v \in ask_s(b)$. This means that $b \models precond_v$ and $b(v) = \perp$. Since $precond_v$ is monotonic, $b' \models precond_v$. If $b'(v) \neq \perp$, this means $v \in vars(b')$, and so regularity holds. If $b'(v) = \perp$, then, since $b' \models precond_v$, $v \in ask_s(b')$, and so regularity holds.

Part 2. Suppose that $(w, x) \in calc_s(b)$. I.e., $b \models precond_w$, $b(w) = \perp$, and $val(term_w, b) = x \neq \perp$. Hence, $FV(term_w) \subseteq vars(b)$, and so $FV(term_w) \subseteq vars(b')$. Thus, $val(term_w, b') = val(term_w, b) = x \neq \perp$.

Suppose first that $b'(w) \neq \perp$. Since b' is attainable, $b'(x) = val(term_w, b')$. Hence $b'(w) = val(term_w, b') = val(term_w, b) = x$. So $b'(w) = x$, *i.e.*, $(w, x) \in b'$, and hence regularity holds. Now suppose that $b'(w) = \perp$. Since $precond_w$ is monotonic, $b' \models precond_w$. Since $b' \models precond_w$, $b'(w) = \perp$, and $val(term_w, b') = x \neq \perp$, it follows that $(w, x) \in calc_s(b')$. Hence, the regularity condition holds. ■

Lemma 2.2.3 *For a regular system s,*
$\forall\, b, b' \in A_s \ : \ (b \subseteq b' \wedge b \neq b') \Rightarrow (ask_s(b) \cap vars(b') \neq \{\})$ *or* $(calc_s(b) \cap b' \neq \{\})$

Proof

Suppose $b, b' \in A_s$, $b \subseteq b'$, $b \neq b'$. Let $[v_1 = x_1, \ldots, v_n = x_n]$ be an attainable assignment sequence such that $\{v_1 = x_1, \ldots, v_n = x_n\} = b'$. Let k be the smallest index such that $v_k \notin vars(b)$. Then $\{v_1, \ldots, v_{k-1}\} \subseteq vars(b)$. Let $b_0 = \{v_1 = x_1, \ldots, v_{k-1} = x_{k-1}\}$. Since b_0 is the set of assignments in the attainable assignment sequence $[v_1 = x_1, \ldots, v_{k-1} = x_{k-1}]$, it follows that $b_0 \in A_s$. Since $v_k = x_k$ is an assignment that can occur in state b_0, then either ($v_k \in V_s$ and $v_k \in ask_s(b_0)$) or ($v_k \in W_s$ and $(v_k, x_k) \in calc_s(b_0)$).

Case 1. Suppose that $v_k \in V_s$ and $v_k \in ask(b_0)$. Since $b_0 \subseteq b$, then by regularity, $ask(b_0) \subseteq vars(b) \cup ask(b)$. Since $v_k \in ask(b_0)$, $v_k \in vars(b) \cup ask(b)$. By construction, $v_k \notin vars(b)$. Hence, $v_k \in ask(b)$. Also, since $v_k \in vars(b')$, it follows that $ask(b) \cap vars(b') \neq \{\}$.

Case 2. Suppose that $v_k \in W_s$ and $(v_k, x_k) \in calc(b_0)$. Since $b_0 \subseteq b$, by regularity, $calc(b_0) \subseteq b \cup calc(b)$. Since $(v_k, x_k) \in calc(b_0)$, $(v_k, x_k) \in b \cup calc(b)$. By construction, $v_k \notin vars(b)$, and so $(v_k, x_k) \notin b$. Hence, $(v_k, x_k) \in calc(b)$. Also, since $(v_k, x_k) \in b'$, it follows that $calc(b) \cap b' \neq \{\}$. ■

Proposition 2.2.4 *For a regular system s,*
$\forall b \in A_s$: *b is maximal in* $A_s \Leftrightarrow b \in F_s$

Proof

It is always the case that (b is maximal $\Rightarrow$ b is final), the simple reason that a non-final binding is by definition extendible. Now we use the regularity assumption to show that b not maximal $\Rightarrow$ b not final.

Suppose that $b \in A_s$ is not maximal in A_s, *i.e.*, that for some $b' \in A_s$, we have that $b \subseteq b'$ and $b \neq b'$. Then by Lemma 2.2.3, $(ask_s(b) \cap vars(b') \neq \{\})$ or $(calc_s(b) \cap b' \neq \{\})$. In the first case, $ask_s(b) \neq \{\}$, and so b is not final in s. In the second case, $calc_s(b) \neq \{\}$, and so b is not final in s. ■

Lemma 2.3.1 *For a regular system s,*
$(b_1, b_2$ *compatible and* $b_1, b_2 \in A_s) \Rightarrow b_1 \cup b_2 \in A_s$

Proof

Suppose that $b_1, b_2 \in A_s$, and that b_1, b_2 are compatible. We proceed by induction on the size of $vars(b_1) - vars(b_2)$.

Base case: $|vars(b_1) - vars(b_2)| = 0$. So $vars(b_1) \subseteq vars(b_2)$. Since b_1 and b_2 are compatible, $b_1 \subseteq b_2$. Hence, $b_1 \cup b_2 = b_2 \in A_s$.

Induction step:

Suppose that $|vars(b_1) - vars(b_2)| = m > 0$. Let $[v_1 = x_1, \ldots, v_n = x_n]$ be an assignment sequence that attains b_1, and let k be the smallest index such that $v_k \notin vars(b_2)$. Then $\{v_1, \ldots, v_{k-1}\} \subseteq vars(b_2)$. Let $b_0 \subseteq b_1$ be the binding $\{v_1 = x_1, \ldots, v_{k-1} = x_{k-1}\}$. Since b_1 and b_2 are compatible, $b_0 \subseteq b_1$, and $vars(b_0) \subseteq vars(b_2)$, it follows that $b_0 \subseteq b_2$.

Since b_0 is the set of assignments in the attainable assignment sequence $[v_1 = x_1, \ldots, v_{k-1} = x_{k-1}]$, it follows that $b_0 \in A_s$. Since $v_k = x_k$ is an assignment that can occur in state b_0, then either ($v_k \in V_s$ and $v_k \in ask_s(b_0)$) or ($v_k \in W_s$ and $(v_k, x_k) \in calc_s(b_0)$).

Case 1. Suppose that $v_k \in V_s$ and $v_k \in ask(b_0)$. By regularity, since $b_0 \subseteq b_2$ and $v_k \in ask(b_0)$, then $v_k \in vars(b_2) \cup ask(b_2)$. Since $v_k \notin vars(b_2)$, then $v_k \in ask(b_2)$. Hence, $b_2' \equiv b_2 \cup \{(v_k, b_1(v_k))\}$ is attainable. Also, b_2' is compatible with b_1, and $|vars(b_1) - vars(b_2')| = |vars(b_1) - vars(b_2)| - 1 = m - 1$. Therefore, by induction, $b_1 \cup b_2' \in A_s$. Finally, $b_1 \cup b_2' = b_1 \cup b_2$, and so we have shown that $b_1 \cup b_2 \in A_s$.

Case 2. Suppose that $v_k \in W_s$ and $(v_k, x_k) \in calc(b_0)$. By regularity, since $b_0 \subseteq b_2$ and $(v_k, x_k) \in calc(b_0)$, then $(v_k, x_k) \in b_2 \cup calc(b_2)$. Since $v_k \notin vars(b_2)$, then $(v_k, x_k) \notin b_2$, and so $(v_k, x_k) \in calc(b_2)$. Hence, $b_2' \equiv b_2 \cup \{(v_k, x_k)\}$ is attainable. Since $b_1(v_k) = x_k = b_2'(v_k)$, and b_1, b_2 compatible, it follows that b_1 and b_2' are compatible. Furthermore, $|vars(b_1) - vars(b_2')| = |vars(b_1) - vars(b_2)| - 1 = m - 1$. Therefore, by induction, $b_1 \cup b_2' \in A_s$. Finally, $b_1 \cup b_2' = b_1 \cup b_2$, and so we have shown that $b_1 \cup b_2 \in A_s$. ■

Definition A.1 (*attsub*$_s(b)$) *For a system s and binding b, let* $attsub_s(b) = \{b' \in A_s \mid b' \subseteq b\}$.

Corollary 2.3.2 *For a regular system s and a binding b, within b there is a unique maximal attainable sub-binding, viz., the union of the attainable sub-bindings of b,* $\cup\{b' \in A_s \mid b' \subseteq b\}$.

Proof

The claim to be proven is that if s is regular, then the maximum member of $attsub_s(b)$ is guaranteed to exist. All of the bindings in $attsub_s(b)$ are compatible, since they are sub-bindings of the one binding b. Thus, by Lemma 2.3.1, $U \equiv \cup attsub_s(b) \in A_s$. Also, since $U \subseteq b$, then $U \in attsub_s(b)$. Hence U must be the maximum member of $attsub_s(b)$. ■

Theorem 2.3.3 *For a regular system s, $core_s(b)$ is uniquely defined, and it equals the maximum attainable sub-binding* $\cup\{b' \in A_s \mid b' \subseteq b\}$.

Proof

The theorem claim is that: $core_s(b) = max(attsub_s(b))$. Let b_0 be a binding returned by one of the non-deterministic executions of the procedure for $core_s$. Clearly, b_0 is attainable and $b_0 \subseteq b$. Hence, $b_0 \in attsub_s(b)$.

Claim: b_0 is is maximal in $attsub_s(b)$.

Proof of Claim. Suppose $b_1 \in attsub_s(b)$, and that $b_0 \subseteq b_1$. Then we have that $b_0, b_1 \in A_s$, and $b_0 \subseteq b_1 \subseteq b$. We show that b_0 is maximal by showing that $b_1 = b_0$.

We will use the following contrapositive form of Lemma 2.2.3: $(b_0, b_1 \in A_s,\ b_0 \subseteq b_1,\ ask(b_0) \cap vars(b_1) = \{\},\ calc(b_0) \cap b_1 = \{\}) \Rightarrow b_1 = b_0$. Thus, given our assumptions, if we can show that $ask(b_0) \cap vars(b_1) = calc(b_0) \cap b_1 = \{\}$, then it will follow that $b_1 = b_0$ and we will be done.

Since the algorithm terminated with b_0, it follows that:

$$(\forall v \in ask_s(b_0)\ :\ b(v) \notin dom_v)\ \ \&\ \ (\forall (w,x) \in calc_s(b_0)\ :\ (w,x) \notin b) \qquad (\dagger)$$

Subclaim 1: $ask(b_0) \cap vars(b_1) = \{\}$.

Proof. We prove this subclaim by showing that $v \in ask(b_0) \Rightarrow v \notin vars(b_1)$. Suppose that $v \in ask(b_0)$. Then $(\dagger)$ implies that $b(v) \notin dom_v$. Since $b_1 \subseteq b$, then either $b_1(v) = \bot$ or $b_1(v) = b(v)$. In either case, $b_1(v) \notin dom_v$. Since b_1 is attainable, $b_1(v) \in dom_v \cup \{\bot\}$. Hence $b_1(v) = \bot$, *i.e.*, $v \notin vars(b_1)$.

Subclaim 2: $calc(b_0) \cap b_1 = \{\}$.

Proof. We prove this subclaim by showing that $(w,x) \in calc(b_0) \Rightarrow (w,x) \notin b_1$. Suppose $(w,x) \in calc(b_0)$. Then $(\dagger)$ implies that $(w,x) \notin b$. Thus, since $b_1 \subseteq b$, it follows that $(w,x) \notin b_1$. ■

Proposition 2.3.4 *For a regular system s,*
$A_s = \mathit{fixpoints}(core_s) = image(core_s)$

Proof

We show that $image(core_s) \subseteq A_s \subseteq \mathit{fixpoints}(core_s) \subseteq image(core_s)$.

1) $image(core_s) \subseteq A_s$**.** This claim says that for all bindings b, $core_s(b)$ is attainable. We've already shown that $core_s(b)$ is the maximum attainable sub-binding of b, so *a fortiori* it is attainable.

2) $A_s \subseteq fixpoints(core_s)$**.** I.e., claim that: $\forall b \in A_s : core_s(b) = b$. Proof: if b is attainable, then clearly b is the maximum attainable sub-binding of b, *i.e.*, $b = core_s(b)$.

3) $fixpoints(core_s) \subseteq image(core_s)$**.** For any operator f, $fixpoints(f) \subseteq image(f)$, because: $x \in fixpoints(f) \Rightarrow f(x) = x \Rightarrow x \in image(f)$. ■

Proposition 2.3.5 *The function $core_s$ is a downward closure operator,* i.e.*, it is monotonic, decreasing and idempotent:*

- $b_1 \subseteq b_2 \Rightarrow core_s(b_1) \subseteq core_s(b_2)$
- $core_s(b) \subseteq b$
- $core_s(core_s(b)) = core_s(b)$

Proof

Monotonicity: We've shown that $core_s(b) = max(attsub_s(b))$, *i.e.*, $core_s = max \circ attsub_s$. Now, it is easy to see that $attsub_s$ is a monotonic function from bindings to sets of bindings, and that max is a monotonic function on sets of bindings that have maximum elements. Since $core_s$ is a composition of two monotonic functions, it is itself monotonic.

Decreasing Property: We've already shown that $core_s(b)$ is the maximum attainable sub-binding of b, and so *a fortiori* $core_s(b) \subseteq b$.

Idempotency:

The previous proposition showed that $image(core_s) = fixpoints(core_s)$. Hence, since $core_s(b) \in image(core_s)$, then $core_s(b) \in fixpoints(b)$, *i.e.*, $core(core(b)) = core(b)$. In other words, $core_s$ is idempotent.[1] ■

Proposition 2.3.6 *For a regular system s and binding b,*
$F_s(b) \neq \{\}$.

Proof

We've shown that $core_s(b)$ exists, and is attainable. Since every attainable binding can be completed to a final binding, the set $F_s(b)$ is nonempty. ■

Proposition 2.3.7 *For a regular system s and binding b,*
$b \in F_s \Leftrightarrow F_s(b) = \{b\}$.

Proof

$b \in F_s \Rightarrow b \in A_s \Rightarrow core(b) = b \Rightarrow F_s(b) = \{b' \in F_s \mid b' \supseteq b\} = \{b\}$.

Since $F_s(b) \subseteq F_s$, then $F_s(b) = \{b\} \Rightarrow b \in F_s$. ■

Proposition 2.4.1 *For a regular system s and formula f,*
$\nexists b \ s.t. \ (b \models^{+}_{s} f) \wedge (b \models^{-}_{s} f)$

[1] In fact, it is easy to show that a function f is idempotent iff $image(f) = fixpoints(f)$.

Proof

By definition,

- $b \models^{\pm}_s f \Leftrightarrow (\forall b' \in F_s(b) \ : b' \models f)$
- $b \models^{=}_s f \Leftrightarrow (\forall b' \in F_s(b) \ : b' \not\models f)$

Since $F_s(b) \neq \{\}$, it is clear that these conditions cannot both be true. ■

Proposition 2.4.2 *For a regular system s, binding $b \in F_s$, formula f, and conclusion $c \in C_s$:*

1. $b \models^{\pm}_s f \Leftrightarrow b \models f$
2. $b \models^{=}_s f \Leftrightarrow b \not\models f$
3. $b \models^{\pm}_s c \Leftrightarrow c \in say_s(b)$
4. $b \models^{=}_s f \Leftrightarrow c \notin say_s(b)$

Proof

Suppose that $b \in F_s$. Then $F_s(b) = \{b\}$. Now,

$b \models^{\pm}_s f \Leftrightarrow (\forall b' \in F_s(b) \ : \ b' \models f) \Leftrightarrow b \models f$

$b \models^{=}_s f \Leftrightarrow (\forall b' \in F_s(b) \ : \ b' \not\models f) \Leftrightarrow b \not\models f$

$b \models^{\pm}_s c \Leftrightarrow (\forall b' \in F_s(b) \ : \ c \in say_s(b')) \Leftrightarrow c \in say_s(b)$

$b \models\!\!\!=_s f \Leftrightarrow (\forall b' \in F_s(b) \ : \ c \notin say_s(b')) \Leftrightarrow c \notin say_s(b)$ ■

Theorem 2.6.1 *For a regular system s, conclusion c, formula q and variable v,*

- *c is q-reachable* $\Leftrightarrow (\exists\, b \in B \ : \ b \models\!\!\!\stackrel{+}{=}_s q$ *and* $b \models\!\!\!\stackrel{+}{=}_s c)$
- *c is q-inevitable* $\Leftrightarrow (\nexists\, b \in B \ : \ b \models\!\!\!\stackrel{+}{=}_s q$ *and* $b \models\!\!\!=_s c)$
- $(b \models\!\!\!\stackrel{+}{=}_s q$ *and* $b \models\!\!\!\stackrel{+}{=}_s c) \Rightarrow$

  each binding in $F_s(b)$ *witnesses the q-reachability of c*

Proof

Here we restate and then prove each of the three claim. The second claim is restated in an equivalent form, by replacing the biconditional $\neg\alpha \Leftrightarrow \neg\beta$ with the equivalent $\alpha \Leftrightarrow \beta$.

Claim 1: $(\exists\, b \in F_s \ : \ b \models q$ and $c \in say_s(b)) \Leftrightarrow (\exists\, b \in B_s \ : \ b \models\!\!\!\stackrel{+}{=}_s q$ and $b \models\!\!\!\stackrel{+}{=}_s c)$
First suppose that $b \in F_s$, $b \models q$, and $c \in say(b)$. By Proposition 2.4.2, $b \models\!\!\!\stackrel{+}{=}_s q$, and $b \models\!\!\!\stackrel{+}{=}_s c$. To show the converse, suppose that for some $b \in B_s$, $b \models\!\!\!\stackrel{+}{=}_s q$, and $b \models\!\!\!\stackrel{+}{=}_s c$. This means that $(\forall b' \in F_s(b) \ : \ b' \models q$ and $c \in say(b'))$. Since $F_s(b) \neq \{\}$, it follows that $(\exists\, b' \in F_s \ : \ b' \models q$ and $c \in say(b'))$.

Claim 2: $(\exists\, b \in F_s \ : \ b \models q$ and $c \notin say_s(b)) \Leftrightarrow (\exists\, b \in B_s \ : \ b \models\!\!\!\stackrel{+}{=}_s q$ and $b \models\!\!\!=_s c)$
First suppose that $b \in F_s$, $b \models q$, and $c \notin say(b)$. By Proposition 2.4.2, $b \models\!\!\!\stackrel{+}{=}_s q$, and $b \models\!\!\!=_s c$. To show the converse, suppose that for some $b \in B_s$, $b \models\!\!\!\stackrel{+}{=}_s q$, and $b \models\!\!\!=_s c$. This means that $(\forall b' \in F_s(b) \ : \ b' \models q$ and $c \notin say(b'))$. Since $F_s(b) \neq \{\}$, it follows that $(\exists\, b' \in F_s \ : \ b' \models q$ and $c \notin say(b'))$.

Claim 3: $b \models^{\pm}_s q$ and $b \models^{\pm}_s c \Rightarrow \forall b' \in F_s(b) : b' \models q$ and $c \in say(b')$.

Suppose that for some $b \in B_s$, $b \models^{\pm}_s q$, and $b \models^{\pm}_s c$. This means that $(\forall b' \in F_s(b) : b' \models q$ and $c \in say(b')$. I.e., all of the members of $F_s(b)$ are witnesses to the q-reachability of c. ■

Corollary 2.6.2 *For a regular system s, suppose that P and N are relations between bindings b and formulas (and conclusions) f, such that the following conditions hold:*

- $P(b, f) \Rightarrow b \models^{\pm}_s f$
- $N(b, f) \Rightarrow b \models^{\mp}_s f$
- $(b \in F_s$ *and* $b \models^{\pm}_s f) \Rightarrow P(b, f)$
- $(b \in F_s$ *and* $b \models^{\mp}_s f) \Rightarrow N(b, f)$

Then for conclusions c, formulas q and variables v:

- *c is q-reachable* $\Leftrightarrow (\exists b \in B : P(b, q)$ *and* $P(b, c))$
- *c is q-inevitable* $\Leftrightarrow (\nexists b \in B : P(b, q)$ *and* $N(b, c))$
- $P(b, q)$ *and* $P(b, c) \Rightarrow$
  each binding in $F_s(b)$ witnesses the q-reachability of c

Proof

Claim 1. First suppose that c is q-reachable. Then by Theorem 2.6.1, for some $b \in B$, $b \models^{\pm} q$ and $b \models^{\pm} c$. Choose any $b' \in F_s(b)$. Then $b' \models q$ and $c \in say(b')$, and, since $b' \in F_s$, then $b' \models^{\pm} q$ and $b' \models^{\pm} c$. Hence, by the theorem assumptions,

$P(b', q)$ and $P(b', c)$. For the converse, suppose that for some $b \in B$, we have that $P(b, q)$ and $P(b, c)$. Then, by the theorem assumptions, $b \models\!\!\!\pm q$ and $b \models\!\!\!\pm c$. Hence, by Theorem 2.6.1, c is q-reachable.

Claim 2. We prove the following: c is not q-inevitable $\Leftrightarrow (\exists\, b \in B : P(b, q)$ and $N(b, c))$.

First suppose that c is not q-inevitable. Then by Theorem 2.6.1, for some $b \in B$, $b \models\!\!\!\pm q$ and $b \models\!\!\!= c$. Choose any $b' \in F_s(b)$. Then $b' \models q$ and $c \notin say(b')$, and, since $b' \in F_s$, then $b' \models\!\!\!\pm q$ and $b' \models\!\!\!= c$. Hence, by the theorem assumptions, $P(b', q)$ and $N(b', c)$. For the converse, suppose that for some $b \in B$, we have that $P(b, q)$ and $N(b, c)$. Then, by the theorem assumptions, $b \models\!\!\!\pm q$ and $b \models\!\!\!= c$. Hence, by Theorem 2.6.1, c is not q-inevitable.

Claim 3. Suppose that c is q-reachable. Then the first half of the proof of Claim (1) shows that each binding $b' \in F_s(b)$ is a witness to the q-reachability of c. ■

Theorem 3.2.1 *For an acyclic production system s,*

$$b \in A_s \Leftrightarrow \quad (\forall\, x \in V_s \cup W_s \,:\, b(x) \neq \perp \Rightarrow b \models (indom_x \wedge precond_x))$$

Proof

By induction on the number of variables in s. Let Γ_s be the set of all bindings b such that: $\forall x \in V_s \cup W_s \,:\, b(x) \neq \perp \Rightarrow b \models (indom_x \wedge precond_x)$. Then the claim to be proven states the following: $A_s = \Gamma_s$.

For $n = 0$, we have $A_s = \Gamma_s = \{\{\}\}$. Suppose that the claim is true for systems with $n - 1$ variables, and suppose that the system s has n variables. Let

z be a variable in s that is a leaf in the restriction of the dependency graph to the variables. Let s_0 be the system that is obtained by removing z from s. By induction, $A_{s_0} = \Gamma_{s_0}$.

Claim 1: $\Gamma_s \subseteq A_s$

Choose $b \in \Gamma_s$. Let $b_0 = b - \{(z, b(z))\}$. Then $b_0 \in \Gamma_{s_0}$. Hence $b_0 \in A_{s_0}$.

Suppose first that $b \not\models \mathit{precond}_z$. Then, since $b \in \Gamma_s$, we have that $b(z) = \perp$. Hence $b = b_0 \in A_{s_0}$. Hence, $b \in A_s$.

Now suppose that $b \models \mathit{precond}_z$. Since $b \models \mathit{precond}_z$, and $z \notin FV(\mathit{precond}_z)$, then $b_0 \models \mathit{precond}_z$. First suppose that z is an input variable. Since $b \in \Gamma_s$, then $b(z) \in \mathit{dom}_z \cup \{\perp\}$. This, combined with the fact that $b_0 \models z$, implies that $b \in A_s$. Now suppose that z is a calculated variable. Then, since $b \in \Gamma_s$, then $b(z) = \perp$ or $b(z) = \mathit{val}(\mathit{term}_z, b)$. This, combined with the fact that $b_0 \models z$, implies that $b \in A_s$.

Claim 2: $A_s \subseteq \Gamma_s$

Choose $b \in A_s$. Let $b_0 = b - \{(z, b(z))\}$. Then $b_0 \in A_{s_0}$. Hence $b_0 \in \Gamma_{s_0}$.

Suppose first that $b \not\models \mathit{precond}_z$. Since $b \in A_s$, and $b \not\models \mathit{precond}_z$, then $b(z) = \perp$. Since $b_0 \in \Gamma_{s_0}$, $b \not\models \mathit{precond}_z$, and $b(z) = \perp$, then $b \in \Gamma_s$.

Now suppose that $b \models \mathit{precond}_z$. First suppose that z is an input variable. Since $b \in A_s$, and $b \models \mathit{precond}_z$, then $b(z) \in \mathit{dom}_z \cup \{\perp\}$. This, combined with the fact that $b_0 \in \Gamma_{s_0}$, implies that $b \in \Gamma_s$. Now suppose that z is a calculated variable. Since $b \in A_s$, and $b \models \mathit{precond}_z$, then $b(z) = \perp$ or $b(z) = \mathit{val}(\mathit{term}_z, b)$. This, combined with the fact that $b_0 \in \Gamma_{s_0}$, implies that $b \in \Gamma_s$. ■

Theorem 3.2.2 *For an acyclic production system s,*

$$b \in F_s \Leftrightarrow \quad ((\forall\, v \in V_s \;:\; \textsf{if } b \models \mathit{precond}_v \textsf{ then } b(v) \in \mathit{dom}_v \textsf{ else } b(v) = \bot) \;\&$$
$$(\forall\, w \in W_s \;:\; \textsf{if } b \models \mathit{pre}_w \textsf{ then } b(w) = \mathit{val}(\mathit{term}_w, b) \textsf{ else } b(w) = \bot))$$

The claim is equivalent to the statement that $b \in F_s$ iff the following four conditions hold:

1. $\forall\, v \in V_s \;:\; b \models \mathit{precond}_v \Rightarrow b(v) \in \mathit{dom}_v$
2. $\forall\, w \in W_s \;:\; b \models \mathit{precond}_w \Rightarrow b(w) = \mathit{val}(\mathit{term}_w, b)$
3. $\forall\, v \in V_s \;:\; b \not\models \mathit{precond}_v \Rightarrow b(v) = \bot$
4. $\forall\, w \in W_s \;:\; b \not\models \mathit{precond}_w \Rightarrow b(w) = \bot$

Proof ($\Leftarrow$)

For a binding b, suppose that (1), (2), (3) and (4) are true. To show that $b \in F_s$, we must show that $b \in A_s$, and that $\mathit{ask}(b) = \mathit{say}(b) = \{\}$.

To show that $b \in A_s$, the previous theorem shows that it is sufficient to prove that: $(\forall\, x \in V_s \cup W_s \;:\; b(x) \neq \bot \Rightarrow b \models (\mathit{indom}_x \wedge \mathit{precond}_x))$. First suppose that $x \in V_s$, and that $b(x) \neq \bot$. Then by (3), $b \models \mathit{precond}_x$. Hence, by (1), $b_x \in \mathit{dom}_x$. Thus $b \models (\mathit{indom}_x \wedge \mathit{precond}_x)$. Now suppose that $x \in W_s$, and that $b(x) \neq \bot$. Then by (4), $b \models \mathit{precond}_x$. Hence, by (2), $b(x) = \mathit{val}(\mathit{term}_x, b)$. Thus, since $b(x) \neq \bot$, then $b \models \mathit{indom}_x$. Hence, $b \models (\mathit{indom}_x \wedge \mathit{precond}_x)$. Therefore, $b \in A_s$.

Next, from (1) it follows that for all $v \in V_s \;:\; b \models \mathit{precond}_v \Rightarrow b(v) \neq \bot$. Hence $\mathit{ask}(b) \equiv \{v \in V_s \mid b \models \mathit{precond}_v \;\&\; b(v) = \bot\} = \{\}$.

Finally, claim that $calc(b) \equiv \{(w, z) : b \models precond_w \ \& \ z = val(term_w, b) \neq \perp \ \& \ b(w) = \perp\} = \{\}$. To prove that $calc(b) = \{\}$, we show that $b \models precond_w \Rightarrow w \notin vars(calc(b))$. Suppose that $b \models precond_w$. Then by (2), $b(w) = val(term_w, b)$. Hence, w cannot belong to $vars(calc(b))$—because if it did, then we would have that $\perp = b(w) \neq val(term_w, b)$. Thus, $vars(calc(b)) = \{\}$, and so $calc(b) = \{\}$. ■

Proof ($\Rightarrow$)

Suppose that $b \in F_s$, *i.e.*, that $b \in A_s$ and $ask(b) = calc(b) = \{\}$. We now prove that statements (1), (2), (3) and (4) are all true.

Since $b \in A_s$, then by the preceding theorem, for all $x \in V_s \cup W_s$: $b(x) \neq \perp \Rightarrow b \models (indom_x \wedge precond_x)$. Choose $x \in V_s \cup W_s$. Then $b \not\models precond_x \Rightarrow b(x) = \perp$. That proves statements (3) and (4).

To prove statement (1), choose $v \in V_s$. Since $ask(b) \equiv \{v \in V_s \mid b \models precond_v \ \& \ b(v) = \perp\} = \{\}$, it follows that $b \models precond_v \Rightarrow b(v) \neq \perp$. Furthermore, since b is attainable, then $b \models precond_v \Rightarrow b(v) \in dom_v$. Thus, statement (1) is proven.

To prove statement (2), choose $w \in W_s$, and suppose that $b \models precond_w$. Then, since $calc(b) \equiv \{(w, z) : b \models precond_w \ \& \ z = val(term_w, b) \neq \perp \ \& \ b(w) = \perp\} = \{\}$, it follows that $val(term_w, b) = \perp$ or $b(w) \neq \perp$. We conclude the proof by showing that this disjunction implies that $b(w) = val(term_w, b)$. First consider the case where $b(w) = \perp$. Then $val(term_w, b) = \perp$, and so $b(w) = val(term_w, b)$. Now suppose that $b(w) \neq \perp$. Then, since b is attainable, there is some attainable sub-binding $b_0 \subseteq b$ in which $b(w)$ was assigned: $b(w) = val(term_w, b_0) \neq \perp$. From this it follows that $val(term_w, b_0) = val(term_w, b)$, and so $b(w) = val(term_w, b)$. ■

We will use the following definition of the Height of an Expression in our inductive proofs—it will serve as the rank function. Informally, (1) subexpressions have smaller height than the expressions which contain them, and (2) a variable has greater height than the expressions upon which it depends, viz., the precondition and answer expressions. That the height is well-defined will follow from the acyclicity of the dependency graph.

Definition A.2 (Height of an Expression)

$$\begin{array}{lcl} height & : & Expr \to \mathbb{N} \\ height(op(e_1, e_2)) & = & 1 + max(height(e_1), height(e_2)) \\ height(const) & = & 1 \\ height(v) & = & 1 + height(precond_v) \\ height(w) & = & 1 + max(height(precond_w), height(term_w)) \end{array}$$

Theorem 3.2.3 *For an acyclic production system s, binding $b \in B_s$, formula f, and term t:*

- $b \models pos_s(f) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \models f)$
- $b \models neg_s(f) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \not\models f)$
- $b \models pos_s(t) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \models (t = t)$
- $b \models neg_s(t) \Rightarrow (\forall\, b' \in A_s(b) \,:\, b' \not\models (t = t)$

Proof

We proceed by induction on the height of f. Our induction hypothesis is that all four of the claims hold on all expressions of smaller height.

Pos: Conjunction

$b \models pos(f_1 \wedge f_2) \Rightarrow$

$b \models pos(f_1)$ and $b \models pos(f_2) \stackrel{\dagger}{\Rightarrow}$

$(\forall\, b' \in A(b) \,:\, b' \models f_1)$ and $(\forall\, b' \in A(b) \,:\, b' \models f_2) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \models (f_1 \wedge f_2))$

(†) by induction

Pos: Disjunction

$b \models pos(f_1 \vee f_2) \Rightarrow$

$b \models pos(f_1)$ or $b \models pos(f_2) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \models f_1)$ or $(\forall\, b' \in A(b) \,:\, b' \models f_2) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \models (f_1 \vee f_2))$

Pos: Negation

$b \models pos(\neg f) \Rightarrow$

$b \models neg(f) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \not\models f) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \models \neg f)$

Pos: Atomic

$b \models pos(p(t_1, \ldots, t_k)) \Rightarrow$

$b \models pos(t_1)\ \&\ldots\&\ b \models pos(t_k)\ \&\ b \models p(t_1, \ldots, t_k) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, val(t_1, b') \neq \bot\ \&\ldots\&\ val(t_k, b') \neq \bot)\ \&\ b \models p(t_1, \ldots, t_k) \stackrel{\dagger}{\Rightarrow}$

$val(t_1, core(b)) \neq \bot\ \&\ldots\&\ val(t_k, core(b)) \neq \bot\ \&\ b \models p(t_1, \ldots, t_k) \stackrel{\ddagger}{\Rightarrow}$

$val(t_1, b) = val(t_1, core(b))$ & ... & $val(t_k, b) = val(t_k, core(b))$ & $b \models p(t_1, \ldots, t_k)$

$\Rightarrow core(b) \models p(t_1, \ldots, t_k) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \models p(t_1, ..., t_k))$

(†) since $core(b) \in A(b)$

(‡) since $b \supseteq core(b)$

Pos: Function

$b \models pos(f(t_1, \ldots, t_k)) \Rightarrow$

$b \models pos(t_1)$ & ... & $b \models pos(t_k) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, val(t_1, b') \neq \perp$ & ... & $val(t_k, b') \neq \perp) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, val(f(t_1, \ldots, t_k), b') \neq \perp)$

Pos: Constant

$(\forall\, b' \in A(b) \,:\, val(c, b') = c \neq \perp)$

Pos: Input Variable

$b \models pos(v) \Rightarrow$

$b \models pos(precond_v)$ & $b \models indom_v \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \models precond_v)$ & $b \models indom_v \Rightarrow$

$core(b) \models precond_v$ & $b(v) \in dom_v \overset{\dagger}{\Rightarrow}$

$(core(b))(v) = b(v) \neq \perp \Rightarrow$

$(\forall\, b' \in A(b) \,:\, val(v, b') = b(v) \neq \perp)$

(†) since $core(b) \models precond_v$, and $b(v) \in dom_v$, the computation of $core(b)$ will

accept the value $b(v)$ as the value for $(core(b))(v)$

Pos: Derived Variable

$b \models pos(w) \Rightarrow$

$b \models pos(precond_w)$ & $b \models pos(term_w)$ & $b \models (w = term_w) \Rightarrow$

$((\forall\, b' \in A(b) : b' \models precond_w)$ & $(\forall\, b' \in A(b) : val(term_w, b') \neq \bot)$ &

$b(w) = val(term_w, b) \neq \bot)$

$\Rightarrow$

$(core(b) \models precond_w$ & $val(term_w, core(b)) \neq \bot$ &

$b(w) = val(term_w, b) \neq \bot)$

$\overset{\dagger}{\Rightarrow}$

$(core(b) \models precond_w$ & $val(term_w, core(b)) = val(term_w, b) = b(w) \neq \bot) \overset{\ddagger}{\Rightarrow}$

$(core(b))(w) = b(w) \neq \bot \Rightarrow$

$(\forall\, b' \in A(b) : val(w, b') \neq \bot)$

(†) since $val(term_w, core(b)) \neq \bot$, and $b \supseteq core(b)$, then $val(term_w, core(b)) = val(term_w, b)$

(‡) since $core(b) \models precond_w$, and $val(term_w, core(b)) = b(w) \neq \bot$, then the computation of $core(b)$ will accept the value $b(w)$ as the value for $(core(b))(w)$

Neg: Conjunction

$b \models neg(f_1 \wedge f_2) \Rightarrow$

$b \models neg(f_1)$ or $b \models neg(f_2) \Rightarrow$

$(\forall\, b' \in A(b) : b' \not\models f_1)$ or $(\forall\, b' \in A(b) : b' \not\models f_2) \Rightarrow$

$(\forall\, b' \in A(b) : b' \not\models (f_1 \wedge f_2))$

Neg: Disjunction

$b \models neg(f_1 \vee f_2) \Rightarrow$

$b \models neg(f_1)$ and $b \models neg(f_2) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \not\models f_1)$ and $(\forall\, b' \in A(b) \,:\, b' \not\models f_2) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \not\models (f_1 \vee f_2))$

Neg: Negation

$b \models neg(\neg f) \Rightarrow$

$b \models pos(f) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \models f) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \not\models \neg f)$

Neg: Atomic

$b \models neg(p(t_1, \ldots, t_k)) \Rightarrow$

$b \models neg(t_1)$ or $\ldots$ or $b \models neg(t_k)$ or $b \models pos(\overline{p}(t_1, \ldots, t_k)) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, val(t_1, b') = \bot)$ or $\ldots$ or $(\forall\, b' \in A(b) \,:\, val(t_k, b') = \bot)$ or

$(\forall\, b' \in A(b) \,:\, b' \models \overline{p}(t_1, \ldots, t_k))$

$\Rightarrow$

$(\forall\, b' \in A(b) \,:\, val(t_1, b') = \bot$ or $\ldots$ or $val(t_k, b') = \bot$ or $b' \models \overline{p}(t_1, \ldots, t_k)) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, b' \not\models p(t_1, \ldots, t_k))$

Neg: Function

$b \models neg(f(t_1, \ldots, t_k)) \Rightarrow$

$b \models neg(t_1)$ or $\ldots$ or $b \models neg(t_k) \Rightarrow$

$(\forall\, b' \in A(b) \,:\, val(t_1, b') = \bot)$ or $\ldots$ or $(\forall\, b' \in A(b) \,:\, val(t_k, b') = \bot) \Rightarrow$

$(\forall\, b' \in A(b) \;:\; val(t_1, b') = \bot \text{ or } \ldots \text{ or } val(t_k, b') = \bot) \Rightarrow$

$(\forall\, b' \in A(b) \;:\; val(f(t_1, \ldots, t_k), b') = \bot)$

Neg: Constant

$b \models neg(c) \Rightarrow$

$b \models false \Rightarrow$

everything

Neg: Input Variable

$b \models neg(v) \Rightarrow$

$b \models neg(precond_v) \Rightarrow$

$(\forall\, b' \in A(b) \;:\; b' \not\models precond_v) \stackrel{\dagger}{\Rightarrow}$

$(\forall\, b' \in A(b) \;:\; b'(v) = \bot) \Rightarrow$

$(\forall\, b' \in A(b) \;:\; val(v, b') = \bot)$

(†) Since b' is attainable

Neg: Derived Variable

$b \models neg(w) \Rightarrow$

$b \models neg(precond_w)$ or $b \models neg(term_w) \Rightarrow$

$(\forall\, b' \in A(b) \;:\; b' \not\models precond_w)$ or $(\forall\, b' \in A(b) \;:\; val(term_w, b') = \bot) \stackrel{\dagger}{\Rightarrow}$

$(\forall\, b' \in A(b) \;:\; b'(w) = \bot) \Rightarrow$

$(\forall\, b' \in A(b) \;:\; val(w, b') = \bot)$

(†) By Theorem 3.2.1, for attainable b', $b' \not\models precond_w \Rightarrow b'(w) = \bot$. Also, the

theorem implies that for attainable b', $val(term_w, b') = \perp \Rightarrow b'(w) = \perp$

■

Theorem 3.2.6 *For an acyclic production system s, and final binding $b \in F_s$,*

$b \models pos_s(f) \Leftrightarrow b \models f$

$b \models neg_s(f) \Leftrightarrow b \not\models f$

Proof ($\Rightarrow$)

Suppose $b \in F_s$. By Corollary 3.2.4, $b \models pos_s(f) \Rightarrow b \models^{\pm}_s f$, and by Proposition 2.4.2, $b \models^{\pm}_s f \Rightarrow b \models f$. Hence, $b \models pos_s(f) \Rightarrow b \models f$. Similarly, $b \models neg_s(f) \Rightarrow b \models^{=}_s f$, and $b \models^{=}_s f \Rightarrow b \not\models f$, and so $b \models neg_s(f) \Rightarrow b \not\models f$. ■

Proof ($\Leftarrow$)

We prove the following, more general proposition. For a binding $b \in F_s$, formula f, and term t

Claim:

- $b \models f \Rightarrow b \models pos_s(f)$
- $b \not\models f \Rightarrow b \models neg_s(f)$
- $b \models (t = t) \Rightarrow b \models pos_s(t)$
- $b \not\models (t = t) \Rightarrow b \models neg_s(t)$

The proof proceeds by induction on $height(e)$. Our induction hypothesis is that for all expressions of height less than k, both clauses of the Claim are true. We break the proof into cases based on the form of e.

Pos: Conjunction

$b \models f_1 \wedge f_2 \Rightarrow$

$b \models f_1$ and $b \models f_2 \overset{\dagger}{\Rightarrow}$

$b \models pos(f_1)$ and $b \models pos(f_2) \Rightarrow$

$b \models pos(f_1 \wedge f_2)$

($\dagger$) by induction

Pos: Disjunction

$b \models f_1 \vee f_2 \Rightarrow$

$b \models f_1$ or $b \models f_2 \Rightarrow$

$b \models pos(f_1)$ or $b \models pos(f_2) \Rightarrow$

$b \models pos(f_1 \vee f_2)$

Pos: Negation

$b \models \neg f \Rightarrow$

$b \not\models f \Rightarrow$

$b \models neg(f) \Rightarrow$

$b \models pos(\neg f)$

Pos: Atomic

$b \models p(t_1, \ldots, t_k) \Rightarrow$

$b \models (t_1 = t_1)\ \ \&\ \ \ldots\ \ \&\ \ b \models (t_k = t_k)\ \ \&\ \ b \models p(t_1, \ldots, t_k) \Rightarrow$

$b \models pos(t_1)\ \ \&\ \ \ldots\ \ \&\ \ b \models pos(t_k)\ \ \&\ \ b \models p(t_1, \ldots, t_k) \Rightarrow$

$b \models pos(p(t_1, \ldots, t_k))$

Pos: Function

$b \models f(t_1, \ldots, t_k) \Rightarrow$

$b \models (t_1 = t_1) \ \& \ \ldots \ \& \ b \models (t_k = t_k) \Rightarrow$

$b \models pos(t_1) \ \& \ \ldots \ \& \ b \models pos(t_k) \Rightarrow$

$b \models pos(f(t_1, \ldots, t_k))$

Pos: Constant

$b \models pos(c) \Leftrightarrow b \models true \Leftrightarrow true$

Pos: Input Variable

$b \models (v = v) \overset{\dagger}{\Rightarrow}$

$b \models (precond_v \wedge indom_v) \Rightarrow$

$b \models pos(precond_v) \ \& \ b \models indom_v \Rightarrow$

$b \models pos(v)$

($\dagger$) since $b \in A_s$

Pos: Derived Variable

$b \models (w = w) \Rightarrow$

$b \models (precond_w \wedge indom_w) \Rightarrow$

$b \models pos(precond_w) \ \& \ b \models term_w \ \& \ b \models indom_w \Rightarrow$

$b \models pos(precond_w) \ \& \ b \models pos(term_w) \ \& \ b \models indom_w \Rightarrow$

$b \models pos(w)$

Neg: Conjunction

$b \not\models f_1 \wedge f_2 \Rightarrow$

$b \not\models f_1$ or $b \not\models f_2 \overset{\dagger}{\Rightarrow}$

$b \models neg(f_1)$ or $b \models neg(f_2) \Rightarrow$

$b \models neg(f_1 \wedge f_2)$

($\dagger$) by induction

Neg: Disjunction

$b \not\models f_1 \vee f_2 \Rightarrow$

$b \not\models f_1$ and $b \not\models f_2$

$b \models neg(f_1)$ and $b \models neg(f_2) \Rightarrow$

$b \models neg(f_1 \vee f_2)$

Neg: Negation

$b \not\models \neg f \Rightarrow$

$b \models f \Rightarrow$

$b \models pos(f) \Rightarrow$

$b \models neg(\neg f)$

Neg: Atomic

$b \not\models p(t_1, \ldots, t_k) \Rightarrow$

$b \not\models (t_1 = t_1)$ or $\ldots$ or $b \not\models (t_k = t_k)$ or $b \models \overline{p}(t_1, \ldots, t_k) \Rightarrow$

$b \models neg(t_1)$ or $\ldots$ or $b \models neg(t_k)$ or $b \models pos(\overline{p}(t_1, \ldots, t_k)) \Rightarrow$

$b \models neg(p(t_1, \ldots, t_k))$

Neg: Function

$b \not\models f(t_1, \ldots, t_k) \Rightarrow$

$b \not\models (t_1 = t_1)$ or $\ldots$ or $b \not\models (t_k = t_k) \Rightarrow$

$b \models neg(t_1)$ or $\ldots$ or $b \models neg(t_k) \Rightarrow$

$b \models neg(f(t_1, \ldots, t_k))$

Neg: Constant

$b \not\models pos(c) \Rightarrow$

$b \not\models true \Rightarrow$

$false \Rightarrow$

everything

Neg: Input Variable

$b \not\models (v = v) \stackrel{\dagger}{\Rightarrow}$

$b \not\models precond_v \Rightarrow$

$b \models neg(precond_v) \Rightarrow$

$b \models neg(v)$

(†) since $b \in F_s$

Neg: Derived Variable

$b \not\models (w = w) \stackrel{\dagger}{\Rightarrow}$

$b \not\models precond_w$ or $b \not\models term_w \Rightarrow$

$b \models neg(precond_w)$ or $b \models neg(term_w) \Rightarrow$

$b \models neg(w)$

(†) since $b \in F_s$ ■

Theorem 3.2.8 *For acyclic production system s, variable $v \in V_s$, and formula q:* $choices_s(v,q) = \{b(v) \mid b \in B \ \& \ b \models (pos_s(q) \wedge pos_s(v))\}$

Proof

First suppose that $z \in choices_s(v,q)$. This means that $z \neq \bot$, and that $\exists b \in F_s$ *s.t.* $b \models q$ and $b(v) = z$. Since $b \in F_s$, and $b \models q$, then by Theorem 3.2.6, $b \models pos(q)$. Next, since $b \in A_s$, and $b(v) \neq \bot$, then by Theorem 3.2.2, $b \models (indom_v \wedge precond_v)$. Since $b \in F_s$, and $b \models precond_v$, then by Theorem 3.2.6, $b \models pos(precond_v)$. Hence, $b \models pos(precond_v) \wedge indom_v$, *i.e.*, $b \models pos(v)$. Thus, $b \models pos(q) \wedge pos(v)$. Therefore $z = b(v)$ belongs to the set on the right hand side of the claimed equation.

Now suppose that z belongs to the set on the right side of the claimed equation, *i.e.*, that for some binding b, we have that $b(v) = z$, and $b \models pos(q) \wedge pos(v)$. Since $b \models pos(q)$, then by Corollary 3.2.4, $b \models^{\pm} q$. Since $b \models pos(v)$, then by the same corollary, $b \models^{\pm}_{(} v = v)$. Choose $b' \in F_s(b)$, *i.e.*, let b' be some final extension of the core of b. Then, since $b \models^{\pm} q$, we have that $b' \models q$, and since $b \models^{\pm} (v = v)$, we have that $b' models (v = v)$, *i.e.*, that $b'(v) \neq \bot$. Therefore, $b'(v) \in choices_s(v,q)$.

Now, since $b \models pos(v)$, then by Theorem 3.2.3, for every attainable extension b' of the core of b, $b' \models (v = v)$, *i.e.*, $b'(v) \neq \bot$. Hence, $core(b)(v) \neq \bot$. Since $core(b) \neq \bot$, and $core(b) \subseteq b$ and $core(b) \subseteq b'$, then $core(b)(v) = b(v) = b'(v)$. Hence, $z = b(v) \in choices_s(v,q)$. ■

Proposition 3.5.1 *For monotone acyclic production systems, the determination of the q-reachability of a conclusion is NP-hard in the size of the system.*

Proof

We give a direct reduction from SAT. Let f be a conjunctive normal formula of propositional logic, and let f' be a representation of f in predicate logic.[2] Let $V = FV(f')$, and let s be a (trivial) system that has one question variable for each v in V, each with true precondition. Then,

$$f \text{ satisfiable} \Leftrightarrow f' \text{ satisfiable} \Leftrightarrow f' \text{ reachable under query } \mathit{true}$$

■

Proposition 3.5.2

$$\mathit{size}(\mathit{fg}(s,q)) = O(s+q)$$

Proof

The basis of this result, which states that the size of the formula graph is compact—linear—in the size of the input system plus the query formula, is the memoization that is performed with the maps *posmap* and *negmap* in the functions *pos* and *neg* on page 86. These memoization statements ensure that for each variable x,

[2]To make such a representation, for each propositional constant z_i, let x_i be an associated variable. Then replace each positive literal z_i in f with the atomic formula '$x_i = 1$', and each negative literal $\neg z_i$ with the atomic formula '$x_i = 0$'. Thus, for example, if f is $(z_1 \vee \neg z_2) \wedge (z_2 \vee \neg z_4)$, then f' would be $(x_1 = 1 \vee x_2 = 0) \wedge (z_2 = 1 \vee z_4 = 0)$.

regardless of how many times the variable x is used, there is just one node in the formula graph for $pos(x)$ and one node for $neg(x)$.

For an expression e, it is easy to see that a call to either $pos(e)$ or $neg(e)$ leads to a traversal of the structure of e, with one AND or OR node being constructed for each node in the abstract syntax tree for e. Thus, for the formula e, there is in the formula graph an embedded edge-reversed tree, isomorphic to e, that arises from the call to $pos(e)$, and another edge-reversed tree arising from the call to $neg(e)$. At the boundaries of these edge-reversed trees are either (1) constraints, which are roots in the overall formula graph, or (2) pointers to nodes for $pos(v)$ or $neg(v)$, for variables v occuring in e.

The construction of $pos(v)$ or $neg(v)$—which is only performed once—therefore results in the construction of of $size(precond_v) + size(term_v)$ logic nodes that may be "charged" to v. Similarly, for a conclusion formula f (or the query formula q), the calls to $pos(f)$ or $neg(f)$ result in the construction of $size(f)$ nodes that can be charged to f. It is not hard to see that every node in the entire formula graph will thus get charged to one of the variables, to one of the conclusions, or to the query formula. Hence the total size of the logic graph is linear in the size of the production system plus the query formula. ■

Proposition 3.5.4 *For a monotone formula f:*

$$f \cong \bigvee_{t \in dnf(f)} \bigwedge t$$

Proof

By induction on the structure of f. For a tuple t, *i.e.*, a set of constraints, define as a shorthand $t' \equiv \bigwedge t$.

Case 1: $f =$ **constraint** c

$\bigvee_{t \in dnf(f)} t' = \bigvee_{t \in \{\{c\}\}} t' = \{c\}' \cong f$

Case 2: $f = f_1 \vee f_2$

$\bigvee_{t \in dnf(f)} t' = \bigvee_{t \in dnf(f_1) \cup dnf(f_2)} t' = (\bigvee_{t \in dnf(f_1)} t') \vee (\bigvee_{t \in dnf(f_2)} t') \cong f_1 \vee f_2 = f$

Case 3: $f = f_1 \wedge f_2$

$\bigvee_{t \in dnf(f)} t' = \bigvee_{t \in \{t_1 \cup t_2 \mid t_1 \in dnf(f_1),\ t_2 \in dnf(f_2)\}} t' =$

$\bigvee_{t_1 \in dnf(f_1),\ t_2 \in dnf(f_2)} t'_1 \wedge t'_2 = (\bigvee_{t_1 \in dnf(f_1)} t'_1) \wedge (\bigvee_{t_1 \in dnf(f_1)} t'_1) = f_1 \wedge f_2 = f$ ■

Proposition 3.5.5 *For a formula f:*

$$|dnf(f)| \leq dc(f)$$

Proof

By induction on the structure of f.

Case 1: $f =$ **constraint** c

$|dnf(c)| = |\{\{c\}\}| = 1 = dc(c)$

Case 2: $f = f_1 \vee f_2$

$|dnf(f_1 \vee f_2)| = |dnf(f_1) \cup dnf(f_2)| \leq |dnf(f_1)| + |dnf(f_2)| \leq dc(f_1) + dc(f_2) = dc(f_1 \vee f_2)$

Case 3: $f = f_1 \wedge f_2$

$|dnf(f_1 \wedge f_2)| = |\{t_1 \cup t_2 \mid t_1 \in dnf(f_1),\ t_2 \in dnf(f_2)\}| \leq |dnf(f_1)| \times |dnf(f_2)| \leq dc(f_1) \times dc(f_2) = dc(f_1 \wedge f_2)$. ■

Proposition 3.5.6 *For a node x in the formula graph $fg(s, q)$ and a tuple $t \in dnf(x)$:*

$$|t| \leq weight(x) \leq size(s) + size(q)$$

Proof

Recall that $weight(x)$ is the sum of the sizes of the constraints that are ancestors of x in the formula graph.

Each tuple $t \in dnf(x)$ is a set of constraints, and each of these constraints is an ancestor of x in the formula graph. Hence $|t| \leq weight(x)$. Also $weight(x) \leq size(fg(s, q)) = O(s + q)$. ■

Proposition A.1 *For a term t:*

$$pos(t) = \bigwedge_{x \in FV(t)} pos(x)$$

Proof

By induction on the structure of t. ■

Proposition A.2 *For a conjunctive formula f:*

$$pos(f) = f \wedge \bigwedge_{x \in FV(f)} pos(x)$$

Proof

Case 1: $f = f_1 \wedge f_2$

$pos(f_1 \wedge f_2) =$

$pos(f_1) \wedge pos(f_2) =$

$(f_1 \wedge \bigwedge_{x \in FV(f_1)}) \wedge (f_2 \wedge \bigwedge_{x \in FV(f_2)}) =$

$f \wedge \bigwedge_{x \in FV(f)} pos(x)$

Case 2: $f =$ **atomic formula** $p(t_1, \ldots, t_k)$

$pos(p(t_1, ..., t_k)) =$

$p(t_1, ..., t_k) \wedge pos(t_1) \wedge \cdots \wedge pos(t_k) =$

$f \wedge (\bigwedge_{x \in FV(t_1)} pos(x)) \wedge \cdots \wedge (\bigwedge_{x \in FV(t_k)} pos(x)) =$

$f \wedge \bigwedge_{x \in FV(f)} pos(x)$ ■

Proposition A.3 *In a conjunctive system s, for a variable x:*

$$pos_s(x) = precond_x \wedge indom_x \wedge \bigwedge_{y \in parents(x)} pos_s(y)$$

Proof

$pos(x) =$

$pos(precond_x) \wedge pos(term_x) \wedge indom_x =$

$precond_x \wedge (\bigwedge_{y \in FV(precond_x)} pos(y)) \wedge (\bigwedge_{y \in FV(term_x)} pos(y)) \wedge indom_x =$

$precond_x \wedge indom_x \wedge \bigwedge_{y \in parents(x)} pos(y))$ ■

Proposition 3.5.7 *For a conjunctive system s and a variable x:*

$$pos_s(x) \;=\; \bigwedge_{y \in ancestors(x)} precond_y \wedge indom_y$$

Proof

Corollary to the previous proposition. ■

Corollary A.4 *For a conjunctive system s and a disjunction-free expression e,*

$$dc(pos_s(e)) = 1$$

Proof

By induction on the height of the expression e.

Case 1: $e =$ constraint c

$dc(pos(c)) = dc(c \wedge \bigwedge_{v \in FV(c)} pos(v)) = \prod_{v \in FV(c)} dc(pos(v)) \stackrel{\dagger}{=} 1$

$dc(pos(f_1 \wedge f_2)) = dc(pos(f_1)) \times dc(pos(f_2)) = 1$

Case 3: $e =$ term t

$dc(pos(t)) = dc(\bigwedge_{v \in FV(t)} pos(v)) = \prod_{v \in FV(t)} dc(pos(v)) = 1$

(†) by induction ■

Corollary 3.5.8 *For a conjunctive system s and disjunction-free expression e,*

$$|dnf(pos_s(e))| \leq 1$$

Proof

By the preceding propositions, we have that $|dnf(pos_s(e))| \leq dc(pos_s(e)) = 1$. ■

Proposition 3.5.9 *For a conjunctive system s and a monotone formula q,*

$$dc(pos_s(q)) = dc(q)$$

Proof

Case 1: $q = q_1 \wedge q_2$

$dc(pos(q_1 \wedge q_2)) = dc(pos(q_1) \wedge pos(q_2)) =$

$dc(pos(q_1)) \times dc(pos(q_2)) = dc(q_1) \times dc(q_2) = dc(q_1 \wedge q_2)$

Case 2: $q = q_1 \vee q_2$

$dc(pos(q_1 \vee q_2)) = dc(pos(q_1)) + dc(pos(q_2)) = dc(q_1) + dc(q_2) = dc(q_1 \vee q_2)$

Case 3: $q =$ **constraint** c

$dc(pos(c)) = dc(c \wedge \bigwedge_{v \in ancestors(FV(c))} precond_v \wedge indom_v) =$

$\prod_{v \in ancestors(FV(c)} dc(precond_v) = 1$ ■

Corollary 3.5.10 *For a conjunctive system s and a monotone formula q,*

$$|dnf(pos_s(q))| \leq dc(q)$$

Proposition A.5 *For a conjunctive system s and formula q, the data complexity, measured in units of constraints, of querying the q-reachability of all the conclusions in s is $O(n \times w)$, where $n = size(s)$ and w is the maximum weight of a node in the formula graph. Since w is $O(n)$, this complexity is $O(n^2)$.*

Proof

For reachability queries, the formulas that must be solved are all the $pos(q) \wedge pos(y)$, where y is either a variable or the precondition of a conclusion. This evaluation can be performed in the following three phases: (1) evaluation of all nodes $pos(x)$, for variables x, (2) evaluation of $pos(q)$, and (3) evaluation of all $pos(q) \wedge pos(y)$. We now show that the cost of each of these phases is $O(n \times w)$.

Phase 1: Evaluation of the $pos(x)$ formulas

For variables x, we can perform the evaluation of the $pos(x)$ formulas bottom-up, starting at the roots and proceeding towards the leaves. By working this way, when $pos(x)$ is being evaluated, then, for each parent p of x, the solutions to $pos(p)$ are immediately available. Let $|x|$ be the sum of the sizes of the expressions for x, *i.e.*, $|x| \equiv |precond_x| + |term_x| + |indom_x|$.

By Proposition A.3, $pos(x) = precond_x \wedge indom_x \wedge \bigwedge_{p \in parents(x)} pos(p)$. Furthermore, by Proposition 3.5.7, since the system is conjunctive, then there is just one tuple in $dnf(pos(p))$, and, like all tuples in the system, its size is bounded by w. Hence, the size of the formulas that will get passed to the conjunctive solver is bounded by $|precond_x| + |indom_x| + |parents(x)| \cdot w$. Since each parent p of x is referenced by an instance of 'p' in either $precond_x$ or $term_x$, we have that $parents(x) \leq |x|$. Hence, the incremental cost of evaluating $pos(x)$ is bounded by $|x| + |x| \cdot w = |x| \cdot (w + 1)$. Summing over all variables x, the total cost is therefore $O(n \times w)$.

Phase 2: Evaluation of $pos(q)$

We do this evaluation after all of the tuples for $pos(x)$ have been computed. Now, it is easy to see that the formula $pos(q)$ can be obtained from the formula q, by replacing each constraint c with $c \wedge \bigwedge_{x \in FV(c)} pos(x)$. This is reflected by the fact that the logic graph, there will be a "co-tree" (an edge-reversed tree) of AND and OR nodes, which mirrors the structure of the abstract syntax tree for q. In this co-tree, there is one root r_c for each constraint c in q, and for each variable x in $FV(c)$, there is an edge from the node for $pos(x)$ to the node r_c. (So r_c is not a root in the whole graph, just a root locally, in relation to the co-tree.)

By the results of Phase 1, all of the solutions to the nodes $pos(x)$ have been evaluated. Hence, the incremental cost of evaluating $pos(q)$ is just the cost of evaluating the nodes in the co-tree that mirrors the structure of q. There are $|q|$ such nodes. In Corollary 3.5.10, we showed that the number of tuples in $pos(q)$ is bounded by $dc(q)$. Now, since $pos(q)$ is at the leaf of the co-tree, and since the disjunctive complexity of the subformulas of $pos(q)$ cannot exceed the disjunctive complexity of $pos(q)$, it follows that the aggregate number of tuples for all of the nodes in the co-tree is bounded by $|q| \cdot dc(q) \equiv c_q$. This value c_q is a constant that is independent of the size of the input production system. Since the size of each tuple is bounded by w, the total size of the constraints passed to the solver is bounded by $c_q \cdot w$. Hence the complexity is $O(w)$.

Phase 3: Evaluation of the $pos(q) \wedge pos(y)$ formulas

To determine the q-reachability of the node y, the tuple for $pos(y)$—there is only one tuple, since the system is conjunctive—must be conjoined with each of the

tuples in $dnf(pos(q))$. In Corollary 3.5.10, we showed that there are at most $dc(q)$ of these tuples. Hence, to solve $pos(y)$, the size of the constraints transferred to the solver is bounded by $(w + w) \cdot dc(q) = O(w)$. Thus, the total complexity is $O(n \times w)$. ■

Lemma A.6 *For a conjunctive system s, variable x, and node y:*

$$x \in ancestors(y) \Rightarrow (neg_s(x) \models neg_s(y))$$

Proof

By induction on the distance from x to y, defined as the maximum length of a path from x to y. For the base case, where the distance is zero, y is the same as x, and we have that $neg(x) \models neg(x)$. For the induction step, suppose that x is a proper ancestor of y. We may write $neg(y) = neg(precond_y) \vee neg(term_y) = ((\bigvee_{z \in FV(precond_y)} neg(z)) \vee \text{other stuff }) \vee \bigvee_{z \in FV(term_y)} neg(z)$.

Since x is a proper ancestor of y, then for some $z \in FV(precond_y) \cup FV(term_y)$, we have that x is an ancestor of z. So, by induction, $neg(x) \models neg(z)$. Hence, $neg(x) \models neg(y)$. ■

Lemma A.7 *Recall that for a formula f, the notation f^- designates the opposite formula. Then for a node n with conjunctive precondition $c_1 \wedge \cdots \wedge c_k$:*

$$neg(n) = \left(\bigvee_{v \in parents(n)} neg(v) \right) \vee \bigvee_{i=1,\ldots,k} \left(c_i^- \wedge \bigwedge_{v \in FV(c_i)} pos(v) \right)$$

Proof

Follows by a simple induction on the number of conjuncts k. For $k = 1$, the lemma just restates the definition of $neg(n)$. The induction step uses the fact that $neg(n) = neg(c_1 \wedge \cdots \wedge c_k) = neg(c_1 \wedge \cdots \wedge c_{k-1}) \vee neg(c_k)$. ■

Proposition A.8 *For a conjunctive system s and formula q, the data complexity in units of constraints of querying the q-inevitability of all the conclusions in s is $O(n \times w)$, where $n = size(s)$ and w is the maximum weight of a node in the formula graph. Since w is $O(n)$, this complexity is $O(n^2)$.*

Proof

For inevitability queries, the formulas that must be solved are all the $pos(q) \wedge neg(y)$, where y is either a variable or the precondition of a conclusion. For this evaluation, we make use of the following results that were computed during the evaluation of the reachability queries: (1) the solutions to $pos(q)$, and (2) the solutions to the $pos(x)$, for the variables x. In Proposition A.5, we showed that these results can be computed with $O(n \times w)$ constraint-complexity.

We evaluate the formulas $pos(q) \wedge neg(y)$ by a bottom-up traversal of the nodes y in the dependency graph, which we modify by the following optimization: whenver $pos(q) \wedge neg(y)$ is found to be satisfiable, then for all descendents y' of y, $pos(q) \wedge neg(y')$ is immediately marked as satisfiable. Then, for such y, the constraints for $pos(q) \wedge neg(y')$ never need to be evaluated.

This optimization is justified as follows. Suppose that a binding b satisfies $pos(q) \wedge neg(y)$. Then $b \models neg(y)$. Now, by Lemma A.6, which showed that

$neg(y) \models neg(y')$, it follows that $b \models neg(y')$. Hence $b \models pos(q) \wedge neg(y')$.

Furthermore, we can arrange this optimization so that only $O(n)$ time gets spent traversing and marking the nodes $pos(q) \wedge neg(y)$. Initially, all of these nodes get marked as unsatisfiable. From that point on, the invariant is maintained that if a node $pos(q) \wedge neg(y)$ is ever marked as satisfiable, then for all descendants y' of y', $pos(q) \wedge neg(y')$ will be marked as satisfiable. Hence, when a node $pos(q) \wedge neg(y)$ is newly found to be satisfiable, then the traversal of the descendents y' can stop whenever a node $pos(q) \wedge neg(y')$ is encountered that has already been marked as satisfiable by a previous traversal.

Now, assuming this optimization is in place, let us consider the cost of determining the satisfiability of $pos(q) \wedge neg(y)$, where y has conjunctive precondition $c_1 \wedge \cdots \wedge c_k$. By Lemma A.7, $pos(q) \wedge neg(y) =$

$$pos(q) \wedge \left(\left(\bigvee_{v \in parents(y)} neg(v) \right) \vee \bigvee_{i=1,\ldots,k} \left(c_i^- \wedge \bigwedge_{v \in FV(c_i)} pos(v) \right) \right)$$

Now, if for some parent v of y, we had that $pos(q) \wedge neg(v)$ were satisfiable, then, by the marking optimization that we just described, $pos(q) \wedge neg(y)$ would have been marked as satisfiable. Hence, if the formula $pos(q) \wedge neg(y)$ is actually being evaluated, then it must be the case that for all parents v of y, that $pos(q) \wedge neg(v)$ is unsatisfiable. That being so, it follows that

$$pos(q) \wedge neg(y) = pos(q) \wedge \bigvee_{i=1,\ldots,k} \left(c_i^- \wedge \bigwedge_{v \in FV(c_i)} pos(v) \right)$$

Now fixed a tuple t in the solutions to $pos(q)$, and let us bound the number of constraints required to determine the satisfiability of $t \wedge neg(y)$. Let t_i be the ith conjunction on the right-hand-side disjunction: $t_i = c_i^- \wedge \bigwedge_{v \in FV(c_i)} pos(v)$. Since

each conjunct in t_i arises from a free variable in c_i, the number of conjuncts in t_i is bounded by $|c_i|$. Hence, $|t_i| \leq |c_i| + |c_i| \cdot w = |c_i| \cdot (1 + w)$, where w is the maximum size of a tuple in the formula graph. Summing over all the tuples t_i, for $i = 1, \ldots, k$, we get $|precond_i| \cdot (1 + w)$ as a bound. Now, each of these tuples must get conjoined with t, and hence the cost associated with $t \wedge neg(y)$ is $|precond_i| \cdot (w + 1 + w)$. By Corollary 3.5.10, the number of tuples in $pos(q)$ is bounded by $dc(q)$, which is a constant that is independent of the size of the input production system. Summing over all nodes y, we then get a constraint complexity that is $O(n \times w)$. ■

Theorem 3.5.11 *For a conjunctive system s and formula q, the data complexity in units of constraints of querying the q-reachability and q-inevitability of all the conclusions in s is $O(n \times w)$, where $n = size(s)$ and w is the maximum weight of a node in the formula graph. Since w is $O(n)$, this complexity is $O(n^2)$.*

Proof

This follows as a corollary to the Propositions A.5, A.8, which we have shown in this appendix. ■

Lemma A.9 *For a monotone formula f:*

$$b \models pos(f) \Rightarrow b \models f$$

Proof

By induction on the structure of f.

Case 1: $f = f_1 \wedge f_2$

$b \models pos(f_1 \wedge f_2) \Rightarrow b \models pos(f_1)$ & $b \models pos(f_2) \Rightarrow b \models f_1 \wedge f_2$

Case 2: $f = f_1 \vee f_2$

$b \models pos(f_1 \vee f_2) \Rightarrow b \models pos(f_1)$ or $b \models pos(f_2) \Rightarrow b \models f_1 \vee f_2$

Case 3: $f = atomic$

$b \models pos(atomic) \Rightarrow (b \models atomic \wedge \bigwedge_{v \in FV(atomic)} pos(v) \Rightarrow b \models atomic$ ■

Lemma A.10 *For a monotone acyclic production system s, an attainable binding $b \in A_s$, and a monotone formula f:*

- $b \models f \Rightarrow b \models pos_s(f)$
- $b \models (t = t) \Rightarrow b \models pos_s(t)$

Proof

In the ($\Leftarrow$) part of the proof of Theorem 3.2.6, on page 165, we proved that for final bindings $b \in F_s$, a stronger conclusion holds than what we claim here, viz., the stronger conclusion that includes the statements that $b \not\models f \Rightarrow b \models neg(f)$, and that $b \not\models (t = t) \Rightarrow b \models neg(t)$. Now, for this lemma, we have a weaker hypothesis, viz., that $b \in A_s$. Also, we have the restriction that all formulas are monotone. Since all formulas are monotone, our claim here is a statement about *only* the *pos* formulas. Inspecting the proof that starts on page 165, we see that the only place that the assumption $b \in F_s$ is made is in the analysis of the $neg(x)$ formulas, for variables x; in all the other cases, only the assumption that $b \in A_s$ is used. Hence the restriction of that proof to the *pos* cases proves the present lemma. ■

Proposition 3.6.1 *For an acyclic production system s with monotone preconditions, and an attainable binding $b \in A_s$, the formulas $(\wedge b)$ and $pos_s(\wedge b)$ are logically equivalent.*

Proof

Since $(\wedge b)$ is a monotone formula, then by Lemma A.9, $pos(\wedge b) \models (\wedge b)$. Claim now that $(\wedge b) \models pos(\wedge b) = pos(v_1 = x_1) \wedge \cdots \wedge pos(v_k = x_k) = (\bigwedge_{v \in vars(b)} pos(v)) \wedge (\wedge b)$. Since $(\wedge b) \models (\wedge b)$, this claim reduces to the claim that $(\wedge b) \models \bigwedge_{v \in vars(b)} pos(v)$. To show this, suppose that for some binding b', $b' \models (\wedge b)$. This means that $b' \supseteq b$. Choose a $v \in vars(b)$. Then $b \models (v = v)$. Hence, by Lemma A.10, $b \models pos(v)$. Hence, since $pos(v)$ is a monotone formula, $b' \models pos(v)$. ■

Proposition 3.6.3 *For a monotone acyclic production system s, let $b \in A_s$ be an attainable binding. Let q be a formula. Let q' be the extended query formula $q \wedge (\wedge b)$, and suppose that $pos(q')$ is satisfiable. Let n be a node in s that is not a descendent of any of the variables in $ancestors(FV(q)) \cup vars(b)$. Then: $status(n, q') = status(n, true)$.*

Proof

Since $status(n, f)$ has three components, this claim breaks down into three subclaims.

Claim 1: $pos(q') \wedge pos(n)$ **satisfiable** $\Leftrightarrow$ $pos(n) \wedge pos(true)$ **satisfiable**

I.e., $pos(q \wedge (\wedge b)) \wedge pos(n)$ satisfiable $\Leftrightarrow$ $pos(n)$ satisfiable. By Proposition 3.6.1,

this is equivalent to the claim that:

$$pos(q) \wedge (\wedge b) \wedge pos(n) \text{ satisfiable} \Leftrightarrow pos(n) \text{ satisfiable}$$

One direction of this implication is immediate: ($\Rightarrow$).

Now, one of our initial assumptions states that

$$ancestors(n) \cap (FV(q) \cup vars(b)) = \{\}$$

Then, since $FV(pos(n)) \subseteq ancestors(n)$, and $FV(pos(q)) \subseteq ancestors(FV(q))$, it follows that $FV(pos(n)) \cap FV(pos(q) \wedge (\wedge b)) = \{\}$, *i.e.*, $FV(pos(n)) \cap FV(pos(q')) = \{\}$. Now, by our initial assumptions, $pos(q')$ is satisfiable. Hence, if $pos(n)$ is satisfiable, then $pos(q') \wedge pos(n)$ is also satiafiable.

Claim 2: $pos(q') \wedge neg(n)$ **satisfiable** $\Leftrightarrow$ $neg(n) \wedge pos(true)$ **satisfiable**

The proof is obtained from the proof of claim (1) by replacing '$pos(n)$' with '$neg(n)$'.

Claim 3: $project(\{n\},\ pos(q') \wedge pos(n)) = project(\{n\},\ pos(n))$

This follows from the assumption that $pos(q')$ is satisfiable, and the fact, which we showed in the proof of claim (1), that $FV(pos(q')) \cap FV(pos(n)) = \{\}$. ■

Proposition 3.6.4 *For an acyclic production system s, let $b \in A_s$ be an attainable binding. Let q be a formula. Let v be a variable in $ask(b)$, and assume that v is not an ancestor of any variable in q. Let x be a value in dom_v, and let b' be the binding $b \cup \{(v, x)\}$. Then:*

$$n \notin descendents(v) \Rightarrow status(n, q \wedge (\wedge b')) = status(n, q \wedge (\wedge b))$$

Proof

Since $status(n, f)$ has three components, this claim breaks down into three sub-claims.

Claim 1: $pos(q \wedge (\wedge b')) \wedge pos(n)$ **satisfiable** $\Leftrightarrow pos(q \wedge (\wedge b)) \wedge pos(n)$ **satisfiable**
By Proposition 3.6.1, this is equivalent to the claim that:

$$pos(q) \wedge (\wedge b) \wedge (v = x) \wedge pos(n) \text{ satisfiable } \Leftrightarrow pos(q) \wedge (\wedge b) \wedge pos(n) \text{ satisfiable}$$

One direction is immediate: ($\Rightarrow$). Now we prove the converse by showing that:

$$v \notin FV(pos(q) \wedge (\wedge b) \wedge pos(n))$$

First, since $v \in ask(b)$, then $v \notin vars(b)$, and so $v \notin FV(\wedge b)$. Second, since $FV(pos(q)) \subseteq ancestors(FV(q))$, and, by our assumptions, $v \notin ancestors(FV(q))$, then $v \notin FV(pos(q))$. Finally, since $n \notin descendants(v)$, then $v \notin ancestors(n)$, and $FV(pos(n)) \subseteq ancestors(n)$, then $v \notin FV(pos(n))$.

Claim 2: $pos(q \wedge (\wedge b')) \wedge neg(n)$ **satisfiable** $\Leftrightarrow pos(q \wedge (\wedge b)) \wedge neg(n)$ **satisfiable**
The proof is obtained from the proof of claim (1) by replacing '$pos(n)$' with '$neg(n)$'.

Claim 3: $project(\{n\},\ pos(q \wedge (\wedge b')) \wedge pos(n)) = project(\{n\},\ pos(q \wedge (\wedge b)) \wedge pos(n))$
By Proposition 3.6.1, this is equivalent to the claim that:

$$proj(n,\ pos(q) \wedge (\wedge b) \wedge (v = x) \wedge pos(n)) = proj(n,\ pos(q) \wedge (\wedge b) \wedge pos(n))$$

This follows from the fact, which we showed in the proof of claim (1), that:

$$v \notin FV(pos(q) \wedge (\wedge b) \wedge pos(n))$$

■

Lemma A.11 *For a monotone acyclic production system s and conjunctive formula f, suppose that f has a single free variable p. Then:*

$$pos_s(f) = pos_s(p) \wedge f$$

Proof

By induction on the structure of f. Suppose that $FV(f) = \{p\}$.

First, if f is an atomic formula, then $pos(f)$ is by definition $pos(p) \wedge f$, and so the claim holds. For the remaining case, suppose that $f = f_1 \wedge f_2$. Now, one of the subformulas f_1, f_2 must have p as a free variable. Let us say that $FV(f_1) = \{p\}$. Then by induction, $pos(f_1) = pos(p) \wedge f_1$.

Now suppose that also $FV(f_2) = \{p\}$. Then by induction, also $pos(f_2) \wedge f_2$. Hence $pos(f) = pos(f_1) \wedge pos(f_2) = pos(p) \wedge f_1 \wedge f_2 = pos(p) \wedge f$. Now suppose that $FV(f_2) \neq \{p\}$. Since $FV(f_1 \wedge f_2) = \{p\}$, the only possibility then is that $FV(f_2) = \{\}$, *i.e.*, f_2 is either constant true or constant false. If f_2 is true, then we have: If f_2 is false, then we have: $pos(f) = pos(f_1 \wedge false) = false = pos(p) \wedge false = pos(p) \wedge f$. ■

Proposition 3.7.2 *For an acyclic production system s, let v be a variable in V_s, and c a conclusion in C_s. Suppose that $precond_v$ and $precond_c$ are conjunctive formulas. Then:*

1. *If v has only one parent p, then $pos_s(v) = pos_s(p) \wedge precond_v \wedge indom_v$*
2. *If c has only one parent p, then $pos_s(c) = pos_s(p) \wedge precond_c$*

Proof

By a direct application of the previous lemma:

$pos(c) = pos(precond_c) = pos(p) \wedge precond_c$

$pos(v) = pos(precond_v) \wedge indom_v = pos(p) \wedge precond_v \wedge indom_v.$ ■

Proposition 3.7.3 *The tree-complexity, measured in units of constraints, of the optimized algorithm is* $O(n)$.

Proof

Let C be a class of tree equivalent-systems. Recall that this means that for all $x, y \in C$, that $skel(x) = skel(y) \equiv skel(C)$, where the skeleton of a system is defined as the fragment of a system which is obtained by removing all of the tree nodes. Fix a query formula q. Let $s \in C$ be an arbitrary member of C.

The optimized algorithm has two modes of operation: (1) for the non-tree nodes, the normal, DNF-based bottom-up evaluation, and (2) for the tree nodes, the recursive routine, which performs temporary destructive updates to a global constraint store. In this proof, we will show that the constraint-complexity of the DNF-evaluation is $O(w)$, and the constraint-complexity of processing the tree nodes is $O(n)$. Since $w \leq n$, that will prove that the overall constraint complexity is $O(n)$.

Our first claim will show that the number of tuples in $dnf(x)$, for nodes $x \in s$, is $O(1)$.

Claim 1: $dc(pos_s(x)) = O(1)$**, for all nodes** $x \in s$

In order to prove this, we first define a system s' by removing all the tree nodes from s, and, for the non-tree nodes, replacing all references to tree nodes with references

to the local roots of these trees. Specifically, for each node $x \in s$, define:

x is a root, or x has multiple parents

$base(x) = base(p)$, if $parents(x) = \{p\}$.

$base(x) = x$, if $parents(x) \neq 1|$

Then define $nodes(s') = \{x \in nodes(s) \mid |parents(x)| \neq 1\}$. Also, for each node $y \in s'$, let f be the precondition of y in s and t be the answer term for y in s; then define the precondition of y in s' to be the formula that is obtained by replacing each variable v in f by the variable $base(v)$, and define the answer term to be the term that is obtained by replacing each variable v in t by the variable $base(v)$.

Now, since each node in s' is one of the non-tree nodes in s, then the nodes of s' can be put into one-to-one correspondence with the nodes of $skel(C)$. Hence, the number of nodes in s', and the size of s', is $O(1)$. Let k be a constant that bounds the size of s'. Since there are only a finite number of systems that can be constructed with under k symbols, there are only a finite number of possible formulas $pos(x)$, for nodes $x \in s'$. Hence, $dc(pos_{s'}(x)) = O(1)$.

Now we claim that $dc(pos_s(x)) = dc(pos_{s'}(base(x))$—and this will serve to prove our claim that $dc(pos_s(x)) = O(1)$. We prove this claim by induction on the depth of x in the dependency graph for s.

Case 1: x is a root in s

Then $base(x) = x$, and also $base(x)$ is a root in s'; hence, the disjunctive complexities $dc(pos_s(x))$ and $dc(pos_{s'}(base(x)))$ are equal.

Case 2: x is a tree-node

Then $base(x) = base(parent(x))$.

By induction, $dc(pos_s(parent(x))) = dc(pos_{s'}(base(parent(x))))$. Also, since x has a conjunctive precondition, it is easy to see that $dc(pos_s(x)) = dc(pos_s(parent(x)))$. Hence, $dc(pos_s(x)) = dc(pos_{s'}(base(x))$.

Case 3: x has multiple parents

Then x is also a node in s'.

For an expression e, let $varlist(e)$ be a listing of all of the occurrences of variables in e, taken in a left-to-right traversal of e. Let $dclist_s(e)$ be the list: $[dc(pos_s(v)) \mid v \in varlist(e)]$.

Now, it is not hard to see that $dc(pos_s(x))$ is a function of: $precond_x$, $term_x$, $dclist_s(precond_x)$, and $dclist_s(term_x)$.

By induction, for each $p \in parents(x)$: $dc(pos_s(p)) = dc(pos_{s'}(base(p)))$. Then, since the expressions for x in s' were obtained from $precond_x$ and $term_x$ by replacing each occurrence of p by the variable $base(p)$, we obtain the result that $dclist_s(precond_x) = dclist_{s'}(precond_x)$ and $dclist_s(term_x) = dclist_{s'}(term_x)$.

Conclusion: $dc(pos_s(x)) = dc(pos_{s'}(x))$.

Since we have shown that $dc(pos_{s'}(x)) = O(1)$, it follows that $dc(pos_s(x)) = O(1)$.

Conclusion: as s ranges over all of the systems in C, and x ranges over all the nodes in s, the number of tuples in $pos_s(x)$ is $O(1)$.

That concludes the proof of claim (1).

Claim 2: $dc(pos_s(q)) = O(1)$

For each variable $v \in FV(q)$, claim (1) has shown us that $dc(pos_s(v)) = O(1)$. Since q is a fixed query formula, it is not hard to see that this implies that $dc(pos_s(q)) = O(1)$.

Claim 3: The work to process the non-tree nodes in s is $O(w)$

Since there is a one-to-one correspondence between the non-tree nodes and the nodes in $skel(C)$, the number of non-tree nodes is $O(1)$. Also, as we showed in claim (1), the number of tuples in $pos(x)$, for non-tree nodes, is $O(1)$. Hence the total number of tuples for all of the *pos* formulas for the non-tree nodes is $O(1)$. Since each tuple results either from the conjunction of two tuples, or is a primitive constraint at a root in the formula graph, then the number of tuples that were passed to the solver to produce these $O(1)$ tuples is also $O(1)$. Hence, in order to compute all the solutions to these $pos(x)$, $O(w)$ constraints were passed to the solver.

After the tuples for the $pos(x)$ formulas are computed, they must get conjoined with the tuples for $pos(q)$. By claim (2), the number of tuples in $pos(q)$ is $O(1)$. Since there are $O(1)$ tuples in all of the $pos(x)$ formulas combined, then $O(1)$ tuples, each of size bounded by w+w, get passed to the solver in the computation of $pos(q) \wedge pos(x)$. Hence $O(w)$ constraints get passed to the solver.

Claim 4: The work to process the tree-nodes in s is $O(n)$

Let T be the set of maximal local trees in s (where a local tree is maximal if it is not a proper subset of another local tree). Since these trees are maximal, the

root of each tree in T is a non-tree node, and hence is a node in $skel(C)$. This association from trees in T to nodes in $skel(C)$ is one-to-one. Hence, the number of maximal local trees is $O(1)$.

Now, the modified bottom-up evaluation makes one call to the routine *query_tree* for each of the maximal local subtrees.[3] Hence, *query_tree* is called $O(1)$ times. When this routine is called for the local root node r, it initializes a mutable constraint store to hold the solutions to $pos(q) \wedge pos(r)$. In claims (1) and (2), we have shown that the number of tuples in $pos(q)$ and $pos(r)$ is $O(1)$. Hence, $O(1)$ tuples of size $w + w$, *i.e.*, $O(w)$ constraints, must get transferred to initialize the global constraint stores.

Finally, the work to process each tree-node x involves conjoining the constraints for x, viz., $precond_x$ and $indom_x$, into the mutable tuples in the global constraint store. We have already shown that there are $O(1)$ tuples in the global store. Hence, the constraints for x have to be transferred to only $O(1)$ tuples, and so the complexity for processing x is $O(|x|)$, where $|x|$ is the sum of the sizes of $precond_x$ and $indom_x$. Summing over all the tree-nodes, we see that the constraint-complexity of processing all the tree-nodes is $O(n)$. ■

Proposition 3.7.4 *We can provide methods for the UMO constraint domain, so that, for classes of tree-equivalent systems where the individual rules are of bounded size, the time-complexity of reachability queries is* $O(n)$.

[3]Recall that this routine is called with the root of the local tree as argument, and returns after the whole tree has been queried.

Proof

For this proof, we use the notations for the class C of tree-equivalent systems, and $skel(C)$, for the skeleton of non-tree nodes for this class, as defined in the preceding proof.

We now describe an organization for the UMO tuples that will support the claimed $O(n)$ tree-complexity. It is based upon the representation, which we presented in Section 3.6.1, on page 107, in which the UMO tuple t takes the form of a mapping from variables to ordered lists of disjoint intervals. Recall that ordered list of intervals $t(x)$ is taken to represent a set of possible values for x, and that the meaning of the whole tuple is the Cartesian product of these sets.

For the tuples that we use for the tree optimization, which should support a fast method of incrementally conjoining a constraint into a tuple, we will implement these maps as *arrays* of interval-lists. Accordingly, we assume a fixed enumeration $v_1, \ldots, v_k$ of all the variables in the system. Then, for a tuple t, the array entry $t[i]$ will be the ordered list of disjoint intervals that gives the range of values for variable v_i.[4] Thus, to conjoin a constraint c on variable v_i—where the constraint c is expressed as an interval-list that limits the value for v_i—we need only to replace $t[i]$ with the conjunction of $t[i]$ and c. Since array access is performed in $O(1)$ time, the time complexity will then be determined by the complexity of conjoining two of the interval-lists.

As we showed in Section 3.6.1, this conjunction can be performed in time that is *linear* in the sizes of the interval-lists. Furthermore, the following claim shows

[4] As we showed in the proof on page 189, the number of such tuples—each of which has size $O(n)$—is $O(1)$.

that the sizes of the components of the tuples is small.

Claim 1: For a formula f and a tuple $t \in dnf(pos(f))$: $|t[i]| = O(1)$

The proof will use our assumption that, for the class C of tree-equivalent systems, the size of the individual rules is bounded by some constant k.

Let t be a tuple in $pos(f)$. Recall that $t[i]$ is a unary constraint on the variable v_i. Now there are three possible sources from which a constraint on v_i can appear in a tuple t: (1) from $indom_{v_i}$, (2) from a literal constraint appearing in f, or (3) from the precondition formula $precond_c$, for a child c of v_i. There are no other formulas in the system that can contain an occurrence of v_i.

In the following subclaim, we will prove that the number of such children c such that $precond_c$ is conjoined in $t[i]$ is $O(1)$. When we combine this result with the assumption that the precondition and indom formulas have bounded size, our claim that the size of $t[i]$ is $O(1)$ will be proven.

Subclaim: the number of children c of v_i such that $precond_c$ appears in $t[i]$ is $O(1)$

Proof.

Let us say that such children c are *t-involved*; the subclaim is then that the number of t-involved children of v_i is $O(1)$.

Let τ be a maximal local tree containing v_i, and let c be a t-involved child of v_i. Since the precondition of c is conjoined in t, and t is a tuple in $pos(f)$, then it is not hard to see that there must be a path Γ from c to f in the dependency graph.[5]

[5]For this argument, we treat f as a node in the dependency graph, just as if it were a conclusion node.

Since $f \notin \tau$, there must be a node in Γ not belonging to τ. Let x be the first node in Γ that does not belong to τ. So x has a parent p in τ. Define $\alpha(c)$ to be this node p.

Now, since τ is maximal, x is not a tree node. Hence $x \in \mathit{skel}(C) \cup \{f\}$. Now, the variable $p = \alpha(c)$ appears somewhere in the formulas for x, and hence $\alpha(c) \in FV(\mathit{skel}(C) \cup \{f\})$.

Now we argue that the mapping α, which sends t-involved children of v_i into $FV(\mathit{skel}(C) \cup \{f\})$, is injective. To show this, suppose that c_1 and c_2 are distinct t-involved children of v_i. Since they are distinct children of the same node, the descendents of c_1 in τ and the descendents of c_2 in τ are disjoint sets of tree-nodes. Hence, since $\alpha(c_1)$ is a descendent of c_1, and $\alpha(c_2)$ is a descendent of c_2, then $\alpha(c_1) \neq \alpha(c_2)$. Thus, α is injective.

Since α is an injective mapping from t-involved children to $FV(\mathit{skel}(C) \cup \{f\})$, and since the number of variables in this set is $O(1)$—since $\mathit{skel}(C) \cup \{f\}$ is a finite set of nodes that is fixed, as the systems s range over C—it follows that the number of t-involved children of v_i is $O(1)$.

That concludes the proof of the subclaim.

Hence claim (1) is proven.

Claim 2: The time-complexity of processing the non-tree nodes is $O(w)$

As we showed in the proof of Proposition 3.7.3, the constraint-complexity of processing the non-tree nodes is $O(w)$. Since the UMO constraints can be conjoined in linear time, it follows that the time-complexity of this processing is also $O(w)$.

Claim 3: The time-complexity of processing the tree nodes is $O(n)$

The cost of processing a single tree-node x is the cost of incrementally conjoining the constraints $precond_x$ and $indom_x$ into the global constraint store, which holds the solutions to $pos(q) \wedge pos(p)$, where p is the parent of x. As we showed in the proof of Proposition 3.7.3, the number of tuples in $pos(q) \wedge pos(p)$ is $O(1)$.

Let t be a tuple in $pos(q) \wedge pos(p)$. Let t_x be the constraints in t on x, and t_p be the constraints in t on p. In claim (1), we showed that $|t_x| = |t_p| = O(1)$. Also, by our initial assumptions, $|precond_x| = |indom_x| = O(1)$. The update operation involves replacing t_x with $t_x \wedge indom_x$, and replacing t_p with $t_p \wedge precond_x$. Since t is implemented as an array, *i.e.*, has $O(1)$ access time, and since the conjunction operation is performed in linear time, it follows that the time required for the update is $O(1)$.

Conclusion: the overall time-complexity is $O(n)$ ■

Proposition 4.1.1 *For a regular system s, binding b, and boolean formula f,* $b \models_s f \Leftrightarrow b \models^{\pm}_s \neg f$

Proof

$b \models_s f \Leftrightarrow (\forall b' \in F_s(b) : b' \not\models f) \Leftrightarrow (\forall b' \in F_s(b) : b' \models \neg f) \Leftrightarrow b \models^{\pm}_s \neg f$ ■

The following definition will be used to formulate a statement that the $\perp$ values appearing in a final binding have the same essential meaning as the truth value False which may be returned by the effective-value function. The definition gives a notation which we will use to equate such $\perp$ values with False.

Definition A.3 ($\cong$)

Define the equivalence relation $\cong$ by the following two specifications:

1. *$v \cong v$, for all domain values v*

2. *$\bot \cong \mathsf{False}$, and $\mathsf{False} \cong \bot$*

Lemma A.12 *For a solution bindings $b \in S_s$, formula f, and term t:*

$b \models f \Leftrightarrow \mathit{eff}(f, b) = \mathsf{True}$

$b \not\models f \Leftrightarrow \mathit{eff}(f, b) = \mathsf{False}$

$\mathit{val}(t, b) \cong \mathit{eff}(t, b) \neq \mathsf{Possible}$

Proof

We proceed by induction on the height of the expression (formula f or term t). Our induction hypothesis is that all three claims hold for all expressions of smaller height.[6]

Conjunction

$b \models f_1 \wedge f_2 \Leftrightarrow$

$b \models f_1$ and $b \models f_2 \Leftrightarrow$

$\mathit{eff}(f_1, b) = \mathsf{True}$ and $\mathit{eff}(f_2, b) = \mathsf{True} \Leftrightarrow$

$\mathit{eff}(f_1 \wedge f_2, b) = \mathsf{True}$

$b \not\models f_1 \wedge f_2 \Leftrightarrow$

[6] We added the statement that $\mathit{eff}(t, b) \neq \mathsf{Possible}$ only for purpose of emphasis; it is is already implied by statement that $\mathit{val}(t, b) \cong \mathit{eff}(t, b)$, since, by definition, the relation $x \cong y$ does not hold if either x or y is possible.

$b \not\models f_1$ or $b \not\models f_2 \Leftrightarrow$

$\mathit{eff}(f_1, b) = \mathsf{False}$ or $\mathit{eff}(f_2, b) = \mathsf{False} \Leftrightarrow$

$\mathit{eff}(f_1 \vee f_2, b) = \mathsf{False}$

Disjunction

The case is dual to the previous one.

Negation

$b \models \neg f \Leftrightarrow b \not\models f \Leftrightarrow \mathit{eff}(f, b) = \mathsf{False} \Leftrightarrow \mathit{eff}(\neg f, b) = \mathsf{True}$

$b \not\models \neg f \Leftrightarrow b \models f \Leftrightarrow \mathit{eff}(f, b) = \mathsf{True} \Leftrightarrow \mathit{eff}(\neg f, b) = \mathsf{False}$

Atomic Formula

$b \models p(t_1, \ldots, t_k) \overset{\dagger}{\Leftrightarrow}$

$(\mathit{val}(t_1, b), \ldots, \mathit{val}(t_k, b)) \in p^I \overset{\ddagger}{\Leftrightarrow}$

$(\mathit{eff}(t_1, b), \ldots, \mathit{eff}(t_k, b)) \in p^I \Leftrightarrow$

$\mathit{eff}(p(t_1, \ldots, t_k), b) = \mathsf{True}$

(†) where p^I is the relation for p under interpretation I

(‡) By induction, $\mathit{val}(t_i, b) = \mathit{eff}(t_i, b)$

Negating both sides of the list equivalence, we get:

$b \not\models p(t_1, \ldots, t_k) \Leftrightarrow$

$\mathit{eff}(p(t_1, \ldots, t_k), b) \neq \mathsf{True} \overset{\dagger}{\Leftrightarrow}$

$\mathit{eff}(p(t_1, \ldots, t_k), b) = \mathsf{False}$

(†) Since, by induction, $\mathit{eff}(t_i, b) \neq \mathsf{Possible}$, then $\mathit{eff}(p(t_1, \ldots, t_k)) \neq \mathsf{Possible}$.

Function

$val(t_1 + t_2, b) = val(t_1, b) + val(t_2, b) \cong \mathit{eff}(t_1, b) + \mathit{eff}(t_2, b) = \mathit{eff}(t_1 + t_2, b)$

Constant

$val(c, b) = c \cong c = \mathit{eff}(c, b)$

Input Variable

First suppose that $b \models \mathit{precond}_v$. Then by induction $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}$. Also, since b is a solution binding and $b \models \mathit{precond}_v$, then $b(v) \in \mathit{dom}_v$. Since $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}$, and $b(v) \neq \bot$, then $\mathit{eff}(v, b) = b(v) \cong b(v)val(v, b)$.

Now suppose that $b \not\models \mathit{precond}_v$. Then by induction $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{False}$. Also, since b is a solution binding and $b \not\models \mathit{precond}_v$, then $b(v) = \bot$. Since $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{False}$, then $\mathit{eff}(v, b) = \mathsf{False} \cong \bot = val(v, b)$.

Derived Variable

First suppose that $b \models \mathit{precond}_w$. Then by induction $\mathit{eff}(\mathit{precond}_w, b) = \mathsf{True}$. Also, since b is a solution binding and $b \models \mathit{precond}_w$, then $b(w) = val(\mathit{term}_w, b)$. Since $\mathit{eff}(\mathit{precond}_w, b) = \mathsf{True}$, then $\mathit{eff}(w, b) = \mathit{eff}(\mathit{term}_w, b)$. By induction, $\mathit{eff}(\mathit{term}_w, b) \cong val(\mathit{term}_w, b)$. Hence, $\mathit{eff}(w, b) \cong val(\mathit{term}_w, b) = b(w) = val(w, b)$.

■

Proposition 4.1.2 *For a boolean system s conforming to Definition 4.1.3:*

$$F_s = S_s$$

Proof

Claim: $F_s \subseteq S_s$

The key to this proof is the fact that, for final bindings $b \in F_s$ and formulas f, we have that $b \models f \iff b \models^{\pm} f$, and $b \not\models f \iff b \models^{} sf$. Let $b \in F_s$ be a final binding. Suppose that $v \in V_s$ is an input variable. If $b \models precond_v$, then $b \models^{\pm}_s precond_v$, and so $precond_v$ should be active in binding b. Hence, since b is final, the user will have been prompted for an input value for v, and so $b \models indom_v$. If $b \not\models precond_v$, then $b \models^{}_s precond_v$, and so it is not true that $b \models^{\pm}_s precond_v$. Hence $precond_v$ should not have been activated in b, and thus $b(v) = \perp$. A similar argument covers the case where $w \in W_s$ is a calculated variable. Hence, $b \in S_s$.

Claim: $S_s \subseteq F_s$

Let $b \in S_s$ be a solution binding. Then $ask_s(b) = \{v \in V_s \mid b \models^{\pm}_s precond_v,\ b(v) = \perp\} = \{v \in V_s \mid b \models precond_v,\ b(v) = \perp\} = \{\}$. Also, $calc_s(b) = \{(w, x) \mid w \in W_s, b \models^{\pm}_s precond_w, b(w) = \perp, x = val(term_w, b) \neq \perp\} =$
$\{(w, x) \mid w \in W_s, b \models precond_w, b(w) = \perp, x = val(term_w, b) \neq \perp\} = \{\}$. Hence, since $ask_s(b) = calc_s(b) = \{\}$, we have that $b \in F_s$. ■

Theorem 4.1.3 *For a general boolean system s with the effective-value-based firing rule:*

$$F_s = S_s$$

Proof

By induction on the number of variables in s. For $n = 0$, we have $F_s = S_s = \{\{\}\}$. Suppose that the claim is true for systems with $n - 1$ variables, and suppose that the system s has n variables. Let z be a variable in s that is a leaf in the restriction of the dependency graph to the variables. Let s_0 be the system that is obtained by removing z from s. By induction, $F_{s_0} = S_{s_0}$.

Claim 1: $S_s \subseteq F_s$

Choose $b \in S_s$. Let $b_0 = b - \{(z, b(z))\}$. Then $b_0 \in S_{s_0}$. Hence $b_0 \in F_{s_0}$.

Suppose first that $b \not\models \mathit{precond}_z$. Then, since $b \in S_s$, we have that $b(z) = \bot$. Hence $b = b_0$. Since $b \not\models z$, and $z \notin FV(\mathit{precond}_z)$, then $b_0 \not\models \mathit{precond}_z$. Since $b_0 \in S_{s_0}$, and $b_0 \not\models \mathit{precond}_z$, then by Lemma A.12, $\mathit{eff}(\mathit{precond}_z, b_0) = \mathsf{False}$. Since b_0 is final in s_0, and $\mathit{eff}(\mathit{precond}_z, b_0) = \mathsf{False}$, then $b_0 = b$ is final in s. I.e., $b \in F_s$.

Now suppose that $b \models \mathit{precond}_z$. Since $b \models \mathit{precond}_z$, and $z \notin FV(\mathit{precond}_z)$, then $b_0 \models \mathit{precond}_z$. Then, since $b_0 \in S_{s_0}$, and $b_0 \models \mathit{precond}_z$, then by Lemma A.12, $\mathit{eff}(\mathit{precond}_z, b_0) = \mathsf{True}$. We consider the two cases of the form of z.

First suppose that z is an input variable. Then, since $b \in S_s$, we have that $b(z) \in \mathit{dom}_z$. Since (1) b_0 is attainable (in both s and s_0), (2) $\mathit{eff}(\mathit{precond}_z, b_0) = \mathsf{True}$, and (3) $b(z) \in \mathit{dom}_z$, then b is attainable in s, *i.e.*, $b \in A_s$. Furthermore, since $b_0 \in F_{s_0}$, then $\mathit{ask}_{s_0}(b_0) = \{v \mid b_0(v) = \bot \text{ and } \mathit{eff}(v, b_0) = \mathsf{True}\} = \{\}$. Hence $\mathit{ask}_s(b_0) = \{\}$. Finally, since $b(z) \neq \bot$, then $z \notin \mathit{ask}_s(b)$. Hence, $\mathit{ask}_s(b) = \{\}$, and so $b \in F_s$.

Now suppose that z is a calculated variable. Then, since $b \in S_s$, we have that $b(z) = \mathit{val}(\mathit{term}_z, b)$. Then, since $z \notin FV(\mathit{term}_z)$, we have that $b(z) =$

$val(term_z, b_0)$. Since (1) b_0 is attainable, (2) $\mathit{eff}(precond_z, b_0) = \mathsf{True}$, and (3) $b(z) = val(term_z, b_0)$, then b is attainable in s, *i.e.*, $b \in A_s$. Furthermore, since $b_0 \in F_{s_0}$, then $calc_{s_0}(b_0) = \{(w, x) \mid b_0(w) = \perp$ and $\mathit{eff}(v, b_0) = \mathsf{True}$ and $x = val(term_w, b_0) \neq \perp\} = \{\}$. So $calc_s(b_0) = \{\}$. Finally, since $b(z) = val(term_z, b)$, then $z \notin vars(calc_s(b))$. Hence, $calc_s(b) = \{\}$, and so $b \in F_s$.

Claim 2: $F_s \subseteq S_s$

Choose $b \in F_s$. Let $b_0 = b - \{(z, b(z))\}$. Then $b_0 \in F_{s_0}$. Hence $b_0 \in S_{s_0}$. Since $b_0 \in S_{s_0}$, then by Lemma A.12, $\mathit{eff}(precond_z, b_0) \in \{\mathsf{True}, \mathsf{False}\}$.

Suppose first that $\mathit{eff}(precond_z, b_0) = \mathsf{False}$. Hence $\mathit{eff}(precond_z, b) = \mathsf{False}$. Since $b \in F_s$, and $\mathit{eff}(precond_z, b) = \mathsf{False}$, then $b(z) = \perp$. Since $b_0 \in S_{s_0}$, and $\mathit{eff}(precond_z, b_0) = \mathsf{False}$, then by Lemma A.12, $b_0 \not\models precond_z$. Hence $b \not\models precond_z$. Since $b_0 \in S_{s_0}$, $b \not\models precond_z$, and $b(z) = \perp$, then $b \in S_s$.

Now suppose that $\mathit{eff}(precond_z, b_0) = \mathsf{True}$. Then, since $z \notin FV(precond_z)$, we have that $\mathit{eff}(precond_z, b) = \mathsf{True}$. We consider the two cases of the form of z.

First suppose that z is an input variable. Since $b \in F_s$, and $\mathit{eff}(precond_z, b) = \mathsf{True}$, then $b(z) \in dom_z$. Since $b_0 \in S_{s_0}$, and $\mathit{eff}(precond_z, b_0) = \mathsf{True}$, then by Lemma A.12, $b_0 \models precond_z$. Hence $b \models precond_z$. Since $b_0 \in S_{s_0}$, $b \models precond_z$, and $b(z) \in dom_z$, then $b \in S_s$.

Now suppose that z is a calculated variable. Since $b \in F_s$, and $\mathit{eff}(precond_z, b) = \mathsf{True}$, then $b(z) = val(term_z, b)$. Since $b_0 \in S_{s_0}$, and $\mathit{eff}(precond_z, b_0) = \mathsf{True}$, then by Lemma A.12, $b_0 \models precond_z$. Hence $b \models precond_z$. Since $b_0 \in S_{s_0}$, $b \models precond_z$, and $b(z) = val(term_z, b)$, then $b \in S_s$. ■

Lemma A.13 *For an expression e and bindings $b, b' \in A_s$ such that $b \subseteq b'$:*

$$\mathit{eff}(e, b) \neq \mathsf{Possible} \Rightarrow \mathit{eff}(e, b') = \mathit{eff}(e, b)$$

Proof

Suppose that $b, b' \in A_s$, and that $b \subseteq b'$. We proceed by induction on the height of the expression e.

Conjunction

$\mathit{eff}(f_1 \wedge f_2, b') = \mathit{eff}(f_1, b') \wedge \mathit{eff}(f_2, b') = \mathit{eff}(f_1, b) \wedge \mathit{eff}(f_2, b) = \mathit{eff}(f_1 \wedge f_2, b)$

Disjunction

$\mathit{eff}(f_1 \wedge f_2, b') = \mathit{eff}(f_1, b') \wedge \mathit{eff}(f_2, b') = \mathit{eff}(f_1, b) \wedge \mathit{eff}(f_2, b) = \mathit{eff}(f_1 \wedge f_2, b)$

Negation

$\mathit{eff}(\neg f, b') = \neg \mathit{eff}(f, b') = \neg \mathit{eff}(f, b) = \mathit{eff}(\neg f, b)$

Atomic Formula

$\mathit{eff}(p(t_1, \ldots, t_k), b') =$

$\mathit{eff}(t_1, b') \wedge \cdots \wedge \mathit{eff}(t_k, b') \wedge ((\mathit{eff}(t_1, b'), \ldots, \mathit{eff}(t_k, b')) \in p^I) =$

$\mathit{eff}(t_1, b) \wedge \cdots \wedge \mathit{eff}(t_k, b) \wedge ((\mathit{eff}(t_1, b), \ldots, \mathit{eff}(t_k, b)) \in p^I) =$

$\mathit{eff}(p(t_1, \ldots, t_k), b)$

Function

$\mathit{eff}(t_1 + t_2, b') = \mathit{eff}(t_1, b') + \mathit{eff}(t_2, b') = \mathit{eff}(t_1, b) + \mathit{eff}(t_2, b) = \mathit{eff}(t_1 + t_2, b)$

Constant

$\mathit{eff}(c, b') = c = \mathit{eff}(c, b)$

Input Variable

We are given that $\mathit{eff}(v, b) \neq \mathsf{Possible}$. First suppose that $\mathit{eff}(v, b) = \mathsf{False}$. Then $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{False} \neq \mathsf{Possible}$. Hence, by induction, $\mathit{eff}(\mathit{precond}_v, b') = \mathit{eff}(\mathit{precond}_v, b)$. Thus, $\mathit{eff}(\mathit{precond}_v, b') = \mathsf{False}$, and so $\mathit{eff}(v, b') = \mathsf{False} = \mathit{eff}(v, b)$.

Now suppose that $\mathit{eff}(v, b) \neq \mathsf{False}$. Then, since our first assumption is that $\mathit{eff}(v, b) \neq \mathsf{Possible}$, it follows that $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}$ and that $\mathit{eff}(v, b) = b(v) \in \mathit{dom}_v$. Since $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True} \neq \mathsf{Possible}$, then by induction it follows that $\mathit{eff}(\mathit{precond}_v, b') = \mathsf{True}$. Also, since $b(v) \in \mathit{dom}_v$, and $b \subseteq b'$, then it follows that $b'(v) = b(v) \neq \bot$. Since $\mathit{eff}(\mathit{precond}_v, b') = \mathsf{True}$ and $b'(v) \neq \bot$, then $\mathit{eff}(v, b') = b'(v)$. Hence, $\mathit{eff}(v, b') = \mathit{eff}(v, b)$.

Derived Variable

We are given that $\mathit{eff}(w, b) \neq \mathsf{Possible}$. If $\mathit{eff}(w, b) = \mathsf{False}$, then the proof from the previous case shows that $\mathit{eff}(w, b') = \mathit{eff}(w, b)$.

Now suppose that $\mathit{eff}(w, b) \neq \mathsf{False}$. Then, since our first assumption is that $\mathit{eff}(v, b) \neq \mathsf{Possible}$, it follows that $\mathit{eff}(\mathit{precond}_w, b) = \mathsf{True}$ and that $\mathit{eff}(w, b) = \mathit{eff}(\mathit{term}_w, b) \in \mathit{dom}_w$. Hence, $\mathit{eff}(\mathit{term}_w, b) \neq \mathsf{Possible}$, and so by induction $\mathit{eff}(\mathit{term}_w, b') = \mathit{eff}(\mathit{term}_w, b)$. Also, since $\mathit{eff}(\mathit{precond}_w, b) = \mathsf{True} \neq \mathsf{Possible}$, then by induction it follows that $\mathit{eff}(\mathit{precond}_w, b') = \mathsf{True}$. Hence, $\mathit{eff}(w, b') = \mathit{eff}(\mathit{term}_w, b') = \mathit{eff}(\mathit{term}_w, b) = \mathit{eff}(w, b)$. ■

Proposition 4.1.4 *For a general boolean system s with the effective-value-based firing rule: s is a regular interactive system.*

Proof

By the definition of regularity, we must show that:

$$b, b' \in A_s,\ b' \supseteq b \Rightarrow \mathit{ask}(b) \subseteq \mathit{vars}(b') \cup \mathit{ask}(b') \text{ and } \mathit{calc}(b) \subseteq b' \cup \mathit{calc}(b')$$

Here are the definitions of *ask* and *calc* for thinksheet systems:

$\mathit{ask}(b) \equiv \{v \in V_s \mid b(v) = \bot \text{ and } \mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}\}$

$\mathit{calc}(b) \equiv \{(w, x) \mid w \in W_s,\ b(w) = \bot,\ \mathit{eff}(\mathit{pre}_w, b) = \mathsf{True},\ x = \mathit{val}(\mathit{term}_w, b) \neq \bot\}$

Suppose $v \in \mathit{ask}(b)$. Then $b(v) = \bot$, and $\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}$. Hence, since $b' \supseteq b$, then by Lemma A.13, $\mathit{eff}(\mathit{precond}_v, b') = \mathsf{True}$. If $b'(v) \neq \bot$, then $v \in \mathit{vars}(b')$. On the other hand, if $b'(v) = \bot$, then, since $\mathit{eff}(\mathit{precond}_v, b') = \mathsf{True}$, it follows that $v \in \mathit{ask}(b')$. Combining these two cases, we have shown that $v \in \mathit{vars}(b') \cup \mathit{ask}(b')$.

Suppose $(w, x) \in \mathit{calc}(b)$. Then $b(w) = \bot$, $\mathit{eff}(\mathit{precond}_w, b) = \mathsf{True}$, and $x = \mathit{val}(\mathit{term}_w, b) \neq \bot$. Hence, since $b' \supseteq b$, then by Lemma A.13, $\mathit{eff}(\mathit{precond}_w, b') = \mathsf{True}$, and also $x = \mathit{val}(\mathit{term}_w, b') = \mathit{val}(\mathit{term}_w, b) \neq \bot$. Now, if $b'(w) = \bot$, then, since $\mathit{eff}(\mathit{precond}_w, b') = \mathsf{True}$ and $x = \mathit{val}(\mathit{term}_w, b') \neq \bot$, then it follows that $(w, x) \in \mathit{calc}(b')$. On the other hand, if $b'(w) \neq \bot$, then, since b' is attainable, $b'(w) = \mathit{val}(\mathit{term}_w, b') = x \neq \bot$, and so $(w, x) \in b'$. Combining these two cases, we have shown that $(w, x) \in b' \cup \mathit{calc}(b')$. ■

Lemma A.14 *For expressions e and bindings b:*

$$\mathit{eff}(e, b) = \mathit{eff}(e, \mathit{core}(b))$$

Proof

We proceed by induction on the height of the expression e.

Conjunction

$\mathit{eff}(f_1 \wedge f_2, b) =$

$\mathit{eff}(f_1, b) \wedge \mathit{eff}(f_2, b) =$

$\mathit{eff}(f_1, \mathit{core}(b)) \wedge \mathit{eff}(f_2, \mathit{core}(b)) =$

$\mathit{eff}(f_1 \wedge f_2, \mathit{core}(b))$

Disjunction

Similar to the previous case.

Negation

$\mathit{eff}(\neg f, b) = \neg \mathit{eff}(f, b) = \neg \mathit{eff}(f, \mathit{core}(b)) = \mathit{eff}(\neg f, \mathit{core}(b))$

Atomic Formula

$\mathit{eff}(p(t_1, \ldots, t_k), b) =$

$\mathit{eff}(t_1, b) \wedge \cdots \wedge \mathit{eff}(t_k, b) \wedge (\mathit{eff}(t_1, b), \ldots, \mathit{eff}(t_k, b)) \in p^I =$

$\mathit{eff}(t_1, \mathit{core}(b)) \wedge \cdots \wedge \mathit{eff}(t_k, \mathit{core}(b)) \wedge (\mathit{eff}(t_1, \mathit{core}(b)), \ldots, \mathit{eff}(t_k, \mathit{core}(b))) \in p^I =$

$\mathit{eff}(p(t_1, \ldots, t_k), \mathit{core}(b))$

Function

$\mathit{eff}(t_1 + t_2, b) =$

$\mathit{eff}(t_1, b) + \mathit{eff}(t_2, b) =$

$\mathit{eff}(t_1, \mathit{core}(b)) + \mathit{eff}(t_2, \mathit{core}(b)) =$

$\mathit{eff}(t_1 + t_2, \mathit{core}(b))$

Constaint

$\mathit{eff}(c, b) = c = \mathit{eff}(c, \mathit{core}(b))$

Input Variable

By induction, we have that $\mathit{eff}(\mathit{precond}_v, b) = \mathit{eff}(\mathit{precond}_v, \mathit{core}(b))$. First suppose that $\mathit{eff}(\mathit{precond}_v, b) = \mathit{eff}(\mathit{precond}_v, \mathit{core}(b)) = \mathsf{False}$. Then $\mathit{eff}(v, b) = \mathit{eff}(v, \mathit{core}(b)) = \mathsf{False}$. Now suppose $\mathit{eff}(\mathit{precond}_v, b) = \mathit{eff}(\mathit{precond}_v, \mathit{core}(b)) = \mathsf{Possible}$. Then $\mathit{eff}(v, b) = \mathit{eff}(v, \mathit{core}(b)) = \mathsf{False}$. For the remaining case, suppose that $\mathit{eff}(\mathit{precond}_v, b) = \mathit{eff}(\mathit{precond}_v, \mathit{core}(b)) = \mathsf{True}$.

Suppose first that $b(v) \notin \mathit{dom}_v$. Now, since $\mathit{core}(b)$ is attainable, $\mathit{core}(b)(v) \in \mathit{dom}_v \cup \{\bot\}$. Combining these facts with the fact that $\mathit{core}(b) \subseteq b$, it follows that $\mathit{core}(b)(v) = \bot$. Hence, $\mathit{eff}(v, \mathit{core}(b)) = \mathsf{Possible} = \mathit{eff}(v, b)$.

Suppose finally that $b(v) \in \mathit{dom}_v$. Then $\mathit{eff}(v, b) = b(v)$.

Since $\mathit{eff}(\mathit{precond}_v, \mathit{core}(b)) = \mathsf{True}$, and $\mathit{core}(b)$ is attainable, then either (1) $\mathit{core}(b)(v) \in \mathit{dom}_v$, or (2) $\mathit{core}(b)(v) = \bot$ and $\mathit{core}(b) \cup \{(v, b(v))\}$ is attainable. But (2) is impossible, since $\mathit{core}(b)$ is maximal in A_s, Hence $\mathit{core}(b)(v) \in \mathit{dom}_v$. Therefore, since $\mathit{core}(b) \subseteq b$, it follows that $\mathit{core}(b)(v) = b(v) \in \mathit{dom}_v$. Hence, $\mathit{eff}(v, \mathit{core}(b)) = b(v) = \mathit{eff}(v, b)$.

Calculated Variable

By induction, we have that $\mathit{eff}(\mathit{precond}_w, b) = \mathit{eff}(\mathit{precond}_w, \mathit{core}(b))$. First sup-

pose that $\mathit{eff}(\mathit{precond}_w, b) = \mathit{eff}(\mathit{precond}_w, \mathit{core}(b)) = \mathsf{False}$. Then $\mathit{eff}(w, b) = \mathit{eff}(w, \mathit{core}(b)) = \mathsf{False}$. Now suppose $\mathit{eff}(\mathit{precond}_w, b) = \mathit{eff}(\mathit{precond}_w, \mathit{core}(b)) = \mathsf{Possible}$. Then $\mathit{eff}(w, b) = \mathit{eff}(w, \mathit{core}(b)) = \mathsf{False}$. For the remaining case, suppose that $\mathit{eff}(\mathit{precond}_w, b) = \mathit{eff}(\mathit{precond}_w, \mathit{core}(b)) = \mathsf{True}$. Then $\mathit{eff}(w, b) = \mathit{eff}(\mathit{term}_w, b)$, and $\mathit{eff}(w, \mathit{core}(b)) = \mathit{eff}(\mathit{term}_w, \mathit{core}(b))$. Since, by induction $\mathit{eff}(\mathit{term}_w, b) = \mathit{eff}(\mathit{term}_w, \mathit{core}(b))$, it follows that $\mathit{eff}(w, b) = \mathit{eff}(w, \mathit{core}(b))$. ■

Lemma A.15 *For bindings b and expressions e:*

$b \models \mathit{pos}(e) \Rightarrow \mathit{eff}(e, b) \notin \{\mathsf{False}, \mathsf{Possible}\}$

$b \models \mathit{neg}(e) \Rightarrow \mathit{eff}(e, b) = \mathsf{False}$

Proof

We proceed by induction on the height of the expression e.

Conjunction

$b \models \mathit{pos}(f_1 \wedge f_2) \Rightarrow$

$b \models \mathit{pos}(f_1)$ and $b \models \mathit{pos}(f_2) \Rightarrow$

$\mathit{eff}(f_1, b) \notin \{\mathsf{False}, \mathsf{Possible}\}$ and $\mathit{eff}(f_2, b) \notin \{\mathit{false}, \mathit{poss}\} \Rightarrow$

$\mathit{eff}(f_1, b) = \mathsf{True}$ and $\mathit{eff}(f_2, b) = \mathsf{True} \Rightarrow$

$\mathit{eff}(f_1 \wedge f_2, b) = \mathsf{True} \notin \{\mathsf{False}, \mathsf{Possible}\}$

$b \models \mathit{neg}(f_1 \wedge f_2) \Rightarrow$

$b \models \mathit{neg}(f_1)$ or $b \models \mathit{neg}(f_2) \Rightarrow$

$\mathit{eff}(f_1, b) = \mathsf{False}$ or $\mathit{eff}(f_2, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(f_1 \wedge f_2, b) = \mathsf{False}$

Disjunction

The proof is dual to proof for the previous case.

Negation

$b \models pos(\neg f) \Rightarrow$

$b \models neg(f) \Rightarrow$

$\mathit{eff}(f, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(\neg f, b) = \mathsf{True}$

Similarly, $b \models neg(\neg f) \Rightarrow \mathit{eff}(\neg f, b) = \mathsf{False}$

Atomic Formula

$b \models pos(p(t_1, \ldots, t_k)) \Rightarrow$

$b \models pos(t_1)\ \&\ \ldots\ \&\ b \models pos(t_k)\ \&\ b \models p(t_1, \ldots, t_k) \Rightarrow$

$\mathit{eff}(t_1, b) \notin \{\mathsf{False}, \mathsf{Possible}\}\ \&\ \ldots\ \&\ \mathit{eff}(t_k, b) \notin \{\mathsf{False}, \mathsf{Possible}\}\ \&\ b \models$

$p(t_1, \ldots, t_k) \Rightarrow$

$\mathit{eff}(p(t_1, \ldots, t_k), b) \notin \{\mathsf{False}, \mathsf{Possible}\}$

$b \models neg(p(t_1, \ldots, t_k)) \Rightarrow$

$b \models neg(t_1)$ or $\ldots$ or $b \models neg(t_k)$ or $b \models pos(\overline{p}(t_1, \ldots, t_k)) \Rightarrow$

$\mathit{eff}(t_1, b) = \mathsf{False}$ or $\mathit{eff}(t_k, b) = \mathsf{False}$ or $\mathit{eff}(\overline{p}(t_1, \ldots, t_k), b) = \mathsf{True} \Rightarrow$

$\mathit{eff}(p(t_1, \ldots, t_k), b) = \mathsf{False}$

Function

$b \models pos(t_1 + t_2) \Rightarrow$

$b \models pos(t_1)$ and $b \models pos(t_2) \Rightarrow$

$\mathit{eff}(t_1, b) \notin \{\mathsf{False}, \mathsf{Possible}\}$ and $\mathit{eff}(t_2, b) \notin \{\mathsf{False}, \mathsf{Possible}\} \Rightarrow$

$\mathit{eff}(t_1 + t_2, b) \notin \{\mathsf{False}, \mathsf{Possible}\}$

$b \models neg(t_1 + t_2) \Rightarrow$

$b \models neg(t_1)$ or $b \models neg(t_2) \Rightarrow$

$\mathit{eff}(t_1, b) = \mathsf{False}$ or $\mathit{eff}(t_2, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(t_1 + t_2, b) = \mathsf{False}$

Constant

Always: $\mathit{eff}(c, b) = c \notin \{\mathsf{False}, \mathsf{Possible}\}$

$b \models neg(c) \Rightarrow b \models \mathit{false} \Rightarrow$ everything

Input Variable

$b \models pos(v) \Rightarrow$

$b \models pos(\mathit{precond}_v)$ and $b \models \mathit{indom}_v \Rightarrow$

$\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}$ and $b(v) \in \mathit{dom}_v \Rightarrow$

$\mathit{eff}(v, b) = b(v) \notin \{\mathsf{False}, \mathsf{Possible}\}$

$b \models pos(v) \Rightarrow$

$b \models pos(\mathit{precond}_v)$ and $b \models \mathit{indom}_v \Rightarrow$

$\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}$ and $b(v) \in \mathit{dom}_v \Rightarrow$

$\mathit{eff}(v, b) = b(v) \notin \{\mathsf{False}, \mathsf{Possible}\}$

Calculated Variable

$b \models pos(w) \Rightarrow$

$b \models pos(precond_w)$ and $b \models pos(term_w) \Rightarrow$

$\mathit{eff}(precond_w, b) = \mathsf{True}$ and $\mathit{eff}(term_w, b) \notin \{\mathsf{False}, \mathsf{Possible}\} \Rightarrow$

$\mathit{eff}(w, b) = \mathit{eff}(term_w, b) \notin \{\mathsf{False}, \mathsf{Possible}\}$

$b \models neg(w) \Rightarrow$

$b \models neg(precond_w)$ or $b \models neg(term_w) \Rightarrow$

$\mathit{eff}(precond_w, b) = \mathsf{False}$ or $\mathit{eff}(term_w, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(w, b) = \mathsf{False}$ ■

Lemma A.16 *For an expression e and a binding $b \in F_s$:*

$\mathit{eff}(e, b) \neq \mathsf{False} \Rightarrow b \models pos_s(e)$

$\mathit{eff}(e, b) = \mathsf{False} \Rightarrow b \models neg_s(e)$

Proof

We proceed by induction on the height of the expression e. Since $b \in F_s$, then by Lemma A.12, we know that $\mathit{eff}(e, b) \neq \mathsf{Possible}$.

Conjunction

$\mathit{eff}(f_1 \wedge f_2, b) \neq \mathsf{False} \Rightarrow$

$\mathit{eff}(f_1 \wedge f_2, b) = \mathsf{True} \Rightarrow$

$\mathit{eff}(f_1, b) = \mathsf{True}$ and $\mathit{eff}(f_2, b) = \mathsf{True} \Rightarrow$

$b \models pos(f_1)$ and $b \models pos(f_2) \Rightarrow$

$b \models pos(f_1 \wedge f_2)$

$\mathit{eff}(f_1 \wedge f_2, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(f_1, b) = \mathsf{False}$ or $\mathit{eff}(f_2, b) = \mathsf{False} \Rightarrow$

$b \models \mathit{neg}(f_1)$ or $b \models \mathit{neg}(f_2) \Rightarrow$

$b \models \mathit{neg}(f_1 \wedge f_2)$

Disjunction

The proof is dual to proof for the previous case.

Negation

$\mathit{eff}(\neg f, b) \neq \mathsf{False} \Rightarrow$

$\mathit{eff}(\neg f, b) = \mathsf{True} \Rightarrow$

$(\neg \mathit{eff}(f, b)) = \mathsf{True} \Rightarrow$

$\mathit{eff}(f, b) = \mathsf{False} \Rightarrow$

$b \models \mathit{neg}(f) \Rightarrow$

$b \models \mathit{pos}(\neg f)$

$\mathit{eff}(\neg f, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(f, b) = \mathsf{True} \Rightarrow$

$b \models \mathit{pos}(f) \Rightarrow$

$b \models \mathit{neg}(\neg f)$

Atomic Formula

$\mathit{eff}(p(t_1, \ldots, t_k), b) \neq \mathsf{False} \Rightarrow$

$\mathit{eff}(p(t_1, \ldots, t_k), b) = \mathsf{True} \Rightarrow$

$\mathit{eff}(t_1, b) \notin \{\mathsf{False}, \mathsf{Possible}\}$ and $\ldots$ and $\mathit{eff}(t_k, b) \notin \{\mathsf{False}, \mathsf{Possible}\}$

and $(\mathit{eff}(t_1, b), \ldots, \mathit{eff}(t_k, b)) \in p^I \overset{\dagger}{\Rightarrow}$

$b \models \mathit{pos}(t_1)$ and $\ldots$ and $b \models \mathit{pos}(t_k)$ and $(\mathit{val}(t_1, b), \ldots, \mathit{val}(t_k, b)) \in p^I \Rightarrow$

$b \models \mathit{pos}(t_1)$ and $\ldots$ and $b \models \mathit{pos}(t_k)$ and $b \models p(t_1, \ldots, t_k) \Rightarrow$

$b \models \mathit{pos}(p(t_1, \ldots, t_k))$

(†) By Lemma A.12, for $b \in F_s = S_s$, we have that $\mathit{eff}(t_i, b) = \mathit{val}(t_i, b)$

$\mathit{eff}(p(t_1, \ldots, t_k), b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(t_1, b) = \mathsf{False}$ or $\ldots$ or $\mathit{eff}(t_k, b) = \mathsf{False}$ or $\mathit{eff}(\overline{p}(t_1, \ldots, t_k), b) = \mathsf{True} \Rightarrow$

$b \models \mathit{neg}(t_1)$ or $\ldots$ or $b \models \mathit{neg}(t_k)$ or $b \models \overline{p}(t_1, \ldots, t_k) \Rightarrow$

$b \models \mathit{neg}(p(t_1, \ldots, t_k))$

Function

$\mathit{eff}(t_1 + t_2, b) \neq \mathsf{False} \Rightarrow$

$\mathit{eff}(t_1, b) \neq \mathsf{False}$ and $\mathit{eff}(t_2, b) \neq \mathsf{False} \Rightarrow$

$b \models \mathit{pos}(t_1)$ and $b \models \mathit{pos}(t_2) \Rightarrow$

$b \models \mathit{pos}(t_1 + t_2)$

$\mathit{eff}(t_1 + t_2, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(t_1, b) = \mathsf{False}$ or $\mathit{eff}(t_2, b) = \mathsf{False} \Rightarrow$

$b \models \mathit{neg}(t_1)$ or $b \models \mathit{neg}(t_2) \Rightarrow$

$b \models \mathit{neg}(t_1 + t_2)$

Constant

Since $\mathit{pos}(c) = \mathit{true}$, then always: $b \models \mathit{pos}(c)$.

$\mathit{eff}(c, b) = \mathsf{False}$ is impossible, since $\mathit{eff}(c, b) = c$.

Input Variable

$\mathit{eff}(v, b) \notin \{\mathsf{False}, \mathsf{Possible}\} \Rightarrow$

$\mathit{eff}(\mathit{precond}_v, b) = \mathsf{True}$ and $\mathit{eff}(v, b) = b(v) \in \mathit{dom}_v \Rightarrow$

$b \models \mathit{pos}(\mathit{precond}_v)$ and $b \models \mathit{indom}_v \Rightarrow$

$b \models \mathit{pos}(v)$

$\mathit{eff}(v, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(\mathit{precond}_v, b) = \mathsf{False} \Rightarrow$

$b \models \mathit{neg}(\mathit{precond}_v) \Rightarrow$

$b \models \mathit{neg}(v)$

Calculated Variable

$\mathit{eff}(w, b) \notin \{\mathsf{False}, \mathsf{Possible}\} \Rightarrow$

$\mathit{eff}(\mathit{precond}_w, b) = \mathsf{True}$ and $\mathit{eff}(\mathit{term}_w, b) \notin \{\mathsf{False}, \mathsf{Possible}\} \Rightarrow$

$b \models \mathit{pos}(\mathit{precond}_w)$ and $b \models \mathit{pos}(\mathit{term}_w) \Rightarrow$

$b \models \mathit{pos}(w)$

$\mathit{eff}(w, b) = \mathsf{False} \Rightarrow$

$\mathit{eff}(\mathit{precond}_w, b) = \mathsf{False}$ or $\mathit{eff}(\mathit{term}_w, b) = \mathsf{False} \Rightarrow$

$b \models \mathit{neg}(\mathit{precond}_w)$ or $b \models \mathit{neg}(\mathit{term}_w) \Rightarrow$

$b \models \mathit{neg}(w)$ ■

Corollary A.17

For general bindings b and formulas f,

$b \models pos_s(f) \Rightarrow b \models^{\pm}_s f$

$b \models neg_s(f) \Rightarrow b \models^{=}_s f$

Proof

Recall the definitions that: (1) $F_s(b) \equiv \{b' \in F_s \mid b' \supseteq core(b)\}$, (2) $b \models^{\pm}_s f$ means $\forall b' \in F_s(b) : b' \models f$, and (3) $b \models^{\pm}_s f$ means $\forall b' \in F_s(b) : b' \not\models f$.

Suppose that $b \models pos_s(f)$. Then by Lemma A.15, $\mathit{eff}_s(f, b) = \mathsf{True}$. Hence, by Lemma A.14, $\mathit{eff}_s(f, core_s(b)) = \mathsf{True}$. Hence, since $\forall b' \in F_s(b) : core(b) \subseteq b'$, then by Lemma A.13, $\forall b' \in F_s(b) : \mathit{eff}_s(f, b') = \mathsf{True}$. Hence, by Lemma A.12, $\forall b' \in F_s(b) : b' \models f$. I.e., $b \models^{\pm}_s f$.

Now suppose that $b \models neg_s(f)$. By definition, this means that $b \models pos_s(\neg f)$. Hence, by the first part of the claim, which we have just proved, $b \models^{\pm}_s \neg f$. This is equivalent to the statement that $b \models^{=}_s f$. ■

Corollary A.18 *For formulas f and final bindings $b \in F_s$:*

$b \models f \Rightarrow b \models pos(f)$

$b \not\models f \Rightarrow b \models neg(f)$

Proof

Suppose $b \in F_s$. Suppose that $b \models f$. Then, since $b \in F_s = S_s$, by Lemma A.12, $\mathit{eff}(f, b) = \mathsf{True}$. Hence, by Lemma A.16, $b \models pos(f)$. Suppose that $b \not\models f$. Then, since $b \in F_s = S_s$, by Lemma A.12, $\mathit{eff}(f, b) = \mathsf{False}$. Hence, by Lemma A.16, $b \models neg(f)$. ■

Corollary A.19 *For boolean production systems s, formulas f and final bindings $b \in F_s$:*

$b \models^{+}_{s} f \Rightarrow b \models pos(f)$

$b \models^{-}_{s} f \Rightarrow b \models neg(f)$

Proof

Follows from the previous corollary, and from the fact that for final bindings b, $b \models f \Leftrightarrow b \models^{+}_{s} f$, and $b \not\models f \Leftrightarrow b \models^{-}_{s} f$, ■

Lemma A.20 *For boolean production system s, variable $v \in V_s$, and formula q:* $choices_s(v, q) = \{b(v) \mid b \in B \ \ \& \ \ b \models (pos_s(q) \wedge pos_s(v))\}$

Proof

First suppose that $z \in choices_s(v, q)$. This means that $z \neq \bot$, and that $\exists b \in F_s$ *s.t.* $b \models q$ and $b(v) = z$. Since $b \in F_s$, and $b \models q$, then by Lemmas A.12 and A.16, we have that $b \models pos(q)$. Next, since $b \in F_s = S_s$, and $b(v) \neq \bot$, then $b \models precond_v$. Hence, by Lemma A.12, $\mathit{eff}(precond_v, b) = \mathsf{True}$, and so by Lemma A.16, $b \models pos(precond_v)$. Since $b \in A_s$, then $b \models indom_v$. Hence, $b \models pos(precond_v) \wedge indom_v$, *i.e.*, $b \models pos(v)$. Thus, $b \models pos(q) \wedge pos(v)$. Therefore $z = b(v)$ belongs to the set on the right hand side of the claimed equation.

Now suppose that z belongs to the set on the right side of the claimed equation, *i.e.*, that for some binding b, we have that $b(v) = z$, and $b \models pos(q) \wedge pos(v)$. Since $b \models pos(q)$, then by Corollary A.17, $b \models^{+} q$. Since $b \models pos(v)$, then $b \models pos(v = v)$, and so, by Corollary A.17, $b \models^{+} (v = v)$. Choose $b' \in F_s(b)$, *i.e.*, let b' be some final extension of the core of b. Then, since $b \models^{+} q$, we have that $b' \models q$, and

since $b \models^{\pm} (v = v)$, we have that $b' models (v = v)$, *i.e.*, that $b'(v) \neq \bot$. Therefore, $b'(v) \in choices_s(v, q)$.

Now, since $b \models pos(v)$, then $b \models pos(v = v)$. Hence, by Lemma A.15, $\mathit{eff}(v = v, b) = \mathsf{True}$. Hence, $\mathit{eff}(v, b) \in dom_v$. Hence, $\mathit{eff}(v, b) = b(v) \neq \bot$. Also, by Lemma A.14, $\mathit{eff}(v, b) = \mathit{eff}(v, core(b))$. Thus, $\mathit{eff}(v, core(b)) = b(v) \neq \bot$. Since $\mathit{eff}(v, core(b)) \neq \bot$, it follows that $\mathit{eff}(v, core(b)) = core(b)(v)$. Hence, $core(b)(v) = b(v) \neq \bot$. Since $b' \supseteq core(b)$, then $b'(v) = core(b)(v) = b(v)$. Hence, $z = b(v) = b'(v) \in choices_s(v, q)$. ■

Theorem 4.2.1 *The query algorithm shown on pages 87 and 86 is correct for acyclic production systems with general boolean preconditions.*

Proof

Syntactically, we have obtained the acyclic production systems from the monotone systems by dropping the assumption that there are no negations in the preconditions. The *only* change that we made to our query algorithm to accomodate this generalization was to add two cases to the definitions of *pos* and *neg*, viz., that $pos(\neg f) = neg(f)$, and $neg(\neg f) = pos(f)$.

Moreover, in this appendix, we have proved generalized versions of every one of the theorems that was used in the proof of correctness for the monotone systems. The starting point for these generalized theorems is that we extended the definition of the satisfaction relation $b \models f$ by adding the case that $b \models \neg f$ means $b \not\models f$. This immediately led to an extension of the definitions $b \models^{\pm}_s f$ and $b \models^{-}_s f$, for boolean formulas f.

Now, examining the proof of correctness for the monotone systems, given in

Theorem 3.3.1 on page 94, we see that it uses only the following results: Corollaries 2.6.2, 3.2.4, 3.2.7, and Theorem 3.2.8. Corollary 2.6.2 holds for all regular systems, and, as we showed in Proposition 4.1.4, the boolean production systems under consideration are regular. For the rest, in this appendix we have proven these *exact same results* in the generalized context boolean production systems s. Here are the correspondences between the earlier proofs, and the generalized proofs in the appendix:

- Corollary 3.2.4 = Corollary A.17
- Corollary 3.2.7 = Corollary A.19
- Theorem 3.2.8 = Lemma A.20

Therefore, the original proof of correctness, as it stands, also shows the correctness of the generalized algorithm. ■

Appendix B

Optimization Experiments

Here we describe some experiments to confirm the theoretical result that we proved in Proposition 3.7.4: when preconditions are univariate comparisons, our implementation has a tree-complexity for reachability queries that is linear in time.

Design of the Experiments

Our experiments are performed on production systems that contain (1) a large internal decision tree, (2) a collection of root nodes, which are referenced by the precondition at the root of the tree, and (3) a collection of conclusion nodes, called *fusion nodes*, whose preconditions are disjunctions of constraints on the leaves of the tree.

Notation:

n = total number of nodes

r = number of roots

f = number of fusion nodes

p = number of parents per fusion node

Further Specifications:

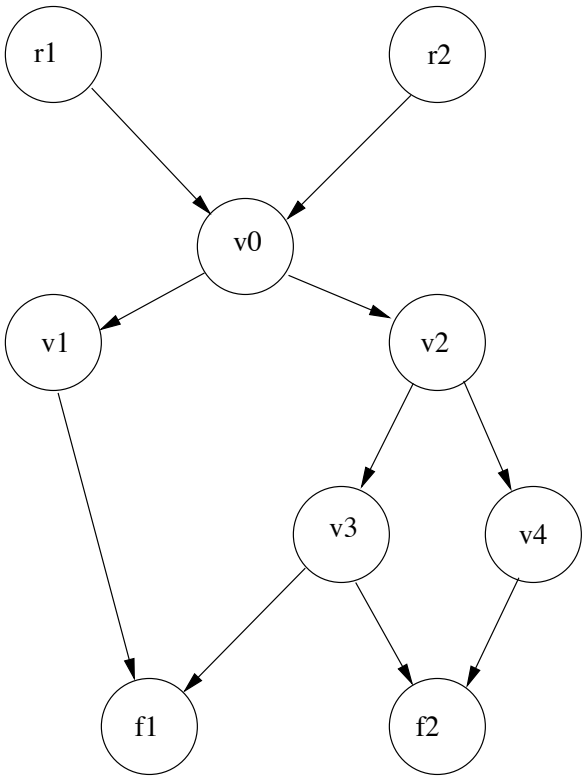
1. The domains of the variables is fixed at {1,2,...,10}
2. The precondition at the root of the tree is a disjunction of two conjunctions of constraints on the roots of the DAG
3. The query formula is a disjunction two univariate constraints on the leaves, e.g., '$v_1 \neq 2 \mid v_3 > 9$'

Example B.0.1 *Figure B.1 shows an example of such a production system, where* $n = 9$, $r = 2$, $f = 2$, *and* $p = 2$.

For fixed values of n, f, p, and r, we wish to measure the time that the algorithm takes to determine the reachability of all the nodes in the system. In order to do this, we will treat the shape of the tree, and the specific constraints in the formulas, as random structures. Our measurement for the data point (n, f, p, r) will be the mean of all the query times for an associated set of randomly generated production systems.

Generation of a Random Tree

In Section 3.7, we have shown that general-purpose datalog evaluation algorithms, which compute more information than we need to solve our problem, attain their



Preconditions:

$precond_{r_1} = true$

$precond_{r_2} = true$

$precond_{v_0} = ((r_1 > 2 \wedge r_2 = 4) \vee (r_1 = 7 \wedge r_2 \neq 9))$

$precond_{v_1} = (v_0 = 1)$

$precond_{v_2} = (v_0 = 7)$

$precond_{v_3} = (v_2 > 5)$

$precond_{v_4} = (v_2 \neq 4)$

$precond_{f_1} = (v_1 = 7 \vee v_3 = 5)$

$precond_{f_2} = (v_3 = 1 \vee v_4 = 2)$

Figure B.1: Production System with a Local Tree ($f = p = r = 2$)

worst-case complexity—quadratic—on trees that are unbalanced. We also proved there that our algorithm has linear complexity, irrespective of the shape of the trees. For purposes of comparing our algorithm with the others, therefore, we will force the randomly generated trees to be unbalanced.[1]

Figure B.2 outlines the method that we use to generate the random trees. The maximum branching factor of a node is fixed at four. The respective sizes of the four subtrees are n_1, n_2, n_3 and n_4. These numbers are required to satisfy the following constraints:

- $n_i \geq 0$
- $n_1 + n_2 + n_3 + n_4 = n - 1$
- $n_1 + n_2 + n_3 \leq \mathit{leftmax}$

where *leftmax* is a constant that controls the "rate of descent" of the unbalanced tree.[2] For these experiments, $\mathit{leftmax} = 3$.

[1]We can generally expect the worst-case behavior of any reachability query algorithm to be attained on trees that are unbalanced, for the following reason. Let v_0 be the local root of the internal tree, and q the query formula. Let $v_0, v_1, ..., v_k$ be a path in the tree from v_0 to v_k. Then, as we have shown previously, v_k is q-reachable iff $\mathit{pos}(q) \wedge \mathit{pos}(v_0) \wedge \mathit{precond}_{v_1} \wedge \mathit{indom}_{v_1} \wedge \cdots \wedge \mathit{precond}_{v_k} \wedge \mathit{indom}_{v_k}$ is satisfiable. Hence, the number of constraints involved in testing v_k for reachability is proportional to the depth of v_k im the tree. Since the expected depth of a node in a tree is maximized when the tree is unbalanced, it follows that unbalanced trees place the greatest strain on a reachability query algorithm.

[2]The numbers $n_1, ..., n_4$ are chosen as follows. Let i_1, i_2 be two randomly chosen numbers in $\{0, ..., \mathit{leftmax}\}$, and, renaming if necessary, assume that $i_1 \leq i_2$. Then, let $n_1 = i_1 - 0$, $n_2 = i_2 - i_1$, and $n_3 = \mathit{leftmax} - i_2$. Finally, let $n_4 = (n-1) - (n_1 + n_2 + n_3)$.

```
node treegen(int n)
begin
    if n = 0 then return NULL
    root = new node
    if n = 1 then return root
    let n1,...,n4 be four randomly generated non-negative integers
       such that (1) n1 + n2 + n3 + n4 = n-1
             and (2) n1 + n2 + n3 <= leftmax
    for i=1,...,4
       if ni > 0 then
           attach treegen(ni) as a subtree of root
    return root
end
```

Figure B.2: Routine to Generate an Unbalanced Random Tree of Size n

Generation of a Random Constraint

We generate a random constraint on variable v by: (1) randomly choosing a comparison operators $\theta \in \{=, \neq, >, <, \leq, \geq\}$, (2) and randomly choosing a number $c \in \{1, ..., 10\}$, and (3) producing the constraint: $v\ \theta\ c$. We prohibit this process from generating either of the constraints '$v < 1$', or '$v > 10$', which are inconsistent with the domain constraint on the variables.[3]

[3]This is reasonable because the loading process for the query system will easily detect such inconsistent constraints, and simplify the preconditions by removing them. Moreover, it is important to prohibit such constraints, for the following reason. If a precondition in a tree is false, then,

The Main Experiment

All experiments were run on a Sun SPARC workstation. The central processor was a 143 MHz UltraSPARC with 96 megabytes of main memory.

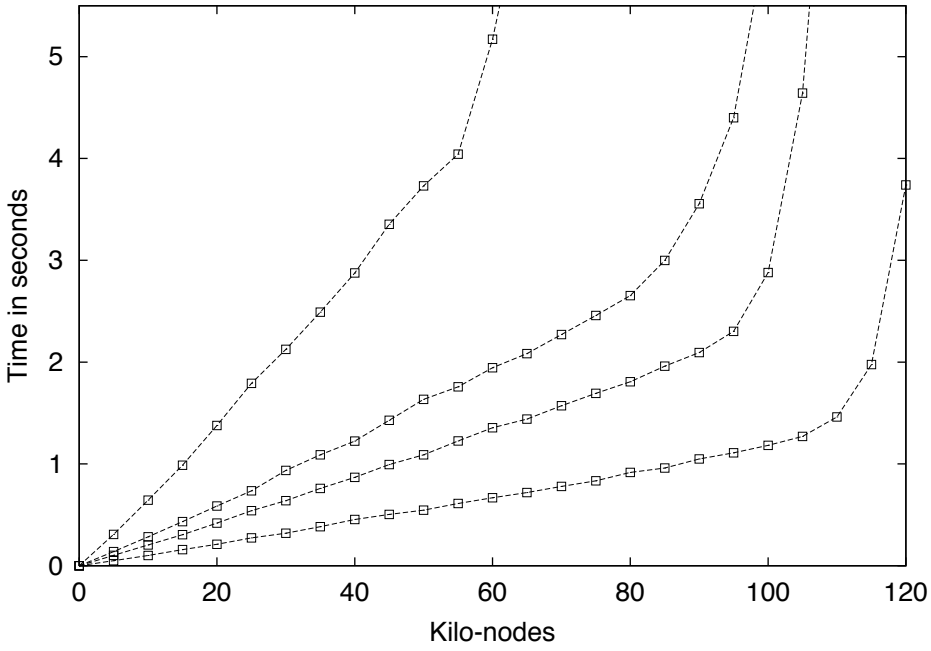
For each data point (n, f, p, r), we generated 160 random production systems with associated query formulas, and computed the mean running time. The graph in Figure B.3 shows four series of observations. In each series, the values of f, p and r are fixed, while n ranges from 0 to 120,000.

Interpretation of the Results

Figure B.3 shows that there is a significant region where the running time is linear in the size of the input. To see this clearly, one can cover up part of the graph with a piece of paper oriented at about 45 degrees, perpendicular to the four curves emanating from the origin. From the slopes of these lines, we see that the times spent per node, in the linear zone, are, respectively, 11, 22, 31 and 72 microseconds, as f=p=r ranges over 1,2,4,8.

The break from linear-time behavior occurs when disk paging begins. The memory usage of the program is $\theta(n)$, where the constant factor increases with f, p and r. Therefore, the value of n at which the linearity breaks down—*i.e.*, the point at which virtual memory usage equals physical memory capacity—gets progressively lower as the values of f, p and r increase.

trivially, all of its descendents in that tree are unreachable. Thus, by avoiding such constraints, we press a reachability query algorithm towards its worst-case behavior.



Parameters:

$f = p = r = 8$ (top curve)

$f = p = r = 4$

$f = p = r = 2$

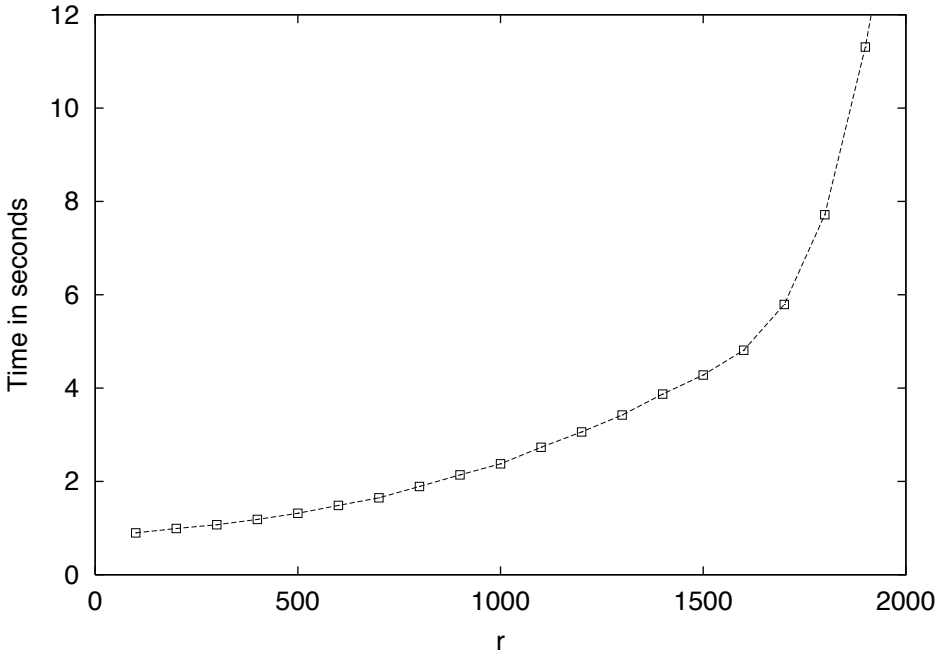
$f = p = r = 1$ (bot curve)

reps = 160

Figure B.3: Results of the Main Experiment

Auxiliary Experiments

In order to probe the dependence of the running time upon f, p and r, we ran a series for each of these parameters. The dependences upon f, p and r are shown, respectively, in Figures B.5, B.6, and B.4. Roughly speaking, disk paging is occurring



Parameters:

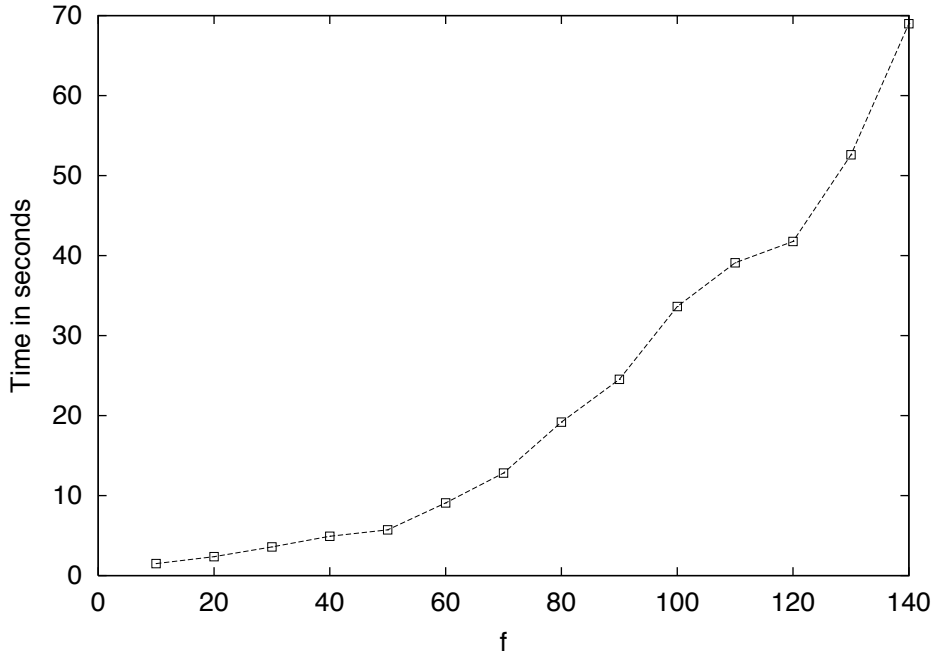
$n = 30,000$

$f = p = 4$

reps = 90

Figure B.4: Results of Auxiliary Experiment 1

in the last three data points of each of these curves.



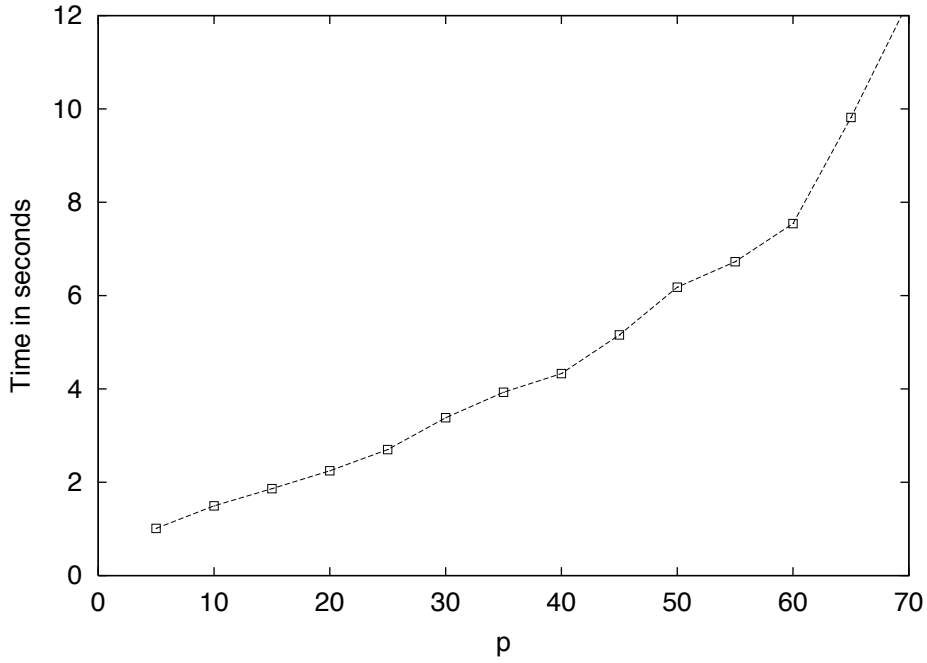
Parameters:

$n = 30,000$

$p = r = 4$

reps $= 90$

Figure B.5: Results of Auxiliary Experiment 2



Parameters:

$n = 30,000$

$f = r = 4$

reps $= 90$

Figure B.6: Results of Auxiliary Experiment 3

Bibliography

[1] K.R. Apt. Logic programming. *Handbook of theoretical computer science, Volume B, van Leeuwen, J. ed.*, pages 493–574, 1990.

[2] A. Brodsky. Constraint databases: Promising technology or just intellectual exercise? *ACM Computing Surveys*, 28(4es), 1996. http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a59-brodsky.

[3] J. Byon and P. Revesz. DISCO: A constraint database with sets. In *Contessa Workshop Constraint Databases and Applications (LCNS 1034)*, pages 68–83, Friedrichshafen, Germany, september 1995.

[4] A.K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21:156–178, 1980.

[5] E. Charniak and M. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, 1985.

[6] J. Cohen. Constraint logic programming. *CACM*, 33(7):52–68, 1990.

[7] A. Colmerauer. An introduction to Prolog III. *CACM*, 33(7):70–90, 1990.

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

[9] T. Cormen, E. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[10] G.B. Dantzig. *Linear Programming*. Springer, 1997.

[11] M. Davis, R. Sigal, and E. Weyuker. *Computability, Complexity, and Languages*. Academic Press, second edition, 1994.

[12] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proc. Fifth Generation Computer Systems*, Tokyo, Japan, 1988.

[13] ECRC. *Eclipse User's Manual*.

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

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

[16] J. Jaffar and Lassez J. L. Constraint logic programming. In *POPL*, pages 111–119, 1987.

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

[18] J. Jaffar, S. Michaylov, P. Stuckey, and R.H.C. Yap. The CLP(R) language

and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

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

[20] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. In *Proceedings of PODS Symposium*, pages 299–313, 1990.

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

[22] D. Kemp and P. Stuckley. Optimizing bottom-up evaluation of constraint queries. *Journal of Logic Programming*, 26(1):1–30, 1996.

[23] S. Lang. *Algebra*. Addison Wesley, third edition, 1992.

[24] W. Leler. *Constraint programming languages : their specification and generation*. Addison-Wesley, 1987.

[25] J. W. LLoyd. *Foundations of logic programming*. Springer-Verlag, second edition, 1987.

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

[27] 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.

[28] K. Podnieks. *Around Goedel's Theorem: Hyper-textbook for students in mathematical logic*. 1992.

[29] I.V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient tabling mechanisms for logic programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 697–711. MIT Press, 1995.

[30] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the International Conference on Very Large Databases*, pages 359–371, 1990.

[31] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic Publishers, 1992.

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

[33] A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization, 1987.

[34] D. Shasha. Conditional transition networks and computational processes for use in interactive computer-based systems, patent no. us05809212, 1998.

[35] D. Shasha and C. Lazere. *Out of their minds: the lives and discoveries of 15 great computer scientists*. Copernicus, 1995.

[36] M. Stefik. *Introduction to Knowledge Systems*. Morgan Kaufmann, San Francisco, Calif., 1995.

[37] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Mass., 1986.

[38] G. Strang. Linear algebra and its applications, 1988.

[39] D. Tanzer. *A Survey of Constraint Relational Databases*. 1998.

[40] D. Tanzer and D. Shasha. Queryable acyclic production systems. In *Proc. 8th ACM Conference Information and Knowledge Management (CIKM)*, pages 284–291, November 1999.

[41] D. Toman. Memoing evaluation for constraint extensions of datalog. *Constraints: An International Journal*, 2:337–359, 1997.

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

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

[44] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[45] P. Van Hentenryck, editor. *Special Issue on Constraint Logic Programming, Journal of Logic Programming 16(3/4)*. 1993.

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