

LOGICAL SPREADSHEETS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Michael Kassoff

August 2011

© 2011 by Michael Adam Kassoff. All Rights Reserved.

Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.

<http://creativecommons.org/licenses/by-nc/3.0/us/>

This dissertation is online at: <http://purl.stanford.edu/gt527cs1860>

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Michael Genesereth, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Vinay Chaudhri

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Computerized spreadsheets are tremendously popular and useful. Despite their success, computerized spreadsheet systems today have significant and unnecessary restrictions that limit their usefulness. One significant restriction is that the formulas used to specify calculations must be functions. Another unnecessary restriction is that propagation can only occur in one direction. We can lift these restrictions, allowing for many-to-many logical constraints and multidirectional propagation. We call such a spreadsheet a *logical spreadsheet*.

We allow for inconsistency between the constraints of the spreadsheet. To allow for the consequences of the spreadsheet to be shown, we create a new paraconsistent entailment relation and show how it can be computed.

We discuss the issues involved in updating a logical spreadsheet, and design a family of domain-independent update operators for updating logical spreadsheets. To allow for domain-dependent behavior, we design a logic called *Markov Change Logic* that can be used to express update policies for spreadsheets. The design of Markov Change Logic is motivated in part by an analysis of dynamic database constraints, in which we prove that all database constraints can be reduced to Markov dynamic constraints if the schema may be reformulated.

We describe the implementation of a logical spreadsheet engine called Webcell which can be used to turn Web pages into logical spreadsheets, and discuss its application to the Stanford Computer Science Master's Program Sheets.

Acknowledgements

This dissertation is the result of 10 years of work. It began, somewhat by chance, as I met Prof. Mike Genesereth as a teaching assistant for his course on Information Integration. I was thoroughly unqualified to TA the course, but, fortunately for me, so was everyone else who might have wanted the position, as it was the first time that he had taught the course. Mike used abstractions based on logic to introduce concepts such as query folding and query inversion. The contrast of the abstract methods used in Mike’s class to the ad hoc approaches I had seen used in industry previously was striking. It would take me many years before I had a full understanding of what Mike had taught in that class that year.

So many years later, the concepts taught in that class no longer seem foreign to me. So it is with the research laid out in this dissertation. In retrospect, it all seems simple and obvious. It is easy to forget that it took so many years of hard work to obtain such “obvious” results!

Throughout it all, Mike was my guide and mentor. I was struck early on by his brilliance and passion for his work. Through the years, he taught me how to do research that is not just publishable, but also practical. He taught me how to express my ideas simply and precisely.

I have been lucky to have a diligent reading committee. Vinay Chaudhri has gone through this dissertation with a fine-toothed comb. He has gone above and beyond the call of duty, and I cannot thank him enough. Hector Garcia-Molina has dedicated much time to my thesis committee as well, and has provided a much-needed database researcher’s point of view on my work.

Mark Musen and John Ousterhout sat on my oral committee and provided their

own unique perspectives on my work. They were both a pleasure to work with.

It would be difficult to finish a PhD without support at home. I cannot thank enough my wife Carla, who was at my side when I first found out the news that I had been admitted to the Stanford PhD program - a joyous event! I would not have been successful without her love and support. She encouraged me through the hard times when I got discouraged. She has always believed in me. And I am thrilled that my flexible PhD schedule has allowed me to spend much time with my wonderful son Noam, who was born during my final years as a PhD student.

Thank you to my parents, who supported me during my 25 or so years of schooling. Whether it was helping me with my science project or proofreading my college application, they have always put significant time into helping me with my studies. Due to their love and support I have always felt confident in my academic abilities. They have always encouraged me to follow my dreams and have helped me to achieve them.

My family has always been proud of me and has cheered me on through the years. My sister Lara, my grandparents, my aunts and uncles and extended family have always been supportive and excited to hear what I am up to.

Thank you to my friends and colleagues, who have made my journey enjoyable. In particular, thank you to the Logic Group members who provided much valuable feedback to me about my research throughout the years.

Preface

My work on logical spreadsheets began in 2002, when I joined Michael Genesereth as a research assistant during my master's degree. The idea of a logical spreadsheet - a spreadsheet with logical formulas instead of mathematical ones - was an idea that had been gestating in his mind since the introduction of spreadsheets. He had many such interesting ideas, but this was the one that resonated with me the most.

The name “logical spreadsheets” is somewhat misleading. A spreadsheet is a grid of cells, but logical spreadsheets need not be grid of cells. They can consist of an arbitrary set of cells, not necessarily laid out on a grid, that are related by logical formulae. Indeed, for several years we spent our efforts on logical spreadsheets that either extended traditional electronic spreadsheet systems like Microsoft Excel or were desktop applications inspired by traditional spreadsheets. At some point, we decided that our efforts were better spent on applying logical spreadsheet technology to forms, and in particular, Web forms. We decided to keep the same “logical spreadsheets” despite this as the name seemed to inspire an enthusiastic response from people and help them quickly understand what our technology did.

Contents

Abstract	iv
Acknowledgements	v
Preface	vii
1 Introduction	1
1.1 Applications	4
1.2 Example	5
1.3 Related Systems	8
1.3.1 Electronic Spreadsheets	8
1.3.2 Logical Spreadsheets	9
1.4 Contributions of this Thesis	13
1.5 Overview	15
2 Formal Background	17
2.1 Databases	17
2.2 Datalog Programs	18
2.3 Open Datalog Programs	21
2.4 Herbrand Logic	21
2.5 Resolution	23
3 Logical Spreadsheets	26
3.1 Formal Definitions	26

3.2	Update	27
3.2.1	Bilevel Update	27
3.2.2	Dealing with unambiguous conflicts during update	29
3.2.3	Dealing with ambiguities during update	31
3.2.4	Dealing with inconsistency in a static spreadsheet	32
3.2.5	Dealing with single values	33
3.2.6	Deletion	34
3.3	Summary	35
4	Existential Ω-Entailment	36
4.1	Introduction	36
4.2	Properties	38
4.3	Comparison to Resolution	41
4.4	Computing Existential Ω -Entailment	42
4.5	Related Work	46
4.5.1	Existential- Ω Entailment	46
4.5.2	Related Consequence Relations	47
5	Dynamics	49
5.1	Motivation	50
5.2	Markov Change Logic	52
5.3	Applications	55
5.3.1	Consistency Maintenance	55
5.3.2	View Maintenance	58
5.3.3	Updates Through Views	59
5.3.4	Database Dynamics	60
5.4	Converting Constraints to MCL	63
5.5	Dynamic Constraints	70
5.5.1	Types of database constraints	71
5.5.2	Formal definitions	72
5.5.3	The Markov Reformulation Theorem	74
5.6	Related Work	78

5.6.1	Event-Condition-Action Rules	79
5.6.2	Logic Programming Approaches	79
5.6.3	Abstract State Machines	81
5.6.4	Summary	83
6	Websheets	84
6.1	Webcell	84
6.1.1	Rule Syntax and Semantics	85
6.1.2	Structured cell names	88
6.2	Application: Program Sheets	89
6.2.1	Background	91
6.2.2	Implementation	92
6.3	Related Work	94
7	Conclusion	95
7.1	Numerical Constraints	95
7.2	Accessibility	96
7.3	Websheets with Large Datasets	100
7.4	Multiuser and Interlinked Spreadsheets	101
7.5	Applications	102
A	Epilogue	103
B	Existential Ω-Entailment Proofs	105
C	Abstract State Machines	110
C.1	Formal definition of ASMs	110
C.1.1	Static Algebras	110
C.1.2	ASM Updates	111
C.1.3	Sequential ASMs	112
C.1.4	Parallel ASMs	113
C.2	Proofs	115

List of Tables

1.1	Logical Spreadsheet Systems	14
4.1	Properties of Existential Ω -entailment	40
5.1	Database Physics Formalisms	83

List of Figures

1.1	Creating an event. (1) The user first sets the title for the event. (2) The user then sets the start time for the event to 1:00 pm. (3a) In one scenario, the user then sets the end time for the event to 3:00pm. The duration is then automatically filled in as 2 hours. (3b) In an alternate scenario, the user instead chooses the duration of the event to be 2 hours. The end time of the event is then automatically filled in as 3:00 pm.	3
1.2	Constraint schemas for the room manager. Schema variables are shown in boldface. Subscripts and dots have no semantic meaning and are simply used to name all of the spreadsheet cells in a consistent fashion. There are four schema variables - E , T , R , and P . E ranges over events, T ranges over times, R ranges over rooms, and P ranges over people. Each constraint schema represents a number of constraints, one for each grounding of the schema variables. For example, one grounding of the second constraint is $event.projection_{e_1}(yes) \wedge event.room_{e_1}(g100) \Rightarrow room.projector_{g100}(yes)$. The first constraint schema relates the schedule table to the event table, i.e. when an event is assigned to a room and a time in one table it must also be assigned to the same room and time in the other table. The second states that events that require a projector must be scheduled in a room with a projector. The third states that only faculty members can reserve room g100. Not shown are the single-value constraints, which declare that, for each cell p in the spreadsheet, $p(X) \wedge p(Y) \Rightarrow X = Y$	5

1.3	After creating three events	6
1.4	After scheduling e1	6
1.5	After scheduling e2	7
1.6	After moving e1 to the evening	7
1.7	Spreadsheet with inconsistency	8
1.8	Distinguishing between conflicts	8
1.9	Showing consequences under inconsistency	9
1.10	A completed spreadsheet	9
3.1	(1) A spreadsheet with two empty cells p and q , and the constraint $p(X) \Rightarrow q(X)$. (2) The user places the value a into cell p . The value a is automatically placed in cell q . (3a) The user removes value a from cell p . The value a is automatically removed from cell q . (3b) Alternatively, when the user removes value a from cell p , the value a remains in cell q	28
3.2	A spreadsheet with two cells, p and q , and the constraint $p(X) \Rightarrow \neg q(X)$. At first, cell q contains value a . The user then attempts to place value a into cell p . Three possible outcomes are depicted: at left, the attempt is rejected; at center, the update results in a conflict between $p(a)$ and $q(a)$; at right, the conflict is automatically resolved.	30
3.3	A spreadsheet with three cells, $d1$, $d2$, and $d3$ and a constraint stating that they cannot all have the value pe . At first, $d2$ and $d3$ contain the value pe . The user then places the values pe into cell $d1$. Four possible outcomes are displayed.	32
3.4	A spreadsheet with two constraints: $p_1(X) \wedge q_1(X) \Rightarrow r(X)$ and $p_2(X) \wedge q_2(X) \Rightarrow r(X)$. The value $q_2(b)$ is about to be inserted. After $q_2(b)$ is inserted, r will have two computed values - $r(a)$ and $r(b)$	34

5.1	Creating an event. (1) Currently the start time and end time are set by the user, and the duration is computed. (2a) Entering in “3 hours” for the event duration results in an inconsistency because the system is unable to determine whether to change the start time, the end time, or both. (2b) The state that we would like the spreadsheet to result in, in which the end time becomes computed rather than user-specified.	51
5.2	MCL being used by an update engine to update a database. An input update, the database, and an MCL program are fed into a Datalog engine. The output model of the MCL program is the output update. The output update is used by an update engine to update the database.	53
5.3	Creating an event. (1) Currently the start time and end time are set by the user, and the duration is computed. (2a) By default, entering in “3 hours” for the event duration results in an inconsistency because the system is unable to determine whether to change the start time, the end time, or both. (2b) Using a MCL program, we can instruct the spreadsheet change the end time, resulting in the above state. . .	56
5.4	An empty schedule	60
5.5	After scheduling e1	61
5.6	After changing the time to the afternoon	61
5.7	After scheduling e1	62
5.8	After changing the time to the afternoon	62
6.1	HTML for a simple Websheet	85
6.2	Bachus–Naur Form for Webcell rules	90
6.3	Foundations requirements for a Stanford Computer Science Master’s Program sheet. The Probability requirement is red as it is unfulfilled.	92

Chapter 1

Introduction

Computerized spreadsheets are a great success. They are often touted in newspapers and magazine articles as the first “killer app” for personal computers. Over the years, they have proven their worth time and again. Today, they are used for managing enterprises of all sorts - from one-person projects to multi-institutional conglomerates. Their applications range from financial planning to scientific data analysis to maintaining shopping lists.

The power of computerized spreadsheets derives in large part from two primary features - the automatic calculation of the values by spreadsheets and the use of mathematical formulas to specify those calculations.

The automatic calculation of values frees the user from the tedious task of doing those calculations manually. The automatic recalculation of values upon changes to the spreadsheet allows for easy “what if” analyses. Once the initial formulas are entered, exploring different scenarios is as simple as changing the initial parameters. This power is easy to take for granted, but consider that before electronic spreadsheets existed calculations were carried out on paper spreadsheets. If one wanted to change assumptions, or if an error was detected early in the computation, then all of the computation would have to be carried out again.

The support for mathematical formulas simplifies the task of setting up the calculations and makes spreadsheet technology accessible to a broad class of users, including those with no background in programming. This ease-of-use derives largely from

the fact that spreadsheets are programmed using familiar mathematical notation, such as $C1=A1+B1$. Furthermore, unlike a traditional programming environment, all of the intermediate computation steps are displayed for the programmer. Programming a spreadsheet is like being in “debug mode” all the time, where one can see all computation steps at once.

Despite their successes, computerized spreadsheet systems today have significant and unnecessary restrictions that limit their usefulness. One significant restriction is that the formulas used to specify calculations must be functions. Spreadsheet systems generally do not provide the ability to encode many-to-many relationships across cell values. For example, one cannot say that the values in cells A1 and A2 are different. Another unnecessary restriction is that propagation can only occur in one direction in a traditional spreadsheet. For example, if one defines $B2=B1$, then one can specify a value for B1 and B2 will obtain that same value, but one cannot then assign a value to B2 and expect B1 to obtain that value as well, even though the cells have equal values.

If these restrictions are lifted, then spreadsheets gain additional utility. Consider, for example, the following spreadsheet that allows for the creation of an event. This spreadsheet has four cells, which allow for the specification of the event’s name, start time, end time, and duration (Figure 1.1). The spreadsheet also has some formulae (not shown), which specify that “the end time must be after the start time” and that “the start time plus the duration equals the end time.” Note that the first formula is many-to-many and that the second formula determines each of the start time / end time / duration cells in terms of the other two. Thus, if the user were to specify 1:00 pm as the start time of the Logic Group Meeting (Figure 1.2) and 3:00 pm as the end time, then the duration can be automatically filled in as 2 hours (Figure 1.3a). Or, since the system can propagate values in any direction, the user could instead specify the start time to be 1:00 pm and the duration to be 2 hours and have the end time automatically filled in as 3:00 pm (Figure 1.3b). Similarly, the user could first fill in the duration to be 2 hours and the end time to be 3:00 pm, and have the start time automatically filled in as 1:00 pm (not shown).

Event Title	Logic Group Meeting
Start Time	
End Time	
Duration	

(1)

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	
Duration	

(2)

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	3:00 pm
Duration	2 hours

(3a)

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	3:00 pm
Duration	2 hours

(3b)

Figure 1.1: Creating an event. (1) The user first sets the title for the event. (2) The user then sets the start time for the event to 1:00 pm. (3a) In one scenario, the user then sets the end time for the event to 3:00pm. The duration is then automatically filled in as 2 hours. (3b) In an alternate scenario, the user instead chooses the duration of the event to be 2 hours. The end time of the event is then automatically filled in as 3:00 pm.

To allow for many-to-many relationships to be expressed between cells, we generalize the spreadsheet formula language from arithmetic function definitions to logical formulae. For example, we might formalize the constraint “the end time must be after the start time” by the rule $startTime(S) \wedge endTime(E) \implies after(E, S)$. A *logical spreadsheet* is a spreadsheet in which the formula language consists of logical formulae.

1.1 Applications

While traditional spreadsheets were built to perform mathematical calculations, logical spreadsheets are ideal for handling symbolic data. Some broad applications are as follows.

Smart Forms. A smart form is a logical spreadsheet with an HTML front end that allows users to fill out online forms in which data is checked for semantic well-formedness and fields are filled in automatically when possible. For example, consider an airline travel site that requires “the number of lap infants traveling must not be greater than the number of adults.”

Design and Configuration. For example, a configuration system to help consumers design their own cars with the constraint “if the car’s exterior color is blue, then the car interior color may be gray, tan or black.” Another example is of a student designing his course schedule which has the constraint “students must take at least 2 math courses to graduate.” A third example is of a room management system with the constraint “only senior managers can reserve the third floor conference room.”

Interactive Documents. Systems can return “interactive answers” to users, which allow a user to experiment by varying certain parameters while the system automatically propagates the consequences of those variations. For example, a logical spreadsheet used by an insurance agent to determine if a client is eligible for a particular kind of insurance. Such a spreadsheet could have rules like “insurance applicants who make at least \$60,000 and are under 50 years old are approved.” An interactive document allows one to perform the “what if” analyses that spreadsheets are

$ \begin{aligned} & \mathit{schedule}_{\mathbf{T},\mathbf{R}}(\mathbf{E}) \Leftrightarrow \mathit{event.time}_{\mathbf{E}}(\mathbf{T}) \wedge \mathit{event.room}_{\mathbf{E}}(\mathbf{R}) \\ & \mathit{event.projection}_{\mathbf{E}}(\mathit{yes}) \wedge \mathit{event.room}_{\mathbf{E}}(\mathbf{R}) \Rightarrow \mathit{room.projector}_{\mathbf{R}}(\mathit{yes}) \\ & \mathit{event.owner}_{\mathbf{E}}(\mathbf{P}) \wedge \mathit{person.faculty}_{\mathbf{P}}(\mathit{no}) \Rightarrow \neg \mathit{event.room}_{\mathbf{E}}(\mathit{g100}) \end{aligned} $
--

Figure 1.2: Constraint schemas for the room manager. Schema variables are shown in boldface. Subscripts and dots have no semantic meaning and are simply used to name all of the spreadsheet cells in a consistent fashion. There are four schema variables - \mathbf{E} , \mathbf{T} , \mathbf{R} , and \mathbf{P} . \mathbf{E} ranges over events, \mathbf{T} ranges over times, \mathbf{R} ranges over rooms, and \mathbf{P} ranges over people. Each constraint schema represents a number of constraints, one for each grounding of the schema variables. For example, one grounding of the second constraint is $\mathit{event.projection}_{e_1}(\mathit{yes}) \wedge \mathit{event.room}_{e_1}(\mathit{g100}) \Rightarrow \mathit{room.projector}_{g100}(\mathit{yes})$. The first constraint schema relates the schedule table to the event table, i.e. when an event is assigned to a room and a time in one table it must also assigned to the same room and time in the other table. The second states that events that require a projector must be scheduled in a rooms with a projector. The third states that only faculty members can reserve room g100. Not shown are the single-value constraints, which declare that, for each cell p in the spreadsheet, $p(X) \wedge p(Y) \Rightarrow X = Y$.

famous for, although there need not be a distinction between the cells used as input parameters and the cells used to output results.

1.2 Example

We illustrate our approach using a room management system as an example. The room manager consists of four tables, shown schematically in Figure 1.3. The top table contains event requests, each of which has an owner, a specification of whether a projector is needed, a room, and a time. The center table contains a schedule of the events. The information is redundant with the first table but is useful because it offers a different view. The bottom-left table lists whether or not each room has a projector. The bottom-right table lists whether each person is a faculty member or not. To distinguish base data from computed data, we will place a triangle in the upper left-hand corner of a cell containing base data.

This spreadsheet has several constraints, which are represented concisely in Figure 1.2 using constraint schemas. Each of these schemas represents a number of

event	owner	projection	room	time
e1	amy	no		
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning			
afternoon			
evening			

room	projector	person	faculty
g100	yes	amy	yes
g200	no	bob	no
g300	no	cal	yes

event	owner	projection	room	time
e1	amy	no	g100	morning
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning	e1		
afternoon			
evening			

room	projector	person	faculty
g100	yes	amy	yes
g200	no	bob	no
g300	no	cal	yes

Figure 1.3: After creating three events

Figure 1.4: After scheduling e1

constraints, one for each grounding (i.e. replacement of constants for variables) of the schema variables. The first of these constraint schemas relates the event table to the schedule table. The second states that events that require a projector must be scheduled in a room with a projector. The third states that only faculty members can reserve room g100.

The spreadsheet also has constraints that state that each of the cells can contain at most one value. For each cell p in the spreadsheet, there is a constraint $p(X) \wedge p(Y) \Rightarrow X = Y$. These constraints are generated automatically by the system.

We consider a room administrator whose task is to assign three new events a room and a time. The administrator starts with an empty schedule table and event table. She creates three new events in the event table and, for each, fills in the event owner's name and whether a projector is needed (Figure 1.3).

The administrator then selects g100 as the room for event e1 and morning as the time for e1 in the event table, causing e1 to show up in the corresponding cell in the schedule table (Figure 1.4). The administrator then assigns e2 to g200 in the afternoon by modifying the schedule table directly, causing the corresponding values to appear in the event table. This leads to the state shown in Figure 1.5. This illustrates our system's ability to do propagation in multiple directions.

event	owner	projection	room	time
e1	amy	no	g100	morning
e2	bob	no	g200	afternoon
e3	cal	yes		

schedule	g100	g200	g300
morning	e1		
afternoon		e2	
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 1.5: After scheduling e2

event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g200	afternoon
e3	cal	yes		

schedule	g100	g200	g300
morning			
afternoon		e2	
evening	e1		

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 1.6: After moving e1 to the evening

Next, the administrator assigns e1 to g100 in the evening by modifying the schedule table, which directly conflicts with its current assignment to g100 in the morning. This results in e1’s time being changed to the evening in the event table and being removed from the morning slot in the schedule table (Figure 1.6). This illustrates how the update algorithm deals with direct conflicts.

The administrator then sets the room assignment for e3 to g200. Since e3 requires a projector but g200 lacks a projector, this leads to a conflict. As shown in Figure 1.7, our system marks the conflicting cells in red. Note that in this case, a conflict between multiple cells, there is no automatic resolution of the conflict, in contrast to when the conflict is between two cells. Chapter 3 explains why this is so.

The administrator does not have to resolve the conflict immediately. She instead proceeds to change e2’s room to g100. This leads to yet another conflict, since Bob is not a faculty member and only faculty members can reserve g100. At this point, there are six cells colored red. So that the user can distinguish between the different sources of conflict, the values involved in the conflict are highlighted when the mouse pointer is placed over a conflicted value (Figure 1.8).

The administrator next sets the time of e3 to the morning. The event e3 then

event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g200	afternoon
e3	cal	yes	g200	

schedule	g100	g200	g300
morning			
afternoon		e2	
evening	e1		

room	projector	person	faculty
g100	yes	amy	yes
g200	no	bob	no
g300	no	cal	yes

event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g100	
e3	cal	yes	g200	

schedule	g100	g200	g300
morning			
afternoon			
evening	e1		

room	projector	person	faculty
g100	yes	amy	yes
g200	no	bob	no
g300	no	cal	yes

Figure 1.7: Spreadsheet with inconsistency

Figure 1.8: Distinguishing between conflicts

appears in the schedule table, as shown in Figure 1.9. Finally, the administrator moves the projector from g100 into g200 (by setting the g200 projector cell to yes and the g100 projector cell to no) and places e2 in g200 in the afternoon, removing the conflicts and resulting in a complete assignment to all events (Figure 1.10).

1.3 Related Systems

To place our work in its historical context, we review past work on logical spreadsheets and explain how this thesis diverges from previous systems. Our presentation is adapted from [34].

1.3.1 Electronic Spreadsheets

The world's first electronic spreadsheet, VisiCalc, was created by Dan Bricklin and Bob Frankston in 1979. The spreadsheet was widely regarded as the “killer app” for personal computers. It managed to hit a sweet spot between usability and functionality - millions of users with no formal training in programming were suddenly enabled to create custom applications of their own.

event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g100	
e3	cal	yes	g200	morning

schedule	g100	g200	g300
morning		e3	
afternoon			
evening	e1		

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 1.9: Showing consequences under inconsistency

event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g200	afternoon
e3	cal	yes	g200	morning

schedule	g100	g200	g300
morning		e3	
afternoon		e2	
evening	e1		

room	projector
g100	no
g200	yes
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 1.10: A completed spreadsheet

Today, the spreadsheet remains as popular as ever. In 2001 there were an estimated 45 million end users of spreadsheets or databases in the United States alone, representing 60% of the American workforce [49], and this number is rising.

Spreadsheets do one thing and they do it well, which is to perform mathematical computations. While spreadsheets do have built-in Boolean functions and conditional functions, it is clear that they were not designed to support logical reasoning. In modern commercial spreadsheets, exemplified by Microsoft Excel, logical functions have an inelegant syntax and editing tools. Worse, the logical reasoning allowed is quite limited - there are no unification routines, negation-as-failure capabilities, etc.

1.3.2 Logical Spreadsheets

LogiCalc [37] was the first logical spreadsheet. It was developed by Frank Kriwazek at Imperial College in the early 1980's as part of his Master's thesis work. In LogiCalc, spreadsheet formulas were written using traditional spreadsheet syntax, but are translated internally to Prolog rules. For example, if C2 is defined as B2 + B3, then the following rule would be created:

```

has-definition("C2",X)
  if has-value("B2", Y)
  and has-value("B7", Z)
  and X = Y + Z

```

LogiCalc allowed regions of the spreadsheet to be saved as database relations and then queried. The queries were restricted to be conjunctive, for example:

```

which((Y,Z): father-of("Bob", Y) and friend-of(Y,Z))

```

The user could either display all solutions to a query by filling a column of the spreadsheet or by displaying a single solution to the query in a cell and then cycling through the answers. Tuples were held in a single cell, as opposed to displaying each component of a tuple in a separate column.

LogiCalc also allowed database tables to define functions. For example, given facts of the form:

```

father-of("Bob", "Dan")
father-of("Art", "Cal")
father-of("Art", "Coe")

```

One could define a new function, `@father-of`, that mapped children to their fathers. These functions can be incorporated into spreadsheet formulae, such as:

```

A3 = @age-of(@father-of(B3))

```

The idea of using a spreadsheet as an interactive constraint solver dates back to LogiCalc. Users could specify constraints over cells, for example:

```

profession-of(A1, "barber")
father-of(A1,A2)

```

```
father-of(A2,A3)
father-of(A3,A4)
```

Given no assignments to the cells **A1-A4**, LogiCalc would fill the cells with values that satisfied the constraints. The user could then choose to cycle through all such satisfying combinations of values, or he could stick with one or more of the values and cycle through the rest. The user also had the ability to declare that a cell either have some given value or that it is not allowed to take a given value.

This interactive constraint solving capability was improved upon by van Emden et al. [54], whose system allowed for previous computation to be reused when a adnew constraint was incrementally added.

The constraints allowed in LogiCalc and van Emden's system were quite simple by Prolog standards. They consisted of a single conjunction of conditions, which is what can be expressed by a single Prolog rule. However, Prolog itself allows for a set of rules to be defined (which allows for disjunctive conditions), and furthermore, the rules may be recursive.

PERPLEX [51], allowed for new constraints to be defined in terms of old ones, using Prolog rules. Like its predecessors, PERPLEX allowed constraints to be defined using database relations. It also allowed for constraints based on built-in predicates, which could handle operations like arithmetic, string concatenation, etc. Each built-in predicate had a set of legal input-output modes, which identified each argument of the predicate as either accepting input or producing output. For example, the `add` predicate (where `add(A1 A2 A3)` means that $A1+A2=A3$), had four input-output modes, namely (in in out), (in out in), (out in in), and (in in in). Constraint propagation was done by finding a constraint with all the input parameters for one input-output mode supplied, computing the outputs, and repeating. For user-defined predicates specified using Prolog rules, the set of legal input-output modes was automatically computed.

PERPLEX's handling of numerical constraints was simple but incomplete. For example, consider the constraint `+(A1 A1 2)`. Though $A1$ must equal 1, PERPLEX cannot determine this. Something more sophisticated than input-output modes is

required to solve this constraint, since adding the input-output mode (out out in) will also trigger propagation when $+(A1 \ B1 \ 2)$ is given, which is not solvable. Similarly, consider constraints $A2 \geq 2$ and $A2 \leq 2$. In this case, it must be that $A2 = 2$, but the constraints must be considered together to determine this, which PERPLEX does not do.

A generalization of logic programming called *constraint logic programming* allows for sophisticated numerical and symbolic constraint solving. While the most basic operation in logic programming is unification, in constraint logic programming, unification is replaced by constraint propagation, of which unification is a special case. There are many flavors of constraint propagation; in particular there are versions which can handle linear constraints over real numbers, or over finite domains.

Knowledgesheet [26] is a logical spreadsheet with a finite domain constraint logic programming engine as its constraint solver, developed in the late 1990's. In Knowledgesheet, the user assigns constraints over the cells, and associates each cell with a finite domain. The user may also assign values to cells. Once satisfied with the constraints, the user presses a "solve" button, at which time the constraints are compiled into a constraint logic program and solved, resulting in the rest of the cells being assigned values if a solution exists. The Knowledgesheet approach is different than the previously described systems in that propagation does not occur automatically as values are assigned to cells and constraints are created and removed. This behavior is desirable for large problems which may take minutes or longer to solve.

A logical spreadsheet system with similar capabilities to Knowledgesheet is CSSOLVER [21]. As opposed to using a logic programming engine to solve constraints, however, a constraint programming engine is used. In addition to finding a solution that simply satisfies the constraints, CSSOLVER also allows the user to specify an optimization function and find an optimal solution.

There has been a recent flurry in activity around extending Microsoft Excel with logical capabilities. NEXCEL [11], LESS [53], XcelLog [46], and the unnamed system of Waltzman *et al.* [52]. In all four of these systems, the tabular format, the immediate feedback that users receive when changing a value or a formula, and the familiar graphical user interface elements of current commercial spreadsheet systems are all

preserved. Furthermore, in all cases the formula language is a conservative extension of the traditional language that users have come to expect from spreadsheet systems. Finally, in all cases the behavior of the spreadsheet is a simple extension of current behavior, namely that the formulas partition the cells into input cells and output cells, where the output cells are a function of the input cells.

The important difference between these previous systems and our logical spreadsheets is the way in which updates are performed. In particular, previous systems do not allow for the spreadsheet data to be inconsistent with the constraints. This is disallowed either implicitly (as in systems that do not allow for many-to-many constraints but only definitional formulae), or explicitly (as in systems that will reject updates that will result in an inconsistency). In addition, our logical spreadsheets allow for the administrator to modify the way that logical spreadsheets are updated via a declarative update language, a feature which is not found in previous logical spreadsheet systems.

A summary of the different logical spreadsheet systems is given in Table 1.1.

1.4 Contributions of this Thesis

This thesis is concerned with the issues surrounding the update of logical spreadsheets. As logical spreadsheets are special cases of databases, the methods presented here are often applicable to databases as well, although the single-valued nature of spreadsheets at times simplifies and at times complicates matters compared to the general database case.

We begin by introducing a domain-independent update semantics for logical spreadsheets, which is determined by the constraints. Unlike previous approaches to database updates, our update semantics allows for inconsistency between the values in the spreadsheet and the constraints. Due to this inconsistency, we are not able to use traditional logical entailment to determine the consequences of the spreadsheet values. Thus, we introduce a new paraconsistent entailment relation called existential Ω -entailment that we use in place of traditional logical entailment. We give an algorithm for computing the existentially Ω -entailed values in our spreadsheet.

Table 1.1: Logical Spreadsheet Systems

Spreadsheet	Mutidirectional Update	Many-to-Many Constraints	Innovative Feature	Year
LogiCalc [37]	x	x	Logical rules and constraints	1985
van Emden et al. [54]	x	x	Less recomputation	1986
PERPLEX [51]	x	x	Numerical Constraints	1989
Knowledgesheet [26]	x	x	Constraint Logic Programming Solver	2000
NEXCEL [11]			Recursively defined views	2007
LESS [53]			PowerLoom [12] integration	2007
XcelLog [46]			Multi-valued cells	2007
Waltzman <i>et al.</i> [52]			Triple-based reasoning	2007

It is sometimes desirable to modify the domain-independent update semantics to accommodate some domain-specific behaviors. To achieve this, we introduce a logic called Markov Change Logic that allows the spreadsheet administrator to specify a complete description of the dynamic behavior of a spreadsheet or database. As its name suggests, Markov Change Logic specifies database dynamics in terms of adjacent pairs of database states. This begs the question of whether a Markov language is sufficient for specifying multistep dynamic constraints. We give a theoretical analysis that shows that the answer is yes, as long the database schema may be extended with additional relations. Finally, we show how to convert a set of constraints into its equivalent Markov Change Logic program.

Finally, we illustrate the use of logical spreadsheets by describing Webcell, a Web-based logical spreadsheet engine. Webcell powers the Stanford Computer Science Master's Program sheet, which is currently used by students to plan their course schedules and ensure that all departmental and university requirements are met.

1.5 Overview

The rest of this thesis is organized as follows.

- Chapter 2 gives the formal background required for the thesis.
- Chapter 3 formally defines logical spreadsheets, and introduces a domain-independent update semantics for logical spreadsheets.
- Chapter 4 discusses our paraconsistent entailment relation, existential Ω -entailment, which is used to show the consequences of the values in a spreadsheet even when there is inconsistency between the spreadsheet values and constraints. We explore the formal properties of the entailment relation and give an algorithm for computing the existentially Ω -entailed values in a database.
- Chapter 5 motivates the need to modify the domain-independent update semantics by using a specification language for database dynamics. We consider whether general dynamic rules are required and conclude that, as long as the

spreadsheet schema can be extended, Markov rules suffice. Finally, we introduce Markov Change Logic, which allows us to express dynamic policies, and describe how it can be used to modify the domain-independent update semantics.

- Chapter 6 describes in detail the implementation of a logical spreadsheet engine called Webcell that can be used to turn Web pages into logical spreadsheets. We describe the application of Webcell to the Stanford University Computer Science Master's Program Sheets.
- Chapter 7 concludes with a discussion of future work.

Chapter 2

Formal Background

In this chapter, we give the formal background assumed by the remainder of the thesis. We begin by defining databases and Datalog, a query language for databases. Our presentation is largely taken from [24]. We then define Herbrand Logic, a logic which we will use to define constraints for our spreadsheets. Finally, we define resolution, a proof procedure which can be used to prove facts in Herbrand Logic.

2.1 Databases

The fundamental building blocks of databases are entities and relations. *Entities* represent objects presumed or hypothesized to exist in the application area of the database. *Relations* represent properties of those objects or relationships among them. A collection of entities and relations together with their arities is called a *vocabulary*.

In our examples here, we refer to entities and relations using strings of letters, digits, and a few non-alphanumeric characters (e.g. “_”). For reasons described below, we prohibit strings beginning with upper case letters; all other combinations are acceptable. Examples include a , p , 123, *father*, *comp225*, and *helen_heavenly*. In abstract examples, we will usually represent entities as letters early in the alphabet (e.g. a , b , c) and relations as letters later in the alphabet (e.g. p , q , r).

The set of all entities that can be used in a database is called the *domain* of the

database. Database domains can be finite or infinite. Most (but not all) database domains include strings and numbers as subsets, and hence they are infinite.

The set of all relations in a database is called the *signature* of the database. Signatures are always finite.

The *arity* of a relation is the number of entities involved in any instance of that relation. Arity is an inherent property of a relation and never changes. If r is a relation, we write $r(X_1, \dots, X_k)$ to signify that r has arity k . We denote the arity of a r as $arity(r)$.

A *database schema* consists of a domain, a signature, and an assignment of arities for each of the relations in the signature.

Given a database, we define a *datum* (plural *data*) to be a structure consisting of an n -ary relation from the signature and n entities from the domain. On occasion, we call a datum a *proposition*. In what follows, we write data using traditional mathematical notation. For example, if r is a binary relation and a and b are entities, then $r(a, b)$ is a datum / proposition.

The *propositional base* for a database schema is the set of all propositions that can be formed from the relations and the entities in the database schema. For a schema with entities a and b and relations p and q where p has arity 1 and q has arity 2, the propositional base is $\{p(a), p(b), q(a, a), q(a, b), q(b, a), q(b, b)\}$.

A *database instance* (also known as an *extension* or *database state*) of a database is a finite subset of its propositional base.

The *cardinality* of a relation r in a database instance is the number of data in that state with relation of r .

2.2 Datalog Programs

Datalog programs are used to define views (i.e. named queries) of databases.

Datalog programs are built up from three disjoint classes of components, viz. relations, entities, and variables. In our examples here, we denote entities and relations as described above; and we write variables as strings of letters, digits, and special characters beginning with an upper case letter, e.g. $X, Y, Z, Age, F1, F2$, and so

forth. (Recall that entities and relations start with lowercase letters and numbers.) A *term* is either an entity or a variable.

An *atom* is an expression formed from an n -ary relation and n terms. As with data, we write Datalog atoms in traditional mathematical notation - the relation followed by its arguments enclosed in parentheses and separated by commas. For example, if r is a binary relation, if a and b are entities, and if Y is a variable, then $r(a, b)$ and $r(b, a)$ are atoms, as are $r(a, Y)$ and $r(Y, a)$ and $r(Y, Y)$.

A *literal* is either an atom or a negation of an atom. An atom is called a *positive* literal. The negation of an atom is called a *negative* literal. We write negative literals using the negation sign \neg . For example, if $r(a, b)$ is an atom, then $\neg r(a, b)$ denotes the negation of this atom. A *fact* is a datum or a negative datum.

A *rule* is an expression consisting of a distinguished atom, called the *head*, and 0 or more literals, together called the *body*. We write rules as in the example shown below. Here, $q(X, Y)$ is the head, and the other literals constitute the body.

$$q(a, Y) : - p(b, Y) \ \& \ \neg r(Y, d)$$

A *Datalog program* is a finite set of atoms and rules of this form. To simplify our definitions and analysis, we occasionally talk about infinite sets of rules. While these sets are useful for purposes of analysis, they are not themselves Datalog programs.

An expression is said to be *ground* if and only if it contains no variables.

A rule in a Datalog program is *safe* if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body. A Datalog program is safe if and only if every rule in the program is safe.

The *dependency graph* for a Datalog program is a directed graph in which the nodes are the relations in the program and in which there is an arc from one node to another if and only if the former node appears in the body of a rule in which the latter node appears in the head. A program is *recursive* if and only if there is a cycle in the dependency graph.

A negation in a Datalog program is said to be *stratified* if and only if there is no

cycle in the dependency graph involving the negation. A Datalog program is stratified if and only if there are no unstratified negations.

In this thesis, we concentrate exclusively on Datalog programs that are both safe and stratified. While it is possible to extend the results here to other programs, such extensions are beyond the scope of this work.

The *propositional base* for a Datalog program is the set of all atoms that can be formed from the entities in the program's schema. Said another way, it is the set of all sentences of the form $r(t_1, \dots, t_n)$, where r is an n -ary relation and t_1, \dots, t_n are entities.

An *instance* of a rule in a Datalog program is a rule in which all variables have been consistently replaced by entities from the program's domain. *Consistent replacement* means that, if one occurrence of a variable is replaced by a given term in a rule, then all occurrences of that variable in the rule are replaced by the same term.

An *interpretation* for a Datalog program is an arbitrary subset of the propositional base for the program. A *model* of a Datalog program is an interpretation that satisfies the program (as defined below).

An interpretation D *satisfies* a Datalog program P if and only if D satisfies every ground instance of every sentence in P . The notion of satisfaction is defined recursively. An interpretation D satisfies a ground atom p if and only if p is in D . D satisfies a ground negation $\neg p$ if and only if p is not in D . D satisfies a ground rule $p : - p_1, \dots, p_n$ if and only if D satisfies p whenever it satisfies p_1, \dots, p_n .

In general, a Datalog program can have more than one model, which means that there can be more than one way to satisfy the rules in the program. In order to eliminate ambiguity, we adopt the minimal model approach to Datalog program semantics, i.e. we define the meaning of a safe and stratified Datalog program to be its minimal model.

A model D of a Datalog program P is *minimal* if and only if no proper subset of D is a model for P . A Datalog program that does not contain any negations has one and only one minimal model. A Datalog program with negation may have more than one minimal model; however, if the program is stratified, then once again there is only one minimal model.

2.3 Open Datalog Programs

Datalog programs as just defined are *closed* in that they fix the meaning of all relations in the program. In open Datalog programs, some of the relations (the inputs) are undefined, and other relations (the outputs) are defined in terms of these. The same program can be used with different input relations, yielding different output relations in each case.

Formally, an *open program* is a Datalog program together with a partition of the relations into two types - *base relations* (also called *input relations*) and *view relations* (also called *output relations*). View relations can appear anywhere in the program, but base relations can appear only in the bodies of rules, not in their heads.

The *input base* for an open Datalog program is the set of all atoms that can be formed from the base relations of the program and the entities in the program's domain. An *input model* is an arbitrary subset of its input base.

The *output base* for an open Datalog program is the set of all atoms that can be formed from the view relations of the program and the entities in the program's domain. An *output model* is an arbitrary subset of its output base.

Given an open Datalog program P and an input model D , we define the overall model corresponding to D to be the minimal model of $P \cup D$. The *output model* corresponding to D is the intersection of the overall model with the program's output base.

Finally, we define the meaning of an open Datalog program to be a function that maps each input model for the program into the corresponding output model.

2.4 Herbrand Logic

The logic used in this thesis, Herbrand Logic, is first-order logic with equality and the following two restrictions.

- no function constants
- unique names assumption

The unique names assumption (UNA) states that every pair of distinct entities is unequal. The domain closure assumption (DCA) states that every object in the domain must be one of the entities in the vocabulary.

We use standard definitions for the syntax of Herbrand Logic, adapted from [29]. *Logical sentences* are defined inductively: an atom is a logical sentence, and if ϕ and ψ are sentences and X is a variable then all of the following are sentences: $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, $\phi \Leftarrow \psi$, $\phi \Leftrightarrow \psi$, $\forall X.\phi$, and $\exists X.\phi$. Variables not captured by a quantifier are called *free variables*. Sentences with free variables are called *open sentences*, and sentences without free variables are called *closed sentences*. The set of all sentences for a given vocabulary is called the *language* for that vocabulary. A *theory* is a set of sentences.

A *clause* is a set of literals and represents a disjunction of the literals in the set. A theory is in *clausal form* if it consists solely of clauses. A *contrapositive* of a clause $\{\phi_1, \dots, \phi_k\}$ is a sentence of the form $\phi_i \Leftarrow \neg\phi_1 \wedge \dots \wedge \neg\phi_{i-1} \wedge \neg\phi_{i+1} \wedge \dots \wedge \neg\phi_k$.

We use fairly standard metalevel notation. A lower-case Greek letter denotes a single sentence, and an upper-case Greek letter denotes a finite set of sentences. Instead of writing out n variables as X_1, \dots, X_n , we will often write \bar{X} ; likewise entities a_1, \dots, a_n will be written \bar{a} .

We use the standard definitions for a model and for satisfaction but take advantage of the UNA to simplify those definitions. A *interpretation* in Herbrand Logic is a set of ground atoms from the language. *Satisfaction* is defined as follows.

The definition for the satisfaction of closed sentences where the interpretation M is represented as a set of ground atoms is as follows.

$\models_M s = t$ if and only if s and t are syntactically identical.

$\models_M p(t_1, \dots, t_n)$ if and only if $p(t_1, \dots, t_n) \in M$

$\models_M \neg\Phi$ if and only if $\not\models_M \Phi$

$\models_M \Phi \wedge \Psi$ if and only if $\models_M \Phi$ and $\models_M \Psi$

$\models_M \Phi \vee \Psi$ if and only if $\models_M \Phi$ or $\models_M \Psi$ or both

$\models_M \Phi \Leftarrow \Psi$ if and only if $\models_M \Phi \vee \neg\Psi$

$\models_M \Phi \Rightarrow \Psi$ if and only if $\models_M \neg\Phi \vee \Psi$

$\models_M \Phi \Leftrightarrow \Psi$ if and only if $\models_M \Phi \Leftarrow \Psi$ and $\models_M \Phi \Rightarrow \Psi$

$\models_M \forall X.\Phi(X)$ if and only if $\models_M \Phi(a)$ for every entity a .

$\models_M \exists X.\Phi(X)$ if and only if $\models_M \Phi(a)$ for some entity a .

An open sentence $\Phi(X_1, \dots, X_n)$ with free variables X_1, \dots, X_n is *satisfied by* M if and only if $\forall x_1 \dots x_n.\Phi(x_1, \dots, x_n)$ is satisfied by M according to the above definition.

A set of sentences is *satisfiable* (or *consistent*) when there is at least one model.

Logical entailment is a *consequence relation* - a relation between theories and sentences. If \approx is a consequence relation and $\Phi \approx \psi$, then $\phi \in \Phi$ are known as *premises* and ψ is known as the *conclusion*.

Entailment queries in Herbrand Logic can be answered using the usual reduction of entailment to unsatisfiability: $\Delta \models \Phi$ if and only if $\Delta \cup \neg\Phi$ is unsatisfiable [29].

2.5 Resolution

Resolution is an automated proof procedure for first-order logic. In this section, we cover the basics of resolution. Our exposition is a condensed version of [23]. We first cover substitutions and unification, upon which resolution is built.

A *substitution* is a finite mapping of variables to terms. We write substitutions as sets of replacement rules, like the one shown below. In each rule, the variable to which the arrow is pointing is to be replaced by the term from which the arrow is pointing. In this case, X is to be replaced by a , Y is to be replaced by b , and Z is to be replaced by V .

$$\{X \leftarrow a, Y \leftarrow b, Z \leftarrow V\}$$

The variables being replaced together constitute the *domain* of the substitution, and the terms replacing them constitute the *range*.

The result of applying a substitution σ to an expression Φ is the expression $\Phi\sigma$ obtained from the original expression by replacing every occurrence of every variable in the domain of the substitution by the term with which it is associated.

A substitution σ and a substitution τ are *composable* if and only if the domain of σ and the range of τ are disjoint. Otherwise, they are *noncomposable*.

The *composition* $\sigma\tau$ of a substitution σ and a substitution τ is computed in two steps. (1) First, we apply τ to the range of σ . (2) Then we adjoin to σ all pairs from τ with different domain variables.

A clause Φ *subsumes* a clause Ψ if and only if there exists a substitution σ such that $\Phi\sigma \subseteq \Psi$.

A substitution σ is a *unifier* for an expression Φ and an expression Ψ if and only if $\Phi\sigma = \Psi\sigma$, i.e. the result of applying σ to Φ is the same as the result of applying σ to Ψ . If two expressions have a unifier, they are said to be *unifiable*. Otherwise, they are *nonunifiable*.

We say that a substitution σ is *as general as* or *more general than* a substitution τ if and only if there is another substitution δ such that $\sigma\delta = \tau$. A *most general unifier*, or *mgu*, σ of two expressions has the property that it is as general as or more general than any other unifier.

If a subset of the literals in a clause Φ has a most general unifier γ , then the clause Φ' obtained by applying γ to Φ is called a *factor* of Φ .

Suppose that Φ and Ψ are two clauses. If there is a literal ϕ in some factor Φ' of Φ and a literal $\neg\psi$ in some factor Ψ' of Ψ such that ϕ and ψ have a most general unifier γ , then we say that the two clauses Φ and Ψ *resolve* and that the new clause $((\Phi' - \{\phi\}) \cup (\Psi' - \{\neg\psi\}))\gamma$ is a *resolvent* of the two clauses.

A *resolution derivation* of a clause ϕ from a set Δ of clauses is a sequence of clauses terminating in ϕ in which each item is either (1) a member of Δ (a *premise*) or (2) the result of applying resolution to earlier items in the sequence. A sentence ϕ is *provable* from a set of sentences Δ by resolution if and only if there is a derivation of the empty clause (called a *resolution refutation*) from the clausal form of $\Delta \cup \{\neg\phi\}$.

The *resolution closure* of a set Δ of clauses is the set of all clauses resolution derivable from Δ . The *resolution closure with subsumption* of a set Δ of clauses is

resolution closure of Δ , minus those clauses δ subsumed by some other clause δ' in the resolution closure of Δ .

Chapter 3

Logical Spreadsheets

In this chapter, we define logical spreadsheets and introduce a domain-independent update semantics for logical spreadsheets. We illustrate the update semantics with a detailed example. Finally, we compare logical spreadsheets to similar systems.

3.1 Formal Definitions

We represent logical spreadsheets as databases of a very special kind. Define a *cell* to be a unary relation of cardinality 0 or 1. A *spreadsheet* is a database in which all relations are cells. For example, if we have a spreadsheet with three cells, p , q , and r , then an instance of the spreadsheet is $\{p(a), r(b)\}$. A *logical spreadsheet* is a spreadsheet with a set of logical *constraints* which represent the relationships between cells. We use Herbrand Logic sentences in clausal form to represent constraints.

For example, the following constraint disallows cells p and q from being equal:

$$\neg p(X) \vee \neg q(X)$$

We will sometimes write constraints in rule form to increase readability. For example, instead of:

$$\neg p(X) \vee \neg q(X) \vee r(X)$$

we might write the logically equivalent rule:

$$p(X) \wedge q(X) \Rightarrow r(X)$$

For each cell p , a logical spreadsheet has a *single-value constraint* $p(X) \wedge p(Y) \Rightarrow X = Y$ which asserts that the relations contain at most 1 value. As every spreadsheet has them, we will usually not explicitly write out these constraints.

Note that this definition of a logical spreadsheet is equivalent to the definition of a constraint satisfaction problem (CSP) [48]. Logical spreadsheets can be thought of as an interactive constraint satisfaction problem [45]. Whereas a traditional CSP is solved completely automatically, in an interactive CSP the user and system cooperate to produce a solution.

3.2 Update

In this section, we introduce a domain-independent semantics for updating logical spreadsheets and discuss the decisions we made when designing the update mechanism. As we shall see, the generalization from unidirectional functions to many-to-many relationships leads to many choices.

3.2.1 Bilevel Update

In a traditional spreadsheet, the formulae partition the cells into *input cells* and *output cells*, where the output cells are functions of the input cells. In a logical spreadsheet, this is no longer the case: regardless of the constraints, any cell can be an input cell or an output cell.

Although a logical spreadsheet has this additional functionality, it should still behave in the same way that a traditional spreadsheet would behave in the case that this additional functionality is not used. Consider a simple spreadsheet with two cells, p and q , together with the constraint $p(X) \Rightarrow q(X)$, shown pictorially in Figure

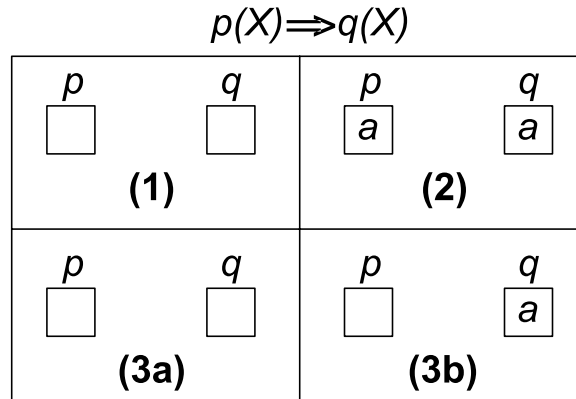


Figure 3.1: (1) A spreadsheet with two empty cells p and q , and the constraint $p(X) \Rightarrow q(X)$. (2) The user places the value a into cell p . The value a is automatically placed in cell q . (3a) The user removes value a from cell p . The value a is automatically removed from cell q . (3b) Alternatively, when the user removes value a from cell p , the value a remains in cell q .

3.1(1). If the user were to place value a into cell p , then the constraints tell us that q must contain value a as well (Figure 3.1(2)). What should happen, then, if the value a is then removed from cell p ? In a traditional spreadsheet, a would automatically be removed from cell q as well, since it is no longer computable by the input cell, namely p , and the constraints (Figure 3.1(3a)). The user would be surprised if the value a were to remain in cell q (Figure 3.1(3b)), because this value is no longer supported by any other input values in the spreadsheet.

To support this behavior, the spreadsheet must keep track of which values were supplied by the user and which values were not. We shall therefore separate the values into *base* values and *computed* values, where the base values are supplied by the user, and the computed values are entailed by the base values and the constraints. Thus, in the example above, $p(a)$ would be a base value assignment and $q(a)$ would be a computed value assignment. Because we have two “levels” of facts - the computed level is built upon the base level - we say that our spreadsheets are *bilevel*, and also that the update algorithm that maintains the relationship between the base and computed cells is *bilevel*.

In the sequel, we will represent the set of base value assignments by Λ , the computed value assignments by Γ , and the constraints by Ω . The computed value assignments Γ are entailed by Λ and Ω .

3.2.2 Dealing with unambiguous conflicts during update

In a traditional spreadsheet, the formulas partition the cells into base cells and computed cells. Computed cells contain a formula, which defines its value in terms of other cells. Base cells are cells that do not contain formulas. The formulas induce a partial ordering on the cells, and updates to cells lower in the ordering can only affect cells higher in the ordering. Thus, we say the traditional spreadsheets only support *unidirectional update*. On the other hand, in a logical spreadsheets, no such ordering is possible, as an update to cell p can change the value to cell q and vice versa, even when the constraints are fixed. Thus we say that logical spreadsheets support *multidirectional update*.

An interesting possibility raised by multidirectional update is how to deal with the case that the value being entered into the spreadsheet conflicts with the values currently in the spreadsheet. For example, consider a spreadsheet with two cells, p and q , and one constraint, $p(X) \Rightarrow \neg q(X)$. The user first places value a into cell q , as depicted at the top of Figure 3.2. What should happen if the user then attempts to place value a into cell p , directly conflicting with value a in cell q ? We consider three possibilities, shown at the bottom of Figure 4.

1. The request is rejected by the system.
2. The request is executed by the system, resulting in a conflict between cells p and q .
3. The request is executed by the system, and in addition, the value a is automatically removed from cell q , thus resolving the conflict.

In a traditional spreadsheet, when a user places a value into a cell, the result is that (1) the cell has that value, and (2) all consequences of that value are propagated

$$\begin{aligned}
& \Lambda \oplus \lambda \\
&= \Lambda \cup \lambda - \{\psi \in \Lambda \mid \lambda \cup \Omega \models \neg\psi\} \\
&= \{q(a)\} \cup \{p(a)\} - \{\psi \in \{q(a)\} \mid \{p(a)\} \cup \{p(X) \Rightarrow \neg q(X)\} \models \neg\psi\} \\
&= \{q(a)\} \cup \{p(a)\} - \{q(a)\} \\
&= \{p(a)\}
\end{aligned}$$

3.2.3 Dealing with ambiguities during update

Things are not always so simple. It may be the case that a new value placed into the spreadsheet does not directly conflict with any value currently in the spreadsheet, but instead conflicts with some combination of values.

For example, consider that we have a spreadsheet with three cells, $d1$, $d2$, and $d3$, representing the departments of the courses a student is taking. Suppose that the spreadsheet also contains a constraint stating that “a student cannot take three physical education classes”, written as $\neg d1(pe) \vee \neg d2(pe) \vee \neg d3(pe)$. So, what should happen when two of the courses are in the physical education department, and the student attempts to take a third course in the physical education department? Figure 3.3 displays four alternatives.

We would like to obey the user’s request, and at the same time, we would like to resolve the conflict as we did in Section 3.2.2. There are three possible ways to resolve the conflict, namely to remove pe from cell $d2$, to remove pe from cell $d3$, and to remove pe from both $d2$ and $d3$. The system has no reason to prefer either $d2$ or $d3$ over the other. Furthermore, while the third option is fair, it deletes information when it is unclear that the information should be deleted. While this example is small, with only three cells, a larger example with, say, 100 cells and a similar constraint would result in 99 deletions, a completely unacceptable loss of information. Thus, we choose the final option: place pe into $d1$, and leave the other cells alone.

Note that Definition 3.1 does exactly this, with no modification. Definition 3.1 removes only those facts that directly conflict with the inserted facts, and nothing else.

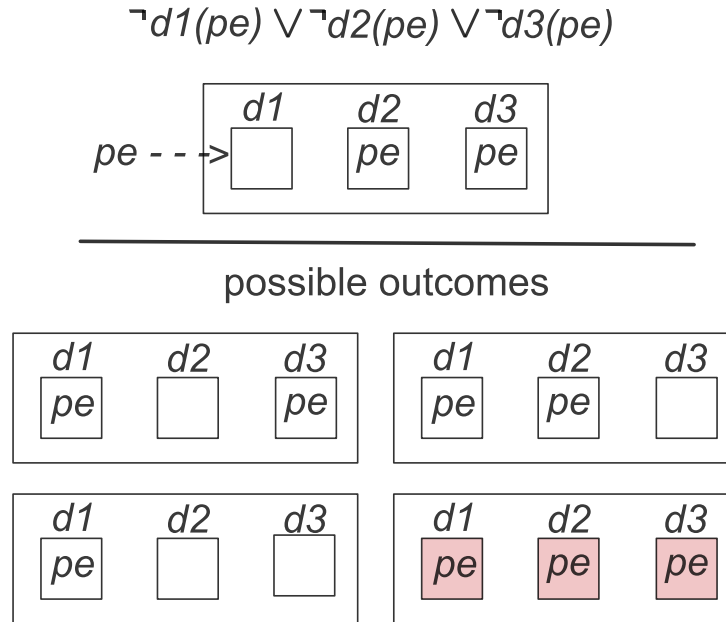


Figure 3.3: A spreadsheet with three cells, $d1$, $d2$, and $d3$ and a constraint stating that they cannot all have the value pe . At first, $d2$ and $d3$ contain the value pe . The user then places the values pe into cell $d1$. Four possible outcomes are displayed.

3.2.4 Dealing with inconsistency in a static spreadsheet

Our decision to allow conflicts between values in our spreadsheet has an unfortunate consequence: we cannot simply define the computed values as the logical consequences of the base value assignments and the constraints. To see why, recall that a set of sentences Π logically entails a sentence ψ if and only if every model of Π is a model of ψ . However, if the base value assignments are inconsistent with the constraints, then this set of sentences has no models, and therefore every model of these sentences is a model of any sentence. Therefore, when the base value assignments are inconsistent with the constraints, *every value is logically entailed in every cell*, thus rendering the spreadsheet useless.

We must therefore turn to a paraconsistent notion of entailment. The notion of entailment we use is as follows.

Definition 3.2. *Let Λ and Ω be sets of sentences. We say that a sentence ϕ is*

existentially Ω -entailed by Λ iff there is some subset $\lambda \subseteq \Lambda$ such that $\lambda \cup \Omega$ is consistent and $\lambda \cup \Omega$ logically entails ϕ . We will use the notation $\Lambda \vDash_{\Omega} \phi$ to indicate that Λ existentially Ω -entails ϕ .

The word “existential” in existential Ω -entailment refers to the fact that there *exists* some Ω -consistent subset λ such that $\lambda \cup \Omega$ logically entails ϕ .

Note that existential Ω -entailment behaves the same as logical entailment as long as the base value assignments are consistent with the constraints. However, when the base values are inconsistent with the constraints, the number of entailed computed values is kept in check.

For example, consider that in addition to cells $d1$, $d2$ and $d3$, we have a fourth cell m , which is meant to contain the student’s major. If cells $d1$, $d2$, and $d3$ all contain value pe , then $\{d1(pe), d2(pe), d3(pe)\} \cup \{\neg d1(pe) \vee \neg d2(pe) \vee \neg d3(pe)\}$ is inconsistent and logically entails $m(pe)$, $m(cs)$, $m(art)$, and so on. However, note that any subset of $\{d1(pe), d2(pe), d3(pe)\}$ taken together with the constraint $\neg d1(pe) \vee \neg d2(pe) \vee \neg d3(pe)$ does not entail any value for m , and therefore no value of m is existentially Ω -entailed by this spreadsheet.

We discuss Existential Ω -entailment in detail and give an efficient method for computing it in Chapter 4.

3.2.5 Dealing with single values

When the data is consistent with the constraints, then the data is consistent with the single-value constraints, and each cell has cardinality at most 1. However, when the data is inconsistent with the constraints, more than one value can be computable in a cell. For example, consider a spreadsheet with five cells, p_1 , q_1 , p_2 , q_2 and r , and two constraints, $p_1(X) \wedge q_1(X) \Rightarrow r(X)$ and $p_2(X) \wedge q_2(X) \Rightarrow r(X)$. Say that the spreadsheet now has base data $\{p_1(a), q_1(a), p_2(b)\}$, and therefore has computed data $\{r(a)\}$ (Figure 3.4). If the user places value b into cell q_2 , then the base data will be $\{p_1(a), q_1(a), p_2(b), q_2(b)\}$, and there will be two existentially Ω -entailed data for r , namely $r(a)$ and $r(b)$.

We choose a *fair* semantics in which neither value is preferred and therefore our

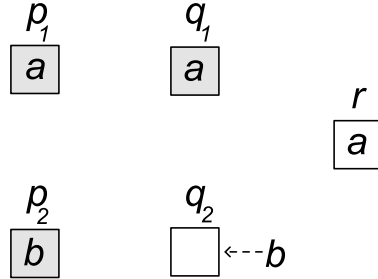


Figure 3.4: A spreadsheet with two constraints: $p_1(X) \wedge q_1(X) \Rightarrow r(X)$ and $p_2(X) \wedge q_2(X) \Rightarrow r(X)$. The value $q_2(b)$ is about to be inserted. After $q_2(b)$ is inserted, r will have two computed values - $r(a)$ and $r(b)$.

computed assignments Γ will have no data for r . In other words, we will define the computed data Γ as follows:

Definition 3.3 (Computed Data). *Given a spreadsheet with base data Λ and constraints Ω , we define the computed data Γ as follows:*

$$\Gamma =_{def} \{\rho(\tau) \mid \Lambda \approx_{\Omega} \rho(\tau) \wedge ((\Lambda \approx_{\Omega} \rho(\sigma)) \Rightarrow \tau = \sigma)\} \quad \square$$

Definition 3.3 says that a value τ is computed in a cell ρ iff τ is the only value computable for ρ .

3.2.6 Deletion

Deletion is as simple as can be: simply remove the facts from the base data.

Definition 3.4 (Spreadsheet Deletion). *Given base data Λ and data λ , define the deletion of λ from Λ as:*

$$\Lambda \ominus \lambda =_{def} \Lambda - \lambda \quad \square$$

This definition can be extended to deal with the bulk deletion of multiple facts, in which case it coincides with set deletion.

3.3 Summary

In this chapter we have given a domain-independent update semantics for logical spreadsheets. We gave a definition for insertion that removes directly conflicting values but retains other values even if they result in a conflict. Deletion was simply defined as set deletion. Multiple entailed values for a cell were treated by keeping the cell empty.

In the next chapter, we define a method for computing the consequences of spreadsheet data that is inconsistent with the constraints.

Chapter 4

Existential Ω -Entailment

In this chapter, we discuss existential Ω -entailment in more detail. In section 4.1 we give an illustrative example of existential Ω -entailment. In section 4.2 we examine its semantic properties. In section 4.3 we compare it to resolution. In section 4.4 we discuss how to compute the existentially Ω -entailed data from a set of constraints and a set of data. In section 4.5 we discuss related work.

4.1 Introduction

Existential Ω -entailment was defined in Definition 3.3. For convenience, we reproduce the definition here.

Definition 3.3 *Let Λ and Ω be sets of sentences. We say that a sentence ϕ is existentially Ω -entailed by Λ iff there is some subset $\lambda \subseteq \Lambda$ such that $\lambda \cup \Omega$ is consistent and $\lambda \cup \Omega$ logically entails ϕ . We will use the notation $\Lambda \approx_{\Omega} \phi$ to indicate that Λ existentially Ω -entails ϕ .*

Existential Ω -entailment is a generalization of existential entailment [40, 5, 19] to allow for a set of nondefeasible constraints Ω . A set of formulae Λ *existentially entails* a sentence ϕ iff there is a consistent subset $\lambda \subseteq \Lambda$ such that λ logically entails ϕ . Note that existential entailment corresponds to existential Ω -entailment where $\Omega = \emptyset$.

Proposition 4.1 (Maximality). *Say that $\Lambda \approx_{\Omega}^M \phi$ iff there is a maximal Ω -consistent subset $\lambda \subseteq \Lambda$ such that $\lambda \cup \Omega$ logically entails ϕ . Then $\Lambda \approx_{\Omega} \phi$ iff $\Lambda \approx_{\Omega}^M \phi$.*

Proof If $\Lambda \approx_{\Omega}^M \phi$, then there is an Ω -consistent subset λ of Λ such that $\lambda \cup \Omega \models \phi$, and thus $\Lambda \approx_{\Omega} \phi$. If $\Lambda \approx_{\Omega} \phi$, then there is an Ω -consistent subset λ of Λ such that $\lambda \cup \Omega \models \phi$. Let λ' be a maximal Ω -consistent subset of Λ that contains λ . By the monotonicity of logical entailment, since $\lambda \cup \Omega \models \phi$ and $\lambda \subseteq \lambda'$ then $\lambda' \cup \Omega \models \phi$. Since λ' is maximal, $\Lambda \approx_{\Omega}^M \phi$. \square

We will be focusing on the application of existential Ω -entailment to sets of formulae Λ where Λ consists solely of data. We will refer to the consequence relation *existential Ω -entailment restricted to data* defined by restricting the domain of existential Ω -entailment to data, and restricting the range facts (ground literals which may be positive or negative).

Existential Ω -entailment is a paraconsistent entailment relation, meaning that it does not entail all conclusions when an inconsistency is present. Intuitively, existential Ω -entailment only entails conclusions that are justified by an Ω -consistent subset of Λ . We illustrate this concept with an example.

Example 4.1. *Consider the following constraints Ω :*

$$\begin{aligned} \neg r(X) \vee \neg d(X) & \quad (\text{One cannot be both Republican and Democrat.}) \\ r(X) \Rightarrow g(X) & \quad (\text{Republicans like George.}) \\ d(X) \Rightarrow b(X) & \quad (\text{Democrats like Bill.}) \\ b(X) \wedge g(X) \Rightarrow u(X) & \quad (\text{People who like Bill and George are undecided.}) \end{aligned}$$

And the following data Λ :

$$\begin{aligned} r(\text{Robert}) & \quad (\text{Robert is a Republican}) \\ d(\text{David}) & \quad (\text{David is a Democrat}) \\ r(\text{Ed}) & \quad (\text{Ed is a Republican}) \\ d(\text{Ed}) & \quad (\text{Ed is a Democrat}) \end{aligned}$$

Then the maximal Ω -consistent sets of Λ are:

$$s_1: \{r(\text{Robert}), d(\text{David}), r(\text{Ed})\}$$

$$s_2: \{r(\text{Robert}), d(\text{David}), d(\text{Ed})\}$$

The data entailed by $s_1 \cup \Omega$ is

$$c_1 = \{r(\text{Robert}), d(\text{David}), r(\text{Ed}), g(\text{Robert}), b(\text{David}), g(\text{Ed})\}$$

The data entailed by $s_2 \cup \Omega$ is

$$c_2 = \{r(\text{Robert}), d(\text{David}), r(\text{Ed}), g(\text{Robert}), b(\text{David}), b(\text{Ed})\}$$

And so the existentially Ω -entailed facts are

$$c_1 \cup c_2 = \{r(\text{Robert}), d(\text{David}), r(\text{Ed}), g(\text{Robert}), b(\text{David}), g(\text{Ed}), d(\text{Ed})\}$$

Note that, although both $g(\text{Ed})$ and $d(\text{Ed})$ are existentially Ω -entailed, $u(\text{Ed})$ is not, due to the constraint $\neg r(X) \vee \neg d(X)$.

4.2 Properties

To help shed light on the behavior of Existential Ω -entailment, in this section we investigate its formal properties. We first note that existential Ω -entailment depends on the syntax of the theory it is applied to, not just the semantics. To make this notion formal, we first need the following definitions.

Definition 4.1 (Logical Equivalence). *Two theories Φ and Ψ are logically equivalent iff for all $\phi \in \Phi$, $\Psi \models \phi$ and conversely for all $\psi \in \Psi$, $\Phi \models \psi$.*

Definition 4.2 (Syntax Dependence). *A consequence relation \approx is syntax independent if, for any logically equivalent theories Φ and Ψ and any sentence γ , $\Phi \approx \gamma$ iff $\Psi \approx \gamma$. Otherwise it is syntax-dependent.*

Now we are ready to prove our assertion.

Proposition 4.2 (Syntax Dependence). *Existential Ω -entailment is syntax dependent.*

Proof Let $\Phi = \{p(a) \wedge q(a)\}$, let $\Psi = \{p(a), q(a)\}$ and let $\Omega = \{\neg q(a)\}$. Then Φ and Ψ are logically equivalent, but $\Psi \models_{\Omega} p(a)$ and $\Phi \not\models_{\Omega} p(a)$. \square

This is discouraging. However, in practice, we will only be using Existential Ω -entailment applied to sets of data.

Proposition 4.3 (Syntax Independence). *Existential Ω -entailment restricted to sets of data is syntax independent.*

Proof This trivially arises from the fact that two sets of data are logically equivalent iff they have the same data. \square

Also importantly, the logical form of the constraints does not matter.

Proposition 4.4 (Equivalence). *Let Ω and Θ be two logically equivalent theories. Then for any theory Λ and sentence ϕ , $\Lambda \models_{\Omega} \phi$ iff $\Lambda \models_{\Theta} \phi$.*

Proof Say that $\Lambda \models_{\Omega} \phi$. Then there is an Ω -consistent subset λ of Λ such that $\lambda \cup \Omega \models \phi$. Since Ω and Θ are logically equivalent, $\lambda \cup \Theta \models \phi$, and thus $\Lambda \models_{\Theta} \phi$. The other case is symmetric. \square

We now turn towards examining properties of existential Ω -entailment. First we define some notation to denote a generic consequence relation.

Definition 4.3 (Consequences). *Let Λ be a theory and let \approx be a consequence relation.*

$$C_{\approx}(\Lambda) =_{def} \{\phi \mid \Lambda \approx \phi\}$$

Table 4.1 lists properties (and non-properties) of existential Ω -entailment. The list of properties studied is adapted from [19] and [36]. Appendix 2 gives proofs of the various properties listed in Table 4.1.

Table 4.1 shows that existential Ω -entailment behaves quite differently than logical entailment, for which all of the properties hold.¹ For the properties that do not hold,

¹ Ω -Subclassicality and Ω -Consistent Classicality do not hold for logical entailment, though Subclassicality and Consistent Classicality do. Subclassicality, which says that \approx is no stronger than \models , is defined as $\Lambda \models \phi$ if $\Lambda \approx \phi$. Consistent Classicality is defined as if $\Lambda \not\models \perp$, $\Lambda \approx \phi$ iff $\Lambda \models \phi$, where \approx is a consequence relation.

Property	Name	Holds?
$\{\psi, \neg\psi\} \vDash_{\Omega} \phi$	(Explosiveness)	No
$\Lambda \cup \Omega \models \phi$ if $\Lambda \vDash_{\Omega} \phi$	(Ω -Subclassicality)	Yes
if $\Lambda \cup \Omega \not\models \perp$, $\Lambda \vDash_{\Omega} \phi$ iff $\Lambda \cup \Omega \models \phi$	(Ω -Consistent Classicality)	Yes
$\Lambda \cup \{\phi\} \vDash_{\Omega} \psi$ if $\Lambda \vDash_{\Omega} \psi$	(Monotonicity)	Yes
$\Lambda \vDash_{\Omega} \phi$ if $\phi \in \Lambda$	(Reflexivity)	No
$\Lambda \cup \{\phi\} \vDash_{\Omega} \gamma$ if $\Lambda \cup \{\psi\} \vDash_{\Omega} \gamma$ and $\models \phi \Leftrightarrow \psi$	(Left logical equivalence)	Yes
$\Lambda \vDash_{\Omega} \phi$ if $\Lambda \vDash_{\Omega} \psi$ and $\vDash_{\Omega} \psi \Rightarrow \phi$	(Right Weakening)	Yes
$\Lambda \cup \{\neg\psi\} \vDash_{\Omega} \neg\phi$ if $\Lambda \cup \{\phi\} \vDash_{\Omega} \psi$	(Contraposition)	No
$\Lambda \vDash_{\Omega} \phi \wedge \psi$ if $\Lambda \vDash_{\Omega} \phi$ and $\Lambda \vDash_{\Omega} \psi$	(And)	No
$\Lambda \vDash_{\Omega} \psi$ if $\Lambda \vDash_{\Omega} \phi$ and $\Lambda \cup \{\phi\} \vDash_{\Omega} \psi$	(Cut)	No
$\Lambda \vDash_{\Omega} \phi \Rightarrow \psi$ if $\Lambda \cup \{\phi\} \vDash_{\Omega} \psi$	(Conditionalization)	Yes
$\Lambda \cup \{\phi\} \vDash_{\Omega} \psi$ if $\Lambda \vDash_{\Omega} \phi \Rightarrow \psi$	(Deduction)	No
$C_{\vDash_{\Omega}}(\Lambda) = C_{\vDash_{\Omega}}(C_{\vDash_{\Omega}}(\Lambda))$	(Idempotence)	No

Table 4.1: Properties of Existential Ω -entailment

they do not hold even when existential Ω -entailment is restricted to data.

We briefly comment on the of the properties of Table 4.1 that are most relevant to logical spreadsheets. As desired, existential Ω -entailment is not explosive; or in other words, it is *paraconsistent*. It generates a subset of the conclusions of logical entailment (Ω -Subclassicality). It coincides with logical entailment on consistent theories (Ω -Consistent Classicality). It is monotonic, so adding more facts will never generate fewer conclusions. It is irreflexive, so not all of its premises are also conclusions. This occurs only when a premise directly conflicts with the constraints Ω . Repeated application of existential Ω -entailment can generate more conclusions (non-Idempotence), so distinguishing between “base facts” and “computed facts” is important.

4.3 Comparison to Resolution

The need for a paraconsistent entailment relation to show the consequences of a spreadsheet arises due to the explosive nature of classical logic (and, consequently, Herbrand Logic). However, it is worth noting that resolution will also not derive all sentences given an inconsistent input. In this section, we see that resolution can be used to derive a paraconsistent entailment relation that derives more conclusions than existential Ω -entailment does.

Example 4.2 (Paraconsistency of Resolution). *Consider the following theory:*

$$\{p \vee q, r, \neg r\}$$

Although the theory is inconsistent due to the presence of r and $\neg r$, there is no resolution derivation of p from the theory.

Indeed, resolution can also be used to construct a paraconsistent entailment relation.

Definition 4.4 (Resolution entailment). *Let Ω be a set of clausal formulae, and let ϕ be a literal. We say that Ω -resolution entails ϕ (written $\Omega \vDash_R \phi$) iff ϕ is subsumed by a non-empty clause that has a resolution derivation from Ω .*

Resolution entailment corresponds to quasi-classical logic [6], restricted to clauses and literals.

The following proposition says that the conclusions of existential Ω -entailment are contained in the conclusions of resolution entailment.

Proposition 4.5. *Let Λ be a set of data, let Ω be a set of clauses, and let ϕ be a literal. Then $\Lambda \cup \Omega \vDash_R \phi$ if $\Lambda \vDash_\Omega \phi$.*

Before proving this proposition, we first note the following lemma:

Lemma 4.6 (The Subsumption Theorem [31]). *Let Ω be a set of clauses, and let ϕ be a clause that is not a tautology. Then $\Omega \vDash \phi$ iff there is a clause δ that is resolution derivable from Ω such that δ subsumes ϕ .*

Proof of Proposition 4.5. Since $\Lambda \approx_{\Omega} \phi$, there is an Ω -consistent subset λ of Λ such that $\lambda \cup \Omega \models \phi$. By the Subsumption Theorem, there is a clause δ that is resolution derivable from Ω such that δ subsumes ϕ . Since $\lambda \cup \Omega \not\models \perp$, we see that δ cannot be the empty clause. Thus $\lambda \cup \Omega \approx_R \phi$, and by monotonicity, $\Lambda \cup \Omega \approx_R \phi$. \square

However, the opposite is not the case, i.e. the conclusions of resolution entailment are not contained in the conclusions of existential Ω -entailment.

Proposition 4.7. *There exists a set of data Λ , a set of clauses Ω , and a literal ϕ such that $\Lambda \cup \Omega \approx_R \phi$ but $\Lambda \not\approx_{\Omega} \phi$.*

Proof Let $\Omega = \{\neg p \vee \neg q, \neg p \vee \neg q \vee r\}$, let $\Lambda = \{p, q\}$, and let $\phi = r$. \square

4.4 Computing Existential Ω -Entailment

We now turn to the question of how to efficiently compute whether a fact is existentially Ω -entailed from a set of data.

Going straight from the definition, we can reduce the problem to resolution as follows. First, we define a procedure for determining if $\Lambda \cup \Omega$ is consistent with a literal ϕ . While this procedure is undecidable in general, we note that it is decidable for the special cases of propositional logic [25], monadic logic [8] and Finite Herbrand Logic [29]:²

Algorithm 4.1 IS-CONSISTENT[Ω]

```

1: if  $\Omega$  resolves to the empty clause then
2:   return false
3: else
4:   return true
5: end if

```

Now, we are able to obtain an algorithm for computing whether a fact is existentially Ω -entailed by a set of data Λ and a set of constraints Ω .

²Broader classes of logic can be decided by using methods other than resolution. For example, to determine if a set of universally quantified sentences is consistent, we can the method given in [47].

Algorithm 4.2 NAÏVE-ALGORITHM $[\Lambda, \Omega, \phi]$

```

1: for all subsets  $\lambda$  of  $\Lambda$  do
2:   if IS-CONSISTENT( $\lambda \cup \Omega$ ) then
3:     if  $\neg$ IS-CONSISTENT( $\lambda \cup \Omega \cup \{\neg\phi\}$ ) then
4:       return true
5:     end if
6:   end if
7: end for
8: return false

```

It is easy to see that the algorithm is correct, as it translates quite directly from the definition of existential Ω -entailment. Unfortunately, even in the best case, this algorithm requires iterating through all subsets of Λ , the number of which is exponential in the size of Λ . In the case of a spreadsheet, if there are k cells with values, this means we need to query all 2^k subsets of the values.

We can do better in general by realizing that, to determine whether a fact that is entailed by *some* subset of data, there is no need to do brute-force search through *every* subset of data. Intuitively, we should intelligently choose what subsets to look through, ignoring other “irrelevant” subsets altogether.

Example 4.3. Consider $\Omega = \{\neg p(X) \vee \neg q(X) \vee r(X), \neg p(a) \vee \neg q(a)\}$ and $\Lambda = \{p(a), p(b), q(a), q(b), s(b)\}$. Intuitively, when querying for $r(b)$, $s(b)$ is irrelevant, in the sense that there is no need to consider subsets of Λ that contain $s(b)$. Only subsets that contain $p(b)$, $q(b)$, or $r(b)$ are relevant.

How can we make this intuitive notion of relevance a precise concept? A proof-theoretic view is of use here. In particular, note that one can prove fact ϕ from data Λ and constraints Ω iff there is a resolution refutation of $\neg\phi$ from the clausal form of $\Lambda \cup \Omega$. Thus, we only need to consider subsets of Λ that are used in some minimal-length resolution refutation of $\neg\phi$. Thus, instead of iterating blindly through all consistent subsets and testing whether they entail ϕ , we can build up a resolution refutation of $\neg\phi$, ensuring that all the premises of the refutation are Ω -consistent.

A *clause with lineage* is a 2-tuple $\langle \phi, \Delta \rangle$, where ϕ is a clause and Δ is a set of clauses. We will use clauses with lineage to represent the fact that ϕ was derived

using facts in Δ .

Using clauses with lineage, we can modify the traditional definition of resolution to derive only Ω -consistent conclusions. Suppose that $\langle \Phi, \Lambda \rangle$ and $\langle \Psi, \Delta \rangle$ are two clauses with lineage. If (1) there is a literal ϕ in some factor Φ' of Φ and a literal $\neg\psi$ in some factor Ψ' of Ψ such that ϕ and ψ have a most general unifier γ and (2) $\Lambda \cup \Delta \cup \Omega$ is consistent, then we say that the two clauses Φ and Ψ Ω -resolve and that the new clause with lineage $\langle ((\Phi' - \{\phi\}) \cup (\Psi' - \{\neg\psi\}))\gamma, \Lambda \cup \Delta \rangle$ is a Ω -resolvent of the two clauses.

A Ω -resolution derivation of a clause ϕ from a set of clauses with lineage Δ is a sequence of clauses with lineage terminating in $\langle \phi, \delta \rangle$ for some δ in which each item is either (1) a member of Δ (a *premise*) or (2) the result of applying Ω -resolution to earlier items in the sequence.

A sentence ϕ is *provable* from a set of sentences Δ by Ω -resolution if and only if there is an Ω -resolution derivation of the empty clause from the set $lineage(\Delta, \Omega, \phi)$, where $lineage(\Delta, \Omega, \phi)$ is defined as consisting of the following clauses with lineage:

- (1) $\langle \psi, \{\psi\} \rangle$, for each ψ in the clausal form of Δ such that ψ is consistent with Ω ;
- (2) $\langle \psi, \{\} \rangle$, for each ψ in the clausal form of Ω ;
- (3) $\langle \psi, \{\} \rangle$, for each ψ in the clausal form of $\neg\phi$.

Ω -resolution does what we hope to accomplish - generate all resolution derivations except those whose premises are not Ω -consistent.

Theorem 4.8 (Soundness and Completeness). $\Lambda \models_{\Omega} \phi$ iff ϕ is provable from Λ by Ω -resolution.

Proof of Theorem 4.8. Let Λ and Ω be sets of sentences, and let ϕ be a sentence. We have:

$$\Lambda \models_{\Omega} \phi$$

iff there is an Ω -consistent subset λ of Λ such that $\lambda \cup \Omega \models \phi$

iff there is a resolution derivation $D = \langle \psi_1, \dots, \psi_k \rangle$ from the clausal form of $\lambda \cup \Omega \cup \{\neg\phi\}$ such that $\psi_k = \emptyset$

iff there is an Ω -resolution derivation D' of \emptyset from $\text{lineage}(\lambda, \Omega, \phi)$, where D' is exactly D except each resolvent ψ_i , $1 \leq i \leq k$ has been replaced by a corresponding clause with lineage $\Psi_i = \langle \psi_i, \delta \rangle$ such that the lineage δ is (1) \emptyset , if the ψ_i is in the clausal form of Ω or $\neg\phi$, (2) ψ_i , if ψ_i is in the clausal form of λ , or (3) the union of the lineages of the clauses that resolve to ψ_i , otherwise. (Note that no union of lineages of clauses can be inconsistent with Ω since if a union of lineages of clauses L were inconsistent with Ω , then since L is a subset of the clausal form of λ , then λ must also be inconsistent with Ω , which it is not.)

iff ϕ is Ω -derivable from $\text{lineage}(\Lambda, \Omega, \phi)$

iff ϕ is provable from Λ by Ω -resolution. \square

We illustrate Ω -resolution with a few examples. We begin by revisiting Example 4.3.

Example 4.4. Let $\Omega = \{\neg p(X) \vee \neg q(X) \vee r(X), \neg p(a) \vee \neg q(a)\}$ and $\Lambda = \{p(a), p(b), q(a), q(b), s(b)\}$. Then we can prove $r(b)$ from Ω and Λ via Ω -resolution:

- | | |
|--|-----------|
| (1) $\langle \neg r(b), \{\} \rangle$ | (goal) |
| (2) $\langle p(b), \{p(b)\} \rangle$ | (premise) |
| (3) $\langle q(b), \{q(b)\} \rangle$ | (premise) |
| (4) $\langle \neg p(X) \vee \neg q(X) \vee r(X), \{\} \rangle$ | (premise) |
| (5) $\langle \neg q(b) \vee r(b), \{p(b)\} \rangle$ | [2,4] |
| (6) $\langle r(b), \{p(b), q(b)\} \rangle$ | [3,5] |
| (7) $\langle \{\}, \{p(b), q(b)\} \rangle$ | [1,6] |

Note that the lineage of the empty clause of the proof is $\{p(b), q(b)\}$. In particular, note that $s(b)$ is not included. This illustrates how Ω -resolution avoids considering irrelevant subsets of Λ , as discussed above.

Also, as desired, we cannot prove $r(a)$ from Ω and Λ via Ω -resolution:

- | | |
|--|-----------|
| (1) $\langle \neg r(a), \{\} \rangle$ | (goal) |
| (2) $\langle p(a), \{p(a)\} \rangle$ | (premise) |
| (3) $\langle q(a), \{q(a)\} \rangle$ | (premise) |
| (4) $\langle \neg p(X) \vee \neg q(X) \vee r(X), \{\} \rangle$ | (premise) |
| (5) $\langle \neg q(a) \vee r(a), \{p(a)\} \rangle$ | [2,4] |
| (6) $\langle \neg q(a), \{p(a)\} \rangle$ | [1,5] |
| (7) $\langle \{\}, \{p(a), q(a)\} \rangle$ | [1,6] |

Step (7) is not allowed since $\{p(a), q(a)\}$ is inconsistent with Ω ; in particular it is inconsistent with $\neg p(a) \vee \neg q(a)$.

4.5 Related Work

4.5.1 Existential- Ω Entailment

Existential Ω -entailment was first introduced in [35]. It was later studied in [30], which gives a method for transforming a set of clausal constraints Ω into an open Datalog program that, together with a set of data, entails the existentially Ω -entailed facts for the data. At a high level, the method consists of (1) computing the resolution closure Ω^* , of Ω , (2) for each contrapositive of each clauses in Ω^* , create a corresponding Datalog rule, and (3) for each of these Datalog rules, append an Ω -consistency check.

One purpose of the algorithm is syntactic - it transforms the constraints into Datalog. Datalog is well understood and has many known query optimization and view maintenance techniques [32, 42]. Datalog is also easily translatable into Relational Calculus, upon which many of today's broadly used database management systems are built [1].

A second desirable feature of the algorithm is theoretical. The algorithm reformulates a problem with non-polynomial data complexity (resolution refutation) into a problem with polynomial data complexity (Datalog evaluation). Although the algorithm may take non-polynomial time in the size of the constraints to produce the compiled Datalog program, this upfront cost is easily amortized away when querying large databases.

The amortized savings comes from (1) precomputing the resolution closure of the constraints, (2) precomputing the consistency checks, and (3) being able to optimize queries based upon the precomputed consistency checks. However, these savings are often not as dramatic in the case of logical spreadsheets, where the size of the constraints is often proportional to the size of the data, the tables are single-valued, and the constraints may change more often than in the case of a large database.

4.5.2 Related Consequence Relations

There are a rich variety of paraconsistent logics. The interested reader is referred to [44] for a recent survey. These logics are generally concerned with modifying the inference rules of traditional logic so as to avoid concluding nonsense in the face of inconsistency. There are a few exceptions. In particular, existential entailment, first defined by [40], is defined as the consequences that are logically entailed by some consistent subset of a theory. Existential Ω -entailment can be thought of as existential entailment with a nondefeasible set of constraints Ω .

Existential entailment was revisited in [5] and [19] which introduced a family of argumentative logics, which are subset based. In addition to existential entailment, argumentative logics include the dual notion of universal entailment, which is defined to be the consequences that are logically entailed by every maximally consistent subset of a theory. Other argumentative logics include **AF**, which is defined as the logical consequences of the subset of the theory that is free of inconsistencies, and the logic **AR** which allows for all logical consequences ϕ of a maximal subset such that $\neg\phi$ is not entailed by another maximal subset. [18] gives an algorithm for constructing arguments in propositional logic using connection graphs.

Defeasible Logic Programming [22] is a logic programming language which partitions the logic program into a set of feasible rules and data and a set of nondefeasible rules and data. Conclusions for which the negation is not also a conclusion are considered to be answers.

Consistent query answering is introduced in [3]. Unlike the previously mentioned

consequence relations, consistent query answering distinguishes between the constraints and the data. A consistent query answer is an answer that is entailed in all minimal repairs of the data.

To understand how the aforementioned consequence relations relate to one another, we can classify them according to following questions:

- Is the consequence relation based on “possible worlds”? In other words, does the consequence relation consider each consistent subset of the theory or the data to produce some results and then combine the results in some way? If so:
 - Does the consequence relation require that a conclusion is true in *every* possible world (it is *universal*) or just *some* possible world (it is *existential*)?
 - Are the possible worlds defined to be *subsets* of the original data, or are they defined to be *minimal repairs* of the original data?
- Does the consequence relation distinguish between constraints and data (in particular, is it parameterized by constraints and operate on data)? Or does it treat all sentences equally?

The answers to these questions are summed up in the following table. The table lists the consequence relations most closely related to existential Ω -entailment and describes whether they are universal (\forall) or existential (\exists), whether they are subset or repair-based, and whether the relations consider the constraints separately from the data, or whether they are treated together in a uniform fashion.

Consequence relation	\forall/\exists	Subsets/Repairs	Separate/Together
Existential Ω -entailment	\exists	Subsets	Separate
Existential entailment	\exists	Subsets	Together
Universal entailment	\forall	Subsets	Together
Consistent query answering	\forall	Repairs	Separate

Chapter 5

Dynamics

In Chapter 3, we discussed how it can be ambiguous how to maintain consistency with the spreadsheet constraints while obeying an update. This led us to allowing for updates that result in states that are inconsistent with the spreadsheet constraints, thus necessitating a paraconsistent entailment capability.

In this chapter, we consider a second possibility: the spreadsheet creator can provide at design time knowledge about how the spreadsheet should react to updates so that consistency can be maintained. We design a new logic, called Markov Change Logic, that allows the spreadsheet creator to explicitly control the dynamic behavior of the spreadsheet. Markov Change Logic allows the spreadsheet creator to specify how to avoid inconsistencies, as well as to specify arbitrary dynamic behaviors. Markov Change Logic is applicable not just to spreadsheets, but to databases as well.

Section 5.1 contains some motivating examples. In Section 5.2 we define Markov Change Logic. Markov Change Logic specifies database dynamics in terms of adjacent pairs of database states. In Section 5.3 we give examples of applications of MCL. In Section 5.4 we show how to convert a set of constraints to Markov Change Logic and compare Markov Change Logic to other formalisms. In Section 5.5 we explore the question of whether a Markov language is sufficient for specifying multistep dynamic constraints. We give a theoretical analysis that shows that the answer is yes, as long as the database schema may be extended with additional relations. Section 5.6 concludes by discussing related work.

5.1 Motivation

Constraints across three or more spreadsheet cells can lead to several possible ways to maintain consistency. Consider the following example.

Example 5.1. *Consider a spreadsheet with constraints $\Omega = \{\neg p(X) \vee \neg q(X) \vee \neg r(X)\}$ and data $\Lambda = \{q(a), r(a)\}$, and an update of $p(a)$. To maintain consistency while accepting the update, one may (1) remove $q(a)$, (2) remove $r(a)$ or (3) remove both $q(a)$ and $r(a)$.*

Without any further semantic knowledge about p , q , and r , there is no way to tell which of these options is “correct.” In the interest of not throwing away important information by making an arbitrary choice, we allow instead for the spreadsheet to be inconsistent with the constraints. We defer all decisions to the spreadsheet user. On the other hand, if the spreadsheet creator were to specify that whenever $p(a)$ is inserted and $q(a)$ and $r(a)$ are true, then $q(a)$ should always be deleted and $r(a)$ retained, then there is no longer any ambiguity, and the spreadsheet can automatically maintain consistency.

As a less abstract example, consider again the event scheduling spreadsheet from Section 1.1 that contains values for the start time, end time, and duration of the event. There are four cells, *title*, *start*, *end* and *duration*, and two constraints, which specify that the start time is before the end time and that the start time plus the duration equals the end time.

As depicted in Figure 5.1(1), the spreadsheet begins with data $\{title("LogicGroupMeeting"), start(1pm), end(3pm)\}$, from which $duration(2hours)$ is entailed. If $duration(3hours)$ is then inserted, the spreadsheet becomes inconsistent with the second constraint (Figure 5.1(2a)). However, it would be desirable in this case to have the spreadsheet instead remove the user’s assertion of the end time as $3pm$ and then compute the end time as $4pm$, thus avoiding the inconsistency (Figure 5.1(2b)).

Similarly, when the start time and duration are specified and the end time is changed, we would like to leave the start time alone and change the duration to accommodate the new end time. Finally, when the duration and end time are specified

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	3:00 pm
Duration	2 hours

(1)

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	3:00 pm
Duration	3 hours

(2a)

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	4:00 pm
Duration	3 hours

(2b)

Figure 5.1: Creating an event. (1) Currently the start time and end time are set by the user, and the duration is computed. (2a) Entering in “3 hours” for the event duration results in an inconsistency because the system is unable to determine whether to change the start time, the end time, or both. (2b) The state that we would like the spreadsheet to result in, in which the end time becomes computed rather than user-specified.

and the start time is changed, we would like to leave the end time alone and change the duration appropriately. With these three preferences, we can avoid inconsistency altogether in the spreadsheet.

In the next section, we define a logic that allows us to express these types of preferences.

5.2 Markov Change Logic

In this section we define a simple logic that allows one to specify the update behaviors of databases and spreadsheets. We call our logic *Markov Change Logic*, or MCL for short.

Markov Change Logic is designed to be used as part of an update engine in a database or logical spreadsheet. The intended use is as follows. An update request (a set of requested insertions and deletions) is generated by an agent. The update request, the current database state, and the MCL program are fed as inputs to a Datalog engine. The Datalog engine produces the output base of the MCL program, which is the output update. The output update is then applied to the database. This process is illustrated in Figure 5.2.

We now formally define Markov Change Logic. Let D be a database with schema S . Let S have relation constants R and domain U . Then the *Markov Change Logic* for S , written $MCL(S)$, is defined as follows.

We define:

$$R^+ =_{def} \{r^+(X_1, \dots, X_k) \mid r(X_1, \dots, X_k) \in R\}$$

$$R^- =_{def} \{r^-(X_1, \dots, X_k) \mid r(X_1, \dots, X_k) \in R\}$$

An extension of $R^+ \cup R^-$ is an *input update*. An extension of R^+ is a *positive input update*, and an extension of R^- is a *negative input update*.

Next, we define:

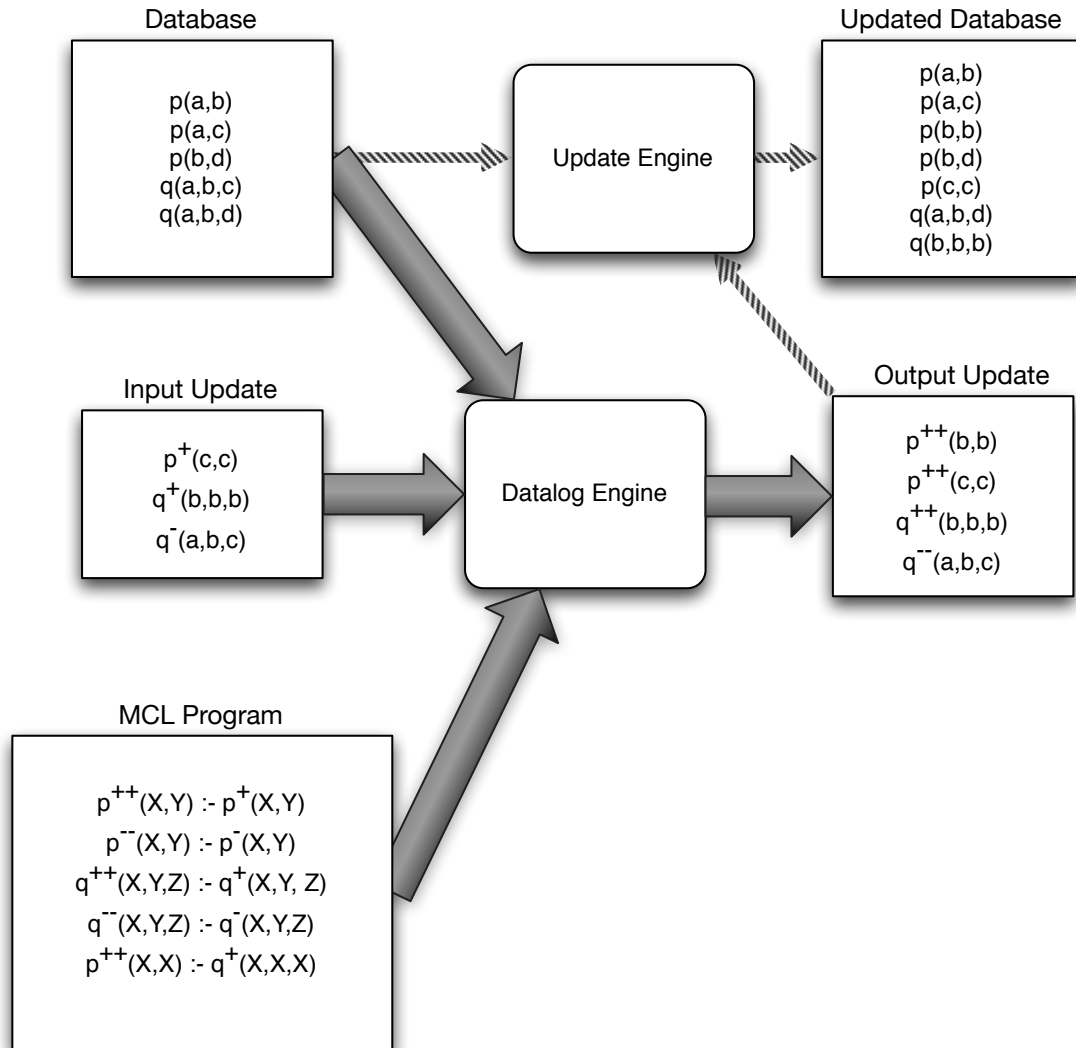


Figure 5.2: MCL being used by an update engine to update a database. An input update, the database, and an MCL program are fed into a Datalog engine. The output model of the MCL program is the output update. The output update is used by an update engine to update the database.

$$R^{++} =_{def} \{r^{++}(X_1, \dots, X_k) \mid r(X_1, \dots, X_k) \in R\}$$

$$R^{--} =_{def} \{r^{--}(X_1, \dots, X_k) \mid r(X_1, \dots, X_k) \in R\}$$

An extension of $R^{++} \cup R^{--}$ is an *output update*. An extension of R^{++} is a *positive output update*, and an extension of R^{--} is a *negative output update*. We will want to apply the output updates to the current state of the database to move to the next state of the database. To do this, we will need to strip off the $^{++}$ and $^{--}$ from the output updates. Let O be a set of data. Then we define:

$$\begin{aligned} strip^{++}(O) &=_{def} \{r(t_1, \dots, t_k) \mid r^{++}(t_1, \dots, t_k) \in O\} \\ strip^{--}(O) &=_{def} \{r(t_1, \dots, t_k) \mid r^{--}(t_1, \dots, t_k) \in O\} \end{aligned}$$

We define corresponding operators on a set of data λ :

$$\begin{aligned} \lambda^+ &=_{def} \{r^+(\bar{t}) \mid r(\bar{t}) \in \lambda\} \\ \lambda^- &=_{def} \{r^-(\bar{t}) \mid r(\bar{t}) \in \lambda\} \\ \lambda^{++} &=_{def} \{r^{++}(\bar{t}) \mid r(\bar{t}) \in \lambda\} \\ \lambda^{--} &=_{def} \{r^{--}(\bar{t}) \mid r(\bar{t}) \in \lambda\} \\ strip^{++}(\lambda) &=_{def} \{r(\bar{t}) \mid r^{++}(\bar{t}) \in O\} \\ strip^{--}(\lambda) &=_{def} \{r(\bar{t}) \mid r^{--}(\bar{t}) \in O\} \end{aligned}$$

We also define:

$$strip(\lambda) =_{def} \{r(\bar{t}) \mid r^{++}(\bar{t}) \in O\} \cup \{\neg r(\bar{t}) \mid r^{--}(\bar{t}) \in O\}$$

$MCL(S)$ has relation constants $R \cup R^+ \cup R^- \cup R^{++} \cup R^{--}$. The domain of $MCL(S)$ is U . An *MCL program* is an open Datalog program with input relations $R \cup R^+ \cup R^-$ and output relations $R^{++} \cup R^{--}$.

We now define the meaning of an MCL program P . If P has schema R , We define $next[P]$ be the function that maps a database with schema $R \cup R^{++} \cup R^{--}$ to a database with schema R , defined as:

$$next[P](\Lambda) =_{def} \Lambda - strip^{--}(\Lambda) \cup strip^{++}(\Lambda)$$

The meaning of an MCL program P is the function:

$$\text{applyMCL}[P] =_{\text{def}} \text{next}[P] \circ P \text{ (the function composition of } \text{next}[P] \text{ and } P)$$

P takes a database as input and computes output update together with the database. $\text{next}[P]$ then applies the output update to the database.

5.3 Applications

Markov Change Logic can be used to serve many functions in a database. In this section, we show that MCL can be used for consistency maintenance, view maintenance, updates through views, and general database dynamics.

5.3.1 Consistency Maintenance

Consider a spreadsheet with three cells, p , q , and r , and the constraint $\neg p(X) \vee \neg q(X) \vee \neg r(X)$. Say that the spreadsheet currently has data $\{q(a), r(a)\}$. An insertion of fact $p(a)$ would lead to the inconsistent state $\{p(a), q(a), r(a)\}$. However, consider the following MCL program:

- (1) $p^{++}(X) :- p^+(X)$
- (2) $q^{++}(X) :- q^+(X)$
- (3) $r^{++}(X) :- r^+(X)$
- (4) $p^{--}(X) :- p^-(X)$
- (5) $q^{--}(X) :- q^-(X)$
- (6) $r^{--}(X) :- r^-(X)$
- (7) $r^{--}(X) :- p^+(X) \ \& \ q(X) \ \& \ r(X)$
- (8) $r^{--}(X) :- p(X) \ \& \ q^+(X) \ \& \ r(X)$
- (9) $q^{--}(X) :- p(X) \ \& \ q(X) \ \& \ r^+(X)$

Rules (1)-(3) state that insertions are successful. Rules (4)-(6) state that deletions are successful. Rule (7) says that when a value is inserted into p and q and r currently

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	3:00 pm
Duration	2 hours

(1)

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	3:00 pm
Duration	3 hours

(2a)

Event Title	Logic Group Meeting
Start Time	1:00 pm
End Time	4:00 pm
Duration	3 hours

(2b)

Figure 5.3: Creating an event. (1) Currently the start time and end time are set by the user, and the duration is computed. (2a) By default, entering in “3 hours” for the event duration results is an inconsistency because the system is unable to determine whether to change the start time, the end time, or both. (2b) Using a MCL program, we can instruct the spreadsheet change the end time, resulting in the above state.

have that value, delete that value from r . Rule (8) says that when a value is inserted into q and p and r currently have that value, delete that value from r . Rule (9) says that when a value is inserted into r and p and q currently have that value, delete that value from r .

Applying the MCL program to the state $\{q(a), r(a)\}$ and the update $p^+(a)$ leads to state $\{p(a), q(a)\}$. In fact, assuming that only one cell is updated at a time, the above MCL program will always ensure the spreadsheet is in a consistent state.

For a more concrete example of consistency maintenance, consider again the event creation example of Section 5.1, which concerns a spreadsheet with the cells *title*, *start*, *end*, and *duration*. We begin with the state shown in Figure 5.3(1), which has data:

title(“Logic Group Meeting”)

start(1pm)
end(3pm)

And from which *duration(2hours)* is computed. Using the standard update semantics brings us to the inconsistent state shown in Figure 5.3(2a). We can revise this behavior by using the following MCL program:

- (1) $start^{++}(S) :- start^+(S)$
- (2) $end^{++}(E) :- end^+(E)$
- (3) $duration^{++}(D) :- duration^+(D)$
- (4) $start^{--}(S2) :- start^+(S) \ \& \ start(S2) \ \& \ S \neq S2$
- (5) $end^{--}(E2) :- end^+(E) \ \& \ end(E2) \ \& \ E \neq E2$
- (6) $duration^{--}(D2) :- duration^+(D) \ \& \ duration(D2) \ \& \ D \neq D2$
- (7) $end^{--}(E) :- duration^+(D2) \ \& \ duration(D) \ \& \ start(S) \ \& \ end(E)$
- (8) $duration^{--}(D) :- start^+(S2) \ \& \ start(S) \ \& \ end(E) \ \& \ duration(D)$
- (9) $duration^{--}(D) :- end^+(E2) \ \& \ end(E) \ \& \ start(S) \ \& \ duration(D)$

Rules (1)-(3) state that updates are successful, while rules (4)-(6) maintain the single-valued nature of the cells. Rule (7) states that when the duration is updated and the start time is already specified, the specified end time is to be erased. Rule (8) states that when the start time is updated and the end time is already specified, the specified duration time is to be erased. Rule (9) states that when the end time is updated and the start time is already specified, the specified duration is to be erased. Note that this MCL program does not correspond to any set of static constraints as it resolves the ambiguity of which of two values are to be erased when a third value is updated.

Now, given the following input update:

$duration^+(3hours)$

Then the output update is:

duration⁺⁺(3hours)
end⁻⁻(3pm)

The result of applying the MCL program to the input update and the database extension is:

title("Logic Group Lunch")
start(1pm)
duration(3hours)

From which, we can compute *end*(4pm). This state is shown in Figure 5.3(2b). Note that consistency has been maintained, and furthermore a particular way to maintain consistency has been specified, namely to delete the end time. For example, if we changed rule (7) to:

$$start^{--}(S) :- duration^+(D2) \& duration(D) \& start(S) \& end(E)$$

Then the start time would have been erased instead. Note that built into the MCL program is an assumption that at most one datum is changed at a time. If this is not the case, then we could for example change rule (7) to:

$$end^{--}(E) :- duration^+(D2) \& duration(D) \& start(S) \& \neg start^-(S) \& end(E)$$

Which would erase the end time unless the start time was being erased anyway.

5.3.2 View Maintenance

It is sometimes useful to *materialize*, or store, the contents of a database view to make querying that view more efficient. Once a view is materialized, it must be

maintained, or kept consistent with the base relations, as the base relations are updated. Active rules [10] have been used to change to the view relations as the base relations are changed. MCL programs are a natural declarative language that can be used to express these changes. By writing the differential formulas detailed in [42] as MCL programs, one can easily maintain any nonrecursively defined database view. For example, consider the following view definition:

$$p(x, y) :- q(x, y) \ \& \ \neg r(y)$$

Then the following MCL program maintains the view:

$$p^{++}(x, y) :- q^+(x, y) \ \& \ \neg r(y) \ \& \ \neg r^+(y)$$

$$p^{++}(x, y) :- q(x, y) \ \& \ \neg q^-(x, y) \ \& \ r^-(y)$$

$$p^{++}(x, y) :- q^+(x, y) \ \& \ r^-(y)$$

$$p^{--}(x, y) :- q^-(x, y) \ \& \ \neg r(y)$$

$$p^{--}(x, y) :- q(x, y) \ \& \ r^+(y)$$

The first three rules define when to insert tuples into p . For example, if $q(a, b)$ is inserted and $r(b)$ is deleted as part of the same update, the third rule will insert $p(a, b)$. The final two rules define when tuples are to be deleted from p .

5.3.3 Updates Through Views

It is sometimes desirable to make an update to a view directly. As views are defined by their base relations, this means that the base relations themselves must be changed. Similarly to the problem we discussed in Section 3.2.3, however, it is sometimes ambiguous how to translate an update to a view into an update on the base relations. For example, consider the following view, which projects out an employee's department:

$$manager(emp, mgr) :- department(emp, mgr, dept)$$

event	owner	projection	room	time
e1	amy	no		
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning			
afternoon			
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 5.4: An empty schedule

If we are to insert a tuple (*Eric, Manny*) into the *manager* view, what should we set the *dept* of Eric to be in the department table? MCL allows us to write the following rule, which sets the employee's department to whatever his manager's department is:

$$department^{++}(emp, mgr, dpt) :- manager^{+}(emp, mgr) \& department(mgr, m2, dpt)$$

5.3.4 Database Dynamics

We now turn to an example of when MCL might be used not to enforce consistency, but to enforce a particular behavior that is better for the end user of the spreadsheet.

Let us return to the event scheduling example of Section 1.1. Recall that the example involved an administrator whose job was to schedule events by assigning them a room and a time. The example contained several tables and constraints, but for our current purposes, we are only concerned with the relationship between the event table and the schedule table. In the event table, for each event \mathbf{E} , there is (among other things) a cell $event.room_{\mathbf{E}}$ and a cell $event.time_{\mathbf{E}}$ which denote the room and time for which the event is scheduled to take place. In the schedule table, for each room \mathbf{R} and each time \mathbf{T} , there is a cell $schedule_{\mathbf{T},\mathbf{R}}$ that denotes the event

event	owner	projection	room	time
e1	amy	no	g100	morning
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning	e1		
afternoon			
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 5.5: After scheduling e1

event	owner	projection	room	time
e1	amy	no		afternoon
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning			
afternoon			
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 5.6: After changing the time to the afternoon

scheduled at that time, in any. For each room \mathbf{R} , time \mathbf{T} , and event \mathbf{E} , there is a constraint $schedule_{\mathbf{T},\mathbf{R}}(\mathbf{E}) \Leftrightarrow event.time_{\mathbf{E}}(\mathbf{T}) \wedge event.room_{\mathbf{E}}(\mathbf{R})$ which relates the two tables in a 1-to-1 fashion.

Say that the administrator starts with a blank schedule table and an event table with the room and time cells also all blank (Figure 5.4). First the administrator sets the value of $schedule_{morning,g100}$ to $e1$, from which $event.room_{e1}(g100)$ and $event.time_{e1}(morning)$ is computed (Figure 5.5). Next, the administrator changes the time of the event to *afternoon* in the event table ($event.time_{e1}(afternoon)$). This conflicts with $schedule_{morning,g100}(e1)$, so it gets erased, which means that $event.room_{e1}(g100)$ is no longer computed and disappears (Figure 5.6).

The problem is caused by the fact that the schedule table atomically represents conjunctions of facts represented by the event table, for example the fact $schedule_{morning,g100}(e1)$ corresponds to the conjunction of facts $event.room_{e1}(g100) \wedge event.time_{e1}(morning)$. Using MCL, this is easily remedied. Consider the following MCL program:

event	owner	projection	room	time
e1	amy	no	g100	morning
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning	e1		
afternoon			
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 5.7: After scheduling e1

event	owner	projection	room	time
e1	amy	no	g100	afternoon
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning			
afternoon	e1		
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 5.8: After changing the time to the afternoon

- (1) $event.room_{e1}^{++}(R) : -event.room_{e1}^+(R)$
- (2) $event.time_{e1}^{++}(T) : -event.time_{e1}^+(T)$
- (3) $schedule_{morning,g100}^{++}(E) : -schedule_{morning,g100}^+(E)$
- (4) $event.room_{e1}^{--}(R) : -event.room_{e1}^-(R)$
- (5) $event.time_{e1}^{--}(T) : -event.time_{e1}^-(T)$
- (6) $schedule_{morning,g100}^{--}(E) : -schedule_{morning,g100}^-(E)$
- (7) $event.room_{e1}^{--}(R2) : -event.room_{e1}^+(R) \ \& \ event.room_{e1}(R2)$
- (8) $event.time_{e1}^{--}(T2) : -event.time_{e1}^+(T) \ \& \ event.time_{e1}(T2)$
- (9) $schedule_{morning,g100}^{--}(E2) : -schedule_{morning,g100}^+(E) \ \& \ schedule_{morning,g100}(E2)$
- (10) $schedule_{morning,g100}^{--}(e1) : -event.room_{e1}^+(R) \ \& \ schedule_{morning,g100}(e1)$
- (11) $schedule_{morning,g100}^{--}(e1) : -event.time_{e1}^+(T) \ \& \ schedule_{morning,g100}(e1)$
- (12) $event.room_{e1}^{++}(g100) : -schedule_{morning,g100}^+(e1)$
- (13) $event.time_{e1}^{++}(morning) : -schedule_{morning,g100}^+(e1)$

Rules (1)-(6) state that updates are successful, while rules (7)-(9) maintain the single-valued nature of the cells. Rules (10) and (11) erase $e1$ from the schedule table when its room or time is set in the event table. Rule (12) sets the value of the room of $e1$ to $g100$ in the event table when $e1$ is chosen as the event in the schedule table in the $morning, g100$. Rule (13) does the same for the $morning$ time.

Consider the following input update, applied to the empty schedule depicted in

Figure 5.4:

$$\text{schedule}_{\text{morning},g100}^+(e1)$$

Then the output update is:

$$\begin{aligned} &\text{schedule}_{\text{morning},g100}^{++}(e1) \\ &\text{event.room}_{e1}^{++}(g100) \\ &\text{event.time}_{e1}^{++}(\text{morning}) \end{aligned}$$

The result of applying the MCL program to the input update and the database extension is shown in Figure 5.7.

Next the administrator applies the following input update to the resultant spreadsheet:

$$\text{event.time}_{e1}^+(\text{afternoon})$$

Then the output update is:

$$\begin{aligned} &\text{event.time}_{e1}^{++}(\text{afternoon}) \\ &\text{schedule}_{\text{morning},g100}^{--}(e1) \end{aligned}$$

The result of applying the MCL program to the input update and the database extension is shown in Figure 5.8.

5.4 Converting Constraints to MCL

It is often desirable to modify the behavior of the insertion and deletion operators defined in Chapter 3 slightly. One way to do this is to convert the operators behavior

to Markov Change Logic, and then modify the resultant MCL program to reflect the desired behavior. In this section, we show how to convert the behavior of the update operators to MCL.

Given a set of spreadsheet constraints Ω with a finite resolution closure, we can convert Ω to MCL by following the algorithm `CONSTRAINTS-TO-MCL`. Note that, in particular, monadic constraints always have a finite resolution closure. We also define some subroutines. `MCL-HEAD` and `MCL-BODY` convert literals to their MCL equivalents in the head and body of a rule, respectively. `MCL-BODY-LITERAL` converts literals with relation p to literals with relation constant p^+ . `MAKE-SAFE` takes a sentence and returns its safe Datalog equivalent. The algorithms rely on a few other subroutines. `RELATION` takes a literal and returns the relation symbol of the literal. `ARGUMENTS` takes a literal and returns the arguments of the literal. `CNF` takes a sentence and converts it to clause normal form. `AMPERSANDIFY` takes a conjunction and replaces \wedge with $\&$.

We illustrate the `CONSTRAINTS-TO-MCL` algorithm with an example.

Example 5.2. *Consider a database with relations $p(X)$, $q(X)$, $r(X, Y)$, and $s(X, Y)$, and the following constraints Ω :*

$$\begin{aligned} &\neg p(X) \vee \neg q(X) \\ &\neg r(X, Y) \vee q(X) \end{aligned}$$

Then, the resolution closure with subsumption of the constraints is:

$$\begin{aligned} &\neg p(X) \vee \neg q(X) \\ &\neg r(X, Y) \vee q(X) \\ &\neg p(X) \vee \neg r(X, Y) \end{aligned}$$

And the output of `CONSTRAINTS-TO-MCL` is:

$$\begin{aligned} p^{++}(X) &:- p^+(X) \\ p^{--}(X) &:- p^-(X) \end{aligned}$$

$$p^{--}(X) :- q^+(X)$$

$$p^{--}(X) :- r^+(X, Y)$$

$$q^{++}(X) :- q^+(X)$$

$$q^{--}(X) :- q^-(X)$$

$$q^{--}(X) :- p^+(X)$$

$$r^{++}(X, Y) :- r^+(X, Y)$$

$$r^{--}(X, Y) :- r^-(X, Y)$$

$$r^{--}(X, Y) :- r(X, Y) \ \& \ p^+(X)$$

$$s^{++}(X, Y) :- s^+(X, Y)$$

$$s^{--}(X, Y) :- s^-(X, Y)$$

Algorithm 5.1 MCL-HEAD[ϕ]

Require: ϕ is a literal

- 1: $q := \text{RELATION}[\phi]$
 - 2: $\bar{t} := \text{ARGUMENTS}[\phi]$
 - 3: **if** ϕ has the form $\neg q(\bar{t})$ **then**
 - 4: **return** $q^{--}(\bar{t})$
 - 5: **else**
 - 6: **return** $q^{++}(\bar{t})$
 - 7: **end if**
-

Algorithm 5.2 MCL-BODY-LITERAL[ϕ]

Require: ϕ is a positive literal

- 1: $q := \text{RELATION}[\phi]$
 - 2: $\bar{t} := \text{ARGUMENTS}[\phi]$
 - 3: **return** $q^+(\bar{t})$
-

CONSTRAINTS-TO-MCL works as follows. Line 1 takes the resolution closure with subsumption of the constraints. Intuitively, this makes all of the implicit relationships between relations explicit. Line 2 loops over each relation q in our database. Lines

Algorithm 5.3 MCL-BODY[ϕ]

Require: ϕ is a conjunction of literals

- 1: write ϕ as $\phi_1 \wedge \dots \wedge \phi_k$
 - 2: **return** $\bigwedge_{i=1}^k \text{MCL-BODY-LITERAL}[\phi_i]$
-

Algorithm 5.4 MAKE-SAFE[$\phi \Leftarrow \psi$]

Require: ϕ is a literal with a $^{--}$ suffix**Require:** ψ is a conjunction of literals

- 1: $q^{--} := \text{RELATION}[\phi]$
 - 2: $\bar{t} := \text{ARGUMENTS}[\phi]$
 - 3: **if** $\phi \Leftarrow \psi$ is safe **then**
 - 4: **return** $\phi :- \text{AMPERSANDIFY}[\psi]$
 - 5: **else**
 - 6: **return** $\phi :- q(\bar{t}) \ \& \ \text{AMPERSANDIFY}[\psi]$
 - 7: **end if**
-

Algorithm 5.5 CONSTRAINTS-TO-MCL[Ω, Q]

Require: Ω is a consistent set of sentences without equality**Require:** Q is a set of relations containing the signature of Ω **Outputs:** The MCL program corresponding to constraints Ω for relations Q

- 1: $\Omega^* := \text{RESOLUTION-WITH-SUBSUMPTION}[\text{CNF}[\Omega]]$
 - 2: **for all** $q \in Q$ **do**
 - 3: **print** $q^{++}(\bar{X}) :- q^+(\bar{X})$
 - 4: **print** $q^{--}(\bar{X}) :- q^-(\bar{X})$
 - 5: **for all** $d \in \Omega^*$ **do**
 - 6: **for all** negative literals $\rho \in d$ where $\text{RELATION}[\rho] = q$ **do**
 - 7: write d as $\rho \Leftarrow \phi$
 - 8: **if** every non-equality literal in ϕ is positive **then**
 - 9: **print** $\text{MAKE-SAFE}[\text{MCL-HEAD}[\rho] \Leftarrow \text{MCL-BODY}[\phi]]$
 - 10: **end if**
 - 11: **end for**
 - 12: **end for**
 - 13: **end for**
-

3 and 4 create MCL rules that state that insertions and deletions of that relation are successful. Then line 5 loops over each clause in the resolution closure with subsumption of the constraints. Line 6 loops over each one of these containing a literal $\neg q(\bar{t})$. Line 7 rewrites the clause as a rule with $\neg q(\bar{t})$ in the head. ϕ denotes the body, which is a conjunction of literals. Line 8 checks whether the rule has no negations in the body, i.e. it is of the form $\neg q(\bar{t}) \Leftarrow p_1(\bar{t}_1) \wedge \dots \wedge p_k(\bar{t}_k)$. If so, rule 9 rewrites the rule as the MCL rule $q^{--}(\bar{t}) :- p_1^+(\bar{t}_1) \& \dots \& p_k^+(\bar{t}_k)$.

Example 5.3. *Consider a database with two relations, $p(X)$ and $q(X, Y)$, and a single constraint, $\neg p(X) \vee \neg r(X, Y)$. Then we call CONSTRAINTS-TO-MCL with $\Omega = \{\neg p(X) \vee \neg r(X, Y)\}$ and $Q = \{p, r\}$. The resolution with subsumption of the conjunctive normal form of Ω is simply itself, so $\Omega^* = \Omega$. Now we iterate through the relations in Q , namely p and r . We start with relation p . On line 3 we output $p^{++}(X_1) :- p^+(X_1)$. On line 4 we output $p^{--}(X_1) :- p^-(X_1)$. Then for each constraint in Ω^* , namely $\neg p(X) \vee \neg r(X, Y)$ (line 5), we consider all the negative literals in $\neg p(X) \vee \neg r(X, Y)$ with relation p , namely $\neg p(X)$ (line 6). We rewrite the constraint as $\neg p(X) \Leftarrow r(X, Y)$ (line 7). Every non-equality literal, namely $r(X, Y)$, is positive in the constraint (line 8), so we output $p^{--}(X) :- r^+(X, Y)$ (line 9).*

We return to line 2, considering relation r . On line 3 we output $r^{++}(X_1, X_2) :- r^+(X_1, X_2)$. On line 4 we output $r^{--}(X_1, X_2) :- r^-(X_1, X_2)$. We then iterate for each constraint in Ω^ , namely $\neg p(X) \vee \neg r(X, Y)$ (line 5), considering all the negative literals in $\neg p(X) \vee \neg r(X, Y)$ with relation r , namely $\neg r(X, Y)$ (line 6). We rewrite the constraint as $\neg r(X, Y) \Leftarrow p(X)$ (line 7). Every non-equality literal, namely $p(X)$, is positive in the constraint (line 8), so we output $r^{--}(X, Y) :- p^+(X)$ (line 9).*

We will now prove that the CONSTRAINTS-TO-MCL algorithm is sound and complete.

Theorem 5.1 (Soundness and Completeness of CONSTRAINTS-TO-MCL). *Let Ω be a consistent set of sentences without equality, and let Q be a set of relation symbols containing the signature of Ω . Let Λ be a database instance with schema Q , let λ be a set of data that is consistent with Ω , and let γ be a set of data.*

Then $\Lambda \oplus \lambda = \text{ApplyMCL}[\text{CONSTRAINTS-TO-MCL}[\Omega, Q]](\Lambda \cup \lambda^+)$ and $\Lambda \ominus \gamma = \text{ApplyMCL}[\text{CONSTRAINTS-TO-MCL}[\Omega, Q]](\Lambda \cup \gamma^-)$.

Proof. Let $P = \text{CONSTRAINTS-TO-MCL}[\Omega, Q]$. We first prove that deletion is translated correctly, i.e. $\Lambda \ominus \gamma = \text{ApplyMCL}[\text{CONSTRAINTS-TO-MCL}[\Omega, Q]](\Lambda \cup \gamma^-)$. The only rules that have “minus” literals (i.e. $r^-(X)$) in the body of a rule in P are the rules $q^-(X) :- q^-(X)$, for each $q \in Q$. Thus, $P(\gamma^-) = \gamma^{--}$, and $\text{ApplyMCL}(\gamma^-)$

$$\begin{aligned} &= \text{next}[P](P(\gamma^-)) \\ &= \text{next}[P](\gamma^{--}) \\ &= \Lambda - \text{strip}^{--}(\gamma^{--}) \cup \text{strip}^{++}(\gamma^{--}) \\ &= \Lambda - \gamma \cup \emptyset \\ &= \Lambda - \gamma \\ &= \Lambda \ominus \gamma \end{aligned}$$

Now we prove that insertion is translated correctly, i.e.

$\Lambda \oplus \lambda = \text{ApplyMCL}[\text{CONSTRAINTS-TO-MCL}[\Omega, Q]](\Lambda \cup \lambda^+)$. The only rules that have “plus” literals in the head of a rule in P are the rules $q^{++}(X) :- q^+(X)$, for each $q \in Q$. Thus, $\text{strip}^{++}(P(\lambda^+)) = \lambda$. It remains to be shown that $\text{strip}^{--}(P(\lambda^+)) = \{\psi \mid \lambda \cup \Omega \models \neg\psi\}$. We have:

$$\begin{aligned} \lambda \cup \Omega \models \neg\phi & \\ \text{iff } \{ \bigwedge_{\psi \in \lambda} \psi \} \cup \Omega \models \neg\phi & \\ \text{iff } \Omega \models \{ \bigwedge_{\psi \in \lambda} \psi \} \Rightarrow \neg\phi & \text{ (by the Deduction Theorem [20])} \\ \text{iff } \exists d \in \Omega^* \text{ and a substitution } \sigma \text{ s.t. } d\sigma \subseteq \text{CNF}[\{ \bigwedge_{\psi \in \lambda} \psi \} \Rightarrow \neg\phi] & \\ \text{(by the Subsumption Theorem (Lemma 4.6))} & \\ \text{iff } \text{MAKE-SAFE}[\text{MCL-HEAD}[d_0] \Leftarrow \text{MCL-BODY}[\bigwedge_{\psi \in \bar{d}} \psi]] \in P, & \\ \text{where } d \text{ can be written as } d_0 \Leftarrow \bar{d}, \text{ such that } d_0\sigma \subseteq \neg\phi & \\ \text{which implies that } \phi \in \text{strip}^{--}(P(\lambda^+)), \text{ as desired.} & \end{aligned}$$

Going in the other direction, we have:

$\phi \in \text{strip}^{--}(P(\lambda^+))$

iff there is a rule $r \in P$ such that $\text{strip}(r)$ subsumes $\lambda \cup \neg\phi$
 which implies that $\text{strip}(r) \in \Omega^*$
 which implies that $\lambda \cup \Omega \models \neg\phi$, as desired.

Thus $\text{strip}^{--}(P(\lambda^+)) = \{\psi \mid \lambda \cup \Omega \models \neg\psi\}$, and we are done. \square

As we discussed at the beginning of this section, the purpose of converting constraints to MCL is to then modify it. Consider the following examples.

Our first example illustrates adding a rule to modify the update behavior.

Example 5.4 (Adding rules). *We return to Example 5.1. Consider again the spreadsheet with the constraint $\neg p(X) \vee \neg q(X) \vee \neg r(X)$. Converting to MCL gives:*

$$p^{++}(X) :- p^+(X)$$

$$p^{--}(X) :- p^-(X)$$

$$q^{++}(X) :- q^+(X)$$

$$q^{--}(X) :- q^-(X)$$

$$r^{++}(X) :- r^+(X)$$

$$r^{--}(X) :- r^-(X)$$

If we want to specify that whenever $p(a)$ is inserted and $q(a)$ and $r(a)$ are true, then $q(a)$ should always be deleted and $r(a)$ retained, we could write the following additional rule:

$$q^{--}(a) :- p^+(a) \ \& \ q(a) \ \& \ r(a)$$

Our second example illustrates removing a rule to modify the update behavior.

Example 5.5 (Removing rules). *Consider the spreadsheet with the constraint $p(X) \Leftrightarrow q(X)$. Converting to MCL gives:*

$$p^{++}(X) :- p^+(X)$$

$$p^{--}(X) :- p^-(X)$$

$$q^{++}(X) :- q^+(X)$$

$$q^{--}(X) :- q^-(X)$$

$$p^{--}(X) :- p(X) \& q^+(X)$$

$$q^{--}(X) :- q(X) \& p^+(X)$$

If we wanted to change the behavior so the changing p could modify q but not vice-versa, we could remove the rule $p^{--}(X) :- p(X) \& q^+(X)$.

5.5 Dynamic Constraints

Markov Change Logic defines database change physics by mapping the current state and current update to a set of changes to be applied to the database. The astute reader might wonder why we are restricting ourselves to this simple case. Why not also allow the database changes to be based on states the database was in prior to its current state?

Most generally, we might consider arbitrary dynamic constraints, i.e. constraints on the evolution of a database. In this section, we give and analyze a formal model of dynamic constraints. We begin by giving an informal classification of the different types of dynamic database constraints. We then make the informal classifications precise. We then move to the major result of this section, The Markov Reformulation Theorem, which tells us that we can simulate arbitrary dynamic constraints with Markov constraints.

5.5.1 Types of database constraints

In this section, we give an informal overview of the different types of database constraints. In the next section, we make these informal notions precise.

A sequence of consecutive database states is called a *history*. A *static constraint* is a condition that must hold true for each individual state of a database. A *dynamic constraint* is a condition that must hold true across two or more states of a database. Examples of dynamic constraints are “salaries never decrease”, and “relation r is append-only.”

To help illustrate the different types of dynamic constraints, we turn to the game of chess. We will represent the state of a chess match with a database consisting of a single relation *board*, which represents the current state of the board. Rows are represented by the numbers 1-8, and columns by the letters $a-h$. Piece types are represented by their names (*pawn, king, queen, rook, knight, bishop*). Colors are *white* and *black*. One state is:

```
board(a, 1, rook, white)
board(b, 8, knight, white)
board(e, 1, king, white)
board(e, 8, king, black)
```

There are a number of constraints on the possible histories of the chess database, corresponding to the rules of the game. For example, each square has at most one piece on it. Since this constraint can be expressed on each individual state, independent of the rest of the database history, it is a *static* constraint. There is also a constraint that states that pieces cannot move onto squares currently holding a piece of the same color. Since this constraint depends only upon two consecutive states of the database, it is known as a *transition* constraint.

A third constraint is the en passant rule of chess, which allows a pawn to capture an opposing pawn that has just moved forward two squares as if the pawn had only moved forward one square, but only during the move immediately following the opposing pawn’s move. This can be captured with a constraint that relates three consecutive

database states. We call such constraints k -deep dynamic constraints, where in this case $k=3$.

A final constraint states that a player cannot castle if either the rook or the king involved have previously moved. Since this constraint can depend on states arbitrarily far in the past, it is a *pure dynamic* constraint.

5.5.2 Formal definitions

We now give formal definitions of the concepts intuitively explained in the previous section.

Database Histories

A *database history* is a sequence of database states. Histories always have a start state and may be finite or countably infinite. For example,

$\langle \{p(a, b), r(a)\}, \{p(a, b), q(c, d, e), r(b)\}, \{p(a, b), q(c, d, e)\} \rangle$ is a history of length 3.

The *initial state* of a database history is the first element of the history.

A *subhistory* of a database history is a consecutive subsequence of the history.

A *prefix* of a database history is a subhistory of the history that includes the initial state of the history. An n -ary prefix is a prefix of length n .

A *transition* is a database history of length 2.

Database Constraints

A *static constraint* for a database is a set of states of the database. States that are in the set are considered *valid*, while states that are not in the set are considered *invalid*.

A *dynamic constraint* for a database is a set of database histories for the database.

A history h is *valid* with respect to a dynamic constraint c iff h is a prefix of some history in c , otherwise h is considered to be *invalid*. For example, given the dynamic constraint $\{\langle\{p, q\}, \{p, r\}, \{s\}\rangle, \langle\{q, r\}, \{s, t\}, \{s\}\rangle\}$, the history $\langle\{p, q\}, \{p, r\}, \{s\}\rangle$ is valid but $\langle\{p, r\}, \{s\}\rangle$ is invalid.

The *dynamic representation* of static constraint c is a dynamic constraint $\llbracket c \rrbracket^{static}$ where a history is in $\llbracket c \rrbracket^{static}$ iff the states of the history are in c .

A *finite-depth constraint* for a database is a set of sequences of database states of length k , for some integer $k \geq 1$, known as the *depth*. The *dynamic representation* $\llbracket c \rrbracket^{depth}$ of a finite-depth constraint c of depth k is a dynamic constraint where a history h is in $\llbracket c \rrbracket^{depth}$ iff all n -length subsequences of h are in c .

A *transition constraint* for a database is a finite-depth constraint of depth 2.

An *initial-state constraint* for a database is a static constraint. The *dynamic representation* of an initial-state constraint c is a dynamic constraint $\llbracket c \rrbracket^{initial}$ where a history h is in $\llbracket c \rrbracket^{initial}$ iff the initial state of h is in c .

The *dynamic representation* of a set of dynamic constraints C is a dynamic constraint $\llbracket C \rrbracket^\cap$ such that $h \in \llbracket C \rrbracket^\cap$ iff h is in every $c \in C$.

If a dynamic constraint c is the dynamic representation of a constraint c' , we say that c is *equivalent* to c' .

A dynamic constraint c' for a database with schema R' and universe U' *maintains* a dynamic constraint c for a database with schema $R \subseteq R'$ and universe $U \subseteq U'$ iff there is a surjection m from c' to c such that for each history $h' = \langle s'_1, \dots \rangle \in c'$, $m(h') = \langle s_1, \dots \rangle$ is a history of the same length as h' , such that for each i , $1 \leq i \leq \text{LENGTH}(h)$, the restriction of s'_i to R and U equals s_i . A set of dynamic constraints C' *maintains* a set of constraints C iff $\llbracket C' \rrbracket^\cap$ maintains $\llbracket C \rrbracket^\cap$.

A history is a *valid prefix* of a dynamic constraint iff it is a prefix of some history in the dynamic constraint.

5.5.3 The Markov Reformulation Theorem

Static constraints are the simplest of all the classes of constraints given in the previous section, as they only consider individual database states. Transition constraints are the next simplest, as they consider only consecutive pairs of states. Beyond this, we have constraints that relate 3 consecutive states or more (including countably infinite numbers of states).

The natural question that arises after defining these classes of constraints is, are they all necessary? For example, in the chess example from Section 5.5.1, we formulated the castling rule as a pure dynamic constraint, using only the *board* relation. However, there is nothing inherent about the castling rule of chess that requires it to be formulated as a pure dynamic constraint. Indeed, we could introduce a new database relation called *piecemoved* that keeps track of whether a piece has moved previously. Given this modified schema, we can reformulate our castling constraint into a transition constraint.

It turns out that, as long as the database schema can be extended with auxiliary relations like *piecemoved*, it is *always* possible to maintain a set of dynamic constraints with a transition constraint and an initial state constraint. We give a proof of this below. This is an important result, as it motivates our choice of Markov Change Logic for expressing database dynamics.

We begin with a few definitions that will allow us to compactly express our theorem. We write $\langle R, U, C \rangle$ for a database with schema R , universe U , and constraints C . If $\langle R, U, C \rangle$ is a database and s' is a state of database $\langle R', U', C' \rangle$ such that $R \subseteq R'$ and $U \subseteq U'$, the *restriction* of s' to R is the database state s such that $r(\bar{t}) \in s$ iff $r(\bar{t}) \in s'$ and $r \in R$. We say that s' *extends* s from R iff s is the restriction of s' to R .

We say that a database $D' = \langle R', U', C' \rangle$ is a *reformulation* of a database $D =$

$\langle R, U, C \rangle$ iff (1) $R \subseteq R'$, (2) $U \subseteq U'$, (3) C' maintains C . A database $D = \langle R, U, C \rangle$ is *Markov* iff C is equivalent to a transition constraint and an initial state constraint. We say that a database $D' = \langle R', U', C' \rangle$ is a *Markov reformulation* of a database $D = \langle R, U, C \rangle$ iff D' is a reformulation of D and D' is Markov.

We are now ready to state the theorem.

Theorem 5.2 (Markov Reformulation). *For every database $D = \langle R, U, C \rangle$, there is a database $D' = \langle R', U', C' \rangle$ such that (1) $R \subseteq R'$, (2) $U \subseteq U'$, (3) C' is equivalent to a transition constraint and an initial state constraint, and (4) C' maintains C .*

We will give a constructive proof. Before delving into the proof, we first give an example reformulation that illustrates the construction we will use in the proof.

Example 5.5. Consider the following database:

Let $R = \{r\}$

Let $U = \{a, b, c, d\}$

Let $C = \{ \{ \langle \{r(a)\}, \{r(b)\}, \{r(c)\} \rangle, \langle \{r(a)\}, \{r(b)\}, \{r(d)\} \rangle, \langle \{r(c)\}, \{r(b)\} \rangle \}$

Let $D = \langle R, U, C \rangle$

Note that C consists of three histories. The first and the second are of length 3 and the last is of length 2. In each database exactly one fact is true, either $r(a)$, $r(b)$, $r(c)$, or $r(d)$.

We will simulate each history h in C with an initial state constraint and a transition constraint. To do this, we break apart each history into its component transitions, and mark where each transition is applicable by keeping track of (1) how deep the transition appears in the history, using a new counter relation *timestep*, and (2) the previous states of the r relation, kept in a new relation r' .

Let $R' = \{r, r', timestep\}$

Let $U' = \{a, b, c, d, 1, 2, 3\}$

Let $initial = \{\{timestep(1), r(a)\},$
 $\{timestep(1), r(c)\}\}$

Let $transition = \{$
 $\{\{timestep(1), r(a)\},$
 $\{timestep(2), r'(a, 1), r(b)\}\},$
 $\{\{timestep(2), r'(a, 1), r(b)\},$
 $\{timestep(3), r'(a, 1), r'(b, 2), r(c)\}\},$
 $\{\{timestep(2), r'(a, 1), r(b)\},$
 $\{timestep(3), r'(a, 1), r'(b, 2), r(d)\}\},$
 $\{\{timestep(1), r(c)\},$
 $\{timestep(2), r'(c, 1), r(b)\}\}\}$

Let $C' = \{\llbracket initial \rrbracket^{initial}, \llbracket transition \rrbracket^{depth}\}$

Then (1) $R \subseteq R'$, (2) $U \subseteq U'$, (3) C' is equivalent to a transition constraint and an initial state constraint, and (4) C' maintains C . \square

We are now ready to give a formal proof of the theorem.

Proof of Theorem 5.2. We give a constructive proof, by generalizing the construction we used in Example 5.5. For each relation r in R with arity m , let r' be a new relation with arity $m + 1$, and let R^* be the set of these new relations. Let $timestep$ be a new relation with arity 1. Let \mathbb{N} be the set of positive integers.

Let $R' = R \cup R^* \cup \{timestep\}$.

Let $U' = U \cup \mathbb{N}$.

Let $c = \llbracket C \rrbracket^\cap$.

Given a database state s and a positive integer k , define $\text{ARCHIVE}[s, k]$ as $\{r'(\bar{t}, k) \mid r(\bar{t}) \in s\}$. For example, given database state $\text{ARCHIVE}[\{r(b), s(c, d)\}, 2] = \{r'(b, 2), s'(c, d, 2)\}$.

For each history $h = \langle s_1, \dots \rangle \in c$, let $h' = \langle s'_1, \dots \rangle$ be a history of the same length as h such that each s'_k , $1 \leq k \leq \text{LENGTH}(h)$, is defined as $\{timestep(k)\} \cup (\bigcup_{i=1}^{k-1} \text{ARCHIVE}[s, i]) \cup s_k$. For example, for history $\langle s_1, s_2, s_3 \rangle = \langle \{r(a)\}, \{r(b)\}, \{r(c)\} \rangle$, $s'_3 = \{timestep(3), r'(a, 1), r'(b, 2), r(c)\}$.

Now, let $c' = \{h' \mid h \in c\}$

Let $C' = \{c'\}$

Let $D' = \langle R', U', C' \rangle$.

Defining m as a function from c' to c such that $m(h') = h$, we have that m is a surjection from c' to c such that for each history $h' = \langle s'_1, \dots \rangle \in c'$, $m(h') = \langle s_1, \dots \rangle$ is a history of the same length as h' , such that for each i , $1 \leq i \leq \text{LENGTH}(h)$, the restriction of s'_i to R and U is s_i . Thus c' maintains c , and therefore C' maintains C .

It remains to be shown that C' is equivalent to a set of initial state constraints and a set of transition constraints.

For each $h' \in c'$, let $h'[\text{init}] = \{s'_1\}$, and let $h'[\text{trans}]$ be the set of transitions of h' .

Let $I' = \bigcup_{h' \in c'} h'[\text{init}]$. Let $T' = \bigcup_{h' \in c'} h'[\text{trans}]$.

Let $C^* = \{\llbracket I' \rrbracket^{initial}, \llbracket T' \rrbracket^{depth}\}$

Let $c^* = \llbracket C^* \rrbracket^\cap$

We claim that $c^* = c'$. We first show that $c' \subseteq c^*$. Say that $h' \in c'$. Since every transition of h' is in $h'[trans]$, and since the start state of h' is in $h'[init]$, we see that $h' \in c^*$ as well. Thus $c' \subseteq c^*$. We next show that $c^* \subseteq c'$. Say that $h^* \in c^*$. Say for the purpose of contradiction that $h^* \notin c'$. Then there must be some finite prefix of h^* that is not a prefix of some history in c' . Let $p^* = \langle p_1^*, \dots, p_k^* \rangle$ be the shortest prefix of h^* such that p^* is not a prefix of some history in c' . If p^* is of length 1, then $p_1^* \in I'$ and thus p_1^* is the initial state of some history in c' , and p^* is a prefix of some history in c' , a contradiction. Thus p^* is of length at least 2. Since p^* is the shortest prefix of h^* such that p^* is not a prefix of some history in c' , it must be that $p^- = \langle p_1^*, \dots, p_{k-1}^* \rangle$ is a valid prefix of some $h' \in c'$. Furthermore, there must be a transition $\langle p_{k-1}^*, p_k^* \rangle \in T'$. But then $p_k^* = h'_k \cup \{timestep(k)\} \cup (\bigcup_{i=1}^{k-1} \text{ARCHIVE}[h', i])$. Thus p^* is a prefix of h' , and we have a contradiction. Thus $h^* \in c'$, and so $c^* \subseteq c'$. Since $c^* \subseteq c'$ and $c' \subseteq c^*$, $c^* = c'$. Thus $\llbracket C^* \rrbracket^\cap = \llbracket C' \rrbracket^\cap$. Since C^* is equivalent to a set of initial state constraints and a set of transition constraints, so is C' .

Thus $D' = \langle R', U', C' \rangle$ is a database with constraints such that (1) $R \subseteq R'$, (2) $U \subseteq U'$, (3) C' is equivalent to a transition constraint and an initial state constraint, and (4) C' maintains C . Since D was arbitrary, we have proved the claim. \square

5.6 Related Work

In this section, we explore the relationship of Markov Change Logic to other formalisms. We begin with a comparison to Event-Condition-Action Rules, followed by a summary of Logic Programming approaches to database update. We then explore in detail the relationship MCL has to a closely related formalism, Abstract State Machines.

5.6.1 Event-Condition-Action Rules

Markov Change Logic plays a similar role to Event-Condition-Action (ECA) rules in databases [17]. ECA rules look quite a bit like MCL rules. They are triggered by an event (a database update), check for a condition in the current database, and if the condition is true an action is fired. The action could be additional insertions or deletions, or also could include other actions with side effects such as notifications. Thus, ECA rules are more general than MCL programs, which can only insert and delete tuples from the database.

ECA rules have a procedural semantics. ECA rules are processed one at a time, and the effects of one rule might trigger other rules. Infinite loops are possible. The processing of a set of ECA rules *terminates* if, given any initial database state, the execution of the rules does not continue indefinitely; it is *confluent* if, for any initial database state, the final state is not influenced by the order of execution of the rules. In general, it is undecidable if a set of ECA rules terminates and is confluent [4, 16]. In comparison, the processing of a set of MCL rules using a Datalog engine always terminates and is always confluent. MCL can be thought of as a declarative alternative to ECA rules, which have a procedural semantics.

5.6.2 Logic Programming Approaches

There are a number of formalisms developed for updating databases. In this section, we describe attempts to extend Datalog or its close cousin, Prolog, with new update operators. Unlike MCL and ECA rules, the updates described by these formalisms are not triggered by update requests. Instead, they are triggered by explicit invocation. A typical Logic Programming update program looks like the following:

$$\begin{aligned} \text{doupd}(X, Y, Z) & :- \text{foo}(X, Y) \ \& \ \text{bar}(Y, Z) \ \& \ \text{assert}(\text{baz}(X, Y)) \ \& \ \text{retract}(\text{qux}(X, Z)) \\ \text{foo}(X, Y) & :- \text{quux}(X, Y, a) \end{aligned}$$

Where *assert* and *retract* insert and delete sentences from the database. In these languages, queries are treated as special cases of updates that have no side effects.

The formalisms differ in their exact syntax of course, but more significantly their semantics differ. The semantics of these approaches is a sequence of databases, each resulting from a insertion/deletion of a sentence or set of sentences. This differs from MCL, which defines a single set of insertions and deletions which is used to move the database to a single next state.

Prolog [15] is a procedural programming language that allows for functional terms in addition to relation constants. Prolog itself contains the operators *assert* and *retract* which allow for the dynamic insertion and deletion of individual facts and rules from the program, as well as *assertall* and *retractall* which allow for bulk insertions and deletions of facts and rules. Dynamic Logic Programming (DLP) [39] is an extension of Prolog that supports *rollback* for transactions - conditions can be checked at any time during a transaction and, if they are false, the database is left unchanged from its original state.

Abiteboul and Vianu [2] describe a family of update languages, ranging from procedural to declarative, from deterministic to non-deterministic. Some of these languages have iteration constructs, and some have constructs for producing new entities. The procedural languages allow for both insertions or deletions, whereas the declarative languages only allow for insertions.

DatalogA [41] is an extension of Datalog that allows for the insertion and deletion of tuples from the database. Like in Prolog, the extensions to Datalog are procedural. Insertions and deletions are executed tuple-at-a-time as they are evaluated, allowing for iteration constructs. DatalogU [38] is another extension of Datalog that has extensions for insertions and deletions of tuples. DatalogU has a similar syntax to DatalogA but has a set-based update semantics, as opposed to the tuple-at-a time semantics of DatalogA. Chen [13] gives another extension of Datalog that allows bulk updates, non-deterministic semantics and hypothetical reasoning.

Transaction Logic (\mathcal{TR}) [7] is a logic that allows for declarative modeling and reasoning about state transitions in databases. Transaction Logic has both a declarative and an executional semantics. \mathcal{TR} allows for arbitrary sentences to be expressed, but it also has a Horn fragment that allows for Datalog-style programs that can be used to compute the next state of a database from the current state. For example, here

is a \mathcal{TR} program called *doit* that inserts $p(a)$ into a database and then deletes $q(b)$ from a database:

$$\textit{doit} :- p.\textit{ins}(a) \otimes q.\textit{del}(b)$$

Here, $p.\textit{ins}$ and $q.\textit{del}$ are *elementary transitions*, where a transition represents a mapping from database states to database states. So, for example $p.\textit{ins}(a)$ maps each database state D into a database state $D \cup \{p(a)\}$, and $q.\textit{del}(b)$ maps each database state D into a database state $D - \{q(b)\}$. Arbitrary elementary transitions can be defined; for example elementary transitions that insert and delete multiple tuples concurrently are possible. The *serial conjunction* operator \otimes orders elementary transitions into sequences. So, the \mathcal{TR} program above, when applied to a database D_0 , first brings the database into the state $D_1 = D_0 \cup \{p(a)\}$, and then into the state $D_2 = D_1 - \{q(b)\}$. In \mathcal{TR} , the elementary transitions are opaque and atomic and so they cannot be combined into new transitions.

\mathcal{TR} is a very powerful and very general formalism. \mathcal{TR} can be used to reason about transactions over arbitrary structures, not just databases, and it can be used to express complex non-Markov constraints over those structures. It allows for transactional rollback and nondeterministic transactions. However, this power does come at the cost of complexity - a \mathcal{TR} theory is defined not just by logical sentences, but also requires the definition of several oracles. The *state data oracle* defines the structures which \mathcal{TR} reasons over, and the *state transition oracle* defines the elementary transitions which \mathcal{TR} uses to transition from one state to another. Both states and transitions are opaque in \mathcal{TR} and cannot be compared or combined, only serialized. MCL can be used to define elementary transitions for the state transition oracle. Thus, MCL and \mathcal{TR} are complimentary formalisms.

5.6.3 Abstract State Machines

Other than ECA rules, the most closely related formalism to MCL is the Abstract State Machine. In this section, we give a theoretical comparison of MCL and Abstract

State Machines. We will show that MCL is more expressive than ASMs without import constructors (which allow for the nondeterministic introduction of symbols into the language).

In first-order logic, an *algebra* is a structure without relations. *Abstract State Machine* (ASM) [27] is a mathematical system which describes dynamic algebras in a fashion similar to how Markov Change Logic describes dynamic databases.¹ An ASM is specified by an algebra and a set of transition rules. The transition rules completely specify the next state of an algebra given the current state of the algebra. ASMs allow for external functions that change from state to state and are meant to represent inputs into the system.

There are many variations of the ASM model. Here we focus on Sequential ASMs, which are the most basic type of ASM and allow for ground transition rules, and Parallel ASMs, which allow for variables in the transition rules. A formal definition of ASMs can be found in Appendix 3.

ASMs and MCL have several analogous components, shown in the following table:

MCL	ASMs
Database State	Structure
Relation	Function
Entity	Nullary function
Datalog Rule	Update rule
Datalog Program	ASM Program

Both ASMs and MCL have Markov update semantics, and both define changes to the current state and define the new state as the current state modified by the changes. Also, both ASMs and MCL fire all of their rules in parallel rather than one after another, as for example is done with Event-Condition-Action rules in active databases.

In Appendix 2, we prove that MCL is more expressive than ASMs without import constructors (which allow for the nondeterministic introduction of symbols into the

¹Abstract State Machines were formerly known as *evolving algebras*.

Table 5.1: Database Physics Formalisms

Formalism	Procedural / Declarative	Deterministic / Nondeterministic	Bulk Update?	Single Step / Multistep	Supports Rollback?
MCL	Declarative	Deterministic	Yes	Single Step	No
ECA Rules [17]	Procedural	Nondeterministic	Yes	Multistep	Yes
Transaction Logic [7]	Declarative	Nondeterministic	Yes	Multistep	Yes
ASMs [27]	Declarative	Deterministic	Yes	Single Step	No
Prolog [15]	Procedural	Deterministic	Yes	Multistep	No
DLP [39]	Procedural	Deterministic	Yes	Multistep	Yes
DatalogA [41]	Procedural	Deterministic	No	Multistep	No
DatalogU [38]	Procedural	Deterministic	Yes	Multistep	No
Chen [13]	Procedural	Nondeterministic	Yes	Multistep	No
Abiteboul and Vianu [2]	Both	Both	Yes	Both	No

language).

5.6.4 Summary

We summarize related work with the following table. We consider whether each approach is procedural or declarative, whether it is deterministic or nondeterministic, whether the approach allows for bulk updates or just tuple-at-a-time updates, and whether the approach allows for multi-step operations or just single-step operations to be defined, and whether the approach allows for rollback or not.

Chapter 6

Websheets

In this chapter we describe an implementation of a logical spreadsheet system called Webcell. Webcell is a Javascript library and can be used to turn Web pages into logical spreadsheets, or *Websheets*. We explain how Webcell is implemented and how it can be used. We also describe the application of Webcell to the Stanford Computer Science Master's Program sheets.

6.1 Webcell

Webcell consists of a single Javascript library.¹ It can be used to turn any Web page into a Web-based logical spreadsheet. Deploying Webcell is simple: import the library into the Web page via a `<script>` tag, and place a set of static constraints and MCL rules into a `<div>` with a special `id`. Upon page load, Webcell parses the constraints and MCL rules, places them in data structures, and assigns a callback function to each cell in the Web page that is called any time the value in that cell is changed. That function calculates the propagated values using a bottom-up Datalog computation and then assigns the calculated values to the appropriate cells.

The HTML for a simple Web page that uses Webcell is shown in Figure 6.1. The Web page consists of a form containing three text fields, with `ids` `p`, `q`, and `r`. There

¹In case the reader is curious: Webcell contains about 1500 lines of code. It took about two man-months to develop.

```

<html>
<head>
<title>Webcell Example</title>
<script type='text/javascript' src='webcell.js'></script>
</head>
<body onload='initialize()'>
<form>
<input type='text' id='p' />
<input type='text' id='q' />
<input type='text' id='r' />
</form>
<div id='rules'>
illegal :- val(p,a) & val(q,a)
val(style(p,color),green) :- val(p,b)
pos(r,X) :- minus(p,X)
</div>
</body>
</html>

```

Figure 6.1: HTML for a simple Websheet

are three rules. The first states that p cannot have value a at the same time that q has value a . Placing value a into p and then placing value a into cell q will result in value a being automatically removed from cell p , and vice versa. The second rule states then when p has value b , then the color of p is **green**. The final rule states that when a value is removed from cell p , the value of r is changed to be that value. Finally, a call to `initialize` is made after the page is loaded. This loads the rules and initializes the state of the Websheet.

6.1.1 Rule Syntax and Semantics

As usual, variables start with capital letters while entities and relations start with lowercase letters or numbers.

There are three basic types of rules in Webcell: illegalities, views, and MCL rules.

Illegalities implement categorical deletion. An example illegality is:

```
illegal :- val(p,a) & val(q,a)
```

This rule states that it is incompatible for p to have value a at the same time that q has value a . If value a is inserted into cell p when q currently has value a , then a is deleted from cell q , and vice versa.

Views define values of cells in terms of other values. For example:

```
val(p,X) :- val(q,X) & val(r,a)
```

This defines the value of p to be the value of q whenever r has value a .

MCL rules define dynamic changes in response to insertions and deletions. For example:

```
pos(r,X) :- minus(p,X)
```

This has the effect of setting the value of r to whatever value is deleted from p .

Negation is allowed, using the \sim operator. For example:

```
val(p,X) :- val(q,X) &  $\sim$ val(r,a)
```

This defines the value of p to be the value of q whenever r does not have value a .

Rules have standard Datalog semantics, with the exception that the lack of a value in a cell is considered an unknown. Since cells are single-valued, then, $\sim r(a)$ is considered true whenever r has a value that is not a . If r is empty, $\sim r(a)$ is not considered true or false. The `__blank` keyword is used to indicate that a cell is currently blank.

For example:

```
val(p,X) :- val(q,X) & val(r,_blank)
```

This defines the value of `p` to be the value of `q` whenever `r` is empty.

Accessing the Document Object Model

The Document Object Model, or DOM, is the data structure in which page elements are accessed in a browser [55]. The DOM contains all Web page information such as form values, static text, object attributes, and visual styles such as colors and fonts. Webcell allows any element in the DOM that has an `id` to be treated as a cell. For form fields such as text fields, drop down lists, and checkboxes, the `value` of the cell is the value of the form field. For DOM elements such as `divs` without a `value` attribute, the value of the cell is the elements' `innerHTML` property.

Webcell also allows access to an element's `style` and `attribute` properties. For example, the following disables a button `p` when `q` is empty:

```
val(attribute(p, disabled),yes) :- val(q,_blank)
```

The following sets the color of `p` to `red` when `q`, `r`, and `s` have the same value:

```
val(style(p,bgColor),red) :- val(q,X) & val(r,X) & val(s,X)
```

Procedural attachments

A *procedural attachment* is a relation that is interpreted rather than explicitly stored. Procedural attachments are useful for implementing infinite relations such as addition or the “greater than” relation. Webcell allows any Javascript function to be used as

a procedural attachment. Procedural attachments are implemented using two keywords, `evaluate` and `execute`.

`execute` is used to execute a Boolean function. If the function returns a value that is interpreted as true in Javascript, the `execute` literal is interpreted as true; otherwise, it is interpreted as false. For example:

```
val(p,X) :- val(q,X) & val(r,Y) & execute(geq(X,Y))
```

This defines the value of `p` to be the value of `q` whenever the value of `q` is greater or equal than the value of `r`.

`evaluate` is used to evaluate a function that returns a value. For example:

```
val(p,Z) :- val(q,X) & val(r,Y) & evaluate(plus(X,Y),Z)
```

This defines the value of `p` as the sum of cells `q` and `r`.

6.1.2 Structured cell names

It is sometimes useful to write rules that apply to many cells. In Webcell, we can achieve this functionality by using *structured cell names*, which are cell names that are structured as functional terms. Examples of structured cell names are `q(3)`, `schedule(morning,g100)`, and `p(g(a,b),c,d)`. There is no limit to the depth of the structure or on the arity of the structured term. Structured cells names are assigned to cells as usual. For example, a `div` with cell name `q(3)` would be written `<div id='q(3)'\>`.

Webcell treats rules containing structured cells as rule schemas. For each instantiation of the variables contained in a structured cell term, a rule is created. So, for example, consider the rule:

```
val(p(X),Y) :- val(q,Y)
```

Given a spreadsheet with cells $p(a)$, $p(b)$, $p(c)$, and q , Webcell would transform the rule into three rules:

```
val(p(a),Y) :- val(q,Y)
val(p(b),Y) :- val(q,Y)
val(p(c),Y) :- val(q,Y)
```

Using structured cell names, it is easy to write the room manager constraints of Figure 1.2 concisely. For example, a rule defining the schedule table as a view of the event table would look like:

```
val(schedule(T,R),E) :- val(event.time(E),T) & val(event.room(E),R)
```

Structured cells were also used in the logical spreadsheet PrediCalc [35]. Structured relation names can be found in the logic programming language HiLog [14].

Formal Syntax

Figure 6.2 gives the Bachus–Naur form for the Webcell rules. Additionally, we require that the rules are safe.

6.2 Application: Program Sheets

We now turn to a deployed application of Webcell. The application is a Websheet used to help students create their course plans for their master’s degrees in computer science at Stanford. It contains constraints which encode the graduation requirements of the program.

```

⟨rule⟩ ::= ⟨illegalRule⟩ | ⟨viewRule⟩ | ⟨mclRule⟩
⟨illegalRule⟩ ::= illegal :- ⟨illegalBody⟩
⟨viewRule⟩ ::= ⟨val⟩ :- ⟨viewBody⟩
⟨mclRule⟩ ::= ⟨plusRule⟩ | ⟨minusRule⟩
⟨plusRule⟩ ::= ⟨static⟩ ⟨plus⟩ ⟨static⟩
⟨minusRule⟩ ::= ⟨static⟩ ⟨minus⟩ ⟨static⟩
⟨illegalBody⟩ ::= ⟨valLiteral⟩ ⟨attachments⟩ & ⟨valLiteral⟩ ⟨attachments⟩
⟨viewBody⟩ ::= ⟨static⟩*
⟨mclHead⟩ ::= ⟨pos⟩ | ⟨neg⟩
⟨valLiteral⟩ ::= ⟨val⟩ | ~ ⟨val⟩
⟨val⟩ ::= val( ⟨structuredCell⟩ , ⟨argument⟩ )
⟨attachments⟩ ::= (& ⟨attachmentLiteral⟩)*
⟨attachmentLiteral⟩ ::= ⟨attachment⟩ | ~ ⟨attachment⟩
⟨attachment⟩ ::= ⟨execute⟩ | ⟨evaluate⟩
⟨static⟩ ::= (⟨valLiteral⟩ | ⟨attachmentLiteral⟩)*
⟨pos⟩ ::= pos( ⟨structuredCell⟩ , ⟨argument⟩ )
⟨neg⟩ ::= neg( ⟨structuredCell⟩ , ⟨argument⟩ )
⟨plus⟩ ::= plus( ⟨structuredCell⟩ , ⟨argument⟩ )
⟨minus⟩ ::= minus( ⟨structuredCell⟩ , ⟨argument⟩ )
⟨execute⟩ ::= execute( ⟨constant⟩ ( ⟨argumentList⟩ ) )
⟨evaluate⟩ ::= evaluate( ⟨constant⟩ ( ⟨argumentList⟩ ), argument )
⟨structuredCell⟩ ::= ⟨argument⟩ | ⟨constant⟩ ( (⟨structuredCell⟩)+ )
⟨argumentList⟩ ::= ⟨argument⟩ (& ⟨argument⟩)*
⟨argument⟩ ::= ⟨constant⟩ | ⟨variable⟩
⟨variable⟩ ::= ⟨uppercaseLetter⟩ (⟨token⟩)*
⟨constant⟩ ::= ⟨constantStart⟩ (⟨token⟩)*
⟨constantStart⟩ ::= ⟨lowercaseLetter⟩ | ⟨digit⟩ | ⟨punctuation⟩
⟨token⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨punctuation⟩
⟨letter⟩ ::= ⟨uppercaseLetter⟩ | ⟨lowercaseLetter⟩
⟨uppercaseLetter⟩ ::= A | ... | Z
⟨lowercaseLetter⟩ ::= a | ... | z
⟨digit⟩ ::= 1 | ... | 9
⟨punctuation⟩ ::= - | . | -

```

Figure 6.2: Bachus–Naur Form for Webcell rules

6.2.1 Background

By the end of their first quarter, students enrolled in the Master of Science in Computer Science (MSCS) Program are required to submit a MSCS program sheet and have it approved by their advisor and the MSCS program administrator. A MSCS program sheet is a plan that details the courses that the student is to take before graduation.

It is the job of the student's advisor to ensure that his students' program sheets are filled out properly, to approve modifications to the standard program, and to determine whether courses taken elsewhere satisfy the requirements. The MSCS program coordinator double checks that the program sheets are properly filled out, approves serious deviations from the standard program, and more generally oversees the entire process, overseeing the content of the the program sheets and making sure that all students get their program sheets filled out on time.

Currently about 80 people use the online program sheets - about 70 students, 10 professors, and the MSCS program coordinator. This represents all of the active MSCS advisors and about half of the active students.

Prior to the introduction of the online program sheets, all program sheets were paper forms. While functional, this had several drawbacks:

- All parties had to be physically present view and approve program sheets. If, for example, an advisor is traveling, he is unable to approve his students' program sheets.
- It was difficult to manage the process. With paper sheets, it was difficult for advisors to keep track of what the status of each student's sheet is, or even to know what his current list of students is.
- Validation was difficult. There are many rules which must be followed and requirements that must be satisfied. It was easy to miss a requirement, break a rule, or make an arithmetic error.

The first two drawbacks are remedied by bringing the program sheets online. The last one is remedied by making them logical spreadsheets.

FOUNDATIONS REQUIREMENT

You must satisfy the requirements listed in each of the following areas; all courses taken elsewhere must be approved by your advisor on a **foundation course waiver form**. Required documents for waiving a course include course descriptions, syllabi, and textbook lists. These documents can be organized here: cs.stanford.edu/degrees/mscs/waivers/. Do not enter anything in the "Units" column for courses taken elsewhere. Signed foundation course waiver forms should be given to Claire in Gates 182.

Note: Enter "**other Stanford degree**" in the approval column for courses that you have applied to another Stanford degree. Enter "**waiver on file**" for courses your advisor has waived via a foundation course waiver form.

Required:	Equivalent elsewhere (course number/title/institution)	Approval	Grade	Units
Logic, Automata and Complexity (<input checked="" type="checkbox"/> CS 103)	<input type="text"/>	<input type="text"/>	<input type="text"/>	3
Probability (<input type="checkbox"/> CS 109, <input type="checkbox"/> STATS 116, <input type="checkbox"/> CME 106, or <input type="checkbox"/> MS&E 220)	<input type="text"/>	<input type="text"/>	<input type="text"/>	
Algorithmic Analysis (<input checked="" type="checkbox"/> CS 161)	<input type="text"/>	<input type="text"/>	<input type="text"/>	
Computer Organization and Systems (<input checked="" type="checkbox"/> CS 107)	CS314/Computer Architecture/Cornell	<input type="text"/>	<input type="text"/>	0
Principles of Computer Systems (<input checked="" type="checkbox"/> CS 110)	<input type="text"/>	<input type="text"/>	<input type="text"/>	
TOTAL UNITS USED TO SATISFY FOUNDATIONS REQUIREMENT:				3

Note: This total may not exceed 10 units.

Figure 6.3: Foundations requirements for a Stanford Computer Science Master's Program sheet. The Probability requirement is red as it is unfulfilled.

6.2.2 Implementation

MSCS program sheets consist of several sections, each with its own set of requirements. We will describe the first section, Foundations, in detail. The other sections are similar.

Figure 6.3 shows the Foundations section of a Master's Program sheet. The Foundations section has 5 required topics for which the student is required to take a course in, such as Probability, Algorithmic Analysis, and Principles of Computer Systems. The student can either take a course at Stanford to fulfill the requirement, or the student can petition to substitute an equivalent course he took elsewhere. For example, a student must choose exactly one of CS 109, STATS 116, CME 106, or MS&E 220 to fulfill the Probability requirement.

This can be represented with the following rules. For each combination of courses, there is a rule stating that at most one can be chosen to fulfill the probability requirement. This is done with a set of rules like:

```
illegal :- val(prob_cs109,yes) & val(prob_stats116,yes)
```

This states that CS 109 and STATS 116 cannot both be chosen to fulfill the Probability requirement. Furthermore, to alert the user in case the requirement is unfulfilled, the following rule is in place:

```
val(style(probability_prompt,color),red) :- val(prob_cs109,__blank) &
    val(prob_stats116,__blank) & val(cme106,__blank) & val(mse220,__blank)
```

The Foundations section also contains the total of the number of units taken, up to a maximum of 10 units. This is accomplished with rules like the following. First, a number of units for each course taken is computed. If that total is given explicitly, we use it; otherwise we define it to be zero:

```
val(prob_units,X) :- val(num_prob_units,X)
val(prob_units,0) :- val(num_prob_units,__blank)
```

The total number of foundations units is then defined as follows:

```
val(foundations_total,Z) :- val(logic_units,X1) & val(prob_units,X2)
    & val(alg_units,X3) & val(org_units,X4) & val(systems_units,X5)
    & evaluate(plus(X1,X2,X3,X4,X5),Y) & evaluate(max(Y,10),Z)
```

There are also some dynamic rules. When a value is entered for the Equivalent Elsewhere, the Units field for that requirement is set to 0. For example:

```
pos(prob_units,0) :- plus(prob_equiv,X)
```

The rules are generated automatically in code, generated based upon a database of the MSCS requirements.

6.3 Related Work

There are two basic styles in which form validation is done in Web pages. In the “submit and refresh” style of validation, if a form is submitted with invalid data, the submission is rejected and a list of errors is presented to the user, who may then make changes and submit again. Alternatively, a form can provide immediate feedback to the user, bringing attention to constraint violations in real-time via inline messages. The later approach has several advantages over the former, including faster form completion, greater completion rates, decreased error rates, and higher user satisfaction [56].

Logical spreadsheets are a natural way to implement real-time form validation. Besides Webcell, there are several other Javascript toolkits on the Web that can be used to do real-time validation of data, e.g. the jQuery validation plugin² and LiveValidation³. Like WebCell, these toolkits provide instant validation of user input; however they do so by using JavaScript callback functions, and do not contain built-in support for declarative constraints. The WUI framework [28] allows for declarative type constraints to be declared, and for the constraints to be validated both client-side and server-side. However, WUI does not provide instant feedback as the user makes changes to the form, providing feedback only as the form is submitted. Furthermore it only allows for type constraints (though these may span multiple fields, e.g. a date picker composed of inputs for the day, month, and year), and not for general semantic constraints. PowerForms [9], is a toolkit that provides instant semantic constraint checking of Web forms. It has an XML-based language that can be used to write complex rules that define whether a field is currently valid, depending on the current values of other fields. Its constraint language is less expressive than that Webcell’s constraint language, as it is lacking variables. Thus a simple constraints such as “these two fields should contain the same value” may not be expressed in PowerForms. Also unlike Webcell, Powerforms does not provide capabilities to compute cell values or CSS styles based on constraints.

²<http://docs.jquery.com/Plugins/Validation>

³<http://livevalidation.com/>

Chapter 7

Conclusion

And there you have it - the logical spreadsheet! This dissertation has explained how to update them, how to display their consequences under inconsistency, and how to modify their dynamic behavior.

In this chapter, we take a look at the issues around logical spreadsheets that were not covered by this dissertation. In Section 7.1, we discuss incorporating numerical constraint solving into logical spreadsheets. Section 7.2, we discuss further research that will be required to make logical spreadsheets more accessible to users. In Section 7.3 we discuss unsolved issues surrounding Websheets. In Section 7.4 we discuss the the issues involved in building multiuser and interlinked spreadsheets. In Section 7.5 we discuss how to determine further uses of logical spreadsheets.

7.1 Numerical Constraints

Logical spreadsheets are good at logical reasoning, while traditional spreadsheets are good a numerical reasoning. Ideally, we could combine the best features of both into a spreadsheet that could allow for multidirectional propagation and many-to-many relationships with both logical and numerical constraints. Constraint Logic Programming [33] brings together logic and numerical constraint solving and should be adaptable to an interactive spreadsheet setting. The main challenges here are (1) figuring out how to deal with inconsistent values in a numerical setting, and (2)

dealing with the fact that systems of equations may take a long time to solve, which limits the interactivensness of a such a system. As an example of (1), let's say that we want to invert a matrix, one of whose values is inconsistent with the constraints. What should the resultant matrix be? Should it be a partial matrix of some sort? What should the eigenvectors of the matrix be? And so on.

7.2 Accessibility

This dissertation describes the core theoretical ideas behind a logical spreadsheet. However, before logical spreadsheets can fulfill their promise as a tool for the masses in the same way that traditional spreadsheets have, more work must be done. Most notably, the constraint editor and the Markov Change Logic editor must be made more accessible. We consider each in turn.

Constraint editor

There two main approaches to editing logical expressions: textual editors and structured editors. Both approaches have their strength and weakness. We discuss each in turn, and then turn to the issue of macros.

Textual editors. Textual editors allow people to express constraints directly as formal logical expressions written in an unstructured textual environment. Webcell is an example of a system in which the constraints are written this way. The constraint syntax is important. Webcell uses Datalog syntax and semantics for rule writing. This is to be contrasted with PrediCalc [35], which used a traditional logical syntax and semantics for writing rules (using `and`, `or`, `not`, `<=>`, `=>` and `<=`). A third approach championed by Michael Genesereth is to use rules with conjunctive bodies and disjunctive heads, where negation is not allowed anywhere in the rule. The lack of negation may make the rules easier to understand for some people. It is unclear which of these approaches is the best overall, or whether there is yet another approach that is superior to these three. A user study would be required to help sort this out.

Structured editors. A number of commercial rule writing systems provide structured environments to aid users in writing rules. These systems provide a structured

way to enter in logical expressions, with buttons that allow the user to add a new conjunct or disjunct, drop down lists of relation names from which to choose from, and so on. Query by Example [57] is an example of this type of environment.

Some of these environments attempt to make rule writing more accessible to users by allowing them to create rules using a structured natural language. For a good overview on controlled natural languages, see [43]. Other rule writing environments allow the user to create rules using graphical representations inspired by semantic networks [50], where nodes are predicates and edges represent attributes.

Analysis. Both textual and structured editors have their place. Textual editors are most useful to power users. Features like copy-and-paste and search-and-replace make writing large numbers of expressions easy. On the other hand, their unstructured nature can be daunting for casual users; the structured editors provide substantially more guidance and prohibit users from making mistakes.

In our experience, the most difficult conceptual leap for users is in using variables. While users understand concrete examples very well, variables bring substantially more conceptual complexity for users. We see research into how best to expose variables to users as being particularly important in bringing the full power of logical spreadsheets to a broad class of users.

Macros. In addition to writing constraints directly, both textual and structured editors can allow for macros like “at most n of these cells can have a value at once” and “the values of all of these cells must be distinct.” The particular macros that are useful depend on the application, though there appear to be certain macros that come up quite often across applications. For example, the “at most n of these cells can have a value at once” macro was used in the MSCS Program Sheet application, and appears to be a useful one across applications as well.

Markov Change Logic editor

Markov Change Logic (MCL) is a powerful mechanism for expressing all kinds of dynamic policies for a spreadsheet or database. However, large MCL programs can be difficult to understand. It can be difficult to determine if a large MCL program does what it is intended to, whether the rules cover all the cases that one cares about, etc.

The problem is particularly bad in databases, where many facts can be inserted and deleted at a time. For example, consider the following MCL program which maintains the view $p(X) :- q(X) \ \& \ r(X) \ \& \ s(X)$:

$$\begin{aligned}
p^{++}(X) & :- q^+(X) \ \& \ r(X) \ \& \ \neg r^-(X) \ \& \ s(X) \ \& \ \neg s^-(X) \\
p^{++}(X) & :- q(X) \ \& \ \neg q^-(X) \ \& \ r^+(X) \ \& \ s(X) \ \& \ \neg s^-(X) \\
p^{++}(X) & :- q(X) \ \& \ \neg q^-(X) \ \& \ r(X) \ \& \ \neg r^-(X) \ \& \ s^+(X) \\
p^{++}(X) & :- q^+(X) \ \& \ r^+(X) \ \& \ s(X) \ \& \ \neg s^-(X) \\
p^{++}(X) & :- q^+(X) \ \& \ r(X) \ \& \ \neg r^-(X) \ \& \ s^+(X) \\
p^{++}(X) & :- q(X) \ \& \ \neg q^-(X) \ \& \ r^+(X) \ \& \ s^+(X) \\
p^{++}(X) & :- q^+(X) \ \& \ r^+(X) \ \& \ s^+(X) \\
p^{--}(X) & :- q^-(X) \ \& \ r(X) \ \& \ s(X) \\
p^{--}(X) & :- q(X) \ \& \ r^-(X) \ \& \ s(X) \\
p^{--}(X) & :- q(X) \ \& \ r(X) \ \& \ s^-(X)
\end{aligned}$$

We must consider the cases when one, two, and three relations are inserted a time, considering also whether deletions are occurring in those relations. And this just to maintain a single view!

We have two ideas to explore for simplifying the MCL editing process. The first is to specify only deviations from the standard update operators defined in Chapter 3. The second is to create higher level languages for specifying MCL programs. We explore each of these in turn.

Deviations. In Definition 3.1 and Definition 3.4, we define insertion and deletion operators for logical spreadsheets. In Section 5.3.2 we show how to convert from these operators to a MCL program that behaves identically to those operators for a given set of constraints. Thus, one way to modify the behavior of a logical spreadsheet is to take a set of constraints, convert it to an equivalent MCL program, and then modify that MCL program to one's liking. This works of course, but it requires working with a potentially large set of MCL rules, which, as we mentioned above, can be difficult to understand.

To reduce the number of rules, instead of converting the constraints to an MCL

program, we can use the insertion and deletion operators to generate a set of insertions and deletions, use the MCL program to generate additional insertions and deletions, and then apply all of the insertions and deletions at once. For example, consider a spreadsheet with data $q(a)$ and a constraint $\neg p(a) \vee \neg q(a)$. Say also that we have a single MCL rule $r^{++}(b) : - p^+(a)$. Inserting $p(a)$ would then lead to $q(a)$ being deleted (via the insertion operator) and $r(b)$ being inserted (via the MCL rule), leading to state $r(b)$.

This works as long as one only needs to add in additional insertions and deletions. To block insertions and deletions from taking place that are generated by the insertion and deletion operators, we can change Markov Change Logic itself. For example, we could define an additional modality $p^{\circ\circ}(a)$, which means that $p(a)$ is *not* to be inserted or deleted, and which overrides the output of the insertion and deletion operators. For example, adding in the rule $q^{\circ\circ}(X) : - p^+(X)$ to the example from the previous paragraph would nullify the deletion of $q(a)$, resulting in the state $q(a), r(b)$.

Higher Level Languages. Markov Change Logic is a “low level” language for expressing database physics policies. We can create higher level languages that “compile down” to MCL, in the same way that high level languages compile down to machine code. These languages might not be as expressive as MCL, but might be well suited to certain tasks.

For example, for consistency maintenance, a simple priority scheme can be defined to determine how to keep cells consistent. In the scheme, each cell is provided a unique priority, and when a conflict is to occur between a set of cells and the constraints, the lowest priority cell’s value is removed to maintain consistency. For example, consider data $\{p(a), q(a)\}$, constraint $\neg p(a) \vee \neg q(a) \vee \neg r(a)$, where priorities of the cells p , q , and r are 1, 2, and 3, respectively, and a lower number means a higher priority. Then upon an insertion of insertion $r(a)$, which would lead to a constraint violation, $r(a)$ is deleted since the priority of r is lower than that of p . This simple scheme it is not as expressive as MCL - for example, it does not allow one to express conditional statements like “when p has value a , (use prioritization scheme A); when p has value b , (use prioritization scheme B).” Nor does it allow one to express MCL statements that do not have anything to do with consistency maintenance. Nonetheless, its simplicity

makes it very appealing for the circumstances in which it does work.

Algorithms for translating from this scheme and others to MCL would make it a lot more accessible to users. Of course, one could also work directly with these schemes and never translate them to MCL. MCL is only necessary to deviate from these schemes, in the same manner as described above.

7.3 Websheets with Large Datasets

Webcell assumes that all data is available locally to the Websheet on the client machine. However, this is not always the case. In particular, it may be the case that the amount of data is so large that having the client hold the data is either impractical or impossible. For example, given today's hardware limitations, one could not reasonably expect a browser to hold information on all people in the United States. However, constraints might still reference information about people, for example to ensure that people have unique Social Security numbers. In these situations, our options are to:

- (1) Do all processing on the client, but send over data from the server to the client piecemeal.
- (2) Do all processing on the server, possibly requiring the client to send data to the server.
- (3) Do some processing in the client, and do some processing on the server.

In each of these cases, the timing and order of the in which the data is sent and processed can make a big difference in the efficiency and responsiveness of the Websheet. For example, sending over all data to the client when the Websheet is first loaded will slow down the load time of the Websheet, but putting off the loading of the data until later may increase the response time of the Websheet upon user changes.

Algorithms and heuristics are needed to determine the best strategies for data transfer and processing. Ideal algorithms will need to take into account factors such

as network speed, the relative speed of processing on the client versus on the server, memory constraints, join selectivities, usage patterns, and so on.

7.4 Multiuser and Interlinked Spreadsheets

In this dissertation, we have assumed that logical spreadsheets are single-user, isolated systems. We can generalize this in two ways: by allowing multiple users to access a spreadsheet, and by allowing spreadsheets to contain references to each others' cells.

To allow for multiple users, the spreadsheet must ensure that different users do not clobber each others changes and so forth. The issues here are not drastically different from the issues that arise in any multiuser application; techniques like locking, merging, and so forth all come into play. A somewhat more interesting case is when each user is only allowed to view and modify part of the spreadsheet. A number of issues arise here such as how to define security policies, and how to display constraint violations against cells which a user is not allowed to see.

Interlinked spreadsheets are spreadsheets that contain constraints that reference cells of other spreadsheets. The interlinked spreadsheets may have different schemas. For example, consider a spreadsheet that contains a value in meters, and another spreadsheet that contains the same value expressed in yards, and a constraint relating the two values. Changing the value of one of these cells would change the value of the other.

Interlinked spreadsheets are conceptually similar to multiuser spreadsheets, in the sense that each interlinked spreadsheet can be thought of as part of a large “universal” spreadsheet, which can only be accessed by certain users. Where interlinked spreadsheets differ is that the different spreadsheets may be running on different machines. Thus, there is a question of where constraint processing takes place. The spreadsheets must either collaboratively decide where processing takes place in a peer-to-peer fashion, or there must be some central facilitator that manages where the computation takes place.

As with single user, isolated spreadsheets, we can turn multiple Web pages into

interlinked Web sheets. Taking the idea to its extreme, we can imagine a “World Wide Websheet” in which all Web pages can be connected with constraints, and propagation and constraint violations occur across Web pages. The question of what processing should be done on the clients, the servers, and possibly the facilitator(s) arises, further complicating the issues.

7.5 Applications

What further uses do logical spreadsheets have in the world? What users would most benefit from the technology? It would be interesting, for example, to investigate the types of rules that advanced users of traditional spreadsheets write, which might be better written as logical rules. Furthermore, these same users should be asked what sort of rules they are unable to write but would like to. Similarly, it would be interesting to do a study of how Web page creators might benefit from logical spreadsheet technology like Webcell, and what types of uses are the most desired? Studies like these would help tremendously in determining how to use logical spreadsheet technology to change the world in the most effective way.

Appendix A

Epilogue

This thesis is the result of many years of thought, application, and refinement of this idea. In particular, we spent years discussing what the default semantics for updating logical spreadsheets should be. The semantics went through many iterations and refinements. The major breakthroughs were: (1) having a bilevel update semantics (separating the base values from the computed values), (2) developing existential Ω -entailment for computing values in the presence of inconsistencies, (3) dealing with multiple computed values by leaving cells blank, (4) only allowing base values to be deleted - not computed values, and (5) allowing for ambiguities to be resolved using Markov Change Logic. The resultant semantics is simple, elegant, and most importantly, works in a way that users find intuitive.

There are still a few controversial decisions which we have made. The major part of our update semantics which we sometimes find that people push back on is on categorical deletion - i.e. deleting values that conflict with an update and the constraints. Indeed, there are situations in which this is not the desired behavior (though, we believe, there are more situations in which it is the desired behavior). Spreadsheet systems in practice need to make the decision of whether to use categorical deletion or not up to the spreadsheet administrator - indeed, on a cell by cell basis.

Also, making deletion simple set subtraction is controversial. For many years, we experimented with a semantics that propagated deletions as well. For example, given two cells, p and q , and one constraint, $p(X) \Rightarrow q(X)$, and a current spreadsheet

instance of $\{p(a), q(a)\}$, one could argue that deleting $q(a)$ should delete $p(a)$ as well, since $p(a)$ plus the constraints logically entails $q(a)$. Ultimately, we decided that removal of a value should simply mean that the value is currently unknown, as opposed to that it means that the negation of the value is asserted. While there are cases when the other semantics are desirable, but we felt that the simple semantics was intuitive and worked as most people expected. Markov Change Logic can be used to make additional removals if desired.

Another controversial decision is to use Herbrand Logic for constraints, as opposed to Datalog. Using Herbrand Logic has the advantage that it treats blank cells as unknown rather than as empty. In other words, if a cell p is blank, then neither $p(a)$ nor $\neg p(a)$ is true. This is semantics I feel is most natural: when presented with an empty set of cells, the user has not asserted that the cells have no value, but simply hasn't had the chance to select values for the cells yet. In addition, Herbrand Logic is explosive, which nicely illustrates the need for a paraconsistent consequence relation. Existential Ω -entailment has a natural definition in Herbrand Logic as well. On the other hand, Herbrand Logic is less expressive than Datalog: it cannot express transitive closure for example (whereas Datalog can because of its minimal model semantics). In addition, one cannot write constraints referencing blank cells in Herbrand Logic, whereas one can in Datalog due to its use of negation-as-failure. This tension led me to decide to use Herbrand Logic when developing the core ideas behind logical spreadsheets, while using a variant of Datalog in the Webcell implementation of logical spreadsheets. I apologize to the reader for using two logics instead of just one, but I felt it was for the best!

This thesis lays out the fundamentals of how logical spreadsheets operate. However, there is still work to be done before logical spreadsheets can be used broadly. In particular, inputting constraints needs to be made easy for people. This could be done either by doing user-centric research on how to make constraint-writing more accessible to non-experts, or by raising the level of expertise among people regarding writing logical expressions, or both. It is our hope that both of these efforts take place.

Appendix B

Existential Ω -Entailment Proofs

In this appendix we prove or disprove the properties given in Table 4.1.

We begin by substantiating our claim that existential Ω -entailment is paraconsistent.

Proposition B.1 (Explosiveness). *Existential Ω -entailment is not explosive.*

Proof We wish to show that $\{\psi, \neg\psi\} \vDash_{\Omega} \phi$ does not hold. Let $\Lambda = \{\phi, \neg\phi\}$. Say for the purpose of contradiction that $\Lambda \vDash_{\Omega} \phi \wedge \neg\phi$. Then some subset λ of Λ such that $\lambda \cup \Omega \not\vDash \perp$ entails $\phi \wedge \neg\phi$. However as $\{\phi \wedge \neg\phi\} \vDash \perp$ it must be that $\lambda \cup \Omega \vDash \perp$ as well, a contradiction. Thus $\{\phi, \neg\phi\} \not\vDash_{\Omega} \phi \wedge \neg\phi$, and existential Ω -entailment is not explosive. \square

The following proposition says that existential Ω -entailment proves no more consequences than logical entailment.

Proposition B.2 (Ω -Subclassicality). *Ω -Subclassicality holds for existential Ω -entailment.*

Proof We wish to show that $\Lambda \cup \Omega \vDash \phi$ if $\Lambda \vDash_{\Omega} \phi$. Say that $\Lambda \vDash_{\Omega} \phi$. Then there is a subset λ of Λ such that $\lambda \cup \Omega \vDash \phi$. By the monotonicity of \vDash , $\Lambda \vDash \phi$. \square

Since existential Ω -entailment is non-explosive, we know that when the theory Λ is inconsistent then if $\Lambda \vDash_{\Omega} \phi$ then $\Lambda \vDash \phi$. Also clearly the reverse does not hold.

What is less obvious is whether existential Ω -entailment coincides with logical entailment when the theory is consistent. The following proposition tells us that it does.

Proposition B.3 (Ω -Consistent classicality). *Ω -Consistent classicality holds for existential Ω -entailment.*

Proof We wish to show that *If $\Lambda \cup \Omega \not\models \perp$, $\Lambda \approx_{\Omega} \phi$ iff $\Lambda \cup \Omega \models \phi$.* Say that $\Lambda \cup \Omega \not\models \perp$. If also $\Lambda \approx_{\Omega} \phi$, then since Λ is Ω -consistent and $\Lambda \subseteq \Lambda \cup \Omega$, we have that $\Lambda \cup \Omega \models \phi$. On the other hand, if $\Lambda \cup \Omega \models \phi$, then since Λ is an Ω -consistent subset of $\Lambda \cup \Omega$, we have that $\Lambda \approx_{\Omega} \phi$. \square

Now that we have established the relationship of existential Ω -entailment with logical entailment, we turn to studying some abstract properties of the entailment relation. The first proposition asserts that existential Ω -entailment is monotonic.

Proposition B.4 (Monotonicity). *Existential Ω -entailment is monotonic.*

Proof We wish to show that *$\Lambda \cup \{\phi\} \approx_{\Omega} \psi$ if $\Lambda \approx_{\Omega} \psi$.* Say that $\Lambda \approx_{\Omega} \psi$. By the definition of \approx_{Ω} , there is a subset λ of Λ such that $\lambda \models \phi$. By the monotonicity of \models , $\lambda \cup \{\phi\} \models \phi$. As $\lambda \cup \{\phi\} \subseteq \Lambda \cup \{\phi\}$ and $\lambda \cup \{\phi\} \models \phi$, $\Lambda \cup \{\phi\} \approx_{\Omega} \psi$, again by the definition \approx_{Ω} . \square

However, existential Ω -entailment is not reflexive - sentences may not existentially Ω -entail themselves! The reason for this is that sentences that contradict the constraints will not be entailed.

Proposition B.5 (Reflexivity). *Existential Ω -entailment is irreflexive, even when restricted to sets of data.*

Proof We wish to show that *$\Lambda \approx_{\Omega} \phi$ if $\phi \in \Lambda$ does not hold.* Let $\Lambda = \{p\}$ and let $\Omega = \{\neg p\}$. Then $p \in \Lambda$ but $\Lambda \not\approx_{\Omega} p$. \square

The following proposition tells us that logically equivalent formulas have exactly the same consequences under existential Ω -entailment.

Proposition B.6 (Left logical equivalence). *Left logical equivalence holds for existential Ω -entailment.*

Proof We wish to show that $\Lambda \cup \{\phi\} \approx_{\Omega} \gamma$ if $\Lambda \cup \{\psi\} \approx_{\Omega} \gamma$ and $\models \phi \Leftrightarrow \psi$. Say that $\Lambda \cup \{\psi\} \approx_{\Omega} \gamma$ and $\models \phi \Leftrightarrow \psi$. By the definition of \approx_{Ω} , there is an Ω -consistent subset λ of $\Lambda \cup \{\psi\}$ such that $\lambda \cup \Omega \models \gamma$. If $\psi \notin \lambda$ then $\lambda \subseteq \Lambda \subseteq \Lambda \cup \{\phi\}$. Thus λ is an Ω -consistent subset of $\Lambda \cup \{\phi\}$ and therefore $\Lambda \cup \{\phi\} \approx_{\Omega} \gamma$. Otherwise let $\lambda' = \lambda - \{\psi\} \cup \phi$. Since ϕ and ψ are logically equivalent so are λ and λ' , and thus $\lambda \subseteq \Lambda \cup \{\phi\}$ and $\Lambda \cup \{\phi\} \approx_{\Omega} \gamma$. \square

The following proposition implies that one may replace logically equivalent formulas by one another on the right side of the \approx_{Ω} symbol.

Proposition B.7 (Right Weakening). *Right weakening holds for existential Ω -entailment.*

Proof We wish to show that $\Lambda \approx_{\Omega} \phi$ if $\Lambda \approx_{\Omega} \psi$ and $\approx_{\Omega} \psi \Rightarrow \phi$. Say that $\Lambda \approx_{\Omega} \psi$ and $\approx_{\Omega} \psi \Rightarrow \phi$. By the former, we see that there is an Ω -consistent subset λ of Λ such that $\lambda \cup \Omega \models \psi$. By the latter, we have that $\emptyset \cup \Omega \models \psi \Rightarrow \phi$, or simply $\Omega \models \psi \Rightarrow \phi$. By this and the monotonicity of \models we see that $\lambda \cup \Omega \models \psi \Rightarrow \phi$. But then $\lambda \cup \Omega \models \phi$, and so $\Lambda \approx_{\Omega} \phi$. \square

Proposition B.8 (Contraposition). *Contraposition fails for existential Ω -entailment, even when restricted to sets of data.*

Proof We wish to show that $\Lambda \cup \{\neg\psi\} \approx_{\Omega} \neg\phi$ if $\Lambda \cup \{\phi\} \approx_{\Omega} \psi$ does not hold. Let $\Lambda = \emptyset$, let $\phi = \{p\}$, let $\psi = \{\neg q\}$, and let $\Omega = \{q\}$. Then $\emptyset \cup \{p\} \approx_{\{\neg q\}} \{\neg q\}$ but $\emptyset \cup \{q\} \not\approx_{\{\neg q\}} \{\neg p\}$. \square

The conjunction of two existentially Ω -entailed sentences is not necessarily itself an existentially Ω -entailed sentence. The reason for this is that the two sentences may have been derived from two mutually exclusive subsets of the antecedents.

Proposition B.9 (And). *And fails for existential Ω -entailment, even when restricted to sets of data.*

Proof We wish to show that $\Lambda \approx_{\Omega} \phi \wedge \psi$ if $\Lambda \approx_{\Omega} \phi$ and $\Lambda \approx_{\Omega} \psi$ does not hold. Let $\Omega = \{p \Rightarrow \neg q, q \Rightarrow \neg p\}$. Let $\gamma = p$. Let $\Lambda = \{p, q\}$. Then $\{p, q\} \models_{\Omega} p$ and $\{p, q\} \models_{\Omega} \neg p$ but $\{p, q\} \not\models_{\Omega} p \wedge \neg p$. \square

Cut is yet another standard property that fails for existential Ω -entailment.

Proposition B.10 (Cut). *Cut fails for existential Ω -entailment, even when restricted to sets of data.*

Proof We wish to show that $\Lambda \approx_{\Omega} \psi$ if $\Lambda \approx_{\Omega} \phi$ and $\Lambda \cup \{\phi\} \models_{\Omega} \psi$ does not hold. Let $\Omega = \{r \Rightarrow \neg s, s \Rightarrow \neg r, r \Rightarrow p, \neg r \Rightarrow (p \Rightarrow q)\}$. Let $\phi = p$, let $\psi = q$, and let $\gamma = r$. Take $\Lambda = \{r, s\}$. Then $\lambda = \{r\}$ is an Ω -consistent subset of Λ that logically entails p , so $\Lambda \approx_{\Omega} \phi$. Also, $\lambda' = \{s, p\}$ is an Ω consistent subset of $\Lambda \cup \{\phi\}$ that logically entails q , so $\Lambda \cup \{\phi\} \models_{\Omega} \psi$. However, there is no consistent subset of Λ that entails q . Thus $\Lambda \not\models_{\Omega} \psi$ even though $\Lambda \approx_{\Omega} \phi$ and $\Lambda \cup \{\phi\} \models_{\Omega} \psi$. \square

The following proposition says that one can conclude an implication from an antecedent and a consequent.

Proposition B.11 (Conditionalization). *Conditionalization holds for existential Ω -entailment.*

Proof We wish to show that $\Lambda \approx_{\Omega} \phi \Rightarrow \psi$ if $\Lambda \cup \{\phi\} \models_{\Omega} \psi$. Say that $\Lambda \cup \{\phi\} \models_{\Omega} \psi$. Then there is an Ω -consistent subset λ of $\Lambda \cup \{\phi\}$ such that $\lambda \cup \Omega \models \psi$. Let $\lambda' = \lambda - \{\phi\}$. Since λ is a subset of Λ consistent with Ω , so is λ' . Since $\lambda' \cup \{\phi\} \models \psi$ we have that $\lambda' \models \phi \Rightarrow \psi$, and therefore $\Lambda \approx_{\Omega} \phi \Rightarrow \psi$. \square

However, the reverse does not hold: one cannot conclude that a antecedent concludes a consequent from a concluded implication.

Proposition B.12 (Deduction). *Deduction fails for existential Ω -entailment, even when restricted to sets of data.*

Proof We wish to show that $\Lambda \cup \{\phi\} \models_{\Omega} \psi$ if $\Lambda \approx_{\Omega} \phi \Rightarrow \psi$ does not hold. Let $\Omega = \emptyset$. Let $\phi = \neg p$, and let $\psi = q$. Let $\Lambda = \{p\}$. Since $\{p\} \models p \vee q$ and $p \vee q$ is equivalent to $\neg\phi \vee \psi$ is equivalent to $\phi \Rightarrow \psi$, we see that $\Lambda \models \phi \Rightarrow \psi$. Since Λ is consistent with

Ω , we have that $\Lambda \models_{\Omega} \phi \Rightarrow \psi$. However there is no consistent subset of $\Lambda \cup \{\phi\} = \{p, \neg p\}$ that logically entails q . \square

Finally, we note that iterated application of existential Ω -entailment does not give the same results.

Proposition B.13 (Idempotence). *Existential Ω -entailment is not idempotent, even when restricted to sets of data.*

Proof We wish to show that $C_{\approx_{\Omega}}(\Lambda) \neq C_{\approx_{\Omega}}(C_{\approx_{\Omega}}(\Lambda))$. Let $\Omega = \{\neg p \vee \neg q, p \Rightarrow p', q \Rightarrow q', p' \wedge q' \Rightarrow s\}$. Let $\Lambda = \{p, q\}$. Then $s \notin C_{\approx_{\Omega}}(\Lambda)$. However, since $p' \in C_{\approx_{\Omega}}(\Lambda)$ and $q' \in C_{\approx_{\Omega}}(\Lambda)$, we have that $s \in C_{\approx_{\Omega}}(C_{\approx_{\Omega}}(\Lambda))$. Thus $C_{\approx_{\Omega}}(\Lambda) \neq C_{\approx_{\Omega}}(C_{\approx_{\Omega}}(\Lambda))$. \square

Appendix C

Abstract State Machines

In this appendix, we first give a formal definition of Abstract State Machines (ASMs). Then we prove that Markov Change Logic is more expressive than Parallel ASMs without import constructors (which allow for the nondeterministic introduction of symbols into the language).

C.1 Formal definition of ASMs

Our definition is based upon Gurevich's definition in [27]. Our definition is broken up into sections. First we define static algebras, ASM updates, then Sequential ASMs, and finally Parallel ASMs.

C.1.1 Static Algebras

A *signature* of an ASM is a finite collection of function constants, each with a fixed arity. Some function names may be marked as *relation constants*, or *static constants*, or both. Every ASM signature contains the following static function constants: the binary function $=$, the nullary function constants **true**, **false**, and **undef**, and the Boolean operations \wedge , \vee , and \neg . **true** and **false** are relation constants.

A *static algebra* S , or *state* of a signature V is a nonempty set X called the *universe* of S , together with interpretations of the function constants in S on X . A

k -ary function constant is interpreted as a function from X^k to X . The interpretation of a k -ary relation constant is a function from X^k to $\{\mathbf{true}, \mathbf{false}\}$. V is known as the *signature* of X and is denoted $Sig(X)$. An X -state is a state with signature X .

The interpretations of \mathbf{true} , \mathbf{false} , and \mathbf{undef} are distinct elements of X . The Boolean operations behave in the usual way on \mathbf{true} and \mathbf{false} and produce \mathbf{undef} if one of the arguments is not Boolean. The equality sign is interpreted as the characteristic function of the identity relation on X . If $f(\bar{x})$ evaluates to \mathbf{true} in S , we say that $f(\bar{x})$ *holds* in S ; if $f(\bar{x})$ evaluates to \mathbf{false} in S , we say that $f(\bar{x})$ *fails* in S .

Terms are defined inductively. (1) A variable is a term. (2) If f is a k -ary function constant and t_1, \dots, t_k are terms, then $f(t_1, \dots, t_k)$ is a term. A term is *ground* if it does not contain variables. By analogy, other syntactical objects without variables are called ground. *Atomic boolean terms* are terms of the form $f(\bar{t})$, where f is a relation constant. *Boolean terms* are built from atomic Boolean terms by means of the Boolean operations.

A state S is *appropriate* for a syntactic object s if $Sig(S)$ includes the collection of function constants that occur in s . That collection is denoted $Sig(s)$.

In an appropriate state S , a ground term $t = f(t_1, \dots, t_k)$ evaluates to an element $Val_S(t) = f(Val_S(t_1), \dots, Val_S(t_k))$. If \bar{t} is a tuple $f(t_1, \dots, t_k)$ of terms, define $Val_S(\bar{t}) = (Val_S(t_1), \dots, Val_S(t_k))$.

C.1.2 ASM Updates

The *reduct* of a Y -state S to a smaller signature Y' is the Y' -state S' obtained from S by “disinterpreting” function constants in $Y - Y'$; S is an *expansion* of S' to Y .

A *carrier* is a state whose signature contains only static function constants. The *carrier* $|S|$ of a state S is the reduct of S to the static part of $Sig(S)$.

A *location over a carrier* C is a pair $l = (f, \bar{x})$, where f is a function constant outside of $Sig(C)$ and \bar{x} is a tuple of elements of C whose length equals the arity of f ; location l is *relational* if f is a relation constant. $Loc_Y(C)$ is the collection of all locations over C with function names in Y . A Y -state S with carrier C will sometimes be viewed as a function from $Loc_Y(C)$ to (the universe of) C ; *locations of*

S are locations in $Loc_Y(C)$.

If a state S is appropriate for a ground term $t_0 = f(\bar{t})$, then the *location* of t_0 in S is the location $(f, Val_S(\bar{t}))$.

An *update* of a state S is a pair $a = (l, y)$, where l is the location of S and y belongs to the universe of $|S|$; if l is relational then y is Boolean. The location l is the *location* $Loc(a)$ of a , and y is the *value* $Val(a)$ of a . The result of *firing* a at S is a new state S' such that $Sig(S') = Sig(S)$, $|S'| = |S|$, $S'(l) = y$, and $S'(l') = S(l')$ for every location l' of S different than l .

An *update set* b over a state S is a set of updates of S . $Loc(b) = \{Loc(a) | a \in b\}$. For each $l \in b$, $Val_b(l) = \{Val(a) | a \in b \wedge Loc(a) = l\}$. An update set is *consistent* at state S if every $Val_b(l)$ is a singleton set; otherwise b is inconsistent.

The result of *firing* a consistent update set b at state S is a new state S' such that $Sig(S') = Sig(S)$, $|S'| = |S|$, and if $l \in Loc(b)$ then $S'(l)$ is the only element of $Val_b(l)$; otherwise $S'(l) = S(l)$. The result of firing an inconsistent update set b at state S is a “new” state $S' = S$.

C.1.3 Sequential ASMs

In this subsection, all terms are ground.

An *update instruction* R is an expression $f(\bar{t}) := t_0$ where f is a non-static function name (the *subject* of the instruction), \bar{t} is a tuple of terms whose length equals the arity of f , and t_0 is another term; if f is a relation constant then t_0 must be a Boolean term.

To execute R at an appropriate state S , fire the update $a = (l, y)$ at S , where $l = (f, Val_S(\bar{t}))$ and $y = Val_S(t_0)$. Define $Updates(R, S) = \{a\}$.

Basic rules are constructed recursively from update instructions by means of two rule constructors: the sequence constructor and the conditional constructor. Semantics is defined by the means of update sets. For each rule R and every state S appropriate for R , we define an update set $Updates(R, S)$ over S . To fire R at S , fire $Updates(R, S)$.

Sequence Constructor. A sequence of rules $[R_1, \dots, R_n]$ is a rule.

If R is a sequence of rules $[R_1, \dots, R_n]$ then $Updates(R, S) = Updates(R_1, S) \cup \dots \cup Updates(R_n, S)$.

Conditional Constructor. If g is a Boolean term and R_0 is a rule, then **if g then R_0 endif** is a rule.

If R is a conditional rule **if g then R endif** then $Updates(R, S) = Updates(R_0, S)$ if g holds in S , otherwise $Updates(R, S) = \emptyset$.

Sequential ASMs have another constructor called the *import constructor* which can be used to create non-basic rules. However the details of this constructor are unimportant for our purposes and we do not define them here.

A *program* P is a rule without free variables. A *basic program* is a basic rule without free variables. To fire P at an appropriate state S , fire $Updates(P, S)$ at S .

A *pure run* of P is a sequence $\langle S_n \mid n < \kappa \rangle$ of states of signature $Sig(P)$ such that each S_{n+1} is obtained from S_n by firing P at S_n . Here and henceforth κ is a positive integer or the first infinite ordinal. In the latter case, $\{n \mid n < \kappa\}$ is the set of all natural numbers.

In general runs may be affected by the environment. The environment manifests itself via some basic functions e_1, \dots, e_k called *external functions*. Call non-external basic functions *internal*. If S is an appropriate state for a program P , let S^- be the reduct of S to the internal vocabulary.

A *run* of a program P is a sequence $\langle S_n \mid n < \kappa \rangle$ of states where (1) every nonfinal S_n is an appropriate state for P and the final state (if any) is a state of the internal vocabulary of P , and (2) every S_{n+1}^- is obtained from S_n by firing P at S_n .

C.1.4 Parallel ASMs

An *atomic variable declaration* is an expression v **ranges over** U , where v is a variable name and U is a unary relation constant. U is the *range* of v . A variable declaration E is a sequence of atomic variable declarations, and $Var(E)$ is the collection of variables in E . A variable is *free* if it is undefined, otherwise it is *bound*. We denote the free variables of a syntactic object s by $Free(s)$ and denote the bound variables of s by $Bound(s)$. A variable declaration E *covers* a syntactic object s if

$Var(E)$ contains $Free(s)$.

An *auxiliary signature* has the form $Y \cup V$, where Y is a signature and V is a finite set of variables, and each $v \in V$ is treated as a nullary function, except that it cannot be the subject of an update instruction. We say that a state S of an auxiliary vocabulary is *appropriate* for a syntactical object s if all function names and all free variables of s occur in $Fun(S)$.

A *first order guard* is defined as follows: (1) if f is an n -ary relation name and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a guard. (2) Any Boolean combination of guards is guard. (3) If g is a guard and U is a unary relation constant, then $(\exists v \in U)g$ and $(\forall v \in U)g$ are guards. First order guards are given semantics that mirrors the definition of truth in first-order logic.

In a Parallel ASM, terms are defined as for Sequential ASMs, except that terms may contain free variables, and guards are first-order. In addition, parallel ASMs have the following rule constructor:

Declaration Constructor. An atomic variable declaration followed by a rule is a rule.

Say that E is a variable declaration, R is a rule, and S is a state of an auxiliary vocabulary. R is (D, S) -*perspicuous* if it satisfies the following conditions: (1) no variable is declared more than once in R , and $Bound(R)$ is disjoint from $Free(R) \cup Var(E) \cup Fun(S)$.

A *program* is a rule without any undeclared variables.

By induction on R , we define the update set $B = Updates(E, R, S)$ generated by a rule R at an appropriate state S under a declaration E that covers R . To fire R at S under D , fire B .

An arbitrary rule R is equivalent, for a given E and S , to a (E, S) -perspicuous rule R' obtained by renaming bound variables. The equivalence means that $Updates(E, R, S) = Updates(E, R', S)$. Thus, it remains to define $B = Updates(E, R, S)$ in the case when R is (E, S) -perspicuous.

If E is not empty, then B is the union of $Updates(\emptyset, R, S')$, where S' ranges over expansions of S such that $Fun(S') = Fun(S) \cup Var(D)$ and S' is consistent with E (so that the values of E variables are within their ranges in S').

Suppose $E = \emptyset$. If R is an update instruction then $B = \text{Updates}(R, S)$. If R is a sequence of rules $[R_1, \dots, R_k]$, then B is the union of update sets $\text{Updates}(\emptyset, R_i, S)$. Suppose that R is the conditional rule **if** g **then** R_0 **endif**. Since R is covered by the empty declaration, g has no free variables. If g holds, then $B = \text{Updates}(\emptyset, R_0, S)$, otherwise $B = \emptyset$. Finally, if R is a declaration rule with declaration d and body R' then $B = \text{Updates}(d, R', S)$.

C.2 Proofs

In this section, we give a series of proofs, culminating in a proof that Markov Change Logic is more expressive than Parallel ASMs without import constructors (which allow for the nondeterministic introduction of symbols into the language). We begin by defining a normal form for ASM rules.

We say that a rule is in *ASM normal form* if it is a sequence of rules $[R_1, \dots, R_n]$, where each R_i is of one of the following three forms:

- (1) B
- (2) **if** G **then** B
- (3) V **if** G **then** B

where V is a variable declaration, G is a first-order guard that is a conjunction of literals, and B is a basic rule.

Lemma C.1. *Every Parallel ASM program without import constructors P is equivalent to a Parallel ASM program P in ASM normal form.*

Proof. We give a constructive proof, via an algorithm:

Let P be a Parallel ASM program without import constructors. Follow the following steps to convert P to normal form.

- (1) *Rename variables.* Rename with variable names of P with globally unique names.

- (2) *Wrap in a sequence constructor.* If the outermost constructor of P is not a sequence constructor, then wrap P with a sequence constructor.
- (3) *Remove inner sequence constructors.* While P is of the form $[R_1, \dots, R_n]$, where some R_i consists of a prefix C followed by a sequence $[S_1, \dots, S_m]$: let $P = [R_1, \dots, R_{i-1}, CS_1, \dots, CS_m, R_{i+1}, R_n]$.
- (4) *Move inner variable declarations to the top level.* Where P is of the form $[R_1, \dots, R_n]$, for each R_i , move all variable declarations to the front of R_i .
- (5) *Merge conditional constructors.* P is now of the form $[R_1, \dots, R_n]$, where each R_i consists of a chain of conditional constructors. While there is a rule R_i of the form **if** C **then** **if** C_2 **then** R , replace R_i with the rule **if** $\text{and}(C, C_2)$ **then** R .
- (6) *Convert guards to disjunctive normal form.* If applicable, convert the guard of each rule R_1, \dots, R_n to disjunctive normal form.
- (7) *Break apart rules with DNF guards into multiple rules with conjunctive guards.* For each rule R_i that has a guard G that is a disjunction $g_1 \vee g_k$, convert P into the rule $[R_1, \dots, R_{i-1}, R_{i+1}, \dots, R_n, R_i(g_1), \dots, R_i(g_k)]$, where $R_i(g_j)$ is just like R_i except the guard is g_j .

P is now necessarily in ASM normal form. Since each step in the transformation transformed P into an equivalent rule, P is equivalent to its original form. \square

As we prove below, every Parallel ASM program without import constructors is equivalent to a MCL program. We give some examples.

(Example 1) The basic rule $r(a, b) : - \text{true}$, where r is a relation. This is equivalent to the MCL rule $r^{++}(a, b)$.

(Example 2) The basic rule $r(a, b) :- false$, where r is a relation. This is equivalent to the MCL rule $r^{--}(a, b)$.

(Example 3) The basic rule $f(a, b) :- c$. This is equivalent to the MCL rules:

$$\{r^{++}(a, b, c), \\ r^{--}(a, b, X) :- r(a, b, X) \ \& \ x \neq c\}.$$

(Example 4) The basic rule **if** $g(a)$ **then** $f(a, b) :- c$, where g is an input relation. This is equivalent to the MCL rules:

$$\{f^{++}(a, b, c) :- g^+(a), \\ f^{--}(a, b, X) :- g^+(a) \ \& \ f(a, b, X) \ \& \ x \neq c\}.$$

(Example 5) The basic rule **if** $g(a)$ **then** $f(a, b) :- c$, where g is a not input relation. This is equivalent to the MCL rules:

$$\{f^{++}(a, b, c) :- g(a), \\ f^{--}(a, b, X) :- g(a) \ \& \ f(a, b, X) \ \& \ x \neq c\}.$$

Theorem C.2. *MCL is at least as expressive as Parallel ASMs without import constructors.*

Proof. Let P be a Parallel ASM program without import constructors with universe X and signature V .

We will construct a corresponding MCL program P' .

For each k -ary function f of P , let f' be a $k + 1$ -ary relation constant.

Define f^* to be f^+ if f is external and f otherwise.

If X is a variable and t is an entity or a variable, define $unroll(X, t)$ to be (1) $X = t$.

If X is a variable and $f(t_1, \dots, t_k)$ is a functional term, define $unroll(X, f(t_1, \dots, t_k))$ to be $f^*(Y_1, \dots, Y_k) \ \& \ unroll(Y_1, t_1) \ \& \ unroll(Y_k, t_k)$, where $Y_1 \dots Y_k$ are new variables.

If $\bar{X} = \langle X_1, \dots, X_n \rangle$ is an n -tuple of variables and $\bar{t} = \langle t_1, \dots, t_n \rangle$ is an n -tuple of terms, then $unroll(\bar{X}, \bar{t})$ is $unroll(X_1, t_1) \ \& \ \dots \ \& \ unroll(X_n, t_n)$.

Let $P = [R_1, \dots, R_n]$ be a program in ASM normal form. We assume that each R_i is of the form $V \ \text{if } G \ \text{then } B$; the other rule forms are simpler and can be handled using a similar technique.

Each R_i corresponds to two MCL rules: a positive rule and a negative rule. We first describe how to produce the corresponding positive rule.

Now, R_i is of the form:

$$\begin{array}{l} V_1 \text{ ranges over } s_1, \dots, V_k \text{ ranges over } s_k \\ \text{if } p_1(\bar{t}_1) \wedge \dots \wedge p_m(\bar{t}_m) \wedge \neg q_1(\bar{w}_1) \wedge \dots \wedge \neg q_l(\bar{w}_l) \\ \text{then } f(u_1, \dots, u_n) :- u \end{array}$$

Then, $pos(R_i)$ is:

$$\begin{array}{l} f^{++}(X_1, \dots, X_n, W) :- \\ s_1(V_1) \ \& \ \dots \ \& \ s_k(V_k) \\ \ \& \ p_1^*(Y_1) \ \& \ unroll(\bar{Y}_1, \bar{t}_1) \ \& \ \dots \ \& \ p_m^*(Y_m) \ \& \ unroll(\bar{Y}_m, \bar{t}_m) \\ \ \& \ \neg q_1^*(\bar{Z}_1) \ \& \ unroll(\bar{Z}_1, \bar{w}_1) \ \& \ \dots \ \& \ \neg q_l^*(\bar{Z}_l) \ \& \ unroll(\bar{Z}_l, \bar{w}_l) \\ \ \& \ unroll(X_1, u_1) \ \& \ \dots \ \& \ unroll(X_k, u_k) \end{array}$$

$\& \text{unroll}(W, u)$

Where $W, X_1, \dots, X_k, Y_1, \dots, Y_m, Z_1, \dots, Z_l$ are new variables.

And $\text{neg}(R_i)$ is:

$$\begin{aligned}
 f^{--}(X_1, \dots, X_n, V) :- \\
 & f(X_1, \dots, X_n, V) \\
 & \& s_1(V_1) \& \dots \& s_k(V_k) \\
 & \& p_1^*(Y_1) \& \text{unroll}(\bar{Y}_1, \bar{t}_1) \& \dots \& p_m^*(\bar{Y}_m) \& \text{unroll}(\bar{Y}_m, \bar{t}_m) \\
 & \& \neg q_1^*(\bar{Z}_1) \& \text{unroll}(\bar{Z}_1, \bar{w}_1) \& \dots \& \neg q_l^*(\bar{Z}_l) \& \text{unroll}(\bar{Z}_l, \bar{w}_l) \\
 & \& \text{unroll}(X_1, u_1) \& \dots \& \text{unroll}(X_k, u_k) \\
 & \& \text{unroll}(W, u) \quad \& V \neq W
 \end{aligned}$$

Where $V, W, X_1, \dots, X_k, Y_1, \dots, Y_m, Z_1, \dots, Z_l$ are new variables.

Corollary C.3. *MCL is more expressive than Parallel ASMs without import constructors.*

Proof. Follows from Theorem C.2 and the fact that all positive and negative rules generated by Theorem C.2 are non-recursive. On the other hand, MCL rules can be recursive. Since recursive datalog is more expressive than non-recursive datalog [1], the theorem follows. \square

Bibliography

- [1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] Serge Abiteboul and Victor Vianu. Procedural and declarative database update languages. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '88, pages 240–250, New York, NY, USA, 1988. ACM.
- [3] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.
- [4] James Bailey, Guozhu Dong, and Kotagiri Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 264–273, New York, NY, USA, 1998. ACM.
- [5] Salem Benferhat, Didier Dubois, and Henri Prade. Argumentative inference in uncertain and inconsistent knowledge bases. In *In Proceedings of Uncertainty in Artificial Intelligence*, pages 411–419. Morgan Kaufmann, 1993.
- [6] Philippe Besnard and Anthony Hunter. Quasi-classical logic: Non-trivializable classical reasoning from inconsistent information. In *Proceedings of the European*

- Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 44–51, London, UK, 1995. Springer-Verlag.
- [7] Anthony Bonner, Michael Kifer, and Mariano Consens. Database programming in transaction logic. In *Workshop on Database Programming Languages*, pages 309–337, 1993.
- [8] George S. Boolos and Richard C. Jeffrey. *Computability and logic*. Cambridge University Press, New York, NY, USA, 1989.
- [9] Claus Brabrand, Anders Mller, Mikkel Ricky, and Michael I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web Journal*, 3, 2000.
- [10] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases, VLDB '91*, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
- [11] Iliano Cervesato. NEXCEL, a deductive spreadsheet. *Knowl. Eng. Rev.*, 22(3):221–236, 2007.
- [12] H. Chalupsky, R. MacGregor, and T. Russ. *PowerLoom Manual*. USC Information Sciences Institute, 2006.
- [13] Weidong Chen. Programming with logical queries, bulk updates, and hypothetical reasoning. *IEEE Trans. on Knowl. and Data Eng.*, 9:587–599, July 1997.
- [14] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, 1993.
- [15] W. F. Clocksin and Chris Mellish. *Programming in Prolog, 3rd Edition*. Springer, 1987.

- [16] Sara Comai and Letizia Tanca. Termination and confluence by rule prioritization. *IEEE Trans. on Knowl. and Data Eng.*, 15(2):257–270, 2003.
- [17] Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.
- [18] Vasiliki Efstathioua and Anthony Hunter. Algorithms for generating arguments and counterarguments in propositional logic. *International Journal of Approximate Reasoning*, 2011.
- [19] Morten Elvang-Gøransson and Anthony Hunter. Argumentative logics: Reasoning with classically inconsistent information. *Data Knowledge Engineering*, 16(2):125–145, 1995.
- [20] Herbert Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2000.
- [21] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, Christian Russ, and Markus Zanker. Developing constraint-based applications with spreadsheets. In *IEA/AIE*, pages 197–207, 2003.
- [22] Alejandro J. García and Guillermo R. Simari. Defeasible logic programming: an argumentative approach. *Theory Pract. Log. Program.*, 4:95–138, January 2004.
- [23] Michael R. Genesereth. CS157: Computational logic lecture notes. <http://logic.stanford.edu/classes/cs157/2010/cs157.html>, 2010.
- [24] Michael R. Genesereth. *Data Integration: A Relational Logic Approach*. Morgan & Claypool, 2010.
- [25] Michael R. Genesereth and Nils Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, 1987.
- [26] Gopal Gupta and Shameem F. Akhter. Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In *Practical Aspects of Declarative Languages (PADL)*, volume 1753 LNCS, pages 308–323. Springer, 2000.

- [27] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993.
- [28] Michael Hanus. Putting declarative programming into the web: translating curry to javascript. In *PPDP '07*, pages 155–166, New York, NY, USA, 2007. ACM.
- [29] Timothy L. Hinrichs. *Extensional Reasoning*. Stanford University, 2007.
- [30] Timothy L. Hinrichs, Jui-Yi Kao, and Michael R. Genesereth. Inconsistency-tolerant reasoning with classical logic and large databases. In *SARA*, 2009.
- [31] Shan hwei Nienhuys-cheng and Ronald de Wolf. The subsumption theorem in inductive logic programming: Facts and fallacies. In *Advances in Inductive Logic Programming. IOS*, pages 265–276. Press, 1995.
- [32] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
- [33] Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20(Supplement 1):503 – 581, 1994.
- [34] Michael Kassoff and Andre Valente. An introduction to logical spreadsheets. *Knowl. Eng. Rev.*, 22(3):213–219, 2007.
- [35] Michael Kassoff, Lee-Ming Zen, Ankit Garg, and Michael Genesereth. PrediCalc: A logical spreadsheet management system. In *VLDB*, 2005.
- [36] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artif. Intell.*, 44:167–207, July 1990.
- [37] Frank Kriwaczek. LogiCalc: a prolog spreadsheet. *Machine intelligence 11*, pages 193–208, 1988.
- [38] Mengchi Liu. Extending datalog with declarative updates. *J. Intell. Inf. Syst.*, 20:107–129, March 2003.

- [39] Sanjay Manchanda and David Scott Warren. A logic-based language for database updates. In *Foundations of Deductive Databases and Logic Programming.*, pages 363–394. Morgan Kaufmann, 1988.
- [40] Ruth Manor and Nicholas Rescher. On inferences from inconsistent information. *Theory and Decision*, 1(1):179–219, 1970.
- [41] Shamim Naqvi and Ravi Krishnamurthy. Database updates in logic programming. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '88, pages 251–262, New York, NY, USA, 1988. ACM.
- [42] Levent Orman. Differential relational calculus for integrity maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):328–341, 1998.
- [43] Jonathan Pool. Can controlled languages scale to the web? In *5th International Workshop on Controlled Language Applications*, 2006.
- [44] Graham Priest and Koji Tanaka. Paraconsistent logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2009 edition, 2009.
- [45] Pearl Pu and Boi Faltings. Effective interaction principles for user-involved constraint problem solving. In *workshop notes, Second International Workshop on User-Interaction in Constraint Satisfaction, the Eighth International Conference on Principles and Practice of Constraint Programming*, September 2002.
- [46] C.R. Ramakrishnan, I.V. Ramakrishnan, and David S. Warren. XcelLog: a deductive spreadsheet system. *Knowl. Eng. Rev.*, 22(3):269–279, 2007.
- [47] Frank P. Ramsey. On a problem in formal logic. *Proc. London Math. Soc.*, 30:264286, 1930.
- [48] Stewart. J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

- [49] Christopher Scaffidi. Estimating the numbers of end users and end user programmers. In *In IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 207–214, 2005.
- [50] John F. Sowa and Alexander Borgida. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Academic Press, 1991.
- [51] Michael Spenke and Christian Beilken. A spreadsheet interface for logic programming. In *CHI '89*, pages 75–80, New York, NY, USA, 1989. ACM.
- [52] Marcelo Tallis, Rand Waltzman, and Robert Balzer. An introduction to logical spreadsheets. *Knowl. Eng. Rev.*, 22(3):255–268, 2007.
- [53] Andre Valente, David Van Brackle, Hans Chalupsky, and Gary Edwards. Adding deductive logic to a COTS spreadsheet. *Knowl. Eng. Rev.*, 22(3):237–253, 2007.
- [54] Maarten H. van Emden, Masaru Ohki, and Akikazu Takeuchi. Spreadsheets with incremental queries as a user interface for logic programming. *New Gen. Comput.*, 4(3):287–304, 1986.
- [55] W3C. Document object model (DOM). <http://www.w3.org/DOM/>, 2005.
- [56] Luke Wroblewski. Inline validation in web forms. <http://www.alistapart.com/articles/inline-validation-in-web-forms/>, September 2009.
- [57] Moshé M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the 1st International Conference on Very Large Data Bases, VLDB '75*, pages 1–24, New York, NY, USA, 1975. ACM.