Functional Programming Formalisms for OODB Methods

Gerd Hillebrand, Paris Kanellakis, and Sridhar Ramaswamy

Brown University, Providence, RI 02912, U.S.A.*

Abstract. Two well-studied functional formalisms in the theory of programming languages are (1) applicative program schemas and (2) typed lambda calculi. We relate these programming formalisms to object-oriented databases (OODBs) and in particular to the description of methods.

The language of method schemas (MS) is a programming formalism based on applicative program schemas with additional key object-oriented features such as classes, methods, inheritance, name overloading, and late binding. From [4], we present its syntax and semantics and survey the state-of-the-art of consistency checking or signature inference for this language, a problem which can be used in studying database schema evolution. We then relate MS with more conventional database query languages by showing that its expressive power over finite ordered databases is PTIME.

Despite its simplicity and applicability, MS does not directly model the tuple, set, and list complex structures that are quite common in databases. Also, it does not treat functions as objects, i.e., methods are different from objects. It is possible to achieve these two capabilities using the typed lambda calculus with equality ($TLC^=$) as a database query language, even without any object-oriented features. From [25], we illustrate how this pure functional language subsumes most conventional database query languages including the relational calculus/algebra, Datalog (with or without negation), and the complex object calculus/algebra (with or without powerset).

In conclusion, we argue that the appropriate programming formalism for OODBs must be a functional language that combines the object-oriented MS with the expressive $TLC^=$ and facilitates operations on sets of objects.

1 Introduction

Object-oriented databases (OODBs) represent a new generation of database technology, which has developed through a combination of advanced programming language features and successful relational/network database techniques, see [9, 30, 44]. Both programming language theory and database theory have

*Research supported by ONR Contract N00014-91-J-4052, ARPA Order 8225.

been proposed as starting points for OODB formal foundations. They are both legitimate starting points, but one should be aware that a naive synthesis of such distinct theories is not necessarily harmonious. For example, it is largely an open question how to match distinct programming paradigms in one language.

In this paper we explore the problem "what is the appropriate formal model for OODBs" from a functional perspective. We view the organization of data into *objects with methods* as the core new feature of OODBs. This is a function-centric (or method-centric) view of data as opposed to the structure-centric view of data in relational databases (e.g., in [28, 41]).

Interesting functional approaches to database query languages have been advocated in the past, e.g., [15, 38], but much remains to be done in this area. Because of the function-centric character of OODBs we believe that this is one of the most promising approaches to the problem of their formalization. Note that some of the actual query languages in OODB prototypes are functional [9] and that recent research on complex-objects and types has emphasized functional models [13, 14].

Two well-studied functional formalisms in the theory of programming languages are (1) applicative program schemas [21, 23] and (2) typed lambda calculi [19, 11, 12]. We relate these two programming formalisms to OODBs and in particular to the description of OODB methods.

In order to accomplish this: (1) We modify applicative program schemas into an OODB formalism called *method schemas* (MS) in [4]. We show that method schemas (over ordered databases) express exactly the PTIME queries. (2) We present the typed lambda calculus as a formalism for expressing a wide range of relational and complex-object database queries [25]. In this case, the elementary time queries (which properly contain the PTIME queries) are expressible, because of a limited ability to create and manipulate list structures. So, in terms of expressive power, the typed lambda calculus is the more general (even if not the more convenient) framework.

By our overview of the approaches in [4, 25] and our new characterization of the expressive power of method schemas we demonstrate how *many notions from relational and object-oriented databases can be formulated, with no loss of generality, in pure functional form.* More specifically, we proceed as follows.

MS is a simple programming formalism for OODBs, based on applicative program schemas with additional features such as *classes*, *methods*, *inheritance*, *name overloading*, and *late binding*. In Section 2.1, we explain the syntax and semantics of method schemas and motivate them with an OODB example. Method schema consistency is a form of "type inference," which involves deciding at compile-time whether a given method schema can lead to an inconsistency at run-time in some interpretation. A more precise term for this problem would be *signature inference*. The incremental version of consistency checking for method schemas is an algorithmic formalization of the database schema evolution problem [4, 10, 26, 39, 43, 45]. In Section 2.2, we survey the state-of-the-art on consistency checking. We use various algorithmic techniques and techniques from the theory of program schemas, e.g., [8, 33].

In Section 3, we examine the expressive power of MS by relating it to the more conventional formalisms of relational calculus [20] and Datalog with negation [18, 31, 6]. For finite ordered interpretations this is a functional characterization of PTIME (analogous to the logical characterizations of [27, 42]). A different functional characterization in terms of equations is presented in [24]. Note also that, Datalog with negation has been used in [5] to provide semantics to a number of variations of method schemas. In Section 3.1 we outline the input-output conventions, in Section 3.2 we argue that every PTIME query is expressible, and in Section 3.3 that only PTIME queries are expressible.

Our expressive power result indicates that MS is a syntactic variation on the very robust concept of PTIME database queries that is attractive because it incorporates many additional desirable object-oriented features. There are some disadvantages, however. Method schemas are an example of a practical functional framework but do not treat functions as objects, i.e., methods are different from objects. To achieve this uniformity one has to examine higher-order frameworks related to the lambda calculus. Another problem is that, as in Datalog, complex structures (such as sets, tuples or lists) are not directly part of the framework but have to be encoded (into relations for Datalog, into the class hierarchy and methods for MS). In typed lambda calculi we can use a simple type discipline to encode complex structures in a far more direct fashion. The notion of types here is structural and quite different from the notion of classes.

In Section 4, we present some basic background on the syntax, the operational semantics, and the types of the simply typed lambda calculus with equality ($TLC^{=}$). We refer to [25] for how to remove equality from this calculus. For the problem of type inference we refer to [29].

It is possible to "naturally embed" in the typed lambda calculus with equality many database query languages including the relational calculus/algebra [20], Datalog (with or without negation) [18, 31, 6], and the complex object calculus/algebra (with or without powerset) [1, 3]. We consider embeddings such that for each query expressed there is a lambda term which when applied to the input database (which is a list-of-tuples lambda term) normalizes to the output database; most importantly, if the query expressed is in PTIME then the normal form can be computed in a number of reduction steps polynomial in the size of input database.

In Section 5.1 we discuss the input-output conventions for representing sets, tuples and lists, as well as the important concept of list iteration. In Section 5.2 we present, as an extended example, the embedding of the relational algebra. Our analysis of the expressive power of the simply typed lambda calculus is based on the analysis of reduction sequences of [40, 34].

We refer to [25] for the more advanced fixpoint and powerset features. Note that these features allow the embedding of higher-order queries, e.g., the second-order queries of [22, 17]. Note that we only use the very core of programming language type systems, i.e., only simple types without any polymorphism. It is interesting that the (otherwise very convenient) use of method names in method schemas and their recursive definition can all be simulated using the fixpoints of [25] in the typed lambda calculus.

The embeddings we present form the first step in understanding the connections between functional programming and databases. Given the emphasis of database theory on efficiency, we can restrict ourselves to PTIME queries. For PTIME queries we present here a method schema framework (Section 3) and the basis for a lambda calculus framework (Section 5 and [25]). For other functional characterizations of PTIME we refer to [24, 32].

In conclusion, in Section 6 we comment on the possible combination of MS with $\text{TLC}^=$ as well as some open research questions.

2 Method Schemas (MS)

2.1 Syntax and Operational Semantics

We assume the existence of the following disjoint countable sets of symbols: of *classes* $(c_1, c_2, \ldots)$ and of *methods* $(m_1, m_2, \ldots)$. For each method m, the arity of m is an integer greater or equal to zero. A *signature* is an expression $c_1, \ldots, c_{k-1} \rightarrow c_k$, where each c_i, for $1 \leq i \leq k$, is a class. For $k = 1$, the expression is $\rightarrow c_1$. In this formalism the signatures play the role of "types" (but since the word "type" has been misused in many different contexts we prefer to use the more precise term signature).

A *base definition* of m at $c_1, \ldots, c_{k-1}$, for $k \geq 1$ is a pair $(m, (c_1, \ldots, c_{k-1} \rightarrow c_k))$, where m is a method of arity $k-1$ and the remaining part is a signature. (For $k = 1$, the method m is zero-ary.) A *coded definition* of m at $c_1, \ldots, c_k$, for $k \geq 1$ is a triple $(m, (c_1, \ldots, c_k), t)$, where m is a method of arity k, for $1 \leq i \leq k$, each c_i is a class, and t is a k-term (i.e., k-terms are our programming formalism).

A *k-term* for some $k \geq 1$ is a finite rooted ordered directed acyclic graph (dag) such that: (1) each vertex is uniquely labeled either by a method or by an integer in $\{1, \ldots, k\}$; (2) each vertex labeled by a method of arity j has j children which are ordered from left to right; (3) all vertices of out-degree zero are called *inputs* and each vertex labeled with an integer is an input; (4) the root is called the *output*; (5) the dag comes with a uniquely defined depth first search order going from left to right because the children of each vertex are ordered.

For example, the terms t and t' in Figure 1 are 1-terms. They are, by definition, also 2-terms, 3-terms, etc. The idea is that k-terms represent functions of k arguments, where a copy of argument i is placed at each input labeled i. Vertices can also have outdegree zero and be labeled by zero-ary methods, i.e., they are constant inputs.

As an example, consider the following definitions:

$$(\mathit{Price}, (\mathit{Basepart} \rightarrow \mathit{Int}))$$
$$(\mathit{Cost}, (\mathit{Basepart}), s)$$
$$(\mathit{Cost}, (\mathit{Part}), t)$$

Let s be a single arc with tail labeled *Price* and head labeled 1, and t be as in Figure 1. These definitions can be represented syntactically using an intuitive lambda notation—instead of the graphical k-term notation—as shown below.

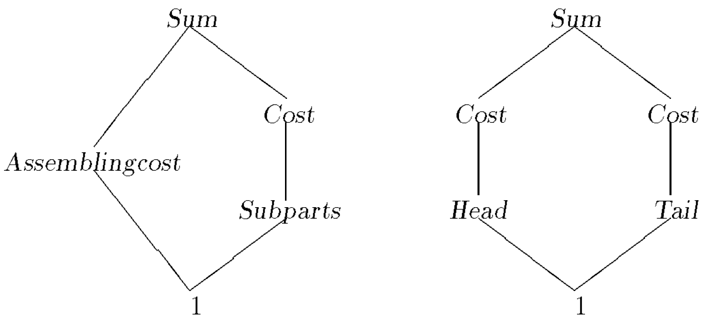

Fig. 1. Term t and t'

Note that we use the symbol colon (:) for representing base methods and the symbol equal (=) for coded methods.

$$
\begin{aligned}
&\mathit{Price} \ @ \ \mathit{Basepart} : \ \mathit{Int} \\
&\mathit{Cost} \ @ \ \mathit{Basepart} = \lambda x.\, \mathit{Price}(x) \\
&\mathit{Cost} \ @ \ \mathit{Part} \quad\ \ = \lambda x.\, \mathit{Sum}(\mathit{Assemblingcost}(x), \mathit{Cost}(\mathit{Subparts}(x)))
\end{aligned}
$$

Definition 1. A *method schema* is a 4-tuple $(C, \leq, \Sigma_0, \Sigma_1)$ where:

1. C is a finite set of classes and $\leq$ is a forest on C. We say that $c \leq c'$ if c is a descendant of c' in this forest. The partial order defined by $\leq$ is called the *Isa* relation.
2. Σ_0 is a set of base definitions with classes from C.
3. Σ_1 is a set of coded definitions with classes from C. Let M_0 be the set of methods defined in Σ_0 and M_1 be the set of methods defined in Σ_1. We require that $M_0 \cap M_1 = \emptyset$ and that the methods that appear in k-terms of Σ_1 are from $M_0 \cup M_1$.
4. Each method of arity k has at most one definition at $c_1, \ldots, c_k$ for each $c_1, \ldots, c_k$.

Example 1. Consider the class hierarchy of Figure 2 and the terms t and t' in Figure 1. Method definitions are presented in Figure 3.

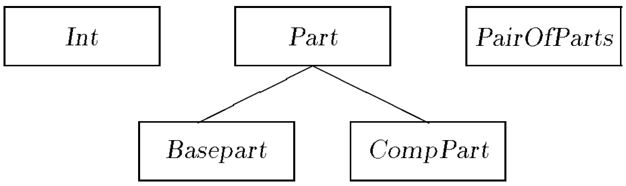

Fig. 2. An example *Isa* hierarchy

Let us briefly survey some important concepts in connection with method schemas.

$$
\begin{array}{llll}
\mathit{Sum} & @ & \mathit{Int}, \mathit{Int} & : \mathit{Int} \\
\mathit{Head} & @ & \mathit{PairOfParts} & : \mathit{Part} \\
\mathit{Tail} & @ & \mathit{PairOfParts} & : \mathit{Part} \\
\mathit{Price} & @ & \mathit{Basepart} & : \mathit{Int} \\
\mathit{Subparts} & @ & \mathit{Part} & : \mathit{PairOfParts} \\
\mathit{Assemblingcost} & @ & \mathit{Part} & : \mathit{Int} \\
\mathit{Cost} & @ & \mathit{Part} & = t \\
\mathit{Cost} & @ & \mathit{Basepart} & = \lambda x.\, \mathit{Price}(x) \\
\mathit{Cost} & @ & \mathit{PairOfParts} & = t'
\end{array}
$$

Fig. 3. Methods

Syntactically restricted families of method schemas: A schema is *monadic* if all its methods have arity 1. A schema is *recursion-free* if there is no directed cycle in the *method dependence graph*, where the vertices of the method dependence graph are the coded methods and there is an arc from m to n iff m occurs in some coded definition of n. (Note that, our definition of recursion-freeness is a reasonable syntactic way of avoiding recursion, but not necessarily the only one.) A schema is *simple* if only base methods occur in its coded definitions. Note that simple schemas are an important kind of recursion-free schemas.

For simple schemas, *covariance* is the following constraint on the signatures of base methods. A simple schema is covariant if for each m and for each pair

$$(m, (c_1, \ldots, c_k \to c)) \ ,$$
$$(m, (c'_1, \ldots, c'_k \to c'))$$

of definitions of a base method m, we have that $(\forall i\ c'_i \le c_i) \Rightarrow c' \le c$.

Method Inheritance: A method schema $(C, \le, \Sigma_0, \Sigma_1)$ contains definitions of methods, i.e., Σ_0, Σ_1, which we call *explicit*. It is also used to define methods implicitly, through *inheritance*. This is for the convenience of code reuse. Therefore, in addition to the explicit definitions for a method, definitions are implicitly inherited along the class hierarchy. If a method m is explicitly defined at classes $c'_1, \ldots, c'_k$, then its definition implicitly applies at $c_1, \ldots, c_k$ where $c_i \le c'_i$ for $i = 1, \ldots, k$.

Name overloading: Method inheritance implies that we can have several definitions for the same base or coded method at the same classes—although by condition 4 of the method schema definition there can be only one explicit definition. This creates overloading of names. To illustrate overloading, consider the method *Cost* in the example. That method is explicitly defined on *Part*, implicitly inherited by *Basepart* and *CompPart*, and explicitly redefined on *Basepart*; it is also explicitly defined for *PairOfParts*.

Resolution of name overloading: This consists of assigning to a given method and given classes, a unique definition. The resolution of a method m at some

$c_1, \ldots, c_k$ is the explicit definition of m for the "component-wise smallest" tuple $c'_1, \ldots, c'_k$ at which m has an explicit definition and $c_i \leq c'_i$, for $i = 1, \ldots, k$ (if such a unique component-wise minimum exists). For example, *Cost* at *CompPart* is resolved to be the explicit definition from *Part*, whereas *Cost* at *Basepart* is *Basepart*'s explicit definition. If m has a resolution at $c_1, \ldots, c_k$, we say that m is *well defined* at $c_1, \ldots, c_k$. Otherwise, we say that m is *undefined* at $c_1, \ldots, c_k$. Non-definition can come either (1) because there is no definition of m above $c_1, \ldots, c_k$ or (2) because there is ambiguity of definitions above $c_1, \ldots, c_k$.

Example 2. Let c_1, c_2 be classes with $c_1 \leq c_2$ and consider the schema with explicit base definitions of m at c_1, c_2 and c_2, c_1. Then m is undefined at c_2, c_2 (no definition) and m is undefined at c_1, c_1 (ambiguity of definitions). For methods of arity 1, ambiguity (2) cannot arise because of the forest hierarchy, but we can have non-definition because of (1).

Object assignments: We assume the existence of a countably infinite set of *objects* $(o_1, o_2, \ldots)$ disjoint from classes and methods. We use object assignments [2] to provide semantics for classes with inheritance. The semantics of base methods is defined using partial functions satisfying certain signature constraints.

Definition 2. Given a method schema $(C, \leq, \Sigma_0, \Sigma_1)$, a *disjoint object assignment* ν for C is a total function from C to finite sets of objects such that distinct classes are mapped to disjoint sets of objects (sets with pairwise empty intersections). For each c, we define $\nu(c*) = \bigcup_{c_1 \leq c} \nu(c_1)$. We will refer to $\bigcup_{c \in C} \nu(c)$ as the set of objects in ν.

Definition 3. An *interpretation* of a schema $S = (C, \leq, \Sigma_0, \Sigma_1)$ is a pair $I = (\nu, \mu)$ where ν is a disjoint object assignment for C and μ a total function from the base methods M_0 in S to partial functions such that:

1. If m has arity k, $\mu(m)$ is a partial function from k-tuples of objects in ν to objects in ν.
2. For each $c_1, \ldots, c_k$, if m is undefined at $c_1, \ldots, c_k$, then $\mu(m)$ is undefined everywhere in $\nu(c_1) \times \cdots \times \nu(c_k)$. Otherwise, let the resolution of m at $c_1, \ldots, c_k$ be $(m, (c'_1, \ldots, c'_k \rightarrow c))$ where $c_i \leq c'_i$ for $i = 1, \ldots, k$. Then we have that $\mu(m)|_{\nu(c_1) \times \cdots \times \nu(c_k)}$ is a total function into $\nu(c*)$.

Intuitively, every class c is populated by a set of objects $\nu(c*)$, some of which are in this class exclusively (those objects that are in $\nu(c)$) and the rest belong to its "subclasses" c_1, where $c_1 \leq c$ and $c_1 \neq c$. When a base method is defined at a class c_1 with signature $c_1 \rightarrow c$ (or is inherited at c_1 with signature $c'_1 \rightarrow c$ such that $c_1 \leq c'_1$), then its meaning at c_1 is a total function from $\nu(c_1)$ to $\nu(c*)$. That is, given an object exclusively in c_1, return an object in c.

The semantics of coded methods is defined using the above semantics of base methods and a rewriting technique. Without recursion, this rewriting is equivalent to function composition.

Late binding: This is a result of the operational definition of rewriting. Let us first give some intuition for the rewriting of coded methods. Consider a k-term that is a single arc. Let its one internal node be labeled *Cost* and its input be replaced by an object o, as a "procedure call" to *Cost*. Based on the class of its argument, we can replace *Cost* either by some object/code if it is defined in a base/coded fashion, or by an error message if it is undefined. In general, we reduce the first method (first in the depth first ordering of the k-term we are processing) that has instantiated leaves as children. We continue this processing until we either obtain a result (i.e., an object), or reach an inconsistency. We might also not terminate. A partial sequence of rewritings for Example 1 is shown in Figure 4, where o is in class *PairOfParts* and o', o'' are in class *Basepart*.

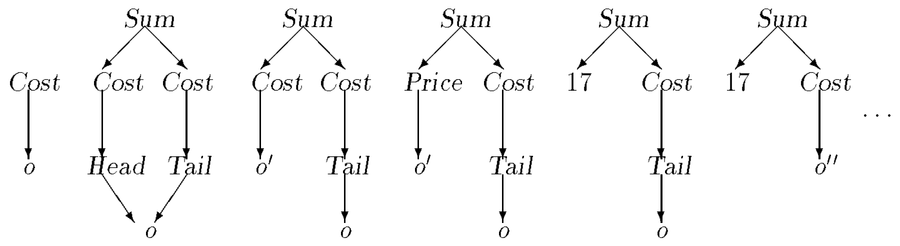

Fig. 4. A reduction sequence

More formally, an *instantiated term* in interpretation $I = (\nu, \mu)$ is a k-term, whose integer labels have been replaced by objects. Let $c_1, \ldots, c_k$ be classes, m be a method of arity k, and for each i let $o_i \in \nu(c_i)$. Then the instantiated term consisting of a single vertex labeled m and k inputs labeled respectively $o_1, \ldots, o_k$ is said to be a *redex*. We define the *first reducible vertex* to be the uniquely determined first vertex in the depth first ordering of an instantiated term, which is the root of a redex.

Definition 4. Let I be an interpretation of method schema S, t be an instantiated term in I, w be the first reducible vertex of t, and m be the label of w. Let $o_1, \ldots, o_k$ be the labels of the children of w belonging respectively to classes $c_1, \ldots, c_k$. The *reduction* $I(t)$ of t in I is obtained as follows:

1. If m is undefined at $c_1, \ldots, c_k$ then $I(t)$ is a special symbol $\perp$.
2. If m is a base method well defined at $c_1, \ldots, c_k$, let its resolution there be $(m, (c'_1, \ldots, c'_k \to c))$. (This implies that $c_i \leq c'_i$ for $i = 1, \ldots, k$.) Then $I(t)$ is the instantiated term obtained by removing from w the outgoing edges and changing its label to $\mu(m)(o_1, \ldots, o_k)$. $\mu(m)$ is a total function from $\nu(c_1) \times \cdots \times \nu(c_k)$ into $\nu(c*)$. Therefore, $\mu(m)(o_1, \ldots, o_k)$ yields an object o such that $o \in \nu(c*)$.
3. If m is a coded method well defined at $c_1, \ldots, c_k$, then $I(t)$ is the instantiated term obtained by substituting, with the natural renaming of inputs, the vertex w by the dag s, where (m, σ, s) is the resolution of m at $c_1, \ldots, c_k$.

When I is clear from the context, we denote the reduction of t to $I(t)$ as $t \to I(t)$ and a finite sequence of reductions from t to t' as $t \xrightarrow{*} t'$.

2.2 Consistency and Signature Inference

We are interested in the rewriting sequences of a particular set of instantiated terms, i.e., these are the initial "procedure calls" that make sense according to a method schema. More formally we have the following: Let S be a method schema with interpretation $I = (\nu, \mu)$, $c_1, \ldots, c_k$ be classes in S, m a method of arity k well defined at these classes, and for each i let $o_i \in \nu(c_i)$. Then the instantiated term consisting of a single vertex labeled m and k inputs labeled respectively $o_1, \ldots, o_k$ is said to be a *start-redex*. We have an *inconsistency* in I if for some start-redex t we have $t \xrightarrow{*} \perp$.

Definition 5. A method schema S is *consistent* if for each interpretation I of S it is impossible to rewrite any start-redex into $\perp$ in a finite number of steps.

A variety of questions are important in this setting, e.g.: (1) Is a schema consistent? (2) Are there possible/certain diverging computations? (3) Is method m (at c) possibly/certainly "reachable" from m' (at c') [26]? We concentrate here on (1) although most of the techniques that we develop can be used for studying other problems such as (2) or (3). Two properties simplify consistency checking: monadic arity and absence of recursion.

In an object-oriented context, it is not reasonable to recompile all methods each time the schema is updated. The problem is to obtain an *incremental* consistency checking algorithm that would avoid redoing the same verifications/computations. A solution that is adopted practically (e.g., [9]) is to maintain a dependency graph of the methods and recompile only methods that may have been affected by the update. Of course, understanding what may be affected is the crucial part. We briefly summarize some results from [4] (on what may be affected) and some open questions. Let n be the size of method definitions in the input method schema and c the size of the class hierarchy.

In the general case of polyadic recursive schemas, consistency is undecidable. Both polyadic methods and recursion are necessary for this result. The proof is by reduction from results in [33]. This undecidability result holds for schemas with methods of arity less than or equal to 3. The decidability of method schema consistency with methods of arity 2 is open.

In the case of monadic schemas, the set of possible computations can be described using a context-free language. An inconsistency may be reached if this language has a non-empty intersection with a particular regular language. To handle overloading, one can use a modification of a technique of [8]. The decision procedure runs in $O(nc^3)$ time.

In the case of monadic and *recursion-free schemas,* checking a whole schema for consistency is logspace-complete in PTIME. This indicates that very efficient incremental solutions for even the simplest syntactic cases may be hard to obtain. One can focus on the key recursion-free case of a single coded method in the context of base methods. Checking a single coded method for consistency is logspace-complete in NLOGSPACE and can be done more efficiently than the recursive case above using finite state automata techniques (in $O(nc^2)$ time). Inheritance and overloading introduce some nondeterminism in the automata theoretic approach to consistency checking, but covariant signatures remove it. Consistency of a single coded method, assuming covariance, is in DLOGSPACE and can be checked in $O(n + c)$ time using a radix-tree data structure. We do not know whether this radix-tree data structure can be extended to solve the consistency of monadic, covariant, recursion-free schemas. We also do not know whether there are efficient incremental algorithms for any of the consistency problems. Since we know that the problem of consistency checking for monadic, recursion-free schemas is PTIME-complete, it would be interesting to know if there are good incremental algorithms for the simple, monadic or the covariant, monadic cases.

In the case of recursion-free schemas, the consistency problem is coNP-complete for a single coded method with two arguments. Some special cases can be shown to be in PTIME, using tree automata techniques. Covariance does not help in the recursive case. However, in the recursion-free covariant case there is a PTIME test for a fixed arity coded method. This is interesting in practice, because it motivates a heuristic for the general case.

In some proposals the class hierarchy is a dag and one has multiple inheritance. Also, in some proposals, it is required that the first argument, called the receiver, of a method must determine the resolution of name overloading. Each instance of a method is then viewed as “attached” to the class of the first argument. We believe that once the resolution mechanism is determined, the above techniques and results are directly applicable also to multiple inheritance and attachment to first argument.

3 The Expressive Power of Method Schemas

When considering method schemas in the context of finite databases, it is natural to ask what kind of queries can be expressed in the method schema framework. Of course, since method schemas are just *schemas*, one has to agree on the interpretation of the base methods in order to make this question precise. In this section, we describe a very simple way of encoding relations as interpreted base methods, and we show that under this interpretation, method schemas express exactly the PTIME queries. We assume some familiarity with Datalog terminology and notation from [28, 41], e.g., rules, EDBs, IDBs etc.

3.1 I/O Conventions

We consider queries over finite ordered relational databases. The input of a query consists of a finite ordered universe U (w.l.o.g an initial segment of the positive integers) and some relations $P, Q, R, \ldots$ over U, and the output is another relation O over U. The arities and attribute names of the input and output relations are given by some fixed relational schema $\mathcal{S}$.

We encode these data in the method schema framework as follows. The elements of the universe are represented by objects of class *Number* and the ordering among them is given by a base method *Pred* @ *Number*: *Number* which computes the predecessor function. The initial element of the universe is represented by an object of class *Zero*, which is a subclass of *Number*. We capture the finiteness of the universe by providing its size as part of the input. More precisely, we provide as part of the input an object N of class *Number* such that N does not appear in the range of *Pred*. We assume that there is a unique object of class *Zero* and that there is at least one non-*Zero* element in the universe. When our interpretations satisfy these semantic conditions we will say that our language is *MS with order*.

A k-ary relation R is represented by a k-ary base method r of signature $r@\,\mathit{Number},\ldots,\mathit{Number}\colon \mathit{Number}$, such that $r(x_1,\ldots,x_k)$ returns an object of class *Number* if the tuple represented by $(x_1,\ldots,x_k)$ belongs to R and an object of class *Zero* if it does not. We will frequently write $r(\vec{x})$ instead of $r(x_1,\ldots,x_k)$ if the arity of the method does not matter.

For example, to encode the relation $R=\{(1,1),(1,2)\}$ over the universe $U=\{0,1,2\}$, we would populate the classes *Number* and *Zero* as $\mathit{Number}=\{1,2\}$ and $\mathit{Zero}=\{0\}$, we would interpret *Pred* as $\mathit{Pred}(2)=1$, $\mathit{Pred}(1)=\mathit{Pred}(0)=0$, and we would interpret r as $r(1,1)=r(1,2)=1$, $r(x,y)=0$ for all other x,y.

Queries are represented by coded methods q with a signature of the form $q@\,\mathit{Number},\ldots,\mathit{Number}\colon \mathit{Number}$. If $q(\vec{x})$ reduces to an object of class *Number*, then the tuple represented by $\vec{x}$ is considered part of the output; if $q(\vec{x})$ reduces to an object of class *Zero*, then the tuple represented by $\vec{x}$ is not considered part of the output.

3.2 Expressing PTIME Queries

We show that under the input/output conventions given above, method schemas can compute inflationary fixpoints of Datalog$^{\neg}$ programs (see [28] for a survey of these concepts). The results of Immerman [27] and Vardi [42] then imply that method schemas can compute all PTIME queries over finite ordered databases.

To begin with, let us illustrate how to code simple Boolean and arithmetical functions. We represent the truth values *True* and *False* by the classes *Number* and *Zero*. Any object of class *Number* (N, for example) represents *True*, and the single object 0 of class *Zero* represents *False*. (Note that we can compute 0 as $0=\mathit{Pred}^*(N)$, where $\mathit{Pred}^*\,@\,\mathit{Number}=\lambda x.\,\mathit{Pred}^*(\mathit{Pred}(x))$ and $\mathit{Pred}^*\,@\,\mathit{Zero}=\lambda x.\,x$.) We will often informally say that a method returns *True* when it returns a *Number* object, or *False* when it returns the *Zero* object.

The Boolean functions can be coded as:

$$
\begin{array}{lcll}
\mathit{Not} & @ & \mathit{Number} & =\lambda x.\,0 \quad (\text{where } 0{:}=\mathit{Pred}^*(N))\\
\mathit{Not} & @ & \mathit{Zero} & =\lambda x.\,N\\
\mathit{And} & @ & \mathit{Zero},\mathit{Zero} & =\lambda xy.\,0\\
\mathit{And} & @ & \mathit{Zero},\mathit{Number} & =\lambda xy.\,0\\
\mathit{And} & @ & \mathit{Number},\mathit{Zero} & =\lambda xy.\,0\\
\mathit{And} & @ & \mathit{Number},\mathit{Number} & =\lambda xy.\,N \quad (\text{and similar for } \mathit{Or})\\
\mathit{Equal} & @ & \mathit{Zero},\mathit{Zero} & =\lambda xy.\,N
\end{array}
$$

$$\begin{aligned}
&\mathit{Equal} \mathbin{@} \mathit{Zero}, \mathit{Number} &&= \lambda xy.\, 0\\
&\mathit{Equal} \mathbin{@} \mathit{Number}, \mathit{Zero} &&= \lambda xy.\, 0\\
&\mathit{Equal} \mathbin{@} \mathit{Number}, \mathit{Number} &&= \lambda xy.\, \mathit{Equal}\big(\mathit{Pred}(x), \mathit{Pred}(y)\big)
\end{aligned}$$

We also need to extend the *Pred* function to k-tuples of objects. Below, we define methods Pred_i^k such that $\mathit{Pred}_i^k(x_1, \ldots, x_k)$ is the i^{th} component of the lexicographical predecessor of the tuple $(x_1, \ldots, x_k)$, for $1 \le i \le k$.

$$\begin{aligned}
&\mathit{Pred}_i^k \mathbin{@} \overbrace{\mathit{Number}, \ldots, \mathit{Number}}^{i-1}, \overbrace{\mathit{Zero}, \ldots, \mathit{Zero}}^{k-i+1} = \lambda x_1 \ldots \lambda x_k.\, N\\
&\mathit{Pred}_i^k \mathbin{@} \overbrace{\mathit{Number}, \ldots, \mathit{Number}}^{i}, \overbrace{\mathit{Zero}, \ldots, \mathit{Zero}}^{k-i} = \lambda x_1 \ldots \lambda x_k.\, \mathit{Pred}(x_i)\\
&\mathit{Pred}_i^k \mathbin{@} \overbrace{\mathit{Number}, \ldots, \mathit{Number}}^{k} \qquad\qquad\quad = \lambda x_1 \ldots \lambda x_k.\, x_i \quad (i \ne k)
\end{aligned}$$

With this machinery in place, we can now code Datalog$^\neg$ programs. For our purposes, we can think of a Datalog$^\neg$ program as a relational expression $E(O; P, Q, R, \ldots)$ involving a designated output relation O, input relations P, Q, R, $\ldots$, and the operators *Union*, *Intersection*, *Complement*, *Times*, *Select*, and *Project*. The semantics of the program are given by the least fixpoint of the mapping $O \mapsto O \cup E(O; P, Q, R, \ldots)$. It is well known that the least fixpoint can be computed by iterating this mapping n^k times, where n is the size of the universe and k is the arity of O.

The relational operators can be coded as follows. Assume that r and s code relations R and S as described above, then:

$$\begin{aligned}
&\mathit{Union}(r, s) &&\equiv \lambda \vec{x}.\, \mathit{Or}\big(r(\vec{x}), s(\vec{x})\big)\\
&\mathit{Intersection}(r, s) &&\equiv \lambda \vec{x}.\, \mathit{And}\big(r(\vec{x}), s(\vec{x})\big)\\
&\mathit{Complement}(r) &&\equiv \lambda \vec{x}.\, \mathit{Not}\big(r(\vec{x})\big)\\
&\mathit{Times}(r, s) &&\equiv \lambda \vec{x} \lambda \vec{y}.\, \mathit{And}\big(r(\vec{x}), s(\vec{y})\big)\\
&\mathit{Select}_{i=j}(r) &&\equiv \lambda \vec{x}.\, \mathit{And}\big(r(\vec{x}), \mathit{Equal}(x_i, x_j)\big)
\end{aligned}$$

Projection is a bit more difficult, because it involves quantifier elimination. The projection of R onto its first $k-1$ columns is encoded as:

$$\mathit{Project}(r) \equiv \lambda x_1 \ldots \lambda x_{k-1}.\, r'(x_1, \ldots, x_{k-1}, N) \ ,$$

where $r'(x_1, \ldots, x_{k-1}, y)$ is *True* iff there exists a $z \in \{0, \ldots, y\}$ such that $r(x_1, \ldots, x_{k-1}, z)$ is *True*. r' can be coded as follows:

$$\begin{aligned}
&r' \mathbin{@} \overbrace{\mathit{Number}, \ldots, \mathit{Number}}^{k-1}, \mathit{Zero} \qquad = \lambda x_1 \ldots \lambda x_{k-1} \lambda y.\, r(x_1, \ldots, x_{k-1}, y)\\
&r' \mathbin{@} \overbrace{\mathit{Number}, \ldots, \mathit{Number}}^{k-1}, \mathit{Number} =\\
&\quad \lambda x_1 \ldots \lambda x_{k-1} \lambda y.\, \mathit{Or}\big(r(x_1, \ldots, x_{k-1}, y), r'(x_1, \ldots, x_{k-1}, \mathit{Pred}(y))\big)
\end{aligned}$$

Finally, we have to show how to compute fixpoints. Let $E(O; P, Q, R, \ldots)$ be a relational expression. Let k be the arity of O, let o be the base method representing O, and let e be the coded method corresponding to E. Then the fixpoint of E with respect to O is given by:

$$\mathit{Fix}(o, e) \equiv \lambda x_1 \ldots \lambda x_k.\, o'(x_1, \ldots, x_k, \overbrace{N, \ldots, N}^{k}) \ ,$$

where $o'(\vec{x}, \vec{n})$ is *True* iff $\vec{x}$ is in the image of the $\vec{n}^{\text{th}}$ iterate of the mapping $O \mapsto O \cup E(O; P, Q, R, \ldots)$, $\vec{n}$ being interpreted as a base-$(N+1)$ number. o' can be defined as follows:

$$\begin{aligned}
&o' \ @ \ \mathit{Number}, \ldots, \mathit{Number}, \overbrace{\mathit{Zero}, \ldots, \mathit{Zero}}^{k} \qquad\quad = \lambda\vec{x}\lambda\vec{n}.\, 0 \\
&o' \ @ \ \mathit{Number}, \ldots, \mathit{Number}, \overbrace{\mathit{Number}, \ldots, \mathit{Number}}^{k} = \\
&\quad \lambda\vec{x}\lambda\vec{n}.\, \mathit{Or}\big(o'(\vec{x}, \mathit{Pred}_1^k(\vec{n}), \ldots, \mathit{Pred}_k^k(\vec{n})), \bar{e}(\vec{x})\big)
\end{aligned}$$

where the method $\bar{e}$ arises from e by replacing every occurrence of $o(\vec{x})$ in e by $o'(\vec{x}, \mathit{Pred}_1^k(\vec{n}), \ldots, \mathit{Pred}_k^k(\vec{n}))$.

Thus, for every Datalog$^\neg$ program (interpreted in an inflationary fashion over an ordered input) one can construct a method schema with order (under the input-output conventions of Section 3.1) that computes the same query. First note that this method schema is consistent. Also note that, under the input-output conventions of Section 3.1, there is an easy to see one-to-one correspondence of finite ordered relational databases and method schema with order interpretations. We can conclude that,

Lemma 6. *Under the input output conventions of Section 3.1, MS with order can express every PTIME query.*

3.3 Simulating Method Schemas in Datalog

In this section, we show that the expressive power of method schemas over ordered finite interpretations is no more than that of Datalog$^\neg$ over ordered input. This, combined with the previous reduction, allows us to deduce that method schemas express exactly the PTIME queries.

We will first explain the intuition behind the simulation. We capture the input/output characteristics of a method of arity k by means of a predicate of arity $k+1$. The understanding is that the first k arguments of the predicate are the inputs of the method and the $(k+1)$th argument is the output. Base methods correspond naturally to EDBs and coded methods to IDBs. The disjoint object assignment to classes in an interpretation of a method schema is simulated by means of a unary EDB predicate. The fact that the input is ordered is captured in the Datalog simulation by means of a special interpreted binary EDB predicate that orders the input.

Two technical problems arise with this approach. The first is that there will be input databases to the Datalog program that do not correspond to any interpretation for the method schema. In order to handle this case, we introduce rules that will "flood" the computation by deriving every possible fact if the input database does not correspond to a valid interpretation. The second technical problem is the issue of inconsistencies. In the case of method schemas, an inconsistency corresponds to an abnormal termination of computation. In the Datalog simulation, we simply fail when an inconsistency occurs in the corresponding method execution. The two cases are similar in the sense that no output is produced for the offending method call.

Consider a schema $S = (C, \leq, \Sigma_0, \Sigma_1)$. Since we are interested in data complexity, we will assume that the size of the method schema is fixed. We will also assume that there are no ambiguous method definitions. (This is a syntactic property that can be checked in constant time.) Further, we will also assume that all the method definitions are explicit at all the classes at which they are defined. This might increase the size of the schema by a factor of c^k where c is the size of the class hierarchy, and k the maximum arity of the methods. Since we assume that S is of fixed size, this does not matter. For the input method schema S, we will construct a Datalog program P_S such that the computations performed by S and P_S are equivalent in a sense which will be explained below.

P_S has the following predicates:

1. For each class $c \in C$, a unary EDB predicate Isa_c. The Isa_c predicates are meant to capture the meaning of the disjoint object assignment.
2. A special interpreted binary predicate *Pred* that orders the input by providing the predecessor function. There will be objects o_0 (zero) and o_N (highest object) such that $\mathit{Pred}(o_0, o_0)$ is a fact and $\mathit{Pred}(X, o_N)$ is not true for any value of X. *Pred* corresponds naturally to the interpreted base method *Pred* in method schemas.
3. For each base method $m \in \Sigma_0$ of arity k, a $(k+1)$-ary EDB predicate m.
4. For each coded method $m \in \Sigma_1$ of arity k, a $(k+1)$-ary IDB predicate m.
5. An IDB predicate *Notequal* that captures inequality between input objects. It is coded in terms of *Pred*.
6. A unary IDB predicate *Flood* that will be used to check the validity of the input database.

For each method definition $m@c_1, c_2, \ldots, c_k = \lambda x_1, \ldots, x_k.\, m_1(\cdots)$ in S, the corresponding IDB predicate m has the following rule. The variables in the rule for m correspond to (1) the inputs to this code for method m and (2) the result and intermediate results of this code for method m.

$$\begin{aligned} m(C_1, C_2, \ldots, C_k, X) :\!\!- \; & \mathit{Isa}_{c_1}(C_1), \ldots, \mathit{Isa}_{c_k}(C_k), \\ & m_{i_1}(\ldots), m_{i_2}(\ldots), \ldots, m_1(\ldots, X); \end{aligned}$$

The understanding here is that the first k predicates act as "guards" making sure that the "code" that gets executed has the arguments in the correct classes. After that, we have the method names that appear in the code of m sorted in left-to-right topological order. (This is the order of execution provided by the built-in interpreter in method schemas.) The arguments to these methods are arranged as they would be in a method schema execution.

We now write down the definitions for the *Flood* predicate. There are four different ways in which the input database can be invalid. They are: (1) the disjoint object assignment could be invalid; (2) a base method could be undefined at some point where it is supposed to be defined (base methods correspond to *total* functions); (3) a base method could be multiply defined at some point; (4) a base method might produce a result of an invalid output class. We handle these one by one by means of the following rules. Before that, we need to code inequality among objects. This is easily done with the following rules:

$$\begin{aligned}
&\mathit{Notequal}(X,Y) :\!\!-\; \mathit{Pred}(X,X), \neg\, \mathit{Pred}(Y,Y);\\
&\mathit{Notequal}(X,Y) :\!\!-\; \neg\, \mathit{Pred}(X,X), \mathit{Pred}(Y,Y);\\
&\mathit{Notequal}(X,Y) :\!\!-\; \mathit{Pred}(X,X'), \mathit{Pred}(Y,Y'), \mathit{Notequal}(X',Y');
\end{aligned}$$

For every pair of classes c_1 and c_2 ($c_1 \neq c_2$), we use the following rule to enforce the disjoint object assignment:

$$\mathit{Flood}(X) :\!\!-\; \mathit{Isa}_{c_1}(C), \mathit{Isa}_{c_2}(C);$$

To make sure that every object belongs to a class, we use this rule:

$$\mathit{Flood}(X) :\!\!-\; \neg\, \mathit{Isa}_{c_1}(C), \neg\, \mathit{Isa}_{c_2}(C), \ldots, \neg\, \mathit{Isa}_{c_l}(C);$$

where $c_1, c_2, \ldots, c_l$ are all the classes in C. These rules ensure that whenever there is an object that does not belong to exactly one class, $\mathit{Flood}(X)$ is derived for all values of X.

We flood the computation by writing down rules of the following form for every IDB predicate p:

$$p(X_1, \ldots, X_k) :\!\!-\; \mathit{Flood}(X);$$

Next, if a base method m in S has the signature $m@c_1, \ldots, c_k\colon c_{k+1}$, we write the following rules that flood the computation if m is not defined everywhere in $\nu(c_1) \times \cdots \times \nu(c_k)$:

$$\mathit{Flood}(X) :\!\!-\; \mathit{Isa}_{c_1}(C_1), \ldots, \mathit{Isa}_{c_k}(C_k), \neg m(C_1, \ldots, C_k, C_{k+1});$$

Next, for a base method m in S with the signature $m@c_1, \ldots, c_k\colon c_{k+1}$, we write the following rule that floods the computation if m is multiply defined somewhere in $\nu(c_1) \times \cdots \times \nu(c_k)$.

$$\begin{aligned}
\mathit{Flood}(X) :\!\!-\; &\mathit{Isa}_{c_1}(C_1), \ldots, \mathit{Isa}_{c_k}(C_k),\\
&m(C_1, \ldots, C_k, C_{k+1}), m(C_1, \ldots, C_k, C'_{k+1}),\\
&\mathit{Notequal}(C_{k+1}, C'_{k+1});
\end{aligned}$$

Finally, for a base method m in S with signature $m@c_1, \ldots, c_k\colon c_{k+1}$, we write the following rule that floods the computation if m is improperly defined, i.e., the result belongs to an invalid class. Let $c_{k_1}, c_{k_2}, \ldots, c_{k_j}$ be the subclasses of c_{k+1}, including c_{k+1} itself.

$$\begin{aligned} \mathit{Flood}(X) :\!\!- \; & \mathit{Isa}_{c_1}(C_1), \ldots, \mathit{Isa}_{c_k}(C_k), \\ & m(C_1, \ldots, C_k, C_{k+1}), \\ & \neg\, \mathit{Isa}_{c_{k_1}}(C_{k+1}), \ldots, \neg\, \mathit{Isa}_{c_{k_j}}(C_{k+1}); \end{aligned}$$

Since in this simulation we are interested in data complexity, the parameter of interest is the size of the input data which we measure by the size of the interpretation I. It is immediately clear that our representation of an interpretation in Datalog$^\neg$ by means of the Isa_c, m, and *Pred* EDB predicates is the same size as the original method schema interpretation.

We now establish a one-to-one correspondence between interpretations for S and "valid" input databases for P_S.

Lemma 7. *Every interpretation for method schema S corresponds to an input database for P_S. Further, this input database is a "valid" input database in the sense that Flood(X) will never be derived for any value of X for such a database.*

Lemma 8. *Every "valid" input database for P_S (one in which Flood(X) is never derived for any value of X) corresponds to an interpretation for S.*

Now, we can use structural induction to show that S and P_S do indeed perform the same computations.

Lemma 9. *A call to method m with arguments $o_1, o_2, \ldots, o_k$ in an interpretation I terminates with a result o_{k+1} if and only if $m(o_1, o_2, \ldots, o_k, o_{k+1})$ can be derived in the corresponding input database in P_S. Further, Flood(X) will never be derived for any value of X in such an input database.*

Lemma 10. *If a fact $m(o_1, o_2, \ldots, o_k, o_{k+1})$ can be derived in P_S with a valid input database, then a call to method m with arguments $o_1, o_2, \ldots, o_k$ terminates with a result o_{k+1} in the corresponding interpretation for P.*

We put everything together in the following theorem:

Theorem 11. *Under the input output conventions of Section 3.1, MS with order expresses exactly PTIME.*

4 The Typed Lambda Calculus with Equality (TLC$^=$)

4.1 Language Definition

The syntax of *types* is given by the grammar $\mathcal{T} \equiv t \mid (\mathcal{T} \to \mathcal{T})$, where t ranges over a set of *type variables*. Thus, α is a type, as are $(\alpha \to \beta)$ and $(\alpha \to (\alpha \to \alpha))$. *Typed λ-terms* are given by the grammar $\mathcal{E} \equiv x \mid (\mathcal{E}\mathcal{E}) \mid \lambda x{:}\,\mathcal{T}.\,\mathcal{E}$, where x ranges over a set of *expression variables*. As usual, $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$ and PQR stands for $(PQ)R$.

Well-typedness of expressions is defined by the following inference rules, where Γ is a function from expression variables to types, and $\Gamma[x{:}\,\tau]$ is the function Γ' augmenting Γ with $\Gamma'(x) = \tau$:

$$(\textsc{Var}) \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x{:}\,\tau}$$

$$(\rightarrow\text{-}\textsc{Int}) \qquad \frac{\Gamma[x{:}\,\tau] \vdash e{:}\,\tau'}{\Gamma \vdash \lambda x{:}\,\tau.\, e{:}\,\tau \rightarrow \tau'}$$

$$(\rightarrow\text{-}\textsc{Elim}) \qquad \frac{\Gamma \vdash e{:}\,\tau \rightarrow \tau' \quad \Gamma \vdash e'{:}\,\tau}{\Gamma \vdash ee'{:}\,\tau'}$$

In what follows, we write "typed λ-term E" to mean E is well typed: $\Gamma \vdash E{:}\,\sigma$ is derivable by the above rules, for some Γ and σ.

The *order* of a type, which measures the higher-order functionality of a λ-term of that type, is defined as $o(t) = 0$ for a type variable t, and $o(\sigma' \rightarrow \sigma'') = \max(1 + o(\sigma'), o(\sigma''))$. We also refer to the order of a typed λ-term as the order of its type.

For well-typed λ-terms e, e', we write $e \rhd_\alpha e'$ (α-reduction) when e' can be derived from e by renaming of a bound variable, for example $\lambda x{:}\,\sigma.\, \lambda y{:}\,\tau.\, y \rhd_\alpha \lambda x{:}\,\sigma.\, \lambda z{:}\,\tau.\, z$. We write $e \rhd_\beta e'$ (β-reduction) when e' can be derived from e by replacing a subterm in e of the form $(\lambda x{:}\,\tau.\, E)E'$ by $E[E'/x]$ (E with E' substituted for all free occurrences of x in E). The fact that the subterm and its replacement have the same type is referred to as the *subject reduction theorem* [12, Theorem 4.2.8]. We write $\rhd$ for the reflexive, transitive closure of $\rhd_\alpha$ and $\rhd_\beta$.

In this paper, we consider a mild modification of the standard presentation above, by enriching the simply-typed λ-calculus with a countably infinite set of constants of type o, and for some fixed type τ, introducing an equality constant $=_\tau{:}\, \mathsf{o} \rightarrow \mathsf{o} \rightarrow \tau \rightarrow \tau \rightarrow \tau$. For constants $a, b{:}\, \mathsf{o}$, $a \not\equiv b$, we add to $\rhd_\beta$ the reduction rules $(=_\tau aa) \rhd_\beta (\lambda x{:}\,\tau.\, \lambda y{:}\,\tau.\, x)$ and $(=_\tau ab) \rhd_\beta (\lambda x{:}\,\tau.\, \lambda y{:}\,\tau.\, y)$.

The modified system enjoys the following well-known properties:

Theorem 12 (Church-Rosser property). *If $e \rhd e'$ and $e \rhd e''$, then there exists a λ-term e''' such that $e' \rhd e'''$ and $e'' \rhd e'''$.*

Theorem 13 (Strong normalization property). *For each e, there exists an integer n such that if $e \rhd e'$, then the derivation involves no more than n β-reductions.*

For more details, see for example [19, 11]. In particular, [11, Section 15.3] discusses the Church-Rosser property for the untyped λ-calculus enriched with equality. We refer to [25] for how to remove equality from the language.

For results concerning the *type inference* problem for TLC and its extensions, i. e. the problem of deciding whether a given (untyped) term can be typed, see for example [29]. The type inference problem for TLC$^=$ is equivalent to first-order unification.

Note that, the language presented above is the "simplest" typed lambda calculus. It has no type polymorphism and is a subset of the core-ML language of [35] without *let* and without any explicit fixpoint construct.

5 The Expressive Power of TLC$^=$

The principal technical tool for this section is *list iteration*. Let us briefly review how it works and its limitations.

Let $\{x_1, x_2, \ldots, x_k\}$ be a set of λ-terms, each of type α; then

$$L \equiv \lambda c{:}\, \alpha \to \sigma \to \sigma.\, \lambda n{:}\, \sigma.\, c\, x_1\, (c\, x_2 \ldots (c\, x_k\, n) \ldots)$$

is a λ-term of type $(\alpha \to \sigma \to \sigma) \to \sigma \to \sigma$, for *any* type σ—in other words, L is a typable term no matter what type σ we choose (though one fixed term must be chosen). We abbreviate this list construction as $[x_1, x_2, \ldots, x_k]$; the variables c and n abstract over the list constructors *cons* and *nil*. List iteration implements primitive recursion.

For example, a standard coding of Boolean logic uses $\mathit{True} \equiv \lambda x{:}\, \tau.\, \lambda y{:}\, \tau.\, x$ and $\mathit{False} \equiv \lambda x{:}\, \tau.\, \lambda y{:}\, \tau.\, y$, both of type $\mathit{Bool} \equiv \tau \to \tau \to \tau$. Define the exclusive or as

$$\mathit{Xor} \equiv \lambda p{:}\, \mathit{Bool}.\, \lambda q{:}\, \mathit{Bool}.\, \lambda x{:}\, \tau.\, \lambda y{:}\, \tau.\, p\, (q\, y\, x)\, (q\, x\, y)$$

and the parity of a list of Boolean values as

$$\mathit{Parity} \equiv \lambda L{:}\, (\mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}) \to \mathit{Bool} \to \mathit{Bool}.\, L\, \mathit{Xor}\, \mathit{False}\ .$$

Unlike circuit complexity, the size of the *program* computing parity is constant, because the iterative machinery is taken from the *data*, i. e., the list L.

On the other hand, to compute the length of a list, we define

$$\mathit{Length} \equiv \lambda L{:}\, (\alpha \to \mathit{Int} \to \mathit{Int}) \to \mathit{Int} \to \mathit{Int}.\, L\, (\lambda x{:}\, \alpha.\, \mathit{Succ})\, \mathit{Zero}\ ,$$

where $\mathit{Succ} \equiv \lambda n{:}\, (\tau \to \tau) \to \tau \to \tau.\, \lambda s{:}\, \tau \to \tau.\, \lambda z{:}\, \tau.\, ns(sz)$ and $\mathit{Zero} \equiv \lambda s{:}\, \tau \to \tau.\, \lambda z{:}\, \tau.\, z$ code successor and zero on the Church numerals (of type $\mathit{Int} \equiv (\tau \to \tau) \to \tau \to \tau$).

These two simple examples point already to a restriction imposed by the simply typed lambda calculus: its lack of polymorphism. There are two facets to this problem.

(1) A list L of Booleans can be coded as a term of type $\mathcal{B} \equiv (\mathit{Bool} \to \sigma \to \sigma) \to \sigma \to \sigma$ for any type σ. To type $\mathit{Parity}\, L$, we must type L with $\mathit{Bool} \equiv \tau \to \tau \to \tau$ substituted for σ, which we write as $\mathcal{B}[\sigma{:}= \mathit{Bool}]$. Similarly, to type $\mathit{Length}\, L$, we need to type $L{:}\, \mathcal{B}[\sigma{:}= \mathit{Int}]$. Thinking of the type σ of L as an output type *variable*, in both instances we see that the output type must be "contaminated" or "raised" to provide the primitive recursive iterator. If we

want the output type of input L to be fixed, and two different output types are needed to carry out the computation, this poses a problem.

(2) If we want the output type to remain σ, this poses yet another difficulty.

We point out that problem (2) can sometimes be handled by alternate encodings, for example

$$\mathit{Length} \equiv \lambda L\colon (\alpha \to \sigma \to \sigma) \to \sigma \to \sigma.\, \lambda s\colon \sigma \to \sigma.\, \lambda z\colon \sigma.\, L\,(\lambda x\colon \alpha.\, s)\, z \quad ,$$

where L is used to iterate over objects of lower type. Problem (1) can be solved by defining the output type to be a suitably defined cross product of *Bool* and *Int*, and using this typing to iteratively generate two copies of the list, with respective output types *Bool* and *Int*.

These techniques, with elaborate variants, are exploited repeatedly in the following sections.

5.1 I/O Conventions

The paradigm we use is computation on finite structures, i. e., a λ-calculus for finite model theory. Inputs and outputs of a computation are *databases*, i. e., sets of relations, which are encoded according to the scheme below.

We start out with a countably infinite supply $o_1, o_2, \ldots$ of *constants*, of some base type o. We also fix, once and for all, a type variable τ.

Boolean values are represented by

$$\mathit{True}\colon \tau \to \tau \to \tau \equiv \lambda u\colon \tau.\, \lambda v\colon \tau.\, u \quad ,$$
$$\mathit{False}\colon \tau \to \tau \to \tau \equiv \lambda u\colon \tau.\, \lambda v\colon \tau.\, v \quad .$$

We abbreviate the type $\tau \to \tau \to \tau$ as *Bool*.

Tuples are represented in the standard way: if $x_1, \ldots, x_k$ are λ-terms of type $\alpha_1, \ldots, \alpha_k$, then

$$\begin{aligned}\langle x_1\,, \ldots, x_k\rangle : (\alpha_1 \to \cdots \to \alpha_k \to \tau) \to \tau \equiv \\ \lambda f\colon \alpha_1 \to \cdots \to \alpha_k \to \tau.\, f\, x_1 \ldots x_k\end{aligned}$$

We abbreviate the type $(\alpha_1 \to \cdots \to \alpha_k \to \tau) \to \tau$ as $\alpha_1 \times \cdots \times \alpha_k$ or, if $\alpha_i \equiv \alpha$ for $1 \le i \le k$, as α^k. In particular, o^k is the type of a k-tuple of constants.

Lists are represented as described above: if $x_1, \ldots, x_k$ are λ-terms of type α, then

$$\begin{aligned}[x_1\,, \ldots, x_k] : (\alpha \to \tau \to \tau) \to \tau \to \tau \equiv \\ \lambda c\colon \alpha \to \tau \to \tau.\, \lambda n\colon \tau.\, c\, x_1\, (c\, x_2 \ldots (c\, x_k\, n) \cdots)\end{aligned}$$

We abbreviate the type $(\alpha \to \tau \to \tau) \to \tau \to \tau$ as $\{\alpha\}$. Note the difference between lists and tuples: a tuple represents a collection of a *fixed* number of terms of *varying* type, whereas a list represents a collection of a *variable* number of terms of a *fixed* type.

Relations are represented as duplicate-free lists of tuples of constants. Thus, the type of a k-ary relation is $\{\mathsf{o}^k\}$. Note that, in all this development we use lists instead of finite sets, because finite sets are not available in the syntax of the λ-calculus; but we show how to eliminate duplicates.

Queries are represented as λ-terms of the form $Q \equiv \lambda R_1 \ldots \lambda R_k . M$, which when applied to encodings $\overline{r_1}, \ldots, \overline{r_k}$ of input relations, reduce to a normal form which is the encoding of the desired output relation. We measure the time complexity of computation by the length of reduction sequences, according to a fixed evaluation strategy, as a function of the database size.

It is interesting to observe that the "relations as iterators" convention above is equivalent to the "relations as characteristic functions" convention adopted in Section 3.1, *provided* that in the latter case, the input comes with an iterator which is a list of all the objects in the universe (the equivalent of the "size parameter" N in the method schemas case). Clearly, without any a priori knowledge of the universe, it is impossible to obtain a listing of a relation from its characteristic function.

However, if we assume the existence of an iterator $U : \{\mathsf{o}\}$ representing the universe, then translations between the two conventions are easily furnished. Indeed, if $R : \{\mathsf{o}^k\}$ is an iterator encoding a relation r, then

$$\begin{aligned}
&\chi_r : \overbrace{\mathsf{o} \to \cdots \to \mathsf{o}}^{k} \to \mathit{Bool} \equiv \\
&\qquad \lambda x_1 : \mathsf{o} \ldots \lambda x_k : \mathsf{o} . \\
&\qquad\qquad \lambda u : \tau . \, \lambda v : \tau . \\
&\qquad\qquad\qquad R \, (\lambda r : \mathsf{o}^k . \, \lambda T : \tau . \, (\mathit{Equal}_k \, r \, \langle x_1 \, , \ldots , x_k \rangle) \, u \, T) \, v
\end{aligned}$$

(where Equal_k, the equality predicate for k-tuples, is defined below) computes the characteristic function of r.

On the other hand, if $\chi_r : \mathsf{o} \to \cdots \to \mathsf{o} \to \mathit{Bool}$ encodes the characteristic function of a relation r, then

$$\begin{aligned}
&R : \{\mathsf{o}^k\} \equiv \\
&\qquad \lambda c : \mathsf{o}^k \to \tau \to \tau . \, \lambda n : \tau . \\
&\qquad\qquad U \, (\lambda x_1 : \mathsf{o} . \, \lambda T_1 : \tau . \\
&\qquad\qquad\qquad U \, (\lambda x_2 : \mathsf{o} . \, \lambda T_2 : \tau . \\
&\qquad\qquad\qquad\qquad \cdots \\
&\qquad\qquad\qquad\qquad\qquad U \, (\lambda x_k : \mathsf{o} . \, \lambda T_k : \tau . \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\chi_r \, x_1 \ldots x_k) \, (c \, \langle x_1 \, , \ldots , x_k \rangle \, T_k) \, T_k) \, T_{k-1}) \ldots T_1) \, n
\end{aligned}$$

is an iterator representing r.

Note that the characteristic function representation of a relation is simpler in the sense that its type is of order 1, whereas the type of an iterator is of order 2. It is possible to exploit this "order economy" of characteristic functions to provide an encoding of all PTIME computable queries in the typed lambda calculus of order 3. The (more simple) encoding given below uses types of order 4.

5.2 Expressing Relational Queries

In this section, we show how to express the relational algebra operators of [20] in the simply typed lambda calculus with equality. Every operator is coded as a λ-term that takes one or two relations in the list-of-tuples format described in Section 5.1 as input and produces another relation in the same format as output. The homogeneity of the input and output format allows the result of one operator application to be used as input to another, so that arbitrary relational algebra expressions can be coded by nesting the λ-terms corresponding to the individual operators.

We begin with a λ-term $Equal_k$ that tests two k-tuples $\lambda f.\, f a_1 \ldots a_k$ and $\lambda f.\, f b_1 \ldots b_k$ for equality. The result of the comparison is a *Bool*, i. e., the term $\lambda u \lambda v.\, u$ if the comparison comes out true and $\lambda u \lambda v.\, v$ otherwise. The code depends on the arity of the tuples involved, so we use the subscript k to indicate that this particular term works for k-tuples.

$$
\begin{array}{l}
Equal_k\colon \mathsf{o}^k \to \mathsf{o}^k \to Bool \equiv \\
\quad \lambda t\colon \mathsf{o}^k.\, \lambda s\colon \mathsf{o}^k. \\
\quad\quad \lambda u\colon \tau.\, \lambda v\colon \tau. \\
\quad\quad\quad t\,(\lambda x_1\colon \mathsf{o} \ldots \lambda x_k\colon \mathsf{o}. \\
\quad\quad\quad\quad s\,(\lambda y_1\colon \mathsf{o} \ldots \lambda y_k\colon \mathsf{o}. \\
\quad\quad\quad\quad\quad ((Eq\, x_1\, y_1) \\
\quad\quad\quad\quad\quad\quad ((Eq\, x_2\, y_2) \\
\quad\quad\quad\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad\quad\quad\quad\quad ((Eq\, x_k\, y_k)\, u\, v)\, v) \cdots v)))
\end{array}
$$

Here, Eq denotes the equality predicate for constants, of type $\mathsf{o} \to \mathsf{o} \to Bool$. Once t and s are instantiated with tuples $\lambda f.\, f a_1 \ldots a_k$ and $\lambda f.\, f b_1 \ldots b_k$, all $(Eq\, x_i y_i)$ terms reduce to either $True \equiv \lambda u.\, \lambda v.\, u$ or $False \equiv \lambda u.\, \lambda v.\, v$. Hence the "body"

$$
\begin{array}{l}
\lambda u\colon \tau.\, \lambda v\colon \tau. \\
\quad t\,(\lambda x_1\colon \mathsf{o} \ldots \lambda x_k\colon \mathsf{o}. \\
\quad\quad s\,(\lambda y_1\colon \mathsf{o} \ldots \lambda y_k\colon \mathsf{o}. \\
\quad\quad\quad ((Eq\, x_1\, y_1) \\
\quad\quad\quad\quad ((Eq\, x_2\, y_2) \\
\quad\quad\quad\quad\quad \cdots \\
\quad\quad\quad\quad\quad\quad ((Eq\, x_k\, y_k)\, u\, v)\, v) \cdots v)))
\end{array}
$$

of the $Equal_k$ predicate reduces to $\lambda u.\, \lambda v.\, u$ if $a_i = b_i$ for all i and to $\lambda u.\, \lambda v.\, v$ otherwise. Note that the final result is reached after $O(k)$ reduction steps (independent of the reduction strategy), which matches the time needed to compare two k-tuples in a procedural language.

The following term checks whether a k-tuple $\lambda f.\, f a_1 \ldots a_k$ is a member of a list $\lambda c \lambda n.\, c t_1 (c t_2 \cdots (c t_m n) \cdots)$ of k-tuples by comparing the tuple to every element of the list.

$$
\begin{aligned}
&\mathit{Member}_k\colon \mathsf{o}^k \to \{\mathsf{o}^k\} \to \mathit{Bool} \equiv \\
&\qquad \lambda t\colon \mathsf{o}^k .\, \lambda R\colon \{\mathsf{o}^k\}. \\
&\qquad\qquad \lambda u\colon \tau .\, \lambda v\colon \tau . \\
&\qquad\qquad\qquad R\,(\lambda r\colon \mathsf{o}^k .\, \lambda T\colon \tau .\, (\mathit{Equal}_k\, r\, t)\, u\, T)\, v
\end{aligned}
$$

For example, $(\mathit{Member}_2\, \langle 1,2\rangle\, [\langle 3,4\rangle\, , \langle 5,6\rangle])$ reduces to

$$\lambda u.\, \lambda v.\, (\mathit{Equal}_2\, \langle 3\, , 4\rangle\, \langle 1\, , 2\rangle)\, u\, ((\mathit{Equal}_2\, \langle 5\, , 6\rangle\, \langle 1\, , 2\rangle)\, u\, v) \quad ,$$

which in turn reduces to $\lambda u.\, \lambda v.\, v$, because both $(\mathit{Equal}_2 \ldots)$ terms evaluate to *False*. If the reductions are done according to an eager reduction strategy (which reduces a redex $(\lambda x.\, M)N$ only after fully reducing N), then the number of reductions needed to reach the normal form is $O(mk)$, where m is the length of the input list, and this again matches the time required by a straightforward procedural implementation.

Note that there are also "bad" reduction strategies which lead to an exponential number of reduction steps. This is due to the fact that the fourth parameter v of the Equal_k predicate occurs k times in the body of the predicate. If a term with unresolved redexes is substituted for v (e. g., the term $((\mathit{Equal}_2\, \langle 5,6\rangle\, \langle 1,2\rangle)uv)$ in the example above), then the number of redexes in the whole expression multiplies by k. Since this happens at every level of Equal_k nesting, it is possible to reduce an expression $(\mathit{Member}_k\, tR)$, where R is of length m, to a term with $O(k^m)$ redexes by never evaluating the fourth argument to Equal_k before substituting it.

With the aid of *Equal* and *Member*, we can now code the set-theoretic operators of the relational algebra:

Intersection: To intersect two k-ary relations R and S, we build a new relation T by "walking down" R and testing each tuple for membership in S. If the tuple occurs in S, it is included in the output, otherwise it is ignored.

$$
\begin{aligned}
&\mathit{Intersection}_k\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \equiv \\
&\qquad \lambda R\colon \{\mathsf{o}^k\}.\, \lambda S\colon \{\mathsf{o}^k\}. \\
&\qquad\qquad \lambda c\colon \mathsf{o}^k \to \tau \to \tau .\, \lambda n\colon \tau . \\
&\qquad\qquad\qquad R\,(\lambda r\colon \mathsf{o}^k .\, \lambda T\colon \tau .\, (\mathit{Member}_k\, r\, S)\, (c\, r\, T)\, T)\, n
\end{aligned}
$$

For example, $(\mathit{Intersection}_2\, [\langle 1,2\rangle]\, [\langle 1,2\rangle\, , \langle 3,4\rangle])$ reduces to

$$\lambda c.\, \lambda n.\, (\mathit{Member}_2\, \langle 1\, , 2\rangle\, [\langle 1\, , 2\rangle\, , \langle 3\, , 4\rangle])\, (c\, \langle 1\, , 2\rangle\, n)\, n \quad ,$$

which further reduces to $\lambda c.\, \lambda n.\, c\, \langle 1,2\rangle\, n \equiv [\langle 1,2\rangle]$.

Set difference: This works like intersection, except that a tuple from R is included in the output if it does *not* occur in S.

$$\mathit{Setminus}_k\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \equiv$$

$$\begin{aligned}
&\lambda R\colon \{\mathsf{o}^k\}.\, \lambda S\colon \{\mathsf{o}^k\}.\\
&\quad \lambda c\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n\colon \tau.\\
&\qquad R\,(\lambda r\colon \mathsf{o}^k.\, \lambda T\colon \tau.\, (\mathit{Member}_k\; r\; S)\; T\; (c\; r\; T))\; n
\end{aligned}$$

Union: The union of two k-ary relations R and S is formed by starting with S and adding those tuples of R that do not occur in S.

$$\begin{aligned}
&\mathit{Union}_k\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \equiv\\
&\quad \lambda R\colon \{\mathsf{o}^k\}.\, \lambda S\colon \{\mathsf{o}^k\}.\\
&\qquad \lambda c\colon \mathsf{o}^k \to \tau \to \tau.\, \lambda n\colon \tau.\\
&\qquad\quad R\,(\lambda r\colon \mathsf{o}^k.\, \lambda T\colon \tau.\, (\mathit{Member}_k\; r\; S)\; T\; (c\; r\; T))\; (S\; c\; n)
\end{aligned}$$

(If we knew that R and S were disjoint, we could form their union by just concatenating them as follows: $\lambda c.\, \lambda n.\, Rc(Scn)$. In the general case, however, we need the more complex term above to ensure a duplicate-free output.)

Having dealt with the "set-oriented" operators, we now examine the "tuple-oriented" operators. We need two auxiliary terms first: The $\mathit{Concat}_{k,l}$ operator concatenates a k-tuple and an l-tuple to form a $(k + l)$-tuple, and the $\mathit{Rearrange}_{k;i_1,\ldots,i_l}$ operator takes a k-tuple and returns an l-tuple consisting of columns $i_1, \ldots, i_l$ of the input.

$$\begin{aligned}
&\mathit{Concat}_{k,l}\colon \mathsf{o}^k \to \mathsf{o}^l \to \mathsf{o}^{k+l} \equiv\\
&\quad \lambda t\colon \mathsf{o}^k.\, \lambda s\colon \mathsf{o}^l.\\
&\qquad \lambda f\colon \mathsf{o} \to \cdots \to \mathsf{o} \to \tau.\\
&\qquad\quad t\,(\lambda x_1\colon \mathsf{o} \ldots \lambda x_k\colon \mathsf{o}.\\
&\qquad\qquad s\,(\lambda y_1\colon \mathsf{o} \ldots \lambda y_l\colon \mathsf{o}.\\
&\qquad\qquad\quad f\; x_1 \cdots x_k\; y_1 \cdots y_l))
\end{aligned}$$

$$\begin{aligned}
&\mathit{Rearrange}_{k;i_1,\ldots,i_l}\colon \mathsf{o}^k \to \mathsf{o}^l \equiv\\
&\quad \lambda t\colon \mathsf{o}^k.\\
&\qquad \lambda f\colon \mathsf{o} \to \cdots \to \mathsf{o} \to \tau.\\
&\qquad\quad t\,(\lambda x_1\colon \mathsf{o} \ldots \lambda x_k\colon \mathsf{o}.\\
&\qquad\qquad f\; x_{i_1} \cdots x_{i_l})
\end{aligned}$$

Cross product: The Cartesian product of k-ary relation R and l-ary relation S is formed by a straightforward nested iteration, in which every tuple of R is concatenated with every tuple of S and appended to the output.

$$\begin{aligned}
&\mathit{Times}_{k,l}\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^l\} \to \{\mathsf{o}^{k+l}\} \equiv\\
&\quad \lambda R\colon \{\mathsf{o}^k\}.\, \lambda S\colon \{\mathsf{o}^l\}.\\
&\qquad \lambda c\colon \mathsf{o}^{k+l} \to \tau \to \tau.\, \lambda n\colon \tau.\\
&\qquad\quad R\,(\lambda r\colon \mathsf{o}^k.\, \lambda T\colon \tau.\\
&\qquad\qquad S\,(\lambda s\colon \mathsf{o}^l.\, \lambda U\colon \tau.\\
&\qquad\qquad\quad c\,(\mathit{Concat}_{k,l}\; r\; s)\; U)\; T)\; n
\end{aligned}$$

For example, $(\mathit{Times}_{1,1}\,[\langle 1\rangle\,,\langle 2\rangle]\,[\langle 3\rangle\,,\langle 4\rangle])$ reduces to

$$
\begin{array}{l}
\lambda c.\,\lambda n.\\
\qquad c\,(\mathit{Concat}_{1,1}\,\langle 1\rangle\,\langle 3\rangle)\\
\qquad\qquad (c\,(\mathit{Concat}_{1,1}\,\langle 1\rangle\,\langle 4\rangle)\\
\qquad\qquad\qquad (c\,(\mathit{Concat}_{1,1}\,\langle 2\rangle\,\langle 3\rangle)\\
\qquad\qquad\qquad\qquad (c\,(\mathit{Concat}_{1,1}\,\langle 2\rangle\,\langle 4\rangle)\,n)))
\end{array}
$$

which further reduces to $[\langle 1,3\rangle\,,\langle 1,4\rangle\,,\langle 2,3\rangle\,,\langle 2,4\rangle]$.

Selection: To select tuples from a k-ary relation R according to a certain condition, say column i = column j, it suffices to iterate over R and include those tuples in the output that satisfy the condition.

$$
\begin{array}{l}
\mathit{Select}_{k;i=j}\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^k\} \equiv\\
\qquad \lambda R\colon \{\mathsf{o}^k\}.\\
\qquad\qquad \lambda c\colon \mathsf{o}^k \to \tau \to \tau.\,\lambda n\colon \tau.\\
\qquad\qquad\qquad R\,(\lambda r\colon \mathsf{o}^k.\,\lambda T\colon \tau.\\
\qquad\qquad\qquad\qquad r\,(\lambda x_1\colon \mathsf{o}\ldots\lambda x_k\colon \mathsf{o}.\,(\mathit{Eq}\,x_i\,x_j)\,(c\,r\,T)\,T))\,n
\end{array}
$$

Projection: This is slightly tricky. The straightforward attempt to project onto columns $i_1,\ldots,i_l$ of a k-ary relation R,

$$
\begin{array}{l}
\mathit{SimpleProject}_{k;i_1,\ldots,i_l}\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^l\} \equiv\\
\qquad \lambda R\colon \{\mathsf{o}^k\}.\\
\qquad\qquad \lambda c\colon \mathsf{o}^l \to \tau \to \tau.\,\lambda n\colon \tau.\\
\qquad\qquad\qquad R\,(\lambda r\colon \mathsf{o}^k.\,\lambda T\colon \tau.\,c\,(\mathit{Rearrange}_{k;i_1,\ldots,i_l}\,r)\,T)\,n \quad ,
\end{array}
$$

has the disadvantage that the output may contain duplicate tuples. To fix this, we include the projection t' of tuple t in the output only if t is the *first* tuple in R whose projection is t':

$$
\begin{array}{l}
\mathit{Project}_{k;i_1,\ldots,i_l}\colon \{\mathsf{o}^k\} \to \{\mathsf{o}^l\} \equiv\\
\qquad \lambda R\colon \{\mathsf{o}^k\}.\\
\qquad\qquad \lambda c\colon \mathsf{o}^l \to \tau \to \tau.\,\lambda n\colon \tau.\\
\qquad\qquad\qquad R\,(\lambda r\colon \mathsf{o}^k.\,\lambda T\colon \tau.\\
\qquad\qquad\qquad\qquad (\mathit{First}\,r\,R)\,(c\,(\mathit{Rearrange}_{k;i_1,\ldots,i_l}\,r)\,T)\,T)\,n \quad ,
\end{array}
$$

where $(\mathit{First}\,r R)\colon \mathit{Bool}$ is an abbreviation for

$$
\begin{array}{l}
\lambda u\colon \tau.\,\lambda v\colon \tau.\\
\qquad R\,(\lambda r'\colon \mathsf{o}^k.\,\lambda T'\colon \tau.\\
\qquad\qquad (\mathit{Equal}_l\,(\mathit{Rearrange}_{k;i_1,\ldots,i_l}\,r)\\
\qquad\qquad\qquad (\mathit{Rearrange}_{k;i_1,\ldots,i_l}\,r'))\\
\qquad\qquad\qquad ((\mathit{Equal}_k\,r\,r')\,u\,v)\,T')\,u \quad .
\end{array}
$$

To see why this works, consider for example the query $(\mathit{Project}_{2;1}\,[\langle 1,2\rangle\,,\langle 1,3\rangle])$, which projects the relation $[\langle 1,2\rangle\,,\langle 1,3\rangle]$ onto its first column. The expression $(\mathit{First}\,\langle 1,2\rangle\,[\langle 1,2\rangle\,,\langle 1,3\rangle])$ reduces to

$$
\begin{array}{l}
\lambda u.\,\lambda v. \\
\qquad (\mathit{Equal}_1\,\langle 1\rangle\,\langle 1\rangle) \\
\qquad\qquad ((\mathit{Equal}_2\,\langle 1\,,2\rangle\,\langle 1\,,2\rangle)\,u\,v) \\
\qquad\qquad ((\mathit{Equal}_1\,\langle 1\rangle\,\langle 1\rangle)\,((\mathit{Equal}_2\,\langle 1\,,2\rangle\,\langle 1\,,3\rangle)\,u\,v)\,u) \quad ,
\end{array}
$$

which further reduces to $\lambda u.\,\lambda v.\,u \equiv \mathit{True}$, whereas $(\mathit{First}\,\langle 1,3\rangle\,[\langle 1,2\rangle\,,\langle 1,3\rangle])$ reduces to

$$
\begin{array}{l}
\lambda u.\,\lambda v. \\
\qquad (\mathit{Equal}_1\,\langle 1\rangle\,\langle 1\rangle) \\
\qquad\qquad ((\mathit{Equal}_2\,\langle 1\,,3\rangle\,\langle 1\,,2\rangle)\,u\,v) \\
\qquad\qquad ((\mathit{Equal}_1\,\langle 1\rangle\,\langle 1\rangle)\,((\mathit{Equal}_2\,\langle 1\,,3\rangle\,\langle 1\,,3\rangle)\,u\,v)\,u)
\end{array}
$$

and then to $\lambda u.\,\lambda v.\,v \equiv \mathit{False}$. Hence, the whole query reduces to

$$
\begin{array}{l}
\lambda c.\,\lambda n. \\
\qquad (\mathit{First}\,\langle 1\,,2\rangle\,[\langle 1\,,2\rangle\,,\langle 1\,,3\rangle]) \\
\qquad\qquad (c\,\langle 1\rangle\,((\mathit{First}\,\langle 1\,,3\rangle\,[\langle 1\,,2\rangle\,,\langle 1\,,3\rangle])\,(c\,\langle 1\rangle\,n)\,n)) \\
\qquad\qquad ((\mathit{First}\,\langle 1\,,3\rangle\,[\langle 1\,,2\rangle\,,\langle 1\,,3\rangle])\,(c\,\langle 1\rangle\,n)\,n)
\end{array}
$$

and then to $\lambda c.\,\lambda n.\,c\,\langle 1\rangle\,n \equiv [\langle 1\rangle]$.

This completes the coding of relational algebra. It is straightforward to verify that every term given above is well-typed and, when applied to one (resp., two) relations coded in the format described in Section 5.1, reduces to a normal form which is the encoding of the desired output relation. Moreover, when the terms are reduced according to an eager reduction strategy (which reduces a redex $(\lambda x.\,M)N$ only after fully reducing N), the length of the reduction sequence is polynomial in the size of the inputs. (In fact, the length of the reduction sequence typically matches the running time of a naive procedural implementation of the operator in question.) Therefore, we have the following theorem.

Theorem 14. *Let $\phi(R_1,\ldots,R_l)$ be a relational algebra expression over relational schema $\mathcal{S} = (R_1,\ldots,R_l)$. Then there exists a term ψ of the simply typed lambda calculus with equality such that for every instance $(r_1,\ldots,r_l)$ of $\mathcal{S}$, the expression $(\psi\overline{r_1}\cdots\overline{r_l})$ reduces to $\overline{\phi(r_1,\ldots,r_l)}$, where $\overline{r}$ denotes the encoding of relation r as described in Section 5.1. Moreover, the normal form of $(\psi\overline{r_1}\cdots\overline{r_l})$ can be computed in a number of reduction steps polynomial in the size of $r_1,\ldots,r_l$.*

It is possible to extend the scheme above to an encoding of fixpoints and even Abiteboul's and Beeri's Complex Object Algebra [1]. This takes us from manipulating "flat" relations to more complicated data structures, namely arbitrary finite trees built from tuple and set constructors. The algebra under consideration is extremely powerful—in fact it expresses any "generic" (see [16]) elementary time computation on finite structures.

Theorem 15. *Let $\phi(R_1, \ldots, R_l)$ be a complex object algebra expression over the relational schema $\mathcal{S} = (R_1, \ldots, R_l)$. Then there exists a term ψ of the simply typed lambda calculus with equality such that for every instance $(r_1, \ldots, r_l)$ of $\mathcal{S}$, the expression $(\psi \overline{r_1} \cdots \overline{r_l})$ reduces to $\overline{\phi(r_1, \ldots, r_l)}$, where $\overline{r}$ denotes the encoding of relation r. Moreover, if the Powerset operator is not used, the normal form of $(\psi \overline{r_1} \cdots \overline{r_l})$ can be computed in a number of reduction steps polynomial in the size of $r_1, \ldots, r_l$.*

6 Conclusions and Some Open Problems

We have explored two functional formalisms, MS and TLC$^=$, in the context of expressing database queries or queries over finite structures. The main point of our exposition is that these formalisms can be related to traditional database query languages and can express all that is expressible using relational calculus/algebra, Datalog, etc.

We believe that the additional features of these functional formalisms (e.g., for MS classes, methods, inheritance, name overloading, late binding, and for TLC$^=$ higher-order functions and list, set, and tuple structures via typing) make them better vehicles for the formal study of OODBs. Thus, one can view our exploration as an attempt to change the vocabulary from a relational to a functional setting, while preserving some of the basic theorems of database theory. The following open problems highlight some topics for future research.

MS: We have investigated the consistency problem of method schemas as a form of signature inference and have highlighted a number of efficiently decidable cases. The decidability of the arity 2 case is still open, as well as many aspects of incremental consistency checking. In our formalization of method schema consistency, all base methods are uninterpreted. As we saw from the analysis of expressive power some degree of semantics, e.g., order, is useful. What about consistency then? For example, let us suppose that we have equality, with its normal interpretation, as a base method. This method will simply tell if the two arguments that it is supplied with are the same. Consistency checking becomes undecidable with monadic coded methods and polyadic base methods when augmented with equality. With equality, we can simulate the Lisp *car* and *cdr* functions so that we can "glue" and "unglue" arguments. However, the problem of checking consistency with monadic coded methods and polyadic base methods without equality is open.

TLC$^=$: The embeddings of database query languages in the typed λ-calculus that we present here are interesting for a number of reasons. They are "pure," without added constructs or added polymorphism. For example, the typed λ-calculus with equality is the simplest syntax to date, that can be used to describe the complex object algebra of [1]. In our embeddings we use lists of tuples and simulate sets by eliminating duplicates. In [1, 15, 13] sets are used as basic constructs, and set iteration replaces list iteration in [15, 13]. One open question is: how can the λ-calculus syntax and semantics be augmented with set iteration? Not all database query languages can be embedded in the typed λ-calculus. Any computation that is not elementary is not captured by our framework. What should be added to the typed λ-calculus to capture exactly the more powerful database query languages such as the computable queries of [16, 2]? The use of reduction strategies in these embeddings provides a link between complexity theory, finite model theory, and the typed λ-calculus, that should be further investigated.

MS+TLC$^=$: We believe that a synthesis of MS and TLC$^=$, with set-at-a-time instead of tuple-at-at-time evaluation, would be the appropriate formalism (or very close to it) for OODB querying. Such a formalism would have all the object-oriented features of MS and the expressibility and data abstraction capabilities of TLC$^=$. Given the organization of data into collections one would like the evaluation mechanism to emphasize sets, as in [1, 15, 13]. Object-oriented dialects of Lisp [7] illustrate such linguistic syntheses (without sets and with a less clear type system). The challenge is to come up with a simply typed (and elegant) formal synthesis, that also facilitates the manipulation of collections of objects.

In our exposition we have explored the relationship with program schemas and simply typed calculi. This is a first step towards more complex database programming language models. For example, *let*- or ML-polymorphism [35] is a very useful feature to include in a functional data model. The language ML has been the basis for many programming language investigations on structure and method inheritance. The literature is too extensive to even attempt a survey, but for pointers to the recent literature and for a programming language view of some of the issues raised here we recommend [36, 37].

References

1. S. Abiteboul and C. Beeri. *On the Power of Languages for the Manipulation of Complex Objects.* INRIA Research Report 846, 1988.
2. S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings ACM SIGMOD*, pp. 159–173, 1989. (Also INRIA Tech. Rep. 1022, 1989.)
3. S. Abiteboul and P. Kanellakis. Database Theory Column: Query Languages for Complex Object Databases. *SIGACT News*, **21** (1990), pp. 9–18.
4. S. Abiteboul, P. Kanellakis, S. Ramaswamy, and E. Waller. *Method Schemas.* Brown University Tech. Rep. CS-92-33, 1992. (An earlier version appeared in *Proceedings 9th ACM PODS*, 1990.)
5. S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and Rules. In *Proceedings ACM SIGMOD*, pp. 32–41, 1993.
6. S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. *JCSS*, **43** (1991), pp. 62–124.
7. N. Adams and J. Rees. Object-Oriented Programming in Scheme. In *Proceedings ACM Lisp and Functional Programming*, pp. 277–288, 1988.

8. E. Ashcroft, Z. Manna, and A. Pnueli. Decidable Properties of Monadic Functional Schemas. *JACM*, **20** (1973), pp. 489–499.
9. F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan-Kaufmann, 1992.
10. J. Banerjee, W. Kim, H-J. Kim, and H. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings ACM SIGMOD*, pp. 311-322, 1987.
11. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.
12. H. Barendregt. Functional Programming and Lambda Calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 321–363. Elsevier, 1990.
13. V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings DBPL3*, pp. 9–19. Morgan-Kaufmann, 1992.
14. V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *Proceedings 4th ICDT*. LNCS, Springer Verlag, 1992.
15. P. Buneman, R. Frankel, and R. Nikhil. An Implementation Technique for Database Query Languages. *ACM TODS*, **7** (1982), pp. 164–186.
16. A. Chandra and D. Harel. Computable Queries for Relational Databases. *JCSS*, **21** (1980), pp. 156–178.
17. A. Chandra and D. Harel. Structure and Complexity of Relational Queries. *JCSS*, **25** (1982), pp. 99–128.
18. A. Chandra and D. Harel. Horn Clause Queries and Generalizations. *JLP*, **2** (1985), pp. 1–15.
19. A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.
20. E. Codd. Relational Completeness of Database Sublanguages. In R. Rustin, editor, *Database Systems*, pp. 65–98. Prentice Hall, 1972.
21. B. Courcelle. Recursive Applicative Program Schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 459–492. Elsevier, 1990.
22. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. *SIAM-AMS Proceedings*, **7** (1974), pp. 43–73.
23. S. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification*. LNCS Vol. 36, Springer, 1975.
24. Y. Gurevich. Algebras of Feasible Functions. In *Proceedings 24th IEEE FOCS*, pp. 210–214, 1983.
25. G. Hillebrand, P. Kanellakis, and H. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. In *Proceedings 8th IEEE LICS*, 1993.
26. R. Hull, K. Tanaka, and M. Yoshikawa. Behavior Analysis of Object-Oriented Databases: Method Structure, Execution Trees and Reachability. In *Proceedings 3rd International Conference on Foundations of Data Organization and Algorithms*, Paris 1989.
27. N. Immerman. Relational Queries Computable in Polynomial Time. *Info. and Comp.*, **68** (1986), pp. 86–104.
28. P. Kanellakis. Elements of Relational Database Theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 1073–1156. Elsevier, 1990.
29. P. Kanellakis, J. Mitchell, and H. Mairson. Unification and ML-type reconstruction. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pp. 444–479. MIT Press, 1991. (Also in *Proceedings 16th ACM POPL*, 1989 and *Proceedings 17th ACM POPL*, 1990.)

30. W. Kim, F. Lochovsky. *Object-Oriented Concepts, Applications, and Databases.* Addison-Wesley, 1989.
31. P. Kolaitis and C. Papadimitriou. Why Not Negation By Fixpoint? In *Proceedings 7th ACM PODS*, pp. 231–239, 1988.
32. D. Leivant and J.-Y. Marion. Lambda Calculus Characterizations of Poly-Time. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, Utrecht 1993. (To appear in *Fundamenta Informaticae.*)
33. D. Luckham, D. Park, and M. Paterson. On Formalized Computer Programs. *JCSS*, **4** (1970), pp. 220–249.
34. H. Mairson. A Simple Proof of a Theorem of Statman. *TCS*, **103** (1992), pp. 387–394.
35. R. Milner. A Theory of Type Polymorphism in Programming. *JCSS*, **17** (1978), pp. 348–375.
36. J.C. Mitchell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proceedings 17th ACM POPL*, pp. 109–124, 1990.
37. J.C. Mitchell, F. Honsell, K. Fisher. A Lambda Calculus of Objects and Method Specialization. In *Proceedings 8th IEEE LICS*, pp. 26–38, 1993.
38. D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM TODS*, **6** (1981), pp. 140–173.
39. A. Skarra and S. Zdonik. Type Evolution in an Object-Oriented Database. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pp. 393–416. MIT Press, 1987.
40. R. Statman. The Typed λ-Calculus Is Not Elementary Recursive. *TCS*, **9** (1979), pp. 73–81.
41. J. Ullman. *Principles of Database Systems*, 2nd ed. Computer Science Press, 1982.
42. M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM STOC*, pp. 137–146, 1982.
43. E. Waller. Schema Updates and Consistency. In *Proceedings 2nd International Conference on Deductive and Object-oriented Databases*, Munich 1991.
44. S. Zdonik, D. Maier. *Readings in Object-Oriented Database Systems.* Morgan-Kaufmann, 1990.
45. R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*, pp. 146–182. Morgan-Kaufmann, 1992.