Transaction Logic Programming

(or, A Logic of Procedural and Declarative Knowledge)

Anthony J. Bonner
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4, Canada
bonner@db.toronto.edu

Michael Kifer
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11790, U.S.A.
kifer@cs.sunysb.edu

Abstract

A new formalism, called *Transaction Logic*, is proposed to account in a clean and declarative fashion for the phenomenon of state changes in logic programs and databases. Transaction Logic extends classical logic with connectives specifically designed for combining simple actions into complex transactions and for providing control over their execution. The proposed logic has a natural model theory and a sound and complete proof theory, but unlike many other logics, it allows users to *program* transactions. This is possible because, like classical logic, Transaction Logic has a "Horn" version which has a procedural as well as a declarative semantics. In addition, the semantics leads naturally to features whose amalgamation in a single logic has proved elusive in the past. These features include both hypothetical *and* committed updates, dynamic constraints on transaction execution, nondeterminism, and bulk updates. Finally, Transaction Logic holds promise as a logical model of hitherto non-logical phenomena, including so-called *procedural knowledge* in AI, *active databases*, and the *behavior* of object-oriented databases, especially methods with side effects. Apart from the applications to Databases and Logic Programming, we also discuss applications to a number of AI problems, such as planning, temporal specifications, and the frame problem.

Technical Report CSRI-323
November 1995

(a substantial revision of Report CSRI-270 of April 1992)

Computer Systems Research Institute
University of Toronto

Related publications, a prototype implementation, and a tutorial are available at the Transaction Logic web page:
`http://www.db.toronto.edu:8020/transaction-logic.html`

Contents

1 **Introduction** 1

2 **Overview and Introductory Examples** 6
- 2.1 Elementary Updates and Sequential Composition . . . 7
- 2.2 Tests and Conditions . . . 10
- 2.3 Non-Deterministic Transactions . . . 11
- 2.4 Rules . . . 12
- 2.5 Transaction Bases . . . 14
- 2.6 Constraints . . . 16

3 **Syntax** 18

4 **Elementary Operations** 19
- 4.1 State Data Oracles . . . 20
- 4.2 State Transition Oracles . . . 20
- 4.3 Examples . . . 21
- 4.4 Implementing Oracles . . . 23

5 **Model Theory** 23
- 5.1 Path Structures . . . 25
- 5.2 Satisfaction on Paths . . . 26
- 5.3 Models . . . 30
- 5.4 Execution as Entailment . . . 31
- 5.5 Examples . . . 32
- 5.6 Transaction Answers . . . 34
- 5.7 Discussion . . . 36

6 **Proof Theory** 38
- 6.1 Serial Horn Programs . . . 38
- 6.2 Two Inference Systems . . . 40
- 6.3 Inference System $\Im^I$ . . . 41
  - 6.3.1 Execution as Deduction . . . 44
  - 6.3.2 Executing Transactions . . . 46
  - 6.3.3 Reverse Execution . . . 47
  - 6.3.4 Example: Non-Deterministic Updates . . . 48
  - 6.3.5 Example: Inference with Unification . . . 49
  - 6.3.6 The Role of the Oracles—Observations . . . 50
- 6.4 Inference System $\Im^{II}$ . . . 51

7 Applications **55**
7.1 Consistency Maintenance . . . 55
7.2 View Updates . . . 56
7.3 Heterogeneous Databases . . . 57
7.4 Simulating Systems with State . . . 59
7.5 Bulk Updates . . . 64
7.6 Non-Deterministic Sampling . . . 66
7.7 Dynamic Constraints on Transaction Execution . . . 68
7.7.1 Constraints Based on Serial Conjunction . . . 69
7.7.2 Constraints Based on Serial Implication . . . 71
7.8 The Frame Problem . . . 73
7.9 Planning . . . 80
7.9.1 Naive Planning . . . 81
7.9.2 Script-Based Planning . . . 83
7.9.3 STRIPS . . . 87
7.9.4 Improved STRIPS . . . 89

8 Hypothetical Reasoning **92**
8.1 Hypothetical Formulas . . . 92
8.2 Retrospection . . . 94
8.3 Proof Theory for Hypotheticals and Retrospectives . . . 94
8.3.1 $\diamond$-Serial-Horn Programs . . . 94
8.3.2 Inference System $\Im^{\diamond}$ . . . 95
8.3.3 Example: Hypotheticals . . . 97
8.3.4 Example: Retrospectives . . . 98
8.3.5 Execution as Deduction in $\Im^{\diamond}$ . . . 99
8.3.6 Normal and Reverse Execution in $\Im^{\diamond}$ . . . 102
8.4 Applications of Hypotheticals . . . 103
8.4.1 Subjunctive Queries and Counterfactuals . . . 104
8.4.2 Imperative Programming Constructs . . . 104
8.4.3 Software Verification . . . 107

9 Perfect-Model Semantics for Negation as Failure **107**

10 Updating General Deductive Databases **110**

11 Comparison with Other Work **112**
11.1 Declarative Languages for Database Updates . . . 112
11.2 Logics of Action . . . 115

A Appendix: Soundness of $\Im^{I}$ **126**

B Appendix: Completeness of $\Im^I$ **128**
B.1 Ground Inference 128
B.2 Completeness of Ground $\Im^I$ 130
B.3 Completeness of $\Im^I$ 133

C Appendix: Executional Deduction in $\Im^I$ **135**

D Appendix: Soundness of $\Im^\diamond$ **138**

E Appendix: Completeness of $\Im^\diamond$ **141**
E.1 Ground Inference 141
E.2 Completeness of Ground $\Im^\diamond$ 143
E.3 Completeness of $\Im^\diamond$ 146

F Appendix: Executional Deduction in $\Im^\diamond$ **148**

1 Introduction

We introduce a novel logic, called Transaction Logic (abbreviated $\mathcal{TR}$), that accounts in a clean, declarative fashion for the phenomenon of updating arbitrary logical theories, most notably, databases and logic programs. Unlike most logics of action, $\mathcal{TR}$ is a declarative formalism for specifying and executing procedures (if the user will permit the use of that oxymoron), procedures that update and permanently change a database, a logic program or, more generally, a logical theory. As a special case, transactions can be defined as logic programs. This is possible because, like classical logic, $\mathcal{TR}$ has a "Horn" version that has *both* a procedural and a declarative semantics, as well as an efficient SLD-style proof procedure. This paper presents the model theory and proof theory of $\mathcal{TR}$, and develops many of its applications.

Overview

$\mathcal{TR}$ was designed with several application in mind, especially in databases, logic programming, and AI. It was therefore developed as a general logic, so that it could solve a wide range of update-related problems. Individual applications were then carved out of different fragments of the logic. These applications, both practical and theoretical, are discussed throughout the paper, especially in Section 7, where many are developed in detail. We outline several of them here.

1. $\mathcal{TR}$ provides a logical account for many update-related phenomena. For instance, in logic programming, $\mathcal{TR}$ provides a logical treatment of the assert and retract operators in Prolog. This treatment effectively extends the theory of logic programming to include updates as well as queries. In object-oriented databases, $\mathcal{TR}$ can be combined with object-oriented logics, such as F-logic [56], to provide a logical account of *methods*—procedures hidden inside objects that manipulate these objects' internal states. Thus, while F-logic covers the structural aspect of object-oriented databases, its combination with $\mathcal{TR}$ would account for the behavioral aspect as well. Applications of $\mathcal{TR}$ to so called *active databases* are described in [22]. In AI, $\mathcal{TR}$ suggests a logical account of planning and design. STRIPS-like actions,[1] for instance, as well as many aspects of hierarchical and non-linear planning are easily expressed in $\mathcal{TR}$. Although there have been previous attempts to give these phenomena declarative semantics, until now there has been no unifying *logical* framework that accounts for them all.

2. Since $\mathcal{TR}$ is a full-fledged logic, it is more flexible and expressive than procedural systems in specifying transactions. Like procedural languages, $\mathcal{TR}$ is a language for combining simple actions into complex ones; but in $\mathcal{TR}$, actions can be combined in a greater variety of ways. In procedural languages, sequential composition is the only combinator, whereas in $\mathcal{TR}$, each logical operator combines actions in its own way. The result is that in $\mathcal{TR}$, one can specify transactions at many levels of detail, from the procedural to the declarative. At one extreme, the user may spell out an exact sequence of operations in excruciating detail. At the other extreme, he may specify loose constraints that the transaction must satisfy. In general, sequences and constraints can be arbitrarily mixed, and in this way, procedural and declarative knowledge are seamlessly integrated.

3. Because of its generality, $\mathcal{TR}$ supports a wide range of functionality in several areas. This functionality includes database queries and views; unification and rule-base inference; transac-

[1] STRIPS was an early AI planning system that simulated the actions of a robot arm [34].

tion and subroutine definition; deterministic and non-deterministic actions; static and dynamic constraints; hypothetical and retrospective transactions; and a wide class of tests and conditions on actions, including pre-conditions, post-conditions, and intermediate conditions. Furthermore, many problems related to the updating of incomplete information can be dealt with efficiently. Perhaps most important, these diverse capabilities are *not* built into $\mathcal{TR}$ separately. Instead, they all derive from a small number of basic ideas embodied in a single logic.

4. Commercial database systems are poor at specifying transactions. For example, in standard SQL, one can only define relatively simple updates, and even then, one must often abandon relational algebra and resort to relatively awkward subqueries. More seriously, one cannot define transaction procedures in SQL. The main reason for this is that there is no way to combine simple transactions into more complex ones. This limitation should be contrasted with SQL's elegant view mechanism, in which views can easily be combined to produce other views. The result is that to define transactions, a user must go outside of SQL and embed it within a procedural language, such as Cobol. A problem with this approach is that embedded SQL is an order of magnitude more difficult to use and optimize than pure SQL. It should not be surprising that difficult problems arise in trying to define database transactions, since relational databases are founded on relational algebra (or, equivalently, first-order logic), which is a language for expressing *queries*, not transactions. $\mathcal{TR}$ shows a way out of this quandary, as it provides foundation for queries *and* transactions alike.

5. For a wide class of problems, the so-called *frame problem* is not an issue for $\mathcal{TR}$. The frame problem arises because, to reason about updates, one must specify what does *not* change, as well as what does. Many ingenious solutions have been proposed to deal with the frame problem, but most of them have addressed the very general case of arbitrary reasoning about updates to arbitrary logical theories. As such, the proposed solutions have been conceptually complex and computationally intractable, even for simple updates. Fortunately, many interesting and important problems do not require a completely general solution to the frame problem. $\mathcal{TR}$, for example, was designed so that the frame problem is not an issue when actions are *executed* or *planned*. This is possible for two reasons. First, transactions in $\mathcal{TR}$ are specified in terms of a small set of basic updates. Once the frame problem is solved for these updates, the solution propagates automatically to complex actions. Thus, $\mathcal{TR}$ programmers do not experience the frame problem, just as conventional programmers do not. In addition, the proof theory for $\mathcal{TR}$ actually updates the database during inference, just as procedural systems update the database during execution. This contrasts sharply with other logical formalisms, like the situation calculus, which reason about updates, but never actually change the database. In this way, $\mathcal{TR}$ provides simple, efficient handling of the frame problem, for both the specification and execution of transactions.

Detailed development of a number of applications appears in Section 7.

Other Logics

On the surface, there would appear to be many other candidates for a logic of transactions, since many logics reason about updates or about the related phenomena of time and action. However, despite a plethora of action logics, researchers continue to complain that there is no clear declarative semantics for updates either in databases or in logic programming [11, 9, 3, 80]. In fact, no action logic has ever been adopted by a database or logic programming system, and none has become a core of database or logic-programming theory. This is in stark contrast to classical logic, which is the foundation of database queries and logic programming, both in theory and in practice.

There appear to be a few simple reasons for this unsuitability of existing action logics.

One major problem is that most logics of time or action are *hypothetical*, that is, they do not permanently change the database. Instead, these logics reason about what *would* happen *if* certain sequences of actions took place. For instance, they might infer that *if* the pawn took the knight, then the rook *would* be threatened. Such systems are useful for reasoning about alternatives, but they do not actually *accomplish* state transitions. Hence, these logics do not provide executable specifications. In contrast, $\mathcal{TR}$ is a logic that supports *both* real and hypothetical updates. Procedures in $\mathcal{TR}$ may commit their updates (and thus permanently change the database), reason about these updates without committing them, or they may do any combination thereof.

Another major problem is that most logics of time or action were not designed for programming. Instead, they were intended for specifying *properties* of programs and for reasoning about them. In such systems, one might say that action A precedes action B, and that B precedes C, from which the system would infer that A precedes C. Thus, these formalisms are useful for *reasoning about* actions, but are not that useful for *specifying* what the actions A, B, and C actually do, and even less so for executing them.

A related problem is that many logics of action cannot assign names to composite transactions. In their intended context (of reasoning about sequences of actions), this is not a shortcoming. However, this renders such logics inappropriate for programming transactions, since specifying transactions without a naming facility is like programming without subroutines. From a programming standpoint, the lack of a straightforward naming facility defeats the purpose of using logic in the first place, which is to free the user from the drudgery of low-level details.

Another problem is that many logics insist on strict separation between non-updating queries and actions with side effects. However, this distinction is blurred in object-oriented systems, where both queries and updates are special cases of a single idea: method invocation (or message passing). In such systems, an update can be thought of as a query with side effects. Every method is simply a *program* that operates on the data. In fact, the ODMG-93 object database standard explicitly states that it does not distinguish a subcategory of method that is side-effect free [25, Section 2.5]. $\mathcal{TR}$ models this uniformity naturally, treating all methods equally, thereby providing a logical foundation for object-oriented databases.

The important point here is that, unlike most formalisms of action, $\mathcal{TR}$ is not *forced* to distinguish between transactions that do updates and queries; but the distinction always exists as an *option.* This is possible because an application can introduce logical sorts, one for queries, and one for updates. This is comparable to the distinction in deductive databases between two sorts of predicates, base and derived. Even when an application makes such distinctions, $\mathcal{TR}$ treats all predicates uniformly, just as classical logic does. Most other logics are not capable of this uniform treatment, since for them, queries are propositions, but updates (or actions) are entities of a different and incompatible nature. For instance, in situation calculus [71], actions are function terms, while in Dynamic and Process Logics [49, 50], actions are modal operators.

It is worth noting that relational databases already attempt to treat queries and updates uniformly. Typically, the two are lumped together under the rubric "Data Manipulation Language," or DML. Indeed, SQL updates are often referred to as queries. However, queries and updates are not well-integrated in SQL. One way to better integrate them is to extend SQL to include transactions. As a logic of transactions, $\mathcal{TR}$ may offer a formal basis for such an extension.

Although $\mathcal{TR}$ is different from logics of action, it is comparable to declarative query languages, like Prolog and SQL. In Prolog, for instance, one does not reason about logic programs. Instead, one specifies and executes them. Thus, given a logic program and a database of atomic facts, one can ask, "Can $p(a)$ be inferred from the current database?", but one cannot ask, "Can $p(a)$ *always* be inferred, *i.e.*, from *every* database?" Logics of action are aimed at the latter kind of question (applied to actions). $\mathcal{TR}$ is aimed at the former, but it can also handle the latter as efficiently as most logics can.

The system that comes closest in spirit to $\mathcal{TR}$ is Prolog. Although Prolog is not a logic of action or time *per se*, transactions can be defined in Prolog by using the operators *assert* and *retract*. Prolog transactions have some of the properties that we want to model: (i) updates are real, not hypothetical; (ii) named procedures can be composed from simpler procedures; (iii) all predicates can have both a truth value and a side effect on the database; and (iv) the frame problem is not an issue. Unfortunately, updates in Prolog are non-logical operations. Thus, each time a programmer uses "assert" or "retract" operators, he moves further away from declarative programming. Due to these drawbacks, transactions are often the most awkward of Prolog programs, and the most difficult to understand, debug, and maintain.

In addition, updates in Prolog are not integrated into a complete logical system. It is therefore not clear how *assert* and *retract* should interact with other logical operators such as disjunction and negation. For instance, what does $assert(X) \vee assert(Y)$ mean? or $\neg assert(X)$? or $assert(X) \leftarrow retract(Y)$? Also, how does one logically account for the fact that the order of updates is important? Finally, what does it mean to update a database that contains arbitrary logical formulas? None of these questions is addressed by Prolog's classical semantics or its update operators.

Transaction Logic

$\mathcal{TR}$ provides a general solution to the aforementioned limitations, both of Prolog and of the logics of action and time. It provides a syntax and a semantics in which elementary updates can be combined logically to build complex transactions.

The emphasis in $\mathcal{TR}$ is not on the specification of elementary updates, but on their logical combination into programs. This approach is in accordance with programming practice. In databases and logic programming (as in Pascal and C), application programmers spend little if any time specifying elementary updates, and a lot of time combining them into complex transactions and programs. For them, specifying elementary updates is not an issue, but combining them is.

Nevertheless, the set of elementary updates is an important feature of a language, and can determine its domain of application. Examples of elementary updates include Prolog's `assert` and `retract` operators, as well as SQL-style bulk updates. In scientific databases, elementary updates might include complex numerical calculations, such as the fast Fourier transform and matrix inversion. Unfortunately, there is probably no small fixed set of elementary updates that is best for all applications, and future applications will demand new ones.

For this reason, $\mathcal{TR}$ was designed to be orthogonal to the choice of elementary updates. The semantics of $\mathcal{TR}$ can thus accommodate *any* set of elementary updates. In effect, $\mathcal{TR}$ treats a database as a collection of abstract datatypes, each with its own special-purpose access methods. These methods are provided to $\mathcal{TR}$ as elementary operations, and they are combined by $\mathcal{TR}$ programs into complex transactions. This approach separates the specification of elementary operations from the logic of combining them. As we shall see, this separation has two main benefits: (i) it allows us to develop a logic for combining and programming transactions without committing to a particular theory of elementary updates; and (ii) it allows $\mathcal{TR}$ to accommodate a wide variety of database semantics, from classical to non-monotonic to various other non-standard logics.

To combine actions, $\mathcal{TR}$ extends the syntax of first-order logic with a single binary operator, $\otimes$, which we call *serial conjunction*. Intuitively, the formula $\psi \otimes \phi$ means, "First execute transaction ψ, and then execute transaction ϕ." Serial conjunction may be combined arbitrarily with the standard logical operators, such as $\wedge$, $\vee$ and $\neg$, to build a wide variety of formulas. Thus, each logical proposition in $\mathcal{TR}$ represents an action, and each logical operator can be interpreted as an action constructor. The idea of interpreting logical propositions as actions has also been considered by van Bentham [95], although his semantics is very different, and was not intended for programming database transactions. Further comparison is provided in Section 11.

Semantically, $\mathcal{TR}$ is related to Process Logic [50], but is different from it in several important respects. As in Process Logic, a model in $\mathcal{TR}$ consists of a set of states, and actions cause transitions from one state to another. In fact, an action may cause a sequence of transitions, passing from an initial state, through intermediate states, to a final state. Like Process Logic, formulas in $\mathcal{TR}$ are *not* evaluated at states. Instead, they are evaluated on *paths*, which are sequences of states. This property enables $\mathcal{TR}$ to express a wide range of constraints on transaction execution. Unlike Process Logic, however, $\mathcal{TR}$ does not distinguish between programs and propositions: In $\mathcal{TR}$, every formula is both a proposition and a program. This is a natural way to model database transactions, since a transaction can both change the database and—as a query—return an answer. Thus, pure queries and pure updates are merely two extremes on a spectrum of possible transactions. In fact, in $\mathcal{TR}$, a single notion of truth subsumes two ideas: (i) the classical concept of a truth value, and (ii) the hitherto non-logical concept of a "side effect" on the database. This uniformity renders $\mathcal{TR}$ suitable for a number of diverse applications in object-oriented databases and logic programming. More discussion of the relationship between $\mathcal{TR}$ and process logics appears in Section 11, where several other related formalisms are also discussed.

Like classical logic, $\mathcal{TR}$ has a "Horn" version that is of particular interest for logic programming. In Horn $\mathcal{TR}$, a transaction is defined by Prolog-style rules in which the premise specifies a *sequence* of queries and updates. Furthermore, just as $\mathcal{TR}$ is an extension of classical first-order logic, Horn $\mathcal{TR}$ is an extension of classical Horn-clause logic. Because of its importance, much of this paper focuses on Horn $\mathcal{TR}$.

Horn $\mathcal{TR}$ has a clean and simple proof theory that is sound and complete with respect to the model theory of Section 5. This proof theory is described in detail in Section 6. Two versions of the proof theory are presented, one suitable for bottom-up inference and another for top-down inference. Section 8 extends the proof theory to hypothetical updates. The proof theory for Horn $\mathcal{TR}$ is much simpler than the proof theory for full $\mathcal{TR}$, which involves encoding $\mathcal{TR}$ in a more general logic of state transitions. The latter will be discussed in a forthcoming report [21]. Proofs of soundness and completeness of the inference system for $\mathcal{TR}$ are given in the appendices.

Other aspects of programming in $\mathcal{TR}$, such as negation-as-failure and updates to the rule-base of the program are discussed in Sections 9 and 10.

2 Overview and Introductory Examples

From the user's point of view, using $\mathcal{TR}$ is similar to using Prolog or using a relational database system. That is, the user may specify rules, and he may pose queries and request updates. In $\mathcal{TR}$, the user sees no obvious or immediate difference between queries and updates. An update is just a query with side effects (which can be detected only by issuing subsequent queries). In general, the user issues transactions, and the system responds by displaying answers and updating the database. This section provides simple examples of how this behavior appears to the user and how it is described formally. The examples also illustrate several dimensions of $\mathcal{TR}$'s capabilities.

One of these capabilities should be mentioned at the outset: *non-deterministic* transactions. Non-determinism has applications in many areas, but it is especially well-suited for advanced applications, such as those found in Artificial Intelligence. For instance, the user of a robot simulator might instruct the robot to build a stack of three blocks, but he may not tell (or care) which three blocks to use. Likewise, the user of a CAD system might request the system to run an electrical line from one point to another, without fixing the exact route, except in the form of loose constraints (*e.g.*, do not run the line too close to wet or exposed areas). These are the kinds of non-deterministic transactions that $\mathcal{TR}$ can specify. In such transactions, the final state of the database is indeterminate, *i.e.*, it cannot be predicted at the outset, as it depends on choices made by the system at run time. $\mathcal{TR}$ enables users to specify what choices are allowed. When a user issues a non-deterministic transaction, the system makes particular choices (which may be implementation-dependent), putting the database into one of the allowed new states.

For all but the most elementary applications, a transaction execution will be characterized not just by an initial and a final state, but rather by a sequence of *intermediate* states that it passes through. For example, as a robot simulator piles block upon block upon block, the transaction execution will pass from state to state to state. Like the final state, intermediate states may not be uniquely determined at the start of the execution. For example, the robot may have some (non-deterministic) choice as to which block to grasp next. We call such a sequence of database states the *execution path* of the transaction. $\mathcal{TR}$ represents execution paths explicitly. By doing so, it can express a wide range of constraints on transaction execution. For example, a user may require every intermediate state to satisfy some condition, or he may forbid certain sequences of states.

To describe the execution of transactions formally, we use statements of the following form, which express a form of logical entailment in $\mathcal{TR}$, called *executional entailment*:

$$\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \psi \tag{1}$$

Here, $\mathbf{P}$ and each $\mathbf{D}_i$ is a logical formula, as is ψ. Intuitively, $\mathbf{P}$ is a set of transaction definitions, ψ is a transaction invocation, and $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is a sequence of databases, representing all the states of transaction execution. Statement (1) means that $\mathbf{D}_0, \ldots, \mathbf{D}_n$ is an execution path of transaction ψ. That is, if the current database state is $\mathbf{D}_0$, and if the user issues the transaction ψ (by typing $?-\ \psi$, as in Prolog), then the database *may* go from state $\mathbf{D}_0$ to state $\mathbf{D}_1$, to state $\mathbf{D}_2$, etc., until it finally reaches state $\mathbf{D}_n$, after which the transaction terminates. We emphasize the word "may" because ψ may be a non-deterministic transaction. As such, it may have many execution paths beginning at $\mathbf{D}_0$, possibly of different length. The proof theory for $\mathcal{TR}$ can derive each of these paths, but only one of them will be (non-deterministically) selected as the actual execution path; the final state, $\mathbf{D}_n$, of that path then becomes the new database.

Unlike other formalisms, $\mathcal{TR}$ does not draw a thick line between transactions and queries. In fact, any transaction that does not cause a state change can be viewed as a query. This state of affairs is formally expressed by the statement $\mathbf{P}, \mathbf{D}_0 \models \psi$, a special case of statement (1) in which $n = 0$. In this case, $\mathbf{D}_0$ is a sequence of databases of length 1. This uniform treatment of transactions and queries is crucial for successful adaptation of $\mathcal{TR}$ to the object-oriented domain, because object-oriented systems do not syntactically distinguish between state-changing methods and information-retrieving methods.

Several other aspects of statement (1) should be mentioned at this point. First, the transaction base $\mathbf{P}$ can be *any* formula in $\mathcal{TR}$. In practice, though, it will often be a conjunction of Prolog-like rules, which we will represent as a finite set of formulas. In any case, regardless of its form, we call $\mathbf{P}$ a *transaction base*. As the name suggests, transaction bases define transactions and, as a special case, queries. The databases $\mathbf{D}_1, \ldots, \mathbf{D}_n$ are also represented as finite sets of formulas; however, they are classical first-order formulas. These formulas are not restricted to be ground atomic facts, though, and it is entirely possible for each $\mathbf{D}_i$ to contain Prolog-like rules. The difference between a database and a transaction base is that formulas in a transaction base may use the full syntax of $\mathcal{TR}$, while formulas in a database are limited to the syntax of first-order logic, a subset of $\mathcal{TR}$.

Statements of the form (1) provide a formal view of all that is possible in the transaction system. However, this formal view is somewhat different from the user's view of the system. From the user's perspective, a $\mathcal{TR}$ logic program has two parts: a transaction base, $\mathbf{P}$, which the programmer provides,[2] and a *single* current database, $\mathbf{D}$, which he wishes to access and possibly modify. The transaction base is immutable; that is, it cannot be changed by other transactions. In contrast, the database constantly changes—it is updated when the user executes transactions defined in $\mathbf{P}$. In particular, if $\mathbf{D}$ contains Prolog-style rules, they also can be modified.

The rest of this section illustrates our notation and the capabilities of $\mathcal{TR}$ through a number of simple examples. The examples illustrate how $\mathcal{TR}$ uses logical operators to combine simple actions into complex ones. These operators include the standard connectives, such as $\wedge$, $\vee$ and $\neg$, and a new connective of sequential composition, $\otimes$. For the purpose of illustration, our example databases are sets of ground atomic formulas, and insertion and deletion of atomic formulas are elementary update operators. However, $\mathcal{TR}$ is restricted neither to this type of databases, nor to these particular elementary update operators.

2.1 Elementary Updates and Sequential Composition

This subsection introduces some of the basic ideas of $\mathcal{TR}$. Starting with purely declarative queries, we show how $\mathcal{TR}$ extends classical logic to represent procedural phenomena, such as updates and action-sequences. We then show how these ideas can be combined to form more complex transactions. The examples of this subsection focus on simple, deterministic transactions, ones that a user might issue directly to a $\mathcal{TR}$ interpreter. Subsequent subsections illustrate more sophisticated transactions.

Since $\mathcal{TR}$ queries are a special kind of transaction, they are defined in the transaction base, $\mathbf{P}$, just as other transactions are. Typically, these definitions are constructed from formulas that are akin to the Horn rules of classical logic programming. We call these formulas *serial-Horn* rules. These rules draw inferences from base facts just as logic programs and deductive databases do. In fact, if the transaction base consists entirely of first-order formulas, then inference in $\mathcal{TR}$ reduces to first-order

[2]The term "programmer" should be taken here in a broad sense. For instance, parts of the transaction base may be provided by the knowledge-base engineer, by the database administrator, or by any number of system or application programmers.

inference. That is, if $\mathbf{P}$ and ϕ are both first-order formulas, then

$$\mathbf{P}, \mathbf{D} \models \phi \qquad \textit{iff} \qquad \mathbf{P} \wedge \mathbf{D} \models^c \phi$$

where $\models^c$ denotes classical entailment. In this way, classical logic—the medium in which queries are traditionally expressed—is the starting point for $\mathcal{TR}$. Logic programs, deductive databases, and first-order knowledge-bases can thus be adapted for use in $\mathcal{TR}$ with minimal (if any) change. That is, they are upward-compatible with $\mathcal{TR}$.

Example 2.1 (Classical Inference) Suppose that the database $\mathbf{D}$ contains the atom $lucky$, and that the transaction base $\mathbf{P}$ contains the rule $happy \leftarrow lucky$. By typing $?-\ happy$, the user is posing a query, asking the $\mathcal{TR}$ interpreter if *happy* is true. In this case, *happy* can be inferred from the contents of the database and the transaction base, so the interpreter returns "true." Likewise, the interpreter returns "true" if the user types $?-\ lucky$ or $?-\ lucky \wedge happy$. Note that in each case, the database remains unchanged. We represent this behavior formally by the following three statements:

$$\mathbf{P}, \mathbf{D} \models lucky \qquad \mathbf{P}, \mathbf{D} \models happy \qquad \mathbf{P}, \mathbf{D} \models lucky \wedge happy$$

□

In $\mathcal{TR}$, all transactions are a combination of queries and updates. Queries do not change the database, and can be expressed in classical logic, as Example 2.1 shows. In contrast, updates do change the database, and are expressed in an extension of classical logic. We call the simplest kind of updates *elementary updates* or *elementary state transitions*. In principle, there are no restrictions on the changes that an elementary update can make to a database, though in practice, we expect them to be simple and efficient. In this paper, we use the insertion and deletion of atomic formulas as canonical examples of elementary updates. Sections 7.5 and 7.6 illustrate other kinds of elementary update—*bulk* updates and non-deterministic *sampling*.

Elementary updates are atomic in that they cannot be decomposed into simpler updates. We therefore represent them by atomic formulas. Like all formulas, elementary updates can have *both* a truth value *and* a side effect on the database. We represent this idea formally by executional entailments of the following form:

$$\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \models u$$

This statement says that the atomic formula u is (the name of) an update that changes the database from state $\mathbf{D}_1$ to state $\mathbf{D}_2$.

Although any atomic formula can be an update, it is a good programming practice to reserve a special set of predicate symbols for this purpose. For example, in this section, for each predicate symbol p, we use another predicate symbol, $p.ins$, to represent insertions into p. That is, $p.ins(\overline{x})$ is an elementary update that transforms a database $\mathbf{D}$ into the database $\mathbf{D} + \{p(\overline{x})\}$. Likewise, we use the predicate symbol $p.del$ to represent deletions from p. That is, $p.del(\overline{x})$ is an elementary update that transforms a database $\mathbf{D}$ into the database $\mathbf{D} - \{p(\overline{x})\}$. Here, $\mathbf{D}$ is a set of ground atomic formulas, and the operators "+" and "−" denote set union and set difference, respectively. Thus, the expression $\{a, b, c\} + \{d\}$ denotes the set $\{a, b, c, d\}$, which in turn stands for the formula $a \wedge b \wedge c \wedge d$.

Observe that as defined here, $p.ins$ and $p.del$ have the following two properties:

- They are not strict. That is, $p \notin \mathbf{D}$ is not a precondition for doing $p.ins$, and $p \in \mathbf{D}$ is not a precondition for doing $p.del$.

- Only one elementary update can take place at a time. Intuitively, $p.ins$ means, "insert p into the database, and do nothing else." Likewise for deletion.

These properties are convenient for illustrative purposes, and are used in some of the examples that follow. We emphasize, however, that the theory of $\mathcal{TR}$ developed in this paper does not depend on these properties, nor on these particular updates, nor on any set of updates at all.

Example 2.2 (Elementary Updates) Suppose in is a binary predicate symbol. Then the atoms $in.ins(sky, pie)$ and $in.del(sky, pie)$ are elementary updates. Intuitively, $in.ins(sky, pie)$ means, "insert the atom $in(sky, pie)$ into the database." Likewise, the atom $in.del(sky, pie)$ means, "delete $in(sky, pie)$ from the database." From the user's perspective, $?-\ in.ins(sky, pie)$ is a request to change the database from $\mathbf{D}$ to $\mathbf{D} + \{in(sky, pie)\}$. Likewise, typing $?-\ in.del(sky, pie)$ changes the database from $\mathbf{D}$ to $\mathbf{D} - \{in(sky, pie)\}$. We express this behaviour formally by the following two statements, which are true for any transaction base $\mathbf{P}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{in(sky, pie)\} \models in.ins(sky, pie)$$
$$\mathbf{P}, \mathbf{D}, \mathbf{D} - \{in(sky, pie)\} \models in.del(sky, pie)$$

□

We emphasize here that insertions and deletions are *not* built into the semantics of $\mathcal{TR}$. Thus, there is no *intrinsic* connection between the names p, $p.ins$ and $p.del$. Our use of these names is merely a convention for purposes of illustration. In fact, p, $p.ins$, and $p.del$ are ordinary predicates of $\mathcal{TR}$ and the connection between them is established outside the logic, via the concept of *state transition oracle*, as explained in Section 4.2.

$\mathcal{TR}$ is not committed to any particular set of elementary updates. Indeed, we expect that each database system will want its own repertoire of elementary updates, tuned to the applications at hand. Within each database system, though, we expect this repertoire of updates to be relatively stable. Thus, for most users of a given system, it will *appear* as though there is a fixed set of elementary updates. The user's job is to combine elementary updates into complex transactions using facilities provided in $\mathcal{TR}$.

A basic way of combining transactions is to sequence them, *i.e.*, to execute them one after another. For example, one might take money out of one account and then, if the withdrawal succeeds, deposit the money into another account. To combine transactions sequentially, we extend classical logic with a new binary connective, $\otimes$, which we call *serial conjunction*. The formula $\psi \otimes \phi$ denotes the composite transaction consisting of transaction ψ followed by transaction ϕ.[3] Unlike elementary updates, sequential transactions often have intermediate states, as well as initial and final states. We express this behavior formally by statements like the following:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \models \psi \otimes \phi$$

This statement says that the transaction $\psi \otimes \phi$ changes the database from $\mathbf{D}_0$ to $\mathbf{D}_1$ to $\mathbf{D}_2$. Here, $\mathbf{D}_1$ is an intermediate state.

[3] Logics of action sometimes use a semicolon to denote serial conjunction. We avoid this notation for two reasons: (i) in Prolog, a semicolon means disjunction, and (ii) there is no natural symbol to stand for the dual of the semicolon (as we shall see, this dual is a useful operator in its own right). In contrast, $\oplus$ naturally stands for the dual of $\otimes$. We can therefore rewrite $\neg(\psi \otimes \phi)$ as $\neg\psi \oplus \neg\phi$.

Example 2.3 (Serial Conjunction) The expression $poor.ins \otimes sad.ins$, where *poor* and *sad* are ground atomic formulas, denotes a sequence of two insertions. This transaction means, "First insert *poor* into the database, and then insert *sad*." Thus, if the initial database is $\mathbf{D}$, and if the user issues a transaction by typing $?-\ poor.ins \otimes sad.ins$, then during execution, the database will change from $\mathbf{D}$ to $\mathbf{D} + \{poor\}$ to $\mathbf{D} + \{poor, sad\}$. We express this behaviour formally by the following statement, which is true for any transaction base $\mathbf{P}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{poor\}, \mathbf{D} + \{poor, sad\} \quad \models \quad poor.ins \otimes sad.ins$$

□

2.2 Tests and Conditions

Serial conjunction applies not only to updates, but to any transaction, including queries. We can therefore place queries that act as tests anywhere in a sequence of updates. This allows us to monitor the progress of a transaction, and force it to abort if certain conditions are not met.

In the simplest case, we can define pre-conditions by placing a query at the beginning of a transaction. For instance, before withdrawing money from an account, we should check that enough money is available. If there is, then the transaction can continue, and the database can be updated; if not, then the transaction should abort, and the database should remain in its original state.

Example 2.4 (Pre-Conditions) If the atom *short* is a query (*i.e.*, has no side effects), then the expression $short \otimes sweet.ins$ denotes a test followed by an update. Intuitively, it means, "check that *short* is true, and if so, insert *sweet* into the database." In other words, if *short* is true in the initial database state, then the transaction succeeds and the database is changed from $\mathbf{D}$ to $\mathbf{D} + \{sweet\}$; otherwise, the transaction fails and the database remains in the initial state, $\mathbf{D}$. Formally,

$$\begin{array}{lll} \textit{if} & \mathbf{P}, \mathbf{D} & \models \ short \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{sweet\} & \models \ short \otimes sweet.ins \end{array}$$

□

By placing tests at other points in a transaction, sophisticated behavior can be specified in a simple and natural way. For instance, tests at the end of a transaction act as post-conditions that query the *final* state of the database. If these tests succeed, then the transaction commits and the database is permanently changed. Otherwise, if the tests fail, then the transaction aborts and the database is rolled back to its original state. Post-conditions are particularly useful for eliminating transactions that have forbidden side effects. For instance, a move in chess is forbidden if it puts you into check. Likewise, changes to a circuit design may be forbidden if the new design violates certain conditions (*e.g.*, limits on cost, size, or power consumption). It is worth noting that post-conditions can be awkward, if not impossible to express in other formalism of action, such as the situation calculus [71, 85].

Example 2.5 (Post-Conditions) If the atom *happy* is a query (*i.e.*, has no side effects), then the expression $won.ins \otimes happy$ denotes an update followed by a test. Intuitively, it means, "insert *won* into the database, and then check that *happy* is true." In other words, the database state is first changed from $\mathbf{D}$ to $\mathbf{D} + \{won\}$. If *happy* is true in the final state, then the transaction succeeds and the change is committed; otherwise, the transaction fails and the database is rolled back to the initial state, $\mathbf{D}$. Formally,

$$\begin{array}{lllll} \textit{if} & \mathbf{P}, \mathbf{D} + \{won\} & \models & happy \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{won\} & \models & won.ins \otimes happy \end{array}$$

□

2.3 Non-Deterministic Transactions

In addition to sequencing, transactions in $\mathcal{TR}$ can be combined using any of the classical connectives. Two of these connectives, disjunction and existential quantification, can be used to build non-deterministic transactions. For instance, the formula $\psi \vee \phi$ means, "do transaction ψ or do transaction ϕ." Likewise, the formula $\exists X\, \phi(X)$ means, "do transaction $\phi(c)$ for some value c."

Example 2.6 (Classical Disjunction) Suppose the transaction is $?-\ won.ins \vee lost.ins$. Then, after execution, the final database will be either $\mathbf{D} + \{won\}$ or $\mathbf{D} + \{lost\}$. In effect, this states that the user does not care which update will be made, so the system chooses one of them and executes it. Formally, both of the following statements are true, for any transaction base $\mathbf{P}$:

$$\begin{array}{lll} \mathbf{P}, \mathbf{D}, \mathbf{D} + \{won\} & \models & won.ins \vee lost.ins \\ \mathbf{P}, \mathbf{D}, \mathbf{D} + \{lost\} & \models & won.ins \vee lost.ins \end{array}$$

□

Like classical disjunction, disjunction in $\mathcal{TR}$ is non-exclusive. Thus, to execute $\psi \vee \phi$, the system may execute $\psi \wedge \phi$, unless ψ and ϕ are mutually exclusive actions. The elementary actions used in this section *are* mutually exclusive, since only one can execute at a time, as noted in Section 2.1. Thus, in Example 2.6, although the system can execute $won.ins \vee lost.ins$, it cannot execute $won.ins \wedge lost.ins$. Section 2.6 expands upon this point.

In addition to defining non-deterministic transactions, it is worth noting that classical disjunction is also needed (in combination with negation) to define implication, exactly as in classical logic.

When classical disjunction is used to define transactions, the non-deterministic alternatives are fixed and independent of the database. Non-determinism can also be achieved with existential quantification, except that the number of alternatives can be infinite and data-dependent.

Example 2.7 (Existential Quantification) Suppose the user issues the transaction $?-\ has.ins(X)$ (or, equivalently, $?-\ \exists X\, has.ins(X)$). The database system will then insert a tuple, $\langle c \rangle$, into relation *has* for some non-deterministically chosen value c. That is, after execution, the database state will be $\mathbf{D} + \{has(c)\}$. Formally, the following statement is true for every value c:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{has(c)\} \models \exists X\, has.ins(X)$$

□

Example 2.8 (Non-Deterministic Copy) Consider $?-\ \exists X\, [handsome(X) \otimes hired.ins(X)]$, a transaction that requests the system to copy a tuple from *handsome* to *hired* non-deterministically. That is, the system will first choose some arbitrary tuple, $\langle mary \rangle$, from relation *handsome*, and then insert the tuple into relation *hired*. Formally, for every value c,

$$\begin{array}{llll} \textit{if} & \mathbf{P}, \mathbf{D} & \models & handsome(c) \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{hired(c)\} & \models & \exists X\, [handsome(X) \otimes hired.ins(X)] \end{array}$$

For instance, suppose the database $\mathbf{D}$ contains the atoms $handsome(mary)$, $handsome(bill)$ and $handsome(kate)$. Then there are three choices for X: $mary$, $bill$, and $kate$. Each choice results in a different update, since either $hired(mary)$, $hired(bill)$, or $hired(kate)$ will be inserted into $\mathbf{D}$. Formally, the following three statements are all true:

$$\begin{aligned}
\mathbf{P}, \mathbf{D}, \mathbf{D} + \{hired(mary)\} \quad &\models \quad \exists X\, [handsome(X) \otimes hired.ins(X)] \\
\mathbf{P}, \mathbf{D}, \mathbf{D} + \{hired(bill)\} \quad &\models \quad \exists X\, [handsome(X) \otimes hired.ins(X)] \\
\mathbf{P}, \mathbf{D}, \mathbf{D} + \{hired(kate)\} \quad &\models \quad \exists X\, [handsome(X) \otimes hired.ins(X)]
\end{aligned}$$

Note that even though the above transaction can be executed in several ways (with different values for X), only *one* value will be chosen in any concrete execution, and so only one person will be hired. □

2.4 Rules

Rules are formulas of the form $p \leftarrow \phi$, where p is an atomic formula and ϕ is any $\mathcal{TR}$ formula. As in classical logic, this formula is just a convenient abbreviation for the formula $p \vee \neg\phi$. This is the formal, declarative interpretation of rules. In addition, rules in $\mathcal{TR}$ have a procedural interpretation. Intuitively, the formula $p \leftarrow \phi$ means, "to execute p, it is sufficient to execute ϕ." This procedural interpretation is important because it provides $\mathcal{TR}$ with a subroutine facility and makes logic programming possible. For instance, in the rule $p(X) \leftarrow \phi$, the predicate symbol p acts as the name of a procedure, the variable X acts as a parameter, and the formula ϕ acts as the procedure body or definition (exactly as in Horn-clause logic programming). Although the rule-body may be any $\mathcal{TR}$ formula, it will frequently be a serial goal of sorts. In this case, the rule has the form $a_0 \leftarrow a_1 \otimes a_2 \otimes \ldots \otimes a_n$, where each a_i is an atomic formula. With such rules, users can define transaction subroutines and write transaction logic programs. Note that this facility is possible because transactions are represented by predicates. This property distinguishes $\mathcal{TR}$ from other logics of action, especially those in which actions are modal operators. In such logics, subroutines are awkward, if not impossible, to express. Finally, for notational convenience, we assume that all free variables in a rule are universally quantified outside the rule. Thus, the rule $p(X) \leftarrow \phi$ is simply an abbreviation for $\forall X\, [p(X) \leftarrow \phi]$.

Example 2.9 (A Simple Rule) The rule $win \leftarrow divided.ins \otimes conquered.ins$ defines a transaction called win. Intuitively, this rule says, "to do win, first insert $divided$ into the database, and then insert $conquered$." Thus, if the user invokes transaction $?-\ win$, then the database changes from $\mathbf{D}$ to $\mathbf{D} + \{divided\}$ to $\mathbf{D} + \{divided, conquered\}$. Formally, if transaction base $\mathbf{P}$ contains this rule, then the following statement is true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{divided\}, \mathbf{D} + \{divided, conquered\} \quad \models \quad win$$

The simplicity of our examples may have blurred an important point made earlier, namely, that the database itself may be a full-fledged deductive database. For instance, $\mathbf{D}$ may contain the rule $unhappy \longleftarrow divided \wedge conquered$. In this case, the query $?-\ unhappy$ will be answered affirmatively if asked *after* the transaction win has been executed. That is, the following statement is true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{divided\}, \mathbf{D} + \{divided, conquered\} \quad \models \quad win \otimes unhappy$$

In addition, as explained in Section 10, transactions can modify the rules found in the database. For instance, a transaction may be programmed to delete the formula $unhappy \longleftarrow divided \wedge conquered$ from the database. □

The rule in Example 2.9 defines a deterministic transaction. However, rules can also define non-deterministic transactions. To take a simple example, the rule $c \leftarrow \phi \vee \psi$ defines c to be the non-deterministic transaction $\phi \vee \psi$. More generally, rules can define non-deterministic transactions even without mentioning disjunctions explicitly. This is possible because, as in classical logic, a rule like $c \leftarrow (\phi \vee \psi)$ is equivalent to a conjunction of the two rules, $(c \leftarrow \phi) \wedge (c \leftarrow \psi)$.

Example 2.10 (Flipping Coins) Let $up(coin, head)$ mean that $coin$ is lying face up. Likewise, $up(coin, tail)$ means that its tail is facing up. The following transaction base $\mathbf{P}$ defines the action of flipping a coin:

$$flip(Coin) \leftarrow up(Coin, Face) \otimes up.del(Coin, Face) \otimes up.ins(Coin, head)$$
$$flip(Coin) \leftarrow up(Coin, Face) \otimes up.del(Coin, Face) \otimes up.ins(Coin, tail)$$

Given a coin, say $dime1$, these rules say the following: to flip $dime1$, first find out what side of $dime1$ is facing up; then delete this fact from the database; then either insert $up(dime1, head)$ or insert $up(dime1, tail)$. Thus, $?-\ flip(dime1)$ is a non-deterministic transaction. Formally, if $dime1$ initially has its head side up, then flipping $dime1$ is represented by the following two statements:

$$\mathbf{P}, \{up(dime1, head)\}, \{\}, \{up(dime1, head)\} \models flip(dime1)$$
$$\mathbf{P}, \{up(dime1, head)\}, \{\}, \{up(dime1, tail)\} \models flip(dime1)$$

This means that we cannot know in advance what the exact outcome of a flipping action will be. □

Even a single rule can give rise to non-determinism. This is possible because, as in classical logic, a rule like $\forall X\, [p \leftarrow \phi(X)]$ is equivalent to the rule $p \leftarrow \exists X\, \phi(X)$, where X does not occur in p, and the body of the latter rule is a non-deterministic transaction. Another way to see this is to think of the former rule as a conjunction of its instances, $p \longleftarrow \phi(a)$, for all a. Each of these instances provides an alternate way of executing p, thereby introducing non-determinism.

Example 2.11 (Non-Deterministic Copy, Continued) In Example 2.8 we showed that it is possible to non-deterministically copy a tuple from one relation to another. We can give this transaction a name by adding the following rule to the transaction base $\mathbf{P}$:

$$hire \leftarrow handsome(X) \otimes hired.ins(X)$$

Formally, this rule is equivalent to the formula $hire \leftarrow \exists X\, [handsome(X) \otimes hired.ins(X)]$. Note that the body of this rule is exactly the formula used in Example 2.8. Intuitively, this rule says, "to execute *hire*, first choose a tuple from relation *handsome* non-deterministically, and then insert this tuple into relation *hired*." Note that the set of non-deterministic alternatives is data-dependent. Formally, for every *handsome*-value c,

$$\begin{array}{ll} \textit{if} & \mathbf{P}, \mathbf{D} \models handsome(c) \\ \textit{then} & \mathbf{P}, \mathbf{D}, \mathbf{D} + \{hired(c)\} \models hire \end{array}$$

This means that we cannot know in advance what the exact outcome of the hiring action will be. However, if the rule above is the only rule defining *hire*, then we can make the following statement, which is true for any pair of databases, $\mathbf{D}_1$ and $\mathbf{D}_2$:

$$\begin{array}{llcl} \textit{if} & \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 & \models & hire \\ \textit{then} & \mathbf{P}, \mathbf{D}_1 & \models & handsome(c) \\ \textit{and} & \mathbf{P}, \mathbf{D}_2 & \models & hired(c) \end{array}$$

for some value c. □

2.5 Transaction Bases

This section gives simple but realistic examples of transaction bases comprised of finite sets of rules. The examples show how updates can be combined with queries to define complex transactions, and they show how $\mathcal{TR}$ improves upon Prolog's update operators.

Example 2.12 (Financial Transactions) Suppose the balance of a bank account is given by the relation $balance(Acct, Amt)$. To modify this relation, we are provided with a pair of elementary update operations: $balance.del(Acct, Amt)$, to delete a tuple from the relation; and $balance.ins(Acct, Amt)$, which inserts a tuple into the relation. Using these two updates, we define four transactions: $change\_balance(Acct, Bal1, Bal2)$, to change the balance of an account from one amount to another; $withdraw(Amt, Acct)$, to withdraw an amount from an account; $deposit(Amt, Acct)$, to deposit an amount into an account; and $transfer(Amt, Acct1, Acct2)$, to transfer an amount from one account to another. In the *transfer* transaction, we also require that the debited account should not to be overdrawn. These transactions are defined by the following four rules:

$$transfer(Amt, Acct1, Acct2) \leftarrow withdraw(Amt, Acct1) \otimes deposit(Amt, Acct2)$$

$$\begin{aligned} withdraw(Amt, Acct) \leftarrow\ & balance(Acct, Bal) \otimes Bal \geq Amt \\ & \otimes change\_balance(Acct, Bal, Bal - Amt) \end{aligned}$$

$$deposit(Amt, Acct) \leftarrow balance(Acct, Bal) \otimes change\_balance(Acct, Bal, Bal + Amt)$$

$$change\_balance(Acct, Bal1, Bal2) \leftarrow balance.del(Acct, Bal1) \otimes balance.ins(Acct, Bal2)$$

Note that, in these rules, the atom $balance(Acct, Bal)$ is a query that retrieves the balance of the specified account, while $Bal \geq Amt$ is a test. All other atoms are updates. □

Observe that the rules in Example 2.12 can easily be rewritten in Prolog, by replacing "$\otimes$" with "," and replacing the elementary transitions, *balance.ins* and *balance.del*, with *assert* and *retract*, respectively. However, the resulting, apparently innocuous, Prolog program will not execute correctly! The problem is that Prolog does not undo updates during backtracking. As an example, consider a transaction involving two transfers, defined as follows:

$$?-\ transfer(fee, client, broker) \otimes transfer(cost, client, seller)$$

That is, a fee is transferred from a client to a broker, and then a cost is transferred from the client to a seller. Because this is intended to be a transaction, it must behave atomically; that is, it must execute entirely or not at all. Thus, if the second transfer fails, then the first one must be rolled back. In this respect, $\mathcal{TR}$ behaves correctly. Prolog, however, does not, since it commits updates immediately and does not undo partially executed transactions. Thus, if the second transfer above were to fail (say, because the client's account would be overdrawn by the transaction), then Prolog would *not* undo the first one, thus leaving the database in an inconsistent state.

Getting around this problem takes much out of the simplicity of Prolog programming. In fact, the non-logical behavior of Prolog updates is notorious for making Prolog programs cumbersome and heavily dependent on Prolog's backtracking strategy. Some Prologs provide the means to implement backtrackable updates through the use of non-logical operations, such as *assert*, *retract*, and cut (!). However, the logic behind the resulting update operations has not been developed. Transaction Logic can be considered to provide the missing semantics and proof theory, although there are many fine differences between simplistic implementations of backtrackable updates and the logic.

The next example uses robot actions to illustrate non-deterministic rules. Planning of robot actions is discussed in detail in Section 7.9.

Example 2.13 (Non-Deterministic Robot Actions) The rules, below, define actions that simulate the movements of a robot arm in the blocks world [79]. States of this world are defined in terms of four database predicates: $on(x,y)$, which says that block x is on top of block y; $wider(x,y)$, which says that x is wider than y; $isclear(x)$, which says that nothing is on top of block x; and $color(x,c)$, which says that c is the color of block x. The rules below define six actions that change the state of the world. Each action evaluates its premises in the order given, and the action fails if any of its premises fails (in which case the database is left in its original state).

$$stackSameColor(Z) \leftarrow color(Z,C) \otimes stackTwoColors(C,C,Z).$$
$$stackTwoColors(C_1,C_2,Z) \leftarrow color(X,C_1) \otimes color(Y,C_2) \otimes stackTwoBlocks(X,Y,Z)$$
$$stackTwoBlocks(X,Y,Z) \leftarrow move(Y,Z) \otimes move(X,Y)$$
$$move(X,Y) \leftarrow pickup(X) \otimes putdown(X,Y)$$
$$pickup(X) \leftarrow isclear(X) \otimes on(X,Y) \otimes on.del(X,Y) \otimes isclear.ins(Y)$$
$$putdown(X,Y) \leftarrow X \neq Y \otimes wider(Y,X) \otimes isclear(Y) \otimes on.ins(X,Y) \otimes isclear.del(Y)$$

The basic actions are $pickup(X)$ and $putdown(X,Y)$, meaning, "pick up block X" and "put down block X on top of block Y," respectively. Both are defined in terms of elementary inserts and deletes to database relations. The remaining rules combine simple actions into more complex ones. For instance, $move(X,Y)$ means, "move block X to the top of block Y," and $stackTwoBlocks(X,Y,Z)$ means, "stack blocks X and Y on top of block Z." Similarly, $putdown(X,Y)$ means that block X is to be stacked on top of Y, provided that X and Y are not the same and that Y is a wider block. These actions are deterministic: Each set of argument bindings specifies only one robot action.

In contrast, the two actions $stackTwoColors$ and $stackSameColor$ are *non-deterministic*. For instance, $stackTwoColors(C1,C2,Z)$ means, "stack two blocks, of colors C_1 and C_2, on top of block Z." The action does not say which two blocks to use, only their colors. To perform the action, the inference system searches the database for blocks of the appropriate color that can be stacked. If several such blocks are available, the system chooses any two arbitrarily. The action $stackSameColor(Z)$ means, "stack two blocks on top of Z that are of the same color as Z." Again, the inference system searches the database for appropriate blocks. In this way, by defining non-deterministic actions, a user can specify what to do (declarative knowledge) and how to do it (procedural knowledge). □

It is worth noting that the rules in Example 2.13 involve queries as well as updates. In the last rule, for instance, the atom $isclear(Y)$ (which itself may be defined by other deductive rules) is a Boolean test that must return *true* in order for the transaction to succeed. In the first rule, the atom $color(Z,C)$ is a query that retrieves the color C of the block Z. The second rule is, perhaps, the most interesting. Here, the atoms $color(X, C_1)$ and $color(Y, C_2)$ are non-deterministic queries. They retrieve two blocks X and Y of colors C_1 and C_2, respectively. The particular blocks retrieved by these queries then determine the course of action taken in the rest of the transaction.

Just as in Example 2.12, the above $\mathcal{TR}$ program can be straightforwardly converted into a Prolog program and, just as before, this program will not work as intended. Indeed, suppose the robot had picked up a block that is *wider* than any *clear* block currently on the table. Because *putdown* checks that a wider block does not go on top of a smaller one, it will fail. Once again, in $\mathcal{TR}$, the *putdown* action will fail and the changes to the underlying database will be undone. But not so in Prolog, which will leave the robot stranded in an inconsistent internal state.

Example 2.14 (Recursive, Non-Deterministic Actions) Suppose the transaction base of Example 2.13 is augmented by the two rules below. Then the transaction $stack(N, X)$ tries to stack N blocks on top of X. If it succeeds, the database is updated and the transaction returns $true$; if it fails (perhaps because N blocks are not available), then the database remains in its original state, and the transaction returns $false$.

$$\begin{aligned} stack(N, X) &\leftarrow N > 0 \otimes move(Y, X) \otimes stack(N-1, Y) \\ stack(0, X) &\leftarrow \end{aligned}$$

In order to stack N blocks on X, the first rule moves a single block Y onto X, and then recursively tries to stack $N - 1$ blocks on Y. Since the second rule has an empty premise, it terminates the recursion by doing nothing, thereby stacking no blocks on top of X. □

2.6 Constraints

Section 2.3 showed how classical disjunction creates non-determinism. This section shows how classical conjunction constrains it. The connection "$\wedge$" is called "classical conjunction" here because its semantics is defined analogously to the classical AND-connective of predicate calculus. However, our $\wedge$ properly extends the classical $\wedge$ when it comes to updates, and it reduces to the classical connective for formulas that do not update the database.

In general, the transaction $\psi \wedge \phi$ is more deterministic than either ϕ or ψ alone. This is because, by definition (given later), any execution of $\psi \wedge \phi$ must be an allowed execution of ψ *and* an allowed execution of ϕ. We illustrate this idea in two ways: first, through an informal example of robot navigation (*i.e.*, a routing problem), and then through two formal examples. More elaborate examples are given in Section 7.7.

Consider the following conjunction of two robot actions:

"Go to the kitchen" $\wedge$ "Don't pass through the bedroom"

Note that each conjunct is a non-deterministic action, since there are many ways in which it can be carried out. The conjunction, however, is more constrained and deterministic than either of the two original conjuncts alone.

This example illustrates two points about classical conjunction. First, it does not cause the conjuncts to be executed concurrently, as two separate transactions. Instead, it combines them into a single, more tightly constrained transaction. Second, classical conjunction constrains the *entire execution* of a transaction (*i.e.*, the way in which it is carried out), not just the final state of the transaction. It can therefore express *dynamic* integrity constraints. We elaborate on these points in Section 7.7.

In general, the conjunction $\phi \wedge \psi$ can execute iff actions ϕ and ψ have common execution paths. In this case, we say that ϕ and ψ are *non-exclusive*. For example, the actions $a \vee b$ and $b \vee c$ are non-exclusive, since they can both execute on any path on which b can execute. If ϕ and ψ have no common execution paths, then we say they are *mutually exclusive*. For example, the actions $a \wedge b$ and $a \wedge \neg b$ are mutually exclusive, since any path that satisfies one will not satisfy the other. We also speak of actions as being exclusive or non-exclusive with respect to an initial database state. A conjunction of mutually exclusive actions always fails.

Example 2.15 (Mutually Exclusive Actions: I)
If a user issued the transaction $?-\ hired.ins \wedge hired.del$, it would fail,[4] leaving the database unchanged. The transaction fails because the actions *hired.ins* and *hired.del* are mutually exclusive: given our definitions of insert and delete, it is not possible to simultaneously insert and delete one and the same atom into a database. Formally, the following statements are both true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{hired\} \ \models \ hired.ins \qquad \mathbf{P}, \mathbf{D}, \mathbf{D} - \{hired\} \ \models \ hired.del$$

However, there is no database $\mathbf{D'}$ such that *both* of the following are true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D'} \ \models \ hired.ins \qquad \mathbf{P}, \mathbf{D}, \mathbf{D'} \ \models \ hired.del$$

Consequently, the following statement is false for every $\mathbf{D'}$:

$$\mathbf{P}, \mathbf{D}, \mathbf{D'} \ \models \ hired.ins \wedge hired.del$$

□

The above transaction fails because its two conjuncts, *hired.ins* and *hired.del*, terminate at different database states, and are thus mutually exclusive. However, two actions may be mutually exclusive even if they start and terminate at the *same* state. This is because their execution paths may pass through *different* intermediate states. Since classical conjunction forces transactions to execute along the *same path*, it is a powerful tool for expressing constraints on transaction execution and for reducing non-determinism in $\mathcal{TR}$ programs.

Example 2.16 (Mutually Exclusive Actions: II)
Consider the two transactions $?-\ bought.ins \otimes wanted.ins$ and $?-\ wanted.ins \otimes bought.ins$. They both transform a database state $\mathbf{D}$ into the state $\mathbf{D} + \{bought, wanted\}$. However, they pass through different intermediate states. The former goes through the state $\mathbf{D} + \{bought\}$, while the latter passes through the state $\mathbf{D} + \{wanted\}$. The two actions are therefore mutually exclusive, since there is no single sequence of states that is a valid execution path of both conjuncts. The conjunction $(bought.ins \otimes wanted.ins) \wedge (wanted.ins \otimes bought.ins)$ therefore fails. Formally, the following two statements are both true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{bought\}, \mathbf{D} + \{bought, wanted\} \ \models \ bought.ins \otimes wanted.ins$$
$$\mathbf{P}, \mathbf{D}, \mathbf{D} + \{wanted\}, \mathbf{D} + \{bought, wanted\} \ \models \ wanted.ins \otimes bought.ins$$

but the following statement is false:

$$\mathbf{P}, \mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n \ \models \ (bought.ins \otimes wanted.ins) \wedge (wanted.ins \otimes bought.ins)$$

for any sequence of databases $\mathbf{D}, \mathbf{D}_1, \ldots, \mathbf{D}_n$ (assuming *bought* and *wanted* are not in $\mathbf{D}$). □

[4] We use the term "failure" in the same sense as it is used in Prolog.

Example 2.17 (Non-exclusive Actions)
Consider the two transactions $?-\ lost.ins \vee found.ins$ and $?-\ lost.ins \vee won.ins$. They are both non-deterministic. They are also non-exclusive, since starting from database $\mathbf{D}$, they can both follow the same path to terminate at $\mathbf{D} + \{lost\}$. In fact, this is the only database that can be reached by both transactions (assuming that $found, won \notin \mathbf{D}$). Hence, if the user issued the transaction $?-\ (lost.ins \vee found.ins) \wedge (lost.ins \vee won.ins)$ then, after execution, the final database would be $\mathbf{D} + \{lost\}$. Formally, the following statement is true:

$$\mathbf{P}, \mathbf{D}, \mathbf{D}' \models (lost.ins \vee found.ins) \wedge (lost.ins \vee won.ins) \quad \textit{iff} \quad \mathbf{D}' = \mathbf{D} + \{lost\}$$

In this way, classical conjunction reduces non-determinism and, in this particular example, yields a completely deterministic transaction. □

In general, the expression $\phi \wedge \psi$ can be read as, "Execute action ϕ in such a way that constraint ψ is satisfied." Section 7.7 illustrates the constraining effect of different kinds of formulas ψ. Many of these formulas involve serial conjunction and negation. For instance, the following formulas each have a natural interpretation as a constraint:

- $\neg(a \otimes b \otimes c)$ means that the sequence $a \otimes b \otimes c$ must not be executed.
- $a \otimes \neg b$ means that action b must *not* immediately follow action a.
- $\neg(a \otimes \neg b)$ means that action b *must* immediately follow action a.

These formulas can often be simplified by using the dual operator $\oplus$, which we call *serial disjunction*. For example, the last formula can be rewritten as $\neg a \oplus b$. We elaborate on these points in Section 7.7.

Finally, it is worth noting that in addition to specifying constraints, classical conjunction has other important functions. For instance, as in classical logic, it is needed to specify conjunctive queries. Likewise, as in deductive databases, it is needed to construct rule-bases, which are just conjunctions of rules. Of course, unlike deductive databases, rules in $\mathcal{TR}$ define transactions as well as queries. Examples of such rules are given in the next section and in Section 7.

3 Syntax

We define the alphabet of a language of $\mathcal{TR}$ to consist of the following symbols:

- $\mathcal{V}$: a countably-infinite set of variables.
- $\mathcal{F}$: a countably-infinite set of function symbols. Each symbol $f \in \mathcal{F}$ has a non-negative number, its *arity*, indicating the number of arguments f can take. There are infinitely many symbols of each arity. Constants are viewed as 0-ary function symbols.
- $\mathcal{P}$: a countably-infinite set of predicate symbols. Like functions, predicate symbols have arity, and $\mathcal{P}$ has infinitely many predicate symbols of each arity. 0-ary predicate symbols are viewed as propositional constants.
- Logical connectives $\vee$, $\wedge$ (classical disjunction and conjunction), $\oplus$, $\otimes$ (serial disjunction and conjunction), $\neg$ (classical negation). Additional connectives will be defined in terms of these later.

- Quantifiers $\forall$, $\exists$.
- Auxiliary symbols, such as "(", ")", and ",".

Terms are defined as usual in first-order logic: A variable is a term; if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. When $n = 0$, we write f instead of $f(\,)$.

Transaction Formulas

$\mathcal{TR}$ extends the syntax of first-order predicate logic with two new binary connectives, $\otimes$ and $\oplus$, called *serial conjunction* and *serial disjunction*, respectively. The resulting logical formulas are called *transaction formulas.*

Formally, transaction formulas are defined recursively as follows. First, an *atomic* transaction formula is an expression of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ is a predicate symbol, and $t_1, \ldots, t_n$ are terms. Second, if ϕ and ψ are transaction formulas, then so are the following expressions:

- $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \oplus \psi$, $\phi \otimes \psi$, and $\neg\phi$.
- $(\forall X)\phi$ and $(\exists X)\phi$, where X is a variable.

The following are examples of transaction formulas:

- $b(X) \otimes c(X, Y)$
- $a(X) \vee \neg[b(X) \otimes c(X, Y)]$
- $\forall X[a(X) \vee \neg b(X) \oplus \neg c(X, Y)]$

Intuitively, the formula $\psi \otimes \phi$ means, "Do ψ and then do ϕ," and the formula $\psi \oplus \phi$ (very roughly) means, "Do ψ now or do ϕ later." Formally, $\oplus$ and $\otimes$ are defined to be dual, so that $\neg(\psi \oplus \phi)$ is equivalent to $\neg\phi \otimes \neg\psi$. Note that the classical first-order formulas are transaction formulas that do not use $\otimes$ and $\oplus$. As in classical logic, we introduce convenient abbreviations for complex formulas. For instance, $\psi \leftarrow \phi$ is an abbreviation for $\psi \vee \neg\phi$, for any transaction formulas ψ and ϕ.

4 Elementary Operations

Classical logic theories always come with a parameter: a language for constructing well-formed formulas. This language is not fixed, since almost any set of constants, variables and predicate symbols can be "plugged into" it. Likewise, $\mathcal{TR}$ theories are also parameterized by language. In addition, they have another parameter: a pair of oracles, called the *data oracle* and the *transition oracle*, which specify elementary database operations. The data oracle specifies a set of primitive database *queries*, *i.e.*, the *static* semantics of states; and the transition oracle specifies a set of primitive database *updates*, *i.e.*, the *dynamic* semantics of states. These two oracles encapsulate elementary database operations. They also separate the specification of elementary operations from the logic of combining them. As we shall soon see, this separation has two main benefits: (i) it allows us to develop a logic for combining and programming transactions without committing to a particular theory of elementary updates; and (ii) it allows $\mathcal{TR}$ to accommodate a wide variety of database semantics, from classical to non-monotonic to various other non-standard logics. Like the language of the logic, the oracles are not fixed, since almost any pair of oracles can be "plugged into" a $\mathcal{TR}$ theory.

4.1 State Data Oracles

One of the goals underlying the design of $\mathcal{TR}$ is to make it general enough to deal with any kind of database state, including relational databases and more general deductive databases. One may therefore be tempted to define a state as an arbitrary first-order formula and close the issue. However, things turn out to be more involved. For one thing, stating that a database state is a first-order formula does not determine the set of truths about that state. This is because in databases and logic programming, one usually assigns a *non-standard* semantics to such databases, such as Clark's completion, a perfect-model or a well-founded model semantics [67, 96, 39]. Because of this, it is wise to insulate the dynamic aspects of transaction execution from the static aspects pertaining the truth at database states. Not only will this allow Transaction Logic to work with different database semantics, but it will also enable us to study the dynamics in separation from those static aspects.

Another problem is that logically equivalent first-order formulas may represent different database states. For instance, in databases and logic programming, $\{p \longleftarrow \neg q\}$ is viewed as a different state than $\{q \longleftarrow \neg p\}$, even though the two formulas are classically equivalent. In the first database, p is considered as true and q as false; in the second state, it is just the opposite.

To achieve the needed generality, the semantics of states is specified by a *state data oracle*. We assume a countable set of symbols, called *state identifiers*, which the oracles use to refer to database states. Note, these symbols are not part of the language of $\mathcal{TR}$, just as the oracles are not part of it. Indeed, transaction formulas in $\mathcal{TR}$ never address database states or oracles directly. However, state identifiers and oracles are used to define the semantics and the proof theory of $\mathcal{TR}$.

Definition 4.1 (State Data Oracle) A *state data oracle* is a mapping, $\mathcal{O}^d$, from the set of state identifiers defined above to sets of first-order formulas. $\Box$

Intuitively, if i is a state identifier, then $\mathcal{O}^d(i)$ is the set of formulas considered to be all the truths known about the state. In practice, it is not necessary to materialize all these truths. Instead, given a logical formulas ϕ, and a state identifier i, the proof theory for $\mathcal{TR}$ only needs to know whether $\phi \in \mathcal{O}^d(\mathbf{D})$. Thus, to do inference in $\mathcal{TR}$, an enumeration of $\mathcal{O}^d(i)$ is all that is needed.

Since the state id uniquely identifies a state, we shall use the terms "state" and "state id" interchangeably. To connote the right intuition, many examples in this paper use first-order formulas (or sets of first-order formulas) as state identifiers. This is appropriate because many useful oracles can be conveniently described in terms of such formulas. However, our results and definitions do not depend on this representation and, in general, any construct can be a state identifier.

To get a better grasp of the idea of an oracle-as-database-state, Section 4.3 provides a number of examples, which are used frequently in this paper.

4.2 State Transition Oracles

The next step is to specify *elementary* changes to the database. One way to define such changes is to build them into the semantics, as in [69, 77, 16, 28, 5, 72]. The problem with this approach is that adding new kinds of elementary transitions requires redefining the very notion of a model and, hence, entails a revamping of the entire theory, including the need to reprove soundness and completeness results. In other words, such theories are not extensible. This is a serious problem, since the ability to add new transitions is by no means an esoteric whim. In Sections 7.5 and 7.6, for instance, we shall use a new kind of transition called *relational assignment* to perform bulk updates and non-deterministic sampling. In scientific databases, an elementary transition may be a sophisticated numerical operation, such as a Fourier transform.

The problem is aggravated by the fact that, for arbitrary logical databases, the semantics of elementary updates is not clear, not even for relatively simple updates like insert and delete. For example, what does it mean to insert an atom b into a database that entails $\neg b$, especially if $\neg b$ itself is not explicitly present in the database? Or, is insertion of $\{q\}$ into $\{p \longleftarrow \neg q\}$ the same as the insertion into $\{q \longleftarrow \neg p\}$? There is no simple answer to this question, and many solutions have been proposed (see [54] for a comprehensive discussion). Furthermore, Katsuno and Mendelzon [54] pointed out that, generally, state transitions belong to two major categories—*updates* and *revisions*—and, even within each category, several different flavours of such transitions are worth looking at. Thus, there appears to be no small, single set of elementary state transitions that is best for all purposes.

For this reason, rather than committing $\mathcal{TR}$ to a fixed set of elementary transitions, we have chosen to treat elementary state transitions as a *parameter* of $\mathcal{TR}$. Each set of elementary transitions thus gives rise to a different version of the logic. To achieve this, elementary state transitions are specified outside $\mathcal{TR}$, using the notion of a *state transition oracle*. In this way, elementary transitions are separated from the issue of specifying complex transactions. $\mathcal{TR}$ can thus work with any algorithmic or declarative language for specifying elementary transitions.

Definition 4.2 (State Transition Oracle) A *state transition oracle*, $\mathcal{O}^t$, is a mapping from (ordered) pairs of database state identifiers into sets of ground (*i.e.*, variable-free) atomic formulas. We will often refer to these ground atoms as *elementary transitions*.[5] □

Intuitively, if i_1 and i_2 are state identifiers, then $\mathcal{O}^t(i_1, i_2)$ is the set of elementary updates that can transform state i_1 into state i_2. An elementary update can thus be non-deterministic, since for each update, the transition oracle defines a binary relation on states. In practice, this relation does not have to be materialized. Instead, for a given update u, and a given state i_1, the proof theory of $\mathcal{TR}$ only needs an enumeration of the possible successor states, i_2.

Because the transition oracle returns ground atoms only, the elementary updates are completely specified. To see this, suppose that $\mathcal{O}^t(i_1, i_2)$ contained the formula $a \vee b$. Intuitively, this would mean that one of a or b transforms the database from state i_1 to state i_2, but we do not know which one.[6]

Finally, the names of elementary transitions, such as $b.ins$ and $b.del$ in Section 2, have *no* special status in $\mathcal{TR}$. That is, they are ordinary atomic formulas that just happen to be mentioned by the oracle. In principle, nothing prevents the user from putting rules for $b.ins$ into the transaction base.[7] Even the fancy names of these predicates is nothing but a convention adopted for this paper.

4.3 Examples

This section gives examples of data and transition oracles. In each example, a database state is a set of data items, as in the theory of transaction management [12]. Intuitively, a data item can be any persistent object, such as a tuple, a disk page, a file, or a logical formula. Formally, however, a database state has no structure, and our only access to it is through the two oracles. Some of

[5] Groundness of elementary transitions is required for simplicity. We could allow arbitrary closed formulas, but it is not yet clear whether this generality has interesting applications.

[6] We could allow arbitrary closed formulas like $a \vee b$ to be returned by the transition oracle, but this would complicate the theoretical development, and it is not clear what applications this generalization would have.

[7] However, preventing the user from tinkering with the definitions of elementary transitions may be a good practice.

the oracles below can be combined to yield more powerful oracles. Typically, such combinations are possible when oracles operate on disjoint domains of data items.

Relational Oracles: A state $\mathbf{D}$ is a set of ground atomic formulas. The data oracle simply returns all these formulas. Thus, $\mathcal{O}^d(\mathbf{D}) = \mathbf{D}$. Moreover, for each predicate symbol p in $\mathbf{D}$, the transition oracle defines two new predicates, $p.ins$ and $p.del$, representing the insertion and deletion of single atoms, respectively. Formally, $p.ins(\overline{x}) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_2 = \mathbf{D}_1 \cup \{p(\overline{x})\}$. Likewise, $p.del(\overline{x}) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_2 = \mathbf{D}_1 - \{p(\overline{x})\}$. SQL-style bulk updates can also be defined by the transition oracle (Section 7.5), as can primitives for creating new constant symbols.

Scientific Oracles: A state is a set of square matrices. For each matrix, B, in a state, the data oracle defines three ternary relations, b, $b.inv$ and $b.dft$, representing the matrix itself, its inverse, B^{-1}, and its two-dimensional discrete Fourier transform, $dft(B)$, respectively. Formally, $b(i,j,v) \in \mathcal{O}^d(\mathbf{D})$ iff $B(i,j) = v$ in $\mathbf{D}$. Likewise, $b.inv(i,j,v) \in \mathcal{O}^d(\mathbf{D})$ iff $B^{-1}(i,j) = v$; and $b.dft(i,j,v) \in \mathcal{O}^d(\mathbf{D})$ iff $dft(B)(i,j) = v$. In this way, the data oracle provides three built-in views of each matrix.[8] The transition oracle also defines three predicates, $b.set$, $b.rswap$ and $b.cswap$, which update matrix b. The first predicate sets the value of an element of the matrix. Formally, $b.set(i,j,v) \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ iff $\mathbf{D}_1 = \mathbf{D}_2$ except that $B(i,j) = v$ in state $\mathbf{D}_2$. Likewise, $b.rswap(i,j)$ swaps rows i and j of the matrix, while $b.cswap$ swaps columns i and j. Note that for main-memory systems, these updates can be implemented with an efficiency comparable to that of variable assignment, *i.e.*, much more efficiently than assert and retract in Prolog.

Classical Oracles: A state $\mathbf{D}$ is a consistent set of variable-free, classical first-order formulas. The data oracle defines all the logical implications of these formulas. Thus $\mathcal{O}^d(\mathbf{D}) = \{\psi \mid \mathbf{D} \models^c \psi\}$, where $\models^c$ denotes classical entailment. The transition oracle defines primitives for adding and removing formulas from the database, resolving any conflicts between the new formulas and existing formulas. Such conflicts can be resolved in numerous ways, as shown by Katsuno and Mendelzon [54]. For instance, for each first-order formula, μ, the transition oracle could define four predicates, $update[\mu]$, $erase[\mu]$, $revise[\mu]$ and $contract[\mu]$ for doing updates, erasure, revision, and contraction as defined in [54].

Well-Founded Oracle: A state $\mathbf{D}$ is a set of generalized-Horn rules[9] and $\mathcal{O}^d(\mathbf{D})$ is the set of literals in the well-founded model of $\mathbf{D}$ [96]. Such oracles can represent any rule-base with well-founded semantics, which includes Horn rule-bases, stratified rule-bases, and locally-stratified rule-bases. For sophisticated applications, one may want to augment $\mathcal{O}^d(\mathbf{D})$ with the rules in $\mathbf{D}$.

Generalized-Horn Oracles: A state $\mathbf{D}$ is a set of generalized-Horn rules and $\mathcal{O}^d(\mathbf{D})$ is a classical Herbrand model of $\mathbf{D}$. Such oracles can represent Horn rule-bases, stratified rule-bases, locally-stratified rule-bases, rule-bases with stable-model semantics [39], and any rule-base whose meaning is given by a classical Herbrand model. Again, one may want to augment $\mathcal{O}^d(\mathbf{D})$ with the rules in $\mathbf{D}$.

[8] Matrix inversion and discrete Fourier transforms are typically provided as built-in operations by scientific software packages.

[9] Generalized-Horn are rules with possibly negated premises.

4.4 Implementing Oracles

Unlike the formulas in a transaction base, we do not expect the oracles to be coded by casual users. Although the oracles allow for many different semantics for states and state changes, we envision that any logic programming system based on $\mathcal{TR}$ will likely have a carefully selected repertoire of built-in database semantics, and a tightly controlled mechanism for adding new ones. This latter mechanism would not be available to ordinary programmers. For this reason, we assume in this paper that the data and transition oracles are fixed.

Unfortunately, there is no general solution to the practical problem of *how* oracles can best be implemented. For the classical oracle described above, the problem has been partly solved by Grahne, Mendelzon, and Winslett [45, 97]. Winslett showed that, in general, the problem of updating propositional formulas is NP-hard. Subsequently, though, Grahne and Mendelzon proved that updating sets of ground atoms with arbitrary propositional formulas can be done in polynomial time. More importantly, this result carries over to deductive databases, in which only the extensional part is updated. In this case, as with relational databases, updates are fast and straightforward.

For scientific databases, the oracles may involve complex calculations and simulations, such as matrix inversion or numerical integration. Good algorithms for carrying out these operations have been carefully developed by scientists and numerical analysts. These algorithms should be incorporated into the oracles. It is completely impractical to expect a user to specify declaratively a complex numerical operation, and expect an optimizer to find the best way of implementing it. This would require the optimizer to automatically reproduce years of research in combinatorial optimization. Instead, oracles allow us to exploit what combinatorialists have discovered, without having to rediscover it.

By design, the issue of specifying and implementing elementary operations is orthogonal to our work. In $\mathcal{TR}$, the data and transition oracles are external parameters, and all that matters practically is the existence of an algorithm to compute the outcome of an operation, or to enumerate the possible outcomes if the operation is non-deterministic. Finally, we note that transaction definitions are independent of the oracles. This latter point contributes to making $\mathcal{TR}$ a lucid and flexible language for defining transactions.

In closing this section, we note a limitation of oracles: Although they may be viewed as efficient black boxes for *executing* elementary operations, they provide only limited help with *reasoning* about them. In databases and logic programming, however, execution is by far the more important issue. Reasoning can still be performed in $\mathcal{TR}$—it is just that $\mathcal{TR}$ does not offer specific advantages in this sphere. However, many reasoning problems can often be reduced to execution. This is shown in Section 7.7, which illustrates constraints, and in Section 7.9, which illustrates planning. In such cases, $\mathcal{TR}$ does offer decided advantages over other reasoning systems. For instance, as discussed in Section 7.8, the infamous *frame problem* is factored out from $\mathcal{TR}$.

5 Model Theory

The model theory of $\mathcal{TR}$ is based on a few simple ideas. We discuss these ideas, and then develop the model theory in detail.

States. As in modal logic, a semantic structure in $\mathcal{TR}$ is based on a set of states. Beyond this similarity, however, the two logics are very different. For instance, in modal logic, the set of states is not fixed, but can vary from one semantic structure to another. In Transaction Logic, the set of states is the same for all semantic structures: it is the set of states that comes with the data and transition oracles. Changing the oracles can change the set of states, and thus the set of semantic structures. This is one way in which different oracles give rise to different versions of $\mathcal{TR}$.

Transaction Execution Paths. When a transaction is executed, the database may change, going from an initial state, $\mathbf{D}_1$, to a final state, $\mathbf{D}_n$. As illustrated in Section 2, the database may also pass through any number of intermediate states, $\mathbf{D}_2, \ldots, \mathbf{D}_{n-1}$, along the way. The sequence $\langle \mathbf{D}_1\ \mathbf{D}_2\ \ldots\ \mathbf{D}_{n-1}\ \mathbf{D}_n \rangle$ is the *execution path* of the transaction. It is said to have *length n*. The semantics of Transaction Logic is based on execution paths, or *paths* for short. As illustrated in Sections 2.6 and 7.7, paths allow us to model a wide range of constraints on transaction execution. For example, we may require that every intermediate state satisfy some condition, or we may forbid certain sequences of states. Because of the emphasis on paths, semantic structures in $\mathcal{TR}$ are called *path structures*.

Unlike modal structures, truth of formulas in path structures is defined on paths, not states. This is because, as in Process Logic [50], transactions should be able to examine and query intermediate states of their execution. For example, we would say that the path $\langle \{a\}\ \{a,b\}\ \{a,b,c\} \rangle$ satisfies the formula $b.ins \otimes a \otimes c.ins$, since it represents an insertion of b followed by an insertion of c, and the proposition a is true in the intermediate state created by the first update. On the other hand, the path $\langle \{a\}\ \{a,c\}\ \{a,b,c\} \rangle$ does *not* satisfy this formula; instead, it satisfies the formula $c.ins \otimes a \otimes b.ins$. This simple example illustrates a general idea in $\mathcal{TR}$: *the truth of a formula on a path corresponds to the common-sense notion of transaction execution.* If the path has length 1, *i.e.*, if it consists of a single state, then the transaction is a query to that state in the traditional sense; if the path has length 2, then the transaction (usually) is an elementary update; and if the path has length greater than 2, then the transaction is a composite update. In this way, queries, updates, and transactions are all treated uniformly.

To define truth on paths in a completely general way, we assign a first-order semantic structure to each path, π. This structure specifies those atomic formulas that are true on π. The satisfaction of complex formulas on π is determined by the semantic structures assigned to π and its subpaths. Intuitively, all formulas—atomic or complex—that are true on a path represent actions that take place along the path. This reliance on paths is reminiscent of a version of Process Logic defined in [50]. However, there is a vast difference in *how* truth is defined and in *what* formulas actually denote. Further comparison is provided in Section 11.

Stored Facts vs. Derived Facts. In classical database systems, a subtle distinction is made between stored facts and derived facts. Typically, update procedures distinguish between these two types of facts, but query processors do not. For example, a deductive database has a set of base facts $\mathbf{D}$, called the *extensional* database, and a set of rules $\mathbf{P}$, called the *intensional* database. The query processor, however, sees only the logical consequences of $\mathbf{D} \cup \mathbf{P}$. In this way, a user cannot tell whether a fact is stored or derived. In contrast, update-procedures act only on the stored facts $\mathbf{D}$. Even if a user requests an update to a derived fact (a view update), the request is translated into an equivalent set of updates to the stored facts. The reason, of course, is that derived facts cannot be changed without changing the base facts that support them.

To reflect this distinction, the path structures of $\mathcal{TR}$ carefully distinguish between a state $\mathbf{D}$ and the path $\langle \mathbf{D} \rangle$ of length 1. Intuitively, the state $\mathbf{D}$ represents the formulas stored in the database, and the path $\langle \mathbf{D} \rangle$ represents formulas derived by combining the database and the transaction base. That is, in database terminology, $\langle \mathbf{D} \rangle$ is a *view* of $\mathbf{D}$.[10] Only the state $\mathbf{D}$ (the stored data) is updated, and only the path $\langle \mathbf{D} \rangle$ (the view) is queried. The user, therefore, sees the state indirectly, through the view. To capture this idea formally, formulas are *never* evaluated at states (unlike modal logic). Thus, one cannot inquire about the truth of a formula ψ at a state $\mathbf{D}$, since this would be tantamount to querying stored facts directly. However, ψ can be evaluated on the path $\langle \mathbf{D} \rangle$, which is tantamount to querying the view over $\mathbf{D}$, as in the classical theory of deductive databases.

5.1 Path Structures

This section makes the preceding discussion precise. In the definitions below, each path structure has a domain of objects and an interpretation for all function symbols, which are used to interpret formulas on every path in the structure.

In the formal definition of path structures, we rely on the notion of classical first-order semantic structures [31]. For our purposes, it is convenient to augment these classical structures with an additional, *special*, structure, denoted by $\top$. It is defined by the property that it satisfies every first-order formula. Even though $\top$ is not a classical first-order structure, here we shall call it "classical" because adding it to classical logic does not change the logic in any essential way—it simply adds one more model to every formula. The use of $\top$ lets us localize possible inconsistency between database states (specified by oracles) and views over these states (specified by transactions).

Recall that $\mathcal{TR}$ includes a state transition oracle, $\mathcal{O}^t$, and a state data oracle, $\mathcal{O}^d$. The domain of these oracles determines the set of database state identifiers (or states). A *path* of length k (or k-*path*) is a finite sequence of these state identifiers, $\langle \mathbf{D}_1 \ \ldots \ \mathbf{D}_k \rangle$, where $k \geq 1$.

Definition 5.1 (Path Structures) Let $\mathcal{L}$ be a language of $\mathcal{TR}$ with the set of function symbols $\mathcal{F}$. A *path structure*, $\mathbf{M}$, over $\mathcal{L}$ is a triple $\langle U, I_{\mathcal{F}}, I_{path} \rangle$, where:

- U is a set, called the *domain* of $\mathbf{M}$.
- $I_{\mathcal{F}}$ is an interpretation of function symbols in $\mathcal{L}$. It assigns a function $U^n \longmapsto U$ to every n-ary function symbol in $\mathcal{F}$.

  Given U and $I_{\mathcal{F}}$, let $Struct(U, I_{\mathcal{F}})$ denote the set of all classical first-order semantic structures over $\mathcal{L}$ of the form $\langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$, where U is the domain of the structure, $I_{\mathcal{P}}$ is a mapping that interprets predicate symbols in $\mathcal{P}$ by relations on U of appropriate arity, and U and $I_{\mathcal{F}}$ are the same as in $\mathbf{M}$. In accordance with an earlier remark, we also assume that $Struct(U, I_{\mathcal{F}})$ contains the special "classical" structure $\top$.
- I_{path} is a mapping that assigns to every path in $Paths(\mathcal{L})$, a semantic structure in $Struct(U, I_{\mathcal{F}})$ (which may be $\top$ as well). Finally, we subject the mapping I_{path} to two restrictions:
  - *Compliance with the data oracle*: $I_{path}(\langle \mathbf{D} \rangle) \models^c \phi$, for every $\phi \in \mathcal{O}^d(\mathbf{D})$, which means that the view over any database state must be a model of whatever the oracle considers to be true of that state.[11] (As before $\models^c$ denotes classical satisfaction in first-order models.)

[10] Of course, the state $\mathbf{D}$ also represents derived information, but only information that is derivable from the formulas in the database.

[11] Observe that unlike the previous versions of $\mathcal{TR}$[19, 20], $I_{path}(\langle \mathbf{D} \rangle)$ is not required to be a model of $\mathbf{D}$. It will be a model, however, if the oracle is such that $\mathbf{D}$ is logically implied by $\mathcal{O}^d(\mathbf{D})$, such as in the case of the Generalized Horn Oracle.

– *Compliance with the transition oracle*: $I_{path}(\langle \mathbf{D}_1\ \mathbf{D}_2 \rangle) \models^c b$ whenever $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$. $\Box$

The mapping I_{path} specifies what atoms are true on what paths. These atoms denote actions that take place along various paths. In arbitrary path structures, these actions are independent of one another. In particular, the semantic structure assigned to path π is independent of the subpaths of π. Intuitively, this means that we know nothing about the relationship between transactions and their subtransactions. Such knowledge, when it exists, is encoded as transaction formulas, as described in Section 5.3. These formulas constrain the mapping I_{path}, forcing the atoms on one path to be related in precise ways to atoms on other paths.

Path structures are also constrained by the data and transition oracles. The oracles specify elementary database operations which all path structures must model. Definition 5.1 formalizes this idea by imposing two restrictions on path structures. Intuitively,

- restriction 1 says that the data oracle is modeled by 1-paths in a path structure; *i.e.*, the semantic structure assigned to path $\langle \mathbf{D} \rangle$ is a classical model of the formulas stored at $\mathbf{D}$;
- restriction 2 says that the transition oracle is modeled by 2-paths in a path structure; *i.e.*, the semantic structure assigned to path $\langle \mathbf{D}_1\ \mathbf{D}_2 \rangle$ is a classical model of the elementary transitions from $\mathbf{D}_1$ to $\mathbf{D}_2$.

One more point about Definition 5.1 is worth noting. In a path structure, $\mathbf{M}$, the mapping $I_{\mathcal{F}}$ is shared by all classical first-order structures mentioned in $\mathbf{M}$. In modal logics, such semantics is said to be based on the assumption of "rigid designators." If this assumption is not made, the number of models of a formula may increase and so certain valid formulas (with respect to rigid designators) may become non-valid. As in modal logics, the decision to use rigid designators has a profound effect on $\mathcal{TR}$. For instance, when equality is allowed, *non*-rigid designators would lead to unintuitive semantics in the context of $\mathcal{TR}$.

As a simple exercise, the reader can verify the following after reading through the rest of this section. Suppose that $I_{\mathcal{F}}$ is allowed to vary from state to state within a path structure. Suppose also that the transaction base contains a single rule, $p(X) \longleftarrow (X = c) \otimes q.ins(b) \otimes (X = c)$. Intuitively, executing the transaction $?-\ p(c)$ on an empty database should cause the insertion of $q(b)$ into the database. However, this is *not* the case with non-rigid designators. Indeed, the interpretation of c in the state corresponding to the empty database may differ from the interpretation of c in the state corresponding to the database $\{q(b)\}$. In this case, the transaction would fail, and the database would remain empty!

5.2 Satisfaction on Paths

Transaction formulas are evaluated on paths. That is, they are true or false on paths in a structure. This subsection makes this idea precise.

Given a path, $\pi = \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle$, it is convenient to define a *split* of π to be any pair of subpaths, π_1 and π_2, such that $\pi_1 = \langle \mathbf{D}_1 \ldots \mathbf{D}_i \rangle$ and $\pi_2 = \langle \mathbf{D}_i \ldots \mathbf{D}_n \rangle$ for some i ($1 \le i \le n$). In this case, we shall write $\pi = \pi_1 \circ \pi_2$. In other words, $\pi = \pi_1 \circ \pi_2$ if and only if π_1 is a prefix of π, π_2 is a suffix of π, and the last state in π_1 is the first state in π_2.

As in classical logic, in order to define the truth value of quantified formulas and of open formulas, it is convenient to introduce variable assignments. A *variable assignment* is a mapping $\nu : \mathcal{V} \longrightarrow \mathcal{U}$, which takes a variable as input, and returns a Herbrand term as output. We extend the mapping from variables to terms in the usual way, *i.e.*, $\nu(f(t_1, \ldots, t_n)) = f(\nu(t_1), \ldots, \nu(t_n))$.

Definition 5.2 (Satisfaction) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{path} \rangle$ be a path structure, let π be an arbitrary path, and let ν be a variable assignment. Then:

1. **Base Case:**
   $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ if and only if $I_{path}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, for any atomic formula $p(t_1, \ldots, t_n)$, where "$\models^c_\nu$" denotes classical satisfaction in first-order predicate calculus.

2. **Negation:**
   $\mathbf{M}, \pi \models_\nu \neg\phi$ if and only if it is not the case that $\mathbf{M}, \pi \models_\nu \phi$.

3. **Classical Conjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \wedge \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ and $\mathbf{M}, \pi \models_\nu \psi$.

4. **Classical Disjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \vee \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ or $\mathbf{M}, \pi \models_\nu \psi$.

5. **Serial Conjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ and $\mathbf{M}, \pi_2 \models_\nu \psi$ for *some* split $\pi_1 \circ \pi_2$ of path π.

6. **Serial Disjunction:**
   $\mathbf{M}, \pi \models_\nu \phi \oplus \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ or $\mathbf{M}, \pi_2 \models_\nu \psi$ for *every* split $\pi_1 \circ \pi_2$ of path π.

7. **Universal Quantification:**
   $\mathbf{M}, \pi \models_\nu (\forall X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X.

8. **Existential Quantification:**
   $\mathbf{M}, \pi \models_\nu (\exists X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *some* variable assignment μ that agrees with ν everywhere except on X.

□

As in classical logic, the mention of variable assignment can be omitted for *sentences*, *i.e.*, for formulas with no free variables. From now on, we will deal only with sentences, unless explicitly stated otherwise. If $\mathbf{M}, \pi \models \phi$, then we say that ϕ is *satisfied* (or is *true*) on path π in structure $\mathbf{M}$.

Several points about Definition 5.2 are worth mentioning:

- The base case captures the intuition behind transaction execution along a path: the truth of an atom $p(t_1, \ldots, t_n)$ on a path π means that transaction p *can* execute along π when invoked with arguments $t_1, \ldots, t_n$.

- The semantics of all "classical" connectives ($\wedge$, $\neg$, etc.) is defined in a classical fashion. For these connectives, truth on a path depends only on the classical structure assigned to the path. The classical nature of these connectives is made precise by Lemma 5.3 below.

- For the non-classical connective $\otimes$, truth on a path depends not just on the path, but on its *subpaths* as well. In this way, actions on different paths can be made to depend on each other. Intuitively, the semantics of $\otimes$ says that a serial transaction succeeds if and only if its subtransactions all succeed. This idea is illustrated in Sections 2.5, 5.3 and 6.3.

- For paths of length 1, the connectives $\otimes$ and $\wedge$ are equivalent, as shown in Lemma 5.4 below. It follows that on 1-paths, every occurrence of $\otimes$ in a transaction formula can be replaced by $\wedge$ to give an equivalent first-order formula. By Lemma 5.3, this formula has a classical semantics. Thus, on paths of length 1, $\otimes$ and $\wedge$ reduce to ordinary conjunction in classical predicate logic. On longer paths, these two connectives extend the classical semantics of conjunction in two different ways.

Lemma 5.3 *If ϕ is a first-order formula, then $\mathbf{M}, \pi \models \phi$ if and only if $I_{path}(\pi) \models^c \phi$, for any path structure M and any path π, where $\models^c$ denotes classical entailment.*

Proof: Follows by a straightforward induction on the structure of ϕ. $\Box$

Lemma 5.4 *If α and β are two transaction formulas, then $\mathbf{M}, \langle \mathbf{D} \rangle \models \alpha \otimes \beta$ if and only if $\mathbf{M}, \langle \mathbf{D} \rangle \models \alpha \wedge \beta$, for any path structure $\mathbf{M}$ and any state $\mathbf{D}$.*

Proof: Follows from the observation that $\langle \mathbf{D} \rangle \circ \langle \mathbf{D} \rangle$ is the only split of path $\langle \mathbf{D} \rangle$. $\Box$

The reader may have noticed that although the definitions of path structures and satisfaction invoke the classical predicate calculus when it comes to the logic of individual states, we did not rely on the actual properties of that calculus. In fact, $\mathcal{TR}$ relies only on the existence of an entailment relation, such as $\models^c$, that can be used by the state data oracle. Indeed, [55] shows how a different logic of states (F-logic [56]) can be used as a simple drop-in replacement for predicate calculus within the framework of $\mathcal{TR}$.

Abbreviations

Before continuing, we define a few useful abbreviations. As usual in first-order logic, we define $\psi \leftarrow \phi$ and $\phi \rightarrow \psi$ to mean $\psi \vee \neg\phi$, and $\psi \leftrightarrow \phi$ to mean $(\psi \leftarrow \phi) \wedge (\psi \rightarrow \phi)$. By replacing $\vee$ and $\wedge$ with $\otimes$ and $\oplus$ we obtain another interesting pair of connectives: the *left serial implication*, $\psi \Leftarrow \phi$, standing for $\psi \oplus \neg\phi$, and the *right* serial implication, $\phi \Rightarrow \psi$, standing for $\neg\phi \oplus \psi$. As follows from Definition 5.2, in plain English these formulas say the following: "action ϕ must be immediately preceded (resp., followed) by action ψ." Note that unlike "$\leftarrow$" and "$\rightarrow$", the connectives "$\Leftarrow$" and "$\Rightarrow$" are not just syntactic variations of each other, *i.e.*, $\psi \Leftarrow \phi$ is not equivalent to $\phi \Rightarrow \psi$; rather, $\psi \Leftarrow \phi$ is equivalent to $\neg\psi \Rightarrow \neg\phi$. Section 7.7 discusses applications of these connectives. Finally, we introduce two more abbreviations, for iterating the operators $\otimes$ and $\oplus$:

$$\otimes^n \psi \quad \equiv \quad \underbrace{\psi \otimes \cdots \otimes \psi}_{n} \qquad\qquad \oplus^n \psi \quad \equiv \quad \underbrace{\psi \oplus \cdots \oplus \psi}_{n}$$

In addition to these abbreviations, we assume that our language contains a predefined propositional constant, `state`, which is true only on paths of length 1, that is, on database states. Formally, for any path structure $\mathbf{M}$ and path π, it is the case that $\mathbf{M}, \pi \models \texttt{state}$ if and only if π is a path of length 1. Thus, `state` is false on every path having more than one state, even on loops like $\langle \mathbf{D}\ \mathbf{D} \rangle$. (Such loops can arise if, for instance, a transaction inserts an atom into a database when the atom is already there. In such cases, the initial and final states of the database are the same.) Another useful constant is `path`, defined as $\texttt{state} \vee \neg\texttt{state}$, which is true on every path in $\mathbf{M}$. In fact, `path` is equivalent to $\phi \vee \neg\phi$ for any formula ϕ, and thus `path` does not increase the expressive power of the logic. In contrast, $\mathtt{state}$ cannot be independently defined in $\mathcal{TR}$, and so it *does* increase the expressive power of the logic.

(In Section 8.1 we shall see that $\mathtt{state}$ can be expressed using $\mathtt{path}$ and the hypothetical operators. To see that $\mathtt{state}$ cannot be expressed using the part of $\mathcal{TR}$ introduced so far, suppose there is a formula ϕ such that $\phi \equiv \mathtt{state}$. Let $\mathbf{M}$ be a path structure, and let $\mathbf{D}$ be a state in $\mathbf{M}$. Choose I_{path} so that $I_{path}(\langle \mathbf{D}\ \mathbf{D} \rangle) = I_{path}(\langle \mathbf{D} \rangle)$. It is easy to verify by structural induction that $\mathbf{M}, \langle \mathbf{D}\ \mathbf{D} \rangle \models \psi$ if and only if $\mathbf{M}, \langle \mathbf{D} \rangle \models \psi$, for any formula ψ. But $\mathbf{M}, \langle \mathbf{D} \rangle \models \phi$, since ϕ is true on *all* states, and thus $\mathbf{M}, \langle \mathbf{D}\ \mathbf{D} \rangle \models \phi$. But this contradicts the assumption that ϕ is true *only* on states.)

It might seem that $\mathtt{state}$ gives a special status to paths of length 1, since $\mathcal{TR}$ can now distinguish them from all other paths. We shall soon see, however, that one can use $\mathtt{state}$ to specify paths of *any* given length. Thus, no particular path length is special.

Observations

Armed with the above definitions, it is easy to verify that the following formulas analogous to De Morgan's laws are *tautologies*, *i.e.*, they are true on every path in every path structure.

$$\begin{array}{rcl} \neg(\phi \oplus \psi) & \leftrightarrow & \neg\phi \otimes \neg\psi \\ \neg(\phi \otimes \psi) & \leftrightarrow & \neg\phi \oplus \neg\psi \\ (\phi \vee \psi) \otimes \eta & \leftrightarrow & (\phi \otimes \eta) \vee (\psi \otimes \eta) \\ (\phi \wedge \psi) \oplus \eta & \leftrightarrow & (\phi \oplus \eta) \wedge (\psi \oplus \eta) \\ (\phi \wedge \psi) \otimes \eta & \rightarrow & (\phi \otimes \eta) \wedge (\psi \otimes \eta) \\ (\phi \vee \psi) \oplus \eta & \leftarrow & (\phi \oplus \eta) \vee (\psi \oplus \eta) \end{array} \tag{2}$$

The first pair of laws establishes duality between the serial connectives $\otimes$ and $\oplus$. The standard duality between $\vee$ and $\wedge$ also holds. The last pair of implications is unidirectional, which means that $\otimes$, $\oplus$ do not fully distribute through $\wedge$ and $\vee$, respectively. Likewise, $\oplus$ and $\otimes$ do not distribute through each other and $\vee$, $\wedge$ do not distribute through $\oplus$ and $\otimes$. The standard De Morgan's laws involving $\vee$, $\wedge$, $\neg$, and quantifiers *do* hold, however. In particular, this implies that every $\mathcal{TR}$-formula has prenex normal form, but not necessarily conjunctive or disjunctive normal forms.

Here are tautologies involving the propositional constant $\mathtt{state}$, which is true only on paths of length 1:

$$\begin{array}{l} \phi \leftrightarrow (\phi \otimes \mathtt{state}) \\ \phi \leftrightarrow (\mathtt{state} \otimes \phi) \\ \phi \leftrightarrow (\phi \Leftarrow \mathtt{state}) \\ \phi \leftrightarrow (\mathtt{state} \Rightarrow \phi) \end{array}$$

Another interesting observation is that $\mathtt{state}$ enables the construction of formulas that are true on paths of any given length. To do this, we first observe that the formula $\neg\mathtt{state}$ is true on paths of length greater than 1, that is, on paths with at least one arc. The formula $\neg\mathtt{state} \otimes \neg\mathtt{state}$ is therefore true on paths with at least two arcs. More generally, the formula $\otimes^n \neg\mathtt{state}$ is true on paths with at least n arcs. Finally, we observe that the negation of this formula has a particularly simple form:

$$\neg \otimes^n \neg\mathtt{state} \quad \equiv \quad \oplus^n \mathtt{state}$$

This formula is true on paths with fewer than n arcs, *i.e.*, fewer than $n+1$ states. Thus, their length is at most n. We can now define the proposition `n-path`, which is true on paths of length *exactly* n:

$$\texttt{n-path} \;\equiv\; \oplus^n \texttt{state} \wedge \neg \oplus^{n-1} \texttt{state}$$

As a special case, we define the proposition `arc` to be equivalent to `2-path`. That is,

$$\texttt{arc} \;\equiv\; (\texttt{state} \oplus \texttt{state}) \wedge \neg \texttt{state}$$

This proposition, which is true on arcs, is analogous to the familiar temporal operator `next`. By iterating `arc`, we obtain an alternate definition of `n-path`, namely, $\texttt{n-path} \equiv \otimes^{n-1}\texttt{arc}$.

5.3 Models

Path structures assign a first-order semantic structure, $I_{path}(\pi)$, to each path, π. For arbitrary path structures, this assignment is completely arbitrary. Intuitively, this represents a situation in which we have no knowledge about actions: each ground atom represents an action, but we do not know on which paths this action takes place. When it exists, knowledge about actions is expressed as logical formulas. In this case, we are not interested in *arbitrary* path structures, but in *models* of the logical formulas. In these models, the semantic structures assigned by I_{path} are *not* arbitrary, but satisfy constraints and dependencies that are implicit in transaction formulas, which constrain the semantic structures on all paths, and can create dependencies between the semantic structures on a path and its subpaths.

Note that the definition of path structures guarantees that they come with certain amount of structure. For example, if u, v are elementary transitions such that $u \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ and $v \in \mathcal{O}^t(\mathbf{D}_2, \mathbf{D}_3)$, where $\mathcal{O}^t$ is the state transition oracle, then the atom u is true on the path $\langle \mathbf{D}_1\ \mathbf{D}_2 \rangle$, and the atom v is true on the path $\langle \mathbf{D}_2\ \mathbf{D}_3 \rangle$. This bare-bones structure has certain logical consequences. For instance, by Item 5 of Definition 5.2, the formula $u \otimes v$ is true on the path $\langle \mathbf{D}_1\ \mathbf{D}_2\ \mathbf{D}_3 \rangle$. However, since each elementary transition refers to a specific arc, a transition oracle cannot impose constraints on arbitrary paths or between paths. This is the job of the transaction base, which is a set of arbitrary transaction formulas.

Definition 5.5 (Models of Transaction Formulas) A path structure $\mathbf{M}$ is a *path-model* (or just *model*) of a $\mathcal{TR}$-formula ϕ if and only if $\mathbf{M}, \pi \models \phi$ for every path π. In this case, we write $\mathbf{M} \models \phi$. A path structure is a model of a set of formulas if it is a model of every formula in the set. □

In the models of a transaction formula, dependencies may exist between the classical semantic structures assigned to a path and its subpaths. To see this, consider a simple transaction formula $p \leftarrow q \otimes r$, where p, q, and r are ground atomic formulas. Let $\langle U, I_{\mathcal{F}}, I_{path} \rangle$ be a model of this formula, and let π be any path. If $\pi_1 \circ \pi_2$ is a split of π, then I_{path} exhibits a dependency between three different semantic structures on three different paths, π, π_1, and π_2:

$$\textit{if} \quad I_{path}(\pi_1) \models^c q \quad \textit{and} \quad I_{path}(\pi_2) \models^c r \quad \textit{then} \quad I_{path}(\pi) \models^c p$$

That is, if q is true on π_1, and r is true on π_2, then $q \otimes r$ is true on π (by Definition 5.2), so p must also be true on π. Informally, if $q \otimes r$ can execute along path π, then so can p. In a logic-programming setting, this means that p is a *name* of the procedure $q \otimes r$.

5.4 Execution as Entailment

We are now ready to define *executional entailment*, a concept that provides a logical account of transaction execution. Intuitively, execution corresponds to truth on a path. A simplified form of this notion was used in Section 2.

A $\mathcal{TR}$ logic program consists of two distinct parts: the transaction base $\mathbf{P}$, and the initial database state $\mathbf{D}$. Each of these parts plays a distinct role in defining executional entailment. Of these two parts, only the database state is updatable. The state (or, more precisely, what the data oracle reveals about it) consists entirely of classical first-order formulas, which represent a view of the actual contents of the state. In practice, certain kinds of database states will arise more frequently than others. The most common databases will be sets of ground atoms (relational databases), function-free Horn clauses (deductive databases), general Horn clauses (logic programs), and disjunctive and existentially quantified formulas (indefinite databases). In contrast, the transaction base contains transaction formulas that define procedures that update the database. As such, it will normally be composed of formulas containing the serial connectives $\otimes$ or $\oplus$, though classical first-order formulas are also allowed.

Definition 5.6 (Executional Entailment) Let $\mathbf{P}$ be a transaction base. Let ϕ be a transaction formula, and $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ be a sequence of database state identifiers. Then, the statement

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \tag{3}$$

is true if and only if $\mathbf{M}, \langle \mathbf{D}_0\, \mathbf{D}_1\, \ldots\, \mathbf{D}_n \rangle \models \phi$ for every model $\mathbf{M}$ of $\mathbf{P}$.

Related to this are the following statements:

$$\mathbf{P}, \mathbf{D}_0 \text{---} \models \phi \tag{4}$$

$$\mathbf{P}, \mathbf{D}_0 \text{---} \mathbf{D}_n \models \phi \tag{5}$$

$$\mathbf{P}, \text{---} \mathbf{D}_n \models \phi \tag{6}$$

which are true if and only if there is a sequence of databases $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ such that Statement (3) is true. □

Intuitively, Statement (3) says that a successful execution of transaction ϕ can change the database from state $\mathbf{D}_0$ to $\mathbf{D}_1 \ldots$ to $\mathbf{D}_n$. Formally, it means that in every model of $\mathbf{P}$, the path $\mathbf{D}_0\, \mathbf{D}_1 \ldots \mathbf{D}_n$ satisfies formula ϕ. The statement is read as follows: "Under the transaction base $\mathbf{P}$, transaction ϕ may transform database $\mathbf{D}_0$ into database $\mathbf{D}_n$ by passing through intermediate states $\mathbf{D}_1, \ldots, \mathbf{D}_{n-1}$."

Normally, users issuing transactions know only the initial database state $\mathbf{D}_0$, so defining transaction execution via (3) is not always appropriate. To account for this situation, the version of entailment in (4) allows us to omit the intermediate and the final database states. Informally, Statement (4) says that transaction ϕ can execute successfully starting from database $\mathbf{D}_0$. Formally, this statement is read as follows: "Under the transaction base $\mathbf{P}$, the transaction ϕ succeeds from database $\mathbf{D}$." When the context is clear, we simply say that transaction ϕ *succeeds*. Likewise, when statement (4) is not true, we say that transaction ϕ *fails*. In Section 6, we present an inference system that allows us to actually *find* a database sequence $\mathbf{D}_1, \ldots, \mathbf{D}_n$ (in fact, to enumerate all sequences) that satisfy Statement (3) whenever a transaction succeeds.

Statement (6) is the dual of (4). Intuitively, it says that ϕ can execute successfully, *terminating* at state $\mathbf{D}_n$. It is discussed more fully in Section 6.4. Statement (5) is also a useful abbreviation that will be used throughout the paper. Informally, it says that ϕ can execute successfully starting at state $\mathbf{D}_0$ and ending at state $\mathbf{D}_n$.

The following lemma lists some straightforward consequences of Definition 5.6.

Lemma 5.7 (Basic Properties of Executional Entailment) *For any transaction base* $\mathbf{P}$*, any sequence of database states* $\mathbf{D}_0, \ldots, \mathbf{D}_n$*, and any closed transaction formulas* α *and* β*, the following statements are all true:*

1. *If* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$ *and* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \wedge \beta$.
2. *If* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_i \models \alpha$ *and* $\mathbf{P}, \mathbf{D}_i, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \otimes \beta$.
3. *If* $\alpha \leftarrow \beta$ *is in* $\mathbf{P}$ *and* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$.
4. *If* $\mathcal{O}^t(\mathbf{D}_0, \mathbf{D}_1) \models^c \alpha$ *then* $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha$.
5. *If* $\mathcal{O}^d(\mathbf{D}_0) \models^c \alpha$ *then* $\mathbf{P}, \mathbf{D}_0 \models \alpha$.

In the last two items, α *is a first-order formula and* $\models^c$ *denotes classical entailment.*

Note that assertions in Lemma 5.7 deal with inference of two kinds of true statements. On the one hand, items 4 and 5 infer truth directly from the transition oracle and the data oracle. Specifically, item 4 deals with elementary updates, and item 5 handles database queries. On the other hand, items 1, 2, and 3 combine existing entailments to infer new truths. Specifically, item 1 infers classical conjunctions; item 2 infers serial conjunctions; and item 3 infers defined transactions. Items 2–5 anticipate the proof procedures given in Section 6, and indeed, they form the model-theoretic basis of these procedures. Item 1 is the basis for a wide class of dynamic constraints, such as those in Section 7.7.

In Lemma 5.7, the case $n = 0$ corresponds to transactions that do not affect the database, *i.e.*, that act as queries. In this case, classical and serial conjunction are identical, as shown earlier in Lemma 5.4. This is reflected in items 1 and 2 above, which collapse to the same form when $n = 0$:

1b. *If* $\mathbf{P}, \mathbf{D} \models \alpha$ *and* $\mathbf{P}, \mathbf{D} \models \beta$ *then* $\mathbf{P}, \mathbf{D} \models \alpha \wedge \beta$.

2b. *If* $\mathbf{P}, \mathbf{D} \models \alpha$ *and* $\mathbf{P}, \mathbf{D} \models \beta$ *then* $\mathbf{P}, \mathbf{D} \models \alpha \otimes \beta$.

Intuitively, this means that the result of evaluating a conjunctive query is the same whether the conjuncts are evaluated sequentially or concurrently.

5.5 Examples

This section gives several examples of entailment in $\mathcal{TR}$. Each entailment is derived using Lemma 5.7. The examples themselves are based on Example 2.13, in which a robot simulator moves blocks around a table-top. In fact, we assume that the transaction base $\mathbf{P}$ contains the following four rules, adapted from that example:

$$
\begin{aligned}
&stackTwoBlocks(X,Y,Z) \leftarrow move(Y,Z) \otimes move(X,Y) \\
&move(X,Y) \leftarrow pickup(X) \otimes putdown(X,Y) \\
&pickup(X) \leftarrow isclear(X) \otimes on(X,Y) \otimes on.del(X,Y) \otimes isclear.ins(Y) \\
&putdown(X,Y) \leftarrow X \neq Y \otimes isclear(Y) \otimes on.ins(X,Y) \otimes isclear.del(Y)
\end{aligned}
$$

For simplicity, our databases are relational, *i.e.*, sets of ground atoms (which is the case for the majority of applications, including most "blocks-world" examples), and the oracle is also relational (defined in Section 5. We also assume that the transition oracle, $\mathcal{O}^t$, gives the intended meaning to atoms like *p.ins* and *p.del*. That is, for any ground atom p, $p.ins \in \mathcal{O}^t(\mathbf{D}, \mathbf{D}')$ if and only if $\mathbf{D}' = \mathbf{D} + \{p\}$; and similarly for *p.del*. In particular,

$$\begin{array}{rcl} on.del(a,b) & \in & \mathcal{O}^t(\{on(a,b), on(b,c), isclear(a)\},\ \{on(b,c), isclear(a)\}) \\ isclear.ins(b) & \in & \mathcal{O}^t(\{on(b,c), isclear(a)\},\ \{on(b,c), isclear(a), isclear(b)\}) \\ isclear.del(a) & \in & \mathcal{O}^t\{on(b,c), isclear(a), isclear(b)\},\ \{on(b,c), isclear(b)\} \end{array}$$

for any blocks a, b, and c.

Finally, all actions in this section start from the following initial database, representing an arrangement of three blocks where *blkA* is on top of *blkC*, which in turn is on *blkD*, and *blkB* stands by itself:

$$\mathbf{D}_0 = \{isclear(blkA), isclear(blkB), on(blkA, blkC), on(blkC, blkD)\}$$

Example 5.8 (Picking up a block) Starting from the initial database, $\mathbf{D}_0$, we infer that the robot can pick up *blkA*. During the pickup action, the database passes through an intermediate state, $\mathbf{D}_1$, and ends up at state $\mathbf{D}_2$, where

$$\begin{array}{rcl} \mathbf{D}_1 & = & \{isclear(blkA), isclear(blkB), on(blkC, blkD)\} \\ \mathbf{D}_2 & = & \{isclear(blkA), isclear(blkB), isclear(blkC), on(blkC, blkD)\} \end{array}$$

Here is the sequence of inferences. The final inference, line 8, states that the action *pickup*(*blkA*) successfully takes the database from state $\mathbf{D}_0$ to state $\mathbf{D}_2$ via state $\mathbf{D}_1$.

1. $\mathbf{P}, \mathbf{D}_0 \models isclear(blkA)$, by item 5 of Lemma 5.7.
2. $\mathbf{P}, \mathbf{D}_0 \models on(blkA, blkC)$, by item 5 of Lemma 5.7.
3. $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models on.del(blkA, blkC)$, by item 4 of Lemma 5.7.
4. $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \models isclear.ins(blkC)$, by item 4 of Lemma 5.7.
5. $\mathbf{P}, \mathbf{D}_0 \models isclear(blkA) \otimes on(blkA, blkC)$, by lines 1 and 2, and item 2 of Lemma 5.7.
6. $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models isclear(blkA) \otimes on(blkA, blkC) \otimes on.del(blkA, blkC)$, by lines 5 and 3, and item 2 of Lemma 5.7.
7. $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \models isclear(blkA) \otimes on(blkA, blkC) \otimes on.del(blkA, blkC) \otimes isclear.ins(blkC)$, by lines 6 and 4, and item 2 of Lemma 5.7.
8. $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \models pickup(blkA)$, by line 7, the rule for *pickup*, and item 3 of Lemma 5.7.

□

Example 5.9 (Moving a block) Starting from the initial state $\mathbf{D}_0$, we infer that the robot can move *blkA* from *blkC* to *blkB*. In the process, the database passes through three intermediate states, $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{D}_3$, and ends up at state $\mathbf{D}_4$. Here,

$$\mathbf{D}_3 = \{isclear(blkA), isclear(blkB), isclear(blkC), on(blkA, blkB), on(blkC, blkD)\}$$
$$\mathbf{D}_4 = \{isclear(blkA), isclear(blkC), on(blkA, blkB), on(blkC, blkD)\}$$

and states $\mathbf{D}_0$, $\mathbf{D}_1$, and $\mathbf{D}_2$ are as before. First, from Example 5.8, we have the following:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \models pickup(blkA)$$

In a similar fashion, it is not hard to show that

$$\mathbf{P}, \mathbf{D}_2, \mathbf{D}_3, \mathbf{D}_4 \models putdown(blkA, blkB)$$

Thus, by item 2 of Lemma 5.7, we get:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3, \mathbf{D}_4 \models pickup(blkA) \otimes putdown(blkA, blkB)$$

Finally, by the rule for *move* and by item 2 of Lemma 5.7, we get:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3, \mathbf{D}_4 \models move(blkA, blkB)$$

and thus also $\mathbf{P}, \mathbf{D}_0 \text{---} \models move(blkA, blkB)$ and $\mathbf{P}, \mathbf{D}_0 \text{---} \mathbf{D}_4 \models move(blkA, blkB)$. □

Example 5.10 (Stacking blocks) By following the method of the previous two examples, we can build a stack of three blocks, $blkC$, $blkA$ and $blkB$. In particular, we can infer the following statement:

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \mathbf{D}_3, \mathbf{D}_4, \mathbf{D}_5, \mathbf{D}_6, \mathbf{D}_7, \mathbf{D}_8 \models stackTwoBlocks(blkC, blkA, blkB)$$

where $\mathbf{D}_0, \ldots, \mathbf{D}_4$ are as before, the state $\mathbf{D}_8$ is

$$\{isclear(blkC), on(blkC, blkA), on(blkA, blkB), on(blkC, blkD)\}$$

and $\mathbf{D}_5 \ldots \mathbf{D}_7$ are left to the reader. As in Example 5.9, each *move operation* causes four state transitions. Thus, during the stacking operation, the database passes through seven intermediate states, $\mathbf{D}_1$–$\mathbf{D}_7$, to end at state $\mathbf{D}_8$. □

The last example suggests that inference may require keeping track of a long history of database states. Fortunately, for a large and important class of transactions, this is not necessary. In fact, the proof procedures presented in Section 6 record only the current database state, $\mathbf{D}_i$, advancing it from $\mathbf{D}_0$ to $\mathbf{D}_1$ ... to $\mathbf{D}_n$ as inference proceeds.

5.6 Transaction Answers

So far, we have defined what it means for a formula to be true, which allows us to perform transactions that return yes/no answers. In databases, however, we often want queries and transactions to return a *set* of answers (*e.g.*, all employees who earn more than $100,000).

For queries, the solution is straightforward: As in classical logic, we can define a *query* to be a first-order formula $\phi(X_0, \ldots, X_k)$ with free variables $X_0, \ldots, X_k$. The answer to this query is the set of variable bindings that makes the formula true. That is, the answer is the following set of tuples:

$$\{\langle c_1, \ldots, c_k \rangle \mid \mathbf{P}, \mathbf{D} \models \phi(c_1, \ldots, c_k)\}$$

where each c_i is a ground term.

For transactions, the situation is more complex. Because transactions are non-deterministic, the answer can depend on non-deterministic choices made during transaction execution. For instance, continuing Example 2.13, suppose we want our robot to build stacks of blocks, and to return (as an answer) the names of all blocks whose tops are clear *after* the stacks are built. We do not care what blocks are used to build the stacks, so long as these stacks satisfy certain criteria (such as height and color). In this case, the particular blocks affected by the stacking operation is non-deterministic (and is determined by the system at run time), so the answer returned by the transaction is non-deterministic, too.

To make this example concrete, we define a specific stacking operation by adding the following rule to the transaction base:

$$stack(X) \leftarrow move(Y, X) \otimes move(Z, Y)$$

The procedure $stack(X)$ stacks two blocks on top of block X. First, it chooses a block, Y, and puts it on X; then it chooses another block, Z, and puts it on top of Y. The final state of the database is non-deterministic since the choices for the two blocks, Y and Z, are non-deterministic.

Having defined this stacking operation, we ask the robot to stack two blocks on top of $blkA$, and to return the set of blocks that are clear after the operation is complete. This transaction is expressed by the following formula:

$$stack(blkA) \otimes isclear(X) \tag{7}$$

That is, first execute $stack(blkA)$, and then retrieve those values of X that make $isclear(X)$ true. Note that the retrieved values depend on the *final* database state, which is non-deterministic.

The answer to a transaction may depend on the *initial* state of the database as well as the final state. For example, we might ask for those blocks X that are clear *both* before and after the stacking operation. This transaction is expressed by the formula

$$isclear(X) \otimes stack(blkA) \otimes isclear(X)$$

that retrieves the values of X that make $isclear(X)$ true in both the initial and the final states.

More generally, the answer to a transaction may depend on the *intermediate* database states. For example, we might ask the robot to build two stacks, one after the other, and to return those blocks that are clear in between the two stacking operations. This transaction is expressed by the following formula:

$$stack(blkA) \otimes isclear(X) \otimes stack(blkB)$$

This formula commands the robot to stack two blocks on $blkA$, then to retrieve those blocks X that are clear, and then to stack two blocks on $blkB$. Note that here the answers may be different for different execution paths, even if the initial and the final states of these paths are the same.

Thus, unlike database queries, the answer returned by a transaction is not determined by the current database alone. Instead, it may depend on the entire execution path of the transaction. Definition 5.11, below, captures this idea. This simple definition is sufficient for our present purposes. However, as with database queries [26], this definition could be augmented with additional requirements, such as computability and genericity.

Definition 5.11 (Abstract Transactions) An *abstract transaction* is a total mapping from *sequences* of database states to sets of tuples of ground terms. □

Note that database queries are captured by a special case of this definition, the case of sequences of length 1. This is appropriate since queries are a special kind of transaction.

In $\mathcal{TR}$, transactions are specified by a transaction base $\mathbf{P}$ and a transaction formula $\psi(X_1, \ldots, X_k)$ with free variables $X_1, \ldots, X_k$. Specifically, these three elements define an abstract transaction Ψ, where

$$\Psi(\mathbf{D}_0, \ldots, \mathbf{D}_n) \;=\; \{\langle c_1, \ldots, c_k\rangle \;|\; \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \;\models\; \psi(c_1, \ldots, c_k)\} \tag{8}$$

where each c_i is a ground term, and each $\mathbf{D}_i$ is a database (a first-order formula).

It follows immediately from Definition 5.11 that when applied to an initial database, $\mathbf{D}_0$, the answer returned by a transaction is non-deterministic. Indeed, from $\mathbf{D}_0$, the transaction may execute along any one of many possible paths, each returning a different answer. This idea is captured by the following definition.

Definition 5.12 (Transaction Answers) Let Ψ be an abstract transaction, and let $\mathbf{D}_0, \ldots, \mathbf{D}_n$ be a sequence of databases. Then $\Psi(\mathbf{D}_0, \ldots, \mathbf{D}_n)$ is a possible answer to Ψ at state $\mathbf{D}_0$. □

Of course, for many sequences of databases, the set (8) will be empty. The transaction does not execute along these paths. It only executes along a path that returns a non-empty answer. If the answer set is empty for every path, then the transaction fails.

5.7 Discussion

Entailment in $\mathcal{TR}$ has certain subtle properties that may not be readily seen from the examples so far. Those examples focus on the special case in which the Relational Oracle is used and rule-bodies of program rules are sequences of atomic actions. More generally, database states may be more complex, involving arbitrary kinds of oracles, and the transaction base may be an arbitrary transaction formula. Furthermore, the transition oracle may mention only a fraction of all possible database states. In these cases, entailment in $\mathcal{TR}$ can display subtle and important properties. This section discusses some of them.

Classical Conjunction

In Definition 5.6, there may be more than one sequence $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ that starts at $\mathbf{D}_0$. In this case, transactions can succeed in many different ways. This property has two consequences worth mentioning, both concerning classical conjunction.

First, although two transactions ϕ and ψ may succeed individually, their conjunction $\phi \wedge \psi$ may fail, as illustrated in Examples 2.15 and 2.16. This is possible because a conjunction requires all its conjuncts to execute along the *same* path.

The second consequence is closely related to the first: It is possible that transactions ψ and $\neg\psi$ may *both* succeed from the same database. Again, the reason is that ψ may succeed along one path, while $\neg\psi$ will succeed along another path. The resolution of this apparent inconsistency is that the transaction $\psi \wedge \neg\psi$ always fails, since it is false on every path.

Example 5.13 (Apparent Inconsistency) Consider a transition oracle that satisfies the following:

$$a \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \qquad\qquad b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_3)$$

where $\mathbf{D}_2 \not\equiv \mathbf{D}_3$, and suppose that the transaction base, $\mathbf{P}$, contains the two rules $c \leftarrow a$ and $\neg c \leftarrow b$. Then *both* the following statements are true:

$$\mathbf{P}, \mathbf{D}_1 \text{---} \models c \qquad\qquad \mathbf{P}, \mathbf{D}_1 \text{---} \models \neg c$$

But the following statement is *false*: $\mathbf{P}, \mathbf{D}_1 \text{---} \models c \wedge \neg c$. $\square$

Inconsistency

The previous example discussed *apparent* inconsistencies in $\mathcal{TR}$. This section discusses *real* inconsistencies. There are three possible sources of inconsistency in $\mathcal{TR}$: the transaction base, the state data oracle, and the interaction between the two. We shall discuss each one in turn.

The first, and simplest, source of inconsistency is the transaction base $\mathbf{P}$. It may contain an inconsistent combination of formulas, such as $\{b,\ c,\ \neg b \vee \neg c\}$. There is only one path-structure that satisfies such a transaction base—one that assigns the special classical structure, $\top$, to every path. Essentially, it is the user's responsibility to avoid this kind of inconsistency, by writing consistent transaction bases. This is easily accomplished in practice, since modern database and logic programming systems infer only positive facts. They are therefore guaranteed to be consistent. Even the advanced examples considered later in this paper, which go well beyond current technology, do not infer negative facts, and so do not lead to inconsistency.

The second source of inconsistency is the transition oracle, $\mathcal{O}^t$, which determines all possible state transitions in the system. In principle, such an oracle may define transitions to inconsistent databases (*i.e.*, states where $\mathcal{O}^d(s)$ is classically unsatisfiable), in which case, transactions may enter such states during execution. If $\mathbf{D}$ is such a state, the only possible view over it is $\top$, *i.e.*, $I_{path}(s) = \top$ in Definition 5.1. Since $\top$ satisfies all formulas,any transaction entering an inconsistent state (or launched from such a state) immediately succeeds. In this sense, inconsistent states act as "black holes" in the state space of $\mathcal{TR}$. The important point here is that inconsistency is "localized;" that is, inconsistency in just one possible database state *does not* spread to all possible database states, and, thus, inconsistent states *do not* render the entire logic useless.

Even if the transaction base $\mathbf{P}$, the data oracle, and the transition oracle are consistent by themselves, they may not be consistent together. This is the third, and most subtle, source of inconsistency. Suppose, for instance, that the transaction base contains the formula $c \leftarrow b$, and that the current database state, s, contains the atom b (assuming the Classical Oracle). Now suppose that we insert the atom $\neg c$ into the database, making a transition to a state where both b and $\neg c$ is true. From here, both c and $\neg c$ are derived. In this case, the database (*i.e.*, the stored formula) is consistent, but the view is not. Again, if $\mathbf{D}$ is such a state, the only possible view over it is $\top$, and it will thus act as a black hole.

For a vast majority of practical applications, however, the above situations do not arise because modern systems normally deal with positive facts only. For instance, for the situation just described, the transition oracle would have to have elementary transitions that insert negative literals into the database. That is, it would have to have entries of the form:

$$\neg c.ins \in \mathcal{O}^t(\mathbf{D}, \mathbf{D} + \{\neg c\})$$

Such updates simply do not occur in modern relational database systems or in classical logic programming systems.[12] Furthermore, as discussed above, logic programming systems do not infer negative

[12] Of course, one may want to *delete* c, but this is radically different from inserting $\neg c$. The operation of deletion was

facts. Thus, there is no way to insert or to derive a negative fact, so inconsistency is impossible. Even the advanced examples considered later in this paper will not lead to inconsistent views, since they do not infer negative facts.

In other applications, however, especially those in Artificial Intelligence, the derivation of both positive and negative facts may be important. In such cases, inconsistency is possible, so users must be careful not to build inconsistent knowledge-bases. This is as true for systems based on first-order logic and modal logic, as it is for systems based on $\mathcal{TR}$. Still, if inconsistency is expected to arise, it would be important to "localize" its effects, so that it will not render the entire knowledge-base useless. Our use of $\top$ addresses this issue. Inconsistency has been addressed in more sophisticated ways in work on para-consistent logics. We mention, in particular, the works [13, 57, 58], since their ideas can be applied to $\mathcal{TR}$ without much difficulty.

6 Proof Theory

Full $\mathcal{TR}$ has a sound and complete proof procedure. Unfortunately, this procedure is not easy to formulate within $\mathcal{TR}$ itself. Indeed, all our attempts to do so resulted in sets of axioms and inference rules that were large, unwieldy, and incomplete. It turns out, however, that $\mathcal{TR}$ (and some other interesting transaction logics) are instances of a more general logic of state change [21], a logic that has a simple and elegant proof theory. We can use this proof theory to prove formulas in $\mathcal{TR}$. The main problem is to translate statements of $\mathcal{TR}$ into statements of the general logic of state change. A complete development of this logic, its proof theory, and the translation of $\mathcal{TR}$ statements is beyond the scope of this paper. The interested reader is referred to a forthcoming work [21] for details.

6.1 Serial Horn Programs

It turns out that it is not always necessary to encode $\mathcal{TR}$ in a more general logic. In certain important situations, $\mathcal{TR}$ has an elegant, top-down, SLD-style proof procedure that can be expressed within $\mathcal{TR}$ itself. These situations are characterized by the conditions listed below, in Definition 6.2. When a $\mathcal{TR}$ program satisfies these conditions, we say that it is a *serial-Horn* program. Like classical Horn programs, serial-Horn programs have both a procedural and a declarative semantics. It is this property that allows a user to *program* transactions within the logic.

Definition 6.1 (Serial-Horn Formulas)

- A *serial goal* is a transaction formula of the form $b_1 \otimes b_2 \otimes ... \otimes b_n$, where each b_i is an atomic formula, and $n \geq 0$.
- An *existential serial goal* is a transaction formula of the form $(\exists \overline{X})\phi$, where ϕ is a serial goal, and $\overline{X}$ is a list of all free variables in ϕ.
- A *serial-Horn rule* is a transaction formula of the form $b \leftarrow \phi$, where b is an atomic formula, and ϕ is a serial goal.
- A *serial-Horn transaction base* is a finite set of serial-Horn rules.

called *erasure* in [54]. For instance, erasure (deletion) of c from $\mathbf{D}$ yields $\mathbf{D} \vee \mathbf{D}'$, where $\mathbf{D}'$ is the result of inserting $\neg c$ into $\mathbf{D}$. If $\mathbf{D}$ is a set of Horn clauses and c is an atom whose predicate does not occur in the rule heads, then erasure of c yields $\mathbf{D} - \{c\}$.

□

The following rules are all serial-Horn:

$$a(Y) \leftarrow b(Y) \qquad a(X,Y) \leftarrow b(X,Y) \otimes c(X,Y) \qquad a(X) \leftarrow b(X,Y) \otimes c(Y,Z) \otimes a(Z)$$

A few comments on Definition 6.1 are in order. First note that if the atomic formulas a_1 ,..., a_n do not update the database, then the expression $a_1 \otimes ... \otimes a_n$ is equivalent to $a_1 \wedge ... \wedge a_n$. Thus, classical Horn rules are a special case of serial-Horn rules. Second, because every serial goal, α, is equivalent to $\alpha \otimes \texttt{state}$, the *empty* serial goal, ( ), can be identified with `state`. Finally, it is important to keep in mind that an empty-bodied rule, $a \longleftarrow (\,)$, is equivalent to $a \longleftarrow \texttt{state}$, and *not* to the atomic formula a. The former rule means that a is true *in every state*, while the latter atom means that a is true *on every path*—not a very useful axiom, since we already have `path`, a proposition that is characterized precisely by this property. For that reason, it is meaningless to include atoms-as-such in transaction bases; their absence also leads to a more uniform proof theory.

To avoid the need to treat atoms `state`, `arc`, and `path` specially in our proof theory, we shall assume that they are defined by the oracles and the transition base. This assumption does not limit the generality, but it simplifies the presentation significantly. Specifically, we assume that `state` is defined by the data oracle so that $\texttt{state} \in \mathcal{O}^d(\mathbf{D})$ for every state identifier $\mathbf{D}$, and `state` does not appear in the head of any rule in the transaction base (and it is not modified by the transition oracle). Similarly, we can assume that $\texttt{arc} \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ for every pair of state identifiers, and that `arc` is defined exclusively by the transition oracle. Finally, `path` can be implemented by the transaction base via the following rules:

$$\texttt{path} \leftarrow \texttt{state} \qquad \texttt{path} \leftarrow \texttt{arc} \otimes \texttt{path}$$

Definition 6.2 (Serial-Horn Conditions) A transaction base, $\mathbf{P}$, a data oracle, $\mathcal{O}^d$, and a transaction formula (or query), Q, satisfy the serial-Horn conditions if:

(i) $\mathbf{P}$ is a set of serial-Horn rules;

(ii) $\mathcal{O}^d$ is a Generalized Horn Oracle;

(iii) Q is an existential serial goal; and

(iv) $\mathcal{O}^d$ is *independent* of $\mathbf{P}$, *i.e.*, for every database state, $\mathbf{D}$, no predicate symbol occurs both in a rule-head in $\mathbf{P}$ and in a rule-body in $\mathcal{O}^d(\mathbf{D})$.

□

In this definition, Condition (iv) intuitively says that database views cannot be defined in terms of transactions. For instance, if the rule $b \leftarrow c$ is in the transaction base, then the rule $a \leftarrow b$ *cannot* be in the database; although the rule $b \leftarrow a$ *can* be in the database. Condition (iv) arises naturally in two situations: (a) when the database is relational (a set of ground atomic formulas), and (b) when a conceptual distinction is desired between updating actions and non-updating queries. In the former case, we can use the Relational Oracle, which is a special case of a Generalized Horn Oracle. In the latter case, the logic would have two sorts of predicates, query predicates and action predicates. Action predicates would be defined only in the transaction base, and query predicates would be defined only in the database. In this case, the independence condition is satisfied with any Generalized Horn Oracle.

The serial-Horn restrictions have a simple interpretation. Primarily, they are a demand for complete information: serial-Horn rules imply that actions are completely specified, and generalized Horn oracles imply that database states are completely specified, *i.e.*, do not contain any indefinite or disjunctive information. In addition, the independence condition simplifies the interaction between the database and the transaction base, and allows the state oracle to return rules with negated premises. Together, these restrictions lead to a simple and practical, SLD-style inference system for action execution, described below.

In contrast to the proof theory developed in this section, the model theory developed in Section 5 is completely general, and does not depend on any syntactic restrictions. If we were interested only in the serial-Horn case, then the model theory of $\mathcal{TR}$ could be simplified. In particular, we would only need to consider the satisfaction of formulas on arcs (pairs of states) instead of on paths (sequences of states). Such a semantics, and the proof theory presented in this section, is powerful enough to deal with the examples in Sections 2.4 and 2.5, which all satisfy the serial-Horn conditions. In general, however, an arc-based semantics has severe limitations. For instance, it does not provide a useful interpretation for the conjunction and the negation of transactions. Furthermore, it does not permit the expression of constraints on transaction execution (since intermediate states are not represented). Consequently, the proof theory developed in this section is not powerful enough to solve path constraints, which require a more general proof theory [21].

The rest of this section develops the proof theory for the serial Horn case.

6.2 Two Inference Systems

Section 5.5 gave detailed examples of inference in $\mathcal{TR}$ in the serial Horn case. These examples were all based on Lemma 5.7, and on items 2–5 in particular. Each of these items is, in effect, an inference rule; and taken together, they form an inference system. In fact, the system based on Lemma 5.7 is sound and complete for the serial-Horn case.[13] We call this simple system the *naive* inference system. Conceptually, the naive system is simple and elegant, but it is not very practical. There are three reasons for this: (*i*) it is a bottom-up system, which does not take the query (transaction) into account, (*ii*) it is a *ground* inference system, one not based on unification, and (*iii*) it records and accesses the *history* of the database, not just the current database state. The first two points mean that the naive proof theory is not an SLD-style inference system, like Prolog; and the third point implies that enormous amounts of storage and disk access may be required, as suggested by Example 5.10.

Fortunately, one can develop more practical inference systems. This section presents two such systems for the serial-Horn case. These systems are dual to each other and have many similarities. For instance, both systems are based on unification, both return most general unifiers (mgu's) as answers to queries, both record only the *current* database state, and both can be operated in either a top-down or a bottom-up mode. Both systems also have exactly three inference rules, which is the minimum number possible, since separate rules are needed to deal with elementary transitions, the transaction base, and the database state. The main difference between the two systems is the kind of inferences they make. In terms of Definition 5.6, one system makes inferences of the form $\mathbf{P}, \mathbf{D} \text{---} \vdash \psi$, while the other makes inferences of the form $\mathbf{P}, \text{---} \mathbf{D} \vdash \psi$.

In both systems, inference can be carried out in two natural ways: top-down (starting at the goal) and bottom-up (starting at the axioms). These two modes of deduction correspond naturally

[13] Item 1 of Lemma 5.7 is not needed here, since it deals with classical conjunction and thus goes beyond the serial-Horn case. It is needed for constraints and is treated in more detail in [21].

to two kinds of execution: forward execution of transactions (the normal kind), and reverse execution (*i.e.*, starting from the current state, and computing subsequent states by executing all updates in reverse). Furthermore, there is an interesting duality that connects the two inference systems and the two modes of deduction: what one system does bottom-up, the other does top-down. This duality has important consequences for implementation. A logic-programming system, for instance, is most naturally implemented as a top-down system, where the query (or transaction) guides the inferences and updates. A database system, however, is most naturally implemented as a bottom-up system, in order to minimize disk accesses. The choice of inference system may thus depend on the kind of software system being built.

6.3 Inference System $\Im^I$

This section develops an inference system for verifying that $\mathbf{P}, \mathbf{D}_0 \text{---} \models \psi$. Informally, this statement says that transaction ψ can successfully execute starting from state $\mathbf{D}_0$. The inference succeeds if and only if it finds an execution path for the transaction ψ, that is, a sequence of databases $\mathbf{D}_1, \ldots, \mathbf{D}_n$ such that $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \psi$. As we shall see, certain inference strategies generate the execution path in a way that corresponds to the intuitive notion of transaction execution. In particular, top-down inference corresponds to normal execution, and bottom-up inference corresponds to reverse execution. We refer to the inference system of this section as $\Im^I$.

Because we are dealing with the serial-Horn case, the transaction ψ is an existential serial goal, *i.e.*, a formula of the form $(\exists \overline{X})\,(a_1 \otimes a_2 \otimes \ldots \otimes a_m)$, where each a_i is an atom. Since *all* free variables in ψ are assumed to be existentially quantified, we shall often omit the $\overline{X}$. However, lest the existential nature of transactions slip our mind, we often leave $(\exists)$ in front of a transaction. Note that this existential quantification is consistent with the traditions of logic programming and databases. The inference rules below all focus on the left end of such transactions. To highlight this focus, we write the rule-bodies as $\phi \otimes rest$, where ϕ is the piece of the conjunction that the inference system is currently focussed on, and $rest$ is the rest of the conjunction.

Recall that a $\mathcal{TR}$ theory comes with a data oracle, $\mathcal{O}^d$, and a transition oracle, $\mathcal{O}^t$, which provide the semantics of databases and elementary transitions, respectively (as described in Sections 3 and 5). $\Im^I$ invokes these oracles to read and update the database.

Definition 6.3 (Inference System $\Im^I$) If $\mathbf{P}$ is a transaction base, then $\Im^I$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.

Axioms: $\mathbf{P}, \mathbf{D} \text{---} \vdash (\,)$.

Inference Rules: In Rules 1–3 below, σ is a substitution, a and b are atomic formulas, and ϕ and $rest$ are serial goals.

1. *Applying transaction definitions:*
   Suppose $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with $b \otimes rest$. If a and b unify with mgu σ, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(b \otimes rest)}$$

2. *Querying the database:*
   If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^d(\mathbf{D}) \models^c (\exists)\,b\sigma$, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, rest\, \sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, (b \otimes rest)}$$

3. *Performing elementary updates:*
If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)\, b\sigma$, then

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\, rest\, \sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, (b \otimes rest)}$$

□

This system manipulates expressions of the form $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$, called *sequents*. The informal meaning of such a sequent is that the transaction $(\exists)\, \phi$ can *succeed* from $\mathbf{D}$, *i.e.*, it can be executed on a path emanating from the database $\mathbf{D}$, *i.e.*, a path whose initial state is $\mathbf{D}$. Each inference rule consists of two sequents, one above the other, and has the following interpretation: If the upper sequent can be inferred, then the lower sequent can also be inferred. Based on the axiom-sequents, the system uses the inference rules to infer more-and-more sequents.

In $\Im^I$, the axioms describe the empty transaction, "( )". By definition, this transaction does nothing and always succeeds. That is, if the user issues the command $?-$ ( ), then the database simply remains in its current state.[14] The axioms formalize this behavior.

The three inference rules describe more complex transactions. They capture the roles of the transaction base, the data oracle, and the transition oracle, respectively. To understand these rules, it is convenient to temporarily ignore the unifier, σ, and to assume that a is identical to b. With this in mind, we can interpret the rules as follows:

1. Inference rule 1 deals with defined transactions. It says that if b is defined by ϕ, and if $\phi \otimes rest$ succeeds from $\mathbf{D}$, then $b \otimes rest$ also succeeds from $\mathbf{D}$. Intuitively, the rule replaces a subroutine definition, ϕ, by its calling sequence, b. This rule is a serial version of modus ponens.

2. Inference rule 2 deals with tests (*i.e.*, queries), which do not cause state changes. It says that if b is true at $\mathbf{D}$, and if $rest$ succeeds from $\mathbf{D}$, then $b \otimes rest$ also succeeds from $\mathbf{D}$. Intuitively, b is a pre-condition that is satisfied by $\mathbf{D}$, and so it can be attached to the front of the transaction $rest$.

3. Inference rule 3 deals with elementary updates. It says that if b changes database $\mathbf{D}_1$ into $\mathbf{D}_2$, and if $rest$ succeeds from $\mathbf{D}_2$, then $b \otimes rest$ succeeds from $\mathbf{D}_1$. Intuitively, b is attached to the front of $rest$, so that the resulting transaction starts from $\mathbf{D}_1$ instead of $\mathbf{D}_2$.

Without the unifier σ, $\Im^I$ would be sound and complete for ground inference, that is, for the special case in which all queries and transactions are ground. Unification "lifts" inference from ground transactions to transactions with free-variables, converting the system into a more-practical, SLD-style system, one that returns most-general-unifiers as answers, as Prolog does. As in classical resolution, any instance of an answer-substitution is a valid answer to a query. Our use of unification exploits the assumption that transactions are existential. In particular, for any substitution σ, if $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi\sigma$ can be inferred, then $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$ can also be inferred. This idea is implicit in each inference rule.

For future reference, we remind the reader of the general notion of deduction in an inference system:

[14] Model theoretically, "( )" corresponds to the special atom `state`, which is true on every path of length 1. In particular, it is true on the 1-path corresponding to any database $\mathbf{D}$.

Definition 6.4 (General Deduction) Given an inference system, a *deduction* (or *proof*) of a sequent, seq_n, is a series of sequents, seq_0, seq_1, ... , seq_{n-1}, seq_n, where each seq_i is either an axiom-sequent or is derived from earlier sequents by an inference rule. □

In $\Im^I$, as in most inference systems, the main purpose of the proof theory is to find deductions for a given sequent. To reduce the search space, many proof theories employ specialized inference strategies. Two well-known strategies are *bottom-up* and *top-down* inference (also known as *forward-chaining* and *backward-chaining*, respectively). In the bottom-up mode, a deduction is constructed by starting with the *first* sequent, seq_0, which is an axiom, then applying an inference rule to derive the second sequent, seq_1, then applying another inference rule to derive the third sequent, seq_2, etc. Bottom-up inference is typical of database query processing. In the top-down mode, a deduction is constructed by starting with the *last* sequent, seq_n, then applying an inference rule *in reverse* to derive the second-last sequent, seq_{n-1}, then the third-last, etc, until an axiom sequent is reached. Top-down inference is typical of logic-programming systems, like Prolog. In $\Im^I$, top-down and bottom-up inference have a special importance, since they correspond closely to transaction execution, as we shall see.

Theorem 6.5 (Soundness and Completeness of $\Im^I$) *Under the serial Horn conditions, the executional entailment* $\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\,\phi$ *holds if and only if there is a deduction in* $\Im^I$ *of the sequent* $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,\phi$.

Proof: See Appendices A and B. □ In this theorem, completeness relies on the serial-Horn conditions,

but soundness does not. The inference system is sound for any transaction base, any data oracle, any transition oracle, and any query. The theorem itself characterizes the inferences in $\Im^I$ model-theoretically in the serial-Horn case.

The examples below illustrate the inference system. In these examples, and in the rest of this paper, all deductions are drawn vertically with the final sequent, seq_n at the top, and the initial sequent, seq_0, at the bottom. With this convention, top-down inference corresponds to building a deduction vertically from the top, down; and bottom-up inference corresponds to building a deduction vertically from the bottom, up. To emphasize the direction of inference, we often insert the word *if* into a deduction. Here and in other examples, we assume the Relational Oracle.

Example 6.6 (Query Processing) Suppose that the transaction base $\mathbf{P}$ contains the following three rules:

$$p \leftarrow q \otimes r \qquad q \leftarrow s \otimes t \qquad r \leftarrow u \otimes v$$

Then the formula p represents a yes/no query, one that is true when the database contains the atoms s, t, u, v. This can be seen from the following deduction (where "$(\exists)$" has been omitted from the queries, since they are variable-free):

	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash p$	
if	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash q \otimes r$	by inference rule 1,
if	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash s \otimes t \otimes r$	by inference rule 1,
if	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash t \otimes r$	by inference rule 2,
if	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash r$	by inference rule 2,
if	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash u \otimes v$	by inference rule 1,
if	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash v$	by inference rule 2,
if	$\mathbf{P}, \{s, t, u, v\} \text{---} \vdash (\,)$	by inference rule 2.

This deduction succeeds because the bottom-most sequent is an axiom. □

Example 6.6 illustrates a similarity between inference system $\Im^I$ and Prolog: operating top-down, both systems process rule-premises from left to right. Our inference system thus provides a formal basis for the left-to-right aspect of Prolog's proof strategy. Of course, $\Im^I$ does not even attempt to mimic certain aspects of Prolog's operational semantics, such as the rule selection strategy, for fear of becoming incomplete.

In the deduction of Example 6.6, the database does *not* change from one sequent to the next, because only inference rules 1 and 2 are used. In the next example, however, the database *does* change, because inference rule 3 is used.

Example 6.7 (Sequential Updates) Suppose that the database is propositional and that the transition oracle, $\mathcal{O}^t$, defines inserts and deletes of single atoms. Thus, for every database state, $\mathbf{D}$, and every atom, d, the transition oracle satisfies the following:

$$d.ins \in \mathcal{O}^t(\mathbf{D}, \mathbf{D} + \{d\}) \qquad\qquad d.del \in \mathcal{O}^t(\mathbf{D}, \mathbf{D} - \{d\})$$

Then the formula $b.ins \otimes c.ins \otimes d.ins$ represents an updating transaction, one that inserts b into the database, then c, and then d. For instance, if the current database state is $\{a\}$, then when the transaction executes, the database changes from $\{a\}$ to $\{a,b\}$ to $\{a,b,c\}$ to $\{a,b,c,d\}$. This behavior is recorded in the following deduction:

$$\begin{array}{lll} & \mathbf{P}, \{a\} \text{---} \vdash b.ins \otimes c.ins \otimes d.ins & \\ \textit{if} & \mathbf{P}, \{a,b\} \text{---} \vdash c.ins \otimes d.ins & \text{by inference rule 3,} \\ \textit{if} & \mathbf{P}, \{a,b,c\} \text{---} \vdash d.ins & \text{by inference rule 3,} \\ \textit{if} & \mathbf{P}, \{a,b,c,d\} \text{---} \vdash (\,) & \text{by inference rule 3.} \end{array}$$

This deduction succeeds because the bottom-most sequent is an axiom. □

6.3.1 Execution as Deduction

Having developed the inference system we must remind ourselves that our original goal was not so much proving statements of the form $\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\,\phi$, but rather of the form

$$\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models (\exists)\,\phi$$

where $\mathbf{D}_0$ is the database state at the time the user issues the transaction $?- (\exists)\phi$. Note that the intermediate database states, $\mathbf{D}_1, \ldots, \mathbf{D}_{n-1}$, and the final state, $\mathbf{D}_n$, are *unknown* by the user at this time. So, an important task for the inference system is to compute these unknown states.

The general notion of deduction is not tight enough to do this conveniently. The problem is that a general deduction may record the execution of many unrelated transactions. Furthermore, this information may be mixed up in a haphazard way, and the various transactions may start from different database states.

Example 6.8 (General Deduction) The deductions from Examples 6.6 and 6.7 can be interleaved to produce perfectly-valid general deductions. The following deduction interleaves the entire deduction from Example 6.7 with the bottom four lines of the deduction from Example 6.6:

8.	$\mathbf{P}, \{s,t,u,v\} \text{---} \vdash r$	by sequent 7.
7.	$\mathbf{P}, \{s,t,u,v\} \text{---} \vdash u \otimes v$	by sequent 4.
6.	$\mathbf{P}, \{a\} \text{---} \vdash b.ins \otimes c.ins \otimes d.ins$	by sequent 5.
5.	$\mathbf{P}, \{a,b\} \text{---} \vdash c.ins \otimes d.ins$	by sequent 2.
4.	$\mathbf{P}, \{s,t,u,v\} \text{---} \vdash v$	by sequent 3.
3.	$\mathbf{P}, \{s,t,u,v\} \text{---} \vdash (\,)$	an axiom.
2.	$\mathbf{P}, \{a,b,c\} \text{---} \vdash d.ins$	by sequent 1.
1.	$\mathbf{P}, \{a,b,c,d\} \text{---} \vdash (\,)$	an axiom.

This deduction interleaves the execution records of two transactions, r and $b.ins \otimes c.ins \otimes d.ins$. Moreover, these two executions start from different database states: r starts from state $\{s,t,u,v\}$, while $b.ins \otimes c.ins \otimes d.ins$ starts from state $\{a\}$. □

Since we are interested in the execution of a particular transaction, it is convenient to introduce a more specialized notion of deduction, which we call *executional deduction*. This notion defines a narrower range of deductions than does Definition 6.4, but without sacrificing completeness. Intuitively, an executional deduction has no extraneous sequents: Each sequent contributes toward proving the final sequent. An executional deduction thus proceeds directly from an axiom sequent, seq_0, to a goal sequent, seq_n, without digressing toward other goals. One of the main characteristics of such deductions is that they record the execution of a single transaction. In particular, the database states mentioned in the sequents correspond exactly to the states the database passes through during transaction execution.

Definition 6.9 (Executional Deduction) Let $\mathbf{P}$ be a transaction base. An *executional deduction* of a transaction, $(\exists)\,\phi$, is any deduction, $seq_0, \ldots, seq_n$, that satisfies the following conditions:

1. The initial sequent, seq_0, has the form $\mathbf{P}, \mathbf{D}' \text{---} \vdash (\,)$, for some database $\mathbf{D}'$.
2. The final sequent, seq_n, has the form $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,\phi$, for some database $\mathbf{D}$.
3. Each sequent, seq_i (for $i > 0$), is obtained from the *previous* sequent, seq_{i-1}, by one of the inference rules of system $\Im^I$ (*i.e.*, seq_{i-1} is the numerator of the rule and seq_i is the denominator).

□

The deductions in Examples 6.6 and 6.7 are both executional. Note how they record the execution of the transactions, keeping track of any changes (or lack thereof) to the database state.

Theorem 6.5 (soundness and completeness) remains valid even if deductions are required to be executional. However, because we now have a stronger form of deduction, we can prove stronger results about it. Theorem 6.5, for instance, does not specify the execution path of the transaction. With executional deduction, we can be specific about the path itself.

Intuitively, the execution path of a transaction is the sequence of states that the database passes through during transaction execution. Each state change is caused by an elementary state transition invoked by the transaction during execution. The execution path is not difficult to extract from an executional deduction. The key observation is that system $\Im^I$ applies elementary transitions exactly when inference rule 3 is invoked. Invoking this rule during inference is the proof-theoretic analogue of executing an elementary transition. Thus, to extract the execution path from an executional

$$
\begin{array}{lll}
 & \mathbf{P}, \mathbf{D}_0 \mathrel{---}\vdash (\exists)\,\phi & \\
 & \quad \ldots & \\
 & \mathbf{P}, \mathbf{D}_0 \mathrel{---}\vdash (\exists)\,\psi_0 & \\
\textit{if} & \mathbf{P}, \mathbf{D}_1 \mathrel{---}\vdash (\exists)\,\phi_1 & \text{by inference rule 3,} \\
 & \quad \ldots & \\
 & \mathbf{P}, \mathbf{D}_1 \mathrel{---}\vdash (\exists)\,\psi_1 & \\
\textit{if} & \mathbf{P}, \mathbf{D}_2 \mathrel{---}\vdash (\exists)\,\phi_2 & \text{by inference rule 3,} \\
 & \quad \ldots & \\
 & \mathbf{P}, \mathbf{D}_{n-1} \mathrel{---}\vdash (\exists)\,\psi_{n-1} & \\
\textit{if} & \mathbf{P}, \mathbf{D}_n \mathrel{---}\vdash (\exists)\,\phi_n & \text{by inference rule 3,} \\
 & \quad \ldots & \\
 & \mathbf{P}, \mathbf{D}_n \mathrel{---}\vdash (\,) &
\end{array}
$$

Figure 1: An Executional Deduction in $\Im^I$

deduction, we need only pick out those points in the deduction where inference rule 3 was applied, as in Figure 1. Given the arbitrary executional deduction in Figure 1, we define its *execution path* to be the sequence $\mathbf{D}_0\,\mathbf{D}_1\,\mathbf{D}_2\,\ldots\,\mathbf{D}_n$. Thus, in Example 6.7, the execution path is $\{a\}$, $\{a, b\}$, $\{a, b, c\}$, $\{a, b, c, d\}$, and in Example 6.6, it is $\{s, t, u, v\}$, a path of length 1. The next theorem provides a model-theoretic meaning for execution paths. It also connects the idea of executional deduction with that of executional entailment, given in Definition 5.6.

Theorem 6.10 (Executional Soundness and Completeness of $\Im^I$) *Under the serial Horn conditions, the following statements are equivalent:*

1. $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\,\phi$.
2. *There is an executional deduction of* $(\exists)\,\phi$ *whose execution path is* $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$.

Proof: See Appendix C. $\Box$ As before, completeness relies on the serial-Horn conditions, but soundness does not.

Using Theorem 6.10, the following two statements can be added to Examples 6.6 and 6.7, respectively:

$$\mathbf{P}, \{s, t, u, v\} \models p$$

$$\mathbf{P}, \{a\}, \{a, b\}, \{a, b, c\}, \{a, b, c, d\} \models b.ins \otimes c.ins \otimes d.ins$$

Notice that these statements make the entire history of the database explicit, yet they are inferred by an inference system that makes only a single database explicit. Notice, too, that any executional deduction can be constructed either top-down or bottom-up.

6.3.2 Executing Transactions

This section defines what it means to *execute* a transaction formula in $\mathcal{TR}$, thus making precise a notion that we have used informally until now. In fact, we shall define two notions of execution, which we call *normal* and *reverse* execution. They correspond naturally to two forms of executional deduction: top-down and bottom-up, respectively.

As mentioned earlier, given a database $\mathbf{D}_0$ and a transaction formula $(\exists)\,\psi$, we wish to prove the statement

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\,\phi \tag{9}$$

for some sequence of databases, $\mathbf{D}_1, \ldots, \mathbf{D}_n$. The important point is that these latter databases are *unknown* at the outset and must be computed during the proof. The process of constructing such a proof is called a *normal execution* of $(\exists)\,\phi$ with respect to $\mathbf{P}$. Thus, given the state $\mathbf{D}_0$, normal execution determines whether the transaction $(\exists)\,\phi$ can start from state $\mathbf{D}_0$, and if so, it computes the successor states, $\mathbf{D}_1, \ldots, \mathbf{D}_n$. Intuitively, this computation corresponds to updating the user's database from $\mathbf{D}_0$ to $\mathbf{D}_1 \ldots$ to $\mathbf{D}_n$.

Figure 1 suggests a natural approach to normal execution: Use top-down inference to prove the sequent $\mathbf{P}, \mathbf{D}_0 \dashrightarrow\vdash (\exists)\,\phi$. Note that this sequent, which is the starting point for top-down inference, contains only known quantities. Furthermore, as the figure shows, by reasoning top-down, the successor states are computed in order, from $\mathbf{D}_1$ to $\mathbf{D}_2 \ldots$ to $\mathbf{D}_n$, thus carrying out a normal execution of the transaction $(\exists)\,\phi$. This can also be seen in Example 6.7, since by constructing the deduction in a top-down fashion, a sequence of databases is computed, from the initial state, $\{a\}$, through the intermediate states, to the final state, $\{a, b, c, d\}$, thus carrying out a normal execution of the transaction $b.ins \otimes c.ins \otimes d.ins$. As these examples show, top-down inference computes databases in the order $\mathbf{D}_0$, $\mathbf{D}_1 \ldots \mathbf{D}_n$. The reason for this behavior can be found in inference rule 3: When this rule is applied in reverse, the database is changed from $\mathbf{D}_1$ to $\mathbf{D}_2$, *i.e.*, from an earlier state to a later state. This is why top-down inference is well-suited to normal execution.

There are other ways of using system $\Im^I$ to perform normal execution. However, none is as natural as top-down inference, and many are impractical and involve huge amounts of unguided search. Large search spaces are often a problem in logic programming, but with updates, a new dimension of search comes into play. To illustrate, consider bottom-up inference. As Figure 1 shows, it begins with the axiom-sequent $\mathbf{P}, \mathbf{D}_n \dashrightarrow\vdash (\,)$ and ends with the goal-sequent $\mathbf{P}, \mathbf{D}_0 \dashrightarrow\vdash (\exists)\,\phi$. The axiom sequent involves an *unknown* quantity, the final database state, $\mathbf{D}_n$. Bottom-up inference cannot begin without a value for this state. To get around this problem, we could choose a value for $\mathbf{D}_n$ at random, and then try to verify the sequent $\mathbf{P}, \mathbf{D}_0 \dashrightarrow\vdash (\exists)\,\phi$ using bottom-up inference. If the verification fails, we would choose another value for $\mathbf{D}_n$ and try again, repeating the process until we stumbled upon the correct value of $\mathbf{D}_n$ by chance. This approach is obviously impractical. It is clearly more practical to compute the new database states directly from the initial state, as top-down inference does.

Not all top-down systems avoid the problem of having to know (or randomly choose) the final database state. For instance, although [69, 68] develop a logical language for transactions, the inference system there requires that *both* the initial and final database states be known *before* inference can begin. This restriction makes the inference system impractical in any mode, be it top-down or bottom-up.[15]

6.3.3 Reverse Execution

So far, we have described how to execute transactions "normally." However, it is also possible to execute them *in reverse*. This possibility leads to a strong duality between top-down and bottom-up

[15] Being aware of this problem, [69, 68] develop an *interpreter* that does not need to know the final state. However, this interpreter is not based on a proof theory and is incomplete, too.

inference.

In reverse execution of $(\exists)\,\psi$, the current database is $\mathbf{D}_n$ and we wish to prove Statement (9) for some sequence of databases, $\mathbf{D}_0, \ldots, \mathbf{D}_{n-1}$. The important point is that these latter databases are *unknown* at the outset and must be computed during the proof. The process of constructing such a proof is called a *reverse execution* of $(\exists)\,\phi$ with respect to $\mathbf{P}$. Thus, given the state $\mathbf{D}_n$, reverse execution determines whether the transaction $(\exists)\,\phi$ can *terminate* at the state $\mathbf{D}_n$, and if so, it computes the predecessor states, $\mathbf{D}_{n-1}, \ldots, \mathbf{D}_0$. Intuitively, this computation corresponds to updating the user's database in reverse, from $\mathbf{D}_n$ to $\mathbf{D}_{n-1}$ ... to $\mathbf{D}_0$.

Bottom-up inference carries out reverse-execution just as naturally as top-down inference carries out normal execution. This duality can be seen in Example 6.7. The deduction in this example is correct no matter how it is constructed, top-down or bottom-up. By constructing it top-down, we start with the initial database, $\{a\}$, and compute its successor states. By constructing it bottom-up, we start with the database $\{a, b, c, d\}$ and compute states in reverse order.

Figure 1 illustrates the general case. In this figure, bottom-up inference starts with the axiom sequent $\mathbf{P}, \mathbf{D}_n \mathbin{---}\vdash (\,)$ and works toward the goal-sequent $\mathbf{P}, \mathbf{D}_0 \mathbin{---}\vdash (\exists)\,\phi$. Inference thus starts with a known quantity, the state $\mathbf{D}_n$, from which it computes other states in reverse order, from $\mathbf{D}_{n-1}$ to $\mathbf{D}_{n-2}$ ... to $\mathbf{D}_0$. The reason for this behavior can be found in inference rule 3: When this rule is applied, the database is changed from $\mathbf{D}_2$ to $\mathbf{D}_1$, *i.e.*, from a later state to an earlier state. This is why bottom-up inference is well-suited to reverse execution. Bottom-up inference thus has the same advantage with reverse execution that top-down inference has with normal execution: it starts with a *known* database state. Likewise, top-down inference now has the disadvantage of starting with an *unknown* state, $\mathbf{D}_0$. It is thus impractical as a method of reverse execution.

Examples 6.6 and 6.7 illustrate other aspects of top-down and bottom-up inference. For instance, when operated top-down, the inference system decomposes transactions from left to right, that is, in the order of normal execution. But, when operated bottom-up, the system builds transactions up from right to left, that is, in the order of reverse execution. These observations further illustrate the duality between the two modes of inference and the two modes of execution.

6.3.4 Example: Non-Deterministic Updates

Having developed the proof theory, we can now apply it to more complex examples. This section examines the proof theory in the context of non-deterministic transactions, that is, transactions that have more than one possible outcome.

Using the same transition oracle as in Example 6.7, we consider a transaction base, $\mathbf{P}$, consisting of the following two rules:

$$p \leftarrow a \otimes c.del \tag{10}$$

$$p \leftarrow b \otimes d.del \tag{11}$$

Here, the atoms a and b act as pre-conditions. That is, c is deleted only if a is true in the initial database state; and d is deleted only if b is true in the initial state. The behavior of transaction p thus depends on which pre-conditions are true. If neither pre-condition is true, then neither rule can execute, so the transaction fails. If only one pre-condition is true, then only one rule can execute, so the transaction is deterministic. If both pre-conditions are true, then both rules can execute, so the transaction is non-deterministic. Formally, the latter possibility means that there can be two proofs of

p, each resulting in a different final state of the database. Here is a proof of $\mathbf{P}, \{a,b,c,d\}, \{a,b,d\} \models p$, a proof that uses Rule (10):

$$
\begin{array}{lll}
 & \mathbf{P}, \{a,b,c,d\} \text{---} \vdash p & \\
\textit{if} & \mathbf{P}, \{a,b,c,d\} \text{---} \vdash a \otimes c.del & \text{by inference rule 1,} \\
\textit{if} & \mathbf{P}, \{a,b,c,d\} \text{---} \vdash c.del & \text{by inference rule 2,} \\
\textit{if} & \mathbf{P}, \{a,b,d\} \text{---} \vdash (\,) & \text{by inference rule 3.}
\end{array}
$$

Likewise, here is a proof of $\mathbf{P}, \{a,b,c,d\}, \{a,b,c\} \models p$, a proof that uses Rule (11):

$$
\begin{array}{lll}
 & \mathbf{P}, \{a,b,c,d\} \text{---} \vdash p & \\
\textit{if} & \mathbf{P}, \{a,b,c,d\} \text{---} \vdash b \otimes d.del & \text{by inference rule 1,} \\
\textit{if} & \mathbf{P}, \{a,b,c,d\} \text{---} \vdash d.del & \text{by inference rule 2,} \\
\textit{if} & \mathbf{P}, \{a,b,c\} \text{---} \vdash (\,) & \text{by inference rule 3.}
\end{array}
$$

This example shows that by generating all possible executional deductions, we can enumerate all possible outcomes of a transaction. Alternatively, by arbitrarily selecting a single deduction, we can execute transactions non-deterministically.

6.3.5 Example: Inference with Unification

In the previous examples of this section, all the formulas were propositional. Being conceptually simple, these examples are a good illustration of the basic ideas behind the proof theory. However, propositional formulas are not typical of either database queries or logic programs. This section shows how inference works in the presence of variables. In this case, the inference system returns a substitution, as Prolog does. The substitution specifies values of the free variables for which the transaction succeeds. The example itself is similar to those in Section 5.5, in which a robot simulator moves blocks around a table top. That section illustrated naive inference, in which inferences are made blindly and inefficiently, with no regard for the transaction actually requested by the user. In contrast, this section is concerned with the efficient execution of a particular, user-specified transaction. For comparison purposes, we rework Example 5.8.

We consider a transaction base, $\mathbf{P}$, containing the following rule, which describes the effect of picking up a block X:

$$pickup(X) \leftarrow isclear(X) \otimes on(X,Y) \otimes on.del(X,Y) \otimes isclear.ins(Y)$$

Suppose the user orders the robot to pick up a block, by typing $?-\ pickup(X)$. Since the block is left unspecified, the transaction is non-deterministic. The inference system attempts to find a value for X that enables the transaction to succeed, updating the database in the process. To illustrate, suppose the initial database represents an arrangement of three blocks where $blkA$ is on top of $blkC$, and $blkB$ stands alone. As shown in Example 5.8, if the robot picks up $blkA$, then the database should be updated from state $\mathbf{D}_0$ to $\mathbf{D}_1$ to $\mathbf{D}_2$, where

$$
\begin{array}{lcl}
\mathbf{D}_0 & = & \{isclear(blkA), isclear(blkB), on(blkA, blkC)\} \\
\mathbf{D}_1 & = & \{isclear(blkA), isclear(blkB)\} \\
\mathbf{D}_2 & = & \{isclear(blkA), isclear(blkB), isclear(blkC)\}
\end{array}
$$

An executional deduction in which the robot picks up $blkA$ is shown in the table below. In this table, each sequent is derived from the sequent immediately below by an inference rule, and the bottom-most sequent is an axiom. Each inference involves unifying the leftmost atom in the transaction against either a database fact (returned by the data oracle), a transaction name (defined in the transaction base), or an elementary transition (returned by the transition oracle). For example, in deriving sequent 5 from sequent 4, the inference system unifies the atom $isclear(X)$ in the transaction against the atom $isclear(blkA)$ in the database, $\mathbf{D}_0$. It is in this way that the system chooses to pick up $blkA$. Likewise, in deriving sequent 4 from sequent 3, the inference system unifies the atom $on(blkA, Y)$ in the transaction against the atom $on(blkA, blkC)$ in the database. In this way, the system retrieves $blkC$, the block on which $blkA$ is resting. Note that each line in the table shows three items: a numbered sequent, the inference rule used in deriving it from the sequent below, and the unifying substitution. The answer substitution is obtained by composing all the unifiers, which yields $\{X/blkA, Y/blkC\}$, and then projecting onto the substitution for X, which yields $\{X/blkA\}$. The operational interpretation of this proof is that the robot has picked up $blkA$.

Rule	Unifier	Sequent
1		6. $\mathbf{P}, \mathbf{D}_0 \text{---} \vdash (\exists)\, pickup(X)$
2	$\{\}$	5. $\mathbf{P}, \mathbf{D}_0 \text{---} \vdash (\exists)\, [isclear(X) \otimes on(X, Y) \otimes on.del(X, Y) \otimes isclear.ins(Y)]$
2	$\{X/blkA\}$	4. $\mathbf{P}, \mathbf{D}_0 \text{---} \vdash (\exists)\, [on(blkA, Y) \otimes on.del(blkA, Y) \otimes isclear.ins(Y)]$
3	$\{Y/blkC\}$	3. $\mathbf{P}, \mathbf{D}_0 \text{---} \vdash (\exists)\, [on.del(blkA, blkC) \otimes isclear.ins(blkC)]$
3	$\{\}$	2. $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash isclear.ins(blkC)$
	$\{\}$	1. $\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\,)$

6.3.6 The Role of the Oracles—Observations

Two aspects of the proof theory—both related to the data and transition oracles—are worth mentioning.

Since the transition oracle is an infinite relation, any implementation of inference rule 3 would include an algorithm for enumerating the oracle. Such implementation of an infinite axiom base by an algorithm is akin to algorithmic implementations of inference rules, such as the resolution rule, which have no finite representation in classical logic.

In $\Im^I$, the enumeration algorithm would take an update b and a database state D as input, and then it would enumerate all possible successor states, *i.e.*, states, D', such that $b \in \mathcal{O}^t(D, D')$. For instance, given $d.ins$ and $\{a, b, c\}$ as input, the algorithm would return $\{a, b, c, d\}$ as output (assuming the Relational Oracle). Thus, procedures that perform elementary updates are treated as black boxes that can be "plugged into" the inference system.

Non-enumerable transition oracles are easy to conceive, and they are sometimes useful in practical programming languages, especially in the case of data oracles. For instance, testing the emptiness of a predicate is a frequent operation in logic programming, but, unfortunately, it is not recursively enumerable. To make use of such an operation in $\mathcal{TR}$, we could introduce a predicate, $p.empty$, for each predicate p, where $p.empty \in \mathcal{O}^d(\mathbf{D})$, if, and only if p has an empty extension in $\mathbf{D}$. Since $\mathbf{D}$ can be an arbitrary logic program, this data oracle cannot be recursively enumerated. Nevertheless, relying on the oracle, Rule 3 of system $\Im^I$ will effectively test for an empty relation whenever an elementary update of the form *p.empty* is encountered. Notice that this is done in a completely monotonic way, without invoking negation-as-failure, which is otherwise the standard logic-programming solution. Of course, in practice, transition oracles that are non-*r.e.* cannot be implemented. In such cases, the algorithms that attempt to enumerate the oracle would not always terminate, or they would be defined only for a practical subset of databases (such as relational or deductive databases), or approximation algorithms could be used. All of these approaches are easy to encode as oracles, but are awkward or impossible to encode by other declarative means.

6.4 Inference System $\Im^{II}$

We now develop an inference system, called $\Im^{II}$, that is dual to system $\Im^I$. In $\Im^{II}$, reverse execution is achieved via *top-down* deduction, while normal transaction execution is obtained via *bottom-up* deduction. The new system manipulates sequents of the form $\mathbf{P}, \text{---}\, \mathbf{D} \vdash \psi$. Intuitively, such a sequent says that transaction ψ can execute in the normal direction along a path *terminating* at database state $\mathbf{D}$, or equivalently, that ψ can execute in reverse *starting* from state $\mathbf{D}$. As with system $\Im^I$, the ultimate goal of $\Im^{II}$ is to prove statements of the form (9). As before, different kinds of deduction correspond to different kinds of execution, normal or reverse.

As in Section 6.3, we assume that the serial-Horn conditions are satisfied. Transaction ψ is thus an existential serial goal, *i.e.*, a formula of the form $(\exists)\,(a_1 \otimes a_2 \otimes \ldots \otimes a_m)$, where each a_i is an atom. Unlike $\Im^I$, though, $\Im^{II}$ focuses on the *right* end of a transaction. We therefore represent transactions as $(\exists)\,(rest \otimes \phi)$, where ϕ is the piece of the transaction that the inference system is currently focussed on, and $rest$ is the part of the rule-body preceding ϕ.

Recall that a $\mathcal{TR}$ theory comes with a data oracle, $\mathcal{O}^d$, and a transition oracle, $\mathcal{O}^t$, which provide the semantics of databases and elementary transitions, respectively (as described in Sections 3 and 5). $\Im^{II}$ invokes these oracles to read and update the database.

Definition 6.11 (Inference System $\Im^{II}$) If $\mathbf{P}$ is a transaction base, then $\Im^{II}$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.

Axioms: $\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\,)$.

Inference Rules: In Rules 1′–3′ below, σ is a substitution, a and b are atomic formulas, and ϕ and $rest$ are serial goals.

1′. *Applying transaction definitions:*
Suppose that $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with $b \otimes rest$. If a and b unify with mgu σ, then:

$$\frac{\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\,(rest \otimes \phi)\sigma}{\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\,(rest \otimes b)}$$

2′. *Querying the database:*
If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^d(\mathbf{D}) \models^c (\exists)\, b\sigma$, then

$$\frac{\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\,(rest \otimes b)}$$

3′. *Performing elementary updates:*
If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)\, b\sigma$, then

$$\frac{\mathbf{P}, \text{---}\,\mathbf{D}_1 \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \text{---}\,\mathbf{D}_2 \vdash (\exists)\, (rest \otimes b)}$$

□

As before, each inference rule consists of two sequents, one above the other, and each means that if the upper sequent can be inferred, then the lower sequent can also be inferred. As with all logical inference systems, $\Im^{II}$ can be operated in a top-down or a bottom-up mode.

These axioms and inference rules correspond closely to those of $\Im^I$. In both systems, the axioms describe the empty transaction, ( ), a transaction that does nothing. Likewise, Rules 1 and 1′ abstract a transaction by replacing a sequence of atoms by a single atom; Rules 2 and 2′ extend a transaction by appending a test (or query) to it; and Rules 3 and 3′ extend a transaction by appending an elementary update to it. An important difference between the two systems is in the way updates are applied in Rule 3′: In going from the upper sequent to the lower sequent, Rule 3′ moves the database from state $\mathbf{D}_1$ to state $\mathbf{D}_2$, that is, from the current state to a *later* state. In contrast, in $\Im^I$, Rule 3 moves the database from the current state to an *earlier* state. Thus, when operated in a bottom-up mode, $\Im^{II}$ executes transactions in a *forward* direction, whereas $\Im^I$ executes them in a *reverse* direction. This is another way in which the two inference systems are dual.

The notion of deduction is the same for $\Im^{II}$ as for $\Im^I$. The definitions of normal and reverse execution need further adjustments, though, as do the theorems of soundness and completeness.

Theorem 6.12 (Soundness and Completeness of $\Im^{II}$) *Under the serial Horn conditions, the executional entailment* $\mathbf{P}, \text{---}\,\mathbf{D} \models (\exists)\,\phi$ *holds if and only if there is a (general) deduction in* $\Im^{II}$ *of the sequent* $\mathbf{P}, \text{---}\,\mathbf{D} \vdash (\exists)\,\phi$.

Proof: Parallel to the proof of Theorem 6.5 presented in Appendices A and B. □

The examples below illustrate the new inference system. They correspond to the examples of Section 6.3 and use the same transition oracle. As in that section, we draw deductions vertically with the final sequent, seq_n, at the top, and the initial sequent, seq_0, at the bottom. Thus, top-down and bottom-up inference still correspond to building a deduction from top to bottom and from bottom to top, respectively.

Example 6.13 (Query Processing) As in Example 6.6, suppose that the transaction base $\mathbf{P}$ contains the following three rules:

$$p \leftarrow q \otimes r \qquad q \leftarrow s \otimes t \qquad r \leftarrow u \otimes v$$

Then p represents a yes/no query; it is true when the database contains the atoms s, t, u, v. This can be seen from the following deduction (where "$(\exists)$" has been omitted from the queries, since they are variable-free):

$$
\begin{array}{lll}
 & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash p & \\
\textit{if} & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash q \otimes r & \text{by inference rule } 1', \\
\textit{if} & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash q \otimes u \otimes v & \text{by inference rule } 1', \\
\textit{if} & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash q \otimes u & \text{by inference rule } 2', \\
\textit{if} & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash q & \text{by inference rule } 2', \\
\textit{if} & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash s \otimes t & \text{by inference rule } 1', \\
\textit{if} & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash s & \text{by inference rule } 2', \\
\textit{if} & \mathbf{P}, \text{---} \{s,t,u,v\} \vdash (\,) & \text{by inference rule } 2',
\end{array}
$$

This deduction succeeds because the bottom-most sequent is an axiom. □

Notice that in Example 6.13, the inference system focuses on the right side of the query, whereas in Example 6.6, it focuses on the left side. Thus, when operating in a top-down mode, $\Im^{II}$ processes queries from right to left, whereas $\Im^{I}$ processes them from left to right.

Since the deduction in Example 6.13 uses inference rules $1'$ and $2'$ only, the databases does not change during inference. In the next example, however, the deduction uses inference rule $3'$, so the database *does* change. Notice that the direction of update is *opposite* to that of system $\Im^{I}$. This is due to inference rules 3 and $3'$, which update the database in opposite directions.

Example 6.14 (Sequential Updates) As in Example 6.7, the formula $b.ins \otimes c.ins \otimes d.ins$ represents an updating transaction, one that inserts b into the database, then c, and then d. Thus, if the current database state is $\{a\}$, then when the transaction executes, the database changes from $\{a\}$ to $\{a,b\}$ to $\{a,b,c\}$ to $\{a,b,c,d\}$. This behavior can be seen in the deduction below by reading it from bottom to top. Alternatively, by reading it from top to bottom, a reverse execution can be seen.

$$
\begin{array}{lll}
 & \mathbf{P}, \text{---} \{a,b,c,d\} \vdash b.ins \otimes c.ins \otimes d.ins & \\
\textit{if} & \mathbf{P}, \text{---} \{a,b,c\} \vdash b.ins \otimes c.ins & \text{by inference rule } 3', \\
\textit{if} & \mathbf{P}, \text{---} \{a,b\} \vdash b.ins & \text{by inference rule } 3', \\
\textit{if} & \mathbf{P}, \text{---} \{a\} \vdash (\,) & \text{by inference rule } 3'.
\end{array}
$$

This deduction succeeds because the bottom-most sequent is an axiom. □

As with system $\Im^{I}$, general deductions are not always as simple as those in Examples 6.13 and 6.14. In general, a deduction may record the execution of many transactions, interleaved in a haphazard way. Since we are interested in the execution of single transactions, we again introduce the stronger notion of *executional* deduction. The deductions in Examples 6.13 and 6.14 are both executional.

Definition 6.15 (Executional Deduction in $\Im^{II}$) Let $\mathbf{P}$ be a transaction base. An *executional deduction* of a transaction, $(\exists)\,\phi$, is any deduction, $seq_0, \ldots, seq_n$, in $\Im^{II}$ that satisfies the following conditions:

1. The initial sequent, seq_0, has the form $\mathbf{P}, \text{---}\, \mathbf{D}' \vdash (\,)$, for some database $\mathbf{D}'$.
2. The final sequent, seq_n, has the form $\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\,\phi$, for some database $\mathbf{D}$.
3. Each sequent seq_i, (for $i > 0$), is obtained from the *previous* sequent, seq_{i-1}, by one of the inference rules of system $\Im^{II}$ (*i.e.*, seq_{i-1} is the numerator of the rule and seq_i is the denominator).

$$
\begin{array}{lll}
 & \mathbf{P}, \text{---}\, \mathbf{D}_n \vdash (\exists)\,\phi & \\
 & \quad \ldots & \\
 & \mathbf{P}, \text{---}\, \mathbf{D}_n \vdash (\exists)\,\psi_n & \\
\textit{if} & \mathbf{P}, \text{---}\, \mathbf{D}_{n-1} \vdash (\exists)\,\phi_{n-1} & \text{by inference rule } 3', \\
 & \quad \ldots & \\
 & \mathbf{P}, \text{---}\, \mathbf{D}_2 \vdash (\exists)\,\psi_2 & \\
\textit{if} & \mathbf{P}, \text{---}\, \mathbf{D}_1 \vdash (\exists)\,\phi_1 & \text{by inference rule } 3', \\
 & \quad \ldots & \\
 & \mathbf{P}, \text{---}\, \mathbf{D}_1 \vdash (\exists)\,\psi_1 & \\
\textit{if} & \mathbf{P}, \text{---}\, \mathbf{D}_0 \vdash (\exists)\,\phi_0 & \text{by inference rule } 3'. \\
 & \quad \ldots & \\
 & \mathbf{P}, \text{---}\, \mathbf{D}_0 \vdash (\,) &
\end{array}
$$

Figure 2: An Executional Deduction in $\Im^{II}$

□

Given an executional deduction for system $\Im^{II}$, Example 6.14 suggests a notion of execution path that is similar to the corresponding notion for $\Im^{I}$. Once again, the key observation is that system $\Im^{II}$ applies elementary transitions exactly when inference rule $3'$ is invoked. This time, however, we define the execution path to go from the bottom to top of a deduction, instead of the other way around. To be more precise, consider an arbitrary executional deduction in $\Im^{II}$, as shown in Figure 2. We define the execution path of this deduction to be the sequence $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$. Thus, in Example 6.14, the execution path is $\{a\}$, $\{a,b\}$, $\{a,b,c\}$, $\{a,b,c,d\}$, and in Example 6.13, it is $\{s,t,u,v\}$, a path of length 1.

The common point in the notions of execution path for $\Im^{I}$ and $\Im^{II}$ is that, in both cases, the existence of an executional deduction, with a corresponding execution path, $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$, is tantamount to the existence of the executional entailment $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\,\phi$. Theorem 6.10, above, establishes this for $\Im^{I}$, and Theorem 6.16, below, establishes it for $\Im^{II}$. These theorems thus provide a model-theoretic meaning for execution paths. They also connect the idea of executional deduction with that of executional entailment, given in Definition 5.6.

Theorem 6.16 (Executional Soundness and Completeness of $\Im^{II}$) *Under the serial Horn conditions, the following statements are equivalent:*

1. $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\,\phi$.
2. *There is an executional deduction of $(\exists)\,\phi$ in $\Im^{II}$ whose execution path is $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$.*

Proof: Parallel to the proof of Theorem 6.10 presented in Appendix C. □

When applied to Examples 6.13 and 6.14, Theorem 6.16 says that the deductions in those examples prove the following executional entailments, respectively:

$$\mathbf{P}, \{s,t,u,v\} \models p$$

$$\mathbf{P}, \{a\}, \{a,b\}, \{a,b,c\}, \{a,b,c,d\} \models b.ins \otimes c.ins \otimes d.ins$$

As with system $\Im^I$, we can execute transactions by constructing executional deductions. As Figure 2 shows, by constructing the deduction from bottom to top, the database is systematically updated from $\mathbf{D}_0$ to $\mathbf{D}_1 \ldots$ to $\mathbf{D}_n$. This is *normal* execution. Likewise, by constructing the deduction from top to bottom, the database is systematically updated from $\mathbf{D}_n$ to $\mathbf{D}_{n-1} \ldots$ to $\mathbf{D}_0$. This is *reverse* execution. Example 6.14 illustrates why transactions can be executed in both forward and reverse directions. When operated bottom-up, system $\Im^{II}$ processes transactions from right to left, that is, in the order of normal execution. When operated top-down, however, it processes transactions from left to right, the order of reverse execution.

7 Applications

A wide variety of interesting and useful formulas can be constructed in $\mathcal{TR}$, formulas that capture many of the novel and important features of database and knowledge-base systems. These features include transaction definition and execution, ad hoc queries, view updates, consistency maintenance, bulk updates, non-determinism, sampling, static and dynamic integrity-constraints, and more. This section describes some of these applications. As part of the discussion, we shall see how $\mathcal{TR}$ handles the so-called frame problem in planning [71]. We shall also see, in Section 8, that the semantics and the proof theory of $\mathcal{TR}$ can easily accommodate a modal necessity operator, $\Box$, which captures a whole new range of applications. These applications include hypothetical reasoning, subjunctive queries and counterfactuals, imperative programming constructs, active databases, software verification, and more. $\mathcal{TR}$ thus provides a wide range of features whose amalgamation in a single declarative formalism has proved elusive in the past. Furthermore, these features all follow naturally from $\mathcal{TR}$'s path-based semantics.

Except for the examples of constraints presented in Section 7.7, all examples in this section satisfy the serial-Horn conditions, so they can be dealt with using the inference systems of Section 6.

7.1 Consistency Maintenance

Often, updating one relation entails making additional updates to other relations in order to maintain the semantic consistency of the database. In such cases, updates to a relation can be done through special procedures that handle the details of consistency maintenance. Such procedures are easily defined in $\mathcal{TR}$.

For example, suppose a university has a database of students, courses, and professors. This database includes the following four base relations:

- *takes(Stud,Crs,Sec)*, which records the students enrolled in each section of each course.
- *enrolled(Crs,N)*, which records the total number of students, N, enrolled in a course.
- *instructs(Prof,Crs,Sec)*, which records the professors who teach each section of each course.
- *load(Prof,N)*, which records each professor's course load, N, *i.e.*, the total number of classes that he teaches.

Per the convention adopted in this paper, we assume that for each base relation, p, the transition oracle underlying the database system defines two elementary update predicates, *p.ins* and *p.del*, for inserting and deleting tuples from relation p.

Using these elementary updates, we define two update-procedures by which students drop courses and professors are relieved from teaching sections of a course. These procedures ensure database consistency by decrementing the enrollment total when a student drops a course, and by decrementing a professor's course load when he is relieved from teaching a course.

$$\begin{aligned}
drop(Stud, Crs, Sec) &\leftarrow takes(Stud, Crs, Sec) \otimes takes.del(Stud, Crs, Sec) \otimes decr_enrolled(Crs) \\
relieve(Prof, Crs, Sec) &\leftarrow instructs(Prof, Crs, Sec) \otimes instructs.del(Prof, Crs, Sec) \otimes decr_load(Prof) \\
decr_enrolled(Crs) &\leftarrow enrolled(Crs, N) \otimes enrolled.del(Crs, N) \otimes enrolled.ins(Crs, N-1) \\
decr_load(Prof) &\leftarrow load(Prof, N) \otimes load.del(Prof, N) \otimes load.ins(Prof, N-1)
\end{aligned}$$

The last two rules define procedures for decrementing the enrollment of a course and the teaching load of a professor, respectively.

7.2 View Updates

Updating a view is often an ill-defined or non-deterministic process, since changes to a view may not uniquely determine the corresponding changes to the underlying stored database. To illustrate the problems and some solutions, consider the university database of Section 7.1 to which we add the following view definition that indicates which professors teach which courses to which students:

$$teaches(Prof, Stud, Crs) \leftarrow takes(Stud, Crs, Sec) \otimes instructs(Prof, Crs, Sec)$$

Since *teaches* is not a base predicate, the transition oracle does not provide update-procedures for it. The problem is that such updates are underspecified, since an update to *teaches* must be carried out in terms of updates to the base predicates *takes* and *instructs*. For instance, a deletion from *teaches* requires either a deletion from *takes* or a deletion from *instructs*. Since there are two choices, this view update is non-deterministic.

$\mathcal{TR}$ offers two solutions to this problem. The first one is to define a distinct procedure for each allowed way of deleting a tuple from a view. The second solution is based on defining a non-deterministic transaction for removing tuples from a view.

To illustrate the first approach, we define a procedure called *rem_student* that allows a user of the view to remove a student from a course. Likewise, we define a procedure called *rem_prof* that allows a user to remove a professor from a course. These two procedures are defined as follows:

$$\begin{aligned}
rem_student(Prof, Stud, Crs) &\leftarrow takes(Stud, Crs, Sec) \otimes drop(Stud, Crs, Sec) \\
rem_prof(Prof, Stud, Crs) &\leftarrow takes(Stud, Crs, Sec) \otimes relieve(Prof, Crs, Sec)
\end{aligned}$$

In this way, a user can do view deletions without knowing what section of a course a student takes or a prof teaches, and without being given direct access to the transactions *drop* and *relieve*. This approach to view updates is similar to that advocated for object-oriented databases, in which a different update method is programmed for each allowed view update [2, 10].

The second approach to the above problem is to define a non-deterministic update-procedure, *rem_teaches*, by combining the above definitions of *rem_student* and *rem_prof*:

$$\begin{aligned}
rem_teaches(Prof, Stud, Crs) &\leftarrow takes(Stud, Crs, Sec) \otimes drop(Stud, Crs, Sec) \\
rem_teaches(Prof, Stud, Crs) &\leftarrow takes(Stud, Crs, Sec) \otimes relieve(Prof, Crs, Sec)
\end{aligned}$$

To delete the fact that a professor teaches a course to a certain student, the system can perform one of two actions: (i) it can drop the student from the course, or (ii) it can relieve the professor from the course. This choice is non-deterministic and is made by the system at run time.

Of course, a user will not usually want to leave such choices entirely to the database system. In such cases, the user can constrain the system's choice, to ensure, for instance, that a deletion from the *teaches* relation does not relieve a professor from a course. To do this, he could specify the following transaction:

$$?-\ load(alberto, N) \otimes rem\_teaches(alberto, mariano, cs100) \otimes load(alberto, N)$$

This transaction removes $teaches(alberto, mariano, cs100)$ from the view, but only if Alberto's course-load remains the same after the update. Thus, the path in which Mariano drops *cs100* will be chosen. Alternatively, the user might want to ensure that the transaction does *not* drop Mariano from the course. In this case, he would write:

$$?-\ enrollment(cs100, N) \otimes rem\_teaches(alberto, mariano, cs100) \otimes enrollment(cs100, N)$$

This transaction succeeds only if the enrollment in the course remains the same after the transaction execution. Thus, the path in which Alberto is relieved from teaching $cs100$ will be chosen. By such means, we can constrain the way in which view updates are carried out. More generally, we can constrain the way in which any transaction is carried out, and without having to reprogram the transaction. Section 7.7 considers more sophisticated kinds of constraints.

7.3 Heterogeneous Databases

The ability to specify the way view updates are to be carried out has important applications in the area of heterogeneous databases.

A *heterogeneous database* is a collection of databases that store similar information in different ways. Such databases often arise when companies merge, or when an institution wants centralized access to a plethora of departmental databases. In such cases, the member databases often retain a large degree of autonomy; so it is not possible to simply merge their data into one large centralized database once and for all. For instance, in a consortium, each company may have its own database, which it controls, as well as access to the data of the other companies. Providing convenient access to the information stored in the disparate member databases is an important issue.

To illustrate some of the problems, consider three databases, $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{D}_3$, with the following schema, which record the same data, but in different ways:

$$\begin{array}{ll}
\mathbf{D}_1: & emp(EmpNum, EmpName, Salary) \\
 & child(EmpNum, ChildName) \\
 & \\
\mathbf{D}_2: & employee(EmpName, EmpNum, Salary) \\
 & dependent(ChildName, EmpNum) \\
 & \\
\mathbf{D}_3: & person(EmpNum, EmpName) \\
 & salary(EmpNum, Salary) \\
 & children(EmpNum, ChildName)
\end{array}$$

To express such situations in $\mathcal{TR}$, we use the predicate symbol $d_i.p$ to denote predicate p in database $\mathbf{D}_i$. Thus, $d_1.p$ and $d_2.p$ denote predicates of the same name stored in different databases. Of course, $d_1.p$ and $d_2.p$ both appear in a single $\mathcal{TR}$ database. In this way, a single logical database can represent a number of heterogeneous databases.

One of the central problems of heterogeneous databases is to present the data in several databases in a way that will allow users and application programs to access and manipulate data without precise knowledge of how this data is distributed, stored or represented in the member databases. They need not even be aware that these databases exists. One common solution to this problem is to let users access data via views that span one or more member databases. If each member database stores the same information, but in a different format, then such a view is easily specified with *existing* database or logic-programming languages. In our example, one could create a view showing each employee and his children, as follows:

$$
\begin{aligned}
child(EmpNum, EmpName, ChildName) &\leftarrow d_1.emp(EmpNum, EmpName, Salary) \wedge \\
&\quad\; d_1.child(EmpNum, ChildName) \\
child(EmpNum, EmpName, ChildName) &\leftarrow d_2.employee(EmpName, EmpNum, Salary) \wedge \\
&\quad\; d_2.dependent(ChildName, EmpNum) \\
child(EmpNum, EmpName, ChildName) &\leftarrow d_3.person(EmpNum, EmpName) \wedge \\
&\quad\; d_3.children(EmpNum, ChildName)
\end{aligned}
$$

Likewise, we can create a unified view of the three databases that shows each employee and his salary, as follows:

$$
\begin{aligned}
salary(Empnum, EmpName, Salary) &\leftarrow d_1.emp(EmpNum, EmpName, Salary) \\
salary(Empnum, EmpName, Salary) &\leftarrow d_2.employee(EmpName, EmpNum, Salary) \\
salary(Empnum, EmpName, Salary) &\leftarrow d_3.person(EmpNum, EmpName) \wedge \\
&\quad\; d_3.salary(EmpNum, Salary)
\end{aligned}
$$

Although views like these allow users and application programs to *see* data in a uniform way, they provide no help in *updating* the data. The two views above are particularly tricky, since they represent union views and most of the rules represent relational joins. Suppose, for instance, that an employee has a new child and that we wish to insert this fact into the view. We do not know about the three underlying databases, $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{D}_3$. We merely want to insert an atom such as $child(1234, alberto, tommy)$ into the view. The database system should handle the details of how to store this information. That is, it should figure out which database to store the information in, and in what format. Physically, however, the database is distributed over three different sites, and we only have the ability to store records at each site. Thus, we cannot store the following disjunction (even if this were acceptable from the user's point of view):

$$d_1.child(1234, tommy) \;\vee\; d_2.dependent(tommy, 1234) \;\vee\; d_3.children(1234, tommy)$$

There are other, related complications too. For instance, if there is no employee called *alberto* with employee number 1234, then the insert should *fail*. In particular, we cannot succeed by storing the following formula, which describes different database states that give rise to the atom $child(1234, alberto, tommy)$ in the view:

$$
\begin{aligned}
& [\,\exists Salary\ d_1.emp(1234, alberto, Salary)\ \wedge\ d_1.child(1234, tommy)\,] \\
\vee\ & [\,\exists Salary\ d_2.employee(alberto, 1234, Salary)\ \wedge\ d_2.dependent(tommy, 1234)\,] \\
\vee\ & [\,d_3.person(1234, alberto)\ \wedge\ d_3.children(1234, tommy)\,]
\end{aligned}
$$

If *alberto* does exist, then the correct solution depends on where the information about *alberto* is stored. If it is stored in database $\mathbf{D}_1$, then we must insert the atom $child(1234, tommy)$ into database $\mathbf{D}_1$; if it is stored at $\mathbf{D}_2$, then we must insert the atom $dependent(tommy, 1234)$ into database $\mathbf{D}_2$; and similarly for $\mathbf{D}_3$. It is also possible that information about *alberto* is stored redundantly, at more than one site. In this case, we may need to insert more than one atom. For instance, we might have to insert $child(1234, tommy)$ into $\mathbf{D}_1$, and $dependent(tommy, 1234)$ into $\mathbf{D}_3$.

These requirements pose no problem for $\mathcal{TR}$, since view updates can be specified directly, as $\mathcal{TR}$-rules. For instance, one can write a transaction that allows a user of the view to add a new tuple whenever an employee has a new child. This transaction queries the member databases to find out where the information about that employee is stored, and then it inserts the information into the correct database in the appropriate format. The rules, below, define a transaction called $addChild(EmpNum, EmpName, ChildName)$ that effectively inserts the tuple $child(EmpNum, EmpName, ChildName)$ into the view. The user must provide the child's name and the employee's number. If the employee's name is left unspecified, then it is returned as an answer. If the user provides the employee name, then it acts as a consistency check. That is, if there is no employee with both the given name and number, then the insertion is not carried out.

$$
\begin{aligned}
addChild(EmpNum, EmpName, ChildName) \leftarrow\ & d_1.emp(EmpNum, EmpName, Salary) \otimes \\
& d_1.child.ins(EmpNum, ChildName) \\
addChild(EmpNum, EmpName, ChildName) \leftarrow\ & d_2.employee(EmpName, EmpNum, Salary) \otimes \\
& d_2.dependent.ins(ChildName, EmpNum) \\
addChild(EmpNum, EmpName, ChildName) \leftarrow\ & d_3.person(EmpNum, EmpName) \otimes \\
& d_3.children.ins(EmpNum, ChildName)
\end{aligned}
$$

The algorithms implementing the transition oracle would execute an expression of the form $d_1.child.ins(empnum, childname)$ as a physical insertion of the atom $child(empnum, childname)$ into database $\mathbf{D}_1$. The important point is that the above rule-base provides the oracle with exactly the right information for deciding which atom to insert and where.

7.4 Simulating Systems with State

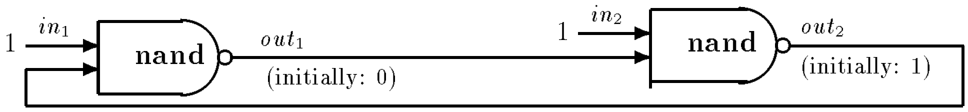
Horn logic is a natural language for simulating certain physical systems. Digital circuits are one example, as long as they have no internal state. Such digital circuits are said to be *combinational*, and their output is a function only of their input. For such circuits, the database can record the input to the circuit, and a logic program can compute the output of the circuit.

Systems with internal state are another matter. Circuits such as counters, shift registers, flip-flops, and computer memories are examples of this. For such systems, the output depends not only on the input, but also on the internal state of the system. A natural way to model internal states is by means of a database, since the persistence of system state is then guaranteed by the persistence of the database state. Moreover, the system state can then be interrogated via database queries, and changed via database updates. Besides its obvious convenience, this approach can be efficiently implemented using existing database technology and indexing methods, which is especially important for large systems. However, this approach to simulation requires that the database be updated as the system is simulated. Unfortunately, since classical logic programming or deductive databases do not embody a clean theory of updates, this natural representation has not been available to logic programmers.

$\mathcal{TR}$ changes this situation. Since it is a logic for updating databases, it is a natural simulation language. $\mathcal{TR}$ has one more advantage in this respect, since it provides not just a specification language, but also a practical proof theory. From a simulation point of view, the important point is that in updating a database, $\mathcal{TR}$ accesses only those parts of the database that change. This property allows $\mathcal{TR}$ to provide *incremental* simulations. These are especially important when simulating large systems. The idea is to simulate only the *changes* to a system's internal state. Little, if any, computational resources should be spent on things that do not change. For instance, in simulating a 1MB VLSI memory chip (2^{20} bytes), if an input instruction causes just one byte of memory to change, then the simulation should not spend any time considering the other $2^{20} - 1$ bytes that do not change. To illustrate these claims, we provide $\mathcal{TR}$ rules for simulating digital circuits with state, and we use them to simulate a flip-flop.

We represent a digital circuit as a collection of devices (*e.g.*, logic gates) connected by electrical lines (*e.g.*, wires). Each device has a set of input and output ports. Each port is attached to one electrical line, and each electrical line may connect two ports. A device called a "split box" divides an electrical line in two, so that the output of one device can be fed to the input of several others. In our representation, each electrical line has a name, denoted by a constant symbol, as well as a value of 0 or 1. The atom $val(L, V)$ means that line L has value V.

To represent a circuit of logic gates, we use the predicate *gate*. There are actually several versions of this predicate, each with a different arity, to represent gates with different numbers of ports. For gates with two inputs and one output, we use the scheme $gate(Type, In_1, In_2, Out)$, where $Type$ is the type of the logic gate (*and*, *or*, etc.), In_1 and In_2 are the names of the two input lines, and Out is the name of the output line. Thus, the atom $gate(and, b, c, d)$ denotes an *and*-gate whose input lines are b and c, and whose output line is d. Likewise, the atom $gate(inv, d, e)$ denotes an inverter gate whose input line is d, and whose output line is e. To handle the multiple arities of *gate*, we use the Prolog convention that predicates of the same name but different arity are deemed to be distinct predicate symbols. To represent split-boxes, we use the predicate $split(In, Out_1, Out_2)$. The structure of a circuit and the values of all its lines are stored in the database, as shown in Example 7.1.



Example 7.1 (Flip-Flops) The following database encodes two *nand*-gates configured as a simple flip-flop:

$$gate(nand, in_1, out_2, out_1) \qquad gate(nand, in_2, out_1, out_2)$$
$$val(in_1, 1) \quad val(in_2, 1) \quad val(out_1, 0) \quad val(out_2, 1)$$

This circuit has two input lines, in_1 and in_2, and two output lines, out_1 and out_2. Note that the output of each gate is connected to the input of the other. This feedback gives the circuit an internal state, so that its output is not just a function of its input, but also of its history. The database records the current state of the circuit, in which both input lines are high, one output line is low, and the other is high. As the behavior of the circuit is simulated in $\mathcal{TR}$, the state is updated. A description of such a simulation is given at the end of the section. □

The first step in simulating such circuits is to represent the truth table of each gate. We do this with a predicate called *table*. Like *gate*, there are several versions of *table*, with different arities, to represent gates with different numbers of ports. For gates with two inputs and one output, we use the scheme $table(Type, In_1, In_2, Out)$, where $Type$ is the type of the logic gate, In_1 and In_2 are the two input values, and Out is the output value. Thus, the atom $table(and, 0, 1, 0)$ says that an *and*-gate with inputs 0 and 1 has an output of 0. Likewise, the atom $table(inv, 1, 0)$ says that an inverter gate with an input of 1 has an output of 0. The truth tables for three types of logic gate are represented by the following rules, which we add to the transaction base.[16]

$$\begin{array}{lll}
table(nand,0,0,1) & table(nor,0,0,1) & table(inv,0,1) \\
table(nand,0,1,1) & table(nor,0,1,0) & table(inv,1,0) \\
table(nand,1,0,1) & table(nor,1,0,0) & \\
table(nand,1,1,0) & table(nor,1,1,0) &
\end{array}$$

To capture the behavior of logic circuits, one needs more than just the truth tables for logic gates. For instance, if one input to a gate changes, then the output may change, but the other input does not change. The truth tables do not provide this kind of dynamic information, since they provide only a static picture. It is convenient to describe the dynamics of logic gates in terms of simple events. The basic event in our description is the change in value of a line. To initiate such an event, a user (or an application program) uses the predicate $set(Line, Val)$, which sets a given line to a given value. Thus, if a user issues the command $?- \; set(b, 1)$, then the system sets the value of line b to 1. This event, like all events, may have ramifications. For instance, setting an input line of an *or*-gate to 1 causes the output line to be set to 1 as well. In this way, one event may trigger other events, which in turn, may trigger still others, etc. Eventually, this cascade of events may settle down, so that the circuit achieves a stable state. In other cases, the circuit may oscillate forever. Such behavior, which is typical of dynamic systems, is captured by the following rules:

$$\begin{array}{rcl}
set(Line, NewVal) & \leftarrow & val(Line, NewVal) \qquad (12) \\
set(Line, NewVal) & \leftarrow & val(Line, OldVal) \otimes (NewVal \neq OldVal) \otimes \qquad (13) \\
 & & \quad val.del(Line, OldVal) \otimes val.ins(Line, NewVal) \otimes \\
 & & \quad propagate(Line)
\end{array}$$

$$\begin{array}{rcl}
propagate(Line) & \leftarrow & dangling(Line) \qquad (14) \\
propagate(In) & \leftarrow & split(In, Out_1, Out_2) \otimes val(In, Val) \otimes \qquad (15) \\
 & & \quad set(Out_1, Val) \otimes set(Out_2, Val) \\
propagate(In) & \leftarrow & gate(inv, In, Out) \otimes val(In, Val_1) \otimes \qquad (16) \\
 & & \quad table(inv, Val_1, Val_2) \otimes set(Out, Val_2)
\end{array}$$

[16]We could equally well store the truth tables as atoms in the database, but then they would not be protected from updates by the user.

$$propagate(In) \leftarrow [gate(type, In, In', Out) \vee gate(type, In', In, Out)] \otimes \quad (17)$$
$$val(In, Val) \otimes val(In', Val') \otimes$$
$$table(type, Val, Val', Val'') \otimes set(Out, Val'')$$

These rules define two mutually recursive predicates, *set* and *propagate*. The command $?- set(Line, NewVal)$ sets the value of a line, and the command $?- propagate(Line)$ propagates the effect of a change in line value to the output(s) of any gate that the line is connected to. The recursion gives rise to a cascade of events. Rule (12) terminates the recursion. This rule represents situations in which the new value of a line is the same as the old value. In this case, there is nothing to do, and no further cascading of events. Rule (13) represents situations in which the new and old line values are different. The rule changes the line value, and then invokes the predicate *propagate*, to propagate the effects of the change.

Propagate is defined by several rules. Rule (14) terminates the recursive cascading of events. It represents situations in which an output line leads nowhere (is dangling) and thus requires no further propagation. Rule (15) encodes the behavior of split boxes, which divide a line in two. The rule sets the two outputs of the split box to the same value as the input. Rule (16) encodes the behavior of inverters, setting the output of an inverter to the inverse of its input. The last rule, (17), represents the behavior of logic gates with two inputs.

Simulating A Flip-Flop

We illustrate the behavior of these rules on the flip-flop of Example 7.1. As given in the example, both input lines are high, one output line is low, and the other is high. Thus, the initial state of the circuit is represented by the following database entries:

$$val(in_1, 1) \quad val(in_2, 1) \quad val(out_1, 0) \quad val(out_2, 1) \quad (18)$$

At this point, if the user issues the command $?- set(in_1, 1)$, then nothing should happen, since input line in_1 is already 1. The simulation reflects this behavior. The command causes Rule (12) to execute successfully. The rule checks that the line value has not changed, and then returns.

The behavior is more interesting if the user issues the command $?- set(in_1, 0)$, which changes the value of line in_1. The first effect of this command is to invoke Rule (13), which changes the line value, and then issues the command $?- propagate(in_1)$ as a subtransaction. This subtransaction propagates the effect of the change in line in_1 through the *nand*-gate to its output line, out_1. Specifically, rule (17) is invoked, which computes the new value of the output line, and then sets it by issuing the command $?- set(out_1, 1)$. This behavior is summarized by the following series of subgoals, which reflects inference system $\Im^I$ operated in top-down mode. (A more detailed trace is given in the next subsection):

$$?- set(in_1, 0)$$
$$?- propagate(in_1)$$
$$?- set(out_1, 1)$$

Notice that the first and last subgoals in this series both involve the predicate *set*. In this way, the first event, setting line in_1 to 0, has triggered a second event, setting line out_1 to 1. The latter event is carried out and propagated, which triggers a third event, setting line out_2 to 1, which is the last event in the simulation. The entire simulation is summarized by the following series of subgoals:

$$
\begin{array}{l}
?- \ set(in_1, 0) \\
?- \ propagate(in_1) \\
?- \ set(out_1, 1) \\
?- \ propagate(out_1) \\
?- \ set(out_2, 0) \\
?- \ propagate(out_2) \\
?- \ set(out_1, 1) \\
?- \ ()
\end{array}
$$

The simulation terminates because the last event has no further effect on the circuit. This is because the line that was changed, out_1, is already high due to an earlier *set*-action. Note that termination of the inference process is indicated by the empty subgoal, (). At the end of this simulation, the new state of the circuit is recorded in the database by the following entries:

$$val(in_1, 0) \quad val(in_2, 1) \quad val(out_1, 1) \quad val(out_2, 0) \tag{19}$$

Thus, we have lowered one of the inputs to the flip-flop, and the output has change state, *i.e.*, both output lines have changed their value. If we now return the input to its original state, by issuing the command $?- \ set(in_1, 1)$, then the output does *not* change back. In this way, $\mathcal{TR}$ simulates the dependence of the output on *history* as well as on input. The second simulation is summarized by the following series of subgoals:

$$
\begin{array}{l}
?- \ set(in_1, 1) \\
?- \ propagate(in_1) \\
?- \ set(out_1, 1) \\
?- \ ()
\end{array}
$$

The simulation terminates with the attempt to set the value of line out_1 to 1. Since its value is already 1, no further propagation is necessary. In terms of inference, the subgoal $?- \ set(out_1, 1)$ invokes rule (12), which simply returns. At the end of this simulation, the new state of the circuit is recorded in the database by the following entries:

$$val(in_1, 1) \quad val(in_2, 1) \quad val(out_1, 1) \quad val(out_2, 0) \tag{20}$$

Notice that in databases (18) and (20), the input lines have the same values, but the output lines do not.

The Simulation in Detail

A detailed trace of part of the above simulation is as follows. The circuit is initially in state (18), and the user issues the command $?- \ set(in_1, 0)$, which changes the value of line in_1. The first effect of the command is to invoke Rule (13), which changes the line value, and then issues the command $?- \ propagate(in_1)$ as a subtransaction. This behavior is summarized by the following series of subgoals, which reflects inference system $\Im^I$ operated in top-down mode (Section 6.3). Recall that atoms within a serial goal are resolved from left to right. As each atom is resolved, the database is queried and/or updated.

$$?-\ set(in_1, 0)$$
$$?-\ val(in_1, OldVal) \otimes (0 \neq OldVal) \otimes val.del(in_1, OldVal) \otimes val.ins(in_1, 0) \otimes propagate(in_1)$$
$$?-\ (0 \neq 1) \otimes val.del(in_1, 1) \otimes val.ins(in_1, 0) \otimes propagate(in_1)$$
$$?-\ val.del(in_1, 1) \otimes val.ins(in_1, 0) \otimes propagate(in_1)$$
$$?-\ val.ins(in_1, 0) \otimes propagate(in_1)$$
$$?-\ propagate(in_1)$$

The last subgoal in this series propagates the effect of the change in line in_1 through the *nand*-gate to its output line, out_1. This takes place in several steps. First, the rules defining *propagate* are invoked. Because in_2 is the first input line of a *nand*-gate, these rules all fail immediately, except for Rule (17). This latter rule retrieves both input values of the *nand*-gate, $Val_1 = 0$ and $Val_2 = 1$, computes the new output value, $Val_3 = 1$, and finally issues the subtransaction $?-\ set(out_1, 1)$ to set the output line to its new value. This behavior is summarized by the following series of subgoals:

$$?-\ propagate(in_1)$$
$$?-\ gate(Type, in_1, In_2, Out) \otimes val(in_1, Val_1) \otimes val(In_2, Val_2) \otimes$$
$$table(Type, Val_1, Val_2, Val_3) \otimes set(Out, Val_3)$$
$$?-\ val(in_1, Val_1) \otimes val(out_2, Val_2) \otimes table(nand, Val_1, Val_2, Val_3) \otimes set(out_1, Val_3)$$
$$?-\ val(out_2, Val_2) \otimes table(nand, 0, Val_2, Val_3) \otimes set(out_1, Val_3)$$
$$?-\ table(nand, 0, 1, Val_3) \otimes set(out_1, Val_3)$$
$$?-\ set(out_1, 1)$$

The last subgoal in this series sets the value of line out_1 to 1. In this way, the first event, setting line in_1 to 0, has triggered another event, setting line out_1 to 1. This latter event is carried out and propagated further, which triggers the third event that sets the line out_2 to 1. After this, the line out_1 is set to 1, which terminates the simulation, as this line is already high.

7.5 Bulk Updates

The ability to perform bulk updates is one of the cornerstones of database languages. It is routine in such database *lingua franca* as SQL or QUEL. For example, inserting a set of tuples into a relation is a basic SQL operation, as is deleting a set of tuples from a relation. Yet, bulk updates like these are conspicuously absent from most logic-based proposals for updating logic programs. The few exceptions are [23, 28, 77, 5], which are discussed in Section 11. The unusual difficulty with this kind of update seems to arise because most logical formulations of updates are based on the insertion and deletion of *single* tuples. This is not how SQL works, however. SQL first computes a query and then inserts the resulting *set* of tuples into a relation. Deletion is handled in a similar fashion.

To capture such behaviour, it appears that we need an elementary state transition that accomplishes bulk updates at the lowest level. In this section, we consider *relational assignment*, which copies the contents of one relation into another relation. Just as variable assignment is a basic operation of procedural programming languages, relational assignment can be used as a basic operation of procedural database languages [26, 27]. The rest of this section shows how to express and use relational assignment in $\mathcal{TR}$.

Unlike [69, 77, 28], specific elementary transitions are not built into the semantics of $\mathcal{TR}$. Relational assignment can therefore be added to $\mathcal{TR}$ by simply expanding transition oracle. To see how, consider a $\mathcal{TR}$ language, $\mathcal{L}$. For every pair of predicate symbols, r and q, of the same arity, let $\mathcal{L}$ contain a propositional constant, denoted $[r := q]$. For simplicity, we restrict our attention to deductive databases in which r is an extensional (*i.e.*, non-derived) predicate. If $\mathbf{D}$ is such a database, then let $\mathbf{D}'$ be the database derived from $\mathbf{D}$ by deleting all r-facts and replacing them by the set $\{r(t_1, \ldots, t_n) \mid \mathbf{D} \models q(t_1, \ldots, t_n)\}$ (*i.e.*, first delete the contents of relation r, and then copy the contents of relation q into r.) Finally, let the transition oracle specify a transition $[r := q] \in \mathcal{O}^t(\mathbf{D}, \mathbf{D}')$ for every such pair of databases. Thus, the transition oracle specifies the following:

$$
\begin{aligned}
&[r := q] \in \mathcal{O}^t(\{q(a), q(b)\},\ \ \{q(a), q(b), r(a), r(b)\}) \\
&[r := q] \in \mathcal{O}^t(\{q(a), q(b), r(c)\},\ \ \{q(a), q(b), r(a), r(b)\}) \\
&[r := q] \in \mathcal{O}^t(\{q(a), q(b), r(c), r(d)\},\ \ \{q(a), q(b), r(a), r(b)\})
\end{aligned}
$$

Note that, by definition, the extent of r after executing the elementary state transition $[r := q]$ is determined entirely by $\mathbf{D}$, the current database state. This has the following important implication: if q is defined by a set of rules where some of the rules are in the database $\mathbf{D}$ and some are in the transaction base $\mathbf{P}$, then only the tuples contributed by the rules in $\mathbf{D}$ are assigned to r.

Having defined relational assignment, we can easily define bulk inserts and deletes. Suppose we wanted to add to r all tuples satisfying some condition ϕ and delete from $\mathbf{D}$ all tuples satisfying ψ. To do so, for each of these operations, we first define two derived relations, $q1$ for the insertion into r, and $q2$ for the deletion from $\mathbf{D}$, as follows:

$$
\begin{aligned}
q1(X) &\longleftarrow r(X) \\
q1(X) &\longleftarrow \phi(X) \\
q2(X) &\longleftarrow s(X) \wedge \neg\psi(X)
\end{aligned}
\tag{21}
$$

Note that the extension of $q1$ is $r \cup \phi$, and the extension of $q2$ is $s - \psi$. To actually perform the updates, we use the elementary transitions $[r := q1]$ and $[s := q2]$, which effectively insert the tuples satisfying ϕ into r, and delete the tuples satisfying ψ from $\mathbf{D}$. In this way, relational assignment captures the update behaviour of SQL, including the use of existential subqueries to perform bulk deletion. It should be clear from an earlier remark that the above rules must all be in $\mathbf{D}$ for the relational assignment to work as intended.

More-complex transactions are also easy to express. For example, consider the transaction, "*Raise the salary of all managers by 7%, and then retrieve all employees whose salary is greater than 100K.*" This transaction can be expressed as follows:

$$
\begin{aligned}
&empl2(E, Sal * 1.07, mngr) \longleftarrow empl(E, Sal, mngr) \\
&empl2(E, Sal, Rank) \longleftarrow empl(E, Sal, Rank) \wedge Rank \neq mngr \\
&result(E) \longleftarrow [empl := empl2] \otimes empl(E, Sal, Rank) \otimes Sal > 100K
\end{aligned}
\tag{22}
$$

The new contents of the employee relation is computed by the first two rules and is held temporarily in the relation $empl2$. As explained above, both these rules must be in the database, $\mathbf{D}$, in order for the assignment $[empl := empl2]$ in the third rule to work as intended. Note that the query $?-\ result(E)$ changes the database state *and* returns all suitable employees as the answer. Observe also that this query simultaneously involves deletion of some tuples with old salaries *and* insertion of tuples with new salaries. Of course, this combined transaction could have been expressed following the methodology for deletions and insertions described earlier, in (21). However, this would have required two relational assignments instead of one. In the above example, we defined the temporary relation *empl*2 in such a way that only one relational assignment is needed.

We should also note that when the auxiliary predicates (such as $q1$, $q2$, *empl*2 above) are non-recursive, it is possible to do away with these predicates and their defining rules. To this end, we can define the following, more general, form of relational assignment: $[q := (\bar{X}).\phi]$. Here ϕ is a first-order formula all of whose predicates are in $\mathbf{D}$ and $(\bar{X})$ is a list of all free variables in ϕ (with possible repetitions). We assume that the length of $\bar{X}$ equals the arity of q. This elementary state transition has the effect of assigning q the relation $\{\bar{x} \mid \phi[\bar{x}] \text{ is true}\}$ — the set of all tuples that when substituted for $\bar{X}$ makes ϕ true.[17] We can now rewrite (22) as follows:

$$result(E) \longleftarrow [empl := (E, Sal, Rank).\phi] \otimes empl(E, Sal, Rank) \otimes Sal > 100K$$

where ϕ is the following first-order formula:

$$\exists S[empl(E, S, mngr) \wedge Sal = 1.07 * S]$$
$$\vee[empl(E, Sal, Rank) \wedge Rank \neq mngr]$$

These generalized bulk updates have all the power of bulk updates in SQL, including subqueries. This is because a generalized bulk update computes an arbitrary first-order query, ϕ, and assigns its output to a base relation, q. As a special case, a bulk update can change the value of q to $q \cup \phi$, thereby expressing arbitrary SQL insertions. Likewise, a bulk update can change the value of q to $q - \phi$, thereby expressing arbitrary SQL deletions.

Finally, it is worth noting that the use of generalized bulk updates in $\mathcal{TR}$ closely parallels the embedding of SQL in procedural programming languages. In both cases, bulk updates are elementary operations invoked from a host language. And in both cases, these bulk updates can have free variables (parameters) that are bound at run time. Furthermore, in an update like $[q := (\bar{X})\phi]$, the base relation q can play the role of a *cursor*. Using the methods introduced in Section 8.4.2, one can iterate over the relation q one tuple at a time, just like an SQL cursor. $\mathcal{TR}$ can thus be seen as a formal basis for embedded SQL.

7.6 Non-Deterministic Sampling

In [61], Krishnamurthy and Naqvi proposed the so-called *choice*-operator to extend LDL, so that one can express non-deterministic queries such as this: "Produce a sample of one employee from each department." The idea was to add a special predicate, $choice((\bar{X}), (\bar{Y}))$, that selects those instantiations of the variables $\bar{X}$ and $\bar{Y}$ that satisfy the functional dependency (abbr., FD) $\bar{X} \rightarrow \bar{Y}$.

The meaning of a program, $\mathbf{P}$, with occurrences of the *choice*-operator was originally given in [61, 78] as follows: first, $\mathbf{P}$ is rewritten into a regular Datalog program, $\mathbf{P}'$, in which each occurrence of the *choice*-operator is replaced by a normal predicate called, say, $choices_i$. $choice_i$ is uniquely associated with a particular occurrence of the choice operator in the original program, $\mathbf{P}$, (where the subscript identifies the occurrence). Models of $\mathbf{P}$ are then defined in terms of Herbrand models of $\mathbf{P}'$.

[17] If ϕ is a disjunction of existentially quantified conjuncts of positive literals then "truth" can be taken to mean truth in all models of ϕ. Otherwise, if literals can be negative, truth should be viewed with respect to a canonic model of ϕ, such as the one described in Section 9.

In particular, for each classical (minimal) Herbrand model, $\mathbf{M}$, of $\mathbf{P}'$, consider those subsets of $\mathbf{M}$ in which the relation assigned to $choices_i$ satisfies the appropriate FD's. A maximal subset of this kind is said to be a model of $\mathbf{P}$.

Later, Sacca and Zaniolo [89] proposed a somewhat simpler rewriting for the choice operator, one in which the semantics can be given as a stable model of the rewritten program. The rewritten program uses negation even if the original program was Horn, which is where stable models come in. The use of stable models is essential, since *choice* does not get a correct interpretation under, say, the well-founded semantics. In other words, incorporating the choice operator *commits* a user to the stable model semantics of negation.

There is a sense, however, in which both accounts are unsatisfactory, since, in these proposals, the *choice* predicate has no status in the logic itself. That is, *choice* cannot be thought of as a normal predicate that is true on some tuples and false on others. To understand the meaning of such a program, the user has to perform a non-local mental transformation of the original program into another, textually more complex, program, and then to try and determine the meaning of that latter program. In our opinion, incorporating new and conceptually-simple functionality via a serious of non-trivial transformations will not win new converts to declarative programming.

In contrast to these approaches, the choice construct in $\mathcal{TR}$ can be understood *directly* in terms of $\mathcal{TR}$'s basic logical concepts. In $\mathcal{TR}$, the *choice*-operator is represented as another kind of elementary bulk update, one closely related to the relational assignment operator of the previous section. For example, consider the following query:

For each department with over 100 employees, choose an employee earning less than \$20K.

With the choice operator, this query could be written as follows:

$$\begin{aligned} answer(D,E) \longleftarrow \quad & dept(E,D) \ \wedge \ size(D,N) \ \wedge \ choice((D),(E)) \wedge \\ & N > 100 \ \wedge \ salary(E,S) \ \wedge \ S < 20K \end{aligned} \tag{23}$$

We can represent this query in $\mathcal{TR}$ using a pair of rules, as follows:

$$\begin{aligned} & eligible(D,E) \longleftarrow dept(E,D) \ \wedge \ size(D,N) \ \wedge \ N > 100 \ \wedge \ salary(E,S) \ \wedge \ S < 20K \\ & answer(D,E) \longleftarrow [sample \overset{1 \rightarrow 2}{:=} eligible] \otimes sample(D,E) \end{aligned} \tag{24}$$

The first rule produces a set of candidate answers, by selecting *all* eligible employees from the appropriate departments. The second rule then samples the set of candidates, selecting *one* eligible employee per department. The heart of this rule is a new type of elementary update, $[sample \overset{1 \rightarrow 2}{:=} eligible]$, that sets the extent of relation *sample* to be a subset of relation *eligible*. Any subset will do as long as it has the following two properties:

- it satisfies the specified FD, $1 \rightarrow 2$; and
- $sample[1] = eligible[1]$, *i.e.*, the projections of *sample* and *eligible* on the first attribute are equal (and thus every department is represented in the sample).

In general, we introduce an elementary update, called *sampling assignment*, denoted $[p \overset{fd}{:=} q]$, where fd is an FD, and p and q are predicate symbols of the same arity (where p is an extensional predicate). Given a database, $\mathbf{D}$, the assignment updates relation p non-deterministically. In particular, it sets the extent of p to be some subset, rel, of the relation $\{\, \bar{x} \mid \mathbf{D} \models q(\bar{x}) \,\}$ that satisfies the following two properties:

1. *rel* satisfies the functional dependency fd; and

2. *rel* is a *maximal* subrelation of $\{\bar{x} \mid \mathbf{D} \models q(\bar{x})\}$ having property 1.

Note that, as with bulk updates, only that part of the extent of q that is determined by $\mathbf{D}$ plays a role in sampling. Thus, for (24) to work correctly, the rule that defines *eligible* must be in the database, $\mathbf{D}$, not in the transaction base $\mathbf{P}$.

As with bulk updates, sampling assignments are defined in the transition oracle. For instance, to define $[p \overset{1 \to 2}{:=} q]$, we add the following entries to the transition oracle (among many others):

$$[p \overset{1 \to 2}{:=} q] \in \mathcal{O}^t(\{q(a,b), q(a,c), q(a,d)\}, \quad \{q(a,b), q(a,c), q(a,d), p(a,b)\})$$
$$[p \overset{1 \to 2}{:=} q] \in \mathcal{O}^t(\{q(a,b), q(a,c), q(a,d)\}, \quad \{q(a,b), q(a,c), q(a,d), p(a,c)\})$$
$$[p \overset{1 \to 2}{:=} q] \in \mathcal{O}^t(\{q(a,b), q(a,c), q(a,d)\}, \quad \{q(a,b), q(a,c), q(a,d), p(a,d)\})$$

As in the previous section, we can generalize the *sampling* assignment by allowing first-order formulas on the right-hand side (rather than just predicates symbols). This would allow us to write (24) more succinctly, as follows:

$$answer(D,E) \longleftarrow [sample \overset{1 \to 2}{:=} (D,E).\phi] \otimes sample(D,E)$$

where ϕ is the following first-order formula:

$$\exists N \exists S \; dept(E,D) \;\wedge\; size(D,N) \;\wedge\; N > 100 \;\wedge\; salary(E,S) \;\wedge\; S < 20K$$

The advantage of this new form is that here the *choice*-assignment, $[sample \overset{1 \to 2}{:=} (D,E).\phi]$, directly corresponds to the *choice*-operator in (23). It can therefore be said that the *choice*-operator has finally been given a declarative semantics within a full-fledged logic.

In sum, Transaction Logic incorporates the *choice*-operator via its transition oracle. The model theory of Section 5 provides a declarative account of *choice*, and the proof theory of Section 6 generates choice-relations automatically, as it processes transactions and queries. The model theory of *choice* in $\mathcal{TR}$ is close to its procedural semantics, which happens to be much simpler than the "declarative" semantics presented in [61, 89].

7.7 Dynamic Constraints on Transaction Execution

Because transactions are defined on paths, it is possible to express a large variety of constraints on the way they execute. For instance, we can place conditions on the state of the database during transaction execution, or we may forbid certain sequences of states. We refer to such conditions as *path constraints*, or *dynamic constraints*. Such constraints are particularly well suited to areas such as planning and design, where it is common to place constraints on the way things are done. This section illustrates a variety of dynamic constraints expressible in $\mathcal{TR}$. These include temporal constraints in the style of James Allen [7], such as, "immediately after," "some time after," "during," "at the start of," or "at the end of."

There are several important problems related to constraints. One such problem, and the main subject of this section, is *constraint satisfaction*. That is, given a transaction and a constraint, we want to execute the transaction in such a way that it satisfies the constraint. For example, we might ask a robot to carry out a task while not entering restricted areas and not executing certain (dangerous) sequences of action. Likewise, we might ask a CAD system to run an electrical line from one point to another while avoiding wet or exposed areas. In general, starting from the current database, we want to find *some* way of executing a transaction while satisfying constraints.

A complementary problem is that of constraint *verification*. That is, given a transaction and a constraint, we ask whether *every* execution of the transaction will satisfy the constraint (for any initial database). For example, we might ask whether $X = 1$ whenever a transaction terminates. Note that such questions are hypothetical and that the transaction is *not* actually executed. Instead, we are asking about what *would* happen *if* the transaction were executed. For this reason, we must postpone a discussion of the verification problem until after Section 8, where $\mathcal{TR}$ is augmented with hypothetical operators. The present section thus focuses on constraint satisfaction.

Constraint satisfaction problems are particularly easy to express in $\mathcal{TR}$ because they correspond to classical conjunction. That is, if ψ and ϕ are transaction formulas, then the formula $\psi \wedge \phi$ means, "Do transaction ψ in such a way that ϕ is satisfied on the execution path." The formula ϕ thus constrains the way in which ψ executes. Intuitively, if ψ is a non-deterministic transaction, then ϕ acts as a filter, removing unwanted execution paths, and reducing the non-determinism of the transaction. If ψ is deterministic, then ϕ acts as a guard, forbidding execution unless the constraints are satisfied. Note that in either case, it is execution *paths* that are constrained.

Two types of path constraint naturally arise in $\mathcal{TR}$: those based on serial conjunction, and those based on serial implication. The former specify that something must be true *somewhere* on a path, and the latter specify that something must be true *everywhere* on a path. These two types of path constraint correspond roughly to two types of database integrity constraint: those based on existential quantification, and those based on universal quantification, respectively. We give examples of both.

The examples center around a planning system for robot navigation. The system is composed of rules defining an action, $goto(Y)$, that instructs the robot to go to location Y. This action is highly non-deterministic since, in general, there will be many ways in which it can be carried out, *i.e.*, many routes that the robot can take. By using dynamic constraints, we can force the planning system to reject certain routes or to focus its attention on others.

It should be noted that since constraints involve classical conjunction, they do not satisfy the serial-Horn conditions of Definition 6.2. Thus, they cannot be solved by the specialized proof theory developed in Section 6. Instead, constraints require the general proof theory developed in [21].

7.7.1 Constraints Based on Serial Conjunction

A simple constraint might require the robot to do something while en route to some location, such as passing certain check points. There are two natural cases to this problem. In the first case, the constraint imposes an order on the way the robot does things. The constrained transaction can then be expressed as a sequence of goals. For instance, suppose we request the robot to go to room A in a building and to pass through room B along the way. This request can be expressed as the serial goal $goto(roomB) \otimes goto(roomA)$, *i.e.*, "go to room B and then go to room A."

In the second, and more interesting case, the constraint does not imply an order on the way things are done. For instance, suppose we request the robot to go to room A and to pass through rooms A_1, A_2 and A_3 along the way. This request does not commit the robot to visiting the rooms in a particular order, and so it cannot be expressed as a single, serial goal. Instead, it is properly expressed as a *conjunction* of serial goals, where each conjunct constrains the robot to pass through a particular room.

To express such constraints, we need the propositional constant path, introduced in Section 5.2. Recall that path is true on every path. The formula $\mathtt{path} \otimes at(X)$ thus specifies a path in which the robot ends up at location X. Likewise, the formula $at(X) \otimes \mathtt{path}$ specifies a path in which the robot starts off at location X. Finally, the formula $\mathtt{path} \otimes at(X) \otimes \mathtt{path}$ specifies a path in which the robot passes through location X. For convenience, we abbreviate this latter formula as $go\_thru(X)$ by adding the following rule to the transaction base:

$$go\_thru(X) \leftarrow \mathtt{path} \otimes at(X) \otimes \mathtt{path}$$

It is now easy to specify paths in which the robot must pass through any number of locations, without specifying an order. For instance, the following formula specifies that the robot go to room A, passing through rooms A_1, A_2 and A_3 along the way:

$$goto(roomA) \ \wedge \ go\_thru(roomA_1) \ \wedge \ go\_thru(roomA_2) \ \wedge \ go\_thru(roomA_3)$$

The use of classical conjunction ensures that this formula is true only on paths in which all four conjuncts are true. In this way, the formulas $go\_thru(X)$ constrain the way in which the transaction $goto(roomA)$ may execute.

Note that the path-based semantics of $\mathcal{TR}$ is crucial for the constraints to work. For instance, an arc-based semantics, where formulas are evaluated on pairs of states, cannot constrain the intermediate states of transaction execution.

We can build up more complex constraints by combining classical and serial conjunction. For instance, the following formula requests that the robot go to room A and that along the way it pass first through rooms B_1 and B_2, in any order, and then through rooms C_1 and C_2, in any order:

$$goto(roomA) \ \wedge \ ([go\_thru(roomB_1) \ \wedge \ go\_thru(roomB_2)] \otimes [go\_thru(roomC_1) \ \wedge \ go\_thru(roomC_2)])$$

We can also express the dual of this constraint, requesting the robot to pass through rooms B_1 and B_2 *in that order*, and to pass through rooms C_1 and C_2 *in that order*, but with no constraints on the relative order of the B_i's and the C_i's:

$$goto(roomA) \ \wedge \ [go\_thru(roomB_1) \otimes go\_thru(roomB_2)] \ \wedge \ [go\_thru(roomC_1) \otimes go\_thru(roomC_2)]$$

These constraints all request that the robot do something at *some* point en route to its final destination. Such constraints are existential in nature and are easily and naturally combined with existential quantification. For example, the following formula specifies that the robot go to room A, passing through a red room and a blue room along the way:

$$goto(roomA) \ \wedge \ \exists X[room(X) \otimes red(X) \otimes go\_thru(X)] \ \wedge \ \exists X[room(X) \otimes blue(X) \otimes go\_thru(X)]$$

The examples of this section all fit a certain abstract pattern: A transaction ψ is executed so that certain other actions ϕ_1, ϕ_2, ..., ϕ_n are carried out at some time during the execution. This pattern can be roughly expressed by the following formula:

$$\psi \ \wedge \ [\textstyle\bigwedge_{i=1}^{n} \mathtt{path} \otimes \phi_i \otimes \mathtt{path}]$$

The next section considers constraints that fit a very different pattern.

7.7.2 Constraints Based on Serial Implication

In this section, we consider constraints based on the binary connectives "$\Leftarrow$" and "$\Rightarrow$" introduced in Section 5.2. Recall that $\psi \Leftarrow \phi$ is defined to be $\psi \oplus \neg\phi$, and $\phi \Rightarrow \psi$ is defined to be $\neg\phi \oplus \psi$. Intuitively, the formula $\phi \Rightarrow \psi$ means that transaction ψ must come immediately after transaction ϕ; or more precisely, *whenever* ϕ occurs, then ψ occurs just after it. The formula $\psi \Leftarrow \phi$ is a kind of dual. It says that, in any valid scenario, whenever ϕ occurs, then ψ must have occurred just before it.

Constraints based on serial implication can constrain a transaction during every moment of its execution. For instance, we can request a robot to remain inside a particular region while executing a task. We can also put constraints on specific actions that the robot might take. For instance, we might request a plan for a sequence of robot actions subject to the following constraints:

(i) Open doors before passing through them.

(ii) Shut doors after passing through them.

(iii) Before leaving a room, turn off all the lights.

(iv) After entering a room, turn on all the lights.

(v) Unlock the rifle before firing it.

(vi) Lock and reload the rifle after firing it.

In these examples, "before" and "after" mean "immediately before" and "immediately after," respectively. Serial implication expresses these two relations. For example, constraints (v) and (vi) are expressed by the following two formulas, respectively:[18]

$$unlock \Leftarrow shoot \qquad\qquad shoot \Rightarrow lock \otimes load$$

Actually, we need a little more, since the constraints above are not only temporal, they are also universally quantified. For example, constraint (vi) means, "*Whenever* you shoot the rifle, you must lock and reload it." The problem is that serial implication by itself does not fully capture the universal quantification expressed in the word "whenever." To see this, suppose that ψ is a transaction that plans a series of robot movements and rifle shots. Then the formula $\psi \wedge (shoot \Rightarrow lock \otimes load)$ effectively breaks ψ into two halves: If the first half is a shooting action, then the second half must be a locking action followed by loading. In other words, if the robot fires the rifle at the beginning of the transaction, then it must immediately lock the rifle *and* terminate the transaction. This is not what we had in mind.

To rectify the problem, we use the proposition $\mathtt{path}$. Since it is true on *any* path, it effectively achieves universal quantification over subpaths. For instance, the formula $\mathtt{path} \otimes shoot$ specifies any path that ends in a shooting action. Likewise, the formula $lock \otimes load \otimes \mathtt{path}$ specifies any path that begins with a locking and loading action. The following formula thus expresses constraint (vi), above:

$$\psi \;\wedge\; [\mathtt{path} \otimes shoot \;\Rightarrow\; lock \otimes load \otimes \mathtt{path}] \tag{25}$$

[18] As our examples become more and more complex, correct parenthesizing becomes crucial. We adopt the convention that negation has the strongest binding. Then come the serial connectives $\otimes$ and $\oplus$, followed by $\Leftarrow$ and $\Rightarrow$. The classical connectives $\wedge$ and $\vee$ come next, and $\leftarrow$ comes last.

Observe that the formula $\texttt{path} \otimes shoot$ allows the constraint to apply to any shot, not just to shots that occur at the beginning of the transaction ψ. Likewise, the formula $lock \otimes load \otimes \texttt{path}$ allows the transaction to continue after the rifle is locked and loaded. This is exactly what we had in mind.

Equation (25) has a number of equivalent forms. The next formula below is one of them. By using this form of the constraint, we separate the temporal relation ($shoot \Rightarrow lock \otimes load$) from the use of the symbol path, thus encapsulating it within a single subformula.

$$\psi \ \wedge \ [\texttt{path} \ \Rightarrow \ (shoot \Rightarrow lock \otimes load) \ \otimes \ \texttt{path}]$$

Note the use of "$\texttt{path} \Rightarrow$" (which is also "$\neg\texttt{path} \otimes$") in the constraint. It ensures that the rest of the constraint holds on *every suffix* of the execution path of action ψ. Likewise, in the constraints below, the use of "$\Leftarrow \texttt{path}$" forces the rest of the constraint to hold on *every prefix* of the execution path.

As another example, we express constraint (v) using the opposite serial implication, "$\Leftarrow$". The following, equivalent formulas both ask the robot to perform a series of rifle shots ψ and to unlock the rifle just before each shot:

$$\psi \ \wedge \ [\texttt{path} \otimes unlock \ \Leftarrow \ shoot \otimes \texttt{path}]$$

$$\psi \ \wedge \ [\texttt{path} \otimes (unlock \Leftarrow shoot) \Leftarrow \texttt{path}]$$

These ideas also apply to some apparently non-temporal constraints. For instance, we might ask the robot to stay in room A while executing task ψ. This constraint can be viewed as a temporal relation $at(roomA)$ that has no pre-condition (*i.e.*, it must always be true). It can therefore be expressed by either of the following formulas:

$$\psi \ \wedge \ [\texttt{path} \ \Rightarrow \ at(roomA) \otimes \texttt{path}]$$

$$\psi \ \wedge \ [\texttt{path} \otimes at(roomA) \ \Leftarrow \ \texttt{path}]$$

Finally, we give formulas expressing the first four constraints listed above. Each constraint has the form $\texttt{path} \Rightarrow \phi \otimes \texttt{path}$ or $\texttt{path} \otimes \phi \Leftarrow \texttt{path}$. All variables are universally quantified at the top level, and the predicates involved should be self-explanatory:

(i) $\texttt{path} \otimes \forall D\,[open(D) \ \Leftarrow \ go\_thru(D) \otimes door(D)] \Leftarrow \texttt{path}$

(ii) $\texttt{path} \Rightarrow \forall D\,[door(D) \otimes go\_thru(D) \ \Rightarrow \ shut(D)] \otimes \texttt{path}$

(iii) $\texttt{path} \otimes \forall R, L\,[turn\_off(L) \ \Leftarrow \ leave(R) \otimes room(R) \otimes light(L) \otimes in(L,R)] \Leftarrow \texttt{path}$

(iv) $\texttt{path} \Rightarrow \forall R, L\,[room(R) \otimes light(L) \otimes in(L,R) \otimes enter(R) \ \Rightarrow \ turn\_on(L)] \otimes \texttt{path}$

So far, this section has focussed on the temporal relations "immediately before" and "immediately after." However, $\mathcal{TR}$ can express many other temporal relations in the style of James Allen's theory of time intervals [7]. These relations include "some time before," "some time after," "during," "at the start of," "at the end of," etc. The table below lists a variety of temporal relations and the transaction formulas used to express them. Each of these formulas is an implication stating that if an action satisfying α occurs, another action, satisfying β, must occur in a certain relationship to it. These constraints can be applied to a particular path or to all its subpaths. As written, each formula, ψ, constrains a single path. By using the formula $\texttt{path} \Rightarrow \psi \Leftarrow \texttt{path}$, however, the constraint ψ applies to all the subpaths.

Formula	Constraint
$\beta \Leftarrow \alpha$	β occurs *immediately* before α
$\alpha \Rightarrow \beta$	β occurs *immediately* after α
$(\beta \otimes \mathtt{path}) \Leftarrow \alpha$	β occurs *some* time before α
$\alpha \Rightarrow (\mathtt{path} \otimes \beta)$	β occurs *some* time after α
$\alpha \rightarrow (\mathtt{path} \otimes \beta \otimes \mathtt{path})$	β occurs on *some* interval during α
$\alpha \rightarrow (\mathtt{path} \Rightarrow \beta \Leftarrow \mathtt{path})$	β occurs on *every* interval during α
$\alpha \rightarrow (\beta \otimes \mathtt{path})$	β occurs on *some* initial segment of α
$\alpha \rightarrow (\mathtt{path} \otimes \beta)$	β occurs on *some* final segment of α
$\alpha \rightarrow (\beta \Leftarrow \mathtt{path})$	β occurs on *every* initial segment of α
$\alpha \rightarrow (\mathtt{path} \Rightarrow \beta)$	β occurs on *every* final segment of α
$(\alpha \otimes \mathtt{path}) \rightarrow (\beta \otimes \mathtt{path})$	β starts at the same time as α
$(\mathtt{path} \otimes \alpha) \rightarrow (\mathtt{path} \otimes \beta)$	β ends at the same time as α

There is a subtlety concerning constraints that involve implication, whether serial or classical. For instance, if a path $\mathbf{D}_0 \ldots \mathbf{D}_n$ is constrained by the formula $\alpha \rightarrow \beta$, we do *not* mean that the formula must be logically entailed on the the path. That is, we do not intend the following statement to be true: $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \rightarrow \beta$. This is too strong a requirement, for it requires $\alpha \rightarrow \beta$ to be true on the path for *all* models of $\mathbf{P}$, even those where α is made true completely arbitrarily. However, the intended meaning of an implicational constraint is that the following statement must be true:

$$\textit{if} \quad \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \qquad \textit{then} \quad \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$$

This is a weaker requirement. Constraints involving serial implication are interpreted similarly: the constraint $\mathtt{path} \otimes \mathit{shoot} \Rightarrow \mathit{load} \otimes \mathtt{path}$ is satisfied on the path $\mathbf{D}_0 \ldots \mathbf{D}_n$ if and only if the following statement is true:

$$\begin{array}{lll} \textit{if} & \mathbf{P}, \mathbf{D}_i \ldots \mathbf{D}_j \vdash \mathit{shoot} & \textit{for some } i \textit{ and } j, \\ \textit{then} & \mathbf{P}, \mathbf{D}_j \ldots \mathbf{D}_k \vdash \mathit{load} & \textit{for some } k. \end{array}$$

Again, this is weaker than requiring the constraint to be logically entailed.

The aforesaid interpretation of implicational constraints is standard in deductive databases. To emulate such behavior, deductive databases rely on the closed-world assumption. By analogy, $\mathcal{TR}$ interprets constraints using the closed-world assumption. That is, formally, saying that a transaction ϕ executes on a path $\mathbf{D}_0 \ldots \mathbf{D}_n$ under a dynamic constraint ψ means that $\mathbf{M}, \mathbf{D}_0 \ldots \mathbf{D}_n \models \phi \wedge \psi$, where $\mathbf{M}$ is the "intended model" of $\mathbf{P}$ determined by some closed-world theory. There are several popular definitions of "intended model," but they all agree on the most common form of rulebase—locally stratified rulebases. In this case, the intended model is identical to the perfect model [83], and in the special case of Horn rulebases, this is just the minimal model. All these ideas can be extended to Transaction Logic, including the notions of local stratification, perfect model and minimal model. Details can be found in Section 9.

7.8 The Frame Problem

One of the hardest issues haunting the logical formalization of actions and planning is the so-called *frame problem* [71]. A number of ingenious approaches to taming this problem have been proposed since the mid 1970's, but the problem always comes back and retaliates in various ways.

Most work on the frame problem has been in search of a formula to represent, preferably in the first-order logic, what can be informally stated in English as "... and everything else remains unchanged." The relative merit of various solutions is then argued on the grounds of which logical representation seems more elegant, compact, and how well it deals with "satellite" problems, such as the *ramification problem* [35]. As far as the user is concerned, one problem with such approaches is that they provide only a *methodology* for solving the frame problem. The ultimate responsibility for writing down the correct frame axioms lies with the user. The most serious drawbacks of these proposals, however, have been their computational complexity and the fact that adding new actions to the system entails re-coding the entire frame problem in non-incremental ways. In our opinion, this latter drawback eliminates most such proposals from consideration as a basis for a knowledge-base programming language. Re-coding of the frame problem, even when it can be done automatically, is a major burden on system maintenance and is likely to take toll on performance.

Solving the frame problem is not usually an end in itself, however. Rather, the main problem is to answer questions about actions, and it is this that leads to the frame problem. After all, until the frame problem is solved, an action is incompletely specified, so certain questions about it must go unanswered. A proposed solution to the frame problem is usually deemed incomplete if it cannot give the correct answer to questions like the following, where ϕ is a first-order formula (or "fluent" [84]):

1. *Action execution*: Given an initial state $\mathbf{D}$, what is the final state of the action?

2. *Frame problem proper*: Is formula ψ unaffected by the action?

3. *Program verification*: Does the outcome of action α always satisfy formula ψ?

4. *Reasoning about action execution*: Given some knowledge of the initial state, and some knowledge of the final state, what can be said about intermediate states?

5. *Planning and design*: What sequence of actions will make formula ψ true?

Many ingenious solutions to the frame problem have been proposed. These proposals usually address the very general problem of arbitrary questions about arbitrary updates to arbitrary logical theories. Because of this generality, these solutions have been conceptually complex and computationally intractable. Worse yet, they remain complex and inefficient even for very simple actions, like inserting or deleting a single tuple from a relational database.

Fortunately, many interesting and important problems in knowledge representation do not require a completely general solution to the frame problem. In fact, one of the most basic questions that can be asked of an action is question 1, above. Answering this question does not require a sophisticated solution to the frame problem, since it can be answered by any procedural language that simply executes (or simulates) the action. Likewise, many planning and design problems do not require a general solution to the frame problem either, as many AI systems have shown (e.g., STRIPS, NOAH, MOLGEN, etc.) What is needed, then, is a general way of solving a wide class of representative problems, while offering simple and efficient solutions to situations commonly arising in practice.

Some logical and quasi-logical systems have achieved this goal to an extent. However, they all impose artificial restrictions on the syntax of transactions and on the way they can be combined (*e.g.*, [86, 69]; cf. also Prolog and SQL). These systems, like procedural systems, do not offer the expressiveness and flexibility of a full-fledged logic. Often, these systems cannot express post-conditions, intermediate conditions, non-determinism, and/or execution constraints. As a result, many interesting problems are beyond their reach, including all of the examples in Section 7.7.

$\mathcal{TR}$ is the first practical logic to avoid these limitations. In fact, $\mathcal{TR}$ is *both* more efficient and more expressive than these logical and quasi-logical systems. Efficiency is possible because the proof theory for $\mathcal{TR}$ materializes the current database state, thus emulating commercial database systems. Consequently, the proof theory for $\mathcal{TR}$ is simpler and more direct than the proof theory for many other logics of action. As an example, consider the situation calculus. While, queries in $\mathcal{TR}$ are evaluated directly against the current database state, queries in the situation calculus are "regressed" through the entire history of updates to the original database state [88, 84]. Regression thus represents a considerable expense when the update history is long, which is the norm in modern commercial systems, which can process hundreds of transactions per second. Like queries, evaluating action pre-conditions in $\mathcal{TR}$ is also simple and direct, without the expense or complexity of goal regression. In fact, tests with the present prototype demonstrate that $\mathcal{TR}$ is comparable in efficiency to procedural systems that update data structures destructively [51].

In addition, because its semantics is based on paths, $\mathcal{TR}$ can express constraints on action execution in a way that many other logics of action cannot. More generally, because it is a full-fledged logic, $\mathcal{TR}$ can solve a wide class of problems in AI, databases and elsewhere. Nevertheless, because it does not require a completely general solution to the frame problem, $\mathcal{TR}$ provides simple and efficient solutions to a wide and important class of AI problems, as illustrated in Sections 7.7 and 7.9.

Since $\mathcal{TR}$ includes classical first-order logic as a special case (and thus the whole of the situation calculus), general solutions to the frame problem can be encoded in $\mathcal{TR}$ as a transaction base. However, for many problems, such as action execution, this approach is both inefficient and unnecessary. For such problems, the *transition oracle* encodes all the relevant information. In effect, the oracle provides a partial solution to the frame problem at the bottom level, *i.e.*, for elementary state transitions. This solution then *automatically propagates* to complex transactions. (We say "partial solution" because the transition oracle is needed only for executing actions, and for this, a complete solution to the frame problem is not needed.)

The transition oracle in effect specifies *pre-computed* state transitions. Like the axiom sets of most logics, the transition oracle may be infinite.[19] In practice, however, transition oracles will work algorithmically. That is, given an input database, the transition oracle will compute an output database. For example, suppose the transition oracle specifies that a first-order formula be inserted into a database consisting of first-order formulas. As discussed in Section 4.2, difficulties arise if the inserted formula and the database contradict each other. Algorithms of reasonable complexity for solving this problem have been proposed by Grahne and Mendelzon [45]. These algorithms could be used to implement the transition oracle. However, even this complexity is not usually required, since in practice, most databases are relational, *i.e.*, sets of ground atomic formulas. For example, most (if not all) blocks-world problems are relational, as are most examples in this paper. Algorithms for inserting and deleting atoms from a relational database are computationally trivial. Even when the database is a set of Horn rules, it is a simple matter to insert and remove rules. It is for this reason that $\mathcal{TR}$ is able to execute many actions efficiently.

Blocks World

To illustrate, consider the examples of Section 5.5, in which a robot simulator moves blocks around a table-top. Example 5.8 shows the exact sequence of inferences and state transitions needed to answer

[19] Like the examples in this paper, however, the oracle may be specified by a finite axiom schema.

the following query:

$$?-\ pickup(blkA)$$

Note that here we are *not* trying to generate a plan for picking up $blkA$; instead, we are asking the inference system to *execute* the action *pickup(blkA)*. If, say, the top of $blkA$ is not clear, this action will fail. To generate a plan, the user would have to ask a different query, as explained in Section 7.9.

Note that although there are two state changes in Example 5.8, none of the inferences deal with the frame problem, *i.e.*, there is no attempt to infer what does *not* change. Instead, each entry in the transition oracle specifies the exact effect of each elementary update on each database. For easy reference, we reproduce two of these entries here:

$$on.del(a,b) \in \mathcal{O}^t(\{on(a,b), on(b,c), isclear(a)\},\ \{on(b,c), isclear(a)\})$$

$$isclear.ins(b) \in (\{on(b,c), isclear(a)\},\ \{on(b,c), isclear(a), isclear(b)\})$$

where a, b, and c are arbitrary blocks. Notice that both entries specify what does *not* change, as well as what does. The first entry, for instance, specifies that the atom $on(a,b)$ is deleted from the database, and that the other two atoms, $on(b,c)$ and $isclear(a)$, remain intact. It is in this sense that the transition oracle solves the frame problem for elementary state transitions.

Once the transition oracle is specified, elementary transitions can be combined to form complex transactions without giving any more thought to the frame problem. For example, from the two entries above, it follows that the transaction $on.del(a,b) \otimes isclear.ins(b)$ causes the database to pass through the following sequence of states:

$$\{on(a,b),\ on(b,c),\ isclear(a)\}$$
$$\{on(b,c),\ isclear(a)\}$$
$$\{on(b,c),\ isclear(a),\ isclear(b)\}$$

We can see now how the solution to the frame problem automatically propagates from elementary state transitions to complex transactions. For example, if $\mathbf{D} = \{on(blkA, blkB), on(blkB, blkC), isclear(blkA)\}$ is the initial database state, the following statements are all true:

$$\begin{aligned}
\mathbf{P},\mathbf{D}\text{---} &\models on.del(blkA,blkB) \otimes isclear.ins(blkB) \otimes isclear(blkB)\\
\mathbf{P},\mathbf{D}\text{---} &\models on.del(blkA,blkB) \otimes isclear.ins(blkB) \otimes isclear(blkA)\\
\mathbf{P},\mathbf{D}\text{---} &\models on.del(blkA,blkB) \otimes isclear.ins(blkB) \otimes on(blkB,blkC)
\end{aligned}$$

The first statement says that $isclear(blkB)$ is true after transaction execution ($isclear(blkB)$ was *not* true in the initial state). The second and third statements say that $isclear(blkA)$ and $on(blkB, blkC)$ are also true after transaction execution. Note that these latter two atoms were true in the initial state. In most other logical systems, the latter two inferences would be handled by general, complex machinery for dealing with the frame problem. Even if this machinery could be optimized for the simple cases at hand, still the atoms $isclear(blkB)$ and $on(blkB, blkC)$ need to be inferred *after* $on.del(blkA, blkB)$ and $isclear.ins(blkB)$ have been executed. In many logics, this is a relatively complex matter. The situation calculus, for example, resorts to goal regression [88], as described above. In contrast, the $\mathcal{TR}$ proof procedure simply examines the final database state, an inference that is computationally trivial.

In conclusion of this discussion we note that specifying frame axioms for non-deterministic actions is one of the harder problems in formalisms based on the situation-calculus. In $\mathcal{TR}$, as long as we stick to questions 1 and 5 above (action execution and planning), handling non-deterministic actions is no more difficult than handling their deterministic brethren. For instance, consider the action $stackSameColor(blkZ)$ of Example 2.13. This action is non-deterministic because the two blocks to be stacked on top of $blkZ$ (and their color) can be chosen in several different ways. In $\mathcal{TR}$, the transaction $?- \ stackSameColor(blkZ)$ would be executed by choosing an appropriate pair of blocks of the same color and then putting them on top of $blkZ$. The model theory and proof theory of $\mathcal{TR}$ are insensitive to whether this transaction is deterministic or not. In effect, non-determinism falls out of $\mathcal{TR}$'s semantics for free.

Yale Shootout

At this point it is worth touching upon the subject of a Yale shooting incident reported in [48]. In 1985, an unknown individual loaded a gun and, after a brief pause, discharged it into Fred (last name unknown). The frame problem comes into picture here in several places; most notably, where it is necessary to conclude that the loaded gun does not get discharged during the pause. The problem is to infer that Fred is dead. It is typical of a class of problems requiring temporal reasoning.

There are several known solutions to this problem, but they are usually either too limited or too general (and complex). The general solutions are typically based on a *non-monotonic* logic, which, we contend, is too strong a medicine for the problem at hand. The result is that, even for simple problems, the solutions are relatively complex and the proof theory (when it exists) is inefficient compared to $\mathcal{TR}$ (*e.g.*, [63, 64, 93]). On the other hand, the more limited, monotonic solutions each have difficulties of their own, ranging from complex encodings of the frame problem, to inefficient proof theories, to ad-hoc syntactic restrictions, to limited expressiveness (*e.g.*, [86], Prolog, SQL).

$\mathcal{TR}$ offers a new type of solution, one based on a *monotonic* logic, with *no* syntactic restrictions, and with an *efficient* SLD-style proof theory. For instance, because $\mathcal{TR}$ provides a logical account of STRIPS-like actions, we can determine the fate of Fred by simulating the sequence of events at Yale, as simply and directly as a procedural system would do. What $\mathcal{TR}$ offers, then, is a logic-programming solution *not* based on negation-as-failure (with all its attendant complexities, controversies, and limitations).

To simulate the events at Yale, we provide the five rules below. It should be noted how much more straightforward this representation is compared to other solutions. For better readability, we define a few extra actions, in addition to the original *load*, *wait*, and *shoot* reported to the police:

$$\begin{array}{l} load \longleftarrow loaded.ins \\ wait \longleftarrow \texttt{no-op} \\ shoot \longleftarrow loaded \otimes unload \otimes die \\ unload \longleftarrow loaded.del \otimes unloaded.ins \\ die \longleftarrow alive.del \otimes dead.ins \end{array}$$

where `no-op` is an elementary state transition such that $\texttt{no-op} \in \mathcal{O}^t(\mathbf{D}, \mathbf{D})$, for every database $\mathbf{D}$. It is not hard to see that execution of the transaction $?- \ load \otimes wait \otimes shoot$ on any database will result in death of the subject. Formally, the following statement is true:

$$\text{if} \quad \mathbf{P}, \mathbf{D} \text{---} \mathbf{D}' \models load \otimes wait \otimes shoot \qquad \text{then} \quad \mathbf{P}, \mathbf{D}' \models dead.$$

We thus see that the frame problem is handled correctly when the gunman[20] takes a pause; that is, the gun does not get miraculously unloaded.

Ramification Problem

To conclude the discussion of the frame problem, we illustrate how $\mathcal{TR}$ alleviates the so-called ramification problem [35]. This problem arises because actions can have both direct and indirect effects. For example, the direct effect of pulling a sink's drain plug is that the drain pipe is no longer covered. This would be encoded explicitly in the definition of the action. In addition, however, the action has the indirect effect of emptying the sink (if it was full). The latter effect is called a *ramification* of the action, since it is not part of the action's definition. Ramifications have been difficult to treat logically since they fly in the face of the frame problem. Indeed, most solutions to frame problem forbid changes that are not *explicitly* specified as part of an action's definition. Unfortunately, these forbidden changes often include ramifications, which occur *implicitly*, as a consequence of an action.

Ramifications are particularly important in database systems, where data is both stored and derived. Derived data (views) is generated automatically from the stored data. When the stored data is changed, the derived data changes too, without any explicit say so from the user. This behavior is well-known to even naive users of database systems, since updates to a stored relation often cause updates to derived relations (views) as a side effect.

A number of attempts have been made to address the ramification problem logically (*e.g.*, [85]). As with the frame problem, many of these attempts have addressed the very general problem of asking arbitrary questions about arbitrary updates to arbitrary logical theories. Because of their generality, these proposals are conceptually complex and computationally intractable. Perhaps their most serious drawback is that views and actions cannot be defined incrementally and independently of each other. For instance, adding a new action can require redefining every view. Likewise, defining a new view can require taking every action into account. In addition, many of these proposals cannot deal with recursively defined views at all. Worse yet, these general solutions remain complex, inefficient and non-incremental even in very simple situations, such as inserting or deleting a single tuple from a relational database.

Fortunately, many interesting and important problems do not require a completely general solution to the ramification problem. Database systems, for instance, have provided views for a long time. To do this, they forgo the ability to ask arbitrary queries about updates. Typically, queries are posed only to the result of an update, not to its definition. Thus, after updating the current database, one can ask, "is $p(a)$ true?" However, one cannot ask, "Will $p(a)$ *always* be true after this update, *i.e.*, for every initial database?"

To achieve their limited (but important) functionality, database systems employ two ideas: (i) they distinguish between stored and derived data, and (ii) they use both procedural and declarative knowledge. Typically, procedures update the stored data, after which, logical rules recompute the derived data. Unfortunately, no database system has integrated these ideas into a unified logical system. Thus, while database systems solve a narrow range of problems very well, their flexibility and expressiveness is limited. For instance, they cannot deal with any of the examples in Section 7.7.

$\mathcal{TR}$ ameliorates this problem by integrating the two ideas above into a single logic. Unlike other attempts to do this (e.g., [69, 64]), $\mathcal{TR}$ does not distinguish between base predicates and derived predicates, since every predicate can be both base *and* derived. In this way, $\mathcal{TR}$ is comparable to

[20] By some accounts, this was a woman, Mary.

Prolog, in which updates and derived data are possible, and yet there is no explicit declaration of what must be derived and what must be stored. This comparison is actually quite close, since both in $\mathcal{TR}$ and in Prolog, the direct effects of actions are specified procedurally, as changes to stored knowledge, while the indirect effects (ramifications) are specified declaratively, as logical inference rules.

Of course, there is no formal distinction in $\mathcal{TR}$ between procedural and declarative knowledge, since both are encoded in the transaction base as transaction formulas. For instance, to encode the drain-plug problem, we use the following two rules:

$$unplugSink \leftarrow drainCovered.del \otimes drainUncovered.ins$$
$$sinkEmpty \leftarrow drainUncovered$$

The first rule encodes the direct effects of the action *unplugSink*, which is to uncover the drain plug. The second rule encodes the indirect effects of the action, which is to empty the sink. Formally, if the transaction base $\mathbf{P}$ contains these two rules, then the following statement is true, for any database $\mathbf{D}$:

$$\text{if} \quad \mathbf{P}, \mathbf{D} \text{---} \mathbf{D}' \models unplugSink \qquad \text{then} \quad \mathbf{P}, \mathbf{D}' \models sinkEmpty$$

That is, after unplugging the sink, the sink becomes empty.

Of course, classifying the effects of an action as direct or indirect can be a subjective process. In principle, every indirect effect could be made into a direct effect by encoding it explicitly in action's definition. In practice, however, this approach is awkward, impractical, and leads to a loss of modularity and encapsulation. For instance, many actions have similar indirect effects, which should be specified only once. Furthermore, indirect effects are often the result of physical constraints, which are most naturally encoded separately. In addition, in database systems, updates to stored relations have indirect effects on views, and it is extremely impractical to require that every update specify its effect on every view. For these reasons, it is crucial to be able to combine procedural knowledge with declarative knowledge, as in the example above.

To make this possible, $\mathcal{TR}$ carefully distinguishes between the updatable database and the non-updatable transaction base, something that other logics do not do. Unlike deductive databases, this distinction is semantic as well as syntactic. In particular, $\mathcal{TR}$ does not create an artificial distinction between base and derived *predicates*. Instead, there is only *one* kind of predicate (as in classical logic), and any predicate can contain *both* stored and derived data. To capture the distinction semantically, path structures distinguish between a state $\mathbf{D}$ and a path $\langle \mathbf{D} \rangle$ of length 1. The state represents stored data (at least, when the database is relational), and the path represents derived data (the view). Thus, as updates change the stored data from state $\mathbf{D}_1$ to $\mathbf{D}_2$, the view changes automatically from $\langle \mathbf{D}_1 \rangle$ to $\langle \mathbf{D}_2 \rangle$. Syntactically, in expressions like $\mathbf{P}, \mathbf{D} \text{---} \models \phi$, the database $\mathbf{D}$ represents the updatable data, and the query ϕ represents derived data. Note that the transaction base $\mathbf{P}$ contains both procedural and declarative knowledge, through which the database is updated and queries are answered, respectively. Intuitively, each formula in the transaction base represents a procedure along with its calling sequence. These procedures operate on the underlying database, either to change it, or to compute answers.

A More Involved Ramification Example

To further illustrate $\mathcal{TR}$'s approach to the ramification problem, we consider another example, a variation of Example 2.13. Here, we modify the robot so that it can pick up not just a single block, but an entire stack of blocks. In Example 2.13, a block had to be clear before the robot could pick it up (*i.e.*, nothing could be on top of the block). We now remove this restriction. That is, given a stack of blocks, the robot can lift the entire stack by (carefully) grasping and lifting the bottom-most block. More generally, the robot can split a stack in two by grasping and lifting a block in the *middle* of the stack. This ability leads to an instance of the ramification problem, since the direct effect of lifting one block has an indirect effect on the blocks on top of it (*i.e.*, they get lifted too). The logical problem is to infer the correct state of the world after the robot picks up a block. For the purpose of this example, we assume, as in most blocks-world examples, that the database is relational.

To describe the direct effects of picking up and putting down a block, we define the following two rules for updating the stored data:

$$pickup(X) \leftarrow on(X,Y) \otimes on.del(X,Y) \otimes isclear.ins(Y) \otimes lifted.ins(X) \qquad (26)$$

$$putdown(X,Y) \leftarrow isclear(Y) \otimes on.ins(X,Y) \otimes isclear.del(Y) \otimes lifted.del(X) \qquad (27)$$

These rules should be compared to the corresponding rules in Example 2.13. Here, we have removed the requirement that a block be clear before it is picked up. We have also added the predicate *lifted*(X), since we are now interested in knowing what blocks have been lifted.

Physically, when a block is lifted, all blocks on top of it are also lifted. However, these changes are *not* encoded into the *pickup*-action above: They are ramifications of the action. They can be derived from the stored data by the following Horn rule:

$$lifted(X) \leftarrow on(X,Y) \wedge lifted(Y) \qquad (28)$$

which we add to the transaction base. This rule says that any block on top of a lifted block is itself lifted. Note that this rule is *recursive*, so that when a block is lifted, *every* block above it is also lifted, even when other blocks intervene. This is precisely the kind of rule that other logical approaches to the ramification problem have trouble dealing with. Also note that data in the *lifted* relation is partly stored and partly derived, reflecting the direct and indirect effects of the *pickup*-action.

As an example, suppose we have a stack of three blocks, where block A is on top of block B is on top of block C. When the robot picks up block C, the atom *lifted*$(blkC)$ is inserted into the database by Rule (26). The atoms *lifted*$(blkB)$ and *lifted*$(blkA)$ are then automatically inferred by Rule (28). These two atoms are the ramifications. Likewise, when the robot puts down block C, the atom *lifted*$(blkC)$ is deleted from the database by Rule (27). The atoms *lifted*$(blkB)$ and *lifted*$(blkA)$ then automatically disappear, since they can no longer be inferred. In this way, we correctly represent the state of the world as the robot picks up and puts down the entire 3-block stack. Formally, let $\mathbf{P}$ be any transaction base containing Rules (26)–(28). If $\mathbf{D}$ is the database just before the *pickup*-action, then the following statement is true:

$$\text{if} \quad \mathbf{P}, \mathbf{D} \text{ --- } \mathbf{D}' \vdash pickup(blkC) \qquad \text{then} \quad \mathbf{P}, \mathbf{D}' \models lifted(blkA) \wedge lifted(blkB)$$

In other words, querying the database after the execution of *pickup*$(blkC)$ will establish that blocks A and B have both been lifted—a ramification of the *pickup*-action.

7.9 Planning

In this section, we illustrate the capabilities of $\mathcal{TR}$ by encoding some well-known AI planning strategies as formulas in $\mathcal{TR}$. Planning is possible in $\mathcal{TR}$ because transaction bases can represent two types of knowledge about actions: action-definitions, which describe how to execute actions; and planning strategies, which describe how and when to execute actions. In many cases, both types of knowledge can be represented as serial Horn rules. Planning itself is carried out by the inference systems developed in Section 6.

This section considers three planning regimes. In the first, *naive* planning, the inference system searches blindly for any sequence of actions that achieves the planning goal. The naive system is simple to formulate, but is very inefficient and so its utility is limited to demonstrating that planning is possible in $\mathcal{TR}$. The other two regimes improve upon the naive system by incorporating knowledge about how goals are to be achieved. In the second regime, planning is described in terms of a hierarchy of "scripts." Scripts suggest ways of achieving goals, and often they summarize known methods in a particular problem domain. This kind of planning, which is exemplified by systems such as NOAH [91, 90] and MOLGEN [36, 94], is natural for $\mathcal{TR}$, since each script corresponds to a high-level, non-deterministic action. The third planning regime that we consider is exemplified by the well-known STRIPS system [34, 65], which plans movements for a robot arm. We show that STRIPS can be naturally represented in $\mathcal{TR}$, and that STRIPS inference rules are sound but incomplete in our semantics. This incompleteness is responsible for certain failures of STRIPS, such as its inability to exchange the contents of two registers [79]. Finally, we point out some additional problems with the STRIPS planning strategy, problems that extend to many other planning systems. We then suggest a solution in the form of an improved STRIPS planner that is firmly grounded in $\mathcal{TR}$.

Before proceeding, it is worth noting that many of the $\mathcal{TR}$-rules considered in this paper are based on the insertion and deletion of tuples from a database. These rules, therefore, bear a conceptual resemblance to STRIPS-actions. This is especially true of the rules in Example 2.13. Several differences are worth noting however:

- Unlike STRIPS-actions, $\mathcal{TR}$-rules are formulas in a rigorous *logical* formalism. They are therefore declarative as well as executable.
- Rules in $\mathcal{TR}$ are *hierarchical* and can be defined at many levels of abstraction. The six rules in Example 2.13, for instance, represent six different levels of abstraction. In contrast, STRIPS only allows actions to be defined at one level, directly in terms of database inserts and deletes; *i.e.*, it does not support intermediate-level actions (subroutines).
- Unlike STRIPS-actions, actions in $\mathcal{TR}$ can be *non-deterministic*. As Example 2.13 shows, non-determinism can make actions simpler and easier to formulate, reducing the amount of detail that a user must specify.
- STRIPS-actions are relatively simple; they consist of a pre-condition (which is a series of tests), followed by a set of deletes, followed by a set of inserts, in that order. In contrast, in $\mathcal{TR}$-rules, inserts, deletes, and tests can be sequenced in any order. Thus, apart from pre-conditions and post-conditions, any number of intermediate tests can also be specified. In fact, even more general actions are possible in $\mathcal{TR}$, since in addition to sequential ordering, formulas may be combined via classical conjunction, disjunction and negation.

7.9.1 Naive Planning

Planning is possible in $\mathcal{TR}$ because actions can be defined both declaratively and procedurally. Thus, one does not have to specify an action as an exact sequence of updates. Instead, actions can be defined declaratively, in terms of what they accomplish, not in terms of how they do it. This section shows how to write rules in $\mathcal{TR}$ that define a *plan* to be any sequence of actions that accomplish a given task (such as building a stack, or designing a circuit). The approach illustrates in a simple way why $\mathcal{TR}$—a logic designed to *execute* actions—can also *plan* with them. However, we say that the resulting planning system is *naive*, since it receives no guidance in achieving its planning goals.

This kind of planning is based on primitive planning actions (or operators). These primitive *actions* differ from elementary *updates* in that they correspond to real-world activities, such as robot actions. In Example 2.13, for instance, there are two primitive robot actions, $pickup(X)$ and $putdown(X,Y)$, in terms of which the other robot actions are defined. These primitive actions are in turn defined in terms of low-level database inserts and deletes, which are elementary updates that do not correspond to robot actions.

To see how naive planning is done in $\mathcal{TR}$, consider a transaction base with n primitive actions, $a_1, \ldots, a_n$. We define a *plan* to be any sequence of these actions. Such sequences are defined by the following $\mathcal{TR}$ rules:

$$
\begin{aligned}
&plan \leftarrow a \otimes plan \\
&plan \leftarrow a \\
&a \leftarrow a_1 \\
&a \leftarrow a_2 \\
&\quad\cdots \\
&a \leftarrow a_n
\end{aligned}
$$

These rules state that a can be any primitive action, and $plan$ is any sequence of such actions. Now, suppose we want to achieve a particular goal, such as $isclear(b)$, which states that block b must be clear. The transaction $?-\ plan \otimes isclear(b)$ would then achieve this goal, and a sequence of primitive actions that results in $isclear(b)$ being true would be recorded in the proof. More generally, we will want to achieve complex goals. For instance, suppose we want to stack block b on top of c on top of d, *i.e.*, the planning goal is $on(b,c) \wedge on(c,d)$. The transaction $?-\ plan \otimes [on(b,c) \wedge on(c,d)]$ achieves this goal. Since the predicate *on* is a query, and thus does not change the database, we can replace the classical conjunction by a serial one and write $?-\ plan \otimes on(b,c) \otimes on(c,d)$.

These ideas are easily generalized. If ψ is an arbitrary planning goal, then $?-\ plan \otimes \psi$ is a transaction that achieves the goal. This transaction therefore specifies a sequence of primitive actions such that ψ is true in the final database state. The most common situation is when ψ is a conjunction of queries, *i.e.*, ψ has the form $g_1 \wedge \ldots \wedge g_m$, where each g_i is a query. In this case, the transaction $?-\ plan \otimes g_1 \otimes \ldots \otimes g_m$ achieves the planning goal. Since this formula is a serial goal, it can be dealt with by the inference system of Section 6. The plan itself can be extracted from the proof generated by the inference system. Indeed, it suffices to retain only those steps in which the primitive actions, $a_1, \ldots, a_n$, were used in the proof of $plan \otimes g_1 \otimes \ldots \otimes g_m$, disregarding lower-level actions used to define the a_i's.

Note that this formulation of planning relies on several specific features of $\mathcal{TR}$ actions:

- Serial conjunction, which allows primitive actions to be combined sequentially.
- Post-conditions, which allow planning goals to be stated.
- Recursion, which allows the inference system to generate sequences of actions of arbitrary length, by applying the same rule(s) over and over again.

- Non-determinism, which allows actions to be carried out in several different ways. Together with recursion, non-determinism allows us to define a plan to be *any* sequence of primitive actions.

The approach to planning which we have just illustrated is conceptually simple, but is too unfocused in its blind search for plans. The inference systems of Section 6, for instance, will simply string actions together at random until a sequence is found that achieves the goal. This behavior is often called forward (or bottom-up) planning. It is interesting to note that even if we operated the inference system of Section 6.3 in a backward (or top-down) mode, it would still result in an unfocused, bottom-up planner. The reason for this apparent paradox is that the inference system works on transactions from left to right. Thus, in the transaction $?-\ plan \otimes \psi$, the planning goal ψ is the last thing to be considered, and so it does not affect the way the plan is generated. If the plan does not achieve the goal, then the inference system simply backtracks and generates another plan. This process continues until a plan that achieves the goal is finally found.

7.9.2 Script-Based Planning

This section illustrates another approach to planning, known as "script-based planning." Unlike naive planning, it is highly focussed. Script-based planning augments action definitions with "scripts" or "skeletal plans" that are known to be useful in achieving certain kinds of goals. For example, a skeletal plan for making coffee might be as follows: grind coffee, boil water, put coffee in filter, pour water into filter [29, Article XV.D1]. In $\mathcal{TR}$, this skeletal plan could be represented by a rule like the following:

$$makeCoffee \ \leftarrow \ grindCoffee \otimes boilWater \otimes fillFilter \otimes pourWater \tag{29}$$

As this example illustrates, skeletal plans often have the form of a procedure (or recipe). For this reason, skeletal plans are often specified in a procedural language. As the rule above illustrates, however, skeletal plans can also be specified *in a logic*, $\mathcal{TR}$, that integrates procedural and declarative knowledge in a single framework.

Numerous AI systems use skeletal plans (e.g., [91, 36, 94, 92]). The rationale behind them is that human planning relies heavily on an individual's extensive experience in the problem domain. The idea is to avoid reinventing general strategies, and to exploit plan outlines that have worked in the past or that have worked on related problems. In fact, people rarely discover totally new plan outlines, and instead, they usually instantiate existing ones [29, Article XV.E]. Script-based planning thus attempts to produce competent plans, not wildly innovative ones. Furthermore, as we shall see in Section 7.9.4, wild innovation has a dark side, since it can produce plans that have unacceptable and even dangerous side effects.

We shall show how the two kinds of knowledge—action definitions and skeletal plans—are smoothly integrated in $\mathcal{TR}$. In fact, the line dividing the two is very fuzzy. The main difference is that actions tend to be specific and low-level, whereas skeletal plans tend to be abstract and high-level. In fact, to some extent, we have already demonstrated this. Example 2.13, for instance, gave a hierarchy of skeletal plans for stacking blocks. For convenience, we reproduce the transaction base of that example here:

$$stackSameColor(Z) \leftarrow color(Z,C) \otimes stackTwoColors(C,C,Z).$$
$$stackTwoColors(C_1,C_2,Z) \leftarrow color(X,C_1) \otimes color(Y,C_2) \otimes stackTwoBlocks(X,Y,Z)$$
$$stackTwoBlocks(X,Y,Z) \leftarrow move(Y,Z) \otimes move(X,Y)$$
$$move(X,Y) \leftarrow pickup(X) \otimes putdown(X,Y)$$
$$pickup(X) \leftarrow isclear(X) \otimes on(X,Y) \otimes on.del(X,Y) \otimes isclear.ins(Y)$$
$$putdown(X,Y) \leftarrow wider(Y,X) \otimes isclear(Y) \otimes on.ins(X,Y) \otimes isclear.del(Y)$$

These rules define a hierarchy of skeletal plans. For instance, the rule for $stackTwoBlocks(X,Y,Z)$ outlines how to stack blocks X on top of Y on top of Z. When the variables X, Y, and Z are instantiated, the skeletal plan becomes a concrete plan. Likewise, the action $stackSameColor(Z)$ can be viewed as a skeletal plan for stacking two blocks of the same color on top of Z. Here, the concrete plan depends not only on the value of Z, but also on the initial state of the database, *i.e.*, on what blocks are available for stacking.

In this example, the planning problem is relatively simple, because each action can be expanded in only one way. In general, however, each action may be defined by any number of rules, in which case the variable bindings and the database state will determine which rules are applicable.

The example above is simple in another way, too: The rules assume that the blocks they need are already clear (*i.e.*, they have nothing on top of them). If blocks X, Y and Z are not clear, then the action $stackTwoBlocks(X,Y,Z)$ fails. This behavior is too simplistic for many AI applications. However, it is not difficult to modify the example so that the robot will try to clear block-tops before stacking them. To illustrate, we introduce a new predicate called $achieve\_isclear(X)$ that tries to clear the top of block X. Using this predicate, we define new versions of $move$ and $stackTwoBlocks$, called $achieve\_on(X,Y)$ and $achieve\_stack(X,Y,Z)$, respectively. These predicates do *not* fail if the blocks X, Y and Z are not initially clear; instead, they try to correct the problem by performing a sequence of actions that will result in the requisite blocks becoming clear. The new actions are defined by the following rules (where $move$, $isclear$ and on are as before):

$$achieve\_stack(X,Y,Z) \leftarrow achieve\_on(Y,Z) \otimes achieve\_on(X,Y)$$
$$achieve\_on(X,Y) \leftarrow achieve\_isclear(X) \otimes achieve\_isclear(Y) \otimes move(X,Y)$$
$$achieve\_isclear(X) \leftarrow isclear(X)$$
$$achieve\_isclear(X) \leftarrow on(Y,X) \otimes achieve\_isclear(Y) \otimes move(Y,Z) \otimes Z \neq X$$

As in the previous example, each of these rules specifies a skeletal plan for achieving a goal. Note that the fourth rule specifies a recursive plan: If there is a block on top of X, then this block must be cleared before X can be cleared. The third rule detects when X is finally clear, thus terminating the recursion. Without the fourth rule, the predicate $achieve\_on$ would behave just like the predicate $stackTwoBlocks$ defined earlier (*i.e.*, the two will succeed in exactly the same cases).

Notice that these rules distinguish two kinds of pre-condition: those that should be actively achieved and those that should be passively tested. In the last rule, for instance, the formula $on(Y,X)$ is passively tested, while the formula $isclear(X)$ is actively achieved. Without such a distinction (*i.e.*, if all pre-conditions were actively achieved), the system would try to put X on top of Y, only to remove it again! We shall return to this issue of "active" vs. "passive" pre-conditions in Section 7.9.4.

Interacting Subgoals and Optimization

One problem with the above solution is that subgoals may interact destructively. For instance, in the rule for $achieve\_on(X,Y)$, the subgoal $achieve\_isclear(Y)$ may undo the effects of $achieve\_isclear(X)$. That is, in trying to clear Y, the robot may inadvertently put a block down on top of X, so that X is no longer clear. Technically, this possibility does not matter, since if X is not clear, then the *move* action will fail. In this case, the system will automatically backtrack and try to clear Y in a different way. This process will continue until the system finds a way of clearing Y without covering X (whence the *move* action will succeed).

It is possible, however, to avoid much of the backtracking, and thus increase the efficiency of planning. To do this, we simply forbid the robot to put anything down on X once X has been cleared. We do this by locking (*i.e.*, protecting) the atomic formula $isclear(X)$, so that it cannot be deleted from the database. Because $\mathcal{TR}$ integrates procedural and declarative knowledge, we can implement locks through a simple programming methodology. First, we define an atom b to be *locked* if and only if the atom $locked\_b$ is in the database. We can then lock and unlock b by inserting and deleting $locked\_b$ from the database. With this in mind, we modify the definition of $achieve\_on$ as follows, where, for convenience, we have introduced special procedures for locking and unlocking the atom $isclear(X)$:

$$
\begin{aligned}
achieve\_on(X,Y) &\leftarrow achieve\_isclear(X) \otimes lock\_isclear(X) \\
&\qquad \otimes achieve\_isclear(Y) \otimes unlock\_isclear(X) \otimes move(X,Y) \\
lock\_isclear(X) &\leftarrow locked\_isclear.ins(X) \otimes unlocked\_isclear.del(X) \\
unlock\_isclear(X) &\leftarrow locked\_isclear.del(X) \otimes unlocked\_isclear.ins(X)
\end{aligned}
$$

These rules ensure that $isclear(X)$ is locked while the robot tries to clear Y. To enforce the locks, each primitive robot action is modified to respect them, *i.e.*, a primitive action is not allowed to delete a locked item. We therefore modify the definitions of *pickup* and *putdown* as follows:[21]

$$
\begin{aligned}
pickup(X) &\leftarrow isclear(X) \otimes on(X,Y) on.del(X,Y) \otimes isclear.ins(Y) \\
putdown(X,Y) &\leftarrow wider(Y,X) \otimes isclear(Y) \otimes [unlocked\_isclear(Y)] \\
&\qquad \otimes on.ins(X,Y) \otimes isclear.del(Y)
\end{aligned}
$$

In this way, the robot cannot put a block on top of X if the atom $isclear(X)$ is locked. Likewise, if block X is sitting on top of block Y, then the robot cannot pick up block X if the atom $on(X,Y)$ is locked.

Observe that the locked/unlocked trick above avoids the use of negation in rule bodies, since we can write $unlocked\_b(Y)$ instead of $\neg locked\_b(Y)$. This is a standard STRIPS device, such as using the predicate $isclear(Y)$ to avoid writing $\neg\exists X\ on(X,Y)$. The advantage of this device is that it can be used with the sound-and-complete proof theory for serial-Horn rules developed in Section 6. There are alternatives to this device. For instance, we could simply insert and delete the atom $locked\_b(Y)$, while the atom $unlocked\_b(Y)$ is inferred as needed by the following rule:

$$unlocked\_b(Y) \leftarrow b(Y) \wedge \neg locked\_b(Y)$$

[21]In these rules, expressions of the form $[\psi]$ are an abbreviation for $\psi \wedge \texttt{state}$, where ψ is a first-order formula, and `state` is the special proposition that is true only on states (Section 5.2). Such formulas force ψ to query the current database state, and are particularly useful when ψ involves negation-as-failure.

where the negation is interpreted as failure. This rule is placed in the database **D**. The serial Horn conditions of Section 6 permit such rules (so-called generalized Horn rules) in the database.

We can view locks as an optimization technique. That is, we can define skeletal plans without using locks, knowing that they can be automatically added by an optimizer. The general method is simple. Suppose the transaction base has the following planning rule:

$$b \leftarrow achieve\_a_1 \otimes achieve\_a_2 \otimes \ \cdots \ \otimes achieve\_a_{n-1} \otimes achieve\_a_n$$

An optimizer can simply rewrite the rule as follows:

$$\begin{aligned} b \leftarrow \quad & achieve\_a_1 \otimes lock\_a_1 \ \otimes \ achieve\_a_2 \otimes lock\_a_2 \otimes \ \cdots \ \otimes achieve\_a_{n-1} \otimes lock\_a_{n-1} \\ & \otimes \ achieve\_a_n \otimes unlock\_a_1 \otimes unlock\_a_2 \otimes \ \cdots \ \otimes unlock\_a_{n-1} \end{aligned}$$

In this way, achieving a_{i+1} will not undo $a_1, ..., a_i$.

Unordered Subgoals

In the examples given so far, the skeletal plans have been overly constrained, since each rule tries to achieve subgoals in a particular order. Ordering subgoals is not always necessary, however. For example, the rule defining $achieve\_on(X,Y)$ demands, unnecessarily, that X be cleared before Y is cleared. This kind of unnecessary ordering precludes some perfectly good plans. Worse yet, it could mean missing the most efficient plans, and it could prevent plans from satisfying dynamic integrity constraints (Section 7.7). Such are the dangers of over-specification. Ideally, the user would specify only what is necessary, from which the inference system would infer which plans are consistent with constraints, and from which an optimizer would choose an efficient plan.

Fortunately, it is not hard to avoid this kind of over-specification: Instead of specifying a particular order for subgoals, we can specify several possible orders or, indeed, any order. For example, instead of specifying $b \otimes c$, we could specify $(b \otimes c) \vee (c \otimes b)$. For convenience, we abbreviate the latter expression to $b \,\&\, c$. That is,

$$b \,\&\, c \ \equiv \ (b \otimes c) \vee (c \otimes b)$$

This connective is sometimes called the *shuffle* operator. In general, we define the transaction $p_1 \,\&\, p_2 \,\&\, \ldots \,\&\, p_k$ to succeed if and only if *some* ordering of the actions $p_1, p_2, \ldots, p_k$ succeeds. Thus, unlike serial conjunction, this new connective is commutative. Notice that rules involving "&" can be handled by the proof theory developed in Section 6, because they can be reduced to serial Horn rules in a straightforward way. For example, the rule $a \leftarrow b \otimes (c_1 \,\&\, c_2) \otimes d$ is equivalent to the following two serial Horn rules:

$$a \leftarrow b \otimes c_1 \otimes c_2 \otimes d \qquad\qquad a \leftarrow b \otimes c_2 \otimes c_1 \otimes d$$

Using the &-connective, the rules for building a stack can be written in a less constraining form, as follows:

$$\begin{aligned} & achieve\_stack(X,Y,Z) \ \leftarrow \ achieve\_on(Y,Z) \otimes achieve\_on(X,Y) \\ & achieve\_on(X,Y) \leftarrow [achieve\_isclear(X) \,\&\, achieve\_isclear(Y)] \otimes move(X,Y) \\ & achieve\_isclear(X) \leftarrow isclear(X) \\ & achieve\_isclear(X) \leftarrow on(Y,X) \otimes achieve\_isclear(Y) \otimes move(Y,Z) \otimes Z \neq X \end{aligned}$$

Here, we have used the shuffle connective in the second rule, since the order in which the two blocks X and Y are cleared is unimportant. However, we have *not* used "&" in the first rule even though we could have. In this case, the shuffle connective would only serve to needlessly increase the search space, since to stack blocks X on top Y on top of Z, the order of the subgoals *is* important. One advantage of skeletal plans is that they allow us to spell things out in this way when we want to.

7.9.3 STRIPS

The ideas above can be modified to simulate the STRIPS planning system [34, 79, 65] in $\mathcal{TR}$. It should be clear by now that translating STRIPS-action definitions into serial Horn rules is a fairly straightforward task (it is also spelled out below). What is not so trivial is to find a translation that not only preserves the semantics of STRIPS-actions, but also enables one to simulate the STRIPS planning strategy. This sections explains how to do this.

Each STRIPS-action is individually translated into a small set of serial Horn rules. One of these rules encodes the STRIPS-action itself, and is used to execute the action. The other rules specify scripts that simulate the STRIPS planning strategy. The entire translation is straightforward and can be carried out in linear time.

A STRIPS-action is a special kind of serial Horn rule: It has pre-conditions, but no post-conditions or intermediate conditions, and it is defined entirely in terms of inserts and deletes of atomic formulas. A STRIPS-action is specified by three sets of atomic formulas, called the pre-conditions, the delete-list, and the add-list. If the pre-conditions are satisfied by the current database, then the formulas on the delete list are deleted from the database, and those on the add list are inserted into the database. Here is a STRIPS-action, taken from [79, Section 7.2], that describes the act of lifting a block, X, that is on top of another block, Y:

$$\begin{array}{ll} \multicolumn{2}{l}{unstack(X,Y):} \\ \quad \text{Pre-conditions:} & handempty, \;\; isclear(X), \;\; on(X,Y) \\ \quad \text{Delete list:} & handempty, \;\; isclear(X), \;\; on(X,Y) \\ \quad \text{Add list:} & holding(X), \;\; isclear(Y) \end{array}$$

Here, $unstack(X,Y)$ is the name of the action. The corresponding $\mathcal{TR}$-representation is straightforward:

$$\begin{array}{rcl} unstack(X,Y) & \leftarrow & handempty \otimes isclear(X) \otimes on(X,Y) \\ & & \quad \otimes\, isclear.del(X) \otimes on.del(X,Y) \otimes handempty.del \\ & & \quad \otimes\, holding.ins(X) \otimes isclear.ins(Y) \end{array} \tag{30}$$

This rule tells us how to *execute* the action. However, to *plan* with such actions in a STRIPS-like manner, we have to envelop them in appropriate scripts:

$$\begin{array}{rcl} achieve\_isclear(Y) & \leftarrow & achieve\_unstack(X,Y) \\ achieve\_holding(X) & \leftarrow & achieve\_unstack(X,Y) \\ achieve\_unstack(X,Y) & \leftarrow & [achieve\_isclear(X) \;\&\; achieve\_on(X,Y) \;\&\; achieve\_handempty] \\ & & \quad \otimes\, unstack(X,Y) \end{array} \tag{31}$$

The first two rules say that to clear Y or to hold X, try unstacking X and Y. The third rule uses the shuffle operator, "&", to convey information that to unstack X and Y, one must first try to satisfy the pre-conditions of the *unstack*-action in any order, and then execute the action itself. This is exactly what the STRIPS system does. Note that when taken together, the scripts for all the actions will normally be mutually recursive.

The rules above describe how to convert goals into subgoals. In addition, we must describe how to *stop* the planning process. This happens when the goals we are trying to achieve are satisfied by the current database. This situation is described by the following rules, which we add to the transaction base. These rules, one for each predicate symbol, terminate the recursion.

$$\begin{array}{rcl} achieve\_on(X,Y) & \leftarrow & on(X,Y) \\ achieve\_isclear(Y) & \leftarrow & isclear(Y) \\ achieve\_holding(X) & \leftarrow & holding(Y) \\ achieve\_handempty & \leftarrow & handempty \end{array} \tag{32}$$

Taken together, the Rules (30–32) specify the planning process; but we must also spell out how planning goals are expressed. To see how this is done, suppose we want to stack block b on top of c on top of d; that is, we want to find a sequence of robot actions such that $on(b,c)$ and $on(c,d)$ are true in the final database state. This planning goal is specified by the following formula:

$$?-\ [achieve\_on(b,c)\ \&\ achieve\_on(c,d)] \otimes on(b,c) \otimes on(c,d)$$

The first part of this formula (in square brackets) represents a sequence of robot actions whose purpose is to achieve $on(b,c)$ and $on(c,d)$. The rest of the formula then checks whether the planning goal was, in fact, achieved. If it was not, then the inference system backtracks and tries to achieve the goal in another way. The reason why it is necessary to check the planning goal is that the STRIPS planning strategy is sensitive to possible interactions between subgoals. For instance, trying to put c on d may result in taking b off of c.

So far, we have shown how to translate a specific STRIPS-action, *unstack*, into serial Horn rules. It is easy to generalize the approach to other actions. To do this, suppose we are given an arbitrary STRIPS-action, specified as follows:

$$\begin{array}{l} xyz\_action: \\ \quad \begin{array}{ll} \text{Pre-conditions:} & p_1, p_2 \ldots p_i \\ \text{Delete list:} & d_1, d_2 \ldots d_j \\ \text{Add list:} & a_1, a_2 \ldots a_k \end{array} \end{array} \tag{33}$$

To specify the execution of this action in $\mathcal{TR}$, we add the following rule to the transaction base:

$$\begin{array}{rcl} xyz\_action & \leftarrow & p_1 \otimes p_2 \otimes \ldots \otimes p_i \\ & & \otimes\, d_1.del \otimes d_2.del \otimes \ldots \otimes d_j.del \\ & & \otimes\, a_1.ins \otimes a_2.ins \otimes \ldots \otimes a_k.ins \end{array} \tag{34}$$

To plan with this action, we add also the following planning scripts, expressed as serial Horn clauses:

$$\begin{array}{rcl} achieve\_a_1 & \leftarrow & achieve\_xyz\_action \\ achieve\_a_2 & \leftarrow & achieve\_xyz\_action \\ & & \ldots \\ achieve\_a_k & \leftarrow & achieve\_xyz\_action \\ achieve\_xyz\_action & \leftarrow & [achieve\_p_1\ \&\ achieve\_p_2\ \&\ \ldots\ \&\ achieve\_p_i] \otimes xyz\_action \end{array} \tag{35}$$

The conditions for terminating the recursion through xyz_action are similar to (32) above:

$$achieve\_p_s \longleftarrow p_s, \quad \text{for each} \quad s = 1, ..., i$$

Finally, to pose a set of planning goals, $g_1, g_2 \ldots g_n$, we use the following transaction:

$$?- \ [achieve\_g_1 \ \& \ achieve\_g_2 \ \& \ \ldots \ \& \ achieve\_g_n] \otimes g_1 \otimes g_2 \otimes \ldots \otimes g_n \qquad (36)$$

Given this formula, the inference system tries to achieve each of the goals g_i in some order. It then checks whether each of the g_i is true in the final database state. If not, the system backtracks and tries to achieve the g_i's in some other order.

The transaction (36) simulates the inner workings of STRIPS quite closely, but not pedanticly. In fact, the translation of a STRIPS-action (33) into the $\mathcal{TR}$-program (34–36) can be seen as an "ultimate STRIPS-style planner," which finds a plan whenever the original STRIPS does, but succeeds in more cases. This is because the formulas (34–36) rely on the *sound and complete* proof theory of $\mathcal{TR}$. Thus, they do not suffer from the idiosyncrasies, such as the inability to swap the contents of two registers [79, Section 7.5], that arise in STRIPS due to the ad hoc nature of its planning strategy.

The original STRIPS has been improved in numerous ways, which overcame the register-exchange problem as well as some of the problems discussed below. We contend, however, that the source of these problems as well as the need in numerous fixes lies in the very ad hoc nature and the lack of a logical base for STRIPS. In contrast, the above reconstruction of STRIPS in $\mathcal{TR}$ does not suffer from these deficiency and many improvements earlier proposed for STRIPS come for free merely due to the fact that $\mathcal{TR}$ is a logical language that has suitable compositional constructs and a simple, sound and complete proof theory.

7.9.4 Improved STRIPS

Many planning systems search for *any* sequence of actions that achieves a given goal. This is true of naive planning, of STRIPS, of systems based on goal regression, and of many other systems. Such systems can produce plans with unusual side effects that may look bizarre or are plainly unacceptable.

Unusual side effects arise for two specific reasons. First, many planning systems treat all outcomes of an action as equal. In particular, they do not distinguish between an intended outcome and a side effect. Second, many planning systems treat all pre-conditions of an action as equal. In particular, they do not distinguish between pre-conditions whose satisfaction needs to be planned, and those pre-conditions that are mere tests and thus should be satisfied without planning. The result in both cases is that many planning systems often have a much larger search space than necessary, one that includes intended plans as well as, so to speak, plan-looking non-plans. This section discusses these two issues in more detail, and shows how to avoid them, both in STRIPS and in $\mathcal{TR}$.

Intended Action Outcomes

Often actions have multiple, generally unrelated effects. For example, shooting may have two outcomes: hitting the target and unloading the gun. In planning, the outcomes of an action can usually be divided into two types: intended outcomes, and side effects. For instance, unloading a gun is normally a side effect of a shooting action, not its intended effect. Nevertheless, many planning systems do not make this distinction. Consequently, they may produce bizarre plans, such as shooting in order to unload a gun; or launching missiles in order to achieve disarmament.

To make the problem more concrete, consider the following STRIPS action of shooting a gun at a subject:

$shoot(X)$:

Pre-conditions:	$alive(X)$, $loaded$
Delete list:	$alive(X)$, $loaded$
Add list:	$dead(X)$, $unloaded$

In the original STRIPS, if the planning goal were *unloaded*, then $shoot(fred)$ would be a valid plan for achieving this goal.[22] The reason such bizarre plans may be generated is that all entries on the add-list are treated equally by the planning system. One way around this problem is to identify the *intended effects* of an action. For instance, the intended effect of $shoot(X)$ is to achieve the goal $dead(X)$, *not* the goal *unloaded*.

It should not be difficult to modify STRIPS to make such distinctions: For each STRIPS-action, we simply mark those entries on the add list that correspond to the purpose or intent of the action. Thus, in the case of $shoot(X)$, we would mark $dead(X)$. To achieve a goal, the planning system would use only those actions that are intended to achieve the goal. In this way, egregious side effects can be avoided.

We incorporate this idea in our translation (34–36) of STRIPS by modifying slightly the script (35); other parts of the translation remain unaffected. The modification consists in omitting those rules in (35) that try to achieve unintended outcomes of xyz_action. This includes all the rules with $achieve\_a_l$ in the head, where a_l is not marked as an intended outcome.

For instance, the $\mathcal{TR}$-definition of $shoot(X)$ corresponding to the above STRIPS-action would be:

$$shoot(X) \leftarrow alive(X) \otimes loaded \otimes alive.del(X) \otimes loaded.del \otimes dead.ins(X) \otimes unloaded.ins \tag{37}$$

and its script would become:

$$\begin{array}{rcl} achieve\_dead(X) & \leftarrow & achieve\_shoot(X) \\ achieve\_shoot(X) & \leftarrow & [achieve\_alive(X) \ \& \ achieve\_loaded] \otimes shoot(X) \end{array} \tag{38}$$

Unlike the script that would result from the general schema in (35), our new script does *not* include the rule $achieve\_unloaded \leftarrow achieve\_shoot(X)$. Thus, $shoot(X)$ cannot be used to unload the gun. Besides alleviating the problem of unintended outcomes, this approach has the added benefit of reducing the planning search space, since there are now fewer ways to achieve a goal.

Action Pre-conditions

In general, an action may have many pre-conditions. In our shooting example, the two pre-conditions were that the gun must be loaded and that the victim must be alive. In planning, the pre-conditions of an action can usually be divided into two types: Those pre-conditions that one can plan to satisfy, and those that should be satisfied without planning. For instance, in (37), one should not plan to bring the subject X to life if he is already dead. Thus, some pre-conditions should be actively achieved (like *loaded*), while others should just be passively tested (like $alive(X)$). As with the outcomes of

[22] One of the theories that the police developed in the aftermath of the Yale shooting incident was that this incident was caused by a stray planner who was blind to the distinction between various outcomes of its actions.

actions, many planning systems do not make this distinction; and consequently, they can produce, what we earlier called, plan-looking non-plans.

For some planning systems, distinguishing between "active" and "passive" pre-conditions, should not require extensive modifications. In STRIPS, for example, one could mark the pre-conditions of each action as either active or passive. Then, during planning, the active pre-conditions would become subgoals, whereas the passive pre-conditions would simply be tested against the current database state. However, unlike $\mathcal{TR}$, each such change would require re-examination and modification of the planning algorithm, since STRIPS does not rely on a general-purpose proof theory that is guaranteed to execute its specifications. In other words, unlike $\mathcal{TR}$, the STRIPS planning algorithm is built into the semantics of STRIPS, and is not specified by logical formulas.

Incorporating this idea in $\mathcal{TR}$ necessitates one additional modification to our translation (34–36) of STRIPS. As before, we leave the action definitions untouched and simply modify the planning scripts (to reflect the new planning strategy). The modification affects only the last rule in (35), where body-literals of the form $achieve\_p_s$ must be removed if p_s is a passive pre-condition.

In the case of $shoot(X)$, the last rule in the script (38) now becomes:

$$achieve\_shoot(X) \quad \leftarrow \quad achieve\_loaded \otimes shoot(X)$$

Here, the pre-condition $alive(X)$ of $shoot(X)$ is considered passive and its satisfaction is not actively sought. Thus, to kill X, the robot may try to shoot X, after first loading the gun. Furthermore, it will not do anything bizarre, like trying to revive X after discovering that X has already been killed by another robot.

In other, less dramatic situations, one still may need to distinguish between active and passive pre-conditions. For instance, consider the following STRIPS-action, taken from [79, Section 7.2], for picking up a block X from the table:

$pickup(X)$:

Pre-conditions:	$ontable(X)$,	$isclear(X)$,	$handempty$
Delete list:	$ontable(X)$,	$isclear(X)$,	$handempty$
Add list:	$holding(X)$		

and the corresponding $\mathcal{TR}$-definition:

$$\begin{aligned} pickup(X) \quad \longleftarrow \quad & ontable(X) \otimes isclear(X) \otimes handempty \\ & \otimes ontable.del(X) \otimes isclear.del(X) \otimes handempty.del \\ & \otimes holding.ins(X) \end{aligned}$$

The problem here is that the original STRIPS can produce useless plans in which the robot must pick up X before it can pick up X. To see this, notice that to pick up X, STRIPS will try to achieve the pre-condition $ontable(X)$. This requires that the robot be holding X, which in turn requires that the robot pick up X. The root of the problem here is that $ontable(X)$ is a pre-condition that should be passively tested for, not actively achieved. The desired behaviour is achieved by the following script:

$$\begin{aligned} achieve\_holding(X) \quad & \leftarrow \quad achieve\_pickup(X) \\ achieve\_pickup(X) \quad & \leftarrow \quad [achieve\_isclear(X) \; \& \; achieve\_handempty] \otimes pickup(X) \end{aligned}$$

In this way, achieving *holding*(X) does not call for achieving *ontable*(X). Instead, after achieving *isclear*(X) and *handempty*, the system tries to execute *pickup*(X) directly. If X is not on the table, then the action simply fails to execute. In this case, the planner must look for some other way to make *holding*(X) true (such as using the action *unstack*(X, Y) described earlier). In this way, the system avoids plans that have senseless loops in them.

Avoiding loops is not the only reason for distinguishing between active and passive pre-conditions. For instance, consider the following robot instructions, which raise the level of drama once again:

To move an inanimate object, carry it.
One way to make an animate object inanimate is to kill it. (39)

Then, if we asked the robot to bring Fred to us, the robot would reason—correctly—that it could do this by first killing Fred. In principle, we could get around this problem by asking the robot to bring Fred to us *alive*; but we might not think of this at the time of sending the robot on the errand. Furthermore, asking the robot to spare Fred's life is no insurance that it won't attempt to break Fred's bones, while trying to persuade him to come along. Who knows what qualifications would be enough to ensure robot's safe operation? This entire qualification problem can be avoided, though, if we distinguish between active and passive pre-conditions. For instance, in (39), we need only mark the pre-condition "inanimate" as passive, so that it will not be actively pursued.

8 Hypothetical Reasoning

Hypothetical queries play an important role in reasoning about knowledge. A frequently cited example comes from the British Nationality Act, which states that, "You are eligible for citizenship if your father *would* be eligible *if* he were alive" [38]. The reader is referred to [14, 16, 17] for numerous other examples of hypothetical queries. Because of such queries, it is often necessary to perform hypothetical updates as well as real ones. For instance, a game-playing program may reason as follows: After performing some given series of actions, α, does the opponent's situation improve? Observe that the actions mentioned in this query are purely hypothetical and are *not* committed. If the answer to the query is "no," then the program could perform action α, at which point the action *is* committed. Otherwise, the program could do further depth analysis and perform the most favourable move that it finds. By distinguishing between real and hypothetical actions, this program combines reasoning about action (planning, exploration of alternatives, etc) with actual execution of actions (committing itself to a particular course of action). $\mathcal{TR}$ is the only logic we are aware of that can do *both* these things.

This section extends the syntax and the semantics of $\mathcal{TR}$ to deal with hypotheticals. Extension of the proof theory then follows. The last subsection discusses a number of applications of hypothetical reasoning in $\mathcal{TR}$.

8.1 Hypothetical Formulas

To express hypotheticals, we introduce a modal operator "$\Diamond$" and a closely related operator "$\Box$". In modal terms, $\Diamond\,\phi$ means that execution of ϕ is *possible* starting at the present state, and $\Box\,\phi$ means that the execution of ϕ is *necessary* at the present state. Intuitively, "necessity" means that ϕ is the only transaction that can succeed from the present state. Formally, it means that ϕ is executable along every path leaving the current state, $\mathbf{D}$. Likewise, "possibility" means that ϕ is executable along *some* path leaving the current state.

To define the meaning of hypotheticals formally, let $\mathbf{M} = \langle U, \mathcal{F}, I_{path} \rangle$ be a path structure, let $\mathbf{D}$ be a state, and let ν be a variable assignment. Then

- $\mathbf{M}, \langle \mathbf{D} \rangle \models_\nu \Diamond\phi$ if and only if there is a path, π, starting at state $\mathbf{D}$, such that $\mathbf{M}, \pi \models_\nu \phi$ holds.
- $\mathbf{M}, \langle \mathbf{D} \rangle \models_\nu \Box\phi$ if and only if for every path, π, starting at state $\mathbf{D}$, it is the case that $\mathbf{M}, \pi \models_\nu \phi$.

Here, $\langle \mathbf{D} \rangle$ is the path of length 1 containing state $\mathbf{D}$. From these definitions, it is easy to verify that the following formulas are tautologies, *i.e.*, that they are true on every path of every structure:

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\Diamond\Diamond\phi & \leftrightarrow & \Diamond\phi & \Box\Box\phi & \leftrightarrow & \Box\phi \\
\Diamond(\phi \vee \psi) & \leftrightarrow & \Diamond\phi \vee \Diamond\psi & \Box(\phi \wedge \psi) & \leftrightarrow & \Box\phi \wedge \Box\psi \\
\Diamond(\phi \wedge \psi) & \rightarrow & \Diamond\phi \wedge \Diamond\psi & \Box(\phi \vee \psi) & \leftarrow & \Box\phi \vee \Box\psi \\
\Diamond(\phi \otimes \psi) & \rightarrow & \Diamond\phi & \Box(\phi \oplus \psi) & \leftarrow & \Box\phi \\
\Diamond(\phi \otimes \psi) & \leftrightarrow & \Diamond(\phi \otimes \Diamond\psi) & & &
\end{array}
\tag{40}
$$

Note that hypotheticals hold immediately, *i.e.*, over paths of length 1 and so they do not cause any real state transitions. This is reflected formally by the following equivalence:

$$\mathtt{state} \;\equiv\; \Diamond\mathtt{path}$$

where, as before, `path` is the special atom that is true on every path, and `state` is the special atom that is true on every path of length 1 (*i.e.*, at every state). To see that this equivalence holds, note that, by definition, $\Diamond$`path` is true *only* on paths of length 1. Hence, $\mathbf{M}, \pi \models \Diamond\mathtt{path}$ if and only if π is path of length 1, *i.e.*, if and only if $\mathbf{M}, \pi \models \mathtt{state}$. (It is shown in Section 5.2 that `state` cannot be expressed in $\mathcal{TR}$ without hypothetical operators.) Hypotheticals, thus, increase the expressive power of our language, since we no longer need the special atom `state`. Henceforth, we shall treat `state` as an abbreviation for $\Diamond$`path`. In fact, since `path` is just an abbreviation for $\phi \vee \neg\phi$, for any ϕ, we no longer need any predefined atoms at all.

Unlike the standard modal operators, our "$\Diamond$" and "$\Box$" are not dual to each other in the normal sense; that is, $\Box\phi \not\equiv \neg\Diamond\neg\phi$ and $\Diamond\phi \not\equiv \neg\Box\neg\phi$. Instead, we have the following two equivalences:

$$\Box\phi \;\equiv\; \mathtt{state} \wedge \neg\Diamond\neg\phi \qquad\qquad \Diamond\phi \;\equiv\; \mathtt{state} \wedge \neg\Box\neg\phi$$

However, it is not hard to define new hypotheticals, $\blacklozenge$ and $\blacksquare$, for which the usual duality holds. They are based on the idea of a continuation path: a path π' is a *continuation* of path π if and only if π is a prefix of π'. Thus, $\langle \mathbf{D}_1\ \mathbf{D}_2\ \mathbf{D}_3\ \mathbf{D}_4 \rangle$ is a continuation of $\langle \mathbf{D}_1\ \mathbf{D}_2 \rangle$.

- $\mathbf{M}, \pi \models \blacklozenge\phi$ if and only if there is a continuation, π', of π such that $\mathbf{M}, \pi' \models \phi$ holds.
- $\mathbf{M}, \pi \models \blacksquare\phi$ if and only if for every continuation, π', of π, it is the case that $\mathbf{M}, \pi' \models \phi$.

It is not difficult to see that $\blacklozenge\phi \equiv \neg\blacksquare\neg\phi$ and $\blacksquare\phi \equiv \neg\blacklozenge\neg\phi$. The previous form of hypotheticals can now be expressed as follows: $\Diamond\phi \equiv \mathtt{state} \wedge \blacklozenge\phi$, and $\Box\phi \equiv \mathtt{state} \wedge \blacksquare\phi$. However, the proposition `state` itself cannot be expressed using these new hypothetical operators, as it can with the old ones. Therefore, the expressive power of the new hypotheticals is not the same. The operators "$\blacksquare$" and "$\blacklozenge$" are useful for reasoning about program execution, and are related to the operator "**suff**" in the version of Process Logic described in [50]. In this paper, however, we shall consider "$\Diamond$" and "$\Box$" only.

8.2 Retrospection

Before proceeding to the proof theory, we introduce another class of modal formulas related to hypotheticals. We call them *retrospective* formulas (abbr., *retrospectives*). Intuitively, a retrospective formula $\phi\Diamond$ tests whether we *could have* executed ϕ to reach the current state. Likewise, $\phi\Box$ tests whether we *must have* executed ϕ to reach the current state.

The semantics of the retrospectives is dual to that of hypotheticals. This duality is closely related to the duality between normal and reverse execution of transactions introduced in Section 6.3. This connection will be seen clearly in the proof theory in the next subsection. To define the meaning of the retrospectives formally, let $\mathbf{M} = \langle U, \mathcal{F}, I_{path}\rangle$ be a path structure, let $\mathbf{D}$ be a state, and let ν be a variable assignment. Then

- $\mathbf{M}, \langle\mathbf{D}\rangle \models_\nu \phi\Diamond$ if and only if there is a path π *ending* at state $\mathbf{D}$ such that $\mathbf{M}, \pi \models_\nu \phi$ holds.
- $\mathbf{M}, \langle\mathbf{D}\rangle \models_\nu \phi\Box$ if and only if for every path π ending at state $\mathbf{D}$, it is the case that $\mathbf{M}, \pi \models_\nu \phi$.

Again, $\langle\mathbf{D}\rangle$ is the path of length 1 made from state $\mathbf{D}$. The following tautologies are analogous to (40):

$$\begin{array}{rcl}
\phi\Diamond\Diamond & \leftrightarrow & \phi\Diamond \\
(\phi \vee \psi)\Diamond & \leftrightarrow & \phi\Diamond \vee \psi\Diamond \\
(\phi \wedge \psi)\Diamond & \rightarrow & \phi\Diamond \wedge \psi\Diamond \\
(\phi \otimes \psi)\Diamond & \rightarrow & \psi\Diamond \\
(\phi \otimes \psi)\Diamond & \leftrightarrow & (\phi\Diamond \otimes \psi)\Diamond
\end{array}
\qquad
\begin{array}{rcl}
\phi\Box\Box & \leftrightarrow & \phi\Box \\
(\phi \wedge \psi)\Box & \leftrightarrow & \phi\Box \wedge \psi\Box \\
(\phi \vee \psi)\Box & \leftarrow & \phi\Box \vee \psi\Box \\
(\phi \oplus \psi)\Box & \leftarrow & \psi\Box
\end{array}$$

Retrospective formulas are not dual to each other in the usual sense, just as hypothetical operators are not. However, we can define a version of retrospectives that obeys duality, just as in the case of hypotheticals.

8.3 Proof Theory for Hypotheticals and Retrospectives

Hypothetical and retrospective formulas are interesting in their own right, but in addition, they bind together the two proof theories of Section 6.2, systems $\Im^I$ and $\Im^{II}$. We demonstrate this in stages. First, we expand system $\Im^I$ so that it infers hypotheticals. Likewise, we expand system $\Im^{II}$ so that it infers retrospectives. We then combine the two into a single, unified inference system.

8.3.1 $\Diamond$-Serial-Horn Programs

The first step is to generalize the idea of serial Horn rules to include hypotheticals and retrospectives. The generalization allows the possibility operator, $\Diamond$, in rules bodies, but not the necessity operator, $\Box$. This is analogous to allowing existential quantifiers in the bodies of classical Horn rules, but disallowing universal quantifiers. Our extension of serial-Horn rules is thus called $\Diamond$-serial-Horn rules. A *$\Diamond$-serial-Horn* rule is a formula of the form $a \longleftarrow \phi$, where a is atomic and ϕ is a *$\Diamond$-serial goal*, defined as follows:

Definition 8.1 ($\Diamond$-Serial-Horn Goals)

- an atomic formula is a $\Diamond$-serial goal.

- if ψ_1 ... ψ_k are $\diamond$-serial-Horn goals, then so is $\psi_1 \otimes \cdots \otimes \psi_k$.
- if ψ is a $\diamond$-serial-Horn goal, then so are $\diamond\psi$ and $\psi\diamond$.

$\square$

The following formulas are all $\diamond$-serial-Horn goals:

$$a(X) \otimes b(X) \qquad a(X) \otimes \diamond b(X) \qquad a(X)\diamond \otimes b(X) \qquad \diamond a(X) \otimes b(X)\diamond$$

$$\diamond[a(X) \otimes \diamond b(X)] \qquad [a(X) \otimes \diamond(b(X) \otimes c(X)\diamond) \otimes d(X)]\diamond\diamond$$

We define a $\diamond$*-serial-Horn transaction* to be an existentially quantified $\diamond$-serial goal, which we write as $(\exists)\,\phi$, where ϕ is a $\diamond$-serial goal. As a special case, any serial-Horn transaction (or rule) is also a $\diamond$-serial-Horn transaction (or rule). We define a $\diamond$*-serial-Horn program* to be any finite set of $\diamond$-serial-Horn rules. The definition of data oracles needs no modification, though we still require them to satisfy the conditions of Definition 6.2, such as independence from the transaction base, etc.

In the above definitions, we assume that special atoms `state`, `arc` and `path` are implemented via the oracles and the transaction base, as described in Section 6.1. This simplifies the theoretical development, but does not reduce the expressive power of $\diamond$-serial-Horn rules.

8.3.2 Inference System $\Im^{\diamond}$

We now present a proof theory that amalgamates our earlier inference systems, $\Im^{I}$ and $\Im^{II}$, and that is sound and complete for $\diamond$-serial Horn programs. The new system, called $\Im^{\diamond}$, has ten inference rules: the six rules of systems $\Im^{I}$ and $\Im^{II}$, plus four new rules—4, 4′ and 5, 5′—defined below. The axioms for the new system are just the axioms for $\Im^{I}$ plus the axioms for $\Im^{II}$. Intuitively, $\Im^{\diamond}$ combines normal execution and reverse execution in a single inference system. This generality is needed since hypotheticals correspond to forward execution and retrospectives correspond to reverse execution.

Each of the four new rules is based on two ideas: (i) combining two executions from the same database state into a single serial execution, and (ii) treating one of the two executions as hypothetical or retrospective. Since there are two kinds of execution (normal and reverse), executions can be paired in four possible ways, which is why there are four new inference rules. For clarity, we divide these rules into two types. Rule 4 extends system $\Im^{I}$ to include hypotheticals, and Rule 4′ extends system $\Im^{II}$ to include retrospectives. Rules 5 and 5′ then combine these two systems into a single system for arbitrary combinations of hypothetical and retrospective formulas.

Introduction of modalities:

Let σ be a substitution, and *rest* and *test* be a $\diamond$-serial goal. Then,

4. $$\frac{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, test\,\sigma \qquad \mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, (\diamond test) \otimes rest} \qquad \text{if } test\,\sigma \text{ and } rest\,\sigma \text{ share no variables.}$$

4′. $$\frac{\mathbf{P}, \text{---}\,\mathbf{D} \vdash (\exists)\, rest\,\sigma \qquad \mathbf{P}, \text{---}\,\mathbf{D} \vdash (\exists)\, test\,\sigma}{\mathbf{P}, \text{---}\,\mathbf{D} \vdash (\exists)\, rest \otimes (test\diamond)} \qquad \text{if } test\,\sigma \text{ and } rest\,\sigma \text{ share no variables.}$$

Intuitively, Rule 4 combines two normal executions into a single normal execution, converting one execution to a hypothetical. Likewise, Rule 4′ combines two reverse executions into a single reverse execution, converting one execution to a retrospective. We can also interpret these rules from the perspective of top-down inference. From this view, to execute $\Diamond test \otimes rest$, Rule 4 separates the hypothetical from the rest of the transaction, and executes it separately, in the background. Likewise, to execute $rest \otimes test\Diamond$, Rule $4'$ separates the retrospective from the rest of the transaction, and executes it separately.

The requirement that $test\,\sigma$ and $rest\,\sigma$ must share no variables simplifies the proof theory. It implies that unification will involve only one sequent at a time. In terms of top-down SLD inference, it means that any variables common to *test* and *rest* should be bound to ground terms before *test* is executed. This kind of requirement is common in logic programming systems. For example, Prolog systems often require that variables in negative goal literal be bound to ground terms before the negative literal is executed. Of course, unlike negation-as-failure, hypotheticals have a well-defined, monotonic semantics, and there are no problems with recursion through hypotheticals.

By adding rules 4 and $4'$ to inference systems $\Im^I$ and $\Im^{II}$, respectively, we get two new inference systems, one for hypotheticals and one for retrospectives. Taken together, Rules 1–4 are sound and complete for $\Diamond$-serial-Horn transaction bases that use hypothetical operators only (where Rules 1–3 come from system $\Im^I$). Likewise, inference Rules $1'$–$4'$ are sound and complete for transaction bases that use retrospective operators only (where Rules $1'$–$3'$ come from system $\Im^{II}$).

Note that Rules 4 and $4'$ use different kinds of sequents. One kind has the form $\mathbf{P}, \mathbf{D} \text{---} \vdash \phi$, as in inference system $\Im^I$. Here, $\mathbf{D}$ is the *initial* state in the execution of transaction ψ. The other kind of sequent has the form $\mathbf{P}, \text{---}\, \mathbf{D} \vdash \phi$, as in $\Im^{II}$. Here, $\mathbf{D}$ is the *final* state in the execution of ψ. So far, these two kinds of sequents have been independent of each other, isolated in separate inference systems. However, to execute transactions that contain both hypotheticals and retrospectives, we need both kinds of sequents. This is because the current database state can be both the *initial* state of a hypothetical execution, and the *final* state of a retrospective execution. The following inference rules capture this idea, integrating the two kinds of sequents into a single inference system.

Mixed introduction of modalities:

Let σ be a substitution, and *rest* and *test* be a $\Diamond$-serial goal. Then,

$$\mathbf{5.}\quad \frac{\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\, test\,\sigma \qquad \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, (test\Diamond) \otimes rest} \qquad \text{if } test\,\sigma \text{ and } rest\,\sigma \text{ share no variables.}$$

$$\mathbf{5'}.\quad \frac{\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\, rest\,\sigma \qquad \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, test\,\sigma}{\mathbf{P}, \text{---}\, \mathbf{D} \vdash (\exists)\, rest \otimes (\Diamond test)} \qquad \text{if } test\,\sigma \text{ and } rest\,\sigma \text{ share no variables.}$$

Intuitively, Rule 5 combines a reverse execution and a normal execution into a single normal execution, converting the reverse execution to a retrospective. Likewise, Rule $5'$ combines a reverse execution and a normal execution into a single reverse execution, converting the normal execution to a hypothetical. We can also interpret these rules from the perspective of top-down inference. From this view, to execute $test\Diamond \otimes rest$, Rule 5 separates the retrospective from the rest of the transaction, and executes it in reverse, but in the background. Likewise, to execute $rest \otimes \Diamond test$, Rule $5'$ separates the hypothetical from the rest of the transaction, and executes it normally, but in the background.

In Definition 6.4 we stated the general notion of deduction. Since this definition is applicable to any inference system, it is applicable to $\Im^\Diamond$ in particular. Using it, we can state the following soundness and completeness result, which is analogous to Theorem 6.5.

Theorem 8.2 (Soundness and Completeness for $\Im^\Diamond$) *If the $\Diamond$-serial-Horn conditions are satisfied, then*

- $\mathbf{P}, \mathbf{D}\text{---} \models (\exists)\,\phi$ *if and only if there is a deduction in* $\Im^{\diamond}$ *of the sequent* $\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,\phi$.
- $\mathbf{P}, \text{---}\mathbf{D} \models (\exists)\,\phi$ *if and only if there is a deduction in* $\Im^{\diamond}$ *of the sequent* $\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\,\phi$.

Proof: See Appendices D and E. $\square$

It is worth noting that with hypotheticals it is possible to reformulate our inference systems using only one type of sequent, $\mathbf{P}, \mathbf{D} \vdash \phi$. These sequents correspond to statements of the form $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \phi$ with $n = 0$, as given in Definition 5.6. The reformulation itself, which we leave as an exercise for the reader, is made possible by the following result.

Lemma 8.3 *If the* $\diamond$*-serial-Horn conditions are satisfied, then*

- $\mathbf{P}, \mathbf{D}\text{---} \models (\exists)\,\phi$ *if and only if* $\mathbf{P}, \mathbf{D} \models (\exists)\,\diamond\phi$
- $\mathbf{P}, \text{---}\mathbf{D} \models (\exists)\,\phi$ *if and only if* $\mathbf{P}, \mathbf{D} \models (\exists)\,\phi\diamond$

8.3.3 Example: Hypotheticals

This section gives an example of inference when real and hypothetical updates are mixed in a single transaction. The example, which uses only one kind of sequent, also illustrates the use of inference rule 4. Unlike the deductions of Section 6, the deduction in this section is not linear, but has a tree-like structure.

The example centers on the transaction $?-\ p \otimes \diamond q \otimes r$. This transaction consists of three subtransactions, p, q and r, where first p is executed, then q is executed hypothetically, and finally r is executed. Because q executes hypothetically, all its updates are rolled back before r starts executing. Thus, $\diamond q$ acts as a test, one that must be satisfied before r can execute. In contrast, updates due to p and r are not rolled back and have a permanent affect on the database, *i.e.*, they are committed.

Execution of the transaction is recorded in the deduction below, which we describe from a top-down perspective. In this deduction, the initial database is empty, and the transaction base contains the following three rules:

$$p \leftarrow a.ins \otimes b.ins \qquad\qquad q \leftarrow c.ins \otimes d.ins \otimes b \otimes d \qquad\qquad r \leftarrow e.ins \otimes f.ins \otimes g.ins$$

Top-down deduction proceeds by first trying to execute p, just as in system $\Im^{I}$. Next, inference rule 4 is applied, which separates the hypothetical test, $\diamond q$, from the rest of the transaction:

$$\begin{array}{lll}
 & \mathbf{P}, \{\}\text{---} \vdash p \otimes \diamond q \otimes r & \\
\textit{if} & \mathbf{P}, \{\}\text{---} \vdash a.ins \otimes b.ins \otimes \diamond q \otimes r & \text{by inference rule 1,} \\
\textit{if} & \mathbf{P}, \{a\}\text{---} \vdash b.ins \otimes \diamond q \otimes r & \text{by inference rule 3,} \\
\textit{if} & \mathbf{P}, \{a, b\}\text{---} \vdash \diamond q \otimes r & \text{by inference rule 3,} \\
\textit{if} & \mathbf{P}, \{a, b\}\text{---} \vdash q & \\
\textit{and} & \mathbf{P}, \{a, b\}\text{---} \vdash r & \text{by inference rule 4.}
\end{array}$$

Notice how inference rule 4 has split the sequent $\mathbf{P}, \{a, b\}\text{---} \vdash \diamond q \otimes r$ into two sequents. Intuitively, one sequent is for the execution of q, and the other is for the execution of r. The proof now splits into two branches, each proceeding exactly as in inference system $\Im^{I}$.

$$
\begin{array}{llll}
\text{The branch for proving } q: & & \mathbf{P}, \{a,b\}\text{---} \vdash q & \\
 & \textit{if} & \mathbf{P}, \{a,b\}\text{---} \vdash c.ins \otimes d.ins \otimes b \otimes d & \text{by inference rule 1,} \\
 & \textit{if} & \mathbf{P}, \{a,b,c\}\text{---} \vdash d.ins \otimes b \otimes d & \text{by inference rule 3,} \\
 & \textit{if} & \mathbf{P}, \{a,b,c,d\}\text{---} \vdash b \otimes d & \text{by inference rule 3,} \\
 & \textit{if} & \mathbf{P}, \{a,b,c,d\}\text{---} \vdash d & \text{by inference rule 2,} \\
 & \textit{if} & \mathbf{P}, \{a,b,c,d\}\text{---} \vdash () & \text{by inference rule 2.} \\
\text{The branch for proving } r: & & \mathbf{P}, \{a,b\}\text{---} \vdash r & \\
 & \textit{if} & \mathbf{P}, \{a,b\}\text{---} \vdash e.ins \otimes f.ins \otimes g.ins & \text{by inference rule 1,} \\
 & \textit{if} & \mathbf{P}, \{a,b,e\}\text{---} \vdash f.ins \otimes g.ins & \text{by inference rule 3,} \\
 & \textit{if} & \mathbf{P}, \{a,b,e,f\}\text{---} \vdash g.ins & \text{by inference rule 3,} \\
 & \textit{if} & \mathbf{P}, \{a,b,e,f,g\}\text{---} \vdash () & \text{by inference rule 3.}
\end{array}
$$

Both branches terminate with an axiom sequent, so they both succeed, and so the entire deduction succeeds.

Notice that the two branches both begin at the same database state, $\{a,b\}$. Thus, the branch for r ignores all the updates carried out by the branch for q. Intuitively, this is because q's updates are hypothetical, and are thus rolled back before r starts executing. Also note that the two branches end at different states, $\{a,b,c,d\}$ and $\{a,b,e,f,g\}$. However, only the latter state is the result of committed updates, so it represents the new state of the database after transaction execution.

8.3.4 Example: Retrospectives

This section modifies the example of Section 8.3.3 so that it combines real and retrospective updates in a single transaction. The example also illustrates the use of inference rule 5 to change from one kind of sequent to another. Again, unlike the deductions of Section 6, the deduction in this section is not linear, but tree-like.

The example centers on the transaction $p \otimes q\Diamond \otimes r$. This transaction consists of three subtransactions, p, q and r, where first p is executed, then q is executed retrospectively, and finally r is executed. Because q executes retrospectively, all its updates are rolled back before r starts executing. In contrast, updates due to p and r are committed.

Execution of the transaction is recorded in the following top-down deduction. In this deduction, the initial database is empty, and the transaction base contains the following three rules:

$$
p \leftarrow a.ins \otimes b.ins \qquad q \leftarrow d \otimes b \otimes d.del \otimes c.del \qquad r \leftarrow e.ins \otimes f.ins \otimes g.ins
$$

Top-down deduction proceeds by first trying to prove p, just as in system $\Im^I$. Next, inference rule 5 is applied, which separates the retrospective test, $q\Diamond$, from the rest of the transaction:

$$
\begin{array}{lll}
 & \mathbf{P}, \{\}\text{---} \vdash p \otimes q\Diamond \otimes r & \\
\textit{if} & \mathbf{P}, \{\}\text{---} \vdash a.ins \otimes b.ins \otimes q\Diamond \otimes r & \text{by inference rule 1,} \\
\textit{if} & \mathbf{P}, \{a\}\text{---} \vdash b.ins \otimes q\Diamond \otimes r & \text{by inference rule 3,} \\
\textit{if} & \mathbf{P}, \{a,b\}\text{---} \vdash q\Diamond \otimes r & \text{by inference rule 3,} \\
\textit{if} & \mathbf{P}, \text{---}\{a,b\} \vdash q & \\
\textit{and} & \mathbf{P}, \{a,b\}\text{---} \vdash r & \text{by inference rule 5.}
\end{array}
$$

Notice how inference rule 5 has split the fourth sequent (involving $q\Diamond \otimes r$) into two, different-style sequents: one is for the reverse execution of q, and the other is for the normal execution of r. The proof now splits into two branches, one using inference system $\Im^{II}$, and the other using $\Im^{I}$. In this way, system $\Im^{\Diamond}$ integrates $\Im^{I}$ and $\Im^{II}$ in one system.

The branch for proving q:		$\mathbf{P}, \text{---}\, \{a,b\} \vdash q$	
	if	$\mathbf{P}, \text{---}\, \{a,b\} \vdash d \otimes b \otimes d.del \otimes c.del$	by inference rule $1'$,
	if	$\mathbf{P}, \text{---}\, \{a,b,c\} \vdash d \otimes b \otimes d.del$	by inference rule $3'$,
	if	$\mathbf{P}, \text{---}\, \{a,b,c,d\} \vdash d \otimes b$	by inference rule $3'$,
	if	$\mathbf{P}, \text{---}\, \{a,b,c,d\} \vdash d$	by inference rule $2'$,
	if	$\mathbf{P}, \text{---}\, \{a,b,c,d\} \vdash ()$	by inference rule $2'$.
The branch for proving r:		$\mathbf{P}, \{a,b\}\, \text{---} \vdash r$	
	if	$\mathbf{P}, \{a,b\}\, \text{---} \vdash e.ins \otimes f.ins \otimes g.ins$	by inference rule 1,
	if	$\mathbf{P}, \{a,b,e\}\, \text{---} \vdash f.ins \otimes g.ins$	by inference rule 3,
	if	$\mathbf{P}, \{a,b,e,f\}\, \text{---} \vdash g.ins$	by inference rule 3,
	if	$\mathbf{P}, \{a,b,e,f,g\}\, \text{---} \vdash ()$	by inference rule 3.

Both branches terminate with an axiom sequent, so they both succeed, and so does the entire deduction.

As in the previous subsection, the branch for r ignores all the updates carried out by the branch that proves q, as q's updates are retrospective and thus are rolled back before r starts executing. Likewise, the two branches end at different states, $\{a,b,c,d\}$ and $\{a,b,e,f,g\}$, but only the latter state is the result of committed updates, so it represents the new state of the database after transaction execution.

8.3.5 Execution as Deduction in $\Im^{\Diamond}$

According to Theorem 8.2, inference system $\Im^{\Diamond}$ allows us to prove statements of the form $\mathbf{P}, \text{---}\,\mathbf{D} \models (\exists)\,\phi$ and $\mathbf{P}, \mathbf{D}\,\text{---} \models (\exists)\,\phi$. By themselves, these statements tell us that a transaction succeeds from a given database state, $\mathbf{D}$; but they do *not* tell us what the resulting database state is. For this reason, we also want to prove statements of the form

$$\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models (\exists)\,\phi \tag{41}$$

In particular, given the initial state, $\mathbf{D}_0$, or the final state, $\mathbf{D}_n$, we want to infer the other states. Intuitively, this corresponds to forward and reverse execution, respectively.

This problem also arose in the development of systems $\Im^{I}$ and $\Im^{II}$ in Section 6. There, we addressed the problem by defining the notion of *executional deduction*. The idea was to weed out spurious sequents so that the execution path, $\mathbf{D}_0, \ldots, \mathbf{D}_n$, could easily be extracted from a proof. For systems $\Im^{I}$ and $\Im^{II}$, this was easy because each inference rule has exactly one premise and one consequent. Executional deductions thus have a linear structure, in which each sequent is derived from the one immediately preceding it.

In system $\Im^{\Diamond}$, things are more complex, because the new inference rules in this system have *two* antecedents. An executional deduction is therefore structured as a *binary tree*. In this tree, each node is a sequent, where leaf nodes are axioms, and internal nodes are derived from their children. Intuitively, one branch in this tree represents the execution of the committed portion of the user's transaction. Tributaries of this branch represent the execution of hypothetical or retrospective subtransactions.

We now formalize these ideas, beginning with the following definition.

Definition 8.4 (Executional Deduction in $\Im^{\diamond}$) Let $\mathbf{P}$ be a transaction base. An *executional deduction* in $\Im^{\diamond}$ of a transaction $(\exists)\,\phi$ is a binary tree of sequents that satisfy the following conditions:

1. The root of the tree is a sequent of the form $\mathbf{P}, \mathbf{D} \,\text{---}\, \vdash (\exists)\,\phi$ or $\mathbf{P}, \,\text{---}\, \mathbf{D} \vdash (\exists)\,\phi$, for some database $\mathbf{D}$.
2. The leaves of the tree are axiom-sequents, *i.e.*, sequents of the form $\mathbf{P}, \mathbf{D}' \,\text{---}\, \vdash (\,)$ or $\mathbf{P}, \,\text{---}\, \mathbf{D}' \vdash (\,)$, for some database $\mathbf{D}'$.
3. Each non-leaf node can be derived from its children by applying one of the inference rules of system $\Im^{\diamond}$; *i.e.*, the node is the consequent of some rule for which the children are the antecedents.

□

It is not hard to see that any topological sort of the sequents in an executional deduction yields a deduction in the general sense of Definition 6.4. The example deductions given in Sections 8.3.3 and 8.3.4 are both executional.

A close examination of the inference rules of System $\Im^{\diamond}$ reveals that in each rule, the consequent involves a formula called *rest*. This is the committed part of the transaction, and in each rule, exactly one antecedent involves this formula. We call this antecedent the *committed antecedent* of the rule. For example, in inference rules 1–3 and 1′–3′, there is only one antecedent, and it is committed. In inference rules 4, 4′ and 5, 5′, there are two antecedents, one is committed, and the other is hypothetical or retrospective.

We can extend this idea to executional deductions. First, in deriving a node (sequent) from its children, we apply a single inference rule. Exactly one of the children corresponds to the committed antecedent of the rule. We call it the *committed child* of the node. Second, starting from the root of the binary tree, we trace a path to a leaf by always moving from the current node to its committed child. This defines a branch of the tree, which we call the *committed branch.* The leaf at the end of this branch is called the *committed leaf.* Intuitively, the committed branch describes a sequence of permanent database updates, and the committed leaf describes the final database state. In contrast, tributaries of the committed branch describe sequences of temporary updates.

The committed path for the deduction in Section 8.3.4 is given in Figure 3. The top-most sequent is the root, and the bottom-most sequent is the committed leaf. Reading the path from top-to-bottom, we see an execution of p followed by an execution of q. Notice that execution of the retrospective, $q\diamond$, is completely ignored. This path is therefore almost identical to the executional deduction for the non-hypothetical transaction $p \otimes r$.

Examining the inference rules, it is not hard to see that every executional deduction has the following property: all sequents on the committed branch have the same style—they are either all of the form $\mathbf{P}, \mathbf{D} \,\text{---}\, \vdash \phi$, as in system $\Im^{I}$, or all of the form $\mathbf{P}, \,\text{---}\, \mathbf{D} \vdash \phi$, as in system $\Im^{II}$. In Figure 3, for instance, the sequents are all $\Im^{I}$-style.

It is now easy to define the notion of the execution path of a deduction in $\Im^{\diamond}$. Consider the committed branch of such a deduction. From the above observation, we know that all sequents on this branch have the same style. If they are $\Im^{I}$-style sequents, then as in $\Im^{I}$-executional deductions,

$\mathbf{P}, \{\} \mbox{---}\vdash p \otimes q\Diamond \otimes r$	
$\mathbf{P}, \{\} \mbox{---}\vdash a.ins \otimes b.ins \otimes q\Diamond \otimes r$	
$\mathbf{P}, \{a\} \mbox{---}\vdash b.ins \otimes q\Diamond \otimes r$	by inference rule 3,
$\mathbf{P}, \{a,b\} \mbox{---}\vdash q\Diamond \otimes r$	by inference rule 3,
$\mathbf{P}, \{a,b\} \mbox{---}\vdash r$	
$\mathbf{P}, \{a,b\} \mbox{---}\vdash e.ins \otimes f.ins \otimes g.ins$	
$\mathbf{P}, \{a,b,e\} \mbox{---}\vdash f.ins \otimes g.ins$	by inference rule 3,
$\mathbf{P}, \{a,b,e,f\} \mbox{---}\vdash g.ins$	by inference rule 3,
$\mathbf{P}, \{a,b,e,f,g\} \mbox{---}\vdash ()$	by inference rule 3.

Figure 3: The Committed Branch of an Executional Deduction

the committed branch has the structure depicted in Figure 1 of Section 6.3.1. The execution path of this deduction is $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$ — a sequence of states picked out each time an inference is made via Rule 3. Likewise, if the sequents on the committed branch are $\Im^{II}$-style, then the branch has the structure depicted in Figure 2 (Section 6.4). The execution path of this deduction has the same form, $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_n$, but the states are picked out via Rule 3′. Thus, for the committed branch in Figure 3, the execution path is

$$\{\},\ \{a\},\ \{a,b\},\ \{a,b,e\},\ \{a,b,e,f\},\ \{a,b,e,f,g\}$$

The next theorem relates the notion of execution path and the notion of executional entailment (see Definition 5.6).

Theorem 8.5 (Executional Soundness and Completeness of $\Im^{\Diamond}$) *Under the $\Diamond$-serial-Horn conditions, the following statements are equivalent:*

1. $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\, \phi$.
2. *There is an executional deduction in $\Im^{\Diamond}$ of $(\exists)\, \phi$ whose execution path is $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$.*

Proof: See Appendix F. □

As an example, using the transaction base from Section 8.3.4, we can make the following statement:

$$\mathbf{P}, \{\}, \{a\}, \{a,b\}, \{a,b,e\}, \{a,b,e,f\}, \{a,b,e,f,g\} \models p \otimes q\Diamond \otimes r$$

At this point, it is instructive to look at the special case in which a transaction is purely hypothetical, *i.e.*, has the form $\Diamond\phi$. In such cases, the committed branch consists of exactly two sequents, and the execution path consists of a single database state. Consider, for instance, the transaction $\Diamond(b.ins \otimes c.ins)$ applied to the database $\{a\}$. This defines the root of an executional deduction. Using top-down inference, we can generate the rest of the deduction. The first step, is to apply inference rule 4 with $test = b.ins \otimes c.ins$ and $rest = ()$. This generates two children for the root. In fact, the entire deduction consists of the following two branches:

$$\begin{array}{ll} \mathbf{P}, \{a\} \text{---} \vdash \Diamond(b.ins \otimes c.ins) & \mathbf{P}, \{a\} \text{---} \vdash \Diamond(b.ins \otimes c.ins) \\ \mathbf{P}, \{a\} \text{---} \vdash b.ins \otimes c.ins & \mathbf{P}, \{a\} \text{---} \vdash () \\ \mathbf{P}, \{a, b\} \text{---} \vdash c.ins & \\ \mathbf{P}, \{a, b, c\} \text{---} \vdash () & \end{array}$$

The left-hand branch is hypothetical, and the right-hand branch is committed. From the committed branch, we see that the execution path consists of the single state $\{a\}$. Thus, using Theorem 8.5, we can make the following formal statement:

$$\mathbf{P}, \{a\} \models \Diamond(b.ins \otimes c.ins)$$

Because the execution path consists of just a single state, hypothetical transactions leave the database unchanged.

8.3.6 Normal and Reverse Execution in $\Im^{\Diamond}$

We are now in a position to extend the ideas of section 6.3.2 to system $\Im^{\Diamond}$. That is, we can define what it means to execute a $\Diamond$-serial-Horn transaction, thus making precise a notion that we have been using informally in this section.

As mentioned earlier, given a database $\mathbf{D}_0$ and a transaction formula $(\exists)\,\psi$, we wish to prove the statement

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models (\exists)\,\phi \tag{42}$$

for some sequence of databases, $\mathbf{D}_1, \ldots, \mathbf{D}_n$. The important point is that these latter databases are *unknown* at the outset and must be computed during the proof. The process of constructing such a proof is called a *normal execution* of $(\exists)\,\phi$ with respect to $\mathbf{P}$. Thus, given the state $\mathbf{D}_0$, normal execution determines whether the transaction $(\exists)\,\phi$ can start from state $\mathbf{D}_0$, and if so, it computes the successor states, $\mathbf{D}_1, \ldots, \mathbf{D}_n$. Intuitively, this computation corresponds to updating the user's database from $\mathbf{D}_0$ to $\mathbf{D}_1$ ... to $\mathbf{D}_n$.

Likewise, *reverse execution* is the process of constructing a proof of (42) when $\mathbf{D}_n$ is known at the outset and $\mathbf{D}_0, \ldots, \mathbf{D}_{n-1}$ are unknown. Reverse execution thus determines whether the transaction $(\exists)\,\phi$ can *terminate* at state $\mathbf{D}_n$, and if so, it computes the predecessor states, $\mathbf{D}_0, \ldots, \mathbf{D}_{n-1}$. Intuitively, this computation corresponds to updating the user's database in reverse, from $\mathbf{D}_n$ to $\mathbf{D}_{n-1}$ ... to $\mathbf{D}_0$.

Inference system $\Im^{\Diamond}$ can execute $\Diamond$-serial-Horn transactions normally and in reverse. All that is needed is to construct an executional deduction of the transaction, from which the execution path can be extracted. In practice, the direction of execution will depend on how the committed branch is constructed, top-down or bottom-up. For instance, if the user's current database is empty, and if the branch in Figure 3 is constructed from the top, down, then the transaction $p \otimes \Diamond q \otimes r$ is executed normally, updating the database from $\{\}$ to $\{a, b, e, f, g\}$. On the other hand, if the user's current database is $\{a, b, e, f, g\}$, and if the branch in Figure 3 is constructed from the bottom, up, then the transaction is executed *in reverse*, updating the database from $\{a, b, e, f, g\}$ to $\{\}$. These ideas were discussed in greater detail in Section 6.3.2.

In Figure 3, the committed branch involves sequents of the form $\mathbf{P}, \mathbf{D} \text{---} \vdash \phi$. However, if the committed branch contained sequents of the other form, $\mathbf{P}, \text{---}\, \mathbf{D} \vdash \phi$, then a dual relationship would hold between the directions of inference and execution. That is, top-down inference would naturally correspond to reverse execution, and bottom-up inference would naturally correspond to normal execution. Thus, as in Section 6, the following properties are closely related: (i) the style of sequent on the committed branch, (ii) the direction of inference on the committed branch, and (iii) the direction of execution. The exact relationship is summarized in the table below.

Sequent Style	Inference Direction	Execution Direction
$\mathbf{P}, \mathbf{D}\text{---}\vdash \phi$	top-down	forward
$\mathbf{P}, \mathbf{D}\text{---}\vdash \phi$	bottom-up	reverse
$\mathbf{P}, \text{---}\mathbf{D} \vdash \phi$	top-down	reverse
$\mathbf{P}, \text{---}\mathbf{D} \vdash \phi$	bottom-up	forward

There is a subtlety about inference in $\Im^{\Diamond}$ that should be pointed out: Although the committed branch of a deduction may be constructed in a top-down or a bottom-up manner, all tributaries of this branch (*i.e.*, its side branches) must be computed top-down in any practical system. The reason for this is quite simple. In $\Im^{\Diamond}$, an executional deduction is a tree. Constructing this tree bottom-up means starting at the leaves and building up towards the root. However, before starting this process, we must know what sequent is at every leaf of the tree. Knowing this is a practical impossibility. In practice, we will know only the current database state, $\mathbf{D}$, and the user's transaction, ϕ. This knowledge is enough to specify only four sequents, the goal sequents $\mathbf{P}, \mathbf{D}\text{---}\vdash \phi$ and $\mathbf{P}, \text{---}\mathbf{D} \vdash \phi$, and the axiom sequents $\mathbf{P}, \mathbf{D}\text{---}\vdash ()$ and $\mathbf{P}, \text{---}\mathbf{D} \vdash ()$. In practice, therefore, we can start at the top of a committed branch or at its leaf, for either sequent style. However, starting inference at any other leaf means knowing the state at which a hypothetical subtransaction terminates. This knowledge simply is not available to us *a priori*. In practice, therefore, hypothetical and retrospective branches must be constructed top-down.

For example, consider the deduction in Section 8.3.3. Whether we construct the committed branch top-down or bottom-up, we reach eventually the sequent $\mathbf{P}, \{a, b\}\text{---}\vdash \Diamond q \otimes r$. At this point, we can construct the branch for q. But since we are now positioned at the top of this branch, we have little choice but to construct it in a top-down fashion. A similar observation is true for the deduction in Section 8.3.4.

In closing this section, we remark that it is possible to develop other inference systems for $\Diamond$-serial-Horn programs, systems that can operate in a completely bottom-up manner. (We leave this as an exercise.) However, such systems cannot operate top-down.

8.4 Applications of Hypotheticals

In Section 8.1, we saw that hypothetical operators endow $\mathcal{TR}$ with additional expressive power. In this subsection we explore a number of non-trivial applications of hypothetical operators, such as subjunctive statements and imperative programming constructs. Most of the examples go beyond the scope of the proof theory of Section 8.3, though they can be handled by the general proof-theory in [21]. Furthermore, the examples in Section 8.4.2 rely on the closed world assumption, and thus they cannot be adequately handled by *any* proof theory. However, as in Prolog, a marriage of the proof theory in Section 8.3 with negation-as-finite-failure might be a practical solution.

8.4.1 Subjunctive Queries and Counterfactuals

An important application for hypothetical operators in $\mathcal{TR}$ is in the area of, so called, *subjunctive* queries [45], which are statements of the form "if ϕ were true, then ψ would be true/possible as well." When ϕ is actually *false* in the present state (*e.g.*, "If Fred were prudent, he would not have gotten into this mess"), the subjunctive query is called a *counterfactual* [62, 43, 52].

The meaning of a subjunctive query is as follows [45]: Update the current database with ϕ; if the resulting database satisfies ψ, then the subjunctive query is true; otherwise, it is false. Following Katsuno and Mendelzon [54], it is widely accepted that "updating" a database, **D**, means finding a database that is closest to **D** according to some metric.[23]

The following are the two major classes of subjunctive queries together with their formulation in $\mathcal{TR}$. To be precise in our formulations, we use the notation of executional entailment.

1. *At state* **D**, *if ϕ were true, then ψ would be true, too*: $P, \mathbf{D} \models \Box(\phi.ins \Rightarrow \psi \otimes \mathtt{path})$
2. *At state* **D**, *if ϕ were true, then ψ would be possible*: $P, \mathbf{D} \models \Box(\phi.ins \Rightarrow \Diamond\psi \otimes \mathtt{path})$

If ψ is a query, then the first subjunctive asks whether the query would be true in the new state, after ϕ is inserted into the database. If ψ is an action, then the second subjunctive asks whether the action would be possible starting from the new state. Note the use of the special proposition `path`, which is true on any path. It permits ψ and $\Diamond\psi$ be evaluated at the updated state itself, not on a path beginning at that state.

Note also the role of the necessity operator "$\Box$" in both cases. This operator ensures that the consequent formula (ψ or $\Diamond\psi$) is true at *every* state obtained from **D** by inserting ϕ. This is important because elementary updates in $\mathcal{TR}$ are not limited to the insertion of a single tuple into a relational database. More generally, an elementary update may insert an arbitrary first-order formula, ϕ, into the database containing other arbitrary first-order formulas. Such updates may be ambiguous and may not result in a unique successor state [33, 32]. Also note that we have used serial implication, "$\Rightarrow$", instead of classical implication, "$\rightarrow$". This requires the consequents to hold *after* the insertion of ϕ. (In contrast, classical implication would require them to hold *during* the insertion.)

8.4.2 Imperative Programming Constructs

Perhaps, one of the most interesting bonuses provided by of the hypothetical operators in $\mathcal{TR}$ is the ability to express the standard imperative constructs, such as **if-then-else** and **while-do** in a simple, declarative way. Consider, for instance, the following clauses:

$$\begin{aligned} if\_\Diamond a\_b\_c &\longleftarrow (\Diamond a) \otimes b \\ if\_\Diamond a\_b\_c &\longleftarrow (\Box\neg a) \otimes c \end{aligned} \tag{43}$$

The transaction $?-\ if\_\Diamond a\_b\_c$, then, has the effect that if a is possible in the current database state, then b will be executed. Otherwise, if $\neg a$ necessarily holds, then c will be executed. The negation "$\neg$" here is of the negation-by-failure variety. In Section 9, we shall discuss the perfect-model semantics for such negation, which is an adaptation from [83].

[23] One metric suitable for updating arbitrary classical theories was proposed by Winslett [97]. Others have also been proposed.

In executing transaction $?\!-\ if\_\diamond a\_b\_c$ above, the subtransaction $?\!-\ a$ was executed hypothetically, *i.e.*, without states being changed. If such an execution is possible, then b is executed "for real" and the system may end up in a different state. In this example, the sentence "$\Box\neg a$" is best interpreted as "otherwise;" that is, "otherwise do c" (here, c is done "for real"). The use of "$\Box$" is crucial to the proper formulation of "otherwise do c." In contrast, the "$\diamond$" in the first clause of (43) could be dropped, leading to another useful form of **if-then-else**:

$$\begin{array}{l} if\_a\_b\_c \longleftarrow a \otimes b \\ if\_a\_b\_c \longleftarrow (\Box\neg a) \otimes c \end{array} \tag{44}$$

The difference between (43) and (44) is that in the latter, the transaction a is done for real (if it is executable), while in the former it is only tested. We call these two conditional statements the "test-version" and the "do-version," respectively.

We shall write **if** $\diamond a$ **then** b **else** c **fi** for $if\_\diamond a\_b\_c$ defined by (43), provided that $if\_\diamond a\_b\_c$ does not occur in the head of any other rule. Likewise, we shall write **if** a **then** b **else** c **fi** for $if\_a\_b\_c$ defined by and (44). Equivalently, these two expressions can be thought of as abbreviations for the formulas $(\diamond a \otimes b) \vee (\Box\neg a \otimes c)$ and $(a \otimes b) \vee (\Box\neg a \otimes c)$, respectively.

In imperative programming, it is often the case that the **else**-part is omitted, which corresponds to "**else** do nothing." In $\mathcal{TR}$, "do nothing" is tantamount to executing `state`. In this case, the "test-version" of **if** $\diamond a$ **then** b **fi** can be expressed as follows:

$$\begin{array}{l} if\_\diamond a\_b \longleftarrow (\diamond a) \otimes b \\ if\_\diamond a\_b \longleftarrow (\Box\neg a) \otimes \mathtt{state} \end{array} \tag{45}$$

The "do-version" of this operator—one in which a is executed for real—is obtained by dropping "$\diamond$" from the above.

Similarly, the **while-do** construct has two forms. The "test-version" is

$$\begin{array}{l} while\_\diamond a\_b \longleftarrow (\diamond a) \otimes b \otimes while\_\diamond a\_b \\ while\_\diamond a\_b \longleftarrow (\Box\neg a) \otimes \mathtt{state} \end{array} \tag{46}$$

and the "do-version" is obtained by removing "$\diamond$":

$$\begin{array}{l} while\_a\_b \longleftarrow a \otimes b \otimes while\_a\_b \\ while\_a\_b \longleftarrow (\Box\neg a) \otimes \mathtt{state} \end{array} \tag{47}$$

In both cases, $while\_\diamond a\_b$ and $while\_a\_b$ are new propositions. Their definitions are recursive, which is what achieves the iterative effect. Notice, again, the role of "$\Box\neg a$" in the second clause of (46) and (47). Here it says that if a cannot be executed, then do nothing, which effectively terminates the loop. As with the **if-then-else** construct, it is suggestive to write

while $\diamond a$ **do** b **od**

for the tail-recursive proposition $while\_\diamond a\_b$ defined by (46), provided that $while\_\diamond a\_b$ does not occur in the head of any other rule. Similarly,

while a **do** b **od**

is a convenient notation for proposition $while\_a\_b$ defined by (47).

Note that in both **while**-versions, if b fails (*i.e.*, cannot be executed) during some iteration, then the entire loop fails, so all previous iterations are undone. This is a form of automatic error recovery. In many cases, however, it may be desirable to not undo previous iterations and to proceed with the loop either by ignoring the failed execution of b or by invoking a designated *error-handling* routine. In $\mathcal{TR}$, this can be expressed as follows:

$$\begin{array}{l} \textbf{while}\ \ a\ \ \textbf{do} \\ \quad \textbf{if}\ \ b\ \ \textbf{then}\ \ \texttt{state}\ \ \textbf{else}\ \ \textit{error-handler}\ \ \textbf{fi} \\ \textbf{od} \end{array}$$

Here, if b fails during any iteration, (*i.e.*, if $\Box\neg b$ is true at the current state), then the error-handling transaction is executed. If no special action is required, we would simply substitute `state` (*i.e.*, "do nothing") for "*error-handler*" in the above.

As an application of our "declarative version" of procedural constructs, we express the transaction "*Raise the salary of all managers by 7%.*" A variation of this transaction was dealt with in Section 7.5 using relational assignment, which can be directly translated into SQL. However, those who are more comfortable with Prolog may prefer to formulate the transaction in the following (slightly stylized) way:

$$?-\ empl(E, Sal, mngr),\ retract(empl(E, Sal, mngr)),\ asserta(empl(E, Sal * 1.07, mngr)),\ fail$$

At first sight, this lean-and-mean Prolog-version seems to score several points over the comparatively bulky SQL-version of Section 7.5. The trouble, however, is that the Prolog formulation defies logic: even though the transaction itself fails, the updates are carried out nonetheless! The standard Prolog practice of using $fail$ to cause looping behaviour is also logically unsound. Furthermore, the fact that the above transaction does the job is due only to the implicit execution control strategy that relies on backtracking. If not for this strategy, each manager's salary could have been raised many times over.

Our goal, then, is to avoid repeating the salary-raising action for one and the same person, and to do so in a logically sound manner. Our strategy is to move all managers to an auxiliary relation, raising their salaries in the process, and then move the managers back to the original relation. In $\mathcal{TR}$ this can be accomplished with two while-do loops, as follows:

$$\begin{array}{ll} ?- & \textbf{while}\ \ empl(E, Sal, mngr)\ \ \textbf{do} \\ & \qquad\quad empl.del(E, Sal, mngr) \otimes manager.ins(E, Sal * 1.07) \\ & \textbf{od} \\ & \otimes \\ & \textbf{while}\ \ manager(E, Sal)\ \ \textbf{do} \\ & \qquad\quad manager.del(E, Sal) \otimes empl.ins(E, Sal, mngr) \\ & \textbf{od} \end{array}$$

In the first loop, each iteration retrieves a manager from the *empl* relation, deletes him from the *empl* relation, and inserts him into the *managers* relation with a new salary. Iteration continues until there are no managers left in the *empl* relation. In the second loop, each iteration retrieves a record from the *manager* relation, deletes it from the *manager* relation, and inserts it into the *empl* relation. Iteration continues until there are no records left in the *manager* relation. The result is that each manager has gets a raise of 7%.

Even though this $\mathcal{TR}$-version is longer than the Prolog-version, it is completely transparent and its semantics do not depend on the peculiarities of a particular control strategy. For instance, it does not rely on the subtle use of *asserta* and *fail*, the way the Prolog-version does. We therefore claim that the $\mathcal{TR}$-version is more natural. In fact, to some, the $\mathcal{TR}$-version may even seem more straightforward than the SQL-version of Section 7.5.

8.4.3 Software Verification

Another promising application for $\mathcal{TR}$ is software verification. Software specification systems, such as VDM [53], are essentially specification languages based on the usual predicate calculus or on one of its multi-valued extensions. The general problem with these languages is that these specifications are not executable. That is, these specifications are intended to explain what the program is supposed to do, but since the program is usually written in a different language (say, C or ADA), the software engineer can do no better than hope that the programmer will adhere to the given VDM (or similar) specifications.

Although much research needs to be done to make the $\mathcal{TR}$ proof theory efficient, it is clear that—at least in theory—$\mathcal{TR}$ is a medium in which programs can be specified, verified, *and* executed. Indeed, suppose that a program, ψ, defined in $\mathcal{TR}$ is supposed to satisfy a post-condition, *post*, provided that the input satisfies a pre-condition, *pre*. To determine whether the rules for ψ comply with this specification, one can pose the following hypothetical query:

$$?-\ \Box(\texttt{arc} \otimes pre \otimes \psi \Rightarrow post \otimes \texttt{path})$$

If the query succeeds, then the programmer can rest assured that the transaction works as intended. Note the role of the special proposition `arc`, which is true on (and only) every arc. It changes the database from its current state to an arbitrary state. This ensures that ψ behaves correctly when executed from *any* state, not just from the current state. If ψ needs to be verified for the current state only, then `arc` can be omitted. Also, as in Section 8.4.1, the special proposition `path` permits *post* to be evaluated at the new state resulting from ψ, instead of on a path beginning at that state.

9 Perfect-Model Semantics for Negation as Failure

The theory presented so far is classical in that logical entailment is defined in terms of *all* models of a set of formulas. However, in Logic Programming and in AI, it is common to treat the negation operator as negation-as-failure rather than in the classical sense. For instance, in Section 7.7.2 we considered constraints, such as (25), that are based on serial implication. Serial implication (like classical implication) involves negation, and to ensure that (25) acts as a constraint, this negation must be interpreted as failure. Likewise, in Section 8.4.2, we remarked that negation in **if-then-else** and **while-do** statements must be interpreted as failure.

One widely accepted formalization of negation-as-failure is based on the so-called perfect-model semantics [83]. In this section, we adapt this semantics to $\mathcal{TR}$. We use perfect-model semantics because it is two-valued, and is therefore easy to transplant to $\mathcal{TR}$. In addition, perfect-model semantics lies in the intersection of all major formalizations of negation as failure (for locally stratified programs) and thus is more universally agreed upon.

The first step is to delineate the class of rules we are prepared to deal with. These rules generalize the serial Horn rules of Section 6 and the $\Diamond$-serial-Horn rules of Section 8.1, and accommodate all applications discussed in this paper.

Definition 9.1 (Generalized-Horn Rules) A *generalized-Horn rule* is a formula of the form $p \leftarrow \phi$, where p is atomic and ϕ is a generalized goal. *Generalized goals* are defined recursively as follows:

- If p is an atom, then p and $\neg p$ are generalized goals.
- If ψ is a generalized goal, then so are $\Diamond\psi$ and $\psi\Diamond$.
- If $\psi_1, \ldots, \psi_n$ are generalized goals, then so are $\psi_1 \otimes \cdots \otimes \psi_n$ and $\psi_1 \wedge \cdots \wedge \psi_n$.

A *generalized-Horn transaction base* is a finite set of generalized-Horn rules. □

As in classical Horn logic programming, the semantics of negation-as-failure can be problematic when recursion occurs through negation. For instance, since the informal meaning of $\neg p$ is that "p cannot be proved," what is the meaning of the rule $p \leftarrow \neg p$? To avoid such problems, we restrict our attention to locally-stratified transaction bases, which is an adaptation from [83].

To define local stratification, let $\mathbf{P}$ be a transaction base. $\mathbf{P}^*$ then denotes the ground instantiation of $\mathbf{P}$, *i.e.*, the set of all ground instances of rules in $\mathbf{P}$. Following [83], we construct a directed graph, $\mathcal{D}(\mathbf{P})$, whose nodes are atomic formulas in $\mathbf{P}^*$. The arcs in the graph are defined as follows:

- An *arc* goes from p to q if and only if p occurs in the body of a rule in $\mathbf{P}^*$ that has q in the head.
- An arc that goes from p to q is *negative* if and only if $\neg p$ occurs in the body of a rule in $\mathbf{P}^*$ that has q in the head.

In either case, the literal p or $\neg p$ may appear inside a hypothetical operator. Thus, the following rules each define an arc from p to q that is *not* negative:

$$q \leftarrow p \qquad q \leftarrow \Diamond p \qquad q \leftarrow r \otimes \Diamond(t \wedge \Diamond p)$$

Likewise, the following rules each define an arc from p to q that *is* native:

$$q \leftarrow \neg p \qquad q \leftarrow \Diamond\neg p \qquad q \leftarrow r \otimes \Diamond(t \wedge \Diamond\neg p)$$

Using this graph, we define two partial orders, $\succeq$ and $\succ$. We write $q \succeq p$ if there is a directed path from p to q, and we write $q \succ p$ if there is a *negative* path from p to q. A path is negative if it has at least one negative arc on it. Finally, a transaction base $\mathbf{P}$ is *locally stratified* if and only if the relation "$\succ$" is well-founded, *i.e.*, it has no infinite sequence of atoms such that $p_1 \succ p_2 \succ \cdots$.

In classical logic programming, the perfect model semantics is defined in terms of a *preference* relation, which allows us to choose between alternative models. Choice arises because negation in a rule body is classically equivalent to disjunction in the rule head. The preference relation in [83] allows us to choose (or prefer) one disjunct over another. We extend this idea to $\mathcal{TR}$.

To motivate the development, first consider a simple example. Suppose that path π can be split into three parts, π_1, π_2 and π_3, so that $\pi = \pi_1 \circ \pi_2 \circ \pi_3$. Consider models of the rule $a \leftarrow p \otimes \neg b \otimes q$ such that p is true only on path π_1 and q is true only on path π_3. In such models, we have a choice: either a is true on path π or b is true on path π_2. In this case, we prefer the former models, *i.e.*, models in which a is true on π and b is false on π_2. Intuitively, this preference minimizes the number of paths on which b is true, at the expense of a.

The following definition formalizes this idea. It follows the outline of [83], except that we do not limit ourselves to Herbrand models.

Definition 9.2 (Perfect Models) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{path}\rangle$ and $\mathbf{M}' = \langle U, I_{\mathcal{F}}, I'_{path}\rangle$ be two path-models of transaction base $\mathbf{P}$. Note that they have the same domain and the same interpretation of function symbols. We say that $\mathbf{M}$ is *preferable* to $\mathbf{M}'$, written $\mathbf{M} \ll \mathbf{M}'$, if and only if for any atom a and any path π,

$$\text{if } \mathbf{M}, \pi \models a \text{ and } \mathbf{M}', \pi \not\models a \text{ then } \mathbf{M}, \pi' \not\models b \text{ and } \mathbf{M}', \pi' \models b$$

for some path π' and some atom b such that $a \succ b$. A *perfect* Herbrand model of $\mathbf{P}$ is any model that is minimal with respect to $\ll$. □

As in [83], it can be verified that every locally-stratified transaction base has a perfect model. An important special case arises when the database is relational[24] and when we are interested exclusively in the *Herbrand domain.*[25] In this case, our perfect-model semantics reduces to a straightforward extension of that in [83]. In particular, every generalized-Horn transaction base has a *unique* perfect Herbrand model.

Note that the preference relation in Definition 9.2 compares path-models on two different paths, π and π'. The motivating example above shows why. In a rule like $a \leftarrow p \otimes \neg b \otimes q$, the atoms a and b are *not* generally evaluated on the same paths. In fact, b is evaluated on a *subpath* of whatever path a is evaluated on. Nevertheless, in comparing models, we would expect to compare the truths of a and b, which means comparing them on different paths.

We can now extend the notion of executional entailment to include negation-as-failure. That is, we can define the meaning of statements like

$$\mathbf{P}, \mathbf{D}_1, \ldots, \mathbf{D}_n \models_{\mathsf{perf}} \phi$$

where ϕ is a generalized goal and $\mathbf{P}$ is a generalized Horn transaction base. The new definition is identical to Definition 5.6 except that instead of looking at *all* models of $\mathbf{P}$, we look only at their *perfect* models. In the special case where $\mathbf{P}$ and ϕ contain no negation signs, the two definitions are equivalent.

Perfect-Model Semantics in the Applications

Sections 7.7.2 and 8.4.2 presented two important applications of the perfect model semantics: constraining transaction execution, and expressing imperative programming constructs. However, syntactically, these applications are not covered by Definition 9.1 of generalized-Horn rules. We show, below, that both of these cases can be represented as generalized-Horn rules.

To constrain transactions, Section 7.7.2 used formulas with serial implication, which represents an implicit use of negation. For example, if p is an atom representing a series of robot shooting actions, then the following formula constrains the robot to unlock the gun before shooting it:

$$p \wedge (\mathtt{path} \otimes \mathit{unlock} \Leftarrow \mathit{shoot} \otimes \mathtt{path}) \tag{48}$$

As described in Section 7.7.2, this formula is evaluated in the perfect models of the transaction base.

[24] *I.e.*, when the data oracle always returns a set of ground atomic formulas.

[25] *I.e.*, when U is the set of all variable-free first-order terms, and $I_{\mathcal{F}}(f)(t_1, \ldots, t_k) = f(t_1, \ldots, t_k)$, for any k-ary function symbol f and any term t_i.

Formulas like (48) can also appear in rule premises. However, they must first be transformed into generalized goals, *i.e.*, into the syntactic form required by Definition 9.1. Fortunately, this is easily accomplished using the equivalences presented in Section 5.2. For example,

$$unlock \Leftarrow shoot \quad \equiv \quad unlock \oplus \neg shoot \quad \equiv \quad \neg(\neg unlock \otimes shoot)$$

These equivalences say that the constraint $unlock \Leftarrow shoot$ is equivalent to the negation of a generalized goal. We can therefore represent the constraint as an atomic formula, *unlock-before-shooting*, defined by the following generalized-Horn rules:

$$\begin{array}{rcl} unlock\text{-}before\text{-}shooting & \longleftarrow & \neg do\text{-}not\text{-}unlock\text{-}then\text{-}shoot \\ do\text{-}not\text{-}unlock\text{-}then\text{-}shoot & \longleftarrow & \neg unlock \otimes shoot \end{array}$$

Formula (48) can now be expressed as a generalized goal:

$$p \wedge (\mathtt{path} \otimes unlock\text{-}before\text{-}shooting \otimes \mathtt{path})$$

In this way, constrained transactions can appear in the premises of generalized-Horn rules. These rules are interpreted by our perfect model semantics.

To express imperative programming constructs, Section 8.4.2 needed a form of negation that is very similar to the classical idea of negation-as-failure, in which $\neg\psi$ succeeds if and only if ψ fails. As discussed in Section 5.7, however, transactions do not behave this way, even with a perfect-model semantics. The difficulty is that a transaction, ψ, and its negation, $\neg\psi$, can *both* succeed from the same database, since ψ may execute along one path while $\neg\psi$ executes along another path. To overcome this difficulty, Section 8.4.2 used the necessity operator, $\Box$, since for every database, exactly one of ψ and $\Box\neg\psi$ will succeed. $\Box\neg\psi$ thus corresponds closely to the idea of classical negation-as-failure.

To interpret the formula $\Box\neg\psi$ using our perfect-model semantics, we must reduce it to a generalized goal. This is possible because of a near-dual relationship between necessity and possibility. In particular, as discussed in Section 8.1,

$$\Box\neg b \;\equiv\; \mathtt{state} \wedge \neg\Diamond b$$

where `state` is the special proposition that is true on paths of length 1, *i.e.*, on states. When b is atomic (which is the important case for imperative programming), the right-hand side of the equivalence can be reduced to a generalized goal, $\mathtt{state} \wedge \neg c$, where c is defined by the following generalized-Horn rule: $c \leftarrow \Diamond b$. In this way, imperative programming constructs can be expressed in $\mathcal{TR}$ using our perfect-model semantics.

10 Updating General Deductive Databases

As mentioned previously, the database part of a $\mathcal{TR}$ logic program may contain Prolog-like rules. It is natural, therefore, to let transactions manipulate database rules in the same way as database facts. This capability has been found useful, for example, in AI programming in Prolog. Unfortunately, rule updates in Prolog are procedural and do not have a logical or declarative semantics. Since $\mathcal{TR}$ allows a database to contain deductive rules, it provides a formal semantics for transactions that do rule updates. The semantics of each rule insertion and deletion is specified by the transition oracle, which may reflect any number of semantics that have been studied in the literature (*e.g.*, [45, 97]). This approach works well and is completely general as long as a database is a collection of first-order formulas. However, many logic programs and deductive databases are not purely first-order, since they interpret negation as failure. This section addresses this point. First, we elaborate on the first-order case; then, we explain the problem that exists with non-first-order databases, after which we show how $\mathcal{TR}$'s model theory handles this case.

In Prolog, one writes $assert(\ a(Y)\ :-\ b(X), c(X,Y)\ )$ to insert the rule $a(Y)\ :-\ b(X), c(X,Y)$ into the database. Here, the argument of the *assert* command is treated as a first-order *term* that represents an encoding of the corresponding *rule*. This kind of "reflection" of logical formulas into ground terms is well-known in logic, and we shall not dwell on it here. We simply assume that some encoding schema is given, allowing us to reflect any database rule into a first-order ground term.

Once the encoding mechanism is in place, we let the transition oracle, $\mathcal{O}^t$, specify elementary state transitions of the following form:

$$(\ a(Y) \longleftarrow b(X) \wedge c(X,Y)\ ).ins \in \mathcal{O}^t(\mathbf{D}, \mathbf{D}')$$

where the database states $\mathbf{D}$ and $\mathbf{D}'$ are defined appropriately. For databases containing arbitrary first-order formulas, the relationship between $\mathbf{D}$ and $\mathbf{D}'$ can be complex, and has been partly worked out by Grahne, Mendelzon, and Winslett [45, 97]. A simple case is rule insertion into Horn databases. Here, we would add the following entry to the transition oracle

$$r'.ins \in \mathcal{O}^t(\mathbf{D}, \mathbf{D} \cup \{r\})$$

for every Horn database $\mathbf{D}$ and every Horn rule r, where r' is the reflection of r.

This approach to updates works for classical first-order databases. However, logic programs that use negation-as-failure are not purely first-order, as they rely on canonic model semantics and related techniques. The basic problem is illustrated in the next example.

Example 10.1 (Updating Rule-Bases: a problem) Consider the following two databases:

$$\mathbf{D}_1 \ = \ \{\, a \longleftarrow \neg b \,\} \qquad\qquad \mathbf{D}_2 \ = \ \{\, b \longleftarrow \neg a \,\}$$

From the viewpoint of classical logic, $\mathbf{D}_1$ and $\mathbf{D}_2$ are both equivalent to $a \vee b$. But not so in logic programming, where these two databases mean quite different things. Now, suppose we are to define the elementary transition $(\ a \longleftarrow \neg b\ ).ins$. Presumably, it should take us from an empty database to $\mathbf{D}_1$, but *not* to $\mathbf{D}_2$. Thus, the semantics must distinguish $\mathbf{D}_1$ from $\mathbf{D}_2$, even though they are classically equivalent. □

In logic programming and deductive databases, the problem of discriminating between $\mathbf{D}_1$ and $\mathbf{D}_2$ is solved by associating a set of *canonic* models to each database. This set is usually much smaller than the set of all models and often there is only one canonic model per database. One way to define canonic models is to use the perfect-model semantics of [83] or, more generally, the well-founded semantics of [96]. In this section, however, we need not commit ourselves to any specific definition of canonic models. In Example 10.1, the canonic (perfect) model of $\mathbf{D}_1$ makes the proposition a true and b false; whereas the canonic (perfect) model of $\mathbf{D}_2$ does the opposite, making b true and a false.

It turns out, however, that $\mathcal{TR}$ does not need new machinery to handle general database states. When one switches from, say, relational databases (that use classical inference) to, say, deductive databases, all that is needed is that we switch from the Relational Oracle to an appropriate non-first-order oracle, like the Well-founded Oracle or the Stable Oracle (Section 5, which will readily account for the particular underlying semantics of database states.

11 Comparison with Other Work

As mentioned earlier, there would appear to be many other candidates for a logic of transactions, since many logics reason about state changes or about the related phenomena of time and action. However, despite the abundance of action logics, none has ever been found suitable for a database or logic programming system, and none has become a core of database or logic-programming theory. This is in stark contrast to classical logic, which is the foundation of database queries and logic programming, both in theory and in practice. The introduction described a few broad reasons for this unsuitability of existing action logics. This section looks at specific formalisms in more detail. We first discuss declarative languages that were designed for updating databases. We then look closely at action logics presented in the literature of Logic and AI.

As far as databases and logic programming are concerned, few logics can match the broad range of applications of $\mathcal{TR}$, especially in comparison to the small number of new features added to first-order classical logic to achieve these capabilities. Apart from the suite of applications discussed here, minor extensions to $\mathcal{TR}$ enable it to seamlessly integrate hypothetical transactions, counterfactual reasoning, and even concurrent and communicating transactions.

One feature that sharply distinguishes $\mathcal{TR}$ from all other formalisms are the data and transition oracles. The transition oracle exploits $\mathcal{TR}$'s distinction between stored and derived data. Since the oracle is a parameter to the logic, many kinds of elementary update can be performed by specialized, efficient algorithms that are commonly used in database systems. Furthermore, many algorithmic features of database systems, such as bulk updates and random sampling, can be relegated to the transition oracle without sacrificing declarativeness (see Sections 7.5 and 7.6). The same is true of heuristic approximations to intractable problems. Indeed, localizing the *ad hoc* nature of such heuristics to the transition oracle makes them invisible to the proof theory, the model theory, and the run-time environment of the system.

In a similar fashion, the data oracle separates the dynamics of the logic from details pertaining to the semantics of states. This has the additional benefit that we can push certain non-monotonic aspects of various problems down to the data oracle. Queries to the data oracle do not even have to be recursively enumerable, since states are treated as black boxes by the proof theory (which is complete modulo the oracles). One example is testing the emptiness of a predicate in a logic program. In logic programming and deductive databases, such problems are typically dealt with by developing non-monotonic, second-order semantics, and by imposing syntactic restrictions on programs (*e.g.*, [83, 96, 39]). $\mathcal{TR}$'s data oracle factors out such problems from the dynamics of the logic, so they can be treated separately. In this way, $\mathcal{TR}$ is able to provide a monotonic, first-order semantics for incorporating complex (even non-monotonic) queries into complex transactions.

In other words, even though elementary database operations may be non-monotonic or non-logical, their *combination* into complex programs is often both logical and monotonic. $\mathcal{TR}$ allows the user to factor out any nastiness in elementary operations, and provides a simple logic for combining them. The examples in this paper demonstrate the utility of this approach in a wide range of applications.

11.1 Declarative Languages for Database Updates

Winslett [97] did foundational work by supplying the first generally acceptable semantic definition for the result of updating a logical theory. Later on, Grahne, Katsuno and Mendelzon [54, 44, 45] axiomatized various theories of state transition and studied tractable cases of what we call "elementary state transitions." Our approach to state transitions is inspired by these results.

While [97, 54] take a model-theoretic view of elementary updates, Fagin, Kuper, Ullman, and Vardi [33, 32] approach the problem syntactically. One interesting consequence of this is that, while in [97, 54] an elementary update is always deterministic, in [33, 32] updating a theory with a formula may lead to several alternative states.

Several authors [24, 69, 78] base their approaches to updates on Dynamic Logic [49]. Casanova [24] applies an adaptation of Dynamic Logic to reasoning about concurrent execution of procedural programs that are built out of relational algebra operators plus an explicit relational assignment operator. Updates are done in the hypothetical mode and no complete proof theory is given.

Manchanda and Warren [69] introduce Dynamic Prolog—a logic system where update transactions "work right," *i.e.*, they do not leave a residue in the database when a transaction fails (the residue left by *assert* and *retract* is the most serious problem that plagues Prolog's update mechanism). Like $\mathcal{TR}$, their logic can be used to update views, and transactions can be nondeterministic. However, they distinguish between update predicates and query predicates (which is a drawback if we keep an eye on object-oriented applications, as explained in the introduction). Furthermore, bulk updates, constraints on transaction execution, and the insertion and deletion of rules cannot be expressed, due to the chosen semantics. In addition, the proof theory for Dynamic Prolog is impractical for carrying out updates, since one must know the final database state *before* inference begins. This is because this proof theory is a verification system: Given an initial state, a final state, and an update procedure, the proof theory can *verify* that the procedure causes the transition; but, given just the initial state and the procedure, it cannot *compute* the final state. In other words, it cannot execute procedures as $\mathcal{TR}$'s proof theory does. Apparently, realizing this drawback, Manchanda and Warren developed an interpreter whose aim was to "execute" transactions. However, this interpreter is incomplete with respect to the model theory and, furthermore, it is not based on the proof theory. To a certain extent, it can be said that Manchanda and Warren have managed to formalize their intuition as a program, but not as an inference system.

Naqvi and Krishnamurthy [77] extended Datalog with update operators, which were later incorporated in the LDL language [78]. Since LDL is geared towards database applications, this extension has bulk updates, for which an operational semantics exists. Unfortunately, the model theory presented in [77, 78] is somewhat limited. First, it matches the execution model of LDL only in the propositional case, and so it does not cover bulk updates. Second, it is only defined for update-programs in which commutativity of elementary updates can be assumed. For sequences of updates in which this does not hold, the semantics turns out to be rather tricky and certainly does not qualify as "model theoretic." Third, the definition of "legal" programs in [77, 78] is highly restrictive, making it difficult to build complex transactions out of simpler ones.

Chen has developed a calculus and an equivalent algebra for constructing transactions [28]. Like $\mathcal{TR}$, this calculus uses logical operators to construct actions from elementary updates. There are several differences however. First, unlike $\mathcal{TR}$, Chen's calculus is not a full-blown logic. It extends the domain relational calculus to include updates, but there is no notion of logical inference. Second, the calculus operates only on relational databases, and not on arbitrary logical theories. Third, the calculus has no analogue of $\mathcal{TR}$'s transition oracle for elementary updates, so it is restricted to the insertion and deletion of single tuples. However, the most profound difference between the two formalisms shows in the semantics of conjunction. Whereas $\mathcal{TR}$ uses $\wedge$ to express constraints, Chen's calculus uses it to express parallel actions. The main motivation here is that parallel actions make bulk updates easy to express, which is an important database feature. However, there are several disadvantages in the way this is achieved. First, the calculus cannot express the kind of sophisticated constraints that $\mathcal{TR}$ can. At the same time, sequential updates often achieve the same effect as parallel updates. Second, parallel actions greatly complicate the semantics, since they require a minimality principle, which makes the algebra non-monotonic even in the absence of negation. This complication is not called for, since, for instance, $\mathcal{TR}$ achieves the same effect in a much simpler, monotonic way, as described in Section 7.5. Third, the syntax is not closed. For instance, negation can be applied to some formulas but not others. In particular, if ψ is an updating transaction, then rules like $p \leftarrow \psi$ are not allowed, since it is equivalent to $p \vee \neg\psi$; indeed, this formula has no meaning in Chen's calculus. It therefore seems unlikely that this calculus can be developed into a full-blown *logic* in a straightforward or satisfying way. Furthermore, the calculus itself is very limited as a programming language, since it has no mechanism for defining recursion or subroutines.

Bry [23] takes a two-tiered approach: first, he uses ordinary queries to determine (generate) the set of atoms to be added/deleted; then, he performs updates meta-logically. Dynamic constraints are also meta-logical; and they involve pairs of different theories. Thus [23] can neither define nor compose complex transactions. It is also impossible to specify constraints on transaction execution or to insert and delete rules. In a sense, [23] provides a *methodology* for expressing what is to be updated rather than a logic for *performing* updates.

Bonner has carried out a thorough analysis of reasoning in Hypothetical Datalog [16, 18, 17, 15, 14], including a sophisticated complexity analysis. Like $\mathcal{TR}$, this work combines elementary updates into complex transactions. Both works also use similar tactics to avoid the frame problem (Section 7.8.). However, Hypothetical Datalog does not commit updates, does not deal with arbitrary logical databases, and does not combine updates with other logical connectives in arbitrary ways. Instead, Hypothetical Datalog is a disjunction-free logic programming system for relational databases. The semantics of Hypothetical Datalog is also based on different principles than $\mathcal{TR}$'s. In particular, it does not embrace the concept of the execution path, which is fundamental to $\mathcal{TR}$. Consequently, dynamic constraints of the kind discussed in Section 7.7 cannot be expressed.

Abiteboul and Vianu developed a family of declarative update languages [5, 4, 1], including impressive results on complexity and expressibility. However, these languages lack several features that are present in $\mathcal{TR}$. First, they apply only to relational databases, not to arbitrary sets of first-order formulas. Thus, it is not possible to insert or delete rules from a deductive database. Second, there is no facility for constraining transaction execution. Indeed, transaction output is the only concern. Third, these languages are not part of a full-blown *logic*: arbitrary logical formulas cannot be constructed, and although there is an operational semantics, there is no model theory and no logical inference system. It is therefore unlikely that these languages have the flexibility to find applications in other domains, such as AI. Finally, these languages do not support transaction subroutines, a facility that any practical programming language must provide. That is, a transaction cannot be given a name, and then be invoked repeatedly from within the language. This lack of subroutines is reflected in the data complexity of some of the languages: they are in PSPACE, whereas recursive subroutines require *alternating* PSPACE, that is, EXPTIME.

The works [80, 30] are related to [5] in that they all borrow much of their syntax from deductive databases and yet their semantics is operational (although inspired by logical model theory). As such, these languages are in a different league than $\mathcal{TR}$; they are also unsuitable for defining transactions, constraints, and for reasoning about actions.

11.2 Logics of Action

The work described above is aimed at specifying and *executing* database updates. In contrast, logics of action are aimed at specifying and *reasoning* about actions. Updates are a special case of actions and, in a sense, execution is a special case of reasoning where one infers the final state of the database from the initial state. With this in mind, we examine a number of prominent action logics from the perspective of databases and logic programming.

Generally, these action logics have far greater expressive power than the database languages discussed above. In a sense, however, they are *too* expressive. One lesson of database theory and practice is that if a declarative language (like SQL) is not too expressive, then optimization and high performance are possible. Often, database languages aim for the *minimal* power needed to get a job done. Indeed, most of the languages discussed above have natural implementations that are reasonably efficient and optimizable. In contrast, most of the logics in this section provide no clues about how to efficiently execute transactions.

Although logics of action can have great expressive power, it is sometimes the wrong kind of expressiveness for databases. For instance, since most action logics were not designed as programming languages, important programming features (such as subroutines) are often awkward, if not impossible to express. In addition, action logics often make a sharp distinction between queries and updates. This distinction may be important in some applications, but it is ill-suited for programming. For example, Prolog-style languages and most object-oriented languages treat queries and updates uniformly. A sharp distinction can also be a hindrance to application development. For instance, to debug a query, a programmer may want to insert an update into its definition to monitor its execution or to count the number of times it is called. More generally, as an application evolves, simple queries may develop into more complex transactions with updates, and vice-versa. An abrupt distinction between queries and updates makes this kind of evolution difficult, if not impossible.

$\mathcal{TR}$ makes no such distinction. It treats queries and updates uniformly, as two extreme points on a spectrum of transactions. Like ordinary programs, every formula in $\mathcal{TR}$ can both update the database *and* return an answer. While $\mathcal{TR}$ does not distinguish between queries and updates, it does not force the user to blur the line between the two. If an application calls for such a distinction, it can easily be achieved by using two sorts of predicates, query predicates and update predicates. One way to distinguish the two is to require that all update predicates be prefixed with *do:*. Thus, *do:enroll(student course)* would be an update, while *grade(student,course)* would be a query. In a similar fashion, deductive databases distinguish two kinds of predicates (base and derived), while the underlying (classical) logic does not.

Finally, most logics of action do not actually perform database updates. Many, in fact, do not even have a notion of database, since they were not intended for database applications. Typically, an action logic describes relationships between different states, without describing how to get from one state to the next. This is fine for reasoning about action; but to actually execute an action, one needs much more information. Unfortunately, large numbers of logical formulas are often needed to completely describe even very simple actions. Even when this is not a problem, however, an action logic is usually just a specification language. Actually executing an action and updating the database must be done outside of the logic. This contrasts with classical logic programming, where executing a query can be carried out by purely logical means, *i.e.*, by the proof theory. Unlike most logics of action, $\mathcal{TR}$ is a language for *both* specifying and executing actions. This is possible because $\mathcal{TR}$ includes logical mechanisms for executing an action and updating the database. This is formalized model-theoretically by the notion of *executional entailment* (Section 5.4). It is even more apparent in the proof theory, in which each elementary update produces a new database state.

Process Logic

As mentioned previously, our path structures are reminiscent of the "path models" in Process Logic [50].[26] However, the meaning and intent of dynamic formulas in [50] is fundamentally different from that of transaction formulas in $\mathcal{TR}$. The former is intended to *reason about* what is true during program execution, while $\mathcal{TR}$ was designed to actually *execute* declaratively specified procedures. This difference in the intent is reflected in the syntax. For instance, Process Logic uses a separate alphabet to represent actions, and a set of modal operators to reason about them. Thus, unlike $\mathcal{TR}$, actions in [50] have a different status than propositions. In particular, actions are not logical formulas, but rather are terms used to construct modal operators. One consequence is that only elementary actions have names. Composite actions cannot be named, and thus the logic lacks a subroutine facility. Another difference between $\mathcal{TR}$ and Process Logic is in the nature of states. As discussed in Section 5, our concept of a state immediately leads to the idea of a transition oracle—a notion thoroughly missing from all incarnations of the process logic. Finally, although actions can be sequenced in Process Logic, this is not done using a *logical* connective of serial conjunction, as in $\mathcal{TR}$. Consequently, there is no counterpart to $\mathcal{TR}$'s dual connective of serial disjunction, nor to serial implication. Thus, there is no obvious way to express constraints based on these connectives, like those illustrated in Section 7.7.2.

Action Logic and Related Proposals

Pratt [82] develops a logic, called *Action Logic*, that has a few similarities with $\mathcal{TR}$ but is different from it in many essential ways. Like $\mathcal{TR}$, Action Logic does not distinguish between actions and propositions: actions are simply propositions that hold on intervals. The semantics and the intent of the two formalisms are very different however. First, Action Logic is not a language for updating databases or defining transactions. In fact, it has no notion of database, no analogue of $\mathcal{TR}$'s transition oracle, and no counterpart to executional entailment. Instead, Action Logic is an extension of regular expressions, and as such, it can be viewed as a formalism for defining languages. Second, in contrast to $\mathcal{TR}$'s semantics, which is based on sequences of states, Pratt develops an algebraic semantics, based on *action algebras*. The proof theory for Action Logic is a (finite) set of equations for reasoning about these algebras.

Nevertheless, Action Logic is superficially similar to $\mathcal{TR}$. For instance, it has operators similar to $\otimes$, $\vee$, `state`, and $\neg$`path` in $\mathcal{TR}$; and it has an iteration operator, "*", which can be expressed in $\mathcal{TR}$ using recursion. What makes the comparison seem especially close, is a pair of connectives, denoted $\leftarrow$ and $\rightarrow$ in Pratt's notation, that look very similar to $\mathcal{TR}$'s connectives for serial implication, $\Leftarrow$ and $\Rightarrow$. Semantically, however, these connectives are very different. For instance, the following equation is an axiom of Action Logic:

$$a(a \rightarrow b) \ + \ b \ = \ b \tag{49}$$

Here, $+$ denotes ordinary disjunction, and concatenation denotes sequential composition. This equation is therefore a sequential version of modus ponens: it says that if $a(a \rightarrow b)$ is true, then b is also

[26] A number of different process logics have been proposed in the literature, beginning with Pratt's original work [81]. The version in [50] is closer to $\mathcal{TR}$ than any other incarnation of Process Logic we are aware of.

true. The same is not true in $\mathcal{TR}$. The analogue of equation (49) in $\mathcal{TR}$ is the following statement:

$$a \otimes (a \Rightarrow b) \ \vee \ b \ \equiv \ b$$

However, this statement is *not* a theorem of $\mathcal{TR}$. The intuitive reason is that the two occurrences of a in this statement do not refer to the same time interval. In fact, the second occurrence of a can begin only when the first occurrence ends. Hence, the truth of $a \otimes (a \Rightarrow b)$ does not imply the truth of b.

There is one more difference worth noting. $\mathcal{TR}$ obtains much of its simplicity by having *two* kinds of conjunction, classical and serial. Combined with negation, they lead to two kinds of disjunction, and two kinds of implication, in a natural way. In contrast, Action Logic obtains much of its simplicity by having *one* type of conjunction (serial conjunction), as pointed out in [82]. Action algebras have a lattice-like "meet" operator that might form the model-theoretic basis for another kind of conjunction, but unfortunately, the meet operation is not always defined.

In [95], van Benthem outlines a number of logical approaches to dynamic information processing. These approaches and $\mathcal{TR}$ share the idea that actions are to be represented as propositions, but the similarities end here. The proposed logics are not path-based and there is no notion analogous to the transition oracle of $\mathcal{TR}$.[27]

One of the approaches in [95] is based on *dynamic interpretation* of classical predicate logic. A dynamic interpretation associates a pair of states to each proposition, which resembles Dynamic Prolog of Manchanda and Warren [69]. However, the states associated to propositions by dynamic interpretations of [95] are not database states but rather variable assignments. van Benthem also discusses an algebraic approach to the logic of dynamic systems, which is akin to the action logic of Pratt [82].

McCarty's Logic of Action

McCarty has outlined a logic of action as part of a larger proposal for reasoning about deontic concepts [72]. His proposal contains three distinct layers, each with its own logic: first-order predicate logic, a logic of action, and a logic of permission and obligation. In some ways, the first two layers are similar to $\mathcal{TR}$, especially since the action layer uses logical operators to construct complex actions from elementary actions. Because of his interest in deontic concepts, McCarty defines two notions of satisfaction. In one notion, called "strict satisfaction," conjunction, $\wedge$, corresponds to parallel action, as it does in Chen's work [28]. In the other notion, called "satisfaction," the same symbol corresponds to constraints, as it does in $\mathcal{TR}$. However, since the focus of this work is on strict satisfaction, the development of path constraints was never considered. Also, there is no analogue of $\mathcal{TR}$'s transition oracle, and the only elementary updates considered correspond to insertion and deletion of atomic formulas. Although obviously interesting, this action logic was not developed in detail. For instance, although a model theory based on sequences of partial states is presented, there is no sound-and-complete proof theory, and no mechanism is presented for executing actions or updating the database.

In contrast, the recent work of McCarty and Van der Meyden [73] is much more detailed, but is orthogonal to $\mathcal{TR}$. The main similarity is that both works are concerned with defining complex actions in terms of simpler ones. However, unlike $\mathcal{TR}$, [73] does not address action execution or the updating of databases. To give an idea of what [73] is about, consider a $\mathcal{TR}$ transaction base

[27] van Benthem actually mentions that a path-based logic would be desirable, but no such logic was developed in [95].

consisting of exactly the following two rules:[28]

$$a \leftarrow c1 \otimes c2 \otimes c3 \qquad\qquad b \leftarrow c2 \otimes c3$$

The main point is that a and b are complex actions defined in terms of the elementary actions $c1, c2, c3$. In $\mathcal{TR}$, the effects of the elementary actions are specified by an oracle, which is invoked to execute them. In contrast, [73] has no mechanism for specifying the effects of elementary actions. Instead, their work focusses on closed-world inferences of the following form:

> If we are told that action a has occurred, then we infer (abductively) that action $c1 \otimes c2 \otimes c3$ has occurred, so action $c2 \otimes c3$ has occurred, so action b has occurred. Thus, an occurrence of action a implies an occurrence of action b.

There are also technical differences between $\mathcal{TR}$ and [73]. For instance, [73] does not allow function symbols in the rules that define complex actions (and even so, most of their reasoning problems are undecidable and/or outside of *re*). In addition, [73] is committed to a particular model of states. The basic theory in [73] is also very different from that of $\mathcal{TR}$, as it is based on circumscription and second-order intuitionistic logic.

Action Language of Gelfond and Lifschitz

Gelfond and Lifschitz have proposed a high-level language called $\mathcal{A}$ for describing the effects of actions [40]. $\mathcal{A}$ has rules for specifying the effects of sequences of actions, causal laws, initial knowledge, etc. Although $\mathcal{A}$ has a model-theoretic semantics, its operational semantics is given via a translation to logic programming. Since, unlike $\mathcal{TR}$, $\mathcal{A}$ was not intended as a language for combining actions into complex transactions, it has no facility for defining procedures and subroutines.

The Situation Calculus

The situation calculus is a methodology for specifying the effects of actions in first-order predicate calculus. It was introduced by McCarthy [70] and then further developed by McCarthy and Hayes [71]. To a large extent, the situation calculus is orthogonal to $\mathcal{TR}$: whereas the situation calculus focuses on specifying the effects of primitive actions, $\mathcal{TR}$ focuses on combining such actions into complex ones and executing them. In particular, the specification of elementary updates is factored out of $\mathcal{TR}$, as a transition oracle, which can be specified either procedurally or declaratively.

Besides this difference in intent, there is a semantic difference in the way the two formalisms define states of the world. Informally, in the situation calculus, states are historical, whereas in $\mathcal{TR}$, they are not. In $\mathcal{TR}$, a state is a database, *i.e.*, a finite set of first-order formulas, and different transactions may terminate at the same state. In contrast, in the situation calculus, a state is identified with a sequence of actions, and different transactions always terminate at different states. For example, the state resulting from *insert a, then insert b* is different from the state resulting from *insert b, then insert a.* To take a less abstract example, in the situation calculus, the following two action sequences result in different states:

- Add two gallons of water to an empty barrel, then add four gallons.

[28] [73] does not use the syntax of $\mathcal{TR}$, but these two rules can be translated into their syntax.

- Add four gallons of water to an empty barrel, then add two gallons.

Equality statements can be used to make the resulting states equal. One might even be able to infer the equality of many states by encoding a theory of commutative actions in first-order logic. Then, in a sequence of commutative actions, such as the two above, the order of the actions would not affect the final state. However, commutativity does not capture all the ways in which states can be equal [12]. For instance, commutativity alone does not tell us that the two action sequences above result in the same state as the following action sequence:

- Add five gallons of water to an empty barrel, then add one gallon.

This example suggests that arbitrary arithmetical reasoning may be needed to infer whether two action sequences terminate at the same state. It is therefore unlikely that all states with equal extent can be inferred to be equal within first-order logic.

In addition to the two basic difference described above, $\mathcal{TR}$ and the situation calculus have numerous other differences, which reflect their different intent and semantics:

- The ability to represent elementary updates procedurally solves the frame problem for many practical situations and makes $\mathcal{TR}$ well-suited to transaction execution. In contrast, the situation calculus deals with the frame problem declaratively, so it is better suited to reasoning about elementary updates in the presence of incomplete information.

- The situation calculus uses *state variables* whereas $\mathcal{TR}$ does not. The value of a state variable is a function term. In general, the state resulting from a sequence of n transactions is a function term with over n distinct subterms. These function terms could therefore reach enormous proportions in database systems, where performance is measured in thousands of transactions per second. State variables provide considerable power for reasoning about action, but they can be cumbersome, and they make programming awkward (just as cpu registers make assembly language awkward).

- The situation calculus makes a sharp distinction between queries and actions: queries are logical formulas, and actions are function terms. This approach forces actions to be deterministic and may simplify reasoning. However, the approach is ill-suited for modeling systems that do not distinguish actions and queries and can hinder application development, as described above.

- In $\mathcal{TR}$, actions can be combined in complex ways. Sequential composition, subroutines, recursion, conditional statements, iterative loops, non-determinism, and dynamic constraints are all available, as are pre-conditions, post-conditions, and intermediate conditions. These constructs make $\mathcal{TR}$ ideal for programming and executing complex transactions. Of course, reasoning about such transactions is hopelessly expensive, in general, and is outside of *r.e.*[29]

  In contrast, the situation calculus provides only two basic mechanisms for defining actions: pre-conditions and sequential composition. Post-conditions, subroutines and other programming constructs are not available. Thus, a complex action is at most a fixed (*i.e.*, database-independent) sequence of elementary actions. Although this frugality is not conducive to the definition of complex transactions, it does make many kinds of reasoning possible, keeping it within *r.e.*

[29] *For instance*, Deciding whether a transaction succeeds from all initial database states is not in *r.e.*

We should also mention the recent work by Reiter on the frame problem in the situation calculus. In [86, 87, 85], Reiter develops a first-order formalization of actions that does not suffer from the usual blow-up in the number of frame axioms, and he partially solves certain satellite problems, such as the *ramification problem* [35, 66]. These results alleviate much of the complexity associated with the frame problem, and permit reasoning about actions using first-order logic. However, the results presented in [86, 87, 85] apply only to transactions defined in the situation calculus. Thus, a user must specify a transaction as a fixed sequence of elementary actions. In addition, all elementary actions and view definitions must be known in advance, and adding new ones requires re-programming the frame axioms for the entire database. In [85], Reiter shows how to do this automatically, but the algorithm is non-incremental, so that every view and every action must be "re-compiled" into frame axioms each time any one of them changes. This makes the addition of transactions to large transaction bases difficult, if not infeasible, since it may require bringing the entire database system down each time a single transaction or view is modified. It also precludes the possibility of ad hoc updates, in the style of SQL, since such updates are not known in advance, and have not been compiled into the system. Finally, actions in [86, 87, 85] can insert and delete atomic formulas, but not arbitrary logical formulas. Thus, an action cannot add new rules to a deductive database, nor can it add tuples with null values to a relational database. None of these limitations apply to transactions in $\mathcal{TR}$. As usual, this is possible because $\mathcal{TR}$ focuses on execution, not on general reasoning. Unfortunately, it seems impossible to generalize Reiter's results to arbitrary transactions and remain within a first-order framework, since, in general, reasoning about transactions is not in *r.e.*

GOLOG

Based on the situation calculus, Reiter et al. have developed a new language, called GOLOG, which includes primitives for combining actions into more complex ones, and which overcomes some of the inefficiencies associated with the situation calculus by including a theory of state materialization. This development addresses some of the problems that motivated $\mathcal{TR}$. However, there are numerous differences between $\mathcal{TR}$ and GOLOG. First, unlike $\mathcal{TR}$, GOLOG depends crucially on a particular theory of elementary updates (the situation calculus), which is intertwined with a particular theory of database states (situation terms). Second, GOLOG relies on a complex encoding into second-order logic for its semantics, while $\mathcal{TR}$ has a simple and direct model theory. Third, GOLOG is not a Horn-like language in the logic-programming tradition, but is more reminiscent of Algol (from whence its name is derived).

This last point reflects a much deeper difference between $\mathcal{TR}$ and GOLOG: GOLOG is fundamentally incompatible with the logic-programming paradigm. In particular, Horn logic programs cannot be combined with GOLOG programs. For instance, the following familiar logic program for appending two lists cannot be combined with or used by a GOLOG program:[30]

$$append([X|List_1], List_2, [X|List_3]) \leftarrow append(List_1, List_2, List_3)$$
$$append([\,], List, List) \leftarrow$$

Instead, to append lists, one must write a (much more procedural) GOLOG program. For the same

[30] This point requires some elaboration. Like Procedural Logic (described below), GOLOG is actually two separate things: a logical framework, and a Prolog implementation. The implementation can certainly invoke arbitrary Prolog predicates. However, this expedient is not accounted for by the logical framework of GOLOG, and thus has no logical semantics. In our discussion, the term "GOLOG" refers to the logical framework.

reason, GOLOG has difficulties in dealing with database views. For instance, GOLOG cannot be combined with Horn rules defining the transitive closure of a relation.

This incompatibility of GOLOG with Logic Programming and deductive databases is rooted in GOLOG's commitment to the situation calculus and the frame problem. In database terms, the frame problem states that when a transaction updates a database, nothing else changes. However, views and logic programs contradict this idea, because they imply that derived predicates can change as a side effect of database updates. (In AI terms, this is a special case of the *ramification problem.*) To deal with database views, the situation calculus effectively compiles all view definitions into all transaction definitions (so that transactions explicitly update views, as well as base data) [85]. Unfortunately, this approach complicates both the theory and practice of databases. More seriously, it works only for non-recursive views, and breaks down entirely when views are recursive.

In contrast, none of these problems arise in $\mathcal{TR}$. This is because $\mathcal{TR}$ was designed as an extension of classical logic programming. Any classical logic program can be used by $\mathcal{TR}$ by simply adding it to the transaction base. Likewise, views, recursive or not, are easily defined in $\mathcal{TR}$ by sets of Horn rules, exactly as in the classical theory of deductive databases. The presense of database updates in $\mathcal{TR}$ does not complicate these matters at all. This is because the semantics of $\mathcal{TR}$ distinguishes between stored data and derived data, as described in Section 5.3.

Apart from the differences mentioned above, some other differences between $\mathcal{TR}$ and GOLOG are worth mentioning. First, the mechanisms of subroutine definition and recursion are first-order in $\mathcal{TR}$, while they are second-order in GOLOG. Second, because the semantics of GOLOG is closely intertwined with the semantics of states, GOLOG needs one interpreter for closed-world databases and another for open-world databases. Third, complex actions in GOLOG are not logical objects (they are macros), so one cannot refer to them in the logic, and in particular, one cannot quantify over and reason about them. Finally, because GOLOG depends on the situation calculus, there are many kinds of states that GOLOG cannot represent, including Prolog programs with negation-as-failure. Likewise, there are many kinds of updates that GOLOG cannot represent, including the insertion of rules into deductive databases, and the insertion of disjunctions into disjunctive databases.

The Event Calculus

The event calculus is also a methodology for encoding actions in first-order predicate logic. It was developed by Kowalski and Sergot for reasoning about time and events in a logic-programming setting [60, 59] and for overcoming some of the problems of the situation calculus. Like the situation calculus (and unlike $\mathcal{TR}$), the event calculus makes a sharp distinction between actions and queries. However, inspired by James Allen's treatment of time and action [7], the event calculus focuses on time intervals instead of states. Its use of intervals is somewhat dual to $\mathcal{TR}$'s use of paths: in $\mathcal{TR}$, actions take place over paths, and facts are true at states; but in the event calculus, facts are true over intervals, and actions take place instantaneously. Like the situation calculus, but unlike $\mathcal{TR}$, the event calculus does not store the current database state. Instead, it records a history of events (database transactions), from which it can infer the state of the database at any given time. Furthermore, under certain conditions, it is possible to index the events and achieve efficiency comparable to assert and retract in Prolog [59]. This is possible, for instance, if events are write-only, *i.e.*, if they change the database without reading it, as in "Set Joe's salary to $50,000." When events can both read and write, efficiency is still possible if the database records not only a description of each event, but also its effects. Thus, given the event "Add $5,000 to Joe's salary," the new salary would be computed and stored in the database, in addition to storing the event itself. In this way, salaries would not have to be recomputed from time zero with each salary query.[31] In a logic-programming setting, the event calculus relies on negation-as-failure to solve the frame problem. In contrast, $\mathcal{TR}$ solves the frame problem in a purely monotonic logic, and its proof theory and model theory are both first-order.

Like the situation calculus, all actions in the event calculus are elementary. Thus, there is no way to define complex actions in terms of simpler ones. A user can specify that one event occurs before another, but he cannot construct transactions using conditional statements, iterative loops, and subroutines. From the database perspective, however, it seems that the event calculus was not intended as a language for building complex transactions. Rather, it can be seen as a logical account of database logs. By centering on semantically meaningful events, it provides flexible support for historical databases.

Thus, the event calculus seems to be complementary to $\mathcal{TR}$. Indeed, since $\mathcal{TR}$ includes all of first-order logic as a special case, this calculus can be incorporated in $\mathcal{TR}$ as a methodology for reasoning about events and supporting historical databases. Most database applications, however, are not historical, and in these cases, $\mathcal{TR}$ supports updates without the added cost that historical databases demand. $\mathcal{TR}$ can also be of benefit to the event calculus, as it provides a logical mechanism for inserting events and their effects into the database, for revising incorrect assertions, and for combining simple events into complex transactions—a mechanism that is ostensibly lacking in [59].

Approaches Based on Temporal Logic

In addition to the situation and event calculi, there are other approaches to modeling actions within first-order logic (*e.g.*, [7, 74]). Generally, these approaches are concerned with difficult problems in Artificial Intelligence and Linguistics. Often they are concerned with the interaction of time and action with belief, intention and causality. As such, the number and the complexity of the axioms for these systems is quite large compared to the proof theory of $\mathcal{TR}$. More importantly, developing a declarative programming language (like $\mathcal{TR}$) for specifying and executing transactions is not their main concern. Indeed, these formalisms were intended for *reasoning about* action, not for actually updating databases. Such reasoning is feasible if actions are relatively simple. However, once actions are as complex as programs, reasoning about them is outside of *r.e.*, and so it cannot be formulated in first-order logic. Inference systems for such reasoning may be sound, but they cannot be complete. In contrast, because $\mathcal{TR}$ focuses on execution, this problem does not arise, and inference in $\mathcal{TR}$ is *both* sound and complete. Many of our comments about the situation calculus also apply to other encodings of action in first-order logic, especially those comments about the frame problem and the use of explicit time variables.

Besides these general comments, the work by James Allen deserves special mention [7, 6, 8]. This is because actions in Allen's logic are not instantaneous, but take place over time *intervals*, much as actions in $\mathcal{TR}$ take place over *paths*. Allen's logic supports a rich set of relations between actions: besides coming before or after one another, actions can overlap, start at the same time, end at the same time, etc. As illustrated in Section 7.7.2, similar relations can be expressed in $\mathcal{TR}$. Allen's logic can also express sequential planning goals, such as, "First boil the water, and then add the coffee" [8]. As illustrated in Section 7.9.2, such goals can also be expressed in $\mathcal{TR}$. Generally, though, the two logics have different applications. Allen's logic is intended for *reasoning* about action, whereas $\mathcal{TR}$ provides *executable specifications* for complex actions. Specifically, Allen's logic uses temporal

[31] R. Kowalski, personal communication

variables, which provide a powerful reasoning capability, while $\mathcal{TR}$ uses a transition oracle, which provides efficient execution.

The work of Halpern and Shoham [47], which is based on Allen's logic of time intervals, develops a propositional modal logic for reasoning about time intervals, including results about undecidability. Although, by following this approach, one can represent some aspects of transaction programming, it is not completely adequate for the task. For instance, there is no counterpart to $\mathcal{TR}$'s transition oracle, nor to the notion of executional entailment, and it is not clear that subroutines can be expressed in a straightforward way. In addition, although actions can be sequenced, this is not done using a logical connective of serial conjunction, as in $\mathcal{TR}$. Consequently, there is no counterpart to $\mathcal{TR}$'s dual connective of serial disjunction, nor to serial implication. Thus, there is no obvious way to express constraints based on these connectives, such as those illustrated in Section 7.7.2. Finally, although the logic is propositional, validity is already outside of *r.e.* for some important models of time, including real time and integer time.

In [37], Gabbay uses USF, a temporal logic of integer linear time with operators *since*, *until*, and *fixpoint*, to model execution of processes. Like $\mathcal{TR}$, USF queries the database via an analogue of the state data oracle, but USF does not use transition oracles. Apart from this and the emphasis on execution, the details and the capabilities of the two logics are very different.

USF uses a standard temporal semantics; in particular, all formulas are true at states, and execution is modeled by the process of constructing temporal models for the time points in the future of now. In contrast, $\mathcal{TR}$'s formulas are true on paths, and execution is modeled by verifying that a transaction holds on a path in all models of the program. Thus, in TRR, execution corresponds to logical entailment, while in USF it corresponds to satisfiability. Consequently, $\mathcal{TR}$ uses a proof theory to execute actions, while USF uses a model-building procedure.

The goals of the two logics are also quite different. USF was intended as a language for process control, so it is best suitable for specifying dynamic properties of executing processes. $\mathcal{TR}$, on the other hand, emphasizes action definition through subroutines and composition, so it is best suited as a logic for defining and executing complex transaction. Moreover, if USF were applied to specifying database transactions, it would have to face the dreaded frame problem.

In [76], Moszkowski develops an imperative programming language, called *Tempura*, based on Interval Temporal Logic (ITL) [46, 75]. Moszkowski shows how each Tempura statement translates into ITL, shows how to specify numerous applications in Tempura, and describes an interpreter for executing Tempura programs. There are important similarities between Tempura and $\mathcal{TR}$. For instance, Tempura emphasizes the *execution* of programs specified in logic. Tempura also evaluates formulas on paths, that is, on sequences of states; and it has an operator of sequential composition, called *chop*, which is identical to serial conjunction in $\mathcal{TR}$. Finally, like serial-Horn $\mathcal{TR}$, Tempura represents a subset of a larger logic. However, there are also many significant differences between Tempura and $\mathcal{TR}$, which we itemize below.

- Tempura is not a *database* programming language. Indeed, there is no notion of database or of persistence in Tempura or in ITL, and there is no analogue to $\mathcal{TR}$'s transition oracle. Furthermore, if applied to databases, Tempura and ITL would both suffer from the frame problem and the ramification problem (Section 7.8).

- Although it has a logical semantics, Tempura is an *imperative* programming language, not a *logic* programming language. Consequently, Tempura has no built-in facilities for unification, for backtracking, or for non-determinism.

- Although a notion of serial conjunction is defined, the dual notion of serial disjunction is not considered, nor is serial implication. Consequently, the kind of dynamic constraints described in Section 7.7 were not developed.
- Tempura represents an executable fragment of ITL, but a fragment without a clean, simple characterization. In particular, Tempura is not a "Horn" fragment of ITL.
- Unlike serial-Horn $\mathcal{TR}$ (and classical Horn logic), Tempura lacks an efficient SLD-style inference system. In fact, proof theory is barely mentioned in [76].
- Although Tempura has subroutines, they cannot be expressed within ITL [76]. It may be possible to extend ITL to include (recursive) subroutines, perhaps by adding a fixpoint operator, but [76] provides no details.

Finally, it is worth comparing $\mathcal{TR}$, ITL and Process Logic (discussed earlier). As in $\mathcal{TR}$ and Process Logic, formulas in ITL are true on paths, that is, on sequences of states. However, the semantics of ITL and Process Logic are fundamentally different from that of $\mathcal{TR}$. For instance, in both ITL and Process Logic, an atom is true on a path iff it is true at the first state of the path. In contrast, in $\mathcal{TR}$, the truth of atoms on a path is interpreted as *execution* and is independent of the states that make up the path. Such dependencies can be described in $\mathcal{TR}$ by logical axioms, but they are not built into the semantics. One consequence is that in ITL and Process Logic, an atom can only represent a static fact, such as $age(tony, 32)$, whereas in $\mathcal{TR}$, an atom can represent an action, such as $promote(tony, manager)$. This is one reason why ITL must be extended in order to define subroutines, since there is no facility for assigning names to complex actions.

Procedural Logic

Georgeff and Lansky develop a formalism called Procedural Logic whose intent is very similar to that of $\mathcal{TR}$ [42, 41]. Both systems focus on executable specifications of transactions, both provide a subroutine facility, both allow for arbitrary primitive actions, non-deterministic actions, and both provide some kind of constraints. Indeed, we are in complete agreement with Georgeff and Lansky on the limitations of existing logics of action. However, there are important differences between Procedural Logic and $\mathcal{TR}$, both practically and philosophically. These difference arise largely because Procedural Logic is actually two separate things: An implementation (called *PRS*), which provides the aforementioned functionality; and a formalization, which accounts for only a fraction of this functionality. It is probably fair to say that $\mathcal{TR}$ is a formalization that accounts for almost all of PRS and more. In the following discussion, we use the term "Procedural Logic" to refer to the formalization.

The most obvious difference between $\mathcal{TR}$ and Procedural Logic is syntactic: Whereas $\mathcal{TR}$ uses Horn-like rules to define transactions, Procedural Logic uses what in AI are called "recursive transition networks," which are a generalization of finite automata. Furthermore, the status of transactions in the two formalisms is quite different. In $\mathcal{TR}$, transaction definitions are logical statements, while in Procedural Logic, they are modal operators, as in Process Logic and Dynamic Logic. Each transition network defines a distinct modal operator.

A more important difference concerns functionality: Procedural Logic does not have a database. As Georgeff and Lansky remark, having an updatable database "introduces all the standard planning issues, such as the frame problem and consistency maintenance" [42]. These are problems that $\mathcal{TR}$ has resolved. Because it lacks a database, Procedural Logic cannot be used for specifying or executing database transactions. The implementation, PRS, does not have this limitation, but it is not a logic and provides no rigorous account of actions. Likewise, only in the implementation can a user specify primitive updates. This is impossible in the logic itself, which has no notion of database and thus no notion of update, primitive or otherwise. In $\mathcal{TR}$, primitive executable actions are specified in the transition oracle.

Another important difference between $\mathcal{TR}$ and Procedural Logic is semantic. Like $\mathcal{TR}$, Procedural Logic claims to have a declarative semantics. However, the declarative semantics of Procedural Logic is still very procedural, and seems to anticipate a Prolog-style backtracking interpreter. It is for this reason that the semantics is burdened with the notions of "successful" and "failed" behaviors. Indeed, as Georgeff and Lansky remark, "the declarative semantics is surprisingly complex" [42], even though databases and updates are not even included! In contrast, $\mathcal{TR}$ has an elegant model-theoretic semantics that accounts for databases, primitive updates, complex transactions, subroutines, and executions that succeed or fail.

Finally, like $\mathcal{TR}$, Procedural Logic has an operational semantics. However, unlike $\mathcal{TR}$, the operational semantics of Procedural Logic is an *interpreter* for the logic, not a logical inference system. Moreover, the interpreter is not complete with respect to the declarative semantics. The main problem is that the interpreter cannot rollback failed transactions, just as Prolog cannot rollback *assert* and *retract* operations. Indeed, as pointed out in [42], soundness *without* completeness "is the best one can really hope for when any particular selection may cause some possibly irreversible action." Note that without rollback, Procedural Logic cannot model sub-transactions. In contrast, $\mathcal{TR}$ provides a logical inference system that is both sound and complete with respect to its model-theory. It thus provides both a theory and a procedure for dealing with transaction rollback and sub-transactions. Furthermore, the procedure is a practical SLD-style inference system based on resolution. Thus, unlike Procedural Logic (or PRS), $\mathcal{TR}$ extends the advantages and elegance of pure Prolog to Prolog with updates.

A Appendix: Soundness of $\Im^I$

This appendix contains proofs of soundness of the inference system $\Im^I$ developed in Section 6.3. The proof for system $\Im^{II}$ is exactly parallel. Unlike completeness, soundness holds for arbitrary formulas and transaction bases, and does not require the serial Horn conditions of Section 6.1.

For convenient reference, we reproduce the axioms and inference rules of system $\Im^I$ below. Recall that the inference system assumes a data oracle, $\mathcal{O}^d$, and a state transition oracle, $\mathcal{O}^t$. As described in Sections 3 and 5, the data oracle specifies the semantics of database states, and the transition oracle specifies the semantics of elementary state transitions. Intuitively, if b is a first-order formula and $b \in \mathcal{O}^d(\mathbf{D})$, then b is true in database state $\mathbf{D}$. Likewise, if $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, then b is an elementary transition that transforms database $\mathbf{D}_1$ into $\mathbf{D}_2$. The rest of this appendix assumes that these two oracles are given.

Definition A.1 (Inference System $\Im^I$) If $\mathbf{P}$ is a transaction base, then $\Im^I$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.

Axioms: $\mathbf{P}, \mathbf{D} \text{---} \vdash (\,)$.

Inference Rules: In Rules 1–3 below, σ is a substitution, a and b are atomic formulas, and ϕ and $rest$ are serial goals.

1. *Applying transaction definitions:*
   Suppose $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with $b \otimes rest$. If a and b unify with mgu σ, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(b \otimes rest)}$$

2. *Querying the database:*
   If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^d(\mathbf{D}) \models^c (\exists)\, b\sigma$, then

$$\frac{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,(b \otimes rest)}$$

3. *Performing elementary updates:*
   If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)\, b\sigma$, then

$$\frac{\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)}$$

□

The following simple theorem is the main result of this section.

Theorem A.2 (Soundness of $\Im^I$) *If* $\mathbf{P}, \mathbf{D} \text{---} \vdash \phi$ *then* $\mathbf{P}, \mathbf{D} \text{---} \models \phi$

To prove Theorem A.2, it is enough to show that the axioms and inference rules of system $\Im^I$ are sound. Lemma A.3 below shows that the axioms are sound. Lemma A.4 shows that the inference rules are sound.

Lemma A.3 $\mathbf{P}, \mathbf{D} \models (\,)$ *for any transaction base* $\mathbf{P}$ *and any database* $\mathbf{D}$.

Proof: Recall that ( ) is an abbreviation for the special proposition `state`, which is true on all states. By definition, $\mathbf{M}, \langle D \rangle \models$ `state` for all path structures, $\mathbf{M}$, and in particular, for all models of $\mathbf{P}$. Thus $\mathbf{P}, \mathbf{D} \models$ `state`. *i.e.*, $\mathbf{P}, \mathbf{D} \models (\,)$. □

Lemma A.4 *Suppose b and $rest$ are two transaction formulas, and σ is a substitution.*

1. *Suppose that $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables may have been renamed, and suppose that a and b unify with mgu σ.*
   If $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,(\phi \otimes rest)\sigma$ *then* $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,(b \otimes rest)$.
2. *Suppose $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^d(\mathbf{D}_1) \models^c (\exists)\,b\sigma$, then*
   If $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,rest\,\sigma$ *then* $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,(b \otimes rest)$.
3. *Suppose $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)\,b\sigma$, then*
   If $\mathbf{P}, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\,rest\,\sigma$ *then* $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,(b \otimes rest)$.

Proof: We prove each item in turn.

• To prove item *1*, first note that since $a \leftarrow \phi$ is in $\mathbf{P}$, every model of $\mathbf{P}$ is also a model of $a \leftarrow \phi$ and, therefore, of $(a \leftarrow \phi)\sigma$. With this in mind, suppose that $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,(\phi \otimes rest)\sigma$. Then for every model $\mathbf{M}$ of $\mathbf{P}$,

$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models (\exists)\,(\phi \otimes rest)\sigma$	
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models_\nu (\phi \otimes rest)\sigma$	for some variable assignment, ν, by Definition 5.2;
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_i \rangle \models_\nu \phi\sigma$ *and* $\mathbf{M}, \langle \mathbf{D}_i \ ... \ \mathbf{D}_n \rangle \models_\nu rest\,\sigma$	for some i, by Definition 5.2;
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_i \rangle \models_\nu a\,\sigma$ *and* $\mathbf{M}, \langle \mathbf{D}_i \ ... \ \mathbf{D}_n \rangle \models_\nu rest\,\sigma$	since $\mathbf{M}$ is a model of $(a \leftarrow \phi)\sigma$;
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_i \rangle \models_\nu b\,\sigma$ *and* $\mathbf{M}, \langle \mathbf{D}_i \ ... \ \mathbf{D}_n \rangle \models_\nu rest\,\sigma$	since $a\,\sigma = b\,\sigma$, as $\sigma = mgu(a, b)$;
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models_\nu (b \otimes rest)\sigma$	by Definition 5.2;
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models_\mu (b \otimes rest)$	where $\mu(X) = \nu(X\sigma)$, for every variable X;
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models (\exists)\,(b \otimes rest)$	by Definition 5.2.

Since this is true for every model, $\mathbf{M}$, of $\mathbf{P}$, it follows that $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,(b \otimes rest)$.

• To prove item *2*, suppose that $\mathcal{O}^d(\mathbf{D}_1) \models^c (\exists)\,b\sigma$. Then $\mathbf{M}, \langle \mathbf{D}_1 \rangle \models (\exists)\,b\sigma$ for *any* path structure, $\mathbf{M}$, by Definition 5.1. Thus $\mathbf{M}, \langle \mathbf{D}_1 \rangle \models_\mu b\sigma$ for some variable assignment, μ. With this in mind, suppose that $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\,rest\,\sigma$. Then for every model, $\mathbf{M}$, of $\mathbf{P}$,

$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models (\exists)\,rest\,\sigma$	
$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models_\nu rest\,\sigma$	for some variable assignment, ν.

Let κ be a variable assignment that coincides with ν on the variables in $rest\,\sigma$, and with μ on the variables in $b\sigma$. This is possible since $b\sigma$ and $rest\,\sigma$ share no variables. Continuing:

$\mathbf{M}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models_\kappa rest\,\sigma$ by the previous item and the construction of κ;
$\mathbf{M}, \langle \mathbf{D}_1 \rangle \models_\kappa b\sigma$ since $\mathbf{M}, \langle \mathbf{D}_1 \rangle \models_\mu b\sigma$ and by construction of κ;
$\mathbf{M}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models_\kappa (b\sigma \otimes rest\,\sigma)$ by Definition 5.2;
$\mathbf{M}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models_\kappa (b \otimes rest)\sigma$
$\mathbf{M}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models_\eta (b \otimes rest)$ where $\eta(X) = \kappa(X\sigma)$, for every variable, X;
$\mathbf{M}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models (\exists)\,(b \otimes rest)$ by Definition 5.2.

Since this is true for every model, $\mathbf{M}$, of $\mathbf{P}$, it follows that $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \models (\exists)\,(b \otimes rest)$.

- To prove item *3*, suppose that $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists) b\sigma$. Then $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \rangle \models (\exists) b\sigma$ for *any* path structure, $\mathbf{M}$, by Definition 5.1. Thus, $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \rangle \models_\mu b\sigma$ for some variable assignment, μ. With this in mind, suppose that $\mathbf{P}, \mathbf{D}_2 \ldots \mathbf{D}_n \models (\exists)\, rest\,\sigma$. Then for every model, $\mathbf{M}$, of $\mathbf{P}$,

$\mathbf{M}, \langle \mathbf{D}_2 \ldots \mathbf{D}_n \rangle \models (\exists)\, rest\,\sigma$
$\mathbf{M}, \langle \mathbf{D}_2 \ldots \mathbf{D}_n \rangle \models_\nu rest\,\sigma$ for some variable assignment, ν.

As before, let κ be a substitution that coincides with ν on the variables in $rest\,\sigma$, and with μ on the variables in $b\,\sigma$. Then, as before, we have $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \rangle \models_\kappa b\sigma$ and $\mathbf{M}, \langle \mathbf{D}_2 \ldots \mathbf{D}_n \rangle \models_\kappa rest\,\sigma$. Therefore,

$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ldots \mathbf{D}_n \rangle \models_\kappa (b\sigma \otimes rest\,\sigma)$ by Definition 5.2;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ldots \mathbf{D}_n \rangle \models_\kappa (b \otimes rest)\sigma$
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ldots \mathbf{D}_n \rangle \models_\eta (b \otimes rest)$ where $\eta(X) = \kappa(X\sigma)$, for every variable, X;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ldots \mathbf{D}_n \rangle \models (\exists)\,(b \otimes rest)$ by Definition 5.2.

Since this is true for every model, $\mathbf{M}$, of $\mathbf{P}$, it follows that $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \models (\exists)\,(b \otimes rest)$. □

B Appendix: Completeness of $\Im^I$

This appendix proves that inference system $\Im^I$ developed in Section 6.3 is complete. (The proof for system $\Im^{II}$ is exactly parallel.) The axioms and inference rules of $\Im^I$ are conveniently reproduced in Appendix A. Our proof strategy is to first prove completeness of ground inference using a canonical-model construction. We then prove completeness of general inference using a Herbrand theorem and a lifting lemma. As in the other appendices, we assume that a Transition Oracle, $\mathcal{O}^t$, and a Data Oracle, $\mathcal{O}^d$, are given. We also assume that all transactions are existential serial goals, that the transaction base is a set of serial Horn rules, and that the other conditions of Definition 6.2 are satisfied.

B.1 Ground Inference

This section develops a ground version of inference system $\Im^I$ called *Ground* $\Im^I$. To simplify the proof of completeness, this new inference system also keeps track of the execution path, $\mathbf{D}_1 \ldots \mathbf{D}_n$, of a transaction. Systems $\Im^I$ and Ground $\Im^I$ use the same data oracle, $\mathcal{O}^d$, and the same transition oracle, $\mathcal{O}^t$. Both systems thus have the same set of database state identifiers, and the same semantics for states and transitions. In addition to the serial Horn conditions, this subsection assumes that the transaction base, $\mathbf{P}$, and all transaction invocations are ground.

Definition B.1 (Inference in Ground $\Im^I$) Let $\mathbf{P}$ be a (possibly infinite) transaction base that is ground and serial Horn. Then Ground $\Im^I$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.[32]

Axioms: $\mathbf{P}, \mathbf{D} \vdash (\,)$

Inference Rules: In Rules $1'$–$3'$ below, b is a ground atomic formula, and ϕ and $rest$ are ground serial goals.

$1'$. *Applying transaction definitions:* If $b \leftarrow \phi$ is a rule in $\mathbf{P}$, then:

$$\frac{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi \otimes rest}{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b \otimes rest}$$

$2'$. *Querying the database:* If $\mathcal{O}^d(\mathbf{D}_1) \models^c b$, then

$$\frac{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash rest}{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b \otimes rest}$$

$3'$. *Performing elementary transitions:* If $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c b$, then

$$\frac{\mathbf{P}, \mathbf{D}_2 ... \mathbf{D}_n \vdash rest}{\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash b \otimes rest}$$

□

Like $\Im^I$, the soundness of Ground $\Im^I$ follows immediately from Lemmas A.3 and A.4, which show that each axiom and inference rule of Ground $\Im^I$ is sound. We thus have the following result.

Theorem B.2 (Soundness of Ground $\Im^I$) *Let ϕ be a ground serial goal.*

$$\textit{If} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models \phi$$

Below, we give some properties of Ground $\Im^I$. The properties in Lemma B.3 are special cases of the inference rules with $rest = (\,)$. Lemma B.4 is a bit more difficult, and is central to our proof of completeness for ground inference.

Lemma B.3 (Basic Properties) *Ground $\Im^I$ has the following properties, where b is a ground atomic formula:*

1. *If $b \leftarrow \phi$ is a rule in $\mathbf{P}$, and if $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi$, then $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b$.*
2. *If $\mathcal{O}^d(\mathbf{D}) \models^c b$ then $\mathbf{P}, \mathbf{D} \vdash b$.*
3. *If $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c b$ then $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \vdash b$.*

Lemma B.4 *Ground $\Im^I$ has the following property, where α and β are ground serial goals:*

$$\textit{If} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash \alpha \quad \textit{and} \quad \mathbf{P}, \mathbf{D}_i ... \mathbf{D}_n \vdash \beta \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \alpha \otimes \beta$$

Proof: We shall transform a derivation of $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash \alpha$ into a derivation of $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \alpha \otimes \beta$. The proof is by induction over the length of the former derivation.

[32] Note that the databases need not be ground; that is, $\mathcal{O}^d(\mathbf{D})$ may contain variables.

Basis: Suppose that the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash \alpha$ can be derived in 1 step. It is therefore an axiom, so $i = 1$ and $\alpha = (\,)$. Thus, $\alpha \otimes \beta = \beta$, so the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \alpha \otimes \beta$ is identical to the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \beta$, which is identical to the sequent $\mathbf{P}, \mathbf{D}_i ... \mathbf{D}_n \vdash \beta$, which is given to be true.

Induction: Suppose the lemma is true for all sequents $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash \alpha$ derivable in at most N steps. Since α is a ground serial goal, it has the form $b \otimes rest$, where b is a ground atomic formula, and $rest$ is a ground serial goal. Suppose the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash b \otimes rest$ can be derived in $N + 1$ steps. We must show that $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b \otimes rest \otimes \beta$ is derivable. There are three cases, depending on the last inference rule used in the derivation.

- If Rule 1 is the last inference rule used, then $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash \phi \otimes rest$, for some rule $b \leftarrow \phi$ in $\mathbf{P}$. Moreover, this sequent can be derived in at most N steps. Hence, by induction hypothesis, $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi \otimes rest \otimes \beta$. Finally, by inference rule 1, we have $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b \otimes rest \otimes \beta$.

- If rule 2 is the last inference rule used, then $\mathcal{O}^d(\mathbf{D}_1) \models^c b$ and $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash rest$. Moreover, the last sequent can be derived in at most N steps. Hence, by induction hypothesis, $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash rest \otimes \beta$. Finally, by inference rule 2, we have $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b \otimes rest \otimes \beta$.

- If rule 3 is the last inference rule used, $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c b$ and $\mathbf{P}, \mathbf{D}_2 ... \mathbf{D}_i \vdash rest$, where the latter sequent can be derived in at most N steps. Hence, by induction hypothesis, $\mathbf{P}, \mathbf{D}_2 ... \mathbf{D}_n \vdash rest \otimes \beta$. Finally, by inference rule 3, we have $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b \otimes rest \otimes \beta$. $\square$

B.2 Completeness of Ground $\Im^I$

To prove the completeness of Ground $\Im^I$, we construct a canonical Herbrand model of the transaction base $\mathbf{P}$. This model, constructed proof theoretically, provides the necessary link between the proof theory and model theory. Once again, in addition to the serial Horn conditions, this subsection assumes that the transaction base $\mathbf{P}$ and all transactions are ground.

Given a language for $\mathcal{TR}$, we define the Herbrand universe and classical Herbrand interpretations in a standard way. The *Herbrand universe* is the set of all ground terms that can be constructed out of the symbols in $\mathcal{F}$, and a *Herbrand interpretation* is a semantic structure where the domain is the Herbrand universe and where function symbols are interpreted in a fixed way: $I_{\mathcal{F}}(f)(t_1, ..., t_n) = f(t_1, ..., t_n)$. The symbol $\mathcal{HB}$ shall denote the set of all ground atomic formulas, which is commonly known as the *Herbrand base*. There is a 1–1 correspondence between Herbrand interpretations and subsets of $\mathcal{HB}$, so we shall often treat them as the same thing.

Recall from Definition 5.1 that a path structure is a triple $\langle U, I_{\mathcal{F}}, I_{path} \rangle$, where U is the domain, $I_{\mathcal{F}}$ interprets function symbols, and I_{path} interprets paths. A *Herbrand path structure* is any path structure whose domain is the Herbrand universe and whose function symbols have the Herbrand interpretation. Thus, since U and $I_{\mathcal{F}}$ are the same for all Herbrand path structures, we can omit them and specify Herbrand path structures in terms of I_{path} alone.

Definition B.5 (Canonical Model) The canonical model of a transaction base $\mathbf{P}$ is the Herbrand path structure $\mathbf{M_P} = \langle I_{path} \rangle$, where

$$I_{path}(\langle \mathbf{D}_1 \; ... \; \mathbf{D}_n \rangle) \quad = \quad \{ \; b \in \mathcal{HB} \;\; | \;\; \mathbf{P}, \mathbf{D}_1, ..., \mathbf{D}_n \vdash b \; \}$$

$\square$

We first observe that $\mathbf{M_P}$ is a legitimate path structure. This follows immediately from Definition 5.1 and the following lemma.

Lemma B.6 *Let* $\mathbf{M_P} = \langle I_{path} \rangle$ *be the canonical model of* $\mathbf{P}$. *Then:*

(i) $I_{path}(\langle \mathbf{D} \rangle) \models^c \phi$ *for every closed first-order formula such that* $\phi \in \mathcal{O}^d(\mathbf{D})$
(so $\mathbf{M_P}, \langle \mathbf{D} \rangle \models \phi$*)*

(ii) $I_{path}(\langle \mathbf{D}_1\ \mathbf{D}_2 \rangle) \models^c b$ *for every* b *such that* $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$
(so $\mathbf{M_P}, \langle \mathbf{D}_1\ \mathbf{D}_2 \rangle \models b$*)*

Proof: Recall that $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ is a set of ground atomic formulas. Item (ii) then follows directly from Lemma B.3 and Definition B.5. That is, if $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ then $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \vdash b$ so $b \in I_{path}(\langle \mathbf{D}_1\ \mathbf{D}_2 \rangle)$ so $I_{path}(\langle \mathbf{D}_1\ \mathbf{D}_2 \rangle) \models^c b$.

To prove item (i), recall that because of the serial Horn restrictions, $\mathcal{O}^d(\mathbf{D})$ is *independent* of $\mathbf{P}$. That is, no predicate symbol in the head of a rule in $\mathbf{P}$ appears in the body of a rule in $\mathcal{O}^d(\mathbf{D})$. Thus, rules in $\mathcal{O}^d(\mathbf{D})$ do not use conclusions inferred by rules in $\mathbf{P}$.

Second, note that in the canonical model, every atom in $I_{path}(\langle \mathbf{D} \rangle)$ comes from a sequent of the form $\mathbf{P}, \mathbf{D} \vdash b$. These expressions are generated only by inference rules 1 and 2. Rule 2 infers $\mathbf{P}, \mathbf{D} \vdash b$ iff $\mathcal{O}^d(\mathbf{D}) \models^c b$, and rule 1 infers $\mathbf{P}, \mathbf{D} \vdash b$ only if b is the head of a rule in $\mathbf{P}$. Let S be the set of ground atomic formulas classically entailed by $\mathcal{O}^d(\mathbf{D})$. Thus, $I_{path}(\langle \mathbf{D} \rangle)$ consists of the atoms in S plus the heads of some rules in $\mathbf{P}$.

By the serial-Horn restrictions on the data oracle, S must contain some model of $\mathcal{O}^d(\mathbf{D})$. Since the rules in $\mathcal{O}^d(\mathbf{D})$ are independent of the rules in $\mathbf{P}$, any set obtained from S by adding some rule-heads of $\mathbf{P}$ is also a model of $\mathcal{O}^d(\mathbf{D})$. Since $I_{path}(\langle \mathbf{D} \rangle)$ is just such a set, it is a Herbrand model of $\mathcal{O}^d(\mathbf{D})$. Thus $I_{path}(\langle \mathbf{D} \rangle) \models^c \phi$ for every $\mathcal{O}^d(\mathbf{D}) \models^c \phi$. $\Box$

The following lemma states a basic result about canonical models. Theorem B.8 partially generalizes this result from atoms to serial goals. A full generalization is given in Corollary B.11.

Lemma B.7 *If* b *is a ground atomic formula, then*

$$\mathbf{M_P}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models b \quad \textit{iff} \quad \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash b$$

Proof: Follows immediately from Definitions B.5 and 5.2. $\Box$

Theorem B.8 *Let* ϕ *be a ground serial goal.*

$$\textit{If} \quad \mathbf{M_P}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models \phi \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash \phi$$

Proof: ϕ has the form $b_1 \otimes \ldots \otimes b_k$, where $k \geq 0$ and each b_i is a ground atomic formula. Our proof is by induction on k. In the base case, $k = 0$ and ϕ is the empty clause (*i.e.*, ( ), or `state`). If the expression $\mathbf{M_P}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models (\,)$ is true, then $n = 1$, since the empty clause is true only on states. But, the sequent $\mathbf{P}, \mathbf{D}_1 \vdash (\,)$ is trivially true, since it is an axiom.

For the inductive case, assume the claim is true for all values of k from 0 to m. We show that it is true for $k = m + 1$:

$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models b_1 \otimes ... \otimes b_m \otimes b_{m+1}$	given;
$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_i \rangle \models b_1 \otimes ... \otimes b_m$ *and* $\mathbf{M_P}, \langle \mathbf{D}_i \ ... \ \mathbf{D}_n \rangle \models b_{m+1}$	for some i, by Definition 5.2;
$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash b_1 \otimes ... \otimes b_m$ *and* $\mathbf{M_P}, \langle \mathbf{D}_i \ ... \ \mathbf{D}_n \rangle \models b_{m+1}$	by induction hypothesis;
$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash b_1 \otimes ... \otimes b_m$ *and* $\mathbf{P}, \mathbf{D}_i ... \mathbf{D}_n \vdash b_{m+1}$	by Lemma B.7;
$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b_1 \otimes ... \otimes b_m \otimes b_{m+1}$	by Lemma B.4.

□

Theorem B.9 $\mathbf{M_P}$ *is a model of* $\mathbf{P}$.

Proof: Let $b \leftarrow \phi$ be any rule in $\mathbf{P}$. Then, for any path $\langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle$,

if	$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models \phi$	
then	$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi$	by Theorem B.8;
	$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash b$	by Lemma B.3;
	$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models b$	by Lemma B.7.

$\mathbf{M_P}$ thus satisfies the rule $b \leftarrow \phi$ on every path. $\mathbf{M_P}$ is therefore a model of every rule in $\mathbf{P}$, so it is a model of $\mathbf{P}$. □

To prove the completeness of $\Im^I$, we do not need the completeness of Ground $\Im^I$ per se. Instead, we need Theorems B.8 and B.9. However, since all the machinery is already in place, we can state the completeness of Ground $\Im^I$ as an easy corollary.

Corollary B.10 (Completeness of Ground $\Im^I$) *For any ground serial goal,* ϕ,

$$\textit{if} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models \phi \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi$$

Proof: Suppose that $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models \phi$. Then:

$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models \phi$	for every model, $\mathbf{M}$, of $\mathbf{P}$, by Definition 5.6,
$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models \phi$	since $\mathbf{M_P}$ is a model of $\mathbf{P}$, by Theorem B.9,
$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi$	by Theorem B.8.

□

Although this completes the proof of completeness in Ground $\Im^I$, we include one more important corollary. This corollary generalizes Lemma B.7 from atoms to serial goals. It shows that inference in Ground $\Im^I$ is equivalent to satisfaction in the canonical model.

Corollary B.11 *If* ϕ *is a ground serial goal, then*

$$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models \phi \quad \textit{iff} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi$$

Proof: The *only if* direction is just Theorem B.8. The *if* direction follows from the soundness of Ground $\Im^I$, since, by Theorem B.9, $\mathbf{M_P}$ is a model of $\mathbf{P}$. □

B.3 Completeness of $\Im^I$

Having proved the completeness of Ground $\Im^I$, we now prove the completeness of $\Im^I$ itself. We first prove a result analogous to the Herbrand Theorem of classical logic, which lets us reduce general entailment to ground inference. Then we prove a lifting lemma, whose purpose is to "lift" ground inferences up to general inferences. In this subsection, we again assume the serial Horn conditions, but not groundness. Thus, transactions and the transaction base $\mathbf{P}$ may be non-ground. The ground instantiation of $\mathbf{P}$ is denoted $\mathbf{P}^*$. We shall need to distinguish between the sequents of inference systems $\Im^I$ and Ground $\Im^I$. In general, sequents of the form $\mathbf{P}, \mathbf{D} \text{---} \vdash \phi$ belong to $\Im^I$, while sequents of the form $\mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi$ belong to Ground $\Im^I$.

Lemma B.12 $\mathbf{M}_{\mathbf{P}^*}$ *is a model of* $\mathbf{P}$*.*

Proof: Let $b \leftarrow \phi$ be a rule in $\mathbf{P}$ with free variables $\overline{X} = \{X_1 ... X_m\}$. For any variable assignment, ν, let θ_ν be a substitution that replaces each variable $X_i \in \overline{X}$ with the Herbrand term $\nu(X_i)$. Note that $(b \leftarrow \phi)\theta_\nu$ is ground. Moreover, by the discussion surrounding Definition 5.2,

$$\mathbf{M}_{\mathbf{P}^*}, \pi \models_\nu b \leftarrow \phi \qquad \textit{iff} \qquad \mathbf{M}_{\mathbf{P}^*}, \pi \models (b \leftarrow \phi)\theta_\nu \tag{50}$$

for any path, π. Observe that the right-hand side of (50) is a true statement, since $(b \leftarrow \phi)\theta_\nu$ is a rule in $\mathbf{P}^*$, and $\mathbf{M}_{\mathbf{P}^*}$ is a model of $\mathbf{P}^*$, by Theorem B.9. Thus, the left-hand side of (50) is also true, for any variable assignment, ν. Thus,

$$\mathbf{M}_{\mathbf{P}^*}, \pi \models \forall \overline{X}\, (b \leftarrow \phi)$$

by Definition 5.2. Since this is true for every path, $\mathbf{M}_{\mathbf{P}^*}$ is a model of $b \leftarrow \phi$, by Definition 5.5. $\mathbf{M}_{\mathbf{P}^*}$ is thus a model of every rule in $\mathbf{P}$, so it is a model of $\mathbf{P}$. □

We can now prove a result that is analogous to Herbrand's Theorem in classical logic.

Corollary B.13 (Herbrand's Theorem for $\Im^I$) *Let ϕ be a serial goal.*

$$\textit{If} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\, \phi \quad \textit{then} \quad \mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi'$$

for some ground instance, ϕ', of ϕ.

Proof: If $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models (\exists)\, \phi$ then

$\mathbf{M}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models (\exists)\, \phi$	for every model, $\mathbf{M}$, of $\mathbf{P}$,
$\mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models (\exists)\, \phi$	since $\mathbf{M}_{\mathbf{P}^*}$ is a model of $\mathbf{P}$, by Lemma B.12,
$\mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models_\nu \phi$	for some variable assignment, ν,
$\mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models \phi\theta_\nu$	as described in the proof of Lemma B.12,
$\mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi\theta_\nu$	by Theorem B.8.

The result follows with $\phi' = \phi\theta_\nu$. □

Lemma B.14 (Lifting Lemma for $\Im^I$) *Let ψ be a serial goal, and let ψ' be one of its ground instantiations.*

$$\textit{If} \quad \mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash \psi' \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, \psi$$

Proof: The proof is by induction on the length of the ground derivation.

Basis: In this case, both inference expressions are axioms, so they are both trivially true.

Induction: Suppose the lemma is true for all ground derivations having at most N steps. Now suppose the sequent $\mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash \psi'$ is derivable in Ground $\Im^I$ in $N+1$ steps. There are three cases, depending on the last inference rule used in the derivation.

- Suppose inference rule $1'$ was used last in the ground derivation. Then ψ has the form $b \otimes rest$ and ψ' has the form $b' \otimes rest'$, where b and b' are atomic formulas. Moreover, for some rule $b' \leftarrow \phi'$ in $\mathbf{P}^*$, the sequent $\mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash \phi' \otimes rest'$ is derivable in Ground $\Im^I$ in at most N steps.

The rule $b' \leftarrow \phi'$ is a ground instantiation of some rule $a \leftarrow \phi$ in $\mathbf{P}$. We show below that the formula $\phi' \otimes rest'$ is a ground instantiation of $(\phi \otimes rest)\sigma$, where σ is the most general unifier (mgu) of a and b. Thus, by induction hypothesis, $\mathbf{P}, \mathbf{D}_1 \dashrightarrow \vdash (\exists)\,(\phi \otimes rest)\sigma$. From this we get $\mathbf{P}, \mathbf{D}_1 \dashrightarrow \vdash (\exists)\,(b \otimes rest)$ by inference rule 1 of $\Im^I$ and rule $a \leftarrow \phi$ in $\mathbf{P}$.

To show that $\phi' \otimes rest'$ is a ground instantiation of $(\phi \otimes rest)\sigma$, note that the atom b' is a ground instantiation of both a and b. The atomic formulas a and b therefore unify. Let σ be their mgu. The atom b' is thus a ground instantiation of $b\sigma = a\sigma$, *i.e.*, there is a ground substitution σ' such that $b' = a\sigma\sigma' = a\sigma\sigma'$. Hence,

$$\begin{aligned} b' \otimes rest' &= (b \otimes rest)\sigma\sigma'\sigma_1 \\ b' \leftarrow \phi' &= (a \leftarrow \phi)\sigma\sigma'\sigma_2 \end{aligned}$$

for some pair of ground substitutions, σ_1 and σ_2; the substitution $\sigma\sigma'$ instantiates the variables in a and b. The substitutions σ_1 and σ_2 instantiate any remaining variables. By inference rule 1, the variables in the rule $a \leftarrow \phi$ have been renamed so that it and $b \otimes rest$ have no variables in common. Thus, $(a \leftarrow \phi)\sigma\sigma'$ and $(b \otimes rest)\sigma\sigma'$ have no variables in common, since $\sigma\sigma'$ is a ground substitution. Thus, σ_1 and σ_2 instantiate different variables. Thus σ_1 and σ_2 commute; that is, $\sigma_2\sigma_1 = \sigma_1\sigma_2$.

Extracting subformulas from the formulas above, we get:

$$\begin{aligned} rest' &= rest\,\sigma\sigma'\sigma_1 = rest\,\sigma\sigma'\sigma_1\sigma_2 \\ \phi' &= \phi\,\sigma\sigma'\sigma_2 = \phi\,\sigma\sigma'\sigma_2\sigma_1 \end{aligned}$$

The rightmost equalities follow because applying substitutions to ground formulas has no effect. Combining these equalities, and using the commutativity of σ_1 and σ_2, we get:

$$\begin{aligned} \phi' \otimes rest' &= \phi\,\sigma\sigma'\sigma_1\sigma_2 \otimes rest\,\sigma\sigma'\sigma_1\sigma_2 \\ &= (\phi \otimes rest)\sigma\sigma'\sigma_1\sigma_2 \end{aligned}$$

Thus, the formula $\phi' \otimes rest'$ is a ground instantiation of $(\phi \otimes rest)\sigma$.

- Suppose inference rule $2'$ was used last in the ground derivation. Then ψ has the form $b \otimes rest$, and ψ' has the form $b\sigma \otimes rest\,\sigma$, for some ground substitution, σ. Moreover, $\mathcal{O}^d(\mathbf{D}_1) \models^c b\sigma$, and the sequent $\mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash rest\,\sigma$ is derivable in Ground $\Im^I$ in at most N steps. Thus, by induction hypothesis, the sequent $\mathbf{P}, \mathbf{D}_1 \dashrightarrow \vdash (\exists)\, rest\,\sigma$ is derivable in $\Im^I$, since $rest\,\sigma$ is (trivially) a ground instance of $rest\,\sigma$. Also, $b\sigma$ and $rest\,\sigma$ share no variables, since they are both ground. Thus, the sequent $\mathbf{P}, \mathbf{D}_1 \dashrightarrow \vdash (\exists)\,(b \otimes rest)$ is derivable in $\Im^I$, by inference rule 2.

- Suppose inference rule $3'$ was used last in the ground derivation. Then ψ has the form $b \otimes rest$, and ψ' has the form $b\sigma \otimes rest\,\sigma$, for some ground substitution, σ. Moreover, $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c b\sigma$, and the sequent $\mathbf{P}^*, \mathbf{D}_2 ... \mathbf{D}_n \vdash rest\,\sigma$ is derivable in Ground $\Im^I$ in at most N steps. By induction hypothesis, the sequent $\mathbf{P}, \mathbf{D}_2 \text{---} \vdash (\exists)\, rest\,\sigma$ is derivable in $\Im^I$, since $rest\,\sigma$ is (trivially) a ground instance of $rest\,\sigma$. Also, $b\sigma$ and $rest\,\sigma$ share no variables, since they are both ground. Thus, the sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(b \otimes rest)$ is derivable in $\Im^I$, by inference rule 3. $\Box$

Theorem B.15 (Completeness of $\Im^I$) *Let ϕ be a serial goal. Then:*

$$\text{If} \quad \mathbf{P}, \mathbf{D} \text{---} \models (\exists)\, \phi \quad \text{then} \quad \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$$

Proof: If $\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\, \phi$ then $\mathbf{P}, \mathbf{D}, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, \phi$ for some databases $\mathbf{D}_2 ... \mathbf{D}_n$, by Definition 5.6. Thus $\mathbf{P}^*, \mathbf{D}, \mathbf{D}_2 ... \mathbf{D}_n \vdash \phi'$, where ϕ' is some ground instance of ϕ, by Corollary B.13. Hence $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$, by Lemma B.14. $\Box$

Although this completes our proof of completeness in $\Im^I$, we include the following important corollary. It shows that inference in $\Im^I$ is equivalent to satisfaction in the canonical model.

Corollary B.16 *Let ϕ be a serial goal. Then:*

$$\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi \quad \textit{iff} \quad \mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}\ \mathbf{D}_2\ ...\ \mathbf{D}_n \rangle \models (\exists)\, \phi \quad \textit{for some databases } \mathbf{D}_2 ... \mathbf{D}_n.$$

Proof: The *only if* direction:

$\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$	given;
$\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\, \phi$	by Soundness, Theorem A.2;
$\mathbf{P}, \mathbf{D}, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, \phi$	for some $\mathbf{D}_2 ... \mathbf{D}_n$, by Definition 5.6;
$\mathbf{M}, \langle \mathbf{D}\ \mathbf{D}_2\ ...\ \mathbf{D}_n \rangle \models (\exists)\, \phi$	for all models, $\mathbf{M}$, of $\mathbf{P}$;
$\mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}\ \mathbf{D}_2\ ...\ \mathbf{D}_n \rangle \models (\exists)\, \phi$	since $\mathbf{M}_{\mathbf{P}^*}$ is a model of $\mathbf{P}$, by Lemma B.12.

The *if* direction:

$\mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}\ \mathbf{D}_2\ ...\ \mathbf{D}_n \rangle \models (\exists)\, \phi$	given;
$\mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}\ \mathbf{D}_2\ ...\ \mathbf{D}_n \rangle \models_\nu \phi$	for some variable assignment, ν, by Definition 5.2;
$\mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}\ \mathbf{D}_2\ ...\ \mathbf{D}_n \rangle \models \phi\theta_\nu$	as described in the proof of Lemma B.12;
$\mathbf{P}^*, \mathbf{D}, \mathbf{D}_2 ... \mathbf{D}_n \vdash \phi\theta_\nu$	by Theorem B.8;
$\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\, \phi$	by the lifting lemma, Lemma B.14.

$\Box$

C Appendix: Executional Deduction in $\Im^I$

Section 6.3.1 developed the notion of executional deduction, a special kind of deduction that corresponds to the intuitive notion of transaction execution. Executional deductions trace out the execution path of a transaction during inference. The proofs of soundness and completeness given in Appendices A and B are for general deductions in inference system $\Im^I$. This appendix adapts these proofs to executional deductions in $\Im^I$. (The proofs for system $\Im^{II}$ are exactly parallel.) As in Appendix B, we assume that all transactions are existential serial goals, that the transaction base is a set of serial Horn rules, and that the other conditions of Definition 6.2 are satisfied.

Our proof strategy is to define a new inference system called *Path* $\Im^I$. Path $\Im^I$ is identical to $\Im^I$ except that it records the execution path of a transaction. Sequents in Path $\Im^I$ have the form $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash \psi$, which intuitively means that $\mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n$ is an (inferred) execution path for transaction ψ. Path $\Im^I$ is similar to inference system Ground $\Im^I$ developed in Appendix B.1, except that the transaction base $\mathbf{P}$ and the transaction ψ need not be ground. Since Path $\Im^I$ is closely related to $\Im^I$ and Ground $\Im^I$, it is easy to show that Path $\Im^I$ is sound and complete. The only remaining step is to show that any general deduction in Path $\Im^I$ can be associated with an executional deduction in $\Im^I$, and vice-versa.

Definition C.1 (Inference in Path $\Im^I$*)* If $\mathbf{P}$ is a transaction base, then Path $\Im^I$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.

Axioms: $\mathbf{P}, \mathbf{D} \text{ --- } \vdash (\,)$.

Inference Rules: In Rules $1'$–$3'$ below, σ is a substitution, a and b are atomic formulas, and ϕ and *rest* are serial goals.

$1'$. *Applying transaction definitions:*
Suppose $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with $b \otimes rest$. If a and b unify with mgu σ, then

$$\frac{\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \text{ --- } \vdash (\exists)\, (\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

$2'$. *Querying the database:*
If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^d(\mathbf{D}) \models^c (\exists)\, b\sigma$, then

$$\frac{\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \text{ --- } \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

$3'$. *Performing elementary updates:*
If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)\, b\sigma$, then

$$\frac{\mathbf{P}, \mathbf{D}_2 \ldots \mathbf{D}_n \text{ --- } \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \text{ --- } \vdash (\exists)\, (b \otimes rest)}$$

□

Path $\Im^I$ is a trivial extension of $\Im^I$, and there is a one-to-one correspondence between derivations in $\Im^I$ and derivations in Path $\Im^I$. The soundness and completeness proofs for $\Im^I$ are easily adapted to Path $\Im^I$, to give Theorem C.2 below. Note that this theorem specifies the execution path both in the $\models$ sequent and in the $\vdash$ sequent. This theorem thus contains more information than the soundness and completeness theorems for $\Im^I$ (Theorems A.2 and B.15), in which the execution paths are unspecified.

Theorem C.2 (Soundness and Completeness of Path $\Im^I$**)**
If ψ *is a serial goal, then the following two statements are equivalent:*

1. $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \models (\exists)\, \psi$

2. *The sequent* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\, \psi$ *is derivable in Path* $\Im^I$.

Proof: Lemmas A.3 and A.4 show that the axioms and inference rules of Path $\Im^I$ are sound. This proves that item *2* implies item *1* (soundness). To prove the converse (completeness), suppose that $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \models (\exists)\, \psi$. Then $\mathbf{P}^*, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash \psi'$ is derivable in Ground $\Im^I$ for some ground instance, ψ', of ψ, by Corollary B.13.[33] We must now show that a derivation in Ground $\Im^I$ can be "lifted" to a derivation in Path $\Im^I$. That is, if ψ is a serial conjunction, and ψ' is one of its ground instantiations, then

if	$\mathbf{P}^*, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash \psi'$	*is derivable in Ground* $\Im^I$
then	$\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash (\exists)\, \psi$	*is derivable in Path* $\Im^I$

The proof of this is a straightforward adaptation of the proof of Lemma B.14. Completeness follows immediately. $\Box$

Theorem C.2 links executional entailment with inference in Path $\Im^I$. The next theorem finishes the job by linking inference in Path $\Im^I$ with executional deduction in $\Im^I$. The reader is referred to Definition 6.9 for a precise description of executional deduction, and to Section 6.3.1 for a description of execution paths.

Theorem C.3 *If* ψ *is a serial goal, then the following two statements are equivalent:*

1. *The sequent* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\, \psi$ *is derivable in Path* $\Im^I$.
2. *There is an executional deduction of* $(\exists)\, \psi$ *in* $\Im^I$ *whose execution path is* $\mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n$.

Proof: Since every executional deduction is also a general deduction, it is easy to show that item *2* implies item *1*. The proof is a straightforward induction on the length of executional deductions. To show the converse, that item *1* implies item *2*, suppose the sequent $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash (\exists)\, \psi$ is derivable in Path $\Im^I$. Our proof is by induction on the length of this derivation.

Basis: If a sequent is derivable in Path $\Im^I$ in one step, then it is an axiom of Path $\Im^I$. It therefore has the form $\mathbf{P}, \mathbf{D} \vdash (\,)$. But $\mathbf{P}, \mathbf{D} \text{---} \vdash (\,)$ is an axiom of $\Im^I$. Thus, there is an executional deduction of ( ) in $\Im^I$ whose execution path is $\mathbf{D}$.

Induction: Suppose that item *1* implies item *2* for all sequents derivable in Path $\Im^I$ in at most N steps. Now suppose the sequent $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\, (b \otimes rest)$ is derivable in Path $\Im^I$ in $N + 1$ steps. There are three cases, depending on the last rule of inference used in the derivation.

- Suppose that inference rule $1'$ is the last rule used in the derivation. Then the sequent $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash (\exists)\, (\phi \otimes rest)\sigma$ is derivable in Path $\Im^I$ in at most N steps. Here, $\sigma = mgu(a, b)$ for some rule $a \leftarrow \phi$ in $\mathbf{P}$. By induction hypothesis, there is an executional deduction of $(\exists)\, (\phi \otimes rest)\sigma$ in $\Im^I$ whose execution path is $\mathbf{D}_1 \ldots \mathbf{D}_n$. Using inference rule 1 in $\Im^I$, this can be extended by one sequent to an executional deduction of $(\exists)\, (b \otimes rest)$. Moreover, since this extension does not involve inference rule 3, the execution path is still $\mathbf{D}_1 \ldots \mathbf{D}_n$.

[33] Recall that $\mathbf{P}^*$ is the ground instantiation of $\mathbf{P}$.

• Suppose that inference rule $2'$ is the last rule used in the derivation. Then, the sequent $\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash (\exists)\, rest\,\sigma$ is derivable in Path $\Im^I$ in at most N steps. Here, $b\sigma$ shares no variables with $rest\,\sigma$ and $\mathcal{O}^d(\mathbf{D}) \models^c (\exists) b\sigma$. By induction hypothesis, there is an executional deduction of $(\exists)\, rest\,\sigma$ in $\Im^I$ whose execution path is $\mathbf{D}_1 \ldots \mathbf{D}_n$. Using inference rule 2 in $\Im^I$, this can be extended by one sequent to an executional deduction of $(\exists)\,(b \otimes rest)$. Since this extension does not involve inference rule 3, the execution path is still $\mathbf{D}_1 \ldots \mathbf{D}_n$.

• Suppose that inference rule $3'$ is the last rule used in the derivation. Then, the sequent $\mathbf{P}, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\, rest\,\sigma$ is derivable in Path $\Im^I$ in at most N steps. Here, $b\sigma$ shares no variables with $rest\,\sigma$ and $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists) b\sigma$. By induction hypothesis, there is an executional deduction of $(\exists)\, rest\,\sigma$ in $\Im^I$ whose execution path is $\mathbf{D}_2 \ldots \mathbf{D}_n$. Using inference rule 3 in $\Im^I$, this can be extended by one sequent to an executional deduction of $(\exists)\,(b \otimes rest)$. The execution path is extended to $\mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n$. $\Box$

Combining Theorems C.2 and C.3, we get the following corollary, which is the main result of this appendix.

Corollary C.4 (Executional Soundness and Completeness of $\Im^I$)
If ψ is a serial goal, then the following two statements are equivalent:

1. $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \models (\exists)\, \psi$.
2. *There is an executional deduction of $(\exists)\, \psi$ in $\Im^I$ whose execution path is $\mathbf{D}_1, \mathbf{D}_2 \ldots, \mathbf{D}_n$.*

D Appendix: Soundness of $\Im^{\diamond}$

In Section 8.3, we developed $\Im^{\diamond}$, an inference system for specifying and executing hypothetical as well as committed transactions. This appendix proves that $\Im^{\diamond}$ is sound. Unlike completeness, soundness holds for arbitrary transaction formulas and transaction bases, and does not require the serial-Horn conditions of Sections 6.1 and 8.3.1.

For convenient reference, we reproduce below the axioms and inference rules of system $\Im^{\diamond}$, developed in Sections 6.3, 6.4, and 8.3. To simplify references during proofs, we have re-numbered the rules. Thus, rules 1 and $1'$ are now called $1f$ and $1r$, respectively. The adornments f and r indicate whether rules correspond to forward or reverse execution during top-down inference. Recall that the inference system assumes a data oracle, $\mathcal{O}^d$, and a state transition oracle, $\mathcal{O}^t$. As described in Sections 3 and 5, the data oracle specifies the semantics of database states, and the transition oracle specifies the semantics of elementary state transitions. The rest of this appendix assumes that these two oracles are given.

Definition D.1 (Inference System $\Im^{\diamond}$) If $\mathbf{P}$ is a transaction base, then $\Im^{\diamond}$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.

Axioms:

$$\mathbf{P}, \mathbf{D}\text{---} \vdash (\,)$$

$$\mathbf{P}, \text{---}\mathbf{D} \vdash (\,)$$

Inference Rules: In the rules below, σ is a substitution, a and b are atomic formulas, and ϕ, $rest$ and $test$ are $\Diamond$-serial goals.

Applying transaction definitions: Suppose $a \leftarrow \phi$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with $b \otimes rest$. If a and b unify with mgu σ, then:

$$1f. \quad \frac{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,(\phi \otimes rest)\sigma}{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, b \otimes rest}$$

$$1r. \quad \frac{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\,(rest \otimes \phi)\sigma}{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\, rest \otimes b}$$

Querying the database: If $b\sigma$ and $rest\,\sigma$ share no variables, and $\mathcal{O}^d(\mathbf{D}) \models^c (\exists)\, b\sigma$, then

$$2f. \quad \frac{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, b \otimes rest}$$

$$2r. \quad \frac{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\, rest \otimes b}$$

Performing elementary transitions: If $b\sigma$ and $rest\,\sigma$ share no variables, then:

$$3f. \quad \frac{\mathbf{P}, \mathbf{D}_2\text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}_1\text{---} \vdash (\exists)\, b \otimes rest} \quad \text{if} \quad \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)\, b\sigma$$

$$3r. \quad \frac{\mathbf{P}, \text{---}\mathbf{D}_2 \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \text{---}\mathbf{D}_1 \vdash (\exists)\, rest \otimes b} \quad \text{if} \quad \mathcal{O}^t(\mathbf{D}_2, \mathbf{D}_1) \models^c (\exists)\, b\sigma$$

Introduction of modalities: If $test\,\sigma$ and $rest\,\sigma$ share no variables, then

$$4f. \quad \frac{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists) test\,\sigma \qquad \mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,(\Diamond test) \otimes rest}$$

$$4r. \quad \frac{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\, rest\,\sigma \qquad \mathbf{P}, \text{---}\mathbf{D} \vdash (\exists) test\,\sigma}{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\, rest \otimes (test\Diamond)}$$

Mixed introduction of modalities: If $test\,\sigma$ and $rest\,\sigma$ share no variables, then

$$5f. \quad \frac{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists) test\,\sigma \qquad \mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\, rest\,\sigma}{\mathbf{P}, \mathbf{D}\text{---} \vdash (\exists)\,(test\Diamond) \otimes rest}$$

$$5r. \quad \frac{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\, rest\,\sigma \qquad \mathbf{P}, \mathbf{D}\text{---} \vdash (\exists) test\,\sigma}{\mathbf{P}, \text{---}\mathbf{D} \vdash (\exists)\, rest \otimes (\Diamond test)}$$

□

Theorem D.2 (Soundness of $\Im^{\Diamond}$) *Let ϕ be a transaction formula.*

$$\textit{If} \quad \mathbf{P}, \mathbf{D}\text{---} \vdash \phi \quad \textit{then} \quad \mathbf{P}, \mathbf{D}\text{---} \models \phi$$

$$\textit{If} \quad \mathbf{P}, \text{---}\mathbf{D} \vdash \phi \quad \textit{then} \quad \mathbf{P}, \text{---}\mathbf{D} \models \phi$$

To prove Theorem D.2, it is enough to show that the axioms and inference rules of system $\Im^{\diamond}$ are sound. Lemma A.3 shows that the axioms are sound. Lemma A.4 shows that inference rules $1f$, $2f$ and $3f$ are sound. Lemma D.3 below shows that inference rules $4f$ and $5f$ are sound. Similar results (with similar proofs) hold for inference rules $1r$ to $5r$.

Lemma D.3 *Suppose rest and test are two transaction formulas, and* σ *is a substitution such that* $test\,\sigma$ *and* $rest\,\sigma$ *share no variables.*

4. *If* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}'_2 ... \mathbf{D}'_m \models (\exists)\, test\,\sigma$ *and* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, rest\,\sigma$
   then $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, (\diamond test) \otimes rest.$

5. *If* $\mathbf{P}, \mathbf{D}'_m ... \mathbf{D}'_2, \mathbf{D}_1 \models (\exists)\, test\,\sigma$ *and* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, rest\,\sigma$
   then $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, (test\diamond) \otimes rest.$

Proof: To prove item *4*, suppose that the *if* part is true. Then, for every model, $\mathbf{M}$, of P,

$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}'_2 \ ...\ \mathbf{D}'_m \rangle \models (\exists)\, test\,\sigma$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models (\exists)\, rest\,\sigma$	by Definition 5.6;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}'_2 \ ...\ \mathbf{D}'_m \rangle \models_\nu test\,\sigma$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\nu rest\,\sigma$	for some variable assignment, ν, which exists since $test\,\sigma$ and $rest\,\sigma$ share no variables;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}'_2 \ ...\ \mathbf{D}'_m \rangle \models_\mu test$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\mu rest$	where $\mu(X) = \nu(X\sigma)$ for all variables, X;
$\mathbf{M}, \langle \mathbf{D}_1 \rangle \models_\mu \diamond test$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\mu rest$	by the definition of $\diamond$;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\mu (\diamond test) \otimes rest$	by Definition 5.2;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models (\exists)\, (\diamond test) \otimes rest$	by Definition 5.2.

Since this is true for every model, $\mathbf{M}$, of $\mathbf{P}$, it follows that $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, (\diamond test) \otimes rest.$

To prove item *5*, suppose that the *if* part is true. Then, for every model, $\mathbf{M}$, of P,

$\mathbf{M}, \langle \mathbf{D}'_m \ ...\ \mathbf{D}'_2\ \mathbf{D}_1 \rangle \models (\exists)\, test\,\sigma$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models (\exists)\, rest\,\sigma$	by Definition 5.6;
$\mathbf{M}, \langle \mathbf{D}'_m \ ...\ \mathbf{D}'_2, \mathbf{D}_1 \rangle \models_\nu test\,\sigma$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\nu rest\,\sigma$	for some variable assignment, ν, which exists since $test\,\sigma$ and $rest\,\sigma$ share no variables;
$\mathbf{M}, \langle \mathbf{D}'_m \ ...\ \mathbf{D}'_2, \mathbf{D}_1 \rangle \models_\mu test$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\mu rest$	where $\mu(X) = \nu(X\sigma)$ for all variables, X;
$\mathbf{M}, \langle \mathbf{D}_1 \rangle \models_\mu test\diamond$ *and* $\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\mu rest$	by the definition of $\diamond$;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models_\mu (test\diamond) \otimes rest$	by Definition 5.2;
$\mathbf{M}, \langle \mathbf{D}_1\ \mathbf{D}_2 \ ...\ \mathbf{D}_n \rangle \models (\exists)\, (test\diamond) \otimes rest$	by Definition 5.2.

Since this is true for every model, $\mathbf{M}$, of $\mathbf{P}$, it follows that $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\, (test\diamond) \otimes rest.$ $\square$

E Appendix: Completeness of $\Im^{\diamond}$

This appendix proves that inference system $\Im^{\diamond}$ developed in Section 8.3 is complete. Recall that $\Im^{\diamond}$ is an inference system for hypothetical and retrospective transactions as well as committed transactions. The axioms and inference rules of $\Im^{\diamond}$ are reproduced in Appendix D. We shall use the numbering system given there to identify inference rules, *i.e.*, $1f$, $1r$, $2f$, $2r$, etc.

As in Appendix B, our strategy is to first prove completeness of ground inference using a canonical-model construction. We then prove completeness of general inference using a Herbrand theorem and a lifting lemma. As in the other appendices, we assume that a Transition Oracle, $\mathcal{O}^t$, and a Data Oracle, $\mathcal{O}^d$, are given. We also assume that all transactions are $\diamond$-serial conjunctions, that the transaction base is a set of $\diamond$-serial Horn rules, and that the other conditions of Definition 6.2 are satisfied.

E.1 Ground Inference

This section develops a ground version of inference system $\Im^{\diamond}$ called *Ground* $\Im^{\diamond}$. As in the case of Ground $\Im^{I}$, this inference system keeps track of transaction execution paths and is an important auxiliary construction used in the proof of completeness. Systems $\Im^{\diamond}$ and Ground $\Im^{\diamond}$ use the same data oracle, $\mathcal{O}^d$, and the same transition oracle, $\mathcal{O}^t$. Both systems thus have the same set of database state identifiers, and the same semantics for states and transitions. In addition to the $\diamond$-serial Horn conditions, this subsection assumes that the transaction base $\mathbf{P}$ and all transaction invocations are ground.

Ground $\Im^{\diamond}$ uses two kinds of sequents. They are based on two turnstyles, $\vdash_f$ and $\vdash_r$, which correspond to the two kinds of sequent in inference system $\Im^{\diamond}$. Specifically, the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \phi$ corresponds to $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash \phi$, while the sequent $\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_1 \vdash_r \phi$ corresponds to $\mathbf{P}, \text{---}\, \mathbf{D}_1 \vdash \phi$. From the perspective of top-down (or goal-directed) inference, $\vdash_f$ corresponds to the forward (or normal) execution of transactions, while $\vdash_r$ corresponds to reverse execution.

Definition E.1 (Inference in Ground $\Im^{\diamond}$*)* Let $\mathbf{P}$ be a (possibly infinite) ground $\diamond$-serial Horn transaction base. Then Ground $\Im^{\diamond}$ is the following system of axioms and inference rules, where $\mathbf{D}$ and $\mathbf{D}_i$ are any database state identifiers.[34]

Axioms:

$$\mathbf{P}, \mathbf{D} \vdash_f (\,)$$

$$\mathbf{P}, \mathbf{D} \vdash_r (\,)$$

Inference Rules: In the rules below, b is a ground atomic formula, and ϕ, *rest* and *test* are ground $\diamond$-serial goals.

Applying transaction definitions: If $b \leftarrow \phi$ is a rule in $\mathbf{P}$, then

$$1f. \quad \frac{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \phi \otimes rest}{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f b \otimes rest}$$

$$1r. \quad \frac{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_1 \vdash_r rest \otimes \phi}{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_1 \vdash_r rest \otimes b}$$

[34]Note that the databases need not be ground; that is, $\mathcal{O}^d(\mathbf{D})$ may contain variables.

Querying the database: If $\mathcal{O}^d(\mathbf{D}_1) \models^c b$, then

$$2f.\quad \frac{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f rest}{\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f b \otimes rest}$$

$$2r.\quad \frac{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_1 \vdash_r rest}{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_1 \vdash_r rest \otimes b}$$

Performing elementary transitions:

$$3f.\quad \frac{\mathbf{P}, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f rest}{\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f b \otimes rest} \quad \text{if } \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c b$$

$$3r.\quad \frac{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_2 \vdash_r rest}{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_2, \mathbf{D}_1 \vdash_r rest \otimes b} \quad \text{if } \mathcal{O}^t(\mathbf{D}_2, \mathbf{D}_1) \models^c b$$

Introduction of modalities:

$$4f.\quad \frac{\mathbf{P}, \mathbf{D}_1, \mathbf{D}'_2 ... \mathbf{D}'_m \vdash_f test \qquad \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f rest}{\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f (\diamond test) \otimes rest}$$

$$4r.\quad \frac{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_2, \mathbf{D}_1 \vdash_r rest \qquad \mathbf{P}, \mathbf{D}'_m ... \mathbf{D}'_2, \mathbf{D}_1 \vdash_r test}{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_2, \mathbf{D}_1 \vdash_r rest \otimes (test \diamond)}$$

Mixed introduction of modalities:

$$5f.\quad \frac{\mathbf{P}, \mathbf{D}'_m ... \mathbf{D}'_2, \mathbf{D}_1 \vdash_r test \qquad \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f rest}{\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f (test \diamond) \otimes rest}$$

$$5r.\quad \frac{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_2, \mathbf{D}_1 \vdash_r rest \qquad \mathbf{P}, \mathbf{D}_1, \mathbf{D}'_2 ... \mathbf{D}'_m \vdash_f test}{\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_2, \mathbf{D}_1 \vdash_r rest \otimes (\diamond test)}$$

□

The soundness of Ground $\Im^{\diamond}$ follows immediately from the proof of soundness of $\Im^{\diamond}$ (Theorem D.2). We thus have the following result.

Theorem E.2 (Soundness of Ground $\Im^{\diamond}$) *Let ψ be a ground $\diamond$-serial goal.*

$$\textit{If} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x \psi \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models \psi$$

where $x \in \{f, r\}$.

Below, we give some properties of Ground $\Im^{\diamond}$. The properties in Lemma E.3 are special cases of the inference rules with $rest = ()$. Lemma E.4 is a bit more difficult, and is central to our proof of completeness of the ground inference.

Lemma E.3 (Basic Properties) *Ground $\Im^{\diamond}$ has the following properties, where b is a ground atomic formula, ϕ is a $\diamond$-serial goal, and $x \in \{f, r\}$:*

- *If* $b \leftarrow \phi$ *is a rule in* $\mathbf{P}$ *and if* $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x \phi$ *then* $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x b$
- *If* $\mathcal{O}^d(\mathbf{D}) \models^c b$ *then* $\mathbf{P}, \mathbf{D} \vdash_x b$
- *If* $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c b$ *then* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \vdash_x b$

- *If* $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \phi$ *then* $\mathbf{P}, \mathbf{D}_1 \vdash_x \Diamond\phi$
- *If* $\mathbf{P}, \mathbf{D}_n ... \mathbf{D}_1 \vdash_r \phi$ *then* $\mathbf{P}, \mathbf{D}_1 \vdash_x \phi\Diamond$

Lemma E.4 *Ground* $\Im^{\Diamond}$ *has the following property, where* α *and* β *are ground* $\Diamond$*-serial goals, and* $x \in \{r, f\}$,

$$\textit{If} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_x \alpha \quad \textit{and} \quad \mathbf{P}, \mathbf{D}_i ... \mathbf{D}_n \vdash_x \beta \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x \alpha \otimes \beta$$

Proof: We prove the lemma for $\vdash_f$; the proof for $\vdash_r$ is similar. The proof is similar to, but more complex than the proof of Lemma B.4. We transform a derivation of $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_f \alpha$ into a derivation of $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \alpha \otimes \beta$. The proof is by induction on the length of the derivation of $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_f \alpha$. (The proof for $\vdash_r$ is by induction on the length of the derivation of $\mathbf{P}, \mathbf{D}_i ... \mathbf{D}_n \vdash_r \beta$.)

Basis: Suppose that the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_f \alpha$ can be derived in 1 step. It is therefore an axiom, so $i = 1$ and $\alpha = (\,)$. Thus, $\alpha \otimes \beta = \beta$, so the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \alpha \otimes \beta$ is identical to the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \beta$, which is identical to the sequent $\mathbf{P}, \mathbf{D}_i ... \mathbf{D}_n \vdash_f \beta$, which is given to be true.

Induction: Suppose the lemma is true for derivations of length at most N, and suppose the expression $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_f \alpha$ can be derived in $N + 1$ steps. We must show that the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \alpha \otimes \beta$ is derivable. There are five cases, depending on the last inference rule used in the derivation of $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_f \alpha$.

- For inference rules $1f$, $2f$, and $3f$, the proof is essentially identical to the proof of Lemma B.4.

- If inference rule $4f$ is the last rule used in the derivation, then α has the form $(\Diamond test) \otimes rest$. Moreover, $\mathbf{P}, \mathbf{D}_1, \mathbf{D}'_2 ... \mathbf{D}'_m \vdash_f test$ for some databases $\mathbf{D}'_2 ... \mathbf{D}'_m$, and the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_f rest$ can be derived in at most N steps. Thus, by induction hypothesis, the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f rest \otimes \beta$ is derivable. Hence, by rule $4f$, $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f (\Diamond test) \otimes rest \otimes \beta$, which is synonymous with $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \alpha \otimes \beta$.

- If inference rule $5f$ is the last rule used in the derivation, then α has the form $(test\Diamond) \otimes rest$. Moreover, $\mathbf{P}, \mathbf{D}'_m ... \mathbf{D}'_2, \mathbf{D}_1 \vdash_r test$ for some databases $\mathbf{D}'_m ... \mathbf{D}'_2$, and the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_f rest$ can be derived in at most N steps. Thus, by induction hypothesis, the sequent $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f rest \otimes \beta$ is derivable. Hence, by rule $5f$, $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f (test\Diamond) \otimes rest \otimes \beta$, which is synonymous with $\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \alpha \otimes \beta$. $\Box$

E.2 Completeness of Ground $\Im^{\Diamond}$

To prove the completeness of Ground $\Im^{\Diamond}$, we construct a canonical Herbrand model of the transaction base $\mathbf{P}$. This model, constructed proof theoretically, provides the necessary link between the proof theory and model theory. Again, in addition to the $\Diamond$-serial Horn conditions, this subsection assumes that the transaction base $\mathbf{P}$ and all transaction invocations are ground.

Given a language for $\mathcal{TR}$, we define the Herbrand universe and classical Herbrand interpretations in a standard way, as in Appendix B.2. The symbol $\mathcal{HB}$ shall denote the set of all ground atomic formulas, *i.e.*, the Herbrand base. There is a 1–1 correspondence between Herbrand interpretations and subsets of $\mathcal{HB}$, so we shall often treat them as the same thing. Recall from Appendix B.2

that Herbrand path structures are determined by the mapping I_{path} alone, since the domain and the interpretation of function symbols are fixed.

Definition E.5 (Canonical Model) The canonical model of a transaction base $\mathbf{P}$ is the Herbrand path structure $\mathbf{M_P} = \langle I_{path} \rangle$, where

$$I_{path}(\langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle) \quad = \quad \{ \ b \in \mathcal{HB} \ \mid \ \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f b \quad \textit{and} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_r b \ \} \qquad \Box$$

We first observe that $\mathbf{M_P}$ is a legitimate path structure. This follows immediately from Definition 5.1 and the following lemma.

Lemma E.6 *Let* $\mathbf{M_P} = \langle I_{path} \rangle$ *be the canonical model of* $\mathbf{P}$*. Then:*

(i) $I_{path}(\langle \mathbf{D} \rangle) \models^c \phi$ *for every closed first-order formula* ϕ *in* $\mathcal{O}^d(\mathbf{D})$
(so $\mathbf{M_P}, \langle \mathbf{D} \rangle \models \phi$*)*

(ii) $I_{path}(\langle \mathbf{D}_1 \ \mathbf{D}_2 \rangle) \models^c b$ *for every* b *in* $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$
(so $\mathbf{M_P}, \langle \mathbf{D}_1 \ \mathbf{D}_2 \rangle \models b$*)*

Proof: The same as for Lemma B.6. $\Box$

The following lemma states a basic result about canonical models; its proof is analogous to that of Lemma B.7. Theorem E.8 partially generalizes this result from atoms to $\diamond$-serial goals. Corollary E.10, below, is a full generalization of this result.

Lemma E.7 *If* b *is a ground atomic formula, then*

$$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models b \quad \textit{iff} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f b \quad \textit{and} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_r b$$

Theorem E.8 *Let* ϕ *be a ground* $\diamond$*-serial goal.*

$$\textit{If} \quad \mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models \phi \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x \phi$$

where $x \in \{f, r\}$*.*

Proof: The proof is by induction on the number of logical operators in ϕ. In the base case, ϕ is a singleton $\diamond$-serial goal, that is, either a ground atomic formula or the empty clause (*i.e.*, `state`). In the former case, the result follows by Lemma E.7. In the later case, if the expression $\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models (\,)$ is true, then $n = 1$, since the empty clause is true only on states. But, the sequent $\mathbf{P}, \mathbf{D}_1 \vdash_x (\,)$ is trivially true, since it is an axiom.

For the inductive case, assume the theorem is true for formulas with at most N logical operators, and suppose that ϕ has $N+1$ logical operators. There are three cases, one for each type of operator.

- If ϕ has the form $\phi_1 \otimes \phi_2$, then ϕ_1 and ϕ_2 have at most N logical operators each. Thus,

$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_n \rangle \models \phi_1 \otimes \phi_2$	given;
$\mathbf{M_P}, \langle \mathbf{D}_1 \ ... \ \mathbf{D}_i \rangle \models \phi_1 \quad \textit{and} \quad \mathbf{M_P}, \langle \mathbf{D}_i \ ... \ \mathbf{D}_n \rangle \models \phi_2$	for some i, by Definition 5.2;
$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_i \vdash_x \phi_1 \quad \textit{and} \quad \mathbf{P}, \mathbf{D}_i ... \mathbf{D}_n \vdash_x \phi_2$	by induction hypothesis;
$\mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x \phi_1 \otimes \phi_2$	by Lemma E.4.

- If ϕ has the form $\diamond\psi$, then ψ has at most N logical operators, and $n = 1$, since hypothetical formulas are true only on states. Thus,

$$\begin{array}{ll} \mathbf{M_P}, \langle \mathbf{D}_1 \rangle \models \diamond\psi & \text{given;} \\ \mathbf{M_P}, \langle \mathbf{D}_1\ \mathbf{D}'_2\ ...\ \mathbf{D}'_m \rangle \models \psi & \text{for some databases } \mathbf{D}'_2 ... \mathbf{D}'_m \text{, by definition;} \\ \mathbf{P}, \mathbf{D}_1, \mathbf{D}'_2 ... \mathbf{D}'_m \vdash_f \psi & \text{by induction hypothesis;} \\ \mathbf{P}, \mathbf{D}_1 \vdash_x \diamond\psi & \text{by Lemma E.3.} \end{array}$$

- If ϕ has the form $\psi\diamond$, then ψ has at most N logical operators, and $n = 1$, since retrospective formulas are true only on states. Thus,

$$\begin{array}{ll} \mathbf{M_P}, \langle \mathbf{D}_1 \rangle \models \psi\diamond & \text{given} \\ \mathbf{M_P}, \langle \mathbf{D}'_m\ ...\ \mathbf{D}'_2, \mathbf{D}_1 \rangle \models \psi & \text{for some databases } \mathbf{D}'_m ... \mathbf{D}'_2 \text{, by definition;} \\ \mathbf{P}, \mathbf{D}'_m ... \mathbf{D}'_2, \mathbf{D}_1 \vdash_r \psi & \text{by induction hypothesis;} \\ \mathbf{P}, \mathbf{D}_1 \vdash_x \diamond\psi & \text{by Lemma E.3.} \end{array}$$

□

Theorem E.9 $\mathbf{M_P}$ *is a model of* $\mathbf{P}$.

Proof: Let $b \leftarrow \phi$ be any rule in $\mathbf{P}$. Then, for any path $\langle \mathbf{D}_1\ ...\ \mathbf{D}_n \rangle$,

$$\begin{array}{lll} \textit{if} & \mathbf{M_P}, \langle \mathbf{D}_1\ ...\ \mathbf{D}_n \rangle \models \phi & \\ \textit{then} & \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x \phi & \text{for } x \in \{f, r\} \text{, by Theorem E.8,} \\ & \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x b & \text{by Lemma E.3,} \\ & \mathbf{M_P}, \langle \mathbf{D}_1\ ...\ \mathbf{D}_n \rangle \models b & \text{by Lemma E.7.} \end{array}$$

$\mathbf{M_P}$ thus satisfies the rule $b \leftarrow \phi$ on every path. It is therefore a model of every rule in $\mathbf{P}$ and, thus, of $\mathbf{P}$. □

To prove the completeness of $\Im^{\diamond}$, we do not need the completeness of Ground $\Im^{\diamond}$ per se. Instead, we need Theorems E.8 and E.9. However, the completeness of Ground $\Im^{\diamond}$ has now become an easy corollary.

Corollary E.10 (Completeness of Ground $\Im^{\diamond}$**)** *For any ground* $\diamond$*-serial goal,* ϕ*,*

$$\textit{If} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \models \phi \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 ... \mathbf{D}_n \vdash_x \phi$$

where $x \in \{f, r\}$.

Proof: The same as for Corollary B.10. □

Although the proof of completeness in Ground $\Im^{\diamond}$ is now finished, we include two more important corollaries. The first generalizes Lemma E.7 from atoms to $\diamond$-serial conjunctions. It shows that inference in Ground $\Im^{\diamond}$ is equivalent to satisfaction in the canonical model. The second shows that in Ground $\Im^{\diamond}$, forward and backward derivations yield the same set of sequents.

Corollary E.11 *If ϕ is a ground $\diamond$-serial goal, and $x \in \{f, r\}$, then*

$$\mathbf{M_P}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models \phi \quad \textit{iff} \quad \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_x \phi$$

Proof: The same as for Corollary B.11. □

Corollary E.12 *For any ground $\diamond$-serial goal, ϕ,*

$$\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_f \phi \quad \textit{iff} \quad \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_r \phi$$

Proof: Follows immediately from Corollary E.11:

$$\mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_f \phi \quad \textit{iff} \quad \mathbf{M_P}, \langle \mathbf{D}_1 \ldots \mathbf{D}_n \rangle \models \phi \quad \textit{iff} \quad \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_r \phi \qquad \square$$

E.3 Completeness of $\Im^{\diamond}$

Having proved the completeness of Ground $\Im^{\diamond}$, we now prove the completeness of $\Im^{\diamond}$ itself. As in Appendix B.3, we first prove a result analogous to the Herbrand Theorem of classical logic, which lets us reduce general entailment to ground inference. We then prove a lifting lemma, to "lift" ground inference up to general inference. In this subsection, we again assume the $\diamond$-serial Horn conditions, but not groundness. Thus, the transaction base $\mathbf{P}$ and transaction invocations may be non-ground. The ground instantiation of $\mathbf{P}$ is denoted $\mathbf{P}^*$. We shall need to distinguish between the sequents of inference systems $\Im^{\diamond}$ and Ground $\Im^{\diamond}$. In general, sequents of the form $\mathbf{P}, \mathbf{D}\text{---} \vdash \phi$ belong to $\Im^{\diamond}$, while sequents of the form $\mathbf{P}^*, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_x \phi$ belong to Ground $\Im^{\diamond}$, where $x \in \{f, r\}$.

Lemma E.13 $\mathbf{M}_{\mathbf{P}^*}$ *is a model of* $\mathbf{P}$.

Proof: The same as for Lemma B.12. □

Corollary E.14 (Herbrand's Theorem for $\Im^{\diamond}$) *Let ϕ be a $\diamond$-serial goal, and $x \in \{f, r\}$.*

$$\textit{If} \quad \mathbf{P}, \mathbf{D}_1 \ldots \mathbf{D}_n \models (\exists)\, \phi \quad \textit{then} \quad \mathbf{P}^*, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_x \phi'$$

for some ground instance, ϕ', of ϕ.

Proof: The same as for Corollary B.13. □

Lemma E.15 (Lifting Lemma for $\Im^{\diamond}$) *Let ψ be a $\diamond$-serial goal, and let ψ' be one of its ground instantiations.*

$$\textit{If} \quad \mathbf{P}^*, \mathbf{D}_1 \ldots \mathbf{D}_n \vdash_f \psi' \quad \textit{then} \quad \mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, \psi$$

$$\textit{If} \quad \mathbf{P}^*, \mathbf{D}_n \ldots \mathbf{D}_1 \vdash_r \psi' \quad \textit{then} \quad \mathbf{P}, \text{---} \mathbf{D}_1 \vdash (\exists)\, \psi$$

Proof: The proof is by induction on the length of ground derivations.

Basis: In this case, both sequent are axioms, so they are both trivially true.

Induction: Suppose the lemma is true for all ground derivations having at most N steps. Now consider derivations of $N+1$ steps. There are ten cases, depending on the last inference rule used in the derivation. We shall consider only the five f-rules, since proofs for the five r-rules are similar. We therefore assume that the sequent $\mathbf{P}^*, \mathbf{D}_1 ... \mathbf{D}_n \vdash_f \psi'$ is derivable in Ground $\Im^{\diamond}$ in $N+1$ steps.

• When the last inference rule used in the ground derivation is $1f$, $2f$, or $3f$, the proof is the same as in Lemma B.14.

• Suppose inference rule $4f$ is the last rule used in the ground derivation. Then ψ has the form $(\diamond test) \otimes rest$, and ψ' has the form $(\diamond test\,\sigma) \otimes rest\,\sigma$ for some ground substitution σ. Moreover, the following two sequents are derivable in Ground $\Im^{\diamond}$ in at most N steps:

$$\mathbf{P}^*, \mathbf{D}_1, \mathbf{D}'_2 ... \mathbf{D}'_m \vdash_f test\,\sigma \qquad\qquad \mathbf{P}^*, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f rest\,\sigma$$

for some databases $\mathbf{D}'_2 ... \mathbf{D}'_m$. Thus, by induction hypothesis, the following two sequents are derivable in $\Im^{\diamond}$:

$$\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, test\,\sigma \qquad\qquad \mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, rest\,\sigma$$

since $test\,\sigma$ and $rest\,\sigma$ are (trivially) ground instantiations of $test\,\sigma$ and $rest\,\sigma$, respectively. Being ground, $test\,\sigma$ and $rest\,\sigma$ share no variables. Thus, the sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(\diamond test) \otimes rest$ is derivable in $\Im^{\diamond}$, by rule $4f$.

• Suppose inference rule $5f$ is the last one used in the ground derivation. Then ψ has the form $(test\diamond) \otimes rest$, and ψ' has the form $(test\,\sigma\,\diamond) \otimes rest\,\sigma$ for some ground substitution, σ. Moreover, the following two sequents are derivable in Ground $\Im^{\diamond}$ in at most N steps:

$$\mathbf{P}^*, \mathbf{D}'_m ... \mathbf{D}'_2, \mathbf{D}'_1 \vdash_r test\,\sigma \qquad\qquad \mathbf{P}^*, \mathbf{D}_1, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f rest\,\sigma$$

for some databases $\mathbf{D}'_m ... \mathbf{D}'_2$. Thus, by induction hypothesis, the following two sequents are derivable in $\Im^{\diamond}$:

$$\mathbf{P}, \text{---}\, \mathbf{D}_1 \vdash (\exists)\, test\,\sigma \qquad\qquad \mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\, rest\,\sigma$$

since $test\,\sigma$ and $rest\,\sigma$ are (trivially) ground instantiations of $test\,\sigma$ and $rest\,\sigma$, respectively. Being ground, $test\,\sigma$ and $rest\,\sigma$ share no variables. Thus, the sequent $\mathbf{P}, \mathbf{D}_1 \text{---} \vdash (\exists)\,(test\diamond) \otimes rest$ is derivable in $\Im^{\diamond}$, by rule $5f$. □

Theorem E.16 (Completeness of $\Im^{\diamond}$) *Let ϕ be a $\diamond$-serial goal.*

$$\textit{If} \quad \mathbf{P}, \mathbf{D} \text{---} \models (\exists)\,\phi \quad \textit{then} \quad \mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,\phi$$

$$\textit{If} \quad \mathbf{P}, \text{---}\,\mathbf{D} \models (\exists)\,\phi \quad \textit{then} \quad \mathbf{P}, \text{---}\,\mathbf{D} \vdash (\exists)\,\phi$$

Proof: As in Theorem B.15. For instance, if $\mathbf{P}, \mathbf{D} \text{---} \models (\exists)\,\phi$, then $\mathbf{P}, \mathbf{D}, \mathbf{D}_2 ... \mathbf{D}_n \models (\exists)\,\phi$ for some databases $\mathbf{D}_2 \ldots \mathbf{D}_n$, by Definition 5.6. Thus $\mathbf{P}^*, \mathbf{D}, \mathbf{D}_2 ... \mathbf{D}_n \vdash_f \phi'$, where ϕ' is some ground instance of ϕ, by Corollary E.14. Hence $\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,\phi$, by Lemma E.15. □

Although this completes our proof of completeness, we include the following important corollary. It shows that inference in $\Im^{\diamond}$ is equivalent to satisfaction in the canonical model.

Corollary E.17 *Let ϕ be a $\diamond$-serial goal. Then:*

$$\mathbf{P}, \mathbf{D} \text{---} \vdash (\exists)\,\phi \quad \textit{iff} \quad \mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}\ \mathbf{D}_2\ ...\ \mathbf{D}_n \rangle \models (\exists)\,\phi \quad \textit{for some databases } \mathbf{D}_2 ... \mathbf{D}_n.$$

$$\mathbf{P}, \text{---}\,\mathbf{D} \vdash (\exists)\,\phi \quad \textit{iff} \quad \mathbf{M}_{\mathbf{P}^*}, \langle \mathbf{D}_n\ ...\ \mathbf{D}_2, \mathbf{D} \rangle \models (\exists)\,\phi \quad \textit{for some databases } \mathbf{D}_n ... \mathbf{D}_2.$$

Proof: The same as for Corollary B.16. □

F Appendix: Executional Deduction in $\Im^\diamond$

Section 8.3.5 developed the idea of executional deduction for hypothetical and retrospective inference. Executional deductions trace out the execution path of a transaction during inference. The proofs of soundness and completeness given in Appendices D and E are for general deductions in $\Im^\diamond$. The present appendix adapts these proofs to executional deductions in $\Im^\diamond$. As in Appendix E, we assume that all transactions are $\diamond$-serial goals, that the transaction base $\mathbf{P}$ is a set of $\diamond$-serial Horn rules, and that the other conditions of Definition 6.2 are satisfied.

As in Appendix C, the main idea is to introduce an intermediate inference system called *Path* $\Im^\diamond$. Path $\Im^\diamond$ is identical to $\Im^\diamond$ except that it records the execution path of a transaction. Sequents in Path $\Im^\diamond$ have the form $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash \psi$, which intuitively means that $\mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n$ is an (inferred) execution path for transaction ψ. Path $\Im^I$ is similar to inference system Ground $\Im^\diamond$ developed in Appendix E.1, except that the transaction base $\mathbf{P}$ and the transaction ψ need not be ground. Path $\Im^\diamond$ is a trivial extension of $\Im^\diamond$, and there is a one-to-one correspondence between derivations in $\Im^\diamond$ and derivations in Path $\Im^\diamond$. The soundness and completeness proofs for $\Im^\diamond$ are thus easily adapted to Path $\Im^\diamond$, to give Theorem F.1 below. The only remaining step is to show that any general deduction in Path $\Im^\diamond$ can be associated with an executional deduction in $\Im^\diamond$, and vice-versa. This is done in Theorem F.2 below. The proofs are a straightforward extension of those in Appendix C, so we simply state the main results.

Theorem F.1 (Soundness and Completeness of Path $\Im^\diamond$)
If ψ is a $\diamond$-serial goal, then the following two statements are equivalent:

1. $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \models (\exists)\, \psi$
2. *The sequent* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\, \psi$ *is derivable in Path* $\Im^\diamond$.

Theorem F.2 *If ψ is a $\diamond$-serial goal, then the following two statements are equivalent:*

1. *The sequent* $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \vdash (\exists)\, \psi$ *is derivable in Path* $\Im^\diamond$.
2. *There is an executional deduction of* $(\exists)\, \psi$ *in* $\Im^\diamond$ *whose execution path is* $\mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n$.

Corollary F.3 (Executional Soundness and Completeness of $\Im^\diamond$)
If ψ is a $\diamond$-serial goal, then the following two statements are equivalent:

1. $\mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_n \models (\exists)\, \psi$.
2. *There is an executional deduction of* $(\exists)\, \psi$ *in* $\Im^\diamond$ *whose execution path is* $\mathbf{D}_1, \mathbf{D}_2 \ldots, \mathbf{D}_n$.

Acknowledgments

The first author was supported in part by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) and by a Connaught Grant from the University of Toronto. The second author was supported in part by NSF grants CCR-9102159, IRI-9404629, and a grant from New York Science and Technology Foundation. Support of Computer Systems Research Institute of University of Toronto is also gratefully acknowledged.

Alberto Mendelzon provided us with many insights regarding updates of logic theories. Thanks to Ray Reiter for commenting on various aspects of this paper, especially on the issues related to the frame problem. Mariano Consens took a very close look at the first draft and many improvements are due to his comments. Comments by Greg Meredith on Pratt's Action Logic, and discussions with Gösta Grahne, Peter Revesz, Fangzhen Lin, Javier Pinto, Dimitris Lagouvardos, Jan Van den Bussche, Roel Wieringa, Jan Chomicki, and Ron van der Meyden are also gratefully acknowledged.

References

[1] S. Abiteboul. Updates, a new frontier. In *Intl. Conference on Extending Database Technology (EDBT)*, pages 1–18, 1988.

[2] S. Abiteboul and A.J. Bonner. Objects and views. In *ACM SIGMOD Conference on Management of Data*, pages 238–247, Denver, Colorado, May 29–31 1991. ACM.

[3] S. Abiteboul, P. Buneman, C. Delobel, R. Hull, P. Kanellakis, and V. Vianu. New Hope on Data Models and Types: Report of an NSF-INRIA workshop. *SIGMOD Record*, 19(4):41–48, December 1990.

[4] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. Technical Report 900, INRIA, Le Chesnay Cedex, France, 1988.

[5] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 240–250, New York, 1988. ACM.

[6] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of ACM*, 26(11):832–843, November 1983.

[7] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, July 1984.

[8] J.F. Allen and J.A. Koomen. Planning using a temporal world model. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 741–747, Karlsruhe, West Germany, 8–12 August 1983.

[9] F. Bancilhon. A logic-programming/Object-oriented cocktail. *SIGMOD Record*, 15(3):11–21, September 1986.

[10] T. Barsalou, N. Siambela, A.M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *ACM SIGMOD Conference on Management of Data*, pages 248–257, Denver, Colorado, May 29–31 1991. ACM.

[11] C. Beeri. New data models and languages—The challenge. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–15, San Diego, CA, June 1992. ACM.

[12] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases*. Addison Wesley, 1987.

[13] H.A. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[14] A.J. Bonner. A logic for hypothetical reasoning. In *National Conference on Artificial Intelligence (AAAI)*, St. Paul, Minn., August 1988. AAAI Press, Menlo Park, CA.

[15] A.J. Bonner. Hypothetical Datalog: Negation and linear recursion. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 286–300, Philadelphia, PA, March 29–31 1989. ACM.

[16] A.J. Bonner. Hypothetical Datalog: Complexity and expressibility. *Theoretical Computer Science*, 76:3–51, 1990.

[17] A.J. Bonner. *Hypothetical Reasoning in Deductive Databases.* PhD thesis, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA, October 1991. Published as Rutgers Technical Report DCS-TR-283.

[18] A.J. Bonner. Hypothetical Reasoning with Intuitionistic Logic. In R. Demolombe and T. Imielinski, editors, *Non-Standard Queries and Answers*, Studies on Logic and Computation, chapter 8, pages 187–219. Oxford University Press, October 1994.

[19] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming (ICLP)*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.

[20] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[21] A.J. Bonner and M. Kifer. A general logic of state change. Technical report, CSRI, University of Toronto, 1995. In preparation.

[22] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In A. Ohori C. Beeri and D.E. Shasha, editors, *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, Workshops in Computing, pages 309–337. Springer-Verlag, February 1994. Workshop held on Aug 30–Sept 1, 1993, New York City, NY.

[23] F. Bry. Intensional updates: Abduction via deduction. In *Intl. Conference on Logic Programming (ICLP)*, Jerusalem, Israel, June 1990.

[24] M.A. Casanova. *The Concurrency Control Problem for Database Systems*, volume 116 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1981.

[25] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Francisco, 1996. Release 1.2.

[26] A.K. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[27] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[28] W. Chen. Declarative specification and evaluation of database updates. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, volume 566 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, December 1991.

[29] P.R. Cohen and E.A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume III. Addison-Wesley Publishing Co., Reading, MA, 1986.

[30] C. de Maindreville and E. Simon. Non-deterministic queries and updates in deductive databases. In *Intl. Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, San Francisco, CA, 1988.

[31] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[32] R. Fagin, G.M. Kuper, J.D. Ullman, and M.Y. Vardi. Updating logical databases. In P.C. Kanellakis, editor, *Advances in Computing Research*, volume 3, pages 1–18. Plenum Press, 1986.

[33] R. Fagin, J.D. Ullman, and M.Y. Vardi. On the semantics of updates in databases. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 352–365, New York, 1983. ACM.

[34] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[35] J. Finger. *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford University, 1986.

[36] P.E. Friedland. *Knowledge-Based Experiment Design in Molecular Genetics*. PhD thesis, Computer Science Department, Stanford University, 1979. Report Number 79-771.

[37] D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In B. Banieqbal, B. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398, pages 409–448. Springer-Verlag, LNCS 398, 1989.

[38] D.M. Gabbay and U. Reyle. N-prolog: An extension of Prolog with hypothetical implications. I. *Journal of Logic Programming*, 1(4):319–355, 1984.

[39] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[40] M. Gelfond and V. Lifschitz. Representing actions in extended logic programming. In *Joint Intl. Conference and Symposium on Logic Programming (JICSLP)*, pages 559–573, November 1992.

[41] M.P. Georgeff and A.L. Lansky. Procedural knowledge. In *Proc. IEEE Special Issue on Knowledge Representation*, volume 74, pages 1384–1398, 1986.

[42] M.P. Georgeff, A.L. Lansky, and P. Bessiere. A procedural logic. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 516–523, San Francisco, CA, August 1985. Morgan Kaufmann.

[43] M.L. Ginsberg. Counterfactuals. *Artificial Intelligence*, 30:35–79, 1986.

[44] G. Grahne. Updates and counterfactuals. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 269–276, Boston, Mass., April 1991.

[45] G. Grahne and A.O. Mendelzon. Updates and subjunctive queries. Technical Report KRR-TR-91-4, CSRI, University of Toronto, July 1991.

[46] J.Y. Halpern, Z. Manna, and B.C. Moszkowski. A high-level semantics based on interval logic. In *International Conference on Automata, Languages and Programming (ICALP)*, pages 278–291. Springer-Verlag, 1983. Number 154 in *Lecture Notes in Computer Science*.

[47] J.Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. In *Intl. Symposium on Logic in Computer Science (LICS)*, pages 279–292, 1986.

[48] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *National Conference on Artificial Intelligence (AAAI)*, pages 328–333. AAAI Press, Menlo Park, CA, 1986.

[49] D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[50] D. Harel, D. Kozen, and R. Parikh. Process Logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, October 1982.

[51] Samuel Hung. Implementation of Transaction Logic. Master's thesis, Department of Computer Science, University of Toronto, Toronto ON, Canada, 1995. In preparation.

[52] P. Jacson. On the semantics of counterfactuals. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 1382–1387, San Francisco, CA, 1989. Morgan Kaufmann.

[53] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990. Second Edition.

[54] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 387–394, Boston, Mass., April 1991.

[55] M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, Lecture Notes in Computer Science, pages 187–212, Singapore, December 1995. Springer-Verlag. Keynote address at the 3d Intl. Conference on Deductive and Object-Oriented databases.

[56] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, pages 741–843, July 1995.

[57] M. Kifer and E.L. Lozinskii. RI: A logic for reasoning with inconsistency. In *4-th Intl Symp. on Logic in Computer Science*, pages 253–262, June 1989.

[58] M. Kifer and E.L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, November 1992.

[59] R.A. Kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, January 1992.

[60] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[61] R. Krishnamurthy and S. Naqvi. Non-deterministic choice in Datalog. In *Proceedings of the 3-d Intl. Conference on Data and Knowledge Bases*, pages 416–424. Morgan-Kaufmann Publ., 1988.

[62] D.K. Lewis. *Counterfactuals.* Blackwell, Oxford, 1973.

[63] V. Lifschitz. Pointwise circumscription. In *National Conference on Artificial Intelligence (AAAI)*, pages 406–410, Philadelphia, PA, August 1986. AAAI Press, Menlo Park, CA.

[64] V. Lifschitz. Formal theories of action. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 966–972, San Francisco, CA, August 1987. Morgan Kaufmann.

[65] V. Lifschitz. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, Timberline, OR, 1987. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, A. Tate (eds.), Morgan-Kaufmann, 1990, 523–530.

[66] F. Lin and R. Reiter. State constraints revisited. Unpublished Manuscript, Department of Computer Science, University of Toronto, 1993.

[67] J.W. Lloyd. *Foundations of Logic Programming (Second Edition).* Springer-Verlag, 1987.

[68] S. Manchanda. *A Dynamic Logic Programming Language for Relational Updates.* PhD thesis, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York, December 1987. Also published as Technical Report TR 88-2, Department of Computer Science, The University of Arizona, Tuscon, Arizona 85721, January, 1988.

[69] S. Manchanda and D.S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, Los Altos, CA, 1988.

[70] J. McCarthy. Situations, actions, and clausal laws, memo 2. Stanford Artificial Intelligence Project, 1963.

[71] J.M. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. Reprinted in *Readings in Artificial Intelligence*, 1981, Tioga Publ. Co.

[72] L.T. McCarty. Permissions and obligations. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 287–294, San Francisco, CA, 1983. Morgan Kaufmann.

[73] L.T. McCarty and R. van der Meyden. Reasoning about indefinite actions. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 59–70, Cambridge, MA, October 1992.

[74] D. McDermott. A temporal logic for reasoning about plans and processes. *Cognitive Science*, 6:101–155, 1982. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, A. Tate (eds.), Morgan-Kaufmann, 1990, 436–463.

[75] B.C. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Department of Computer Science, Stanford University, July 1983. Available as Technical Report STAN-CS-83-970.

[76] B.C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, U.K., 1986.

[77] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 251–262, New York, March 1988. ACM.

[78] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, MD, 1989.

[79] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publ. Co., Paolo Alto, CA, 1980.

[80] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A deductive database system. In *ACM SIGMOD Conference on Management of Data*, pages 308–317, New York, 1991. ACM.

[81] V.R. Pratt. Process logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 93–100, January 1979.

[82] V.R. Pratt. Action logic and pure induction. In *Workshop on Logics in AI*, volume 478 of *Lecture Notes in Computer Science*, pages 97–120. Springer-Verlag, 1990.

[83] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[84] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Aritifial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarty*, pages 359–380. Academic Press, 1991.

[85] R. Reiter. Formalizing database evolution in the situation calculus. In *Conf. on Fifth Generation Computer Systems*, 1992.

[86] R. Reiter. On formalizing database updates: Preliminary report. In *Proceedings of the Third Intl. Conference on Extending Database Technology (EDBT)*, March 1992.

[87] R. Reiter. On specifying database updates. Technical report, Univerity of Toronto, Department of Computer Science, 1992.

[88] R. Reiter. The projection problem in the situation calculus: A soundness and completeness result, with an application to database updates. Technical report, Univerity of Toronto, January 1992.

[89] D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 205–217, New York, April 1990. ACM.

[90] E.D. Sacerdoti. The non-linear nature of plans. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 206–214, 1975. Also appears in *Readings in Planning*, pp. 162–170. Morgan Kaufmann, San Mateo, CA, 1990.

[91] E.D. Sacerdoti. *A Structure for Plans and Behavior.* PhD thesis, Computer Science Department, Stanford University, 1975. Also published as Technical Note 109, SRI International, Inc., Menlo Park, CA.

[92] R.C. Schank and R.P. Abelson. *Scripts, Plans, Goals, and Understanding.* Freeman Publ. Co., San Francisco, CA, 1975.

[93] Y. Shoham. *Reasoning about Change.* The MIT Press, 1988.

[94] M.J. Stefik. *Planning with Constraints.* PhD thesis, Computer Science Department, Stanford University, 1980. Report Number 80-784.

[95] J. van Benthem. *Language in Action: Categories, Lambdas and Dynamic Logic.* Elsevier Science Pub. Co., Amsterdam, New York, 1991.

[96] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[97] M. Winslett. A model based approach to updating databases with incomplete information. *ACM Transactions on Database Systems*, 13(2):167–196, 1988.