Deductive and Object Data Languages: A Quest for Integration *

Michael Kifer

Department of Computer Science
University at Stony Brook
Stony Brook, NY 11794, U.S.A.
kifer@cs.sunysb.edu

Abstract. According to rumors, the early hybrids of object-oriented and deductive languages were mutants that escaped from secret Government AI labs. Whether this is true or not, the fact is that by mid-80's, database and logic programming communities began to take notice. The temptation was hard to resist: the object-oriented paradigm provides a better way of manipulating structured objects, while logic and deduction offer the power and flexibility of ad hoc querying and reasoning. Thus, hybrid languages have the potential for becoming an ideal turf for cultivating the next generation of information systems.

The approaches to integration of the two paradigms range from logic-based languages with unified declarative semantics, to message-passing prologs, to Prolog/C++ cocktails.

In the past eight years, my colleagues and I have been developing a unified object-based logic intended to capture most of the essentials of the object-oriented paradigm. The overall plot here is that once the fundamentals are in place, the actual hybrid programming languages can be carved out of the logic the same way as deductive data languages have been carved out of the classical logic.

There are two distinct aspects in object-orientation: structural (the statics) and procedural (the dynamics). The static aspect was somewhat easier to capture in logic and after taking a few wrong turns F-logic was developed [17]. Taming the dynamics was harder, because there was no widely accepted solution to the problem of state changes even for traditional deductive languages. Our recent work on Transaction Logic [8, 7] appears to provide an adequate framework for specifying the dynamics in classical deductive languages.

This paper first reviews some core aspects of F-logic and Transaction Logic, and then shows how the two can be combined in a natural way to yield a unified foundation for future work on deductive object-oriented languages both in theory and practice. At the end, we discuss areas that still remain to be explored.

* Work supported in part by the NSF grant IRI-9404629.

1 Introduction

The last decade saw a number of attempts at providing formal logical foundations for the object-oriented programming paradigm. The rational for this quest is that the object-oriented paradigm provides a better way of specifying and manipulating structured objects, while logic and deduction offer the power and flexibility of ad hoc querying and reasoning. Furthermore, both logic and the object-oriented paradigm promise to eliminate the infamous impedance mismatch,[2] leading to a much closer integration of programs and data.

One of the most important features of the object-oriented paradigm is the use of data abstraction, i.e., encapsulation of objects with complex internal structure accessible through publicly known functions called *methods*. Data abstraction has two faces: structural and behavioral.

Structurally, an object is observable through a collection of *signatures* associated with methods. Method signatures determine the type of arguments each method can take and the type of the results they return. Furthermore, semantically similar objects are classified into *classes* and the latter are organized in hierarchies, called *IS-A hierarchies* or *class hierarchies.*

IS-A hierarchies are useful because they help factor out information shared by most of the members in a class. Such information can be explicitly attached to the class and then implicitly *inherited* by subclasses and individual objects. Subclasses and objects that do not share the common information constitute *exceptions* and can overwrite the inheritance. This kind of inheritance is called *non-monotonic* and it has been extensively studied in the literature.

Another important idea underlying object-oriented languages is the notion of *type*. Generally, this term refers to arbitrary sets of abstract values. The collection of all types used in the language is called a *type system* and is specified via a *type expression sublanguage*. In object-oriented languages, types are used to specify collections of structurally similar objects, where "similarity" is manifested by a common set of methods applicable to objects of the same type. Methods are typed by specifying their signatures, as explained earlier.

Since objects that populate the same class share *semantic similarity*,[3] one can expect that they would be *structurally* similar as well. Moreover, the type of objects in a class C is inherited by every subclass of C, so the type of a subclass is a subset of the type of a superclass (this is sometimes called the "inclusion semantics" of class hierarchies). We therefore believe that in object-oriented languages typing is secondary to semantic classification.[4]

[2] A mismatch between the data manipulation language and the host programming language on another.

[3] From the data modeling point of view, semantic affinity is the only proper reason for placing different objects in the same class.

[4] Some people argue that the type hierarchy should be separate from the class hierarchy. While these arguments have merits, our view is that separation of the hierarchies leads to unjustified increase in complexity of the resulting formalisms and programming languages.

F-logic [17] was developed in order to capture the above aspects in a logical framework and thus provide a foundation for deductive object-oriented databases and logic programming. However, types and class hierarchies account only for the static aspect of objects. Accordingly, in F-logic one can specify the current state of an object base (which may include virtual objects, *i.e.*, objects defined via deductive rules) and then query it. However, F-logic cannot specify methods that change the internal state of an object.

For instance, the database may contain an object that records the current state of a robot. The object would have methods for querying the current location of the robot, the state of its arm, etc. However, it cannot have a method that would implement the command, "move north-east for one minute and report your new location and the battery state."

Handling state changes in logic has been known as a difficult issue. In fact, there is no widely accepted solution to the problem even in the classical setting, and the current practice of deductive databases and logic programming relies on a host of ad hoc solutions. We believe that Transaction Logic [7, 8] is such a solution.

Transaction Logic was developed to address the problem of specifying *and executing* logical procedures that update and permanently change the database. It accounts for update-related problems in several areas, such as databases, logic programming, and artificial intelligence. It has a clean model theory and a procedural interpretation that is sound and complete with respect to this semantics.

In the following sections, we first review the general ideas for combining deductive and object-oriented paradigms and try to put them in a perspective. We then survey the core ideas of F-logic and Transaction Logic and show how the two can be combined in a simple unified formalism, which we call Transaction F-logic. We conclude with a list of open problems pertaining Transaction F-logic.

2 Approaches to DOOD

To place our effort in the context of other works in the area, we will divide the different approaches to combining logic and objects into several broad categories. However, we have no intention here to undertake the perilous job of surveying the field. What's more, the division we propose is admittedly coarse and one may be tempted to argue with our classification of some works. Finally, by admitting that the bibliography is far from being complete (and was not intended to be so), we hope to avoid any possible embarrassment.

We distinguish three main categories of works that purport to combine logic and objects paradigm:

1. prolog++
2. other prolog extensions
3. object-oriented logics

The prolog++ category includes works where Prolog or some other logic-based language is tightly integrated with C++, Smalltalk, or other object-oriented imperative language. One of the earliest proposals in this vein was, perhaps, Bancilhon's object-oriented cocktail [6]. The general idea is to use C++ or Smalltalk to handle objects' structure, while logic is used as a language for specifying methods. Typical examples of this approach are [5, 31].

The second group of approaches tries to achieve integration by extending logic programming or deductive database paradigms with message passing, methods, and types (*e.g.*, [26, 13, 35]). Some commercial vendors also went in this direction.

In contrast to these approaches, object-oriented logics attempt to provide a unifying logical formalism for modeling objects and other aspects of the object-oriented paradigm. However, works in this category vary widely and wildly in the amount of object-orientation they capture and the degree to which they adhere to logical standards.

Some approaches are based on non-standard logics (*e.g.*, Linear Logic [4], rewrite logic [27, 28], dynamic logic [34]); others are based on modifications to classical logic (*e.g.*, [21, 9, 1]). The third subgroup here tries to develop new logics to address the issue of object-orientation. These include [2, 23, 18, 11] among others. Our own work is also in this subcategory.

Although integration of C++ with prologs is probably one of the fastest ways to add object-oriented flavor to otherwise logic-based languages, this is not a very satisfactory approach, as the resulting system consists of two clearly distinct parts with no unifying semantics. Likewise, extending existing Prolog systems with object-oriented features usually adds yet another ad hoc layer to systems that are already saddled with non-logical constructs (not to mention their incomplete evaluation strategy).

While agreeing that coupling logic languages with C++ and the like is very desirable, we believe that only by developing a unified logical foundation for the object-oriented paradigm true integration can be achieved. Attempting to build such a foundation using existing logics is very attractive, since this does not require the development of new logics. However, most logics were designed to address needs unrelated to objects, and squeezing objects out of them often meets limited success.

This experience led us to develop new logics, specifically designed to address the needs of object-oriented programming. The rest of this paper describes this approach, which, to the best of our knowledge, is much more developed (both in depth and breadth) than other works on object-oriented logics.

3 An Overview of F-logic

Object identity. Central to many object-oriented data languages is the notion of *object identity* (*oid*)—a simple concept that was in the center of heated debates in the days when object-orientation was the hottest and unchallenged trend in town. Our view is that objects are abstract or concrete entities in the real world and object identity is a syntactic gadget needed to refer to these objects in the programming language. More accurately, object identity has two faces: the *physical oid* is a purely implementational device used as a surrogate or a pointer to an object; the *logical oid* is a syntactic counterpart of the physical oid—it is a piece of syntax that programming languages use to provide handles to objects.

In our model, objects are referred to via logical oids, which are nothing but first-order terms, *e.g.*, *13*, *_256*, or *john32*. Function symbols can be used to construct more complex oids, *e.g.*, *father(mary)*, *head(csdept(stonybrook))*.

Attributes. In the relational model, a relation is a set of tuples, where a tuple can be viewed as a function mapping the set of attributes of a relation into a domain values. A dual view is that attributes are partial functions from the set of all objects into the domain. This dual view, due to Marek and Pawlak [29, 24], is essentially the beginning of an object-oriented data model. Despite being two decades old, the connection between this approach and object-oriented databases has not been widely recognized.

Casting this model in a concrete syntax, we obtain a simple kind of object-oriented logic where facts about objects are represented using *molecular formulas* like this:

$$john[name \rightarrow \text{“}John\ Doe\text{”};\ salary \rightarrow 20000]$$
$$father(mary)[address \rightarrow \text{“}Main\ St.\ USA\text{”};\ spouse \rightarrow sally]$$

Here, *john*, *father(mary)*, and *sally* are logical object ids that represent information about persons; *“John Doe”*, *“Main St. USA”* are oids of string-objects; 20000 is an oid of an integer-object; and *name*, *salary*, *address*, and *spouse*, are attributes. Note that elementary values, such as integers or strings, are objects and they serve as their own oids.

Set-valued attributes. The above model can account only for a very limited type of complex objects. For instance, there is no straightforward way of saying that John has children, Mary, Bob, and Alice. Nevertheless, it is easy to extend this model to support full-fledged complex objects. We just have to allow some attributes to be *set-valued*, i.e., be defined as functions mapping objects to the *powerset* of the domain (which is a set of all oids), rather than to the domain itself. For instance, we could define a set-valued attribute *children* and assert $john[children \twoheadrightarrow \{mary, bob, alice\}]$. What we called “attributes” earlier will now become *functional* (or *scalar*) attributes.

Methods. A *method* is a function that accepts arguments and—in the context of a specific object—returns result, which can be a single value of a set of values. Mathematically, the object that provides context is merely the first argument of the function; the other arguments are *proper* method arguments. The distinction between the first argument and the rest is made because in the object-oriented parlance, methods are invoked by sending a *message* to the object that provides the context; the message itself contains the name of the method and a list of proper arguments.

Although some object-oriented systems draw a sharp line between attributes and methods, conceptually, attributes are nothing but 0-ary methods, *i.e.*, methods that take no proper arguments (mathematically they are unary functions whose only argument is the context-object).

Class hierarchies. The next step is to add the class hierarchy—a common way of grouping semantically related objects together. To this end, we can introduce a separate sort $\mathcal{C}$ of constants, called *class-objects*, and a new kind of formulas, called *IS-A assertion*. With their help we could write $john : \mathbf{student}$, meaning that $john$ is an instance of the class **student**; or $\mathbf{student} :: \mathbf{person}$, meaning that the class **student** is a subclass of the class **person**. Note that we use two kinds of IS-A assertions: $a : b$, which denotes class membership, and $a :: b$, which denotes the subclass relation.

Higher-order syntax. The above model is essentially from [18, 20]; other works (*e.g.*, [23, 11]) also provided accounts for various parts of that model. However, it was not until higher-order syntax was introduced in [16] when logical formalization of object-oriented languages became fairly complete. There are several areas where higher-order syntax is desirable:

- Defining, manipulating, and reasoning about classes and their instances using the same language.
- The ability to define virtual classes (or views) and their attributes using deductive rules.
- Inheritance of schema and behavior.
- Schema exploration and browsing.

The danger in using higher-order logic as a basis for a programming language is that the resulting syntax may not be computable. However, higher-order syntax does not necessarily lead to computational problems, especially if, as in [10], the semantics remains first-order (*i.e.*, if higher-order variables range over intensional representations of sets rather than the sets themselves).

In F-logic, higher-order syntax is obtained by eliminating the hard syntactic distinction between classes, attributes, and oids. In particular, we no longer need to rely on the special sort $\mathcal{C}$ in order to represent classes. The same syntactic term may now be construed as an oid, a class, or a method, depending on the syntactic position of that term in the formula. Logical variables can be instantiated by attribute names as well as objects. This creates an opportunity for asking queries that return sets of attributes, classes, or any other aggregation that involves these higher-order entities. For instance, to inquire about all classes to which the object *john* belongs, write:

$$?-\ john : X$$

Here X is a variable that ranges over oids. Due to its syntactic position, the only oids that may satisfy the query after being substituted for X are oids of classes. Note that here and henceforth we use the standard Prolog notation whereby variables are capitalized.

Similarly to the above, to find all superclasses of the class *student*, one can write:

$$?-\ student :: X$$

The capabilities for browsing are actually quite extensive. To get a feeling of what can be expressed, consider

$$Obj[interesting\_attrs\_of(Class) \twoheadrightarrow \{A\}] \ \longleftarrow \ Obj[A \rightarrow V] \wedge V : Class$$

For each class *Class*, this defines a parameterized attribute *interesting_attrs_of (Class)*, whose value for any object *Obj* is the set of functional attributes (because of the single-headed arrow "$\rightarrow$" in the rule body) that are defined on *Obj* and return values that are objects from class *Class*. We could then pose a query

$$?- \ john[interesting\_attrs\_of(person) \twoheadrightarrow \{A\}]$$

to find out which functional attributes that return values of class *person* are defined for object *john*.

The next example shows how parametric classes can be defined in F-logic:

$$\begin{array}{l} nil : list \\ cons(X, L) : list(T) \longleftarrow X : T \wedge L : list(T) \end{array}$$

and how they can be used in a program:

$$\begin{array}{l} L[append@nil \rightarrow L] \longleftarrow L : list(T) \\ cons(X, L)[append@M \rightarrow cons(X, N)] \ \longleftarrow L : list(T)[append@M \rightarrow N] \wedge X : T \end{array}$$

Here, *append* is a unary method defined for objects in class *list*; its proper argument must also be a list. This definition is fairly close to the canonical example of *append* in logic programming, except that it is written in the object-oriented style.

Formal syntax. Formally, the three basic types of formulas are *IS-A assertions*, *F-molecules*, and *equations*. They can be combined into more complex formulas using the usual connectives $\wedge$, $\vee$, $\neg$, $\longleftarrow$, etc.

- *Is-a assertions* are statements of the form $P : Q$ or $P :: Q$
- *F-molecules* have the form P[*method expression; method expression; ...*]
- *Equations* have the form $P \doteq Q$

In the above, P and Q are Prolog-like terms, *i.e.*, tree-like structures built out of function symbols, variables, and constants. In F-logic, they are called *id-terms* and their intended use is to represent *logical object id's* (abbr., *oid*). An object can represent an individual entity, such as *honda*123, or a class of entities, such as *car*. Moreover, an object may play the role of a class in one context and of a member of a class in another context. For instance, in *honda*123 : *car*, the object represented by the oid *car* acts as a class and *honda*123 plays the role of a member of the class *car*. In contrast, in *car* : *vehicle_type*, the object *car* plays the role of a member of the class *vehicle_type* and in *car* :: *vehicle* it is a subclass of the class *vehicle*. We thus see that to differentiate between class membership and subclassing, F-logic uses two different symbols, ":" and "::".

This uniform treatment of classes and objects pays off in many ways. One advantage is that both data and schema of the knowledge base can be manipulated in the same language—a boon for applications dealing with ad hoc schema exploration by the user [17]. The other advantage is the ease with which object-oriented views can be defined in F-logic.

Method expressions in a molecule may be of six different kinds. The first four are called *data* expressions. They are used to describe properties of objects and classes; the last two types of method expressions are called *signature* expressions; they are intended as type definitions. The exact syntax of method expressions is as follows:

– A *non-inheritable* k-ary *scalar data* expression $(k \geq 0)$ has the form:

$$ScalarMethod@Q_1, ..., Q_k {\rightarrow} T$$

– A *non-inheritable* l-ary *set-valued data* expression $(l, m \geq 0)$ is of the form:

$$SetMethod@R_1, ..., R_l {\rightarrow\!\!\!\rightarrow} \{S_1, \ \ldots \ , \ S_n\}$$

– An *inheritable* k-ary *scalar data* expression $(k \geq 0)$ has the form:

$$ScalarMethod@Q_1, ..., Q_k \bullet\!\!\rightarrow T$$

– An *inheritable* l-ary *set-valued data* expression $(l, m \geq 0)$ is of the form:

$$SetMethod@R_1, ..., R_l \bullet\!\!\rightarrow\!\!\!\rightarrow \{S_1, \ \ldots \ , \ S_n\}$$

– A *scalar signature* expression $(n, r \geq 0)$:

$$ScalarMethod@V_1, ..., V_n {\Rightarrow} A$$

– A *set-valued signature* expression $(s, t \geq 0)$:

$$SetMethod@W_1, ..., W_s {\Rightarrow\!\!\!\Rightarrow} B$$

In the above, $ScalMethod$, $SetMethod$, the R_i's, the V_i's, A, B, etc., are all id-terms that denote objects. In particular, this means that methods and attributes are represented via oids and can be manipulated the same way as objects are.

Both data expressions and signatures can be either scalar or set-valued. A molecule with a scalar data expression, *e.g.*, $p[scalMthd@q, r{\rightarrow}t]$, says that when $scalMthd$ is invoked with arguments q and r on the object with the oid p, it returns an object with the oid t. A molecule with a set-valued data expression, $p[setMthd@s{\rightarrow\!\!\!\rightarrow}\{u, v\}]$, says that invocation of $setMthd$ on the object p with argument s will return a set of objects that *contains*[5] objects u and v.

[5] The fact that this set *contains* (rather than equals) the set $\{u, v\}$ is a subtle and important point, as explained in [17]. The use of set-containment instead of equality leads to computational benefits as well as the ease of programming tasks involving the all-important operation of grouping data into sets.

Data expressions can be inheritable or non-inheritable. The arrows $\rightarrow$, $\twoheadrightarrow$ denote non-inheritable expressions and $\bullet\!\!\rightarrow$, $\bullet\!\!\twoheadrightarrow$ are used in inheritable expressions. The difference between inheritable and non-inheritable expressions shows only when it comes to inheritance and typing (naturally!). In all other respects, the two categories of data expressions are identical.

The actual language of F-logic described in [17] is slightly more general than what we just described—it allows predicates to be used as stand-alone atomic formulas and inside F-molecules. However, these details are unimportant here.

Semantics. The semantics of F-logic is fully described in [17]. Since our prime concern in this paper is the dynamics of objects, we shall drop the subjects concerning inheritance and typing, and will limit the discussion to Herbrand-style semantics for formulas that involve IS-A assertions and non-inheritable data expressions only. All other method expressions will be ignored.

In this limited setting, the semantics is fairly simple. First, we define Herbrand F-structures where the subclass relationship "::" is transitive and ":" works as behooves the membership relation. In addition, functional methods are made to look like functions.

Formally, the *Herbrand universe* $\mathcal{U}$ is a set of all *ground* (*i.e.*, variable-free) terms (or oids in the F-logic terminology) and the *Herbrand base* $\mathcal{B}$ is a set of all ground F-molecules and IS-A assertions.

A Herbrand F-structure $\mathbf{I}$ is a subset of the Herbrand base that satisfies the conditions below. First, the equations in $\mathbf{I}$ must satisfy the usual properties of equality: reflexivity, symmetry, transitivity, and substitution. Second, IS-A assertions must satisfy:

IS-A reflexivity: $p :: p \in \mathbf{I}$, for any $p \in \mathcal{U}$

IS-A transitivity: If $p :: q, q :: r \in \mathbf{I}$ then $p :: r \in \mathbf{I}$

IS-A acyclicity: If $p :: q, q :: p \in \mathbf{I}$ then $p \doteq q \in \mathbf{I}$

Subclass inclusion: If $p : q, q :: r \in \mathbf{I}$ then $p : r \in \mathbf{I}$

Functional methods must satisfy the property of *scalarity*:

$$\text{If } p[\mathit{scalM}@q_1, \ldots, q_k \rightarrow r_1],\ p[\mathit{scalM}@q_1, \ldots, q_k \rightarrow r_2] \in \mathbf{I} \text{ then } r_1 \doteq r_2$$

In addition, if $\alpha \in \mathbf{I}$ then every *sub-molecule* of α is also in $\mathbf{I}$. Submolecules are obtained from F-molecules by removing some method expressions and set-elements from set-valued method expressions. The order of method expressions (or set-elements in set-valued method expressions) is immaterial.

For simplicity, we limit our attention to Horn rules, *i.e.*, formulas of the form

$$\alpha \longleftarrow \beta_1 \wedge \cdots \wedge \beta_n \tag{1}$$

where α and the β_j's are IS-A assertions, equations, or F-molecules. All variables in such a formula are assumed to be implicitly universally quantified outside the formula. Similarly to the classical logic, the implication sign here is a short-hand for $\alpha \vee \neg(\beta_1 \wedge ... \wedge \beta_n)$.

The definition of satisfaction of such molecules in Herbrand F-structures is essentially identical to satisfaction in the classical case (see, *e.g.*, [22]). Namely, for a ground molecular formula, an equation, or an IS-A atom, $\mathbf{I} \models \alpha$ (read: $\mathbf{I}$ satisfies α) if and only if $\alpha \in \mathbf{I}$. For a ground rule R of the form (1), $\mathbf{I} \models R$ if and only if whenever $\beta_1, ..., \beta_n \in \mathbf{I}$, then also $\alpha \in \mathbf{I}$. A non-ground rule of the form (1), is satisfied by $\mathbf{I}$ if and only if every ground instantiation of it is satisfied by $\mathbf{I}$.[6]

Summary. We described a fragment of F-logic and outlined its semantics. The logic extends classical logic by allowing direct representation of complex objects. This makes it an ideal foundation for object-oriented deductive databases and logic programming. In fact, F-logic was designed to satisfy the equation

$$\frac{\text{deductive object-oriented paradigm}}{\text{F-logic}} = \frac{\text{deductive relational paradigm}}{\text{classical logic}}$$

where the deductive relational paradigm includes classical deductive databases, logic programming, and relational calculus.

Since F-logic also has a proof theory that is sound and complete with respect to the semantics [17], F-logic programs have operational semantics. However, just like classical logic, F-logic has one important limitation: it is not very well suited for describing the dynamics of objects. That is, it provides extensive facilities for defining and querying object states, but not for changing them. This is where Transaction Logic takes charge.

4 A Primer on Transaction logic

Transaction Logic was designed with several applications in mind, especially in databases, logic programming, and AI. It was therefore developed as a general logic for solving a wide range of update-related problems. These problems, both practical and theoretical, are discussed in great detail in [8]. For instance, Transaction Logic provides a logical account for many update-related phenomena. In logic programming, this leads to a clean, logical treatment of *assert* and *retract* operators in Prolog and effectively extends the theory of logic programming to include updates as well as queries. In object-oriented databases, Transaction Logic can be combined with F-logic to provide a logical account of methods that manipulate objects' internal states. While F-logic covers the structural aspect of object-oriented databases, its combination with Transaction Logic would account for the behavioral aspect as well (see Section 5). In AI, Transaction Logic provides a declarative account for planning [8]. Despite the previous efforts to give these phenomena declarative semantics, Transaction Logic is the first unifying framework to account for all of them.

[6] As usual, a ground instantiation of a formula is obtained from that formula by consistent simultaneous replacement of all variables with ground terms.

The syntax of Transaction Logic is built around three basic ideas: *serial conjunction*, *elementary state transitions*, and *uniformity* in representing queries and transactions. The first is intended to capture situations where one transaction is executed right after the other; elementary transitions account for the idea of elementary state changes in a database, such as adding or deleting a tuple, or assigning the contents of one relation to another.

The third idea is what makes Transaction Logic relevant to object-oriented databases. All previous work on declarative treatment of updates makes a clear distinction 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 method invocation. In such systems, an update can be thought of as a query with side effects. In fact, every method is simply a *program* that operates on data. Transaction Logic models this uniformity naturally by treating methods and queries equally, thereby providing a logical foundation for the behavioral aspect of object-oriented databases.

We illustrate these ideas using an example that models movements of a robot arm in a world of toy blocks. States of this world are defined in terms of three database predicates: $on(x, y)$, which says that block x is on top of block y; $clear(x)$, which says that nothing is on top of block x; and $wider(x, y)$, which says that x is wider than y. The following rules define four 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).

$$\begin{array}{lcl} stack(N, X) & \leftarrow & N > 0 \otimes move(Y, X) \otimes stack(N-1, Y) \\ stack(0, X) & \leftarrow & \\ move(X, Y) & \leftarrow & pickup(X) \otimes putdown(X, Y) \\ pickup(X) & \leftarrow & clear(X) \otimes on(X, Y) \otimes on.del(X, Y) \otimes clear.ins(Y) \\ putdown(X, Y) & \leftarrow & wider(Y, X) \otimes clear(Y) \otimes on.ins(X, Y) \otimes clear.del(Y) \end{array} \tag{2}$$

Here "$\otimes$" is a symbol we use for *serial conjunction*—a new connective of Transaction Logic. Intuitively, $a \otimes b$ means "do a then do b." The symbols *clear.ins*, *on.ins*, *on.del* and *clear.del* are predicates associated with elementary state transitions (see later) that accomplish tuple insertion/deletion into/from the corresponding relations *on* and *clear*. There is nothing magic about these predicates in Transaction Logic, and their fancy syntax (*on.ins*, *clear.del*) is just a convention we use. The only special thing here is that we assume that their meaning is defined by the transition oracle (see later). We also note that elementary transitions need not be limited to simple insertions and deletions, and the database states are not limited to relational databases. (See [8] for an extensive discussion.)

The basic actions $pickup(X)$ and $putdown(X, Y)$ mean, "pick up block X," and "put down block X on top of block Y," respectively. Both are defined via elementary inserts and deletes to database relations. For instance, the rule for *putdown* reads:

> To execute $putdown(X, Y)$ for some X and Y, first make sure that Y is *wider* than X and that $clear(Y)$ holds in the current database state. If all is well, insert the tuple $\langle X, Y \rangle$ in relation *on*, making a transition to a new database state. Next delete the tuple $\langle Y \rangle$ from *clear*, jumping to yet another state. If preconditions $clear(Y)$ or $wider(Y, X)$ fail, then $putdown(X, Y)$ is not executed at the current state.

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." It is performed by first doing *pickup(X)* then *putdown(X, Y)*. Similarly, $stack(N, X)$ means, "stack N arbitrary blocks on top of block X." This is done by doing nothing if $N = 0$ and, if $N \neq 0$, by moving some block Y on top of X and then recursively stacking $N - 1$ blocks on top of Y. To build a tower of ten blocks with $blkC$ at the bottom, we can now invoke the following transaction:

$$?-\ stack(A, blkC) \tag{3}$$

The actions *pickup* and *putdown* are deterministic: Each set of argument bindings specifies only one robot action. In contrast, the action *stack* is *non-deterministic*. To perform this action, the inference system searches the database for blocks that can be stacked. If, at any stage of the search, several such blocks are eligible, the system arbitrarily chooses one of them.

Non-determinism has applications in many areas, but it is especially well-suited for advanced applications, such as those found in Artificial Intelligence, CAD, and intelligent information systems. For instance, the user of a robot simulator might ask the robot to build a stack of blocks, but she may not say which blocks to use. In a trip planning information system, we may ask for a trip plan without fixing the exact route, except in the form of constraints (such as certain intermediate points, duration, etc.). 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. Transaction Logic enables users to specify what choices are allowed. When a user issues a non-deterministic transaction, the system makes particular choices and takes the database into an allowed new state. The run-time choices may be implementation-dependent, but they must be in accord with the semantics and the proof theory.

Observe that (2) can be easily re-written in Prolog form, by replacing "$\otimes$" with "," and the elementary state transitions (*e.g.*, *on.ins* or *clear.del*) with *assert* and *retract*. However, the resulting, seemingly innocuous, Prolog program will not execute correctly! Indeed, suppose the robot had picked up block $blkA$ and is now searching for another block to put $blkA$ down on. Now, suppose that $blkA$ 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, $blkA$ cannot be used for stacking.

In Transaction Logic, the evaluation strategy will then try to find another block to pick up. This behavior is firmly supported by Transaction Logic's proof theory, which not only establishes truth value, but also *actually executes* transactions. In contrast, Prolog's behavior is ad hoc. It's evaluation strategy will backtrack, leaving the database in an inconsistent state: If $blkA$ was previously on top of $blkB$, then $on(blkA, blkB)$ will remain deleted and $clear(blkB)$ will stay in the database. Clearly, this is inconsistent with the robot's internal state since, after backtracking, $blkA$ must stay on top of $blkB$, and the latter block should not be clear. Fixing this Prolog program will make it much more cumbersome and heavily dependent on Prolog's backtracking strategy.

In (2), the action *stack* is highly non-deterministic, *i.e.*, if the user invokes this action by posing the goal $?-\ stack(10, blkC)$, then the robot may stack 10 blocks on top of $blkC$ in many different ways. Suppose, however, that the robot is now used to perform a more specific task, which requires that whenever it stacks a red block, the next block to be stacked must be blue. Instead of re-programming the entire transaction (3), the user can impose a constraint on *transaction execution* itself. A modified transaction may look like this:

$$\begin{array}{l} ?-\ stack(10, blkC) \\ \quad \wedge[color(X, red) \otimes move(X, Y) \otimes color(Z, blue) \Rightarrow move(Z, X)] \end{array} \tag{4}$$

The connective $\Rightarrow$ here is called *serial implication*; it is defined as follows: $a \Rightarrow b \ \equiv\ \neg(a \otimes \neg b)$. A statement of the form $a \Rightarrow b$ has the following English meaning: whenever action a is performed, the next action must be b. The connective "$\wedge$" is an extension of classical conjunction. Its semantics in Transaction Logic is such that the two conjuncts in (4) must behave consistently with each other, *i.e.*, they must execute along the same execution path. The ability to express constraints on transaction execution is a very powerful feature of Transaction Logic, which gives the ability to modify transactions without re-programming them.[7]

In Transaction Logic, the user can specify a wide range of constraints, such as "these two transactions terminate at the same time," or "this transaction must precede the other," etc. In particular, [8] shows that it is easy to express the well-known Allen's temporal constraints [3]. The ability to specify constraints on transaction execution paths is a unique and powerful feature Transaction Logic. What is usually referred to as "dynamic constraints" in database literature is a much weaker kind of constraint. Such a constraint may say that database states before and after execution of a transaction should stand in certain relationship to each other (*e.g.*, "employee's salary must not decrease"), but it can say nothing about sequencing of actions to be performed (such as (4) above or such as the constraint, "shut the door after leaving the room").

We should note that the core of Transaction Logic can be efficiently evaluated using either a top-down or a bottom-up strategy, as described in [8]. This core includes programs such as (2).

Syntax. For our subset of Transaction Logic, we assume a language with a countably infinite set of function symbols $\mathcal{F}$ and predicates $\mathcal{P}$ plus the logical

[7] In truth, the above constraint is not doing what we said it would. The correct constraint is a notch more complex. The reader is referred to [8] for an extensive discussion of constraints on transaction execution paths.

connectives $\wedge$, $\otimes$, $\neg$, and the quantifier $\forall$. Terms are defined as in first-order logic, and formulas are constructed out of terms, predicates, and connectives as usual.

The connective $\wedge$ is called *classical conjunction* and $\otimes$ is a new connective, called *serial conjunction.* Actually, calling $\wedge$ a "classical" connective is somewhat of a misnomer. Both $\otimes$ and $\wedge$ reduce to classical conjunction when it comes to static matters. However, when things turn dynamic, both $\wedge$ and $\otimes$ extend classical conjunction (albeit in different directions). As we have seen from the examples, $\otimes$ is used to say that one action immediately follows the other. In contrast, $\phi \wedge \psi$ says that two actions must execute along the same sequence of states. In other words, ψ can be viewed as a constraint on the execution path of ϕ (or vice versa). We have seen an example of this in (4).

Additional connectives can be defined in terms of the existing ones. We have seen from the examples that, as in the classical case, what is needed for programming is the implication connective; it is defined like in classical logic:

$$\begin{array}{l} a \leftarrow b \ \equiv a \vee \neg b, \quad \text{where} \\ a \vee b \equiv \neg(\neg a \wedge \neg b) \end{array}$$

As with $\wedge$ and $\otimes$, $\leftarrow$ and $\vee$ reduce to their classical counterparts in static situations. However, in dynamic situations, $\vee$ is a means of specifying non-determinism in the execution, while " $\leftarrow$ " lets us define complex operations in terms of simpler ones. For instance, $a \leftarrow b \otimes c$ means, to execute a, first do b then c.

In (4), we have seen another connective, $\Rightarrow$, which has no counterpart in classical logic. The statement $a \Rightarrow b$ means, b must execute right after a (this is different from $a \otimes b$—it should be interpreted as a constraint rather than definition). In (4), this statement was used to impose a constraint on how the transaction $stack(10, blkC)$ can execute. Despite not having a classical counterpart, $\Rightarrow$ is not an independent logical connective. It can be expressed as:

$$\begin{array}{rcl} a \Rightarrow b & \equiv & \neg a \oplus b, \quad \text{where} \\ a \oplus b & \equiv & \neg(\neg a \otimes \neg b) \end{array}$$

In that sense, it is a serial analogue of classical implication.

States, transitions, and oracles. In a sense, Transaction Logic is a true logic for specifying complex behavior. It does not concern itself with such banalities as the particular semantics of the underlying database states or what the *elementary* state changes do to these states. Instead, it can take almost any theory of database states (with almost any closed-world-style semantics) and any theory of elementary state changes and yield a rich formalism where complex state changes can be defined in terms of simpler ones. This approach has many advantages.

First, the semantics of database states is already well-developed. Perfect models [30], stable models [12], well-founded models [33], Clark's completion [22] are part of a lingo that everybody understands (or pretends to understand). The only problem is that we cannot agree on which one is The Semantics. In fact, some people strongly argue that no single semantics is suited for all kinds of applications. Transaction logic does not depend on the resolution of this controversy. (Actually, most logics for specifying updates work with the simplest kind of states—relational databases. So, in that respect, Transaction Logic with its data oracle may be too far ahead of our times.)

Second, by factoring out the issue of elementary updates, we managed to split a complex problem into two simpler ones. All approaches we have seen so far attempt to deal with the problem of updates as a whole. (We purposely avoid making specific references here, as this issue is discussed at length in [8], which also surveys a large number of works in this area.) The problem with such monolithic approaches is that it is hard to achieve the desired generality of results, and it is hard to see the relationship between different works. Moreover, the majority of works on actions actually deals with states that are essentially relational databases, and the actions they consider are sequences of insertions and deletions of tuples. In many cases, the big issue is how to reign in The Infamous Frame Problem [25], *i.e.*, how to say that "nothing else changes" except the changes specified explicitly and the possible ramifications of those explicit changes.

In fact, for the works where states are sets of atomic formulas (*i.e.*, relational databases), elementary transitions are not really an issue. Indeed, in this case there is a simple and useful repertoire of elementary updates: for the most part, one can live with run-of-the-mill tuple deletions and insertions, and fancier updates (*e.g.*, relational assignment) can be thrown in if needed. One can simply plug in such an obvious theory of elementary updates and obtain a powerful logic for specifying complex actions over relational databases. The same is easily done for object bases, *e.g.*, sets of molecular formulas in F-logic.

Even more complicated theories of states have useful and well-understood theories of elementary updates. For instance, in deductive databases and logic programming, an update may insert or delete a fact, as in the relational case, but it can also insert or delete a deductive rule. The latter kind of update is commonly used in logic programming and it is computationally inexpensive. Note that adding or deleting a rule is *not* the same as adding or deleting a formula to a logical theory (such as, *e.g.*, in [15]), so it does not require elaborate theories. Plugging such a theory of elementary changes into Transaction Logic leads to an expressive formalism for specifying complex actions over deductive databases.

To make a long story short, recall that any classical logic theory has a parameter—a language for constructing well-formed formulas. In addition to this, Transaction Logic is parameterized by a pair of oracles, a *data oracle* and a *transition oracle*. The data oracle handles matters pertaining the semantics of states, and the transition oracle handles simple state changes. The data oracle's job is to field questions of the form "is this first-order formula true in a given state?" The transition oracle answers questions such as, "can I jump from this state to that state using this elementary operation?" Note that these "elementary operations" can actually be quite complex. They can be Fourier transforms, weather simulations, or even updates made by dusty decks of Cobol cards. In other words, to Transaction Logic, an operation is elementary as long as its internals are unknown to the logic. As far as the logic is concerned, the oracles need not even be computable. In fact, the proof theory in [8] is formulated in such a way that its completeness holds *modulo* the oracles.

The oracles come with a collection of states. These are the only states the logic knows about (while the oracles are plugged in). For instance, for relational oracles (whose elementary changes are tuple deletion and insertion), the set of states is, naturally, the collection of all relational databases. For deductive database oracles (whose elementary changes are fact and rule deletions or insertions), the states are sets of facts and rules, *i.e.*, deductive databases.

Formally, each oracle is a mapping. The data oracle, $\mathcal{O}^d$, is a mapping from states to sets of first-order formulas. Likewise, the transition oracle, $\mathcal{O}^t$, is a mapping from pairs of states to sets of ground atomic formulas.

Intuitively, $a \in \mathcal{O}^d(\mathbf{D})$ means that a is true at state $\mathbf{D}$. This does not necessarily mean that a is a logical consequence of $\mathbf{D}$. Indeed, database states may, for instance, rely on a closed-world semantics, so the oracle may pronounce a to be true because $\neg a$ could not be proved. Similarly, $b \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$ means that b is an elementary update that changes state $\mathbf{D}_1$ into state $\mathbf{D}_2$.

Primitive data access is thus specified outside the logic via the oracles. We do not expect the oracles to be coded by casual users. Although the oracles allow for many different semantics, we envision that any programming system based on Transaction Logic will likely provide a carefully selected repertoire of built-in database semantics and a tightly controlled mechanism for adding new ones. This latter mechanism will not be available to ordinary programmers. For this reason, we can assume that the data and transition oracles are fixed.

Herbrand semantics. For simplicity, we give only a Herbrand-style semantics to the fragment of Transaction Logic outlined earlier. Recall that the Herbrand universe, denoted by $\mathcal{U}$, is the set of all ground first-order terms that can be constructed out of the function symbols in the language. The Herbrand base $\mathcal{B}$ is a set of all ground atomic formulas in the language, and a classical Herbrand structure is any subset of $\mathcal{B}$.

Since Transaction Logic deals with execution, its formulas are evaluated on execution paths rather than at states.[8] To capture this idea, we introduce what we call *path structures*.

Definition 1 (Herbrand Path Structures). A *path* of length k over $\mathcal{L}$ is an arbitrary finite sequence of states, $\langle \mathbf{D}_1, \ldots, \mathbf{D}_k \rangle$, where $k \geq 1$.

A *Herbrand path structure* is a mapping, $\mathbf{M}$, that assigns a classical Herbrand structure to every path. The mapping is subject to the following restrictions:

1. *Compliance with data oracle*:
   For every state s and every $\phi \in \mathcal{O}^d(s)$, $\mathbf{M}(\langle s \rangle) \models^c \phi$, where $\models^c$ denotes

[8] Some versions of Process Logic [14] also take this approach. However, the semantics, the intent, and the application domains are entirely different—see [8] for details.

classical entailment. This means that the semantic structure assigned to the path $\langle s \rangle$ must be a classical model of the formulas that the data oracle says are true in s;

2. *Compliance with transition oracle*:
For every pair of states $\mathbf{D}_1, \mathbf{D}_2$, $\mathbf{M}(\langle \mathbf{D}_1, \mathbf{D}_2 \rangle) \models^c \phi$ whenever $\phi \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$. In other words, elementary transitions specified by the transition oracle are indeed transitions in the path structure.

The mapping $\mathbf{M}$ says which atoms are true on what paths. These atoms denote actions that take place along various paths. As we shall see, these actions can be partially or completely specified by logical formulas. For instance, the formula $a \leftarrow \phi \otimes \psi$ says that action a occurs whenever action ϕ is followed by action ψ. In a logic-programming context, the atom a can be viewed as a *name* for the complex action $\phi \otimes \psi$. The ability to name complex actions is tantamount to having a subroutine facility.

Earlier we mentioned that formulas in Transaction Logic are evaluated on paths, not at states. Now we can make this statement precise:

Definition 2 (Satisfaction). Let $\mathbf{M}$ be a Herbrand path structure and π be an arbitrary path. Let ν be a variable assignment, *i.e.*, a mapping from variables to the Herbrand universe $\mathcal{U}$. Then:

1. **Base Case:** $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ iff $\mathbf{M}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, where $p(t_1, \ldots, t_n)$ is an arbitrary atomic formula and $\models^c_\nu$ is the classical satisfaction relation.
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. **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* paths π_1 and π_2 whose concatenation is π.[9]
5. **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.

If $\mathbf{M}, \pi \models \phi$, then we say that ϕ is *satisfied* (or is *true*) on path π in the path structure $\mathbf{M}$ (as in classical logic, the mention of variable assignment can be omitted for closed formulas).

In Definition 2, the base case captures the intuition behind transaction execution along a path: In Transaction Logic, the truth of $p(t_1, \ldots, t_n)$ on a path π means that transaction p *can* execute along π when invoked with arguments $t_1, \ldots, t_n$.

The "classical" connectives $\wedge$ and $\neg$ are defined in a classical fashion. For these connectives, truth on a path depends only on the path itself. This similarity is the main reason for us calling these connectives "classical." However, given

[9] A path π is a concatenation of paths $\langle \mathbf{D}_1, ..., \mathbf{D}_n \rangle$ and $\langle \mathbf{D}'_1, ..., \mathbf{D}'_k \rangle$ iff $\mathbf{D}_n = \mathbf{D}'_1$ and $\pi = \langle \mathbf{D}_1, ..., \mathbf{D}_n = \mathbf{D}'_1, \mathbf{D}'_2, ..., \mathbf{D}'_k \rangle$.

that truth is defined on paths rather than states, these connectives are not the same as the classical ones.

In contrast, for the new connective $\otimes$, truth depends not on the path, but on its *subpaths*. Also note from the definitions that on paths of length 1, both "$\otimes$" and "$\wedge$" reduce to the usual classical conjunction and both "$\oplus$" and "$\vee$" reduce to the classical "$\vee$." Thus, the "classical" and the "serial" connectives extend the usual connectives of the classical logic from states to executions (paths), albeit in different ways.

Earlier we mentioned that a statement like $a \leftarrow body$ can be viewed as an assignment of a name, a, to a procedure specified by *body*. To see why this is so, suppose $\mathbf{M}, \pi \models a \leftarrow body$. By Definition 2, if *body* is true along π then so must be a. In the minimalistic sense, this means that to execute a one needs to execute *body*. Now, suppose *body* is actually a serial conjunction, say, $b \otimes c$. Then *body* is true on π if and only if b is true on a prefix of π and c is true on the remainder of π. Thus, executing a amounts to first executing b and then c. The reader can verify that if a were defined via two or more rules then it would become a non-deterministic transaction: executing a would amount to executing the body of any one of the rules defining a. Further details can be found in [8].

The above discussion suggests that in path structures that satisfy implications, dependencies may exist between classical Herbrand structures associated with a path and its subpaths. Similarly to classical logic programming, the user writes down definitions of transactions as a program and the meaning is determined by the *models* of this program. In Transaction Logic, $\mathbf{M}$ is a *model* of a formula ϕ, written $\mathbf{M} \models \phi$, if and only if $\mathbf{M}, \pi \models \phi$ for each path π in $\mathbf{M}$. $\mathbf{M}$ is a model of a set of formulas if and only if it is a model of each formula in the set.

Executional entailment. In classical logic, one of the most interesting questions usually is, does one formula imply the other? In databases, this translates into a question of whether an answer to a query is implied by the definition of the query and by the database. Such questions are still of interest to our logic, but a more important issue is addressed by the notion of *executional entailment*—a concept that provides logical account for transaction execution.

Definition 3 (Executional Entailment). Let $\mathbf{P}$ be a *transaction base*, *i.e.*, a set of formulas in Transaction Logic and let ϕ be a transaction formula. Let $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ be a sequence of database states. Then, the statement

$$\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \phi \tag{5}$$

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 (5) is the statement

$$\mathbf{P}, \mathbf{D}_0 \text{---} \models \phi \tag{6}$$

which is true if and only if there is a sequence of databases $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ such that Statement (5) is true. The importance of (6) is in that neither the user nor the system usually knows what the states $\mathbf{D}_1, \ldots, \mathbf{D}_n$ are going to be. Instead, only $\mathbf{P}$ (the program) and $\mathbf{D}_0$ (the initial state) are known. The subsequent states, $\mathbf{D}_1, \ldots, \mathbf{D}_n$, are to be *generated as part of the execution* of the transaction ϕ. This is precisely what the proof theory in [8] is meant for.

Lemma 4 (Basic Properties of Executional Entailment). *For any transaction base* $\mathbf{P}$*, any database sequence* $\mathbf{D}_0, \ldots, \mathbf{D}_n$*, and any transaction formulas* α *and* β*, the following statements are all true:*

1. *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* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \otimes \beta$.
2. *If* $\alpha \leftarrow \beta$ *is in* $\mathbf{P}$ *and* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ *then* $\mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$.
3. *If* $\alpha \in \mathcal{O}^t(\mathbf{D}_0, \mathbf{D}_1)$ *then* $\mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha$.
4. *If* $\mathcal{O}^d(\mathbf{D}_0) \models^c \psi$ *then* $\mathbf{P}, \mathbf{D}_0 \models \psi$*, where* ψ *is a first-order formula, and* $\models^c$ *denotes classical entailment.*

Example. To illustrate how executional entailment is proved, consider a rule from (2)

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

and suppose the relational transition oracle defines delete and insert operations in the usual way. In particular,

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

for all blocks. Suppose the initial state $\mathbf{D}_0$ represents an arrangement of three blocks where $blkA$ is on top of $blkC$, and $blkB$ stands by itself:

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

Consider the transaction $?\!-\ pickup(X)$ (pick up a block) executed at the initial state $\mathbf{D}_0$. We can infer that the robot can pick up $blkA$ and, in the process, the execution may pass through the intermediate state $\mathbf{D}_1$ and end up at state $\mathbf{D}_2$, where

$$\begin{aligned} \mathbf{D}_1 &= \{isclear(blkA), isclear(blkB)\} \\ \mathbf{D}_2 &= \{isclear(blkA), isclear(blkB), isclear(blkC)\} \end{aligned}$$

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. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models isclear(blkA)$, by item 4 of Lemma 4.
2. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models on(blkA, blkC)$, by item 4 of Lemma 4.
3. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models on.del(blkA, blkC)$, by item 3 of Lemma 4.
4. $\mathcal{B}, \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \models isclear.ins(blkC)$, by item 3 of Lemma 4.

5. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \ \models\ isclear(blkA) \otimes on(blkA, blkC)$, by lines 1 and 2, and item 1 of Lemma 4 (item 1 works here because the concatenation of paths $\langle \mathbf{D}_0 \rangle$ and $\langle \mathbf{D}_0 \rangle$ is $\langle \mathbf{D}_0 \rangle$).
6. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \ \models\ isclear(blkA) \otimes on(blkA, blkC) \otimes on.del(blkA, blkC)$, by lines 5 and 3, and item 1 of Lemma 4.
7. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \ \models\ isclear(blkA) \otimes on(blkA, blkC) \otimes on.del(blkA, blkC) \otimes isclear.ins(blkC)$,
   by lines 6 and 4, and item 1 of Lemma 4.
8. $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \ \models\ pickup(blkA)$, by line 7, the *pickup*-rule, and item 2 of Lemma 4.

Summary. We have sketched a subset of Transaction Logic. We discussed the main underlying ideas, the semantics, and hinted at how the proof theory works. Transaction Logic is actually much richer than what we outlined here. It includes operators and connectives that let the user define hypothetical actions, imperative constructs, and even concurrently running processes. See [8] for the full development.

5 Transaction F-logic — The Ultimate Brew

As defined above, Transaction Logic is suitable for specifying dynamics in the context of traditional deductive databases. However, the design is generic enough to be applicable to other logics, F-logic included.

There is more than one way to combine F-logic and Transaction Logic. We find that the simplest way to explain how this can be done is by appealing to a connection that exists between F-logic and classical predicate calculus.

In [17], it was shown that much of F-logic can be encoded in predicate calculus. To see how, consider an F-molecule, say,

$$bob[name \rightarrow \text{“}Bob\text{”};\ salary \rightarrow 50000;\ children \twoheadrightarrow \{mary, sally\}]$$

This is roughly equivalent to

$$\begin{aligned} &name(bob, \text{“}Bob\text{”}) \wedge salary(bob, 50000) \\ &\qquad\qquad \wedge children(bob, mary) \wedge children(bob, sally) \end{aligned}$$

(the correct translation is more involved, but the above is fine for our purposes). This translation suggests a simple way of implanting Transaction Logic in F-logic (or F-logic in Transaction Logic, depending on one's point of view). Namely, where previously we would view *name.ins* and *name.del* as predicates defined by the transition oracle, we shall now consider them as attributes (or, more generally, methods) defined by this oracle. Similarly, where previously states were sets of first-order formulas, they will now be sets of F-logic formulas. The data oracle, then, will be telling the system which F-logic formulas are true at which states.

We call the resulting eclectic logic *Transaction F-logic.* To illustrate the idea, we modify and expand example (2) from Section 4. For convenience, we use one

$$
\begin{aligned}
R[pickup@Blk \to Frm] \longleftarrow\ & R : robot \otimes Blk : block \otimes Frm : block \\
& \otimes Blk[top \to clear; bottom \to Frm] \otimes R[state \to idle] \\
& \otimes Blk[bottom.del \to Frm] \otimes Frm[top.ins \to clear] \\
& \otimes R[state.repl \to holding] \\
R[putdown@Blk \to To] \longleftarrow\ & R : robot \otimes Blk : block \otimes To : block \otimes To[top \to clear] \\
& \otimes wider(To, Blk) \otimes R[state \to holding] \\
& \otimes Blk[bottom.ins \to To] \otimes To[top.del \to clear] \\
& \otimes R[state.repl \to idle]
\end{aligned}
$$

$$R[move@Frm, To \to Blk] \longleftarrow R[pickup@Blk \to Frm \otimes putdown@Blk \to To]$$

$$
\begin{aligned}
R[stack@N, BaseBlk \to nil] \longleftarrow\ & N > 0 \otimes R[move@Frm, BaseBlk \to TmpBlk \\
& \quad \otimes stack@N-1, TmpBlk \to nil]
\end{aligned}
$$

$$
\begin{aligned}
R[walk@Velocity \to nil] \longleftarrow\ & R[state.not \to holding] \otimes R[location \to Loc] \\
& \otimes R[state.repl \to moving \\
& \qquad \otimes velocity.repl \to Velocity \\
& \qquad \otimes startPos.repl \to Loc \\
& \qquad \otimes startTime.repl \to T] \\
& \otimes T = clock()
\end{aligned}
$$

$$
\begin{aligned}
R[location \to Loc] \longleftarrow\ & R[state \to moving;\ startPos \to L;\ startTime \to T;\ velocity \to V] \\
& \otimes Loc = L + V * (clock() - T)
\end{aligned}
$$

$$R[location \to Loc] \longleftarrow R[state.not \to moving] \otimes R[stopPos \to Loc]$$

$$
\begin{aligned}
R[stop \to Loc] \longleftarrow\ & R[location \to Loc] \otimes R[state.del \to moving] \\
& \otimes R[stopPos.ins \to Loc \otimes startPos.del \to S \otimes startTime.del \to T]
\end{aligned}
$$

Fig. 1. A Robot Simulation in Transaction F-logic

more kind of elementary update, *attr.repl*, which replaces an old value of the attribute *attr* with a new one. The rules in Figure 1 specify some of the possible behaviors of objects in class *robot*. The meaning of various components of these rules should be clear from the preceding discussion.

The notation $R[pickup@Blk \to Frm \otimes putdown@Blk \to To]$ means that the object R first executes the action *pickup* and then the action *putdown*, *i.e.*, it is a shorthand for $R[pickup@Blk \to Frm] \otimes R[putdown@Blk \to To]$. The rules for *pickup*, *putdown*, *move*, and *stack* work analogously to (2) of Section 4.

Rules 6 and 7 here define the method *location*, which is a pure query with no side effects. It calculates the location of the robot using its velocity and the information about the time interval this velocity was in effect. The action defined via method *walk* succeeds only if the robot initially is not holding anything (*i.e.*, it is either idle or moving). If the precondition holds, the action sets *state* to *moving* (which actually may not lead to any change, if the robot was already moving) and changes the velocity of the robot. It then queries the location and saves its current value using the attribute *startPos*, which records the location of the last change in velocity. It also calls the function *clock*() and records the time of the change.

The method *state.not* in Rules 5 and 7 is similar to elementary transitions, but is defined via the data oracle. This means that $R[state.not \rightarrow moving]$ is essentially a query to the current state whose intent is to verify whether $R[state \rightarrow moving]$ holds at that state. Recall that Transaction Logic is independent of the semantics of states and the data oracle serves as a gateway from the logic to the states. Interestingly, as far as Transaction Logic is concerned, the above program is Horn and monotonic, even though $R[state.not \rightarrow moving]$ is a negative query. The non-monotonic part here is delegated to the oracles, which should be able to handle such queries more efficiently. Therefore, the dynamic and the static aspects of the problem are handled separately: the dynamic part does not need to get bogged down in the details of how queries are evaluated at states and the static part does not need to be concerned with how states evolve.

Syntax and semantics. We shall now review the changes that need to be made to the syntax and semantics of our logics to obtain Transaction F-logic.

First the syntax. The above example suggests that to add dynamics to F-logic, we need to enrich F-logic with Transaction Logic's serial conjunction $\otimes$.

The other things that we need to model the dynamics are the two oracles. The data oracle's job is the same as in Transaction Logic, except that it would verify truth of F-logic formulas at states that are F-logic object bases. Transition oracles are also similar to those we saw earlier: they are mappings from pairs of states to sets of ground F-molecules that contain exactly one method expression in them. Examples of such F-molecules are:

$$john[boss.del \rightarrow bob]$$
$$mary[publications.ins@1995 \twoheadrightarrow \{title1, title2\}]$$

With appropriately defined oracles, executing the first of the above molecules should take us from the current database to one where John has no boss. Executing the other would result in the addition of a pair of titles to Mary's publication record for 1995.

It is interesting to note here that with F-logic's signature expressions (which were mentioned only in brief in this paper), we can have elementary transitions for updating schema information. For instance,

$$employee[boss.ins \Rightarrow manager]$$

would result in the addition of the type *manager* to the co-domain of the attribute *boss* in class *employee*. This ability to manipulate schema information in Transaction F-logic may provide a valuable logical framework in which schema evolution can be discussed.

The semantics of Transaction F-logic is also straightforward. We only need to define path structures for the new logic, which we call *path F-structures*. While in Transaction Logic path structures are mappings that assign classical

Herbrand structures to paths, path F-structures (in Transaction F-logic) map paths to Herbrand F-structures. The rest of the definitions (satisfaction in path F-structures, executional entailment, etc., go without change). The proof theory from [8] extends to Transaction F-logic with minimal, trivial changes.

Summary. We introduced a combined logic, called Transaction F-logic, which accounts for static as well as dynamic aspects of object-oriented databases. We also alluded to the possibility that this logic can be used both for defining schema evolution strategies and for reasoning about this evolution.

Above all, in our opinion, the most interesting observation about the exercise performed in this section is the ease with which Transaction Logic was transplanted from classical predicate calculus to F-logic. This flexibility was one of the main design goals for Transaction Logic.

6 Conclusions

This paper is an overview of our work aimed at the development of a unified logical foundation for deductive object-oriented databases and logic programming. The overall plot consists in amalgamating the two logics, F-logic and Transaction Logic into a single formalism, called Transaction F-logic.

F-logic was developed to account for the structural aspects of object-oriented languages, such as complex objects, IS-A hierarchies, typing, etc. Transaction Logic was designed to provide for the dynamics of objects in object-oriented systems. It also has applications in AI, discrete event simulation, heterogeneous databases, and more [8].

A prototype of a system based on F-logic was developed by Georg Lausen's group at the University of Freiburg. A prototype of Transaction Logic, developed by Tony Bonner's group, is available from the University of Toronto. Both prototypes can be tested via the *Foundations of DOOD* home page at

`http://www.cs.sunysb.edu/~kifer/dood/`

This page provides further information on our on-going efforts and has useful links to other related sites.

Even though this paper indicates that most of the fundamentals are already in place, the really hard problems are yet to be cracked. One such problem is query evaluation in the presence of inheritance. Since programs with inheritance are non-monotonic even in the Horn case, it is unclear which optimization strategies are most appropriate here. Furthermore, although the semantics for inheritance defined in [17] seems to be doing the right thing, it is a bit too procedural for our taste, and we are not convinced that this is "The One and Only True Way To Go."

Another issue concerns type checking in F-logic. F-logic has a rich type system (or, rather, a system of type constraints) that allows defining types via deductive rules. It has been argued in [17, 19] that the existing type systems are not sufficiently flexible for object-oriented deductive languages. However, generally, type-correctness in F-logic is undecidable, which defeats the purpose of having it in the first place. It is possible to define subsets of F-logic where the type system is decidable. However, the challenge is to find a sufficiently large and useful subset with this property.

Transaction Logic opens up another vast area for optimization problems. These range from intelligent caching of execution paths needed to avoid duplicate computation to source-code rewriting techniques (a là supplementary magic [32]) to techniques for reducing non-determinism. These optimizations also seem to have a lot of room for time/space trade-offs and related studies.

Acknowledgments. The blame for Transaction Logic and F-logic is fully borne by Tony Bonner, Georg Lausen, and James Wu with whom I had the pleasure to work. Many people share the blame for providing their much appreciated comments (see the relevant papers for the full text of indictment). The blame for inviting me to speak is on the organizers of DOOD-95 and the blame for the contents of this paper is squarely on the author.

References

1. S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD Conference on Management of Data*, pages 159–173, New York, 1989. ACM.
2. H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.
3. J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, July 1984.
4. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(4):445–473, 1991.
5. N. Arni, K. Ong, S. Tsur, and C. Zaniolo. LDL++: A second-generation deductive database system. Submitted for publication., 1994.
6. F. Bancilhon. A logic-programming/Object-oriented cocktail. *SIGMOD Record*, 15(3):11–21, September 1986.
7. A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.
8. A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, April 1995. *ftp://csri.toronto.edu/csri-technical-reports/323/report.ps.Z*.
9. A. Brogi, E. Lamma, and P. Mello. Objects in a logic programming framework. In A. Voronkov, editor, *First Russian Conference on Logic Programming*, number 592 in Lecture Notes in Artificial Inteligence, pages 102–113. Springer-Verlag, 1991.
10. W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.
11. W. Chen and D.S. Warren. C-logic for complex objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 369–378, New York, March 1989. ACM.

12. 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.
13. S. Greco and P. Rullo. Complex-Prolog: A logic database language for handling complex objects. *Information Systems*, 14(1):79–87, 1989.
14. D. Harel, D. Kozen, and R. Parikh. Process Logic: Expressiveness, decidability, completeness. *Journal of Computer and System Sciences*, 25(2):144–170, October 1982.
15. 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.
16. M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and schema. In *ACM SIGMOD Conference on Management of Data*, pages 134–146, New York, 1989. ACM.
17. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, May 1995.
18. M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 379–393, New York, March 1989. ACM.
19. M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. In *Intl. Symposium on Logic in Computer Science (LICS)*, pages 310–321, Amsterdam, The Netherlands, July 1991. Expanded version: TR 90/23 under the same title, Department of Computer Science, University at Stony Brook, July 1990.
20. M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77–120, August 1993.
21. E. Laenens, D. Sacca, and D. Vermeir. Extending logic programming. In *ACM SIGMOD Conference on Management of Data*, pages 184–193, New York, June 1990. ACM.
22. J.W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer-Verlag, 1987.
23. D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington D.C., August 1986.
24. W. Marek and Z. Pawlak. Information storage and retrieval systems: Mathematical foundations. *Theoretical Computer Science*, 1:331–354, 1976.
25. 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.
26. P. Mello and A. Natali. Objects as communicating Prolog units. In *ECOOP'87: European Conference on Object-Oriented Programming*, number 276 in LNCS, pages 181–192, Paris, June 1987. Springer-Verlag.
27. J. Meseguer. Multiparadigm logic programming. In *Algebraic and Logic Specifications*, number 632 in Lecture Notes in Computer Science, pages 158–200. Springer-Verlag, September 1992.
28. J. Meseguer and X. Qian. A logical semantics for object-oriented databases. In *ACM SIGMOD Conference on Management of Data*, pages 89–98, New York, May 1993. ACM.
29. Z. Pawlak. Mathematical foundations of information retrieval. In *Proceedings of Symposium on Mathematical Foundations of Computer Science*, pages 135–136,

High Tatras, Czechoslovakia, 1973.
30. 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.
31. D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 158 – 170. Morgan Kaufmann, San Francisco, CA, August 1993.
32. J.F. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 2*. Computer Science Press, Rockville, MD, 1989.
33. 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.
34. R.J. Wieringa. Formalization of objects using Equational Dynamic Logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Second Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 566 in Lecture Notes in Computer Science, pages 431–452, Munich, Germany, December 1991. Springer-Verlag.
35. C. Zaniolo. Object-oriented programming in Prolog. In *IEEE Symposium on Logic Programming (SLP)*, pages 265–270, February 1984.