Space Usage in Functional Query Languages

Serge Abiteboul* Gerd Hillebrand†

Abstract

We consider evaluation strategies for database queries expressed in three functional query languages: the complex value algebra, the simply typed lambda calculus, and method schemas. Each of these query languages derives its expressive power from a different primitive: the complex value algebra from the *powerset* operator, the simply typed lambda calculus from *list iteration*, and method schemas from *recursion*. We show that "natural" evaluation strategies for these primitives may lead to very inefficient space usage, but that with some simple optimizations many queries can be evaluated with little or no space overhead. In particular, we show: (1) In the complex value algebra, all expressions with set nesting depth at most 2 can be evaluated in PSPACE, and this set of expressions is sufficient to express all queries in the polynomial hierarchy; (2) In the simply typed lambda calculus with equality and constants, all query terms of order at most 5 (where "query term" is a syntactic condition on types) can be evaluated in PSPACE, and this set of terms expresses exactly the PSPACE queries; (3) There exists a set of second-order method schemas (with no simple syntactic characterization) that can be evaluated in PSPACE, and this set of schemas is sufficient to express all PSPACE queries.

1 Introduction

A number of database languages with a functional flavor have been proposed, including, for instance, relational algebra [Cod70] and FQL [BFN82]. The algebra for complex values [AB88], which generalizes Codd's Algebra, is of the style [Bac78] based on a small set of operations that can be composed. An alternative approach based on variables and λ-abstractions is used in [HKM93]. Yet a third one with an object-oriented flavor is considered in the method schema model of [AKRW92]. These approaches seem to lead to a very inefficient use of space. In this paper, we present techniques for avoiding (or at least limiting) waste of space in the complex value algebra, λ-calculus, and method schema frameworks.

In the complex value algebra, we reconsider a nice result of [SP94] that states that any algebraic query expressing transitive closure essentially requires exponential space if evaluated naturally. We exhibit an evaluation technique that allows to compute a large class of queries (including TC) using only polynomial space. For such queries, we avoid computing powersets but instead produce a stream of elements that are treated by subsequent operators. More precisely, the tree of the algebraic expression can be viewed as a tree-network of processors with one processor for each vertex in the expression. Data flows through the edges of the tree towards the root. Each processor has a polynomial space local memory and produces the result of evaluating the corresponding subexpression as a stream that goes to its parent in the tree.

The algebra wastes space for two reasons: (1) the functional framework (and that is somewhat corrected by our technique), (2) the lack of explicit recursion/iteration, which has to be simulated using *powerset*. The other two languages that are considered provide iteration and recursion, respectively. In both cases, we can show that within PSPACE, we can avoid unnecessary waste of space. The techniques in both cases come down to representing functions extensionally as tables. These tables can be viewed in some sense as the relations used in a fixpoint evaluation.

We study the evaluation of database queries expressed in Church's *simply typed λ-calculus* [Chu41] with equality and constants, TLC$^=$ for short. The computation of PTIME queries was considered in [Hil94]. It was shown that the order 4 fragment of TLC$^=$ expresses exactly the PTIME queries. The proofs are based on

*Contact author. INRIA Rocquencourt, BP 105, F-78153 Le Chesnay cedex, France. Phone: +33 (1) 39 63 55 37. Email: Serge.Abiteboul@inria.fr.

†INRIA Rocquencourt, BP 105, F-78153 Le Chesnay cedex, France. Email: Gerd.Hillebrand@inria.fr

an order-efficient representation of relational algebra and fixpoint queries as λ-terms and a space- and time-efficient evaluation strategy for such terms. We generalize here these techniques to define a space-efficient evaluation strategy for the order 5 fragment of $\mathrm{TLC}^=$. Indeed, we provide a characterization of PSPACE in terms of the order 5 fragment of $\mathrm{TLC}^=$. As a consequence, all PSPACE problems can be computed in this functional language using polynomial space and not more.

Finally, we extend the method schemas of [AKRW92] to allow higher-order methods, which can receive other methods as arguments and which may return methods. We show that second-order method schemas can simulate *while* queries (and thus express all PSPACE queries) and that with a space-efficient evaluation strategy, the simulation can be carried out in PSPACE. However, our evaluator cannot evaluate *all* second-order method schemas in PSPACE, so at this point we do not know whether second-order method schemas express exactly the PSPACE queries.

The paper is organized as follows. Each of the next three sections addresses one particular paradigm. Section 2 deals with the algebra for complex values, Section 3 with the simply typed λ-calculus, and Section 4 with method schemas. Since the techniques used for method schemas are similar to those introduced in Section 3, our presentation in Section 4 will be mostly informal and sketchy. We conclude in Section 5 with a summary of the paper and with some directions for further research.

2 The Complex Value Algebra

In this section, we first recall the algebra and calculus for complex values [AB88]. We then discuss how some simple queries are evaluated in this algebra "naturally" with a poor usage of space. We finally present techniques to improve space usage. In particular, we present a general evaluator that computes all queries with "set-nesting 2" in polynomial space.

Let us briefly review the relevant definitions. We assume the existence of the following countably infinite and pairwise disjoint sets of atomic elements: relation names $\{R_1, R_2, \dots\}$, attributes $\{A_1, A_2, \dots\}$, constants $D = \{d_1, d_2, \dots\}$. The abstract syntax and the interpretations of *sorts* are given by:

1. $\sigma = D \mid \langle B_1 : \sigma, \dots, B_k : \sigma\rangle \mid \{\sigma\}$, where $k \geq 0$ and $B_1, \dots, B_k$ are distinct attributes,
2. $[\![D]\!] = D$
3. $[\![\{\sigma\}]\!] = \{\{v_1, \dots, v_j\} \mid v_i \in [\![\sigma]\!], i = 1, \dots, j\}$,
4. $[\![\langle B_1 : \sigma_1, \dots, B_k : \sigma_k\rangle]\!] = \{\langle B_1 : v_1, \dots, B_k : v_k\rangle \mid v_j \in [\![\sigma_j]\!], j = 1, \dots, k\}$,

where σ denotes a sort and $[\![\sigma]\!]$ its interpretation. An element of a sort is called a *complex object*. A complex object of the form $\langle \dots\rangle$ (resp., $\{\dots\}$) is said to be a *tuple* (resp., a *set*). The tuple fields are viewed as unordered. We therefore do not distinguish, for instance, between sorts $\langle A : D, B : D\rangle$ and $\langle B : D, A : D\rangle$, or between objects $\langle A : 2, B : 2\rangle$ and $\langle B : 2, A : 2\rangle$. Note also that (because of the empty set) a complex object may belong to more than one sort.

A *(database) schema* is a pair $(\mathcal{R}, \mathcal{S})$ where $\mathcal{R}$ is a set of relation names and $\mathcal{S}$ is a function from $\mathcal{R}$ to sorts. A *(database) instance* $\mathcal{I}$ of a schema $(\mathcal{R}, \mathcal{S})$ is a function from $\mathcal{R}$ such that:

$$\text{for each } R \text{ in } \mathcal{R},\ \mathcal{I}(R) \in [\![\{\mathcal{S}(R)\}]\!].$$

The *complex value algebra*, denoted by ALG^{cv}, is a (many-sorted) algebra extending relational algebra to complex values. It contains set operations (e.g., intersection, union, difference, setting, set flattening) that include a powerset operation, tuple operations (e.g., selection, projection, tupling, tuple flattening), and cross product. For a definition of these operations and others, see [AB88].

The *complex value calculus*, denoted by CALC^{cv}, is a (many sorted) calculus based on higher-order logic over finite domains. For each sort, we assume the existence of a countably infinite set of variables of that sort. Let $(\mathcal{R}, \mathcal{S})$ be a schema. A constant or a variable is a *term*. If $t_1, \dots, t_k$ are terms and $B_1, \dots, B_k$ distinct attributes, then $\langle B_1 : t_1, \dots, B_k : t_k\rangle$ is a term; furthermore, if the t_i are of the same sort, $\{t_1, \dots, t_k\}$ is a term. The sorts of terms are defined in the obvious way. (Note again that a term may have several sorts because of the empty set.) A *positive literal* is an expression of the form: (1) $R(t)$, where $R \in \mathcal{R}$ and t is of sort $\mathcal{S}(R)$, (2) $t = t'$, or (3) $t \in t'$ (with the appropriate sort restrictions)[1]. *Formulas* are defined from atomic formulas using the standard connectors and quantifiers: $\wedge, \vee, \neg, \forall, \exists$. A *query* is an expression of the form $\{x \mid \psi\}$ where x is the only free variable in ψ. The *answer* to a query q on an instance $\mathcal{I}$, denoted $q(\mathcal{I})$, is defined analogously to the relational model. We assume that the domain consists of only those constants occurring in $\mathcal{I}$, so the answer to a query is always finite.

The equivalence of the algebra and the calculus has been shown in [AB88]. A similar result for the (in some sense more general) Logical Data Model was shown earlier [KV93b]. It is also known [HS91, KV93a] that ALG^{cv} (respectively, CALC^{cv}) expresses the *elementary queries*, i.e., the queries that are computable [CH80b] in hyper-exponential time (space) w.r.t. the database size.

An essential difference between relational calculus and complex value calculus is that the latter can express transitive closure [AB88] whereas the former cannot [AU79]. Indeed, this is done very simply with the following CALC^{cv} query:

$$\{y \mid \forall x(closed(x) \wedge contains\_R(x) \rightarrow y \in x)\}$$

where

- $closed(x) \equiv \forall u, v, w\,(\langle A : u, A' : v\rangle \in x \wedge \langle A : v, A' : w\rangle \in x \rightarrow \langle A : u, A' : w\rangle \in x)$,
- $contains\_R(x) \equiv \forall z(R(z) \rightarrow z \in x)$,
- $sort(x) = \{sort(R)\}$, $sort(y) = sort(z) = sort(R)$, $sort(u) = sort(v) = sort(w) = \mathbf{dom}$.

Intuitively, the formula specifies the set of pairs y such that y belongs to each binary relation x containing R and transitively closed. It is interesting to consider the algebraic expression for transitive closure proposed in [AB88]. In a schematic form, it looks like:

$$\bigcap \sigma(powerset(D \times D)),$$

where D is a query computing the set of domain elements and σ a query expression filtering the relations that contain R and are closed transitively. The operator $\bigcap$ takes the intersection of the relations that passed the test. Observe that this construction requires by essence exponential space in the database size because of powerset. Indeed, in a recent paper [SP94], it is shown that "any expression in ALG^{cv} needs exponential space to compute transitive closure." Now, let us examine this statement more closely. There is a basic assumption underlying it: expressions are evaluated in the "natural" manner; and for this particular style of evaluation, TC requires exponential space. We are concerned here with more sophisticate evaluation techniques that allow for a better management of space.

We consider three such evaluation strategies. The first two are of little interest and are here only to highlight the limits of what we may expect to prove:

The cheater evaluator. By [Var82], we know that PSPACE corresponds to partial fixpoint logic (or the language *while*). There is a simple way of coding partial fixpoint queries into complex value expressions. The evaluator, called *cheater*, works as follows. Before starting the computation, *cheater* tests whether this is (syntactically) a translation of a partial fixpoint. If yes, the query is evaluated using PSPACE. Otherwise, the standard (perhaps exponential space) computation is performed. Therefore,

> For each query q in PSPACE, there is an expression in ALG^{cv} that computes q and is evaluated by *cheater* in PSPACE.

From this, it is clear that evaluators have to be ranked also by the aesthetics of the optimization.

The second-order logic evaluator. Given an expression in the powerset algebra like the one for transitive closure above, we can translate it into a calculus expression. Then, one can check whether this calculus expression is in second-order logic, i.e., whether quantifications are over domain elements or sets of domain elements (not sets of sets). If yes, one can evaluate the expression in PSPACE. Otherwise, we go with the standard evaluation. Using this technique, we have a slightly weaker "result" since some PSPACE queries (e.g., quantified formula) would require exponential space unless the polynomial hierarchy (PH) and PSPACE coincide.

[1] Strictly speaking, the symbols = and $\in$ are also many sorted.

We next consider a last evaluator called the *space-careful evaluator*. This evaluator provides a generic optimization technique that (sometimes) allows to avoid going to exponential space. To simplify, we first assume that an ordering of the domain elements is provided. We will remove this assumption at the end of the section.

The only operation that introduces exponential space is *powerset*. The space-careful evaluator avoids when possible to compute a powerset—when this cannot be avoided, the standard potentially EXPSPACE computation is used. Our evaluator only works for expressions of set nesting 2. (We consider "deeper" expressions at the end of the section.) The *set nesting* of an algebraic expression is n if the type-tree of each intermediary result has at most n set-constructions on any path from the root to a leaf. For instance, $\{\langle\{D\}, D\rangle\}$ is of set nesting 2, and $\{\{\{D\}\}\}$ is of set nesting 3.

The *space-careful* evaluator works as follows. Let E be an algebraic expression. This expression can be viewed as a tree. With each vertex of the tree we associate a processor. Each vertex can produce for its parent a stream of relations or tuples. The parent may at any time require the first relation/tuple in the output stream of a child and the next one in the stream. We will see that each of these processors require only a polynomial local space, so the evaluator is PSPACE.

For the example above, the powerset vertex produces a stream of sets and this stream is filtered by the next processors. A last processor receives a stream of relations and computes the intersection keeping the intermediate results in its local store. Observe that we do not require that a stream be duplicate-free. So a stream can be viewed as a representation of a list (and not a set). We deal with duplicate elimination later.

More precisely, the leaves of the tree are base relations, and since we are assuming a total ordering of the database, there is clearly a processor that produces the tuples of the relation one at a time. Each internal vertex implements the corresponding operation of the algebra. We illustrate this with some examples:

1. The projection of a relation is implemented as follows. To produce its output stream, the projection processor resets its child and then asks for the elements of its input stream, one at a time, and produces the projection (possibly with duplicates).

2. For powerset, since set nesting is limited to 2, the input stream is of set nesting 1. Thus it is of polynomial size and can be loaded into the local store. The elements of the powerset are then produced one at a time in lexicographical order on the output stream.

3. An intersection of two sets of sets is implemented as follows: iterate over the first stream, and for each X in this stream, iterate over the second stream; if X is found in that second stream, X is delivered on the output stream.

Before stating the main result of the section, we make two observations, the first is on duplicate elimination and the second on the absence of order:

1. If a result is of set nesting 1, we can store it in some local store and eliminate duplicates. If the set nesting is 2, consider a tree that computes the solution possibly with duplicates. Say its root is processor *sol_with_dup*. We add a processor *dup_elim* that receives the stream produced by *sol_with_dup* and receives also a duplicate-free enumeration of the candidate solutions. For each candidate solution X, *dup_elim* examines the entire stream produced by *sol_with_dup* to see whether X is in there.

2. The absence of order is not a problem since we are presenting an implementation. At the implementation level, we have a representation of data elements that provides an ordering that we can use in the computation. Observe that since algebra expressions are order-independent, the result will not depend on the ordering.

In the following, we assume that our devices write the result (without duplicates) on a write-only/write-once output tape and that the size of this output tape is not counted. One can show:

Theorem 2.1 *Each algebraic expression in ALG^{cv} of set nesting at most 2 can be evaluated with the space-careful evaluator in* PSPACE.

A straightforward consequence is that for each query q in second-order logic (i.e., the polynomial hierarchy), there is an expression in ALG^{cv} (indeed, the "natural translation" of the CALC^{cv} expression q) that computes q and is answered by the *space-careful* evaluator in PSPACE. In particular, queries such as TC, which requires EXPSPACE with the natural evaluation, or *even* (is the cardinality of an input set even?), which is not even expressible in *while* (the natural order-free counterpart of PSPACE), are both computed using polynomial space resources.

On the other hand, there is still a big gap between the actual space complexity of TC and the space required by its evaluation with the *space-careful* evaluator. Furthermore, we would like to be able to compute *all* PSPACE queries using polynomial space. But, the iteration required by a partial fixpoint requires an extra level of set nesting (unless PSPACE = PH.) This suggests the following open problems:

Open Problem 2.2 *(a) Find a "nice" evaluator such that for each query q in* PSPACE*, there is an expression in ALG^{cv} that computes q and is evaluated in* PSPACE*. (b) Similarly, for* PTIME.

In the statement of the problems, we use the word "nice". Indeed, observe that *cheater* satisfies (a) if we remove this word. The difference between *cheater* and *space-careful* is that the optimization used in the former is based on detecting some particular syntactic patterns, whereas the latter does not attempt to modify the query but tries to save space when performing the evaluation. Clearly, we need formal criteria for nicety.

3 The Simply Typed Lambda Calculus

As our next topic, we study database queries expressed in Church's *simply typed λ-calculus* (with equality and constants). We briefly review the way databases and queries can be represented as typed λ-terms and mention previous results obtained for these representations. Then, we generalize the techniques of [HK94] to define a space-efficient evaluation strategy for order 5 λ-terms, which leads to a characterization of PSPACE in terms of the order 5 fragment of TLC$^=$.

3.1 The Simply Typed Lambda Calculus: TLC and TLC$^=$

The simply typed λ-calculus [Chu41] (TLC for short) is a very basic functional framework built around the concepts of *abstraction* and *function application*, and it forms the backbone of most modern functional programming languages (e.g., ML [G78, Mil85], Miranda [Tur85], and Haskell [HW88]). Let us briefly review its definition (cf. [Bar84, HS86]).

(but only if the default

The syntax of TLC *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 $(\alpha \to (\alpha \to \alpha))$. TLC *terms* are given by the grammar $\mathcal{E} \equiv x \mid (\mathcal{E}\mathcal{E}) \mid \lambda x.\,\mathcal{E}$, where x ranges over a set of *expression variables*, and the condition of *well-typedness*. As usual, the type $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$ and the term $P\,Q\,R$ stands for $(P\,Q)\,R$. For lack of space, we have to assume that the reader is familiar with the notion of *well-typedness* in TLC and with α- and β-reduction. We denote by $\rhd$ the transitive closure of α- and β-reduction. A term is in *normal form* if it does not $\rhd$-reduce to another term.

We obtain the simply typed λ-calculus *with equality and constants* (TLC$^=$ for short) by enriching the simply-typed λ-calculus syntax with: (1) a countably infinite set $\{o_1, o_2, \ldots\}$ of atomic constants of type o (some fixed type variable), and (2) a function constant Eq of type $\mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$ (for some fixed type variable τ different from o). For every pair of constants $o_i, o_j : \mathsf{o}$ and every pair of terms $P, Q : \tau$, we add to $\rhd$ the reduction rule

$$(Eq\,o_i\,o_j\,P\,Q) \;\rhd \begin{cases} P & \text{if } i = j, \\ Q & \text{if } i \neq j. \end{cases}$$

Thus, the term $(Eq\,o_i\,o_j\,P\,Q)$ is the equivalent of "if $o_i = o_j$ then P else Q". Just as TLC, TLC$^=$ is Church-Rosser and strongly normalizing, and TLC$^=$ principal types can be inferred from unadorned terms. (See [Bar84, HS86, KMM91] for a discussion of these concepts.)

3.2 Representing Databases and Queries

Relations are represented in TLC$^=$ as follows. Let $O = \{o_1, o_2, \ldots\}$ be the set of constants of the TLC$^=$ calculus. For convenience, we assume that this set of constants also serves as the universe over which relations are defined.

Definition 3.1 *Let $r = \{(o_{1,1}, o_{1,2}, \ldots, o_{1,k}), (o_{2,1}, o_{2,2}, \ldots, o_{2,k}), \ldots, (o_{m,1}, o_{m,2}, \ldots, o_{m,k})\} \subseteq O^k$ be a k-ary relation over O. An* encoding *of r, denoted by $\overline{r}$, is a TLC$^=$ term*

$$
\begin{array}{l}
\lambda c.\, \lambda n. \\
\quad (c\, o_{1,1}\, o_{1,2} \ldots o_{1,k} \\
\quad\quad (c\, o_{2,1}\, o_{2,2} \ldots o_{2,k} \\
\quad\quad\quad \ldots \\
\quad\quad\quad\quad (c\, o_{m,1}\, o_{m,2} \ldots o_{m,k}\, n) \ldots)),
\end{array}
$$

in which every tuple of r appears exactly once. If r contains exactly one tuple $(o_{1,1}, o_{1,2}, \ldots, o_{1,k})$, we also allow $\lambda c.\, c\, o_{1,1}\, o_{1,2} \ldots o_{1,k}$ as an encoding of r.[2]

The encoding $\overline{r}$ can be thought of as a generalized Church numeral that not only iterates a given function a certain number of times, but also provides different data at each iteration. Note that there can be many encodings of a relation, depending on how the input is ordered.

If r contains at least two tuples, the principal type of $\overline{r}$ is $(\mathsf{o} \to \cdots \to \mathsf{o} \to \sigma \to \sigma) \to \sigma \to \sigma$, where σ is a free type variable.[3] The order of this type is 2, independent of the arity of r. We abbreviate this type as o_k^σ. Instances of this type, obtained by substituting some type expression θ for σ, are abbreviated as o_k^θ. It is fairly easy to see that any closed normal-form term of type o_k^σ is an encoding of a relation.

In [HK94], a hierarchy of functional query languages based on fixed-order fragments of TLC$^=$ is defined:

Definition 3.2 *The language of* typed list iteration of order $i+3$ with equality, *denoted as TLI$_i^=$, is the set of all closed (i.e., not containing any free variables) TLC$^=$ terms Q that can be typed as $Q{:}\,\mathsf{o}_{k_1}^{\theta_1} \to \mathsf{o}_{k_2}^{\theta_2} \to \cdots \to \mathsf{o}_{k_l}^{\theta_l} \to \mathsf{o}_k^\sigma$, where $l, k_1, \ldots, k_l$ and k are non-negative integers, $\theta_1, \ldots, \theta_l$ are types of order $\leq i$, and σ is a type variable different from o.*

Note that a TLC$^=$ term Q of type $\mathsf{o}_{k_1}^{\theta_1} \to \mathsf{o}_{k_2}^{\theta_2} \to \cdots \to \mathsf{o}_{k_l}^{\theta_l} \to \mathsf{o}_k^\sigma$ has the property that whenever $\overline{r_1}, \ldots, \overline{r_l}$ are encodings of relations of arities $k_1, \ldots, k_l$, the expression $(Q\, \overline{r_1}, \ldots, \overline{r_l})$ is closed, well typed, and of type o_k^σ. Hence its normal form must be an encoding of a relation, i.e., Q defines a mapping from encodings of relations to encodings of relations. Thus, the somewhat obscure typing constraint in the definition of TLI$_i^=$ is merely a syntactic condition to guarantee the proper semantic behavior of query terms.

As a simple example of the representation of queries in this framework, consider the following term $\mathit{Select}_{2;1=o_1}$, which selects those tuples from a binary relation R whose first component is o_1:

$$
\begin{array}{l}
\mathit{Select}_{2;1=o_1} \equiv \lambda R. \\
\qquad\qquad \lambda c.\, \lambda n. \\
\qquad\qquad\quad R\, (\lambda x_1.\, \lambda x_2.\, \lambda S.\, \mathit{Eq}\, x_1\, o_1\, (c\, x_1\, x_2\, S)\, S)\, n
\end{array}
$$

If R encodes the relation $\{(o_1, o_3), (o_2, o_3)\}$, i.e.,

$$R \equiv \lambda c'.\, \lambda n'.\, c'\, o_1\, o_3\, (c'\, o_2\, o_3\, n'),$$

we have the following reduction sequence (some steps are omitted):

$$
\begin{array}{l}
(\mathit{Select}_{2;1=o_1}\, R) \;\rhd\; \lambda c.\, \lambda n. \\
\qquad\qquad\qquad (\lambda c'.\, \lambda n'.\, c'\, o_1\, o_3\, (c'\, o_2\, o_3\, n')) \\
\qquad\qquad\qquad\quad (\lambda x_1.\, \lambda x_2.\, \lambda S.\, \mathit{Eq}\, x_1\, o_1\, (c\, x_1\, x_2\, S)\, S) \\
\qquad\qquad\qquad\quad n
\end{array}
$$

[2] This is because $\lambda c.\, c\, o_{1,1}\, o_{1,2} \ldots o_{1,k}$ is an η-reduct (see [Bar84]) of $\lambda c.\, \lambda n.\, c\, o_{1,1}\, o_{1,2} \ldots o_{1,k}\, n$.

[3] If r is empty or contains only one tuple, this type is only an instance of the principal type of $\overline{r}$.

$$\begin{aligned}
\rhd\ & \lambda c.\, \lambda n. \\
& \quad ((\lambda x_1.\, \lambda x_2.\, \lambda S.\, Eq\, x_1\, o_1\, (c\, x_1\, x_2\, S)\, S)\, o_1\, o_3 \\
& \quad\quad ((\lambda x_1.\, \lambda x_2.\, \lambda S.\, Eq\, x_1\, o_1\, (c\, x_1\, x_2\, S)\, S)\, o_2\, o_3\, n)) \\
\rhd\ & \lambda c.\, \lambda n. \\
& \quad (\lambda S.\, Eq\, o_1\, o_1\, (c\, o_1\, o_3\, S)\, S) \\
& \quad\quad (Eq\, o_2\, o_1\, (c\, o_2\, o_3\, n)\, n) \\
\rhd\ & \lambda c.\, \lambda n.\, (\lambda S.\, c\, o_1\, o_3\, S)\, n \\
\rhd\ & \lambda c.\, \lambda n.\, c\, o_1\, o_3\, n,
\end{aligned}$$

which leads indeed to an encoding of the desired result $\{(o_1, o_3)\}$. It is easy to see that the term $Select_{2;1=o_1}$ can be typed as $\mathsf{o}_2^\sigma \to \mathsf{o}_2^\sigma$, where σ is a type variable, so this term is a $\mathrm{TLI}_0^=$ query.

It was shown in [HKM93] how to express relational algebra, Datalog$^\neg$, and complex object algebra queries as typed λ-terms acting on λ-calculus representations of relations or complex values. In [HK94], it is proved that $\mathrm{TLI}_0^=$ expresses at least the first-order queries[4] and that $\mathrm{TLI}_1^=$ expresses exactly the PTIME queries. The proofs are based on an order-efficient representation of relational algebra and fixpoint queries as λ-terms and a space- and time-efficient evaluation strategy for such terms.

3.3 Simulating PSPACE Computations

In the following, we extend the results of [HK94] to obtain a characterization of PSPACE. This provides a sublanguage of $\mathrm{TLC}^=$ which expresses PSPACE and such that each query in this language can be evaluated in PSPACE. We highlight the main arguments of the proof. A detailed presentation of the technique can be found in [Hil94].

Theorem 3.3 *The database queries definable by $TLI_2^=$ terms are exactly the* PSPACE *queries.*

The proof involves two parts: An encoding of generic PSPACE computations as $\mathrm{TLI}_2^=$ queries and an evaluation algorithm that evaluates arbitrary $\mathrm{TLI}_2^=$ queries in PSPACE.

For the first part, it essentially suffices to show how to iterate a first-order query Q an exponential number of times. This is because an exponential iteration can simulate a *while* query and the latter are sufficient to express all of PSPACE [Var82]. Alternatively, one can think of Q as a first-order representation of a Turing machine transition function that is iterated exponentially often over polynomial-size machine configurations encoded as relations.

The iteration is accomplished by a term of the form

$$Crank\ Q\ Nil,$$

where $Crank$ is a Church numeral of exponential size (i.e., a term $\lambda f.\, \lambda x.\, f\,(f \ldots (f\, x))$, where there is an exponential number of f's), Q is a $\mathrm{TLC}^=$ encoding of the first-order query to be iterated, and $Nil \equiv \lambda c.\, \lambda n.\, n$ encodes the empty relation. To build $Crank$, one writes, e.g.,

$$Crank \equiv Exp\,(Length\,(R \times \cdots \times R)),$$

where R is an encoding of the input, $\times$ is a $\mathrm{TLC}^=$ term computing the Cartesian product, $Length$ is a $\mathrm{TLC}^=$ term computing the length of a relation (as a Church numeral), and Exp is a $\mathrm{TLC}^=$ term exponentiating Church numerals. It can be shown that this query can be implemented (with certain modifications) in $\mathrm{TLI}_2^=$. Thus, one direction of the theorem is established.

For the other direction, we use a special evaluation strategy that represents λ-terms not as syntactic objects, but as *tables* describing their functional behavior. For example, a term $f{:}\,\mathsf{o} \to \mathsf{o}$ is represented as the binary relation $\{(o_i, o_j) \mid (f\, o_i) \rhd o_j\}$, which describes how f acts on constants. When evaluating a particular query, it suffices to record the behavior of f on the constants appearing in the input database,

[4] Under certain assumptions about the ordering of the input, it can be shown that $\mathrm{TLI}_0^=$ expresses *exactly* the first-order queries, cf. [Hil94].

so f can be represented by a table of size quadratic in the size of the active domain. Thus, the evaluator is essentially a *semantic* evaluator—it manipulates *values* of terms (in a certain simple model) rather than terms themselves. (Observe that a purely syntactical evaluator might require up to doubly exponential space to evaluate a $\text{TLI}_2^=$ expression of the form *Crank Q Nil*: If Q does not eliminate duplicate tuples, the size of the result can increase by a polynomial factor at each step of the iteration.)

To make the discussion more precise, consider a fixed $\text{TLI}_2^=$ term Q applied to inputs $\overline{r_1}, \ldots, \overline{r_l}$. W.l.o.g., we may assume that Q is of the form $\lambda R_1 \ldots \lambda R_l. \lambda c. \lambda n. Q'$, so the evaluation of the query consists of determining the relation encoded by the normal form of $\lambda c. \lambda n. Q' [R_1 := \overline{r_l}, \ldots, R_l := \overline{r_l}]$. For the purpose of evaluating $Q' [R_1 := \overline{r_l}, \ldots, R_l := \overline{r_l}]$, we can treat the variables c and n as ***constants*** associated with the fixed types $\mathsf{o} \to \cdots \to \mathsf{o} \to \tau \to \tau$ and τ, respectively, where the number of o's is given by the arity of the output of Q, and τ is some type variable different from o. (For convenience, we pick the same variable that is used in the typing $Eq{:}\, \mathsf{o} \to \mathsf{o} \to \tau \to \tau \to \tau$ and we assume that no variables other than o and τ appear in the type of Q.)

The evaluation of $Q' [R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]$ involves the manipulation of terms whose types are built from the type variables τ and o and whose symbols are either variables or constants drawn from the set $\{o_1, \ldots, o_N, Eq, c, n\}$ (where $\{o_1, \ldots, o_N\}$ is the active domain of the input). Let $\mathcal{T}$ be the set of simple types containing only the type variables τ and o and for each type $\gamma \in \mathcal{T}$, let $\mathcal{E}_\gamma$ be the set of $\text{TLC}^=$ terms of type γ built from $o_1, \ldots, o_N, Eq, c, n$ and variables with types in $\mathcal{T}$. Let $\overline{\mathcal{E}}_\gamma$ be the set of terms in $\mathcal{E}_\gamma$ which are closed (i. e., built from $o_1, \ldots, o_N, Eq, c, n$ and bound variables) and in normal form. It is easy to see that $\overline{\mathcal{E}}_{\mathsf{o}}$ contains just the constants $\{o_1, \ldots, o_N\}$ and that $\overline{\mathcal{E}}_\tau$ consists of all lists of the form

$$
\begin{array}{l}
(c\, o_{1,1}\, o_{1,2} \ldots o_{1,k} \\
\quad (c\, o_{2,1}\, o_{2,2} \ldots o_{2,k} \\
\qquad \ldots \\
\qquad\quad (c\, o_{m,1}\, o_{m,2} \ldots o_{m,k}\, n) \ldots)),
\end{array}
$$

where $m \geq 0$ and the $o_{i,j}$ are drawn from the set $\{o_1, \ldots, o_N\}$. Thus, each term in $\overline{\mathcal{E}}_\tau$ represents a relation $\{\langle o_{i,1}, \ldots, o_{i,k} \rangle \mid 1 \leq i \leq m\} \subseteq D^k$.

For the output of a query, the exact normal form of $Q' [R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]$ does not matter—the evaluation algorithm merely has to determine the relation that it represents. Thus, it can identify terms in $\overline{\mathcal{E}}_\tau$ that represent the same relation, or, more generally, terms in $\overline{\mathcal{E}}_\gamma$ that represent the same mapping from relations to relations, the same operator on mappings, etc. More formally, we introduce an equivalence relation $\sim_\gamma$ on every set $\overline{\mathcal{E}}_\gamma, \gamma \in \mathcal{T}$, by defining

$$
\begin{array}{rcl}
t \sim_{\mathsf{o}} t' & \iff & t \equiv t' \\
t \sim_\tau t' & \iff & t \text{ and } t' \text{ represent the same relation} \\
t \sim_{\alpha \to \beta} t' & \iff & \forall s \in \overline{\mathcal{E}}_\alpha{:}\, \mathrm{nf}\,(t\, s) \sim_\beta \mathrm{nf}\,(t'\, s),
\end{array}
$$

where $\mathrm{nf}\,(t\, s)$ denotes the normal form of $t\, s$.

We can view equivalence classes in $\overline{\mathcal{E}}_{\mathsf{o}}$ as constants from D and equivalence classes in $\overline{\mathcal{E}}_\tau$ as k-ary relations over D. More interestingly, we can view equivalence classes in $\overline{\mathcal{E}}_{\alpha \to \beta}$ as mappings from equivalence classes in $\overline{\mathcal{E}}_\alpha$ to $\overline{\mathcal{E}}_\beta$:

Lemma 3.4 *Let $t \sim_{\alpha \to \beta} t'$ and $s \sim_\alpha s'$. Then* $\mathrm{nf}\,(t\, s) \sim_\beta \mathrm{nf}\,(t'\, s')$.

Since we are only interested in the equivalence class of the normal form of $Q' [R_1 := \overline{r_1}, \ldots, R_l := \overline{r_l}]$, it suffices to manipulate equivalence classes of terms during the evaluation of the query, rather than actual λ-terms. Thus, an intermediate result t of type $\alpha_1 \to \cdots \to \alpha_m \to \beta$, where β is o or τ, can be stored as a table describing the action of t on each vector of equivalence classes from $\overline{\mathcal{E}}_{\alpha_1} \times \cdots \times \overline{\mathcal{E}}_{\alpha_m}$, rather than the term t itself. The trick of representing intermediate results as graphs rather than λ-terms imposes a fixed bound on the size of these representations, but unfortunately the bound is quite large. For example, there are exponentially many equivalence classes in $\overline{\mathcal{E}}_\tau$ (one for each k-ary relation over D), and thus a table describing an order 1 term, such as the "accumulator value" passed between stages of an iteration, would have to be of exponential size.

However, there is an important optimization that can be made. On closer inspection of the structure of terms in $\overline{\mathcal{E}}_{\tau\to\alpha}$, where α is any type in $\mathcal{T}$, one finds that such terms cannot define arbitrary mappings from equivalence classes in $\overline{\mathcal{E}}_\tau$ (i. e., relations) to equivalence classes in $\overline{\mathcal{E}}_\alpha$. This is because such terms cannot distinguish between different inputs—a closed term of type τ is a "black box" whose contents are inaccessible to any other TLC$^=$ term. Thus, a term in $\overline{\mathcal{E}}_{\tau\to\alpha}$ must either ignore its input altogether, or depend linearly on it, in the sense that it passes every tuple in the input through to the output. More precisely, we have the following lemma:

Lemma 3.5 *Let t and t' be terms in $\overline{\mathcal{E}}_{\tau\to\alpha}$ and e and f be terms in $\overline{\mathcal{E}}_\tau$ corresponding to the relations $\emptyset$ and D^k, respectively. If $\mathrm{nf}\,(t\,e) \sim_\alpha \mathrm{nf}\,(t'\,e)$ and $\mathrm{nf}\,(t\,f) \sim_\alpha \mathrm{nf}\,(t'\,f)$, then $\mathrm{nf}\,(t\,r) \sim_\alpha \mathrm{nf}\,(t'\,r)$ for every $r \in \overline{\mathcal{E}}_\tau$, i. e., $t \sim_{\tau\to\alpha} t'$.*

By virtue of this lemma, the equivalence class of an order 1 term t having p inputs of type o and q inputs of type τ can now indeed be stored in polynomial space. It suffices to record the output of t for each of the $N^p\,2^q$ possible assignments of constants $o_1,\ldots,o_N$ to inputs of type o and relations $\emptyset, D^k$ to inputs of type τ. To recover the output of t for some input $(o_{i_1},\ldots,o_{i_p},r_1,\ldots,r_q)$ containing relations $r_1,\ldots,r_q$ other than $\emptyset$ or D^k, one proceeds as follows: First, the output of t on input $(o_{i_1},\ldots,o_{i_p},r_1,\emptyset,\ldots,\emptyset)$ is determined by looking up the output of t on input $(o_{i_1},\ldots,o_{i_p},\emptyset,\emptyset,\ldots,\emptyset)$ and, if the latter is different from the output of t on input $(o_{i_1},\ldots,o_{i_p},D^k,\emptyset,\ldots,\emptyset)$, adding the tuples in r_1 to it; then the same technique is used to determine the output of t for input $(o_{i_1},\ldots,o_{i_p},r_1,r_2,\emptyset,\ldots,\emptyset)$, and so forth, until finally the output of t for input $(o_{i_1},\ldots,o_{i_p},r_1,\ldots,r_q)$ is found.

Using the above table representation of order 1 terms, the evaluation of a query $\lambda R_1 \ldots \lambda R_l.\,\lambda c.\,\lambda n.\,Q'$ on input $\overline{r_1},\ldots,\overline{r_l}$ now proceeds as a recursive descent into Q'. Subexpressions of order 1 are fully evaluated and turned into a table as soon as they are encountered. Subexpressions of order 2 or more are evaluated only when applied to enough arguments to bring the overall type down to 1, in which case the result fits into polynomial space. Thus, redexes $(\lambda x.\,M)\,N$ are evaluated according to the rule: If the order of N is 1 or 0, evaluate N to obtain its equivalence class and evaluate M in an environment where x is bound to the equivalence class of N. If the order of N is 2 or more, evaluate M in an environment where x is bound to N itself. In other words, redexes up to order 2 are evaluated in applicative order and redexes of higher order are evaluated in normal order. One can show that (with certain technical modifications, see [Hil94]), this evaluation procedure can be implemented using a polynomial-size stack and a polynomial-size heap. Thus, the other direction of the theorem follows.

The proof of the theorem implies in particular:

Corollary 3.6 *For each query q in* PSPACE*, there is a query in $TLI_2^=$ that computes q and that can be evaluated in* PSPACE.

To conclude this section, we note that similar techniques can be used to show that EXPTIME and $\mathrm{TLI}_3^=$ coincide [Hil94].

4 Method Schemas

In this section, we first review some definitions and results on method schemas and we extend the model to allow second-order methods. Then we show (using a simulation of *while* queries) that all PSPACE queries can be expressed by second-order method schemas that are evaluable in polynomial space using a technique similar to the previous section. Due to space limitations, our presentation will be sketchy and in particular, we refer to [AKRW92] for a formal presentation of the original model.

Method schemas are a simple programming formalism for OODBs, based on applicative program schemas with additional features such as *classes*, *methods*, *inheritance*, *name overloading*, and *late binding*. They are intended to model the object/class/method paradigm of object-oriented programming in the same way that applicative program schemas [Cou90, Gre75] model traditional programming languages. Method names in a method schema fall into two categories: *base methods*, whose interpretation is part of the input, and *coded methods*, which are defined in a *method schema* in terms of base methods and possibly other coded methods. One may view base methods, coded methods, and method schemas as functional analogs of EDB predicates, IDB predicates, and Datalog programs. For example, the following method schema declares a base method *Price* of type *Basepart* $\rightarrow$ *Int* and defines a coded method *Cost*, whose expansion depends on whether its argument is of class *Part* or *Basepart*:

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

Informally, the input to a method schema is a finite set of objects, each one belonging to a unique class, and a set of interpretations for the base methods as functions over the set of objects. Given an input, the evaluation of a method invocation $m(e_1, \ldots, e_k)$, where $e_1, \ldots, e_k$ are variable-free expressions, proceeds recursively as follows. First, $e_1, \ldots, e_k$ are evaluated to produce objects $o_1, \ldots, o_k$. Then, if m is a base method, the return value is obtained from the interpretation of m. If m is a coded method, the term is rewritten according to the definition of m that applies for the classes of the o_i's. (The result is undefined if there is no applicable definition.) Thus, the evaluation of a coded method proceeds essentially by applicative-order β-reduction, augmented by a late binding process to do name resolution.

In summary, the main features of method schemas and the ones that distinguish them from typed λ-calculus are (1) *recursive definitions*, (2) *name overloading* resolved by *late binding*, and (3) a restriction to first-order terms, i.e., the exclusion of methods that operate on or return other methods.

It is known that over ordered databases (encoded in a functional way), method schemas express exactly the PTIME queries [HKR93]. We briefly sketch how this is obtained. We consider queries over finite ordered relational databases. The input of a query consists of a finite ordered universe O (w.l.o.g an initial segment of the positive integers) and some relations $R_1, R_2, \ldots, R_l$ over O, and the output is another relation R over O.

We encode the input in the method schema framework as follows. The elements of the universe are represented by objects $\{o_0, \ldots, o_{n-1}\}$. Objects $\{o_1, \ldots, o_{n-1}\}$ are assigned class *Number* and object o_0 is assigned class *Zero*, which is a subclass of *Number*. We use a separate class for o_0 to facilitate Boolean tests: a conditional of the form "if x=0 then P else Q" can be expressed as $\mathit{Test}_{P,Q}(x)$, where $\mathit{Test}_{P,Q}$ is defined as $\mathit{Test}_{P,Q} \ @ \ \mathit{Zero} = P$, $\mathit{Test}_{P,Q} \ @ \ \mathit{Number} = Q$. The ordering of the universe is represented by a unary base method *Pred* of type *Number* $\rightarrow$ *Number*, which computes the predecessor function, and a zero-ary base method *Max* : *Number*, which returns the largest element in the universe, i.e., o_{n-1}. Finally, for each k-ary database relation R, there is a k-ary base method r of type $(\mathit{Number}, \ldots, \mathit{Number}) \rightarrow \mathit{Number}$ coding the *characteristic function* of R, i.e., such that $r(x_1, \ldots, x_k)$ returns o_0 if the tuple represented by $(x_1, \ldots, x_k)$ does not belong to R and some other object if it does. (Thus, the test "if $(x_1, \ldots, x_k) \in R$ then P else Q" can be expressed as $\mathit{Test}_{Q,P}(r(x_1, \ldots, x_k))$.)

For example, to encode the relation $R = \{(1,1),(1,2)\}$ over the universe $O = \{0,1,2\}$, 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.

A query is represented by a coded method q of type $(\mathit{Number}, \ldots, \mathit{Number}) \rightarrow \mathit{Number}$ coding the characteristic function of the result. That is, 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. For example, the query computing the union of two relations r, s is represented by the coded method $q = \lambda\vec{x}.\, r(\vec{x}) \vee s(\vec{x})$ (where the disjunction is not actually part of the language, but can be simulated similarly to the "if-then-else" statement above).

We study here an extension of this framework: higher-order method schemas where methods are first-class citizens that can be passed to and returned by other methods. For instance, we allow definitions such as

$$\mathit{Compose} = \lambda f.\, \lambda g.\, \lambda x.\, f(g(x)),$$

which defines an operator that composes two methods.

This extension raises the question of how to handle name resolution in the presence of functional arguments. For our purposes, a very simple solution suffices, where method overloading is not allowed to discriminate on the signature of functional arguments. That is, name resolution simply ignores all functional arguments of a method and picks the applicable definition based on the classes of the remaining arguments. Furthermore, a method returned as a result of a method invocation cannot be overloaded at all because it does not have a name. This is certainly not the best possible approach, but since our main concern here are evaluation techniques, we will not dwell on the issue.

Recall that over ordered databases, (first-order) method schemas express exactly the PTIME queries. We next show that second-order method schemas (where "order" is defined as in the λ-calculus section) can simulate *while* queries and thus express all PSPACE queries.

The essential ingredient for a simulation of *while* queries is the ability to iterate a first-order query an exponential number of times. The iteration can be easily implemented using recursion; what is more difficult is to terminate the recursion. For this, we need to maintain a counter whose value is decremented for each level of recursion and which terminates the recursion once its value reaches zero. In the simulation of PTIME queries by first-order method schemas in [HKR93], tuples of objects are used for this purpose, since they allow counting up to some polynomial value. To implement an exponential counter, we use second-order methods. More precisely, we encode a counter as a first-order method with domain O^k and range $\{o_0, o_{n-1}\}$, with the convention that o_0 means *False* and o_{n-1} means *True*. Such a method may be viewed as a bit string of length n^k, where each bit contains the value of the method for one k-tuple of arguments. Observe that there are 2^{n^k} such methods and that they can easily be ordered. For instance, one can choose the method

$$\mathit{Bigmax} = \lambda x_1 \ldots x_k.\, \mathit{True}$$

as the maximal element and it is a rather tedious but straightforward exercise to code second-order methods *Decr* and *IsZero* to do decrement and test for zero.

Finally, let us describe how to perform an exponential iteration of a relational query using the counter technique we have just developed. Suppose we are given a relational algebra expression $\phi(R, R_1, R_2, \ldots, R_l)$ and we want to iterate the mapping $R \mapsto \phi(R, R_1, R_2, \ldots, R_l)$, starting from the empty relation. We first obtain, using the techniques of [HKR93], a second-order method M_ϕ computing $R \mapsto \phi(R, R_1, R_2, \ldots, R_l)$. Observe that M_ϕ takes a relation, i.e., a first-order method, as argument and returns a new first-order method. Now we use the following code:

$$\begin{aligned} \mathit{Fix}_\phi &= \mathit{Crank}_\phi(\emptyset, \mathit{Bigmax}) \\ \mathit{Crank}_\phi &= \lambda R.\, \lambda C.\, \mathbf{if}\ \mathit{IsZero}(C)\ \mathbf{then}\ R\ \mathbf{else}\ \mathit{Crank}_\phi(M_\phi(R), \mathit{Decr}(C)) \end{aligned}$$

where $\emptyset$ is a first-order method representing the empty relation of the same arity as R and the "if-then-else" statement is coded as shown earlier.

This completes the simulation of *while* queries, so we have:

Theorem 4.1 *Under the representation conventions listed above, second-order method schemas over ordered databases can express every* PSPACE *query.*

Let us now discuss how to evaluate second-order method schemas. The straightforward approach, where functional arguments and results are stored as λ-terms, may lead to very inefficient space usage, as the following example shows. Consider a second-order method *Project* that implements projection of a binary relation onto its first column. If r is a first-order method coding a binary relation, then *Project*(r) will expand to something like

$$\lambda x.\, r(x, \mathit{Max}) \vee r(x, \mathit{Pred}(\mathit{Max})) \vee \cdots \vee r(x, 0)$$

Observe that this term cannot be reduced further because the value of the parameter x is not yet known. Thus, the result of this call is a λ-term n times as large as the term for r. A blowup like this can happen at every stage of an iteration, so the end result of an exponential iteration may be a λ-term of doubly exponential size.

The remedy is to represent first-order methods not *intensionally*, i.e., as pieces of code, but *extensionally*, i.e., as tables describing their action on constants, very much like base methods. Thus, in the expression above, the interpreter should continue the evaluation by substituting objects $o_0, \ldots, o_{n-1}$ for x, evaluating the term $r(x, \mathit{Max}) \vee r(x, \mathit{Pred}(\mathit{Max})) \vee \cdots \vee r(x, 0)$ once for each value of x, and collecting the results in a table. In this fashion, any first-order method of arity k can be represented in space $O(n^k)$, independent of the complexity of its code.

Now observe that with this evaluation strategy, the code of Fix_ϕ above can be executed in PSPACE. The computation of Fix_ϕ proceeds as follows. We start with the term $Crank(\emptyset, Bigmax)$. The extensions of $\emptyset$ and $Bigmax$ are computed and substituted for the formal parameters R, C in the definition of $Crank$. We now have subterms $IsZero(C)$, $M_\phi(R)$ and $Decr(C)$. Since the extensions of R and C are known, each one of these can be fully evaluated to produce another extension (or an object in the case of $IsZero(C)$). Depending on the value returned by $IsZero(C)$, the computation either terminates and returns the current value of R, or proceeds with another invocation of $Crank$, using the previously computed extensions as arguments. Note that this is a tail-recursive call, so there is no need to maintain a stack to unwind the recursion. Thus, the only storage used during the computation is a polynomial-size workspace holding the current values of the formal parameters R and C in extension format (plus some additional polynomial workspace required by the $IsZero$, M_ϕ, and $Decr$ methods). Therefore, we have:

Theorem 4.2 *For each query q in* PSPACE, *under the representation conventions listed above, there exists a second-order method schema over ordered databases that computes q and can be evaluated in* PSPACE.

Observe that we do not guarantee here that every query expressed using second-order method schemas can be evaluated in PSPACE. Indeed, it is easy to find examples of second-order method schemas that cannot be evaluated in PSPACE by the evaluator described above. However, we conjecture that all second-order method schemas can be evaluated in EXPTIME by translating them into an inflationary fixpoint query over an exponential domain, similar to the translation of first-order method schemas to an inflationary fixpoint query over a polynomial domain in [HKR93].

5 Conclusions

We have studied space-efficient evaluation strategies for three functional query languages: the complex value algebra with set nesting 2, typed list iteration of order 2, and method schemas of order 2. For the complex value algebra, a stream-based implementation of *powerset* led to a PSPACE evaluator for all queries in the polynomial hierarchy, and for $\mathrm{TLI}_2^=$ and second-order method schemas, an extensional representation of order 1 terms led to a PSPACE evaluator for all queries in PSPACE. Furthermore, we showed that $\mathrm{TLI}_2^=$ expresses exactly the PSPACE queries.

The following problems are still open:

1. Find a "nice" evaluator such that for each query q in PSPACE, there is an expression in ALG^{cv} that computes q and is evaluated in PSPACE. Similarly, for PTIME.
2. Characterize the expressive power of $\mathrm{TLI}_k^=$ for $k > 3$.
3. Characterize the expressive power of second- and higher-order method schemas.

References

[AB88] S. Abiteboul and C. Beeri. *On the power of languages for the manipulation of complex objects.* INRIA Research Report 846, 1988.

[AKRW92] S. Abiteboul, P. Kanellakis, S. Ramaswamy, and E. Waller. Method schemas. Technical Report CS-92-33, Brown University, 1992. (An earlier version appeared in *Proc. 9th ACM PODS*, 1990).

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, pp. 110–117, 1979.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style of programming and its algebra of programs. *Communications of the ACM* 21(8):613–641, 1978.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[BFN82] P. Buneman, R. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Transactions on Database Systems* 7(2):164–186, 1982.

[CH80b] A. K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[Chu41] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cou90] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 459–492. Elsevier, 1990.

[G78] M. J. Gordon, R. Milner, L. Morris, M. Newey and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. 5th ACM Symposium on Principles of Programming Languages*, pp. 119–130, 1978.

[Gre75] S. Greibach. *Theory of program structures: schemes, semantics, verification.* Lecture Notes in Computer Science 36, Springer Verlag, 1975.

[Hil94] G. Hillebrand. Finite model theory in the simply typed lambda calculus. Ph.D. thesis, Dept. of Computer Science, Brown University, 1994. Available by anonymous ftp from ftp.cs.brown.edu as /pub/techreports/94/cs94-24.ps.Z.

[HK94] G. Hillebrand and P. Kanellakis. Functional database query languages as typed lambda calculi of fixed order. In *Proc. 13th ACM Symposium on Principles of Database Systems*, pp. 222–231, 1994.

[HKM93] G. Hillebrand, P. Kanellakis, and H. Mairson. Database query languages embedded in the typed lambda calculus. In *Proc. 8th IEEE Symposium on Logic in Computer Science*, pp. 332–343, 1993.

[HKR93] G. Hillebrand, P. Kanellakis, and S. Ramaswamy. Functional programming formalisms for OODB methods. In *Proc. of the NATO ASI Summer School on OODBs*, Turkey 1993. To appear in Lecture Notes in Computer Science, Springer Verlag.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to combinators and λ-calculus.* Cambridge University Press, 1986.

[HW88] P. Hudak and P. Wadler, editors. *Report on the functional programming language Haskell.* Technical Report YALEU/DCS/RR656, Dept. of Computer Science, Yale University, 1988.

[HS91] R. Hull and J. Su. On the expressive power of database queries with intermediate types. *Journal of Computer and System Sciences* 43(1):219–267, 1991.

[KMM91] 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.

[KV93a] G. Kuper and M. Y. Vardi. On the complexity of queries in the logical data model. *Theoretical Computer Science* 116:33–58, 1993.

[KV93b] G. M. Kuper and M. Y. Vardi. The logical data model. *ACM Transactions on Database Systems* 18(3):379–413, 1993.

[Mil85] R. Milner. The standard ML core language. *Polymorphism*, 2(2), 1985. (An earlier version appeared in *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming.*)

[SP94] D. Suciu and J. Paredaens. Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure. In *Proc. 13th ACM Symposium on Principles of Database Systems*, pp. 201–209, 1994.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pp. 1–16. Lecture Notes in Computer Science 201, Springer Verlag, 1985.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM SIGACT Symposium on the Theory of Computing*, pp. 137–146, 1982.